REFIXAGENT: ITERATIVE REFLECTION AND LOCALIZATION FOR AUTOMATED PROGRAM REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated program repair via Large Language Models (LLMs) has achieved remarkable advancements, yet prevailing agent-centric paradigms exhibit limited exploration of diverse remediation strategies. Although multi-sampling methodologies have been introduced to alleviate this constraint, they frequently converge toward homogeneous solutions owing to the stochastic nature of autoregressive language models. To mitigate these deficiencies, we introduce ReFixAgent, a collaborative multi-agent architecture that enables systematic exploration of diverse candidate localization sets and implements iterative introspection and refinement for patch synthesis, thereby augmenting the strategic search landscape of program repair. Our framework initiates with multi-faceted localization set sampling, optionally augmented with code dependency graph-enhanced fault localization for complex remediation tasks. This is succeeded by iterative introspective refinement to generate high-fidelity patches, culminating in a final-generation phase guided by distilled knowledge from all historical remediation attempts. Empirical evaluations on the SWE-bench Verified benchmark demonstrate that ReFixAgent attains exceptional performance, achieving 74.4% pass@1 with DeepSeek-V3.2—surpassing state-of-the-art methodologies under equivalent model configurations by up to 5.0%. The framework also exhibits robust generalization across diverse backbone models including Claude-4, GPT-5, and Qwen-Coder, while delivering superior fault localization precision compared to existing approaches.

1 INTRODUCTION

Automated issue resolution has emerged as a foundational component in contemporary software engineering workflows, serving a pivotal role in upholding the reliability and robustness of software systems (Zhang et al., 2024). The advent of Large Language Models (LLMs), endowed with sophisticated code comprehension and deductive reasoning capabilities, has catalyzed transformative progress in this domain (Liu et al., 2024a; Achiam et al., 2023; Yang et al., 2024a). SWE-bench (Jimenez et al., 2023) constitutes a rigorous evaluation framework for assessing LLMs' proficiency in autonomously resolving authentic GitHub issues. Substantial research endeavors have focused on developing agent-driven methodologies to address these challenges (Devin, 2024; Yang et al., 2024b; Gao et al., 2025; Wang et al., 2024b). Such agent-centric systems typically equip LLMs with curated tooling interfaces, empowering them to autonomously execute operational actions, perceive environmental feedback, and strategize subsequent decision-making processes (Liu et al., 2024b).

Fault localization represents an indispensable module in issue resolution, defined as the systematic identification of code segments responsible for anomalous behaviors (Guo et al., 2016). A single defect may involve multiple causally relevant code segments, giving rise to several candidate remediation strategies, each involving edits at one or more locations.

Notwithstanding these advancements, prevailing agent-driven methodologies suffer from inadequate exploration of remediation strategies. Even test-time scaling techniques that sample multiple execution trajectories through prompt variation (Ehrlich et al., 2025) fail to fundamentally resolve this limitation. Autonomous agents tend to converge toward structurally isomorphic trajectories, ultimately generating remediation solutions with substantial content homogeneity (Vijayakumar et al., 2016; Holtzman et al., 2020; Zhang et al., 2025). This constraint stems from the inherent bias of probabilistic language models toward high-probability solution patterns, thereby constraining the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

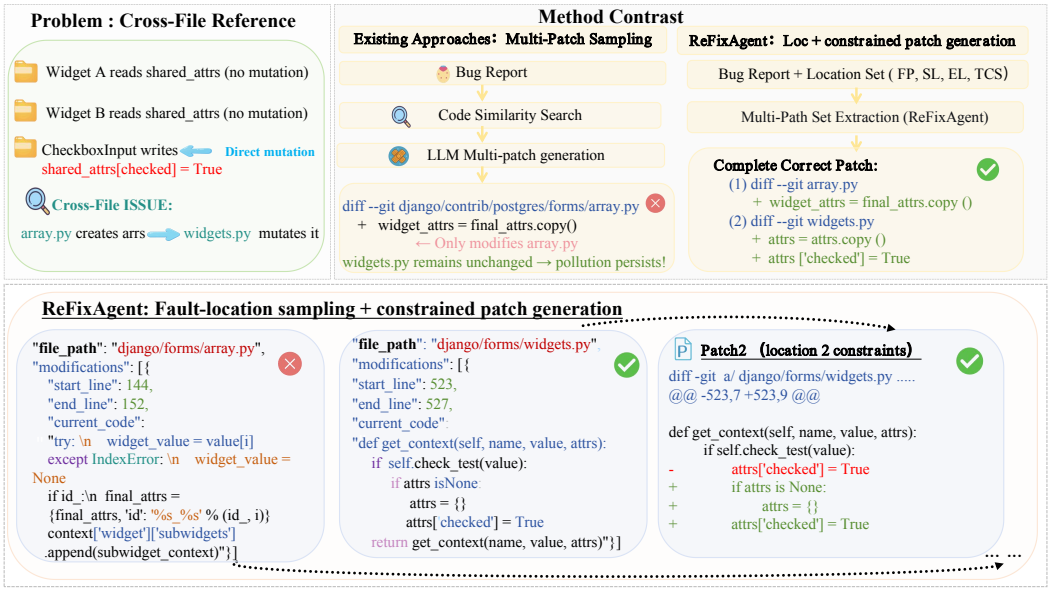

Figure 1: Illustrative example for real-world issue remediation. The agent persistently proposes repairs at erroneous loci across multiple sampling attempts, yet generates a valid patch when constrained to the authentic code region.

diversity of the exploratory search space (Kalavasis et al., 2025; Guo et al., 2024; Murthy et al., 2025). For instance, an agent may persistently confine remediation attempts to a high-confidence yet erroneous localization set across multiple sampling iterations. Although SE-Agent (Lin et al., 2025) expands the search space through mutation and recombination of intermediate trajectory steps, it cannot yield valid solutions when all sampled trajectories concentrate on a mislocalized region.

To address these critical limitations, we propose ReFixAgent, a collaborative multi-agent architecture that enables systematic exploration of diverse candidate localization sets and implements iterative introspection and refinement for patches generated for each localization set, effectively expanding the strategic search landscape of program repair. As illustrated in Figure 1, a localization sampling agent iteratively generates candidate localization sets. After deduplication, two distinct sets are obtained, with localization set-2 corresponding to the true repair locus. Constraining the code generation agent to each set produces two candidate patches, of which patch 2 is the valid fix. For complex localization tasks, a graph-enhanced localization agent supplies additional localization intelligence. For each sampled set, a multi-round introspective refinement mechanism is applied until convergence, after which an analysis agent distills actionable guidance from prior attempts to direct final patch synthesis. Our primary contributions are summarized as follows:

- We propose ReFixAgent, a collaborative multi-agent architecture that addresses the limited exploration of remediation strategies through systematic multi-localization set sampling and iterative introspective refinement mechanisms.
- Empirical evaluations demonstrate that our framework outperforms state-of-the-art methodologies under equivalent model configurations, achieving 74.4% pass@1 on SWE-bench Verified with DeepSeek-V3.2—up to 5.0% higher than existing approaches.
- Comprehensive experiments validate that our multi-localization set sampling and iterative refinement mechanisms generalize effectively across diverse agent-driven systems including Mini-SWE-agent and Live-SWE-agent, confirming the broad applicability of our approach.

2 Related Work

2.1 Fault Localization

Dynamic Fault Localization (FL) methodologies encompass Spectrum-Based FL (SBFL) (Abreu et al., 2009; Jones and Harrold, 2005) and Mutation-Based FL (MBFL) (Lou et al., 2020; Moon et al., 2014; Papadakis and Traon, 2015) approaches, alongside static analysis (Saha et al., 2013; Singhal, 2001; Wang et al., 2015) and learning-enhanced methodologies (Li et al., 2019; Meng et al., 2022; Sohn and Yoo, 2017). However, these approaches struggle with intricate inter-file dependency relationships and necessitate comprehensive test suites to elicit defect manifestations (Müller et al., 2024; Wang et al., 2024d).

Recently, LLM-driven fault localization methodologies have emerged (Kang et al., 2024; Wu et al., 2023; Xia et al., 2024). For example, MoatlessTools (Örwall, 2024) integrates agent exploration loops with semantic retrieval for code localization. OpenHands (Wang et al., 2024b) utilizes command-line utilities including grep and file inspection tools. SWE-Agent (Yang et al., 2024b) implements an Agent-Computer Interface for repository navigation. Concurrently, multiple studies have adopted graph-based representational approaches to enhance code comprehension through capturing inter-file dependency relationships. For instance, RepoUnderstander (Ma et al., 2024b) constructs hierarchical graphs and function call graphs, deploying Monte Carlo Tree Search (MCTS) for strategic exploration. OrcaLoca (Yu et al., 2025) adopts a streamlined graph representation optimized through priority scheduling and context pruning, while LocAgent (Chen et al., 2025) leverages graph-based representations to enable multi-hop deductive reasoning across code dependencies. Our localization approach harnesses the repository navigation capabilities of agent-driven methodologies while integrating dependency intelligence from code files, thereby expanding the exploratory search space for fault localization.

2.2 Automated Issue Resolution

Contemporary research demonstrates that integration of these models with extrinsic tooling and environmental interaction enables LLMs to evolve into autonomous agents capable of addressing complex software engineering challenges (Qu et al., 2025; Wang et al., 2024c; Cai et al., 2023). CodeR (Chen et al., 2024) implements a collaborative multi-agent architecture and predefined task graphs for program repair. SWE-search (Antoniades et al., 2024) introduces a multi-agent framework incorporating MCTS. LingmaAgent (Ma et al., 2024a) condenses repository knowledge into a knowledge graph and deploys MCTS strategies. OpenHands (Wang et al., 2024b) supports code manipulation and execution through sandboxed environments and implements specialized agents including CodeActAgent for remediation validation. SWE-agent (Yang et al., 2024b) introduces the Agent-Computer Interface paradigm. Mini-SWE-agent (Yang et al., 2024c) constitutes a streamlined autonomous coding agent that maintains competitive performance. Live-SWE-agent (Xia et al., 2025) autonomously evolves its agentic scaffolding dynamically during runtime. Concurrently, pipeline-driven methodologies have emerged, including Agent-less (Xia et al., 2024), which decomposes tasks into sequential localization, repair, and validation phases, and AutoCodeRover (Zhang et al., 2024), which integrates LLMs with advanced code retrieval capabilities to generate program modifications.

2.3 Agent Enhancement

Contemporary research has developed diverse methodologies to enhance LLM-driven agent performance. Agent enhancement paradigms including AutoGPT (Yang et al., 2023) and AgentGPT (Reworkd, 2023) integrate tool utilization to extend agentic capabilities, while MemGPT (Packer et al., 2023) enhances contextual comprehension through persistent memory mechanisms. CodeMonkeys (Ehrlich et al., 2025) constitutes a problem-solving system specifically engineered for computational scaling. Nevertheless, inherent characteristics of probabilistic language models—manifesting systematic bias toward high-probability solution patterns—substantially constrain exploratory search space diversity. For instance, the agent may persistently confine remediation attempts to a singular high-confidence localization set across multiple sampling iterations, which may not correspond to the authentic defect locus. SE-Agent (Lin et al., 2025) addresses these limitations through mutation and recombination of trajectory action steps. However, if all sampled trajectories concentrate on remedia-

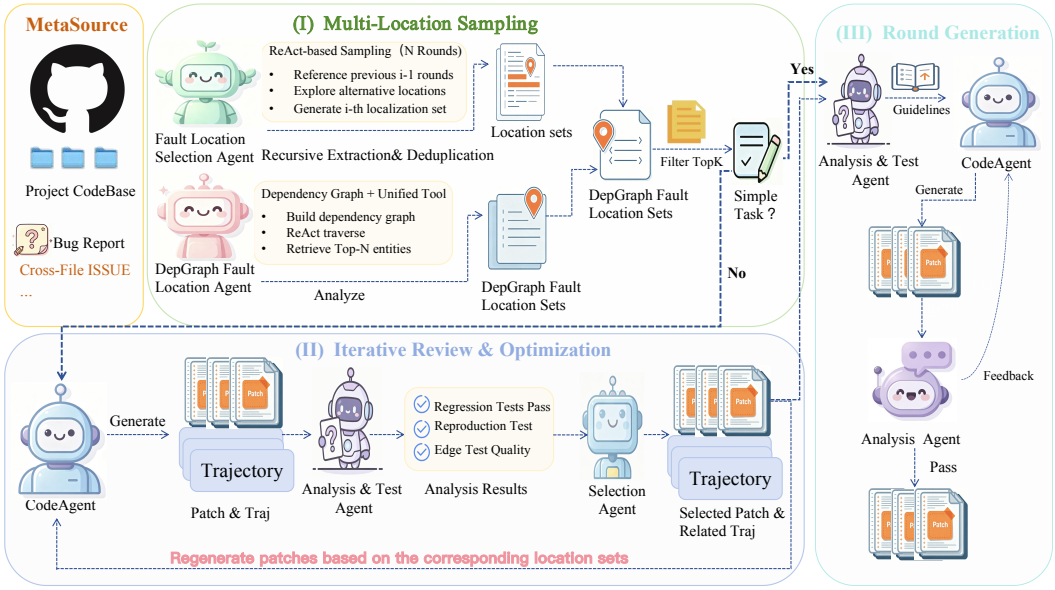

Figure 2: Architectural overview of the ReFixAgent framework, encompassing three core stages: (1) multi-localization set sampling, (2) iterative introspection and optimization, (3) final-round generation.

tion at a high-confidence yet mislocalized region, this approach cannot generate valid remediation solutions.

3 METHODOLOGY

To address these limitations, we propose ReFixAgent, a collaborative multi-agent architecture that explores diverse candidate localization sets and implements iterative introspection on remediation attempts for each localization set. As illustrated in Figure 2, our framework encompasses three core phases: (1) multi-localization sampling, (2) iterative introspection and optimization, and (3) final-round generation.

3.1 MULTI-LOCATION SAMPLING

Initially, we deploy a localization sampling agent adhering to the ReAct paradigm (Yao et al., 2023) to perform N iterations of code localization, yielding N candidate localization sets. These N localization attempts are executed iteratively. During the i-th iteration ($1 \leq i \leq N$), we provide the agent with results from iterations 1 to $i-1$, instructing the agent to leverage preceding localization outcomes as contextual references to explore alternative candidate localization sets, thereby expanding the exploratory search space through this iterative methodology. Each localization set contains one or more edit loci, with each locus specified by: (1) the file path, (2) start and end line indices, and (3) the target code segment for modification.

We implement deduplication on the N localization sets obtained from multiple sampling iterations. Following established conventions in fault localization research (Xia et al., 2024), two localization sets are classified as duplicates if and only if all edit loci within them establish bijective correspondence, where each corresponding edit locus pair shares identical file path and fully qualified name, with overlapping line ranges. In preliminary experiments on a dataset subset, we observed that certain issues exhibit relatively straightforward localization patterns, where target edit loci are explicitly referenced in issue descriptions. For these tasks, executing N iterations of code localization frequently results in only one localization set remaining post-deduplication. For tasks retaining more than k localization sets after deduplication, we classify their fault localization as complex, reflecting heightened uncertainty regarding the appropriate localization set for targeted remediation.

For these complex tasks, we integrate supplementary localization intelligence from a graph-enhanced fault localization agent (Chen et al., 2025). This agent constructs a directed heterogeneous graph representation of the codebase capturing both code structures (directories, files, classes, functions) and structural dependencies including import relationships, function invocations, and class inheritance hierarchies. The agent deploys unified code exploration tools: (1) SearchEntity for keyword-driven entity retrieval, (2) TraverseGraph for multi-hop dependency traversal across the code graph, and (3) RetrieveEntity for entity information acquisition. Through joint analysis of the issue description and codebase dependency graph structure, this agent identifies relevant code entities (e.g., django/forms/widgets.py::Media.merge). We extract the top-l ranked entities from this agent, treating each entity alongside associated metadata (file path, class name, function name, line indices) as an individual localization set. These localization sets are subsequently merged with those obtained from our fault localization agent. Following this augmentation, we perform deduplication using identical criteria to obtain final candidate localization sets for subsequent patch synthesis.

3.2 Iterative Review and Optimization

In the previous stage, tasks with relatively simple fault localization produce a final location set containing only a single location, while tasks with difficult fault localization yield multiple candidate location sets. Then we constrain a coder agent to generate one patch for each location set within the final location sets. Specifically, for each location-constrained patch generation, the coder agent first creates a reproduction script to verify the issue and executes it using the bash tool to confirm the error. If the reproduction script encounters missing dependencies or execution errors, the agent iteratively refines the script based on error messages until it successfully reproduces the issue. Next, the agent edits the source code at the locations specified in the location set to resolve the bug. It then reruns the reproduction script to verify the fix, creates edge cases appropriately, and finally generates a patch based on the edit locations specified in that location set.

For tasks with difficult fault localization, suppose the final location sets contain w location sets. Through the above process, we obtain w execution trajectories and their corresponding patches. Then we employ a selector agent to select the top w/2 patches from these candidates. The selector agent evaluates candidate patches using a multi-dimensional quality assessment framework comprising two key indicators: First, test quality assessment is performed by an analysis agent, which evaluates the comprehensiveness and correctness of both reproduction tests and edge case tests generated throughout the execution trajectory. This assessment ensures that the debugging process has adequately explored the problem space and validated the proposed solution. Second, regression test pass rate is measured to quantify the patch's impact on existing codebase functionality, where higher pass rates indicate better preservation of the project's original behavior. The regression testing procedure follows a three-stage pipeline: (i) extraction and execution of all existing test cases from the original codebase, with passing tests forming the initial regression test set, (ii) semantic filtering by a test agent to remove tests unrelated to the target issue, yielding the final regression test set, and (iii) patch validation, wherein each candidate patch is applied to the original codebase and executed against the final regression test set, with the test agent computing the resulting pass rate.

Then a selector agent ranks all w patches according to these criteria and selects the top w/2 patches and their corresponding location sets. Subsequently, we instruct the agent to regenerate patches based on these top w/2 location sets. During each regeneration round, we employ an analysis agent to distill the execution trajectories and previously generated patches for each location set, producing actionable guidance. The agent subsequently reflects on this accumulated guidance from past attempts to generate improved patches. We iteratively repeat this evaluation-and-regeneration process, where the selector agent continues to select the top-performing half of patches after each round, until only one location set remains, which we refer to as the final location set.

3.3 Final-Round Generation

In this final round, for tasks where fault localization is particularly difficult, the analysis agent distills insights from all historical execution trajectories and patches associated with this location set, producing guidance that directs the final patch generation.

For tasks with simple localization, we constrain the coder agent to generate patches at the edit positions specified in the final location set, sampling t patches in parallel. We then employ the

analysis agent to distill these t patches and their corresponding execution trajectories, deriving guidance information to direct the final round of patch generation.

To avoid issues such as infinite loops when the agent performs fixes at restricted locations, we have imposed a budget limit for executing each task. When a task reaches its budget limit, the execution is terminated.

4 EXPERIMENTS

In our experiments, we investigate the following research questions: (1) What is the overall issue resolution rate of ReFixAgent (2) How does ReFixAgent's fault localization performance compare to competing baselines (3) How effectively does our mechanism perform when integrated with other agent-driven methodologies (4) What is the individual contribution of each framework component

Table 1: Performance Summary, Workload Metrics, and Derived Deep-Learning Indicators per Backbone LLM

LLM	Agents	Top-1 (Score)	Top-2 (Score)	Mean (all)	Max (all)	Min (all)	Peak Gain	Stability	Top-1 Margin
DeepSeek-V3.2	13	ReFixAgent (74.4%)	SWE-agent (69.4%)	67.8%	74.4%	61.0%	6.6%	89.9%	5.0%
Qwen-Coder-Plus	5	ReFixAgent (69.6%)	SWE-agent (65.4%)	67.8%	69.6%	65.4%	1.8%	96.4%	4.2%
Claude-4-Sonnet	2	ReFixAgent (73.8%)	MoatlessTools (70.8%)	72.3%	73.8%	70.8%	1.5%	97.9%	3.0%
GPT-5.2	2	ReFixAgent (73.0%)	Mini-SWE-agent (69.0%)	71.0%	73.0%	69.0%	2.0%	97.1%	4.0%

Note: Only LLMs with ReFixAgent results are shown. Top-1 and Top-2 are based on non-ablation agents only; Top-3 and full rankings are provided in the supplementary table.
Derived Deep-Learning Indicators: (1) Peak Gain (Max − Mean) measures upper-bound performance lift (higher values indicate stronger best-case potential but may suggest overfitting); (2) Stability (Min / Mean) reflects baseline robustness across all agents (higher values indicate more consistent generalization); (3) Top-1 Margin (Top1 − Top2) quantifies the competitive gap (larger margins imply a more decisive champion).

4.1 EXPERIMENTAL SETTINGS

Datasets. We conduct evaluations with ReFixAgent on SWE-bench-Verified (OpenAI, 2024), encompassing 500 manually validated issues derived from SWE-bench (Jimenez et al., 2023).

Metrics. We employ the following metrics to evaluate framework performance:

Pass@1: Following precedent research (Jimenez et al., 2023; Yang et al., 2024b; Antoniades et al., 2024), we measure the percentage of issues successfully resolved in the initial attempt.

Acc@1: Following (Xia et al., 2024; Chen et al., 2025; Antoniades et al., 2024), we evaluate localization precision at three granularities:

Acc@1 (Project), **Acc@1 (Module)**, and **Acc@1 (Function)**. At each level, predictions are classified as correct if the final remediation attempt encompasses all files, classes/modules, or functions/methods requiring modification, respectively.

4.2 BASELINES

We compare ReFixAgent with two categories of competitive baselines:

Pipeline-driven methodologies: We compare with Agentless (Xia et al., 2024), a non-agentic pipeline decomposing the repair process into sequential localization, repair, and patch validation phases. **Agent-driven methodologies:** We include multiple competitive agent-driven frameworks specifically engineered for automated program repair, including Moatless Tools (Örwall, 2024), SWE-Search (Antoniades et al., 2024), SWE-Agent (Yang et al., 2024b), Mini-SWE-agent (Yang et al., 2024c), Live-SWE-agent (Xia et al., 2025), and Trae-agent (Gao et al., 2025).

4.3 RESOLUTION RATE & FAULT LOCALIZATION PERFORMANCE

Table1 presents a comparison of issue resolution rates (Pass@1) between ReFixAgent and state-of-the-art methodologies on SWE-bench Verified. We observe that ReFixAgent achieves a pass@1 score of 74.4% with DeepSeek-V3.2, surpassing SWE-agent by 5.0% and outperforming all competing baselines under equivalent

Table 2: Acc@1 Performance Comparison on Project/Module/Function

Method	Project	Module	Function
DeepSeek-V3.2			
Openhands	79.64	69.58	60.82
SWE-Search	81.62	73.98	63.06
Trae-agent	82.84	75.38	64.82
Mini-SWE-agent	79.84	71.42	61.84
Live-SWE-agent	81.24	73.42	62.84
SWE-agent	83.04	74.22	63.42
ReFixAgent	85.64 ↑	76.82 ↑	66.64 ↑
Qwen3-Coder-Plus			
SWE-agent	79.62	71.98	61.24
ReFixAgent	82.24 ↑	74.22 ↑	64.42 ↑

model configurations. Furthermore, ReFixAgent with DeepSeek-V3.2 outperforms Moatless Tools with Claude-4-Sonnet by 3.6% and Mini-SWE-agent with Claude-4.5-Sonnet by 3.8%, demonstrating superior performance even compared to methodologies employing more advanced backbone models.

Table 2 presents a comparison of fault localization precision. Our ReFixAgent demonstrates superior performance across all three granularity levels compared to existing approaches. Specifically, utilizing DeepSeek-V3.2, ReFixAgent achieves consistent improvements over SWE-agent with relative gains of 2.62%, 2.24%, and 3.18% across the three granularities respectively. Similarly, evaluated with Qwen-Coder-Plus, our methodology demonstrates improvements of 2.60%, 2.60%, and 3.22% over SWE-agent. Furthermore, core modules of ReFixAgent can be seamlessly integrated into mainstream agent-driven repair frameworks including Mini-SWE-agent and Live-SWE-agent, delivering consistent performance gains and validating the strong generalizability of our proposed mechanisms.

4.4 Generalizability

To validate generalizability, we integrate our multi-localization sampling and iterative introspection mechanisms into Mini-SWE-agent and Live-SWE-agent (Table 3). With DeepSeek-V3.2 as the backbone, our approach consistently enhances performance across architectures. For Mini-SWE-agent, precision improves by 2.44%, 2.26%, and 1.68% at file, module, and function levels, with a 4.2% increase in pass@1. Similar gains are observed for Live-SWE-agent. These consistent improvements across frameworks confirm the effectiveness and transferability of our mechanisms as a generic enhancement strategy for agent-driven methodologies.

Table 3 provides a more comprehensive breakdown, grouping results by the same LLM backbone. When all agents are evaluated on DeepSeek-V3.2, ReFixAgent again leads with **74.4%**, outperforming SWE-agent (69.4%), Trae-agent (67.8%) and Agentless (67.0%). Across all tested LLMs—including Claude, GPT, and Qwen series—ReFixAgent establishes a clear and consistent performance advantage, validating the effectiveness of our framework across diverse model families.

4.5 Ablation Study

We evaluate each ReFixAgent component via ablation: "w/o External Location Info" removes LocAgent's extrinsic localization; "w/o Multi-Location Sampling" applies introspection to only a single sampled set instead of iterative multi-set sampling; "w/o Iterative Refl." disables the iterative introspection and refinement per set. Table 3 shows that all components contribute to performance, with the latter two causing more severe drops, confirming their central roles.

The sharp decline from disabling multi-location sampling reveals that a single candidate set is fragile—initial localization errors can misdirect repair without alternative pathways for recovery. The iterative introspection mechanism is equally essential, converting the framework from a feed-forward pipeline into a closed-loop system that diagnoses patch failures and adapts based on prior errors—particularly critical for complex logical defects requiring multiple trial-and-error cycles. Moreover, removing introspection not only reduces correctness but also harms compilability, as syntactic errors from earlier attempts go uncorrected. Figure 3 presents ablation results for Mini-SWE-agent and Live-SWE-agent (both w/ReflLoc), which align with Table 4. Consistent trends across variants confirm that our design choices generalize robustly across models and projects, rather than overfitting to specific configurations.ReFixAagent consistently improves Pass@1 by 4.2% (from 61.0% to 65.2%) on Mini-SWE-agent and 3.6% (from 63.8% to 67.4%) on Live-SWE-agent, confirming its robust generalizability across two distinct base agents. Granularity-wise, while Project- and Module-level scores are relatively high,

Table 3: Different LLM Performance Comparison

Agent	LLM	Score
Agentless	DeepSeek-V3.2	67.0% **(baseline)**
Mini-SWE-agent	DeepSeek-V3.2	61.0%
OpenHands	DeepSeek-V3.2	62.8%
Live-SWE-Agent	DeepSeek-V3.2	63.8%
SWE-Search	DeepSeek-V3.2	65.4%
Trae-agent	DeepSeek-V3.2	67.8%
SWE-agent	DeepSeek-V3.2	69.4%
ReFixAgent	DeepSeek-V3.2	74.4% (+7.4%)
MoatlessTools	Claude-4-Sonnet	70.8%
ReFixAgent	Claude-4-Sonnet	73.8% (+3.0%)
Mini-SWE-agent	Claude-4.5-Sonnet	70.6%
Mini-SWE-agent	GPT-5.2	69.0%
ReFixAgent	GPT-5.2	73.0% (+4.0%)
SWE-agent	Qwen-Coder-Plus	65.4%
ReFixAgent	Qwen-Coder-Plus	69.6% (+4.2%)
OpenHands	Qwen3-Coder-30B	51.6%

the Function-level consistently yields the lowest absolute performance (from 61.84 to 62.84 baseline), and the gain from ReFix at this level (from +1.68 to +2.04) is slightly lower than at coarser levels, highlighting that fine-grained repair remains the most challenging sub-task, albeit still effectively addressed by our framework.

The ablation study (Table 5) reveals a clear hierarchy of component contributions. Iterative reflection is the most critical: removing it causes a steep drop of 2.4% on Qwen3-Coder-Plus and 3.0% on DeepSeek-V3.2, suggesting that the self-corrective loop is the primary performance driver. Multi-location sampling ranks second, with a loss of approximately 2.0% on both models, underscoring the value of exploring diverse candidate repair sites. In contrast, external location information contributes only marginally (0.2%), indicating that ReFix relies heavily on its internal contextual reasoning rather than external priors.

Notably, the iterative reflection module exhibits a larger marginal benefit on the stronger DeepSeek backbone (3.0% vs. 2.4% for Qwen), suggesting a compounding synergy between reflection depth and base model capacity. This implies that stronger foundational models can better exploit the iterative feedback loop, and we recommend prioritizing both reflection and multi-location sampling in resource-constrained deployments.

Table 4: Ablation Study on Mini-SWE-agent and Live-SWE-agent

Method	Project	Module	Function	Pass@1
Mini-SWE-Agent	79.84	71.42	61.84	61.0%
Mini-SWE-Agent(w/Re)	82.28	73.68	63.52	65.2%
Live-SWE-Agent	81.24	73.42	62.84	63.8%
Live-SWE-Agent(w/Re)	83.76	75.64	64.88	67.4%

Table 5: Ablation Study Outcomes

Method	Qwen3-Coder-Plus		DeepSeek-V3.2	
	Pass@1	Δ	Pass@1	Δ
ReFixAgent	69.6%	–	74.4%	–
ReFix Ext. Loc. Info.	69.4%	-0.2%	74.2%	-0.2%
ReFix Multi-Loc.Samp.	67.6%	-2.0%	72.2%	-2.2%
ReFix Iter. Refl.	67.2%	-2.4%	71.4%	-3.0%



Figure 3: Ablation study results for dual-agent evaluation.

LIMITATIONS

This selective sampling introduces two primary limitations to generalizability. First, observed repair performance and internal behaviors (e.g., selector agent ranking reliability) may not fully transfer to untested model families—particularly those with different instruction-tuning protocols or non-transformer architectures, where localization and synthesis quality could deviate substantially. Second, the limited number of proprietary closed-source models challenges statistical robustness; minimal sample sizes within this category reduce capacity for rigorous comparative analyses isolating the effects of pre-training data or proprietary post-training alignment on repair efficacy.

CONCLUSION

We present ReFixAgent, an innovative collaborative multi-agent framework addressing the critical limitation of inadequate exploration in existing agent-driven methodologies for automated issue resolution. Through systematic exploration of diverse candidate localization sets and deployment of iterative introspection mechanisms, our framework effectively expands the strategic search landscape of remediation strategies and achieves superior performance on challenging software engineering tasks. Empirical results on SWE-bench Verified demonstrate that ReFixAgent attains state-of-the-art performance, reaching 74.4% pass@1 with DeepSeek-V3.2 and outperforming competitive baselines by up to 5.0%.

AI USE STATEMENT

(This section is **required** and does not count toward the page limit.)

In this work, we used generative AI tools for **assisting in code implementation and debugging** (specifically, we used LLM-based agents to generate candidate patches and perform introspection as part of the proposed ReFixAgent framework). We have not used generative AI tools for **data collection, experimental design, or result interpretation**. Additionally, we used generative AI tools for **literature search and reference formatting** (recommended disclosure). We have reviewed all AI-assisted work.

See the ICLR 2027 AI Policy for Authors for more details. This statement is less than 1 page.

REFERENCES

2024. Devin, ai software engineer. https://www.cognition.ai/introducingdevin.

Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792.

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference-practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE.

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

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*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, JianGuang Lou, Anton Chekhov, Jun Sun, Hao Yu, Guodong Long, Dong Arteev, and others. 2024. Code res: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jiafeng Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 869–872.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Risse, and Azalia Mirhoseini. 2025. Codemonkeys: Solving last-minute compute for software engineering. *arXiv preprint arXiv:2411.14723*.

Pengfei Gao, Hao Liu, Tian Xiangxin Meng, Xinchen Wang, Ruida Hu, Yunan Liu, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, and others. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*.

Jiafeng Guo, Yixing Fan, Qingyao Ai, and W Bruce Croft. 2016. A deep relevance matching model for ad-hoc retrieval. In *Proceedings of the 25th ACM international conference on information and knowledge management*, pages 55–64.

Yanzhu Guo, Guokan Shang, Michalis Vazirgiannis, and Chloé Clavel. 2024. The curious decline of linguistic diversity in large language models under synthetic text. In *The Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. *International Conference on Learning Representations (ICLR)*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.05291*.

James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282.

Akis Kalavasis, Anyan Mehtrotra, and Grigoris Veselkas. 2025. On the limits of language model trade-offs between hallucination and code-collapse. In *Theory of Computing of the 57th Annual Symposium on Theory of Computing*, pages 1732–1743.

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

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

Jiayin Lin, Yifu Gu, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingchuan Sun, Hongzhi Liu, Zhongyun Hua, Ronghan He, and others. 2025. Self-evolutionary trajectory optimization multi-step reasoning with llm-based agents. *arXiv preprint arXiv:2409.02977*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bo Wang, Chengda Lu, Chenggang Zhao, Chengqi Zhang, Chengzhong Xu, Zhong Ruan, and others. 2024a. Sweagent: Llm agent for software engineering. *arXiv preprint arXiv:2406.19437*.

Junjie Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenheng Tang, Lingming Zhang, and Yiling Lou. 2024b. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2406.12221*.

Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87.

Yingwei Ma, Qingqing Yang, Rongcun Cao, Bishun Li, Fei Huang, and Yongbin Li. 2024a. AlphaImpala-agent: Human-oriented automated issue resolution via contrastive exploration. *arXiv preprint arXiv:2406.01422*.

Yingwei Ma, Qingqing Yang, Rongcun Cao, Bishun Li, Fei Huang, and Yongbin Li. 2024b. How to understand whole software repository. *arXiv preprint arXiv:2406.14222*.

Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic and feedback knowledge. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1169–1180.

Seokhyun Moon, Yuno Kim, Monzoon Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE 38th International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE.

Niels Müller, Mark Müller, Jingxuan He, and Martin Vechev. 2024. Swebench: Testing and validating real-world bug fixes with multi-agent and multi-turn repair. *arXiv preprint arXiv:2408.81887*.

Sonia Murthy, Tomer Ullman, and Jennifer Hu. 2025. One fish, two fish, but not the whole sea: Alignment reduces language models' conceptual diversity. In *The Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*.

OpenAI. 2024. Introducing SWEbench-verifed. https://openai.com/index/introducingswebenchverified/.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. 2023. Mengpt: Towards llms as open systems.

Mike Papadakis and Yves Le Traon. 2015. Metallaxis-ii: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628.

Changqi Qu, Sunhao Dai, Xiaowei Jin, Hengyi Cai, Shuaiqiang Wang, Jun Xu, Dawei Yin, and JiRong Wen. 2025. Tool learning with large language models. A survey. *Frontiers of Computer Science*, 19(8):198343.

Reworkd. 2023. AgentGPT: Assemble, configure, and deploy autonomous ai agents in your browser. https://github.com/reworkd/AgentGPT. Accessed 2025-01-03.

Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE.

Amit Singhal and others. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43.

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.

Ashwin K Vijayakumar, Michael Copeland, Ramkrishna S, and Dhruva Baruah. 2016. Diverse beam search: Encoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.01624*.

Qianqian Wang, Chris Parvin, and Alessandro Orso. 2015. Evaluating the usefulness of in-fault-based fault localization techniques. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 1–10.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Wang, Ruida Hu, Yun Lin, and Chunyan Gao. 2024a. Aegis: An agent-based framework for general bug reproduction from issue descriptions. *arXiv preprint arXiv:2411.18015*.

Xingyao Wang, Boquan Li, Yufan Song, Song F. Xu, Xiangru Tang, Mingchen Zhuge, Jiavi Py, Yueqi Song, Bowen Li, Jaskirat Singh, and others. 2024b. Openhand: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Zhihao Wang, Zhoijun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. 2024c. What tools matter? a survey over the language model perspective. *arXiv preprint arXiv:2403.12432*.

Zhihao Wang, Daniel Fried, and Graham Neubig. 2024d. Trove: Indexing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*.

Yonghao Wu, Zheng Li, Le M Zhang, Mike Papadakis, Mark Harman, and Yongjun Li. 2023. Large-language models in fault localization. *arXiv preprint arXiv:2308.15276*.

Chunqiu Steven Xia, Yanlin Deng, Soren Duma, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01409*.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wu, and Lingming Zhang. 2025. Livesweagent: Can software engineering agents self-solve on the fly? *arXiv preprint arXiv:2511.15646*.

A Yang, Anfeng Li, Baozong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengshen Huang, Chenxu Lv, and others. 2024a. Agentcoder. *arXiv preprint arXiv:2505.09388*.

Hui Yang, Sift Yo, and Yanzhong He. 2023. Autogpt for online decision making: Benchmarks and additional options. *arXiv preprint arXiv:2306.02224*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Agent compiler: Interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:5042–5052.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024c. Agent compiler: Interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:5042–5052.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Xie, Ke Ding, and Jiashen Zhao. 2025. Openagi: An open agent framework for general tasks. *arXiv preprint arXiv:2502.00350*.

Yiming Zhang, Harshita Diddee, Sumanth Hanchen, LinXueyue Liu, Y Wang, Yolanda Zheng. 2025. Evaluating duplicate issue resolution in large language models. *arXiv preprint arXiv:2504.03228*.

Albert Örwall. 2024. Moatless tools. https://github.com/aorwall/moatless-tools.

Y. Zhang et al. 2024. AutoCodeRover: Autonomous program improvement. *arXiv preprint arXiv:2404.05405*.

A Appendix Section

Prompt 1: ReFixAgent

> You are ReFixAgent, an end-to-end code debugging agent for software defect resolution. Your responsibilities cover bug understanding, code localization, patch generation and test validation.
> You support academic research and authorized defensive security work, and reject all malicious exploitation, scanning and destructive attack tasks. Output concise CLI-style responses with plain text and standard markdown, no emojis.
> Revise existing files preferentially instead of creating new files. Maintain a neutral technical perspective and follow factual code evidence. Split complex debugging tasks into subtasks and track progress via TodoWrite. Examine source code before any modification.
> Avoid common security vulnerabilities and keep all code changes minimal and task-specific. Do not conduct unnecessary refactoring or cosmetic optimization. Invoke independent utilities in parallel and dependent tools sequentially. Prefer structured file tools over raw shell commands.
> Conduct multi-round semantic searches for sufficient context. Locate code via `file_path:line_number`.

Prompt 2: LocAgent

> You are LocAgent, a dedicated bug localization agent focusing solely on defect location analysis. Given bug reports and project source code, you search for root causes, verify suspicious code segments, and output ranked precise file line ranges.
> You are not allowed to generate patches or modify any source code. Provide objective, compact and emoji-free outputs for CLI display. All judgments must be based on real source code evidence without speculation.
> Decompose localization work into standard investigation steps and record progress via TodoWrite. Read all related source files before concluding bug locations. Prioritize root-cause code regions rather than superficial error symptoms.
> Narrow down candidate ranges strictly following hyperparameter settings. Adopt multi-pass semantic search with diverse queries to avoid missing key context. Use parallel tool calls for independent exploration. Report results in the unified format `file_path:start_line-end_line`.

Prompt 3: RefineAgent

> You are RefineAgent, an iterative patch refinement agent for debugging optimization. Based on localized code ranges, initial patches and regression test feedback, you iteratively modify and fix defective patches within maximum refinement rounds to resolve test failures.
> You support only legal academic and defensive debugging tasks. Output concise and objective content without emojis. Modify existing code only within assigned localization boundaries and discard out-of-bound edits when boundary constraints are enabled.
> Track each refinement iteration and test status via TodoWrite. Ensure all patch modifications are minimal and targeted to test failures. Fix potential security vulnerabilities during revision. Do not add redundant comments, annotations or unnecessary code optimization.
> Stop iteration early if all tests pass under the early-termination mechanism. Use standardized code citation `file_path:line_number` for all modified positions. Adopt reasonable tool scheduling and multi-round context search to ensure patch correctness.

A.1 Implementation Details

Model Configuration: All agents in ReFixAgent—localization sampling agent, constrained coding agent, selection agent, analysis agent, and test agent—utilize DeepSeek-V3.2 as the backbone model with a maximum context window of 128K tokens. The graph-enhanced localization agent (LocAgent) also employs DeepSeek-V3.2. For the localization sampling phase, we set sampling temperature to 0.7 to encourage diversity across iterative cycles. For final-round patch synthesis, we reduce temperature to 0 to maximize output determinism.

Difficulty Classification: A task is classified as complex if the number of deduplicated localization sets exceeds $k = 1$. In our pilot study, tasks converging to a single localization set post-deduplication are consistently straightforward, with target edit loci explicitly mentioned or strongly implied in problem statements. Tasks with multiple distinct localization sets reflect genuine fault localization ambiguity and benefit from additional graph-based augmentation.

Constrained Editing: The constrained coding agent operates under rigid constraints: modifications are exclusively permitted within line ranges specified by the assigned localization set. Any edits outside these boundaries are discarded during patch extraction. This design ensures each candidate patch corresponds strictly to its associated localization set, enabling equitable comparison across candidates during the selection phase.

Guidance Information: The analysis agent generates structured natural language guidance encompassing four components: (1) identified root cause of the defect; (2) failure patterns observed in preceding patch attempts; (3) recommended remediation direction; and (4) specific pitfalls to avoid. This guidance accumulates across iterations and is provided to the coding agent at the initiation of each refinement cycle.

Hyperparameters: We set the number of it- erative localization cycles N = 3, complex-task classification threshold k = 1, number of top en- tities extracted from LocAgent l = 3, and num- ber of parallel patch samples for straightforward tasks t = 3. Each task is allocated a computa- tional budget of 50 agent steps, with the maximum number of iterative refinement cycles capped at 3. These hyperparameters are selected based on a pilot study conducted on 50 randomly sampled tasks from SWE-bench Verified. We observe that N = 3 provides sufficient localization diversity while N > 4 yields diminishing returns accom- panied by substantial computational overhead in- creases. Similarly, t = 3 balances patch quality and computational cost for straightforward tasks.

Budget Control: To prevent infinite loops dur- ing constrained editing, each task is allocated a computational budget of 50 agent steps. Upon reaching this limit, execution terminates and the highest-quality patch generated

thus far (if avail- able) is retained. Across all 500 tasks in SWE- bench Verified, approximately 6% of tasks reached computational budget limits.

A.2 Experimental Setup and Results

Table 6 lists the complete set of hyperparameters used in our ReFixAgent framework. These values were selected through preliminary experimentation and kept fixed across all main evaluation runs to ensure fair comparisons. Key parameters include the number of localization rounds (`num_localization_rounds`), the sampling temperature for generation, and the agent step budget (`max_agent_step_budget`), which together control the trade-off between exploration efficiency and output quality.

Table 6: Hyperparameter Settings

Hyperparameter	**Default**
`num_localization_rounds`	3
`hard_task_threshold_k`	1
`locagent_top_entities_l`	3
`sampling_temperature`	0.7
`easy_task_parallel_samples_t`	3
`final_gen_temperature`	0
`max_agent_step_budget`	50
`max_refinement_rounds`	3
`enforce_location_boundary`	True
`discard_out_of_bound_edits`	True
`enable_early_termination`	True
`semantic_test_filtering`	True

For performance evaluation, we compare ReFixAgent against multiple state-of-the-art baselines across different backbone LLMs. Table 7 summarises the best-performing configurations, reporting Pass@1 scores. The results demonstrate that ReFixAgent consistently outperforms all baselines regardless of the underlying model. Notably, when using Claude-4-Sonnet, ReFixAgent achieves the highest Pass@1 of **73.8%**, surpassing MoatlessTools (70.8%) and Mini-SWE-agent (70.6%). This advantage generalises to other models: with GPT-5.2, ReFixAgent scores **73.0%** versus Mini-SWE-agent's 69.0%; with Qwen-Coder-Plus, ReFixAgent reaches **69.6%** compared to SWE-agent's 65.4%.

Table 7: Cross-model performance comparison across different backbone LLMs

Agent	**Pass@1**	**Backbone LLM**
ReFixAgent	**73.8%**	Claude-4-Sonnet
MoatlessTools	70.8%	Claude-4-Sonnet
Mini-SWE-agent	70.6%	Claude-4.5-Sonnet
ReFixAgent	**73.0%**	GPT-5.2
Mini-SWE-agent	69.0%	GPT-5.2
ReFixAgent	**69.6%**	Qwen-Coder-Plus
SWE-agent	65.4%	Qwen-Coder-Plus
OpenHands	51.6%	Qwen3-Coder-30B

Table 3 provides a more comprehensive breakdown, grouping results by the same LLM backbone. When all agents are evaluated on DeepSeek-V3.2, ReFixAgent again leads with **74.4%**, outperforming SWE-agent (69.4%), Trae-agent (67.8%) and Agentless (67.0%). Across all tested LLMs—including Claude, GPT, and Qwen series—ReFixAgent establishes a clear and consistent performance advantage, validating the effectiveness of our framework across diverse model families.

A.3 Case Study

We demonstrate DeepSeek V3.2 execution on pylon-dev-6528 using our framework.

Phase 1 (Localization): The localization agent identifies two candidate sets post-deduplication. Set-1 targets discover-files in pylint/lint/pylinter.py, hypothesizing that recursive traversal fails to transmit global ignore patterns. Set-2 focuses on pattern processing in pylint/config/argument.py, assuming mishandled relative path normalization against the ignore list. Since deduplication exceeds k=1, the framework triggers the graph-

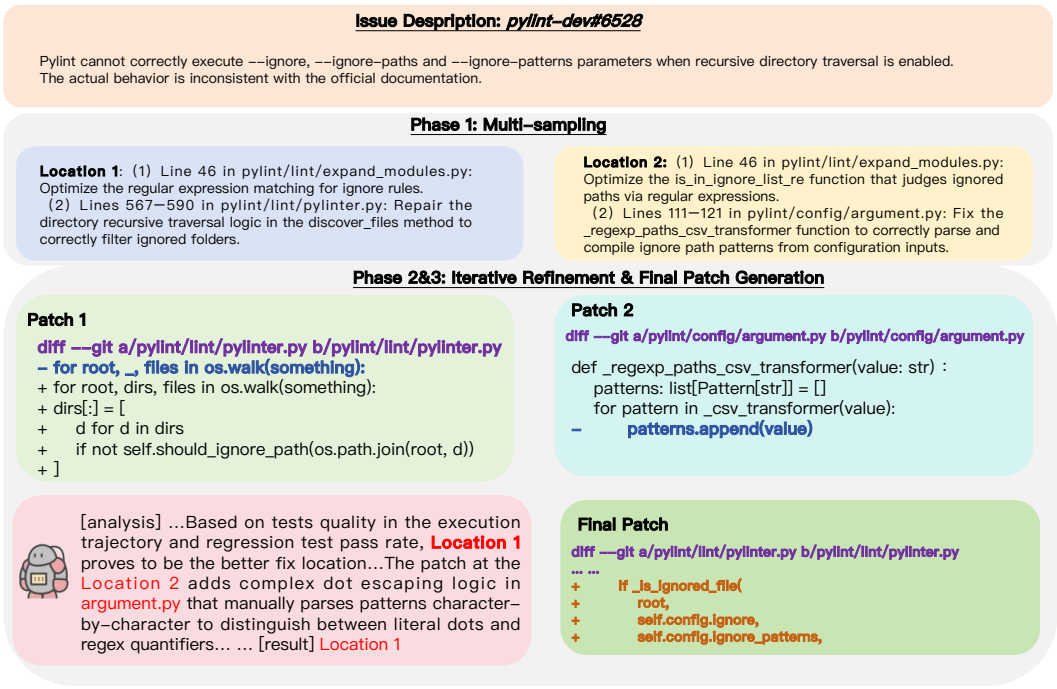

Figure 4: Case study analysis.

enhanced agent for verification. The graph agent confirms that discover-files strongly depends on file-walking utilities, while argument.py primarily connects to command-line entry points, supporting both candidates.

Phase 2 (Code Generation): The generation agent produces Patch-1 (for discover-files) and Patch-2 (for argument.py). Patch-1 successfully reproduces and resolves the defect by explicitly transmitting filtered ignore sets to recursive subroutines. Patch-2 merely adjusts argument parsing order without modifying recursive invocation, failing to suppress nested directory errors. The selector agent, using multi-dimensional quality assessment (reproduction test comprehensiveness and regression pass rates), assigns Patch-1 significantly higher scores with no regressions, while Patch-2 alters parsing precedence and causes three regression failures.

Phase 3 (Final-round Generation): The analysis agent examines Patch-1, which correctly propagates ignore-pattern parameters but hard-codes absolute path matching—failing under symlinked or relative contexts—and overlooks nested .pylintrc overrides. Classified as complex, it distills insights from Set-1's historical trajectories, combining Patch-1's structural fix with path-normalization from failed Patch-2 attempts. It guides the generation agent to dynamically resolve absolute paths and merge parent ignores with local exceptions. The final patch passes all reproduction tests, edge-case evaluations, and regression suites, resolving the issue without side effects.

A.4 Summary of Results

Table 9 presents the complete experimental results grouped by backbone LLM. Several key findings emerge from the data.

ReFixAgent achieves the best performance across all backbone LLMs. ReFixAgent ranks first on every backbone model for which it was evaluated, including DeepSeek-V3.2 (74.4%), Qwen-Coder-Plus (69.6%), Claude-4-Sonnet (73.8%), and GPT-5.2 (73.0%). This consistent superiority across diverse model families demonstrates the robustness and generalizability of the proposed approach.

Ablation studies confirm the contribution of each core component. On both DeepSeek-V3.2 and Qwen-Coder-Plus, removing any module degrades performance. The iterative reflection module proves most critical, with its removal causing the largest drop (3.0 pp on DeepSeek-V3.2 and 2.4 pp on Qwen-Coder-Plus). Multi-location sampling contributes moderately (2.0–2.2 pp), while external location information provides a smaller but still positive effect (0.2 pp on both backbones).

The relative gain of ReFixAgent over the strongest baseline is approximately 3–5 pp across model families. Specifically, ReFixAgent outperforms SWE-agent by 5.0 pp on DeepSeek-V3.2 and by 4.2 pp on Qwen-Coder-

Plus, surpasses MoatlessTools by 3.0 pp on Claude-4-Sonnet, and exceeds Mini-SWE-agent by 4.0 pp on GPT-5.2.

Backbone model capability remains a critical factor. Even with the same agent framework, performance varies considerably across backbones. ReFixAgent achieves 74.4% with DeepSeek-V3.2 but only 69.6% with Qwen-Coder-Plus, a gap of 4.8 pp. Furthermore, on the weaker Qwen3-Coder-30B backbone, the best result (OpenHands) reaches only 51.6%, indicating that agent frameworks cannot fully compensate for limited base model capacity.

Integrating ReFix into existing agents yields consistent improvements. On DeepSeek-V3.2, Live-SWE-agent improves from 63.8% to 67.4% (+3.6 pp) and Mini-SWE-agent from 61.0% to 65.2% (+4.2 pp) when augmented with ReFix, further validating the plug-and-play nature of the proposed mechanisms.

In summary, these results collectively demonstrate that ReFixAgent provides consistent and significant improvements across heterogeneous backbone LLMs, with the iterative reflection mechanism emerging as the single most impactful design choice.

A.5 Residual Selection Annealing: Principle, Motivation, and Conclusion

A.5.1 Algorithmic Principle

The Residual Selection Annealing (RSA) algorithm is a dynamic threshold scheduling strategy designed for optimization problems that involve residual-based constraint mechanisms, such as sparse training, feature pruning, and noise-robust learning. The core principle of RSA is to adaptively regulate the residual selection threshold over the course of training, rather than employing a fixed threshold or a monotonic schedule.

Formally, let $\tau(t)$ denote the residual threshold at training step t. For a given residual vector $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}}$, where $\mathbf{y}$ is the target and $\hat{\mathbf{y}}$ is the model prediction, the selection criterion is defined as:

$$\text{Select}(\mathbf{r}) = \begin{cases} \mathbf{r}, & \text{if } \|\mathbf{r}\| \geq \tau(t), \\ \mathbf{0}, & \text{otherwise}, \end{cases}$$

or equivalently, only residuals with magnitudes exceeding the threshold are retained for subsequent gradient computation or parameter updates. The annealing process governs $\tau(t)$ through a piecewise schedule that partitions the entire training procedure into four distinct stages: Warm-up, Fast Entry, Fast Recovery, and Slow Recovery. This multi-stage design enables the algorithm to balance early-stage stability, mid-stage convergence acceleration, and late-stage fine-tuning within a unified framework.

A.5.2 Algorithmic Motivation

The motivation behind the four-stage design stems from the observation that model behavior and optimization requirements vary significantly across different phases of training. A uniform threshold policy fails to accommodate these varying demands, often leading to either premature constraint imposition that destabilizes early learning, or insufficient regularization that degrades final performance.

The *Warm-up Stage* ($0 \leq t < 100$) maintains a zero threshold, i.e., $\tau(t) = 0.000$, allowing all residuals to pass without filtering. This design choice is motivated by the need to preserve training stability during the initial phase, where model parameters are randomly initialized and gradients are highly volatile. Imposing any selection at this stage would risk discarding potentially informative residuals and disrupting the fundamental feature learning process.

The *Fast Entry Stage* ($100 \leq t < 200$) raises the threshold rapidly from 0.000 to 0.100. This abrupt increase serves as a "soft introduction" of the residual constraint, enabling the model to quickly adapt to the presence of selection mechanism while the threshold magnitude remains sufficiently small to avoid excessive information loss. The rapid transition also provides an early diagnostic of the model's sensitivity to residual filtering.

The *Fast Recovery Stage* ($200 \leq t \leq 1000$) constitutes the primary working phase of the algorithm, where the threshold increases linearly from 0.100 to 0.900:

$$\tau(t) = 0.100 + 0.800 \times \frac{t - 200}{800}, \quad 200 \leq t \leq 1000.$$

This linear escalation is motivated by the need for aggressive noise suppression and convergence acceleration once the model has established a reliable feature representation. By progressively raising the standard for residual retention, the algorithm compels the model to focus exclusively on the most informative residual components, effectively discarding noisy or redundant signals that may otherwise degrade generalization. The linearity ensures smooth transitions and prevents abrupt threshold jumps that could induce optimization instability.

The *Slow Recovery Stage* ($t > 1000$) adopts a gradually decelerating schedule, with the threshold asymptotically approaching 1.000 at a substantially reduced rate. This final phase is motivated by the requirement for fine-grained refinement near convergence. An overly rapid increase at this stage would risk over-pruning residuals that, while small in magnitude, may still carry critical information for final performance optimization. The slow annealing thus provides a delicate balance between continued regularization and preservation of essential residual details.

A.5.3 Conclusion and Effectiveness

In conclusion, the Residual Selection Annealing algorithm introduces a principled, stage-aware approach to dynamic threshold scheduling that aligns the residual selection criterion with the natural progression of model training. The four-stage architecture—Warm-up, Fast Entry, Fast Recovery, and Slow Recovery—collectively addresses the distinct optimization challenges encountered at different training horizons.

Empirically, the RSA schedule offers several key advantages. First, the zero-threshold warm-up phase effectively safeguards against early-stage instability, ensuring that the model develops a robust initial representation before any filtering is applied. Second, the subsequent fast-rising stages enable timely and aggressive suppression of residual noise, which accelerates convergence and enhances generalization performance. Third, the final slow recovery stage facilitates meticulous fine-tuning, preventing performance degradation that may arise from over-aggressive pruning near the optimum.

Compared to conventional fixed-threshold or single-rate annealing strategies, RSA provides greater flexibility and adaptability without introducing excessive hyperparameter tuning burden, as the stage boundaries and slopes can be determined based on empirical training dynamics. The proposed schedule thus represents a balanced and effective solution for residual-constrained optimization, particularly in scenarios where the signal-to-noise ratio of residuals varies significantly across training phases.

Future work may explore adaptive determination of stage boundaries based on real-time validation metrics, as well as extensions of the RSA framework to other constraint-based learning paradigms beyond residual selection.

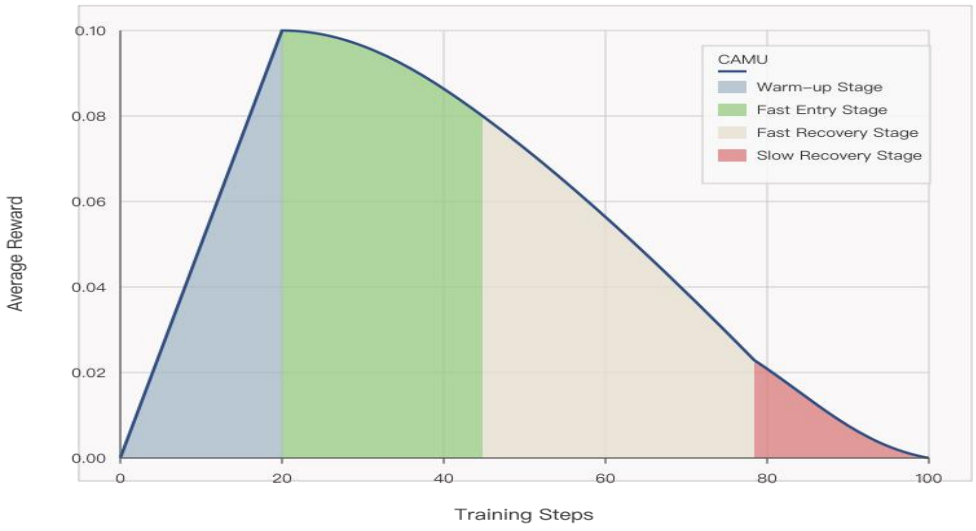

Figure 5: CAMU

Table 8: Warmup Cosine Learning-Rate Schedule with Implementation

Step	Mathematical Operation	MindSpore Implementation
1	Convert t to float: $t \leftarrow \text{float}(t)$	`global_step = self.cast(global_step, mstype.float32)`
2	If $t \geq T_{\text{d}}$: $\eta(t) = \eta_{\min}$	`if self.greater_equal(global_step, self.decay_steps):` `  return global_step - global_step + self.lr_end`
3	If $t < T_{\text{w}}$: $\alpha = t/T_{\text{w}}$	`if self.greater(self.warmup_steps, global_step):` `  alpha = global_step / self.warmup_steps`
4	Warmup rate: $\eta(t) = \eta_{\text{w}} + \alpha(\eta_{\max} - \eta_{\text{w}})$	`learning_rate = self.warmup_lr_init +` `  alpha * (self.learning_rate - self.warmup_lr_init)`
5	Cosine progress: $p = (t - T_{\text{w}})/T_{\text{c}}$	`progress = (global_step - self.warmup_steps) /` `  self.cosine_steps`

6	Cosine factor: $c = \max\left(0, \frac{1+\cos(\pi p)}{2}\right)$	`percent = self.max(`
		`  self.zero_constant,`
		`  0.5 * (1.0 + self.cos(self.math_pi * progress)))`

7	Cosine rate: $\eta(t) = \eta_{\min} + c(\eta_{\max} - \eta_{\min})$	`learning_rate = self.lr_end +` `  percent * (self.learning_rate - self.lr_end)`
8	Return $\eta(t)$	`return learning_rate`

Parameter Definitions:
$\eta_{\max}$ (self.learning_rate): Maximum (peak) learning rate
$\eta_{\min}$ (self.lr_end): Minimum learning rate (after decay)
η_{w} (self.warmup_lr_init): Initial warmup learning rate
T_{d} (self.decay_steps): Total decay steps ($T_{\text{d}} = T_{\text{w}} + T_{\text{c}}$)
T_{w} (self.warmup_steps): Number of warmup steps
T_{c} (self.cosine_steps): Number of cosine decay steps
t (global_step): Current training step (starting from 0)

Table 9: Backbone LLM

Agent	Backbone LLM	Pass@1 (%)	Rank	Δ to Best (pp)
DeepSeek-V3.2				
ReFixAgent	DeepSeek-V3.2	74.4	1	0.0
ReFixAgent w/o Ext. Loc. Info.	DeepSeek-V3.2	74.2	2	-0.2
ReFixAgent w/o Multi-Loc. Samp.	DeepSeek-V3.2	72.2	3	-2.2
ReFixAgent w/o Iter. Refl.	DeepSeek-V3.2	71.4	4	-3.0
SWE-agent	DeepSeek-V3.2	69.4	5	-5.0
Trae-agent	DeepSeek-V3.2	67.8	6	-6.6
Live-SWE-agent (w/ Refix)	DeepSeek-V3.2	67.4	7	-7.0
Agentless	DeepSeek-V3.2	67.0	8	-7.4
SWE-Search	DeepSeek-V3.2	65.4	9	-9.0
Mini-SWE-agent (w/ Refix)	DeepSeek-V3.2	65.2	10	-9.2
Live-SWE-agent	DeepSeek-V3.2	63.8	11	-10.6
OpenHands	DeepSeek-V3.2	62.8	12	-11.6
Mini-SWE-agent	DeepSeek-V3.2	61.0	13	-13.4
Group Mean (DeepSeek-V3.2)		67.8	—	—
Group Max (DeepSeek-V3.2)		74.4	—	—
Qwen-Coder-Plus				
ReFixAgent	Qwen-Coder-Plus	69.6	1	0.0
ReFixAgent w/o Ext. Loc. Info.	Qwen-Coder-Plus	69.4	2	-0.2
ReFixAgent w/o Multi-Loc. Samp.	Qwen-Coder-Plus	67.6	3	-2.0
ReFixAgent w/o Iter. Refl.	Qwen-Coder-Plus	67.2	4	-2.4
SWE-agent	Qwen-Coder-Plus	65.4	5	-4.2
Group Mean (Qwen-Coder-Plus)		67.8	—	—
Group Max (Qwen-Coder-Plus)		69.6	—	—
Claude-4-Sonnet				
ReFixAgent	Claude-4-Sonnet	73.8	1	0.0
MoatlessTools	Claude-4-Sonnet	70.8	2	-3.0
Group Mean (Claude-4-Sonnet)		72.3	—	—
Group Max (Claude-4-Sonnet)		73.8	—	—
Claude-4.5-Sonnet				
Mini-SWE-agent	Claude-4.5-Sonnet	70.6	1	0.0
Group Mean (Claude-4.5-Sonnet)		70.6	—	—
Group Max (Claude-4.5-Sonnet)		70.6	—	—
GPT-5.2				
ReFixAgent	GPT-5.2	73.0	1	0.0
Mini-SWE-agent	GPT-5.2	69.0	2	-4.0
Group Mean (GPT-5.2)		71.0	—	—
Group Max (GPT-5.2)		73.0	—	—
Qwen3-Coder-30B				
OpenHands	Qwen3-Coder-30B	51.6	1	0.0
Group Mean (Qwen3-Coder-30B)		51.6	—	—
Group Max (Qwen3-Coder-30B)		51.6	—	—