REFINE: A General Context-Aware Agent for Evolving Program Repair Agents

Anonymous ACL submission

Abstract

When applying LLM agents to automatic program repair (APR), existing systems often produce DRAFT PATCHES—partially correct fixes that either fail to fully resolve bugs or overfit public test cases due to incomplete contextual reasoning and limited test coverage. To address this, we present REFINE, a general context-aware refinement agent that evolves existing APR agents by transforming DRAFT PATCHES into robust and correct fixes. Operating in a black-box, plug-and-play manner, REFINE augments diverse APR systems without model retraining or architectural changes. REFINE tackles three challenges: (1) resolving ambiguous issue context through structured contextualization, (2) improving patch quality via test-time scaling and diversified generation, and (3) consolidating partial fixes through LLM-driven refinement. We integrate REFINE into multiple APR systems to demonstrate its generality and system-agnostic design. On SWE-Bench Lite, REFINE achieves state-of-the-art performance, improving AutoCodeRover by 14.67% to 51.67% resolution. These results identify refinement as a missing yet essential layer in APR pipelines and highlight agent-level refinement as a practical path toward more correct and robust program repair systems. We also open source our code here: *Anonymous GitHub*.

1 Introduction

Large Language Models (LLMs) have shown strong capabilities in automatic program repair (APR) (Zhang et al., 2024; Ruan et al., 2024; Wang et al., 2025; Yang et al., 2024; Zhang et al., 2024; Xia et al., 2024). In repository-level APR, the input consists of an issue description, the repository source code, and public regression tests, with the goal of synthesizing a correct patch through repository-level reasoning. The generated patch is validated against both public and hidden test suites..

Limitations. Despite recent progress, APR systems still struggle on complex repositories due to limited code-context understanding and overfitting to public test suites. Our study shows these issues often produce DRAFT PATCHES—partially correct patches that either incompletely fix the bug (INCOMPLETE PATCHES) or overfit to weak public tests (OVERFITTED PATCHES). These failure modes are common across both open-agent-based and workflow-based APR systems, yet existing methods lack systematic mechanisms to refine them into correct patches. This gap motivates our work: *developing principled strategies that can refine DRAFT PATCHES into correct and robust fixes.*

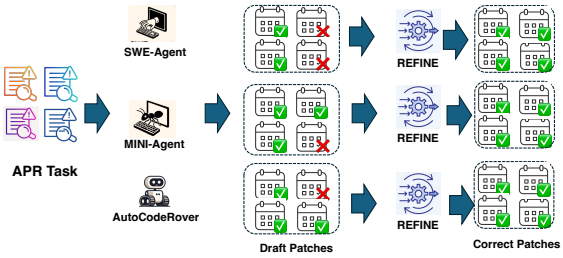

Figure 1: An example of patch refinement

Prior work (Yu et al., 2019; Fei et al., 2025; Tian et al.; Smith et al., 2015; Ye and Monperrus, 2024) has explored patch overfitting and iterative refinement in function-level repair, but these methods are typically tool-specific and tightly coupled to particular repair pipelines. Consequently, they generalize poorly to repository-level APR systems with diverse search and patch generation strategies, limiting the reuse of refinement techniques across different APR frameworks.

To address this gap, there is a clear need for a black-box, tool-agnostic refinement module that serves as an evolutionary layer for existing APR systems, refining near-correct patches from heterogeneous agents into fully correct solutions (as shown in Fig. 1). Designing such a generic refinement module introduces three key challenges: (1) *Lack of Precise Context.* Refining a DRAFT PATCH requires rich context, but issue descriptions

are often vague and key code context is missing, leading to misinterpretation or incomplete fixes. (2) *Limited Exploration.* Existing methods under-explore the complementary space of possible patch deltas, reducing the likelihood of finding effective refinements. (3) *Weak Patch Selection.* Even with multiple candidates, current approaches lack principled selection mechanisms, often resulting in suboptimal refined patches.

Our Design. To address the aforementioned challenges, we propose Refine, including the following three modules: (1) *Context Regularization Agent*: an agent that disambiguates and augments issue and code context for better refinement; (2) *Diverse Delta Generation*: test-time scaling to produce multiple *Delta Patch* candidates for broader exploration; and (3) *Aggregation Agent*: a code review agent that combines useful signals from *Delta Patches* with the original Draft Patch to produce the final fix.

We first evaluate Refine on five APR systems, consistently improving their resolution rates on SWE-Bench Lite and demonstrating its tool-agnostic generality. We then conduct a full evaluation with AutoCodeRover on SWE-Bench Lite and SWE-Bench Verified, achieving 14.67% and 12.2% improvements, respectively, and outperforming prior baselines on SWE-Bench Lite. Finally, ablation studies show that each component contributes to refining Draft Patches, confirming the effectiveness of the full framework.

We summarize our contribution as follows:

1. **Problem Novelty.** We introduce *patch refinement*, a new task that iteratively transforms Draft Patches into correct patches. Our refinement module is general and compatible with existing APR systems.
2. **Technique Novelty.** We present Refine, a black-box plug-in framework for patch refinement with three components: (1) context agents for context regularization, (2) a test-time scaling module that generates diverse `patch deltas`, and (3) a code review agent that aggregates these deltas into correct patches.
3. **Evaluation.** Refine achieves state-of-the-art performance, reaching 51.67% on SWE-Bench Lite and 63.8% on SWE-Bench Verified, improving resolve rates by 14.67 and 12.2 points, respectively.

2 Background & Related Work

2.1 LLM for Software Engineering

With the rise of large language models (LLMs), there has been growing work on applying them to software engineering tasks such as code generation (Chen et al., 2021, 2024a, 2025; Jain et al., 2024; Ding et al., 2024), program repair (Gao et al., 2022; Li et al., 2024), test generation (Kang et al., 2023; Chen et al., 2024c; Ryan et al., 2024), code review (Li et al., 2022), malware detection (Chen et al., 2020; Al-Karaki et al., 2024; Jelodar et al., 2025), as well as code summarization, translation, and refactoring (Sun et al., 2024; Pan et al., 2024; Chen et al., 2024b; Luo et al., 2024).

Among these, code generation has received particular attention, driven by tools such as GitHub Copilot, Amazon CodeWhisperer, and Claude Code, alongside a rapidly expanding ecosystem of LLMs ranging from closed models (e.g., ChatGPT, Claude, Gemini) to open-source alternatives (e.g., StarCoder, CodeLlama, DeepSeek-Coder). Beyond generation, LLMs are increasingly used to automate and improve other software engineering tasks, including test creation, code review, documentation, translation, and refactoring, enabling more efficient and automated development workflows.

2.2 Automatic Program Repair

Traditional APR Techniques. Early APR methods include search-based approaches (Weimer et al., 2009), which generate patches via predefined code mutations and test-based filtering, and semantics-based approaches (Mechtaev et al., 2016; Nguyen et al., 2013), which solve repair constraints derived from test specifications. While effective, both suffer from limited scalability and patch diversity.

Learning-based APR Techniques. To overcome these limitations, learning-based methods (Lutellier et al., 2020; Zhu et al., 2021) use neural models trained on large code corpora, often incorporating GitHub issues and bug reports (Koyuncu et al., 2019) to improve repair accuracy.

LLM-based APR for Repository-level Repair. Recently, LLM-based agentic approaches have enabled repository-level repair, supported by benchmarks such as SWE-Bench (Jimenez et al., 2023; Rashid et al., 2025; Zan et al., 2025; Mathai et al., 2024). These methods are broadly divided into (1) *open-agent frameworks*, where LLMs dynamically plan and use tools (e.g., SWE-Agent (Yang et al., 2024), OpenHands (OpenHands CodeAct

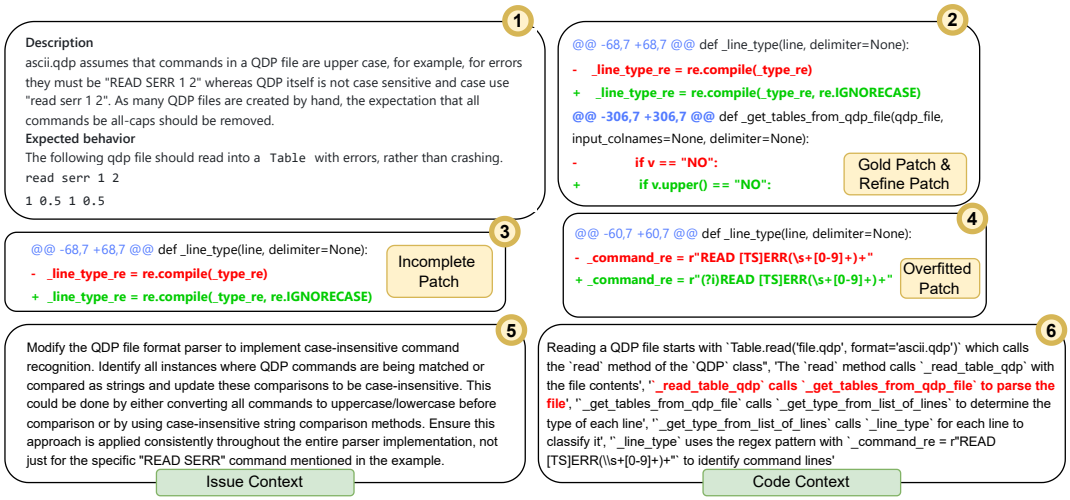

Figure 2: Motivating Example - ❶ is the issue description, ❷ is the developer patch & REFINE patch, ❸ and ❹ are incomplete/overfitted patches, ❺ is the REFINE generated issue semantics, ❻ is the REFINE generated code semantics.

2.1, 2024)), and (2) *workflow-based frameworks*, which follow a fixed Search-Edit-Test pipeline (e.g., Agentless (Xia et al., 2024), AutoCodeRover (Zhang et al., 2024), SpecRover (Ruan et al., 2024), Patch Pilot (Li et al., 2025)).

3 Motivation

3.1 Astropy Issue

To motivate our approach, we examine `astropy-14635`, a SWE-Bench Lite task that challenges many SOTA methods. As shown in Figure 2, the user-reported issue (box ❶) highlights a bug in the Astropy (Project, 2011) library's QDP file reader, which incorrectly assumes all input must be uppercase. The report includes a crashing example—"read serr 1 2"—and argues that the reader should accept lowercase input, as QDP is case-insensitive. We observe that SOTA agents often overanalyze the issue description, producing patches that either are incomplete (i.e., fail to generalize to edge cases) or overfit to the user inputs (Figure 2, boxes ❸ and ❹, respectively). In contrast, REFINE generates a consistent, generalizable patch aligned with the developer-written fix (box ❷) by leveraging execution, issue, and code semantics.

3.2 Issue Context

The patch in box ❹ illustrates *overfitting*, where the agent fixes only the specific example input in the issue report while missing the broader underlying bug. We find this behavior common in SOTA agents, including Agentless (Xia et al., 2024), which often hyper-localize based on the issue text. This leads to superficial fixes that act as band-aids rather than addressing root causes. We hypothesize that supplying agents with a deeper understanding of the issue (i.e., *issue semantics*) can mitigate overfitting. Box ❺ shows REFINE generated issue semantics, which generalize the problem and explicitly guide edits across multiple components, enabling more robust and complete patches.

3.3 Code Context

The patch in box ❸ illustrates an *incomplete* fix—while it addresses the core issue, it overlooks necessary updates elsewhere in the code (e.g., a related `if` condition). Advanced agents like SpecRover (Ruan et al., 2024) are able to avoid "overfitted" patches and instead generate such "incomplete" patches. We hypothesize that fine-grained code semantics can help agents reason about these secondary changes. Box ❻ shows a semantic trace from the QDP module; the red-highlighted step prompts REFINE to identify and apply additional edits, resulting in a complete and consistent patch.

To this end, we show that *issue*, and *code* context are crucial for generating complete patches. We now describe REFINE's overall methodologies behind each component (§4).

4 Methodology

4.1 Problem Formulation

We formally define the *patch refinement* problem as the task of incrementally improving an initial patch,

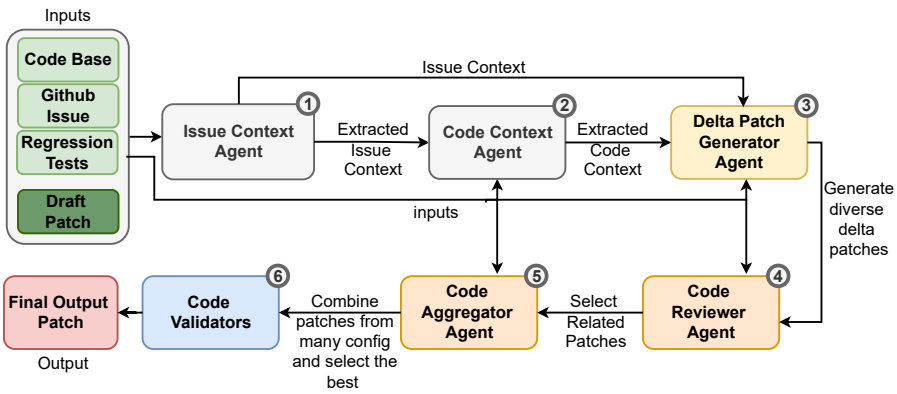

Figure 3: **Pipeline of REFINE.** REFINE takes as input a codebase, GitHub issue, regression tests, and an initial draft patch. First, an issue context extractor analyzes the issue description and codebase to extract relevant context ①. Then, a code context extractor uses this context and the draft patch to select relevant code regions ②. Leveraging both contexts and the original inputs, REFINE generates diverse delta patches ③ via test-time scaling. A reviewer agent merges each delta with the draft patch ④, and an aggregator agent ranks them based on alignment with the issue context ⑤. Finally, a validation phase ⑥—combining test execution and LLM-based judgment—selects the final patch.

generated by a program repair tool, toward a final, correct repair. Consider the standard repository-level program repair setting. Let $x = (\mathcal{I}, \mathcal{D}, \mathcal{T})$, where x denotes a program repair instance, fully characterized by the issue statement $\mathcal{I}$, the repository codebase $\mathcal{D}$, and a publicly available regression test suite $\mathcal{T}$. An existing program repair tool, denoted by $\mathsf{R}_{\text{init}}(\cdot)$, takes x as input and generates an initial (seed) patch: $\mathcal{P}_{\text{init}} = \mathsf{R}_{\text{init}}(x)$. Recall from §3, due to over-approximation or under-approximation, the initial *draft patch* may be either overfitted or incomplete. The goal of patch refinement is to provide a general solution for further improving these initial seed patches. Formally, the objective of patch refinement is to enhance the quality and correctness of $\mathcal{P}_{\text{init}}$ by applying a refinement function $\mathsf{F}(\cdot)$, resulting in a refined, and more accurate patch: $\mathcal{P}_{\text{final}} = \mathsf{F}(\mathcal{P}_{\text{init}})$.

4.2 Design Overview

Fig. 3 presents the design overview of REFINE. Given a program repair instance $x = (\mathcal{I}, \mathcal{D}, \mathcal{T})$ and an initial patch $\mathcal{P}_{init}$ generated by an existing APR tool, REFINE iteratively perform the following three main steps to produce more accurate patch:

1. *Context Extraction and Regularization.* In the first step, (① & ② of Fig. 3), REFINE leverages an agent to extract and regularize context information from both the issue statement and the draft patch. This process mitigates the inherent vagueness and ambiguity, while also providing a more structured and logical interpretation of the draft patch. As a result, subsequent steps benefit from clearer and more comprehensive contextual information (see §4.3).

2. *Diverse Delta Patch Generation.* After extracting the structural context from both the original issue statement and the draft patch, we integrate this information with the draft patch itself and employ an LLM agent to generate multiple diverse delta patches (③). In this step, we adopt the philosophy of test-time scaling and perform multiple sampling rounds, enabling us to explore a wider range of possible refinements and ultimately produce a set of diverse delta patches.

3. *Aggregated Patch Synthesis.* Finally, REFINE selects a subset of the generated diverse delta patches, aggregates them with the draft patch, and produces a final refined patch. If the resulting refined patch does not pass the public regression tests, it is treated as the new draft patch for the next iteration (④-⑥).

4.3 Context Extraction and Regularization

GitHub issue statements are often vague or incomplete, and draft patches from existing APR tools may lack structured semantic reasoning. To provide more precise guidance for repair, REFINE first extracts and regularizes context from both the issue statement and the draft patch. We define structured representations for these contexts and use an LLM agent to infer and refine them for reasoning.

4.3.1 Issue Context Format

Formally, we define an issue context (I) as a 5-tuple

$$I = (T, L, A, C, G) \tag{1}$$

where T (Target) denotes the system components implicated by the issue, L (Logic) specifies the intended behavioral change, A (Actions) describes the high-level steps required to implement the fix, C (Constraints) captures properties that must be preserved, and G (Generalization) defines the scope of applicability beyond the specific instance. An example of issue context could be found in Appendix A.1.

4.3.2 Code Context Format

We define the code context $\mathcal{C}(P)$ for a patch P as:

$$\mathcal{C}(P) = (\mathsf{DD}, \mathsf{CD}, \mathsf{IC}, \mathsf{CG}), \tag{2}$$

where (DD) denotes data dependencies, representing variables and expressions that are read or written, including transitive flows (e.g., line content and regex matches); (CD) denotes control dependencies, capturing control-flow constructs that affect execution such as conditional branches and loops; (IC) denotes invariant constraints, representing semantic assumptions and correctness conditions that must be preserved before and after the patch (e.g., input format validity and crash avoidance); and (CG) denotes the call graph context, i.e., a backward-traceable call chain showing which higher-level components transitively reach the location, thereby capturing its functional role within the system. An example of code context could be found in Appendix A.1.

4.3.3 Context Extraction

To ensure that the LLM agent extracts context from the original issue statement and the initial draft patch in accordance with the specified format, we leverage the few-shot learning capabilities of LLMs (Brown et al., 2020) and design two specialized context extraction agents using demonstration examples. Specifically, we define the extraction process as follows:

$$\mathcal{I}' = \text{Issue_Context_Extractor}(\mathcal{I}), \tag{3}$$

$$\mathcal{C}' = \text{Code_Context_Extractor}(\mathcal{P}_{\text{init}}) \tag{4}$$

where $\mathcal{I}'$ denotes the regularized context extracted from the issue statement, and $\mathcal{C}'$ represents the structured code context extracted from the initial draft patch.

4.4 Diverse Delta Patch Generation

After collecting the regularized contexts, we provide these information to the LLM agent to facilitate deeper reasoning about the limitations of the current initial seed patches and to generate delta patches that enhance the seed patch. To more thoroughly explore the solution space at this stage, we adopt the philosophy of *test-time scaling*, querying the LLM agent multiple times with the same prompt in sampling mode to collect diverse responses. *Test-time scaling* refers to the practice of generating multiple outputs from an LLM during inference—by varying sampling parameters such as temperature—to increase diversity and coverage in the generated solutions, a technique widely studied in prior work (Chen et al., 2021).

In detail, REFINE combines the regularized context with the initial seed patch to construct a prompt, then queries the LLM agent multiple times using this prompt. At each query, the agent is asked to generate an edit in a simple diff format, efficiently producing candidate patches. The diff format includes two parts: the search snippet (the code to be replaced) and the replacement snippet (the new code to insert). To apply the generated diff to the original patch, we simply match the search code snippet in the codebase and replace it with the replacement. This simple diff format avoids generating the complete code and instead focuses on producing small, targeted edits, which are not only more cost-efficient but also more reliable and accurate, with reduced risk of hallucinations. These delta patch variants reflect diverse perspectives on the issue and repair process. An aggregator module then reconciles the resulting set of refined patches into a unified, semantically consistent pool that preserves both the intent of the issue and the correctness of the code.

4.5 Aggregated Patch Synthesis

After collecting diverse delta patches in §4.4, REFINE aggregates them into a single valid fix for the target issue. This process consists of three components: a code reviewer agent, an aggregation agent, and a patch validator.

4.5.1 Code reviewer agent

While *test-time scaling* helps explore a broader search space and enables the generation of more diverse patches, it can also produce unrelated or irrelevant patches. To address this, REFINE simulates the code review process by leveraging the concept of LLM-as-Judge and employs a dedicated *code*

reviewer agent. Specifically, REFINE combines each delta patch with the seed patch and the issue context, then queries the LLM agent to determine whether the revised patch adequately addresses the issue described in the context. The LLM agent is asked to provide a simple yes or no response. In this way, REFINE effectively filters out unrelated patches, retaining only those that are relevant to the issue.

4.5.2 Aggregation agent

After filtering unrelated delta patches, REFINE removes duplicates and groups conflicting patches that modify the same code differently. The LLM then aggregates each group into a unified patch by merging compatible edits or resolving conflicts based on the issue description, and combines the resulting patches with the initial draft patch. Finally, the consolidated diffs are applied to the codebase to generate the final patch, while syntax correctness is enforced through linting and semantic issues are corrected iteratively in later refinement rounds.

4.5.3 Patch Validator

After generating the consolidated final patch, we validate its correctness using the publicly available regression tests. If the final patch passes all regression tests, REFINE returns it as the solution. Otherwise, the final patch is treated as a new initial seed patch, and the process is repeated until the maximum number of retry iterations is reached. Our evaluation results (§6.3) show that even with just one retry iteration, REFINE demonstrates significant improvements in bug-fixing accuracy.

5 Experimental Setup

RQ1. **Plugin Effectiveness:** To what extent does integrating REFINE as a plugin enhance the performance of different APR approaches?

RQ2. **Bug-Fixing Capability:** What is the overall performance of REFINE in resolving real-world bugs and generating valid patches?

RQ3. **Module Contribution:** What is the contribution of each individual module within REFINE to its overall performance and effectiveness?

5.1 Evaluation Dataset

We evaluate REFINE on SWEBench Lite & Verified (Jimenez et al., 2023), benchmarks consisting of 300 & 500 real-world GitHub issues across 12 Python repositories. For each issue, REFINE receives the issue description and pre-fix codebase with regression tests, and outputs a single patch in git diff format, which is evaluated for successful resolution.

5.2 Comparison Baselines

We compare REFINE with both workflow-based and agent-based APR systems. More details could be found in Appendix A.2.

Workflow-Based Approaches. Workflow-based approaches rely on manually defined pipelines. Specifically, we include (1)AutoCodeRover (Zhang et al., 2024), (2) Agentless (Xia et al., 2024), (3) SpecRover (Ruan et al., 2024), (4) ExpeRepair-v1.0 (Mu et al., 2025), and (5) OrcaLoca + Agentless-1.5 (Yu et al., 2025; Xia et al., 2024).

Agent-Based Approaches. These APR systems provide greater autonomy to underlying models, enabling more dynamic interaction with the development environment. Specifically, we consider: (1) SWE-Agent (Yang et al., 2024), (2) Moatless Tools / SWE-Search (Antoniades et al., 2024), (3) DARS Agent (Aggarwal et al., 2025), (4) Devlo (devlo.ai, 2024), (5) Blackbox AI Agent (Blackbox AI, 2024), (6) Globant Code Fixer Agent (Globant, 2024), (7) CodeV (Zhao et al., 2025), (8) Codart AI (Codart AI, 2025), (9) CodeStory Aide (Developers, 2025), and (10) Lingxi (Yang et al., 2025).

5.3 Implementation Details

To ensure a fair and controlled evaluation, REFINE uses Gemini 2.5 Pro as its backend LLM. Rather than upgrading to a newer generation model, **we deliberately select an LLM from the same capability tier as the baseline seed APR systems** (which utilize models like Gemini 2.5/3.0 or Claude 3.5/3.7/4.0). In fact, REFINE often employs a slightly weaker LLM configuration than these baselines, thereby demonstrating that our approach's performance gains stem from our design rather than backend LLM. In §6.3, we study the effect of different backend LLMs as well as key hyperparameters. By default, we use greedy decoding (temperature = 0) for deterministic patch generation, and optionally set temperature to 0.7 with up to 5 retry loops for more diverse exploration, stopping early when a valid patch is found. Further implementation details are provided in Appendix A.4.

Table 1: Overall effectiveness of REFINE in enhancing the performance of existing APR tools on SWE-Bench Lite.

APR	Initial	Post refinement	Increasement
AutoCodeRover	36.67%	56.67%	20.00%
Codev	50.00%	53.33%	3.33%
ExpeRepair	40.00%	60.00%	20.00%
Agentless	40.00%	56.67%	16.67%
BlackBoxAI	50.00%	60.00%	10.00%

Table 2: Ablation Study: Impact of Each Module.

Module			Resolved	Acc Drop
Ctx.	Div Delta	Reviewer	Issue Rate	
			37.00%	-7.66%
	✓	✓	37.33%	-7.33%
✓		✓	40.33%	-4.33%
✓	✓		42.00%	-2.66%
✓	✓	✓	44.66%	-

6 Results

6.1 RQ1. Results

Setup. We first apply REFINE to a diverse set of state-of-the-art APR systems, including AutoCodeRover, Agentless, CodeV, BlackBoxAI, and ExpeRepair, covering both open- and closed-source, as well as workflow- and agent-based methods. For each system, we collect its seed patches and use REFINE to further refine them, reporting the resolution resolving rate before and after refinement.

Overall Enhancement. The overall effectiveness of REFINE in enhancing existing APR tools is presented in Table 1. In this table, the "APR" column lists the baseline APR tool names, "Initial" shows the issue resolution rate achieved by the original tools, and "Post refinement" reports the resolution rate after applying REFINE's refinement process. The "Increase" column quantifies the improvement brought by REFINE. From the results, we observe that integrating REFINE leads to consistent improvements across all baseline tools, with resolution rates increasing by up to 20% and 14% on average. This demonstrates REFINE's ability to robustly enhance the bug-fixing performance of various APR methods, regardless of their original effectiveness.

6.2 RQ2. Results

Setup. We next evaluate the overall effectiveness of REFINE on the SWE-bench benchmark, using AutoCodeRover as the seed APR system due to its open-source availability, and compare the results against a range of baselines.

Bug-Fix Correctness. Table 3 shows the performance of REFINE against state-of-the-art APR baselines on SWE-Bench Lite & Verified. REFINE successfully resolves 155 issues (51.67%), outperforming all baselines with a 14.67 percentage point improvement over its initial seed. This increase, which accounts for 44 additional resolved issues, highlights the effectiveness of REFINE in refining patches leveraging issue and code semantics. We further validate this on SWE-Bench Verified, where REFINE refines patches from AutoCodeRover to improve the resolution rate from 51.6% to 63.8% - with an absolute increase of 12.2 percentage points and correctly resolving an additional 61 issues.

Bug Localization. The bug localization performance of REFINE is driven by its patch refiner. This component detects when an initial localization is fundamentally incorrect and uses that information to guide a more precise attempt in a subsequent round if applicable.

Cost Analysis. On average, our approach requires 22.4 minutes to solve an issue, with a cost of $6.59 and 1.15 million tokens per issue under the parameters described in the §5. These costs can be reduced to $4.77 per issue across multiple runs when utilizing issue semantics and repair stage caching. Notably, the median cost per issue is $4.87, which is substantially lower than the average due to a small number of costly outliers—such as the `scikit-14087` GitHub repository (over $30)—that skew the average cost.

6.3 RQ3. Results

Ablation Study. Table 2 presents the results of an ablation study evaluating the impact of each module within our approach on both the resolved issue rate and accuracy drop. The results clearly demonstrate that the inclusion of each module contributes to improved bug-fixing performance. The baseline configuration, which omits all three modules, achieves a resolved issue rate of 37.00% with an accuracy drop of -7.66%. Adding any two modules provides only marginal improvement. Notably, the inclusion of the context extraction and code reviewer modules without the diverse delta patch generation module yields a 40.33% resolved rate and reduces the accuracy drop to -4.33%. Introducing the diverse delta patch generation module further boosts performance, with the combination

Table 3: Comparison of Bug-Fixing Accuracy, Cost, and Patch Location Correctness Between REFINE and Baselines

Approach	LLM	Correctness		% Correct Location	Cost	
		SWE-bench Lite	SWE-bench Verified	File	Avg. Cost ($)	Avg. Tokens
Moatless Tools	Claude 3.5 Sonnet	38.30% (115)	N/A	N/A	0.17	N/A
Codart AI	Claude 3.5 Sonnet	41.67% (125)	N/A	N/A	N/A	N/A
Openhands	CodeAct v2.1	41.67% (125)	53.00% (265)	N/A	2.14	N/A
Lingxi	N/A	42.67% (126)	N/A	N/A	N/A	N/A
CodeStory Aide	N/A	43.00% (129)	62.20% (311)	N/A	N/A	N/A
DARS Agent	Claude 3.5 Sonnet	47.00% (141)	N/A	N/A	12.24	N/A
devlo	Claude 3.5 Sonnet	47.33% (142)	N/A	N/A	N/A	N/A
SWE-agent	Claude 3.7 Sonnet	48.00% (144)	62.40% (312)	N/A	1.62	521k
Blackbox AI Agent	Claude 3.5 Sonnet	49.00% (147)	62.80% (314)	75.00%	N/A	N/A
Globant Code Fixer Agent	N/A	48.33% (145)	N/A	N/A	N/A	N/A
Gru	N/A	48.67% (146)	57% (285)	N/A	N/A	N/A
Codev	Claude 3.5 Haiku + Gemini 2.5 Pro + o4-mini	49.00% (147)	N/A	79.00%	N/A	N/A
ExpeRepair-v1.0	Claude 3.5 Sonnet + o3-mini	48.33% (145)	N/A	81.00%	2.07	N/A
AutoCodeRover	N/A	37.00% (111)	51.60% (258)	N/A	N/A	N/A
Agentless-1.5	Claude 3.5 Sonnet	40.67% (122)	50.80% (254)	76.67%	1.12	N/A
OrcaLoca + Agentless-1.5	Claude 3.5 Sonnet	41.00% (123)	N/A	N/A	N/A	N/A
REFINE	Claude 3.7 Sonnet + Gemini 2.5 Pro	51.67% (155)	63.8% (319)	85.67%	6.59	1.15M

Table 4: Performance with different LLM backends.

Module			Resolved	Acc Inc
Ctx.	Div Delta	Reviewer	Issue Rate	
-	-	-	37%	-
C-3.7	C-3.7	C-3.7	44.66%	7.66%
C-3.7	C-3.7	C-4	46.00%	9.00%
C-3.7	C-3.7	C-3.7+C-4+G-2.5	46.33%	9.33%
G-2.5	G-2.5	G-2.5	50.33%	13.33%
C-3.7	C-3.7	G-2.5	51.67%	14.67%

Table 5: Performance under different hyperparameters.

Hyperparameters	Setting	Resolved Issue Rate
# Retry Loops	1 Retry (Temp 0.7)	49.00% (147)
	3 Retries (Temp 0.7)	50.00% (150)
	5 Retries (Temp 0.7)	51.67% (155)
Temperature	Temp 0.0 (5 Retries)	48.33% (145)
	Temp 0.3 (5 Retries)	49.33% (148)
	Temp 0.7 (5 Retries)	51.67% (155)

of all three modules achieving the highest resolved issue rate of 44.66% and eliminating the observed accuracy drop. These results highlight the complementary effect of the three modules, particularly the importance of integrating both context extraction and code review for maximizing patch correctness and reliability.

Backend LLMs Impact. Table 4 summarizes the impact of using different LLM backends in each module of the REFINE framework. The baseline configuration, which is the original APR tool without the refinement process (i.e. AutoCodeRover), achieves a resolved issue rate of 37%. Replacing all three modules with Claude 3.7 Sonnet (C-3.7) significantly increases the resolved issue rate to 44.66% with an accuracy improvement of 7.66%. Switching the backend of all modules to Gemini 2.5 Pro (G-2.5) further improves performance, yielding a resolved issue rate of 50.33% and an accuracy increase of 13.33%. The best performance is achieved when the context extraction and delta generation modules use C-3.7, while the reviewer module uses G-2.5, resulting in a resolved issue rate of 51.67%. These results highlight the importance of strong LLM backends and module-specific backend selection, with heterogeneous configurations achieving the best correctness and accuracy.

Hyperparameters Impact. Table 5 presents the performance of REFINE under different hyperparameter settings, specifically varying the number of retry loops and the temperature parameter. The results show that increasing the number of retry loops consistently improves the resolved issue rate: from 49.00% with 1 retry, to 50.00% with 3 retries, and reaching 51.67% with 5 retries (all at a temperature of 0.7). Similarly, varying the temperature parameter while keeping the number of retries fixed at 5 also impacts performance. A temperature of 0.0 yields a resolved issue rate of 48.33%, while increasing the temperature to 0.3 improves the rate to 49.33%, and a temperature of 0.7 achieves the highest rate of 51.67%. These results suggest that more retries and higher temperatures improve patch refinement by increasing exploration and patch diversity.

7 Conclusion

We present REFINE, a context-aware patch refinement framework that significantly enhances automated program repair. REFINE improves AutoCodeRover's performance by 14.67% on SWEBench Lite and raises the resolution rate on SWEBench Verified by 12.2%. When applied across multiple APR systems, REFINE yields consistent gains—improving resolution rates by an average of 14%, demonstrating its broad effectiveness and generalizability in refining patches.

Limitations

Our reported results may have the following limitation, we adress these limitation through the following efforts.

External Validity. Our evaluation focuses on the SWE-Bench Lite benchmark, which—while representative—may not fully capture the diversity of bugs and codebases encountered in the wild. To minimize this threat we also report result on SWE-Bench Verified benchmark. In particular, the GitHub issues in SWE-Bench are often well-structured and accompanied by relevant test cases, whereas real-world bug reports may be noisier, less detailed, or lack reproducible test environments. The effectiveness of REFINE may vary when applied to such settings. Our formulation of DRAFT PATCHES—including INCOMPLETE and OVERFITTED patches—is grounded in our empirical observations and aligns with known limitations of LLM-based APR systems. However, other categories of failure modes may exist that we do not explicitly model. Additionally, our reliance on regression test outcomes and LLM-based voting for validation introduces potential biases in defining what constitutes a "correct" patch.

Internal Validity. REFINE depends on several heuristic decisions (e.g., context extraction strategies, delta patch sampling procedures, aggregation rules) that could influence outcomes. While our design aims to generalize across different agents and workflows, we do not perform ablation studies on every component, which could limit interpretability of their individual contributions. Our implementation uses specific LLMs (Claude 3.7 and Gemini 2.5 Pro) as underlying agents. Different models may produce qualitatively different behavior, especially in context interpretation and code generation. Our results may therefore not directly generalize to other LLMs or future model versions with different capabilities.

Scalability and Performance. Although REFINE is designed to work on repository-level repair tasks, its performance may degrade with very large codebases due to context window limits and increased computational cost of test-time scaling. Future work is needed to assess scalability across industrial-scale systems.

Ethics Statement

This work studies the use of large language models for APR and proposes a refinement framework to improve patch correctness and robustness. Our goal is to enhance the reliability of AI-assisted software engineering by reducing incomplete, overfitted, or semantically incorrect patches generated by existing APR systems. We evaluate our approach exclusively on publicly available benchmarks and open-source repositories, following the SWE-bench evaluation protocol without accessing private developer data or external proprietary resources.

Although our framework is designed to improve software quality, automated code generation and repair systems may still produce incorrect or unintended patches. Such systems should therefore be used as developer-assistance tools rather than fully autonomous replacements for human review.

Our approach also relies on commercial LLMs, which may inherit biases, limitations, or nondeterministic behaviors from their training data and underlying architectures. Model outputs may vary across versions and providers, potentially affecting reproducibility and fairness of evaluation. In addition, the computational cost of repository-level reasoning may increase energy consumption and environmental impact when scaled to large software systems. We therefore encourage future research on more efficient, transparent, and reproducible repair systems, as well as stronger safeguards for validating AI-generated code.

References

Vaibhav Aggarwal, Ojasv Kamal, Abhinav Japesh, Zhijing Jin, and Bernhard Schölkopf. 2025. Dars: Dynamic action re-sampling to enhance coding agent performance by adaptive tree traversal. *arXiv preprint arXiv:2503.14269*.

Jamal Al-Karaki, Muhammad Al-Zafar Khan, and Marwan Omar. 2024. Exploring llms for malware detection: Review, framework design, and countermeasure approaches. *arXiv preprint arXiv:2409.07587*.

Antonis Antoniades, Kexun Zhang Albert Örwall, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement.

Blackbox AI. 2024. Blackbox - ai code generation. https://www.blackbox.ai/.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Simin Chen, Soroush Bateni, Sampath Grandhi, Xiaodi Li, Cong Liu, and Wei Yang. 2020. Denas: automated rule generation by knowledge extraction from neural networks. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 813–825.

Simin Chen, Xiaoning Feng, Xiaohong Han, Cong Liu, and Wei Yang. 2024a. Ppm: Automated generation of diverse programming problems for benchmarking code generation models. *Proceedings of the ACM on Software Engineering*, 1(FSE):1194–1215.

Simin Chen, Zexin Li, Wei Yang, and Cong Liu. 2024b. Decix: Explain deep learning based code generation applications. *Proceedings of the ACM on Software Engineering*, 1(FSE):2424–2446.

Simin Chen, Pranav Pusarla, and Baishakhi Ray. 2025. Dynamic benchmarking of reasoning capabilities in code large language models under data contamination. *arXiv preprint arXiv:2503.04149*.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024c. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576.

Codart AI. 2025. Codart AI: Multi-agent system. https://srics96.github.io/Codart-AI/. Reported performance of 41.67 on the SWE-bench Lite leaderboard (https://www.swebench.com/index.html)). Accessed May 2025.

Codestory AI Developers. 2025. Aide: Codestory ai code assistant. https://github.com/codestoryai/aide.

devlo.ai. 2024. devlo - your ai software engineer. https://www.devlo.ai/.

Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems*, 37:60275–60308.

Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. 2025. Patch correctness assessment: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–50.

Xiang Gao, Yannic Noller, and Abhik Roychoudhury. 2022. Program repair. *Preprint*, arXiv:2211.12787.

Globant. 2024. Globant code fixer agent. https://ai.globant.com/wp-content/uploads/2024/11/Whitepaper-Globant-Code-Fixer-Agent.pdf.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Hamed Jelodar, Samita Bai, Parisa Hamedi, Hesamodin Mohammadian, Roozbeh Razavi-Far, and Ali Ghorbani. 2025. Large language model (llm) for software security: Code analysis, malware analysis, reverse engineering. *arXiv preprint arXiv:2504.07137*.

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*.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. *Preprint*, arXiv:2209.11515.

Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '19, page 314–325. ACM.

Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2024. Hybrid automated program repair by combining large language models and program analysis. *ACM Transactions on Software Engineering and Methodology*.

Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025. Patchpilot: A stable and cost-efficient agentic patching framework. *arXiv preprint arXiv:2502.02747*.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating code review activities by large-scale pre-training. *Preprint*, arXiv:2203.09095.

Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. 2024. Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2448–2449.

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 101–114, New York, NY, USA. Association for Computing Machinery.

Alex Mathai, Chenxi Huang, Petros Maniatis, Aleksandr Nogikh, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. 2024. Kgym: A platform and dataset to benchmark large language models on linux kernel crash resolution. *Advances in Neural Information Processing Systems*, 37:78053–78078.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701.

Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. 2025. Experepair: Dual-memory enhanced llm-based repository-level program repair.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 772–781. IEEE Press.

OpenHands CodeAct 2.1. 2024. An open, state-of-the-art software development agent.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

The Astropy Project. 2011. Astropy. https://github.com/astropy/astropy.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. 2025. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *Preprint*, arXiv:2504.08703.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. Specrover: Code intent extraction via llms. *Preprint*, arXiv:2408.02232.

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971.

Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 532–543.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2024. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959*.

Haoye Tian, Li Li, Hongyu Zhang, John Grundy, Jacques Klein, Tegawendé F Bissyandé, and Bach Le. Patch overfitting in program repair: A survey.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

Jiayuan Zhou Yang, Xu, 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*.

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. volume 37, pages 50528–50652.

He Ye and Martin Monperrus. 2024. Iter: Iterative neural repair for multi-location patches. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13.

Zhongming Yu, Hejia Zhang, Hanxian Huang Yujie Zhao, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. Orcaloca: An llm agent framework for software issue localization.

Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24(1):33–67.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *Preprint*, arXiv:2504.02605.

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*, pages 1592–1604.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, and Ziyuan Nan et al. 2025. Codev: Empowering llms with hdl generation through multi-level summarization. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. *CoRR*, abs/2106.08253.

A Appendix

A.1 Examples

Issue Context Examples. For the issue illustrated in Figure 2, the corresponding issue context is as follows: the **target** (T) is the QDP parser's command matching logic; the **logic** (L) involves transforming case-sensitive command recognition into case-insensitive behavior; the **actions** (A) include identifying all locations where command strings are compared and updating them to use case-insensitive methods (e.g., normalization or tolerant matching); the **constraints** (C) require preserving the parser's existing behavior for uppercase inputs and avoiding crashes on lowercase inputs; and the **generalization** (G) specifies that the fix should apply to all QDP commands, not just the specific instance of READ SERR.

Code Context Examples. For example, in the QDP parser issue described in Figure 2, the candidate patch modifies the '_line_type' function responsible for identifying command lines. The data dependencies include the regular expression '_command_re', which encodes the expected command syntax, and the input lines being matched against it. The control dependencies span the call chain from 'Table.read', through 'QDP.read', etc. The invariant constraints require that lowercase commands such as `read serr 1 2` be recognized as valid without breaking compatibility with existing uppercase-only behavior, and without introducing regressions or crashes elsewhere in the parsing process.

A.2 Baselines

Workflow-based Baselines.

- **`AutoCodeRover`** (**Zhang et al., 2024**)**:** A repair pipeline combining AST-aware code search, spectrum-based fault localization, and iterative patch generation.
- **`Agentless-1.5`** (**Xia et al., 2024**)**:** Emphasizes modular and sequential processing of search, edit, and test stages without integrated agentic reasoning.
- **`SpecRover`** (**Ruan et al., 2024**)**:** Improves upon AutoCodeRover by using developer intent signals for enhanced fault localization.
- **`ExpeRepair-v1.0`** (**Mu et al., 2025**)**:** Leverages past repair experiences and examples to generate more effective patches.

- **`OrcaLoca + Agentless-1.5`** (Yu et al., 2025; Xia et al., 2024): Extends the `Agentless` framework with the `OrcaLoca` module for improved fault localization.

Agent-Based Baselines.

- **`SWE-agent`** (Yang et al., 2024): Employs a ReAct-style loop to iteratively interact with a sandboxed coding environment.

- **`OpenHands`** (Wang et al., 2025): A generalist agent framework that executes shell commands and modifies codebases to complete tasks.

- **`Moatless Tools / SWE-Search`** (Antoniades et al., 2024): Applies Monte Carlo Tree Search (MCTS) to systematically explore the solution space.

- **`DARS Agent`** (Aggarwal et al., 2025): Utilizes Dynamic Action Re-Sampling to recover from suboptimal decisions by exploring alternative actions.

- **`devlo`** (devlo.ai, 2024): An AI developer agent designed to autonomously resolve GitHub issues.

- **`Blackbox AI Agent`** (Blackbox AI, 2024): A closed-source agent for automated coding assistance and bug-fixing.

- **`Globant Code Fixer Agent`** (Globant, 2024): An autonomous agent designed to identify, analyze, and automatically fix bugs.

- **`CodeV`** (Zhao et al., 2025): A multi-agent framework designed for collaborative code generation and repair.

- **`Codart AI`** (Codart AI, 2025): A multi-agent system designed for automated software engineering tasks combining specialized agents with advanced code analysis tools.

- **`CodeStory Aide`** (Developers, 2025): Leverages the history and context of code changes to inform the repair process.

- **`Lingxi`** (Yang et al., 2025): An open-source, multi-agent framework that coordinates specialized agents within a graph-based workflow.

A.3 Evaluation Metrics

Correctness Metrics. To evaluate the correctness of our method in addressing real-world bugs, we adopt the *Resolved Issue Rate* (Zhang et al., 2024) as our evaluation metric, following prior work. The *Resolved Issue Rate* is defined as the percentage of patches that successfully pass all hidden tests.

Bug Localization Metrics. In addition to the *Resolved Issue Rate*, we also report the *Correct Location Rate* in accordance with established practice. The *Correct Location Rate* measures the percentage of problems for which the patch generated by the tool includes the edit locations present in the ground-truth developer patch. Consistent with previous work, we evaluate this metric at file granularity. A patch is considered to contain the correct location if it modifies a superset of all the locations specified in the ground-truth patch.

Cost Metrics. Besides the correctness evaluation, we also consider two cost metrics. The first metric is *Average Cost*, which measures the average monetary cost of running the tool. The second metric is *Average Tokens*, for which we report both the input and output tokens required by our tool to generate a patch for one GitHub issue.

A.4 Implementation Details

Refine requires an existing APR tool to generate the initial patch, which serves as the seed for further refinement. In our experiments on SWEBench Verified & Lite §6.2, we use AutoCodeRover(Zhang et al., 2024) as the default underlying APR approach for seed patch generation, due to its strong performance, extensibility and open-source nature. In §6.1, we replace the seed generation module with other existing APR tools to investigate how Refine can enhance different APR approaches. For AutoCodeRover configuration, we set the number of agent conversation rounds during localization and API extraction to 15. Empirically, fewer than 8 rounds often result in incomplete localization. Since the localization process can terminate early if the LLM confirms that the correct file has been localized, this cap helps avoid unnecessary LLM calls.

Refine is implemented using `Gemini 2.5 Pro` as the reviewer agent for delta patch aggregation. In §6.3, we further evaluate the impact of different backend LLMs on the overall performance of Refine.

By default, Refine employs greedy sampling

with a temperature set to zero for deterministic results during delta patch generation. For more diverse generation, we set the temperature to 0.7 and run up to 5 retry loops per issue, terminating early if a suitable patch is found. We evaluate the impact of both the *temperature* and the *number of retry loops* in §6.3 to understand how these hyperparameters influence the performance of REFINE.

To evaluate the correctness of each APR tool in fixing real-world GitHub issues, we strictly follow the SWE-bench benchmark guidelines. In this setting, the inputs include a user-submitted issue description (typically in natural language), the complete codebase of the repository, and a set of public test cases (usually from regression tests). The goal is to automatically generate a correct patch by reasoning over the entire codebase. The correctness of each generated patch is then assessed using private, rigorous test suites to ensure comprehensive validation. In accordance with the SWE-bench protocol, each tool is allowed a single *pass1* attempt, without access to test-specific metadata, hints, or external web resources. All generated patches must be self-contained and executable within the provided context.

A.5 Modifications To The Reproducer

SpecRover contains a Reproducer Agent that takes in the issue description and generates a standalone python file that when executed in the repository reproduces the issue. The generated reproducer is determined to be successful if it raises an Assertion Error and exits with a non zero code with the prompt generating the reproducer specifically mentioning these conditions. In our experiments we have noticed that this system exit sys.exit() usually in a caught error can return stack traces that aren't helpful in localizing the faulty issues. We first improve the feedback prompt to specifically mention if a reproducers stack trace doesn't have an assertion error or exits with a zero code which significantly speeds up how many rounds it takes to generate a successful reproducer.

Next we add another component that takes in the best possible reproducer and runs it through an execution feedback loop with the goal of refining the reproducer and modifying it generate a more useful stack trace that can better help with issue localization. The feedback loop consists of passing the reproducer, the stack trace and the issue to the LLM and prompt it using a custom prompt that focuses on refining the reproducer to - (1) better reproduce the issue and (2) to remove potential try catch, system exit codes and assertion errors and generate a reproducer that gives the most detailed stack trace errors and incorporate an LLM call that decides if the stack trace is sufficiently useful or not.

We find that this step significantly improves the quality of stack trace errors produced and constitutes the first step of our execution semantics.

A.6 Identification of Suspicious Files From the Execution Trace

Once we have this refined reproducer, we run it and process the execution trace to identify all the relevant files that were called in the reverse order of contact i.e, the first file in the trace is the file that was last used before the crash occurred. We then identify all the files in the repository and store them in a set. Next we filter out the files obtained from the execution trace an only include those files that are both in the execution trace as well as in the repository, this gives us an ordered list of suspicious files that were last used when the crash happened.

We then take the top 7 files and give this to the localization agent along with a prompt explaining why they are relevant. This is the second step in the execution semantics. The final step is running Spectrum-based Fault Localization and obtaining the top 5 most suspicious functions along with their class and file and sending this to the localization agent with a suitable prompt that similarly explains why these files are useful and how they were obtained.

A.7 Repair Stage Caching

When the repair stage is executed for a given issue (I) on a buggy file (F) using an initial patch (P), it produces a set of candidate patches (p_i). We store these results in a cache, using the tuple (I, F, P) as the key and the resulting patches (p_i) as the value. This caching mechanism helps avoid redundant repair runs for previously encountered inputs, leading to significant cost savings. The cache is intended to be used primarily when running powerful models with temperature set to 0, to minimize variance. Use of the cache is **entirely optional** and was solely designed to help reduce cost.

A.8 Prompts Used:

Generic System Prompt for Issue Semantics, Workflow generation, Context Retrieval and Fix Generation

You are a software developer working on fixing an issue in your code base. Another software developer has looked into this issue and has given a fix that solves the main issue or solves it to the best of the developers ability. Now, your goal is to analyze the possible call chains in a flle, go through and analyze them and determine if any additional changes need to be made in addition to the main change to maintain consistency in the code base while also fixing any edge cases or existing functionality that might have broken.

Prompt To Generate Issue Semantics

The issue might show a very specific example failing or not clearly explain the intent of the developer, your goal is to understand the exact intent of what the issue conveys, and explain if this issue is part of a broader issue and if it is generalizable, and explain what the code should reflect to solve this issue as a whole, then summarize everything you've said. Finally end it by by giving a paragraph that would direct an AI agent to fix changes that need to be made in the code base, be as general as possible. The given issue you need to generalize is: "{issue statement}". The final paragraph that directs an AI agent should be in a $< directions >$ ... $< /directions >$ block.

Prompt To Generate Workflows

You are an AI assistant trying to solve an issue: {issue semantics}, you have correctly figured out the main buggy file and fixed it using the patch: "{initial patch}" you however miss other changes required to maintain the consistency of the fix throughout the file. the content of the file: "{buggy file name}" is: "{code in buggy file}". Your task is to figure out the different flows of execution of the file along with the names of the methods called. We will then look at each execution flow and determine if there needs to be other fixes in addition to the main fix to maintain consistency.

Each flow must be in the following format:

$< flow >$

$< step >$ describe the approach...$< /step >$

$< step >$ ... $< /step >$
...
$< /flow >$

$< flow >$
$< step >$ ... $< /step >$
$< step >$ ... $< /step >$
...
$< /flow >$

...

Each flow much be encased in ¡flow¿ ¡/flow¿ and each step must be in ¡step¿ ¡/step¿.

Prompt To Get Relevant Context For A Step

You are analyzing a call chain, and are given steps that describe steps in a flow such as '_read_table_qdp()' calls '_get_tables_from_qdp_file()', along with the entire code in the file that the call chain resides in. Your goal is to understand the relevant code snippets that need to be extracted in order to analyze this step and solve an issue. Your output should be a JSON with the key describing the code snippet, and the value being the code in the code snippet. NOTE: If you are giving the code of a method then you must give the ENTIRE code in the method. Also note that the key and values of the JSON must be strings. For example, for the '_read_table_qdp()' calls '_get_tables_from_qdp_file()' step, the key should be something like "code in the _get_tables_from_qdp_file() method", and the value should be the code in the _get_tables_from_qdp_file method. the resulting JSON structure should look like: "code in the _get_tables_from_qdp_file() method": Get all tables from a QDP file. Parameters .... (rest of the _get_tables_from_qdp_file method code) The file_content is: "{File Content}", and the step in the call chain is: "{Step}". Your goal is to find the relevant code snippets as mentioned above. NOTE: Try to focus the majority at a method level of granularity, however you can also use a class level or expression level of granularity. This information will be given to another AI agent who will suggest code changes to fix the bug and maintain consistency throughout the file.

Prompt To Generate Fixes

You are given an execution flow: ¡flow¿"{Flow }"¡/flow¿ and general directions on how to fix it: ¡directions¿" Issue Semantics "¡/directions¿. In this flow we focus on the step: ¡step¿""{Step}"¡/step¿ .the relevant code snippets are: ¡codesnippets¿""{Relevant Code Snippets}"¡/codesnippets¿, and another AI agent has given an initial patch: ¡patch¿""{Initial Patch}"¡/patch¿ that could potentially have solved the bulk of the issue. Your goal is to analyze these code snippets and identify changes that need to be made to maintain consistency with the general directions provided to solve the overall issue. These can be minor or major changes that try to maintain consistency in the file, fix edge cases that might have been missed, or fix modified code that might break important existing functionality. Please be EXTREMELY thorough, go through each line of context given to make sure that you have not missed anything that needs to be changed. You can give your reasoning, however your final changes must be in ¡changes¿...¡/changes¿ angle brackets, with the original code, patched code and reasoning, preferably in a ¡original¿...¡/original¿ ¡patched¿...¡/patched¿ ¡reason¿...¡/reason¿ format. If there are no changes to be made then the output must be "No changes" in the angle brackets, i.e ¡changes¿No changes¡/changes¿.

Prompt For Repair Stage Fixes Reviewer

System Prompt:

You are an expert software code reviewer who evaluates suggestions made by multiple software engineers attempting to solve a specific issue in a code file.

Your role is to:
- Understand the issue in depth.
- Analyze each suggestion carefully, identifying which are necessary, which are helpful, and which may be incorrect or unnecessary.
- Apply clear and well-reasoned judgment to select the best combination of suggestions to fully and correctly resolve the issue.

You prioritize correctness and consistency in your review. Use precise reasoning when justifying your decisions.

User Prompt:

You are reviewing and trying to solve the following issue in a code file:
< issue > "{Issue Statement}" < /issue >

The full content of the file is:

< filecontent > "{File Content}" < /filecontent >

A number of software engineers have provided suggestions, each aiming to solve the issue along with a starting fix to build on that most likely fixes the bulk of the issue:

< startingfix > "{Patch Content}" < /startingfix >

These suggestions consist of the original code, the patched code, and the reasoning for why this change was suggested. Your goal is to filter out unnecessary suggestions while keeping the useful ones.

Useful suggestions fall into one of these categories:
- Suggestions ensuring the consistency of the fix throughout the file.
- Suggestions identifying edge cases that might have been missed by the starting fix.
- Suggestions fixing code that may have been broken by the starting fix.
- Suggestions that solve the core issue if the starting patch identifies a wrong fix.

where as,

Unnecessary suggestions fall into one of these categories: - Changes that break existing functionality. Use your best judgment when deciding this.
- The addition of unnecessary try catch statements or assertions which look good in theory but are unnecessary as they might break unit testing for that particular functionality and could have already been caught else where.
- If the issue is incredibly simple to fix, then avoid overtly complex suggestions which are prone to break some existing functionality.

You will be given "{Number of Patches}" suggestions in the format of:

0 :< original > ... < /original >< patched > ... < /patched >< reason > ... < /reason >
1 :< original > ... < /original >< patched > ... < /patched >< reason > ... < /reason >

and so on, with 0: representing the 0th patch, 1: the 1st, etc...

Your output must be a JSON object in the following format:
{{
"0": {{
"reason": "Explanation of why this suggestion is necessary or not.",
"required": "Required or Not Required",
}},
"1": {{

”reason”: ”Explanation of why this suggestion is necessary or not.”,
”required”: ”Required or Not Required”,
}},
...
}}

- Each key corresponds to the suggestion ID.
- The ”reason” field must contain a concise, clear justification for why the suggestion is necessary or not.
- The ”required” field should be ‘Required‘ if the suggestion is required to solve the issue, or ‘Not Required‘ if it is not needed.
- Always begin with your reasoning in the ”reason” field, followed by the decision in the ”required” field.

The suggestions are:

”{Patches Enumerated One By One}”

If there are no suggestions then return None.
”””

A.9 How Refine Resolves Pallets-4045 - An Issue That No Other APR Could Resolve

Finally, we take a look at how Refine refines a partially correct patch from AutoCodeRover to resolve a Flask issue that **none** of the top SOTA models have been able to successfully fix similar to Astropy 14635 in our motivating example.

Issue Overview of Pallets-4045: The issue arises from Flask's support for nested Blueprints, where the dot character serves as a delimiter. To avoid ambiguity, Blueprint-related names (e.g., for endpoints, view functions, CLI groups) must not contain dots.

From Initial Draft to Complete Solution: The refinement process highlights how a basic fix can evolve into a robust solution through contextual reasoning.

Initial Patch: AutoCodeRover's patch introduces a check in the Blueprint constructor:

```
+     if "." in name:
+         raise ValueError("Blueprint
    names should not contain a dot.")
```

While this handles the main issue, it misses other cases—such as CLI groups and view function names—where dots can also cause conflicts. This patch can be thought of as a near miss draft patch since it solves the bulk of the issue but doesn't fully resolve the issue.

Refined Patch: Refine identifies relevant code contexts of the Blueprint class such as the call chain of the function or code required for CLI registration or URL routing and uses this code context to make context aware fixes that patch the missing edge cases in conjunction with the issue and initial patch. It then adds validations for CLI group names, replaces brittle `assert` statements with robust `ValueError` exceptions, and introduces logic to check view function names only when an endpoint is not specified. This progressive enhancement ensures a comprehensive and correct fix.

```
+     if "." in name:
+         raise ValueError("Blueprint
    names should
...
+     if cli_group is not _sentinel and
    cli_group is not None and "." in
    cli_group:
+         raise ValueError("Blueprint CLI
     group names should not contain a
    dot.")
...
+     if (
+         cli_resolved_group is not
    _sentinel
+         and cli_resolved_group is not
    None
+         and "." in cli_resolved_group
+     ):
+         raise ValueError("Blueprint CLI
     group names should not contain a
    dot.")
...
-     if endpoint:
-         assert "." not in endpoint, "
    Blueprint endpoints should not
    contain dots"
-     if view_func and hasattr(view_func,
     "__name__"):
-         assert (
-             "." not in view_func.
    __name__
-         ), "Blueprint view function name
     should not contain dots"
+     if endpoint is not None:
+         if "." in endpoint:
+             raise ValueError("Blueprint
     endpoints should not contain a dot
    .")
+     elif view_func and hasattr(
    view_func, "__name__") and "." in
    view_func.__name__:
+         raise ValueError(
+             "Blueprint view function
    name should not contain a dot"
+             " when an endpoint is not
```

```
    provided."
+         )
      self.record(lambda s: s.
          add_url_rule(rule, endpoint,
          view_func, **options))
```

By enforcing dot restrictions consistently across all naming conventions, the refined patch offers a reliable, complete resolution of the issue—demonstrating the power of context-aware patch refinement.