

REFINE: Enhancing Program Repair Agents through Context-Aware Patch Refinement

Anvith Pabba
Columbia University
ap4450@columbia.edu

Simin Chen
Columbia University
sc5687@columbia.edu

Alex Mathai
Columbia University
am6215@columbia.edu

Anindya Chakraborty
anindya99@gmail.com

Baishakhi Ray
Columbia University
rayb@cs.columbia.edu

Abstract

Large Language Models (LLMs) have recently shown strong potential in automatic program repair (APR), especially in repository-level settings where the goal is to generate patches based on natural language issue descriptions, large codebases, and regression tests. However, despite their promise, current LLM-based APR techniques often struggle to produce correct fixes due to limited understanding of code context and over-reliance on incomplete test suites. As a result, they frequently generate DRAFT PATCHES—partially correct patches that either incompletely address the bug or overfit to the test cases. In this work, we propose a novel patch refinement framework, REFINE, that systematically transforms DRAFT PATCHES into correct ones. REFINE addresses three key challenges: disambiguating vague issue and code context, diversifying patch candidates through test-time scaling, and aggregating partial fixes via an LLM-powered code review process. We implement REFINE as a general refinement module that can be integrated into both open-agent-based and workflow-based APR systems. Our evaluation on the SWE-Bench Lite benchmark shows that REFINE achieves state-of-the-art results among workflow-based approaches and approaches the best-known performance across all APR categories. Specifically, REFINE boosts AutoCodeRover’s performance by 14.67%, achieving a score of 51.67% and surpassing all prior baselines. On SWE-Bench Verified, REFINE improves the resolution rate by 12.2%, and when integrated across multiple APR systems, it yields an average improvement of 14%—demonstrating its broad effectiveness and generalizability. These results highlight the effectiveness of refinement as a missing component in current APR pipelines and the potential of agentic collaboration in closing the gap between near-correct and correct patches. We also open source our code here: [Link to Anonymous GitHub Repo](#).

ACM Reference Format:

Anvith Pabba, Simin Chen, Alex Mathai, Anindya Chakraborty, and Baishakhi Ray. 2018. REFINE: Enhancing Program Repair Agents through Context-Aware Patch Refinement. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXX.XXXXXX>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXX.XXXXXX>

1 Introduction

Large Language Models (LLMs) have demonstrated impressive capabilities across a wide range of software engineering tasks. Among these, automatic program repair (APR) stands out as particularly impactful, offering the potential to significantly reduce manual debugging effort, accelerate software maintenance, and enhance overall code quality. Recent advances in LLM-based APR have moved beyond isolated function-level fixes to focus on repository-level repair—a setting that more accurately reflects real-world scenarios, such as resolving user-reported bugs in GitHub repositories. Formally, the repository-level APR task is defined as follows: Given a user-submitted issue description (typically in natural language), the complete source code of the repository, and a suite of public test cases (usually regression tests), the goal is to automatically synthesize a correct patch by reasoning over the large and complex codebase. The correctness of the generated patch is then validated using the provided regression test suite [28].

There has been a recent surge in such APR agents, driven by the agentic paradigm in which Large Language Models (LLMs) are treated as autonomous decision-makers augmented with tool use. Under this framework, a growing body of work explores both *open-agent-based* [52, 59] and *workflow-based* [46, 64, 65] approaches. These methods endow LLMs with tool-augmented capabilities such as searching the codebase [52, 59], editing source files [64], executing tests [56], and more. The primary distinction lies in how these paradigms manage the workflow. *Open-agent-based approaches* delegate tool selection and execution to the LLM, allowing it to dynamically decide which tools to invoke and when, based on intermediate feedback. In contrast, *workflow-based approaches* follow a fixed, manually specified sequence of tool invocations, with no adaptive decision-making by the LLM during the repair process.

Challenges. While recent APR systems show considerable promise, they still struggle to generate high-quality patches for complex, real-world codebases. We identify two primary reasons for this limitation. First, current systems often exhibit a **limited understanding of the broader code context**, resulting in patches that address only part of the underlying issue. Second, they tend to **rely too heavily on test suites** as the sole signal of correctness. Consequently, patches that pass all tests may still be incorrect or only partially correct, as the test cases often fail to capture the full semantic intent of the issue.

Our initial study shows that these limitations frequently give rise to what we term NEAR-CORRECT PATCHES—patches that are close to being correct but ultimately fall short. We observe two recurring patterns: (1) INCOMPLETE PATCHES, which address only a subset of

the necessary changes due to shallow reasoning over the code; and (2) **OVERFITTED PATCHES**, which pass the available tests but fail to generalize, typically because they overfit to weak or underspecified test cases provided in the issue description (see Section 3 for examples). These partially correct patches are widespread across both open-agent-based and workflow-based APR systems. We collectively refer to these failure modes as **DRAFT PATCHES**—patches that fall short of correctness but contain promising partial signals. Despite their prevalence, current APR systems offer no systematic mechanisms for refining such patches. This gap motivates our work: *to develop principled strategies that can refine DRAFT PATCHES into correct and robust fixes*.

Although prior work [21, 48, 50, 62] has also observed patch overfitting in function-level program repair and proposed training specialized models to iteratively refine incorrect patches into correct ones [61], these efforts are typically tool-specific. In other words, a refinement method designed for one APR tool cannot be applied to others, which prevents leveraging optimization strategies across different approaches. This limitation becomes even more restrictive in repository-level program repair, where APR agents adopt diverse search strategies, repair paradigms, and patch generation mechanisms. To overcome this, we seek to extend the concept of patch refinement in a generalizable way using LLM agents, enabling a single framework to refine patches across heterogeneous APR tools in a black-box manner.

There are three main challenges in designing a generic patch refinement module.

- (1) *Lack of Precise Context*. Refining a **DRAFT PATCH** requires providing the LLM with rich contextual information. However, natural language issue descriptions are often vague or ambiguous, leading to misinterpretation or omission of key details. Additionally, the required code context for generating a complete patch is often unknown, reducing refinement accuracy.
- (2) *Limited Exploration of Delta Patch Space*. Existing methods frequently fail to explore the diverse space of possible delta patches, limiting the chances of discovering more effective or semantically correct refinements.
- (3) *Ineffective Selection of Refined Patches*. Even when multiple candidate refinements are produced, current approaches lack principled mechanisms for evaluating and selecting the best ones. This often results in suboptimal patches being chosen, undermining the impact of the refinement process.

To address the aforementioned challenges, we propose the following approach: (1) *Getting the Right Context*: We introduce an agent to disambiguate and enrich both the issue and code context, providing the LLM with clearer and more relevant information necessary for effective patch refinement. (2) *Diverse Delta Patch Generation*: We apply test-time scaling to generate multiple *Delta Patch* candidates, enabling broader exploration of the solution space and increasing the likelihood of discovering the correct patch. (3) *Aggregated Patch Synthesis*: We simulate the code review process by introducing a code review agent. This agent aggregates the partially correct fixes from each *Delta Patch* candidate and combines them with the original **NEAR-CORRECT PATCH**, ultimately producing a correct and comprehensive patch.

To this end, we implement a novel framework, **REFINE** that refines **DRAFT PATCHES** generated by APR systems. We first use **REFINE** to refine the patches generated by a variety of APR approaches such as **ExpeRepair**[33], **BlackBoxAI**[9], **Agentless**[56], **CodeV**[34], and **AutoCodeRover**[64] on a subset of **SWEBench lite** where our results show that **REFINE** consistently increases the resolution rate across all approaches on the benchmark. Next, we perform a full **SWEBench Lite & Verified** run with **REFINE** using **AutoCodeRover** as the seed, and compare it against a set of baseline APR tools, our results show that we improve the resolution rate of **AutoCodeRover** on **SWEBench Lite** by 14.67% and by 12.2% on **SWEBench Verified**, with **REFINE** out-performing all other baselines on **SWEBench Lite**. Finally, we conduct an ablation study to investigate the contribution of each component of **REFINE** to its bug-fixing performance. The results demonstrate that every component of **REFINE** positively contributes to correcting more **DRAFT PATCHES**.

We summarize our contribution as follows:

- (1) **Problem Novelty**. We introduce and investigate a new problem: patch refinement, which aims to iteratively transform **DRAFT PATCHES** into correct ones. Our proposed patch refinement module is designed to be general and can be seamlessly integrated to enhance both agent-based and workflow-based APR techniques.
- (2) **Technique Novelty**. We present **REFINE**, a concrete implementation of patch refinement. Specifically, **REFINE** consists of three key components: (1) two agents that get the `Issue Context` and `Code Context`, (2) a test-time scaling module that generates diverse plausible `patch deltas` to expand the search space, and (3) a code review agent that aggregates multiple `patch deltas` from the test-time scaling module to produce correct patches.
- (3) **Evaluation**. **REFINE** demonstrates state-of-the-art performance, when seeded with initial patches from **AutoCodeRover** - achieving a resolution rate of 51.67% on **SWE-Bench Lite** and 63.8% on **SWE-Bench Verified**. These results represent an absolute performance increase of 14.67 and 12.2 percentage points, respectively. Furthermore, our evaluation shows that **REFINE** is effective at refining patches from other leading Automated Program Repair (APR) systems improving their average resolve rate on **SWE-Bench Lite** by 14% showing its generalizability and effectiveness in refining patches.

2 Background & Related Work

2.1 LLM for Software Engineering

With the rise of large language models (LLMs), there has been a surge of research on leveraging LLMs for a wide range of downstream software engineering tasks, such as code generation [11, 13, 15, 20, 25], malware detection [8, 12, 27], test creation [16, 29, 47], automated code review [36], and program repair [22, 32].

Among these applications, code generation has attracted particular attention from both academia and industry, as evidenced by the emergence of AI-powered development tools like **GitHub Copilot**, **Amazon CodeWhisperer**, and **Claude Code**. Over the past decade, the landscape of code-focused LLMs has rapidly expanded, starting from early models like **PaLM** to a diverse ecosystem of models spanning a range of sizes and capabilities. Today, this includes powerful closed-source models such as **ChatGPT**, **O3**, **Claude**, and **Gemini**, as well as

open-source alternatives like StarCoder, CodeLlama, Code Gemma, and DeepSeek-Coder, among others.

Beyond code generation, LLMs have demonstrated significant potential across a spectrum of other software engineering tasks. For example, LLMs have been employed for automated test case generation [42, 54, 57, 60, 66], enabling more thorough and efficient testing processes by suggesting comprehensive and context-aware test scenarios. In the domain of code review, LLMs can automatically identify bugs, suggest improvements, and assist in enforcing coding standards, thereby streamlining the review process and improving code quality. Other applications include code summarization, documentation generation [7, 18, 49], code translation [14, 26, 38, 44, 58], and refactoring, all of which leverage LLMs’ ability to understand and manipulate natural and programming languages. Collectively, these advancements are transforming the software development lifecycle, making it more automated, efficient, and accessible.

2.2 Automatic Program Repair

The evolution of APR techniques [31] can be traced through several key paradigms: search-based, semantics-based, learning-based, and, most recently, LLM-based approaches.

Traditional APR Techniques. Early research in APR predominantly explored search-based and semantics-based techniques. Search-based methods [55] start from a faulty program and apply a predefined set of code mutations to generate candidate patches. These candidates are then validated against a suite of tests, with successful patches being those that pass all relevant test cases. Semantics-based approaches [41, 43] take a different route, formulating repair constraints derived from test-suite specifications and solving these constraints to generate patches. Despite their effectiveness, both search-based and semantics-based APR methods often face challenges in scalability and limited patch diversity.

Learning-based APR Techniques. To address these limitations, learning-based APR techniques emerged [39, 67]. Early works in this area trained neural machine translation models to predict code fixes, leveraging large corpora from code repositories and incorporating code syntax and semantics. Some studies further improved repair effectiveness by utilizing GitHub issues [30] and bug reports as additional training signals.

LLM-based APR for Repository-level Repair. Motivated by the need for repository-scale solutions, recent research has turned to LLM-driven, agent-based approaches for automated program repair at the repository level. The advent of benchmarks such as SWE-Bench and its successors [28, 40, 45, 63] has enabled systematic evaluation of these techniques by introducing realistic repository-level tasks and reliable test-based validation. These benchmarks have catalyzed extensive research into agent-based frameworks that can autonomously generate and validate patches for complex, multi-file software projects. The landscape of agent-based repository-level APR can be broadly divided into two main approaches: (1) *Open Process Agent-based Frameworks*: In these frameworks, LLMs act as agents equipped with tools to interact with the software environment [3, 53, 59]. The agent dynamically plans and executes actions—such as searching, editing, and testing—based on ongoing feedback, without being restricted to a fixed workflow. Examples include SWE-Agent [59], which provides interfaces for code navigation and execution, and OpenHands [53],

which leverages CodeAct [51] to enable a wide range of actions, including internet search. (2) *Workflow-based Frameworks* These approaches follow a predefined Search-Edit-Test pipeline to address repository-level bugs. The process typically involves localizing faulty code, proposing edits, and validating the fixes through regression tests. Representative systems include Agentless [56], which systematically guides LLMs through the repository to identify context and validate patches; AutoCodeRover [65], which integrates program analysis for precise context extraction; SpecRover [46], which adds specification generation and patch review; and Patch Pilot [35], which incorporates further patch refinement.

3 Motivation

In this section, we (i) present a motivating example from SWE-Bench and highlight common patches from other SOTA agents (§3.1), and (ii) discuss the need for a deeper understanding of issue (§3.2), and code context (§3.3).

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 1, the user-reported issue (box ❶) highlights a bug in the Astropy [1] 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 1, 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 [56], 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 [46] 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.

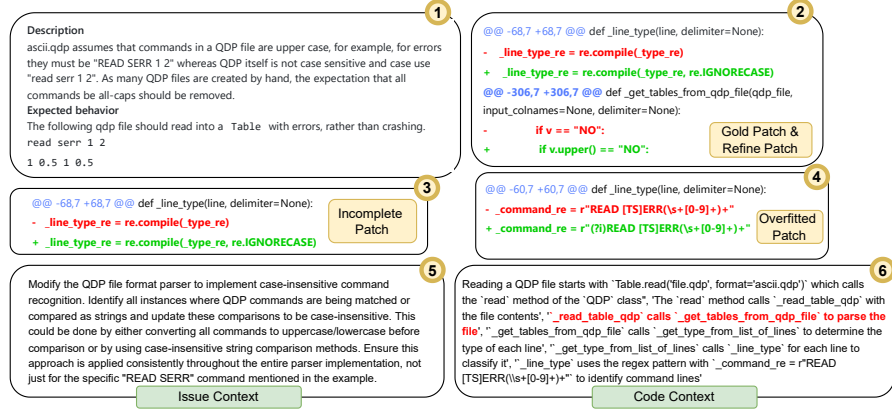


Figure 1: Motivating Example - ① is the issue description, ② is the developer patch & REFINE patch, ③ and ④ are incomplete/overfitted patches, ⑤ is the REFINE generated issue semantics, ⑥ is the REFINE generated code semantics.

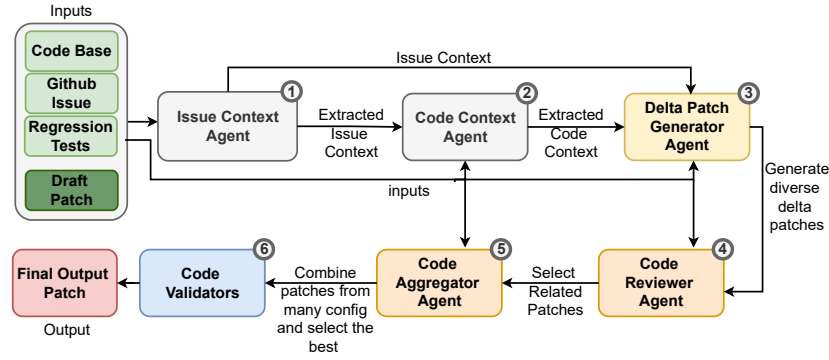


Figure 2: Pipeline of REFINe. REFINe takes as input a codebase, GitHub issue, regression tests, and an initial draft patch. First, an issue context extractor analyzes the issue description and codebase to extract relevant context ①. Then, a code context extractor uses this context and the draft patch 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.

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, generated by a program repair tool, toward a final, correct repair. Consider the standard repository-level program repair setting. Let $x = (I, \mathcal{D}, \mathcal{T})$, where x denotes a program repair instance, fully characterized by the issue statement I , the repository codebase $\mathcal{D}$, and a publicly available regression test suite $\mathcal{T}$. An existing program repair tool, denoted by $R_{\text{init}}(\cdot)$, takes x as input and generates an initial (seed) patch: $\mathcal{P}_{\text{init}} = R_{\text{init}}(x)$. 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 $F(\cdot)$, resulting in a refined, and more accurate patch: $\mathcal{P}_{\text{final}} = F(\mathcal{P}_{\text{init}})$.

4.2 Design Overview

Fig. 2 presents the design overview of REFINe. Given a program repair instance $x = (I, \mathcal{D}, \mathcal{T})$ and an initial patch $\mathcal{P}$ generated by an existing APR tool, REFINe produces a refined and more accurate patch as output. Specifically, REFINe iteratively perform the following three main steps to refine a draft patch:

- (1) *Context Extraction and Regularization*. In the first step, (① & ② of Fig. 2), REFINe leverages an LLM agent to extract and regularize context information from both the issue statement and the draft patch. This process mitigates the inherent vagueness and ambiguity often found in issue statements, 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

The issue statements collected from GitHub repositories are often vague and ambiguous, as developers frequently use informal or incomplete descriptions. Additionally, draft patches generated by existing APR tools may lack a structured and logical interpretation of the underlying code semantics. To address these limitations and provide more precise context for the LLM agent to reason about the repository, REFINE first extracts and regularizes the context from both the original issue statement and the initial draft patch.

Specifically, we begin by defining the context format for both the issue statement and the initial draft patch. We then describe how REFINE leverage a LLM agent to extract and regularize this context to support effective downstream reasoning.

4.3.1 Issue Context Format. GitHub issues often include natural language descriptions, examples, and reproduction steps, but LLMs may overfit to specific instances (e.g., block ④ in Figure 1) or fail to generalize from under-specified inputs. To mitigate this, we compute the *issue context* $I' = \mathcal{G}(I, \mathcal{D})$ by grounding the issue statement I in the structure and behavior of the codebase $\mathcal{D}$. This gives us a structured abstraction that captures the relevant intent, scope, and behavioral expectations underlying a reported software issue. It transforms unstructured natural language into actionable guidance for reasoning about and constructing valid repairs.

Formally, we represent an issue context I as a 5-tuple: $I = (T, L, A, C, G)$, where:

- **T: Target** — The system components implicated by the issue.
- **L: Logic** — The intended change in system behavior.
- **A: Actions** — High-level plans required to realize the change.
- **C: Constraints** — Conditions that must be preserved after the fix.
- **G: Generalization** — The scope of applicability beyond the specific instance.

For the issue illustrated in Figure 1, 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`.

The issue context acts as an intermediate reasoning layer between informal user-reported issues and concrete code-level repair actions. By providing a structured abstraction over the problem space, it guides patch generation in a semantically meaningful and generalizable manner.

4.3.2 Code Context Format. To generate robust patches, it is not enough to understand the issue alone—REFINE must also capture how the code behaves in relation to the issue and the proposed fix. We define the *code context* as $C' = \mathcal{S}(\mathcal{D}, p_{\text{init}})$, which models the interaction between the initial patch and the codebase $\mathcal{D}$, including call graphs, data flow, control flow, and runtime side effects. This step builds a understanding of code intent, usage, and its relevance to the bug.

To this end, we define **Patch Context** as a structured representation of the semantic environment surrounding a candidate patch. It captures the relevant data dependencies, control structure, behavioral constraints, and dynamic call relationships that determine how the patch interacts with its surroundings.

We define the patch context $C(P)$ for a patch P at location ℓ as: $C(P) = (DD_\ell, CD_\ell, IC_\ell, CG_\ell)$, where :

- **DD $_\ell$: Data Dependencies** — Variables and expressions read-/written at ℓ , including transitive flows (e.g., line content, regex matches).
- **CD $_\ell$: Control Dependencies** — Control-flow constructs affecting the execution of ℓ , such as conditional branches and loops.
- **IC $_\ell$: Invariant Constraints** — Semantic assumptions and correctness conditions that must be preserved before and after the patch (e.g., input format validity, crash avoidance).
- **CG $_\ell$: Call Graph Context** — A backward-traceable call chain showing which higher-level components transitively reach ℓ , capturing its functional embedding in the system.

For example, in the QDP parser issue described in Figure 1, 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.

The patch context serves as a semantic scaffold for reasoning about the correctness and consistency of candidate patches, enabling principled refinement and post-hoc validation of learned edits.

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 [10] and design two specialized context extraction agents using demonstration examples. Specifically, we define the extraction process as follows:

$I' = \text{Issue_Context_Extractor}(I)$, $C' = \text{Code_Context_Extractor}(\mathcal{P}_{\text{init}})$

where I' denotes the regularized context extracted from the issue statement, and 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 [11].

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 a set of diverse delta patches from §4.4, the next step is to aggregate these patches into a single, valid patch that can effectively address the bugs described in the original issue statement. To achieve this, REFINe incorporates three steps: 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 out unrelated delta patches, REFINe first performs de-duplication to remove redundant patches. Next, it groups the remaining delta patches based on

conflicts-i.e., when two patches modify the same line differently, similar to a merge conflict. To do this, all patches are passed to the LLM, which is prompted to group conflicting ones; all patches in a group along with the issue description, are then passed to the LLM for aggregation. Here aggregation works like a developer resolving a merge conflict: if patches are subsets, their changes are merged; if they diverge, the LLM selects or combines changes based on the issue description, producing one unified delta patch per group. These non-conflicting unified patches are then merged with the initial draft patch using an LLM that checks for overlaps or conflicts and produces a consolidated set of diffs.

These diffs, containing the old and new code snippets, are passed to a git-diff extraction program that locates the original code in the codebase and replaces it with the new code while handling indentation mismatches, and finally the git diff command is run to generate the final consolidated patch. We also ensure syntactic correctness during aggregation by linting the code (e.g., with pylint) to catch syntax errors whereas semantic correctness is addressed iteratively - if a patch introduces a semantic issue, it is corrected in the next iteration when that patch serves as the draft.

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

In this paper, we conduct empirical evaluations and seek to answer the following research questions, due to the space limit, more results could be found on our website.

- 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 [28], 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

To evaluate the performance of REFINe, we compare it with a comprehensive range of state-of-the-art approaches in Automated Program Repair (APR). The selected baselines encompass two primary methodologies: manually defined workflow-based frameworks

and more autonomous agentic frameworks. This comparison allows us to contextualize the contributions of REFINE and demonstrate its efficacy.

Workflow-Based Approaches. These methods adopt a structured search–edit–test sequence, with dedicated modules orchestrating each stage.

- **AutoCodeRover [64]:** A repair pipeline combining AST-aware code search, spectrum-based fault localization, and iterative patch generation.
- **Agentless-1.5 [56]:** Emphasizes modular and sequential processing of search, edit, and test stages without integrated agentic reasoning.
- **SpecRover [46]:** Improves upon AutoCodeRover by using developer intent signals for enhanced fault localization.
- **ExpeRepair-v1.0 [33]:** Leverages past repair experiences and examples to generate more effective patches.
- **OrcaLoca + Agentless-1.5 [37,56]:** Extends the Agentless framework with the OrcaLoca module for improved fault localization.

Agent-Based Approaches. These tools grant greater autonomy to the underlying models, allowing them to interact more dynamically with the development environment.

- **SWE-agent [59]:** Employs a ReAct-style loop to iteratively interact with a sandboxed coding environment.
- **OpenHands [53]:** A generalist agent framework that executes shell commands and modifies codebases to complete tasks.
- **Moatless Tools [23]:** Applies Monte Carlo Tree Search (MCTS) to systematically explore the solution space.
- **DARS Agent [6]:** Utilizes Dynamic Action Re-Sampling to recover from suboptimal decisions by exploring alternative actions.
- **dev1o [19]:** An AI developer agent designed to autonomously resolve GitHub issues.
- **Blackbox AI Agent [9]:** A closed-source agent for automated coding assistance and bug-fixing.
- **Globant Code Fixer Agent [24]:** An autonomous agent designed to identify, analyze, and automatically fix bugs.
- **CodeV [34]:** A multi-agent framework designed for collaborative code generation and repair.
- **Codart AI [17]:** Analyzes code context to generate precise, targeted patches for identified bugs.
- **CodeStory Aide [2]:** Leverages the history and context of code changes to inform the repair process.
- **Lingxi [4]:** An open-source, multi-agent framework that coordinates specialized agents within a graph-based workflow.

For all baseline approaches, we follow the protocol of prior work [56] and report their best performance metrics as published in their respective papers, official websites, blogs, or leaderboard and the seed patches produced by these APR approaches, which we use in our experiments, can be obtained from the SWEBench website [5].

5.3 Evaluation Metrics

Correctness Metrics. To evaluate the correctness of our method in addressing real-world bugs, we adopt the *Resolved Issue Rate* [64] as our evaluation metric, following prior work. The *Resolved Issue Rate*

is defined as the percentage of generated patches that successfully pass all hidden test cases.

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.

5.4 Experiment Process

RQ1. Process. To address this question, we apply REFINE on a diverse set of state-of-the-art APR approaches including AutoCodeRover, Agentless, CodeV, BlackBoxAI, and ExpeRepair - spanning closed and open-source systems, workflow and agentic-based methods and different LLMs. We then use REFINE to improve a subset of thirty initial patches generated by these methods and measure the resolution rate both with and without REFINE. These thirty issues were selected by repeatedly sampling from the full distribution until we found a subset where AutoCodeRover resolved similar number of issues in the subset (36.67%) vs the full dataset (37%), it had at least one issue from each repository, and the resulting distribution was similar to the full dataset using a Chi-square goodness-of-fit test & ensuring that the p-value exceeded 0.05 (obtained p-value was 0.0935). By comparing the results before and after applying REFINE, we can assess its effectiveness as a general-purpose plugin for enhancing current APR tools. Secondly, we perform a fine-grained analysis to identify the number of unique issues that are successfully fixed by REFINE using AutoCodeRover as the seed, but were not solved by any other SOTA on the full SWEBench dataset.

RQ2. Process. To address this research question, we evaluate the overall effectiveness of REFINE on the SWEBench benchmark when using AutoCodeRover as the seed and compare it to the performance of other approaches. For each issue, we use REFINE to generate a patch then assess its correctness using the hidden test cases, following the SWE-Bench standards. A patch is considered correct only if it passes all hidden test cases.

RQ3. Process. To answer this question, we conduct several experiments to evaluate the contribution of each module within REFINE and to investigate how its hyperparameters affect overall performance. First, we perform ablation studies by removing each core module from REFINE—specifically, the *context regularization*, *diverse delta patch generation*, and *code reviewer* modules—and then measure the bug-fixing capability of the modified system. Second, we examine the impact of different backend LLMs on the overall performance of REFINE. We evaluate the system using single backend models, such as Claude3.7-Sonnet and Gemini2.5-Pro, and further explore how varying the backend LLM specifically for the *code reviewer* module affects REFINE’s effectiveness. Finally, we assess

REFINE’s performance under various hyperparameter settings to understand their influence on the results. In particular, we investigate the *number of retry loops* used for iterative patch refinement, which balances the trade-off between cost and search space, as well as the *temperature* parameter, which controls the randomness and diversity of the LLM’s output.

5.5 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[64] 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 Claude 3.7 Sonnet as the backend LLM and 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.

6 Results

6.1 RQ1. Results

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

Table 1: The overall effectiveness of REFINE in enhancing existing APR tools performance on SWEBench Lite

APR	Initial	Post refinement	Increase
AutoCodeRover	36.67%	56.67%	20%
Codev	50%	53.33%	3.33%
ExpeRepair	40%	60%	20%
Agentless	40%	56.67%	16.67%
BlackBoxAI	50%	60%	10%

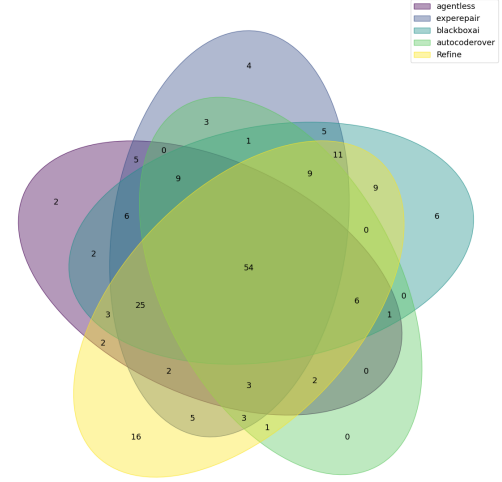


Figure 3: Fine-grained patch analysis: Venn diagram comparing the sets of issues resolved by Five APR approaches on SWEBench. REFINE (built on top of AUTOCODEROVER) resolves the largest number of unique issues (16)

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.

Fine-grained Fix Analysis. Fig. 3 presents a fine-grained comparison of fixed issues across different APR approaches. The Venn diagram illustrates overlap and uniqueness in resolved issues among five methods: four baseline APR tools as mentioned above based on their official SWEBench submissions, and REFINE when refining patches generated by AutoCodeRover. Notably, REFINE resolves 16 unique issues not fixed by any other method while also outperforming all other baselines, each of which resolves at most 6 unique issues. An interesting observation is that different base APR tools produce notably distinct sets of patches, with substantial variation in the bugs each tool can resolve. The overlaps between the sets indicate that while there is some commonality in the bugs fixed, each APR approach also contributes uniquely, capturing bugs that others may miss. Notably, the integration of our refinement module (REFINE) enables the recovery of a significant number of additional unique patches, further improving the overall coverage. These results yield two key takeaways. First, patch refinement is a highly effective strategy for boosting APR performance, not only by significantly increasing resolution rates (as shown in Table 1) but also by significantly expanding bug

coverage to fix unique issues that other tools miss (Fig. 3). Second, the success of REFINE’s refinement process highlights the critical importance of leveraging both high-level issue descriptions and fine-grained code context in resolving difficult multi-function bugs.

6.2 RQ2. Results

Bug-Fix Correctness. Table 2 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 3 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 of all three modules achieving the highest resolved issue rate of 44.66% when strictly using Claude 3.7 Sonnet 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% and the highest accuracy gain of 14.67%. Using C-3.7 for context extraction and delta generation with C-4 or with a combination of multiple reviewers (C-3.7, C-4, and G-2.5) also leads to substantial improvements, although not surpassing the mixed C-3.7 and G-2.5 setup. These results highlight not only the importance of leveraging powerful LLM backends but also the benefits of module-specific backend selection, with a heterogeneous configuration offering the greatest gains in both correctness and accuracy.

Hyperparameters Impact. Table 5 presents the performance of REFINE under different hyperparameter settings when using C-3.7 & G-2.5, 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 findings indicate that both a higher number of retry loops and an increased temperature setting can enhance the effectiveness of the patch refinement process, likely by promoting greater exploration and diversity in the generated patches.

6.4 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

Table 2: Comparison of Bug-Fixing Accuracy, Cost, and Patch Location Correctness Between REFINe and Baseline Methods

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 3: Ablation Study: Impact of Each Module on Issue Resolution Rate and Accuracy Drop. "Ctx." = Context Extraction, "Div Delta" = Diverse Delta Patch Generation, "Reviewer" = Code Reviewer.

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

Table 4: The performance of REFINe with different LLM backend

Module			Resolved Issue Rate	Acc Inc
Ctx.	Div Delta	Reviewer		
-	-	-	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 of REFINe under different hyperparameters

Hyperparameters	Approach Setting	Resolved Issue Rate
# of 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)

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.

7 Threats to Validity

Our reported results may have the following threats to validity.

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.

Construct Validity. 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.

Tooling and Model Dependency. 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.

Additionally, REFINE shares common risks associated with any APR system deployed in real-world scenarios. These include the injection of vulnerabilities i.e code that introduces new security flaws in the codebase, failure to account for edge cases, and the possibility of unintended consequences in production environments. While our approach tries to minimize errors such as unaccounted edge cases, it remains crucial that the output is thoroughly reviewed before integration without blind overtrust in Autonomy.

8 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.

References

- [1] 2025. Astropy. <https://github.com/astropy/astropy>
- [2] 2025. Codestory Aide. <https://github.com/codestoryai/aide>

- [3] 2025. Devin. <https://www.cognition.ai/blog/introducing-devin>
- [4] 2025. Lingxi. <https://github.com/nimasteryang/Lingxi>
- [5] 2025. SWE-bench website. <https://github.com/SWE-bench/experiments?tab=readme-ov-file#-viewing-logs-trajectories>
- [6] 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* (2025).
- [7] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*. 1–5.
- [8] 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* (2024).
- [9] Blackbox AI. 2024. BLACKBOX - AI Code Generation. <https://www.blackbox.ai/>
- [10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [11] 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, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Henggen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374 [cs.LG]* <https://arxiv.org/abs/2107.03374>
- [12] 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*. 813–825.
- [13] Simin Chen, Xiaoning Feng, Xiaohong Han, Cong Liu, and Wei Yang. 2024. Ppm: Automated generation of diverse programming problems for benchmarking code generation models. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1194–1215.
- [14] Simin Chen, Zexin Li, Wei Yang, and Cong Liu. 2024. DeciX: Explain Deep Learning Based Code Generation Applications. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2424–2446.
- [15] Simin Chen, Pranav Pussarla, and Baishakhi Ray. 2025. Dynamic benchmarking of reasoning capabilities in code large language models under data contamination. *arXiv preprint arXiv:2503.04149* (2025).
- [16] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.
- [17] Codart AI. 2024. Codart AI. <https://codart.ai/>
- [18] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the Effectiveness of LLM-as-a-judge for Code Generation and Summarization. *IEEE Transactions on Software Engineering* (2025).
- [19] devlo.ai. 2024. devlo - Your AI Software Engineer. <https://www.devlo.ai/>
- [20] 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 (2024), 60275–60308.
- [21] 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 (2025), 1–50.
- [22] Xiang Gao, Yannic Noller, and Abhik Roychoudhury. 2022. Program Repair. *arXiv:2211.12787 [cs.SE]* <https://arxiv.org/abs/2211.12787>
- [23] Mehdi Ghisassi. 2024. Introducing Moatless: An open-source framework for autonomous software development. <https://www.moatless.com/blog/introducing-moatless>
- [24] Globant. 2024. Globant’s Code-Fixer Agent. Official Leaderboard Submission. Information sourced from SWE-Bench leaderboard and public announcements..
- [25] 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* (2024).

- [26] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. Cotran: An Llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In *ECAI 2024*. IOS Press, 4011–4018.
- [27] Hamed Jelodar, Samita Bai, Parisa Hamed, 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* (2025).
- [28] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv:2310.06770* [cs.CL]. <https://arxiv.org/abs/2310.06770>
- [29] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *arXiv:2209.11515* [cs.SE]. <https://arxiv.org/abs/2209.11515>
- [30] 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)*. ACM, 314–325. doi:10.1145/3338906.3338935
- [31] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. doi:10.1145/3318162
- [32] 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* (2024).
- [33] Hongwei Li, Zhipeng Gao, Yuheng Tang, Wenbo Guo, Yuekang Li, Yi Liu, Gelei Deng, and Tianyu Liu. 2025. ExpeRepair: A Framework for Experience-based Program Repair. *arXiv:2402.10091* [cs.SE]. <https://arxiv.org/abs/2402.10091>
- [34] Hongwei Li, Yuheng Tang, Zhipeng Gao, and Shiqi Wang. 2024. CodeV: A Flexible Multi-Agent Framework for Automated Software Engineering. *arXiv:2405.15079* [cs.SE]. <https://arxiv.org/abs/2405.15079>
- [35] Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025. PatchPilot: A Stable and Cost-Efficient Agentic Patching Framework. *arXiv preprint arXiv:2502.02747* (2025).
- [36] 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. *arXiv:2203.09095* [cs.SE]. <https://arxiv.org/abs/2203.09095>
- [37] Yi Liu, Yuekang Li, Gelei Deng, Zhipeng Gao, Yao Wan, Tianyu Liu, Haoyu Wang, Zhouyang Jia, and Zhenyu Chen. 2024. OrcaLoca: A Fault Localization Approach with LLM-enhanced Ranking. *arXiv:2406.18370* [cs.SE]. <https://arxiv.org/abs/2406.18370>
- [38] 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*. 2448–2449.
- [39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshé 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 (Virtual Event, USA) (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. doi:10.1145/3395363.3397369
- [40] 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 (2024), 78053–78078.
- [41] 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)*. 691–701. doi:10.1145/2884781.2884807
- [42] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. Test intention guided Llm-based unit test generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 779–779.
- [43] 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 (San Francisco, CA, USA) (ICSE '13)*. IEEE Press, 772–781.
- [44] Rangeet Pan, Ali Reza Ibrahimzade, Rahul Krishna, Divya Sankar, Lambert Pouquem 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*. 1–13.
- [45] 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. *arXiv:2504.08703* [cs.SE]. <https://arxiv.org/abs/2504.08703>
- [46] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. SpecRover: Code Intent Extraction via LLMs. *arXiv:2408.02232* [cs.SE]. <https://arxiv.org/abs/2408.02232>
- [47] 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 (2024), 951–971.
- [48] 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*. 532–543.
- [49] 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* (2024).
- [50] Haoye Tian, Li Li, Hongyu Zhang, John Grundy, Jacques Klein, Tegawendé F Bissyandé, and Bach Le. [n. d.]. Patch Overfitting in Program Repair: A Survey. ([n. d.]).
- [51] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. *arXiv:2402.01030* [cs.CL]. <https://arxiv.org/abs/2402.01030>
- [52] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.
- [53] 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, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv:2407.16741* [cs.SE]. <https://arxiv.org/abs/2407.16741>
- [54] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. Hits: High-coverage Llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.
- [55] Westley Weimer, Thanh Vu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. 364–374. doi:10.1109/ICSE.2009.5070536
- [56] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. *arXiv:2407.01489* [cs.SE]. <https://arxiv.org/abs/2407.01489>
- [57] Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024. Llm4fin: Fully automating Llm-powered test case generation for fintech software acceptance testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1643–1655.
- [58] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint arXiv:2310.04951* (2023).
- [59] 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. *arXiv:2405.15793* [cs.SE]. <https://arxiv.org/abs/2405.15793>
- [60] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1607–1619.
- [61] 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*. 1–13.
- [62] 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 (2019), 33–67.
- [63] 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. *arXiv:2504.02605* [cs.SE]. <https://arxiv.org/abs/2504.02605>
- [64] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.
- [65] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. *arXiv:2404.05427* [cs.SE]. <https://arxiv.org/abs/2404.05427>
- [66] Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. 2024. LLM-based Unit Test Generation via Property Retrieval. *arXiv preprint arXiv:2410.13542* (2024).
- [67] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. Syntax Guided Neural Program Repair. *CoRR* abs/2106.08253 (2021). *arXiv:2106.08253* <https://arxiv.org/abs/2106.08253>

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009