BitsAI-Fix: LLM-Driven Approach for Automated Lint Error Resolution in Practice

Yuanpeng Li*
ByteDance
Hangzhou, China
liyuanpeng.meta@bytedance.com

Qi Long*‡
Carnegie Mellon University
Pittsburgh, United States
qilong@andrew.cmu.edu

Zhiyuan Yao‡
Zhejiang University
Hangzhou, China
yaozhiyuan@zju.edu.cn

Jian Xu†
ByteDance
Hangzhou, China
xujian.1502@bytedance.com

Lintao Xie
ByteDance
Hangzhou, China
xielintao@bytedance.com

Xu He
ByteDance
Hangzhou, China
hexu.324@bytedance.com

Lu Geng
ByteDance
Hangzhou, China
genglu.32@bytedance.com

Xin Han
ByteDance
Hangzhou, China
hanxin.hx@bytedance.com

Yueyan Chen
ByteDance
Beijing, China
chenyueyan@bytedance.com

Wenbo Duan
ByteDance
Beijing, China
duanwenbo@bytedance.com

***Abstract*—As enterprise codebases continue to grow in scale and complexity, the volume of lint errors far exceeds engineers' manual remediation capacity, leading to continuous accumulation of technical debt and hindered development efficiency. This paper presents BitsAI-Fix, an automated lint error remediation workflow based on Large Language Models (LLMs), designed to address this critical challenge in industrial-scale environments. BitsAI-Fix employs tree-sitter for context expansion and generates search-and-replace format patches through specially trained LLMs, followed by lint scan re-verification to output final remediation results. Additionally, our approach introduces an innovative progressive reinforcement learning (RL) training strategy that can automatically acquire verifiable training data during the project cold-start phase and continuously iterate the model by collecting online samples through feedback after system deployment. Furthermore, we designed a targeted rule-based reward mechanism that combines format rewards and correctness rewards while penalizing redundant modifications. We also propose a "code diff matching" methodology to continuously track online effectiveness. In production deployment at ByteDance, our solution has supported over 5,000 engineers, resolved more than 12,000 static analysis issues, achieved approximately 85% remediation accuracy, with around 1,000 weekly active adopters. This work demonstrates the practical feasibility of LLM-based code remediation solutions in enterprise environments and serves as a reference for automated code fix in large-scale industrial scenarios.**

***Index Terms*—Lint Error, Automated Program Repair, Large Language Model, Reinforcement Learning**

I. Introduction

Code fixing is an indispensable yet labor-intensive task in modern software development. As enterprise codebases at organizations such as ByteDance expand in scale and complexity, the volume of issues identified by automated Lint scans significantly surpasses engineering capacity for manual remediation. Lint Scans is the process of automatically analyzing code to detect potential bugs, style violations, and maintainability issues, and it is essential for ensuring code quality and consistency at scale. The sheer volume of warnings and errors discourages timely remediation, consequently leading to accumulated technical debt that degrades code health and impedes feature iteration. To address this scalability challenge, automated code repair has emerged as a promising solution to reduce manual effort and improve code maintenance efficiency. Automated code repair techniques primarily rely on a `"generate-and-validate"` paradigm. These approaches include search-based, template-based and constraint-based methods that generate candidate patches and validate them using test cases [1]–[3]. However, these traditional methods are commonly plagued by challenges such as low repair success rates [4].

Recent advances in LLMs have demonstrated remarkable capabilities in understanding, generating, and refactoring source code [5]–[7], as well as other code-related tasks such as code review [8]. Leveraging these capabilities, researchers have begun exploring LLM-based solutions for automated code fixing, with two primary research paradigms emerging. One line of research focuses on enhancing LLMs' intrinsic code repair capabilities through post-training optimization. Supervised fine-tuning (SFT) approaches have demonstrated effectiveness in domain-specific scenarios, with existing literature [9] showing improved SQL code error correction through targeted model adaptation. RL methods have gained particular traction, where carefully designed reward mecha-

*Equal Contribution
‡Work done during internship at ByteDance
†Corresponding Author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00288

nisms encourage LLMs to explore the generation of correct code. Notably, GRPO [10] has proven that RL with rule-based rewards can yield substantial improvements in code generation quality. In parallel, agent-based methodologies have emerged as a complementary paradigm, particularly within industrial applications [11][12]. These approaches leverage the autonomous planning capabilities of LLMs to address code error resolution through iterative, multi-step processes. Rather than relying solely on single-shot generation, agent-based systems orchestrate multiple LLM interactions to systematically analyze, plan, and execute complex code fix.

Nevertheless, existing methods encounter several challenges when addressing lint error fixing tasks. 1) Lint error repair demands high accuracy standards, as low fix rates can cause significant user disruption given the massive volume of errors encountered in practice. Current direct model approaches fail to achieve satisfactory accuracy levels [13]. While post-training methods such as SFT or RL could potentially improve accuracy, there is limited practical guidance on approach selection, training data acquisition or reward strategy design for RL-based solutions. 2) Agent-based approaches demonstrate substantially improved accuracy but suffer from prohibitive latency and cost overhead when applied to large-scale lint error scenarios, making them overly complex for this specific use case. 3) Lint error detection typically occurs during the merge request (MR) phase, where feedback acquisition remains insufficient. Users commonly resolve issues within their local IDE environments rather than through the repository interface, hindering effective feedback collection and impeding iterative system improvement and data accumulation efforts. To address these limitations, we propose an industrial-grade automated lint error fixing system with an effective model training framework. Specifically, our work makes the following three major contributions:

- **Lightweight Lint Error Fix Workflow** We propose an streamlined LLM-based workflow rather than an agent-based approach to effectively handle massive volumes of lint errors. Our solution incorporates sophisticated mechanisms including context extraction, result verification, and failure retry strategies to enhance overall repair accuracy. This approach effectively offloads repetitive repair tasks from engineers, accelerates issue resolution, and significantly improves code quality in industrial settings.
- **Progressive RL Training Strategy** We introduce a novel RL training strategy comprising progressive verifiable data collection and adaptive reward rule design. Our data collection framework operates in two progressive phases: initially cold-start data acquisition through dependency mocking and compilation minimization, followed by continuous feedback collection post-deployment for iterative enhancement. We implement evolving rule-based reward strategies specifically designed for different data acquisition methods at each phase. This progressively refined RL training approach ultimately delivers high-precision minimal patches with non-intrusive repair characteristics through iterative improvement.
- **Effective Assessment&Iteration Mechanism** We introduce a production-environment assessment mechanism that measures the true acceptance of AI-generated fixes by computing textual and semantic similarity between developer-committed patches and system-suggested repairs. This approach enables the construction of ground-truth labeled datasets from actual development workflows, facilitating continuous self-improvement of the LLM-RL integrated system through real user feedback.

To validate these contributions, we conducted a comprehensive large-scale deployment at ByteDance, our system achieves about 85% repair accuracy and has automatically resolved more than 12,000 issues to date. This significantly alleviates engineer workload in lint error handling, helps reduce technical debt related to code quality issues, and contributes to improved development efficiency and code maintainability.

II. Related Work

A. Traditional Automated Code Fix

To reduce the large amount of time software engineers spent on fixing code, researchers in the field have dedicated significant efforts to realize automated program repair [1]. For non-learning based methods, there are several distinct approaches. Firstly, the search-based methods [2][3] abstract the repair process into a search problem in a space including all possible patches. Secondly, the constraint-based approaches transform the repair process into a constraint solving problem [14][15]. Thirdly, the template-based method utilizes a predefined program fix template to generate patches [16]–[18]. Learning-based methods subsequently transformed code fix into a code generation task in text format given buggy codes. This paradigm shift began with the application of Neural Machine Translation (NMT) techniques [19], which inspired increasingly more methods to build on training deep learning models [20]–[22]. Recently, The field has experienced a revolutionary advancement with the surge of Large Language Model (LLM) technology, which has pushed the boundaries of Automated Program Repair (APR) significantly [23], demonstrating clear superiority over non-learning based methods.

B. LLM-based Code Fix

LLMs have brought revolutionary advances to code repair techniques. The LLM-based code fix process typically involves pretraining, SFT and RL. Evaluation work [24] shows consistent improvement through pretraining for vulnerability repair tasks, while supervised fine-tuning on APR data effectively enhances code fix capabilities [24][25]. For RL, rewards can be derived from similarity comparison with ground truth [26][27], compilation results [28], and unit test outcomes [29]. SWE-Bench[30]stands as a challenging benchmark with 2294 tasks from real-world software projects, requiring models to understand complex codebases and generate appropriate fixes across multiple files. To address these challenges, agent-based approaches like SWE-agent [31] employ ReAct-style loops to interact with shell commands and specialized tools, while

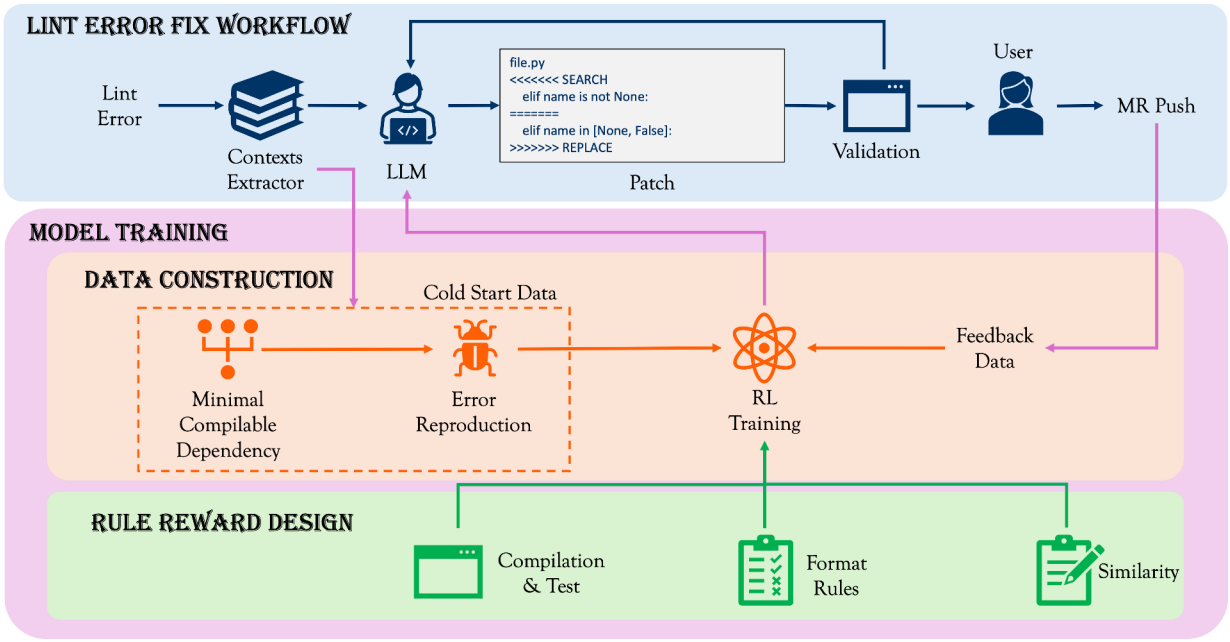

Fig. 1: Framework of BitAI-Fix, including Lint Error Fix Workflow and Model Training.

pipeline-based methods such as Agentless [32] decompose the fix workflow into a predetermined sequence of stages—bug reproduction, file localization, code editing, etc.—and solve each step in turn. Evidence suggests LLM-first approaches that allow models to reason and manipulate environments may outperform methods that insert LLMs at specific points in engineered workflows. As the field evolves, hybrid approaches combining agent-based flexibility with pipeline-based structure are emerging to tackle increasingly complex software engineering tasks.

C. Industrial Applications of Code Fix

In recent years, there has been growing interest in applying LLM for code fix in industrial settings. These applications typically focus on specific vertical scenarios to address real-world software engineering challenges. For instance, CrashFixer [33] refined the pipeline of hypothesis, patch generation and patch validation to specialize in addressing kernel crash issues and has achieved notable results in production environments. Similarly, DR.FIX [34] deployed a retrieval system on external race example database for knowledge injection, concentrating on resolving data race problems with impressive efficiency.Furthermore, the Google Team [35] has made significant strides by improving the reference retrieval system with LLM assistance in changing and validation processes, demonstrating great success in code migration at Google. These industrial applications highlight the practical value of automated code repair technologies beyond academic research, showing how theoretical advances can be translated into real-world solutions for software development teams.

III. Methodology

The overall framework of BitsAI-Fix is illustrated in Figure 1, which comprises two primary components: the Lint Error Fixing Workflow, encompassing context extraction, patch generation, and validity verification; and the RL model training module, incorporating verifiable dataset construction and targeted rule-based reward design. Each component is detailed in the following sections.

A. The Lint Error Fix Workflow

The online workflow consists of three fundamental modules: Context Extraction, Patch Generation, and Validity Validation. This architecture leverages direct model invocation to produce repair patches, circumventing the computationally intensive multi-round interactions typical of agent-based methodologies, thus rendering it suitable for large-scale lint error repair. Additionally, user-accepted fixes are incorporated as training exemplars for continuous model improvement. Each module is described below:

Context Extraction. For each lint warning (unused imports, missing error checks, style violations, etc.), we employ a two-step context extraction process. Since the location of the issue is already known in lint error scenarios, we can purposefully extract the relevant context. First, we use `tree_sitter`[1] to perform function-level expansion, identifying the minimal syntactic unit containing the issue. Second, we extract one layer of dependencies by gathering all directly referenced symbols and their definitions. Our experiments demonstrate that this approach optimally balances repair quality with

[1]https://tree-sitter.github.io/tree-sitter/

context efficiency, providing sufficient information for accurate LLM-based fixes while avoiding context overflow.

Patch Generation. We feed the extracted context snippet and original lint message into a specially trained LLM designed for code repair tasks. The model is trained to output search-and-replace–style unified diff format (see Figure 2 for details), leveraging domain-specific training to better recognize lint error patterns and produce contextually appropriate solutions. The unified diff format provides three critical benefits: (1) robustness against line number changes during code evolution, (2) minimal token overhead by targeting only modified lines, and (3) standard formatting that enables seamless automatic application upon developer approval.

```
### project/project_workflow_main_loop.go
<<<<<<< SEARCH
 errg.Go(func() error {
  if err = task.InitIDL(ctx); err != nil {
        return err
  }
  return nil
 })
=======
 errg.Go(func() error {
  defer func() {
   if r := recover(); r != nil {
     logs.CtxInfo(ctx,
                  "Recovered from panic: %v", r)
   }
  }()
  return task.InitIDL(ctx)
 })
>>>>>>> REPLACE
```

Fig. 2: Search Replace format for patch generation

Validity Verification. Each generated diff undergoes automatic validation before being presented to developers. The system applies the diff to the codebase and performs a lint scan to confirm the original issue is resolved. Failed validations trigger automatic regeneration, with up to 3 retry attempts by default. Successfully validated diffs are then presented in the pull-request interface, where developers can *accept*, *reject*, or *modify* each suggestion. Only explicitly accepted diffs are converted into actual commits.

B. The Strategy of RL Training

In this section, we present our model training methodology, providing comprehensive descriptions of our data construction pipeline and reward design mechanism. As discussed above, models trained solely through SFT demonstrate insufficient precision in code fix scenarios and fail to achieve minimal, non-intrusive modifications. Therefore, we adopt RL approaches to address these limitations.

The establishment of robust verification frameworks is essential to the efficacy of RL paradigms. In code generation and repair tasks, there has two predominant verification strategies: *(1) Text-matching verification:* this approach assumes the existence of a "golden patch" for each buggy sample. The generated patch is compared against the ground-truth label (e.g., via string similarity or edit distance) to compute a reward. While it supports fine-grained control over specific repair patterns, obtaining golden patches is labor-intensive. Notably, such reference data is inherently absent in the initial cold-start scenario. *(2) Execution-based verification:* for execution-based verification, we require that each generated patch compiles successfully and turns a previously failing test case into a passing one—a fail-to-pass validation exactly as in SWE-Bench[30]. As evidenced here, this approach obviates the need for ground-truth solutions, enabling reward computation through direct validation of the generated solution's correctness. This paradigm provides a viable pathway for reward signal acquisition in scenarios where reference answers are unavailable.

Given these considerations, we propose a progressive training paradigm: initially leveraging execution-based verification during cold-start when reference data is unavailable, then transitioning to text-matching verification as we accumulate high-quality patches from system deployment. The methodology proceeds as follows:

1) Cold-start data construction: In this phase, we primarily focus on constructing executable samples. However, rather than pursuing executable samples with test cases, we creatively approximate executability through compilability and lint tool validation, which has demonstrated remarkably effective results in practice. To ensure sample executability, one feasible approach is to directly execute the entire project. However, industrial-scale projects are typically quite large, requiring substantial time and resources for building, compilation, and execution. This leads to low efficiency in both data acquisition and training. Therefore, we designed a "semi-synthetic" data generation process that, through the following three steps, retains only the minimal executable dependencies relevant to the current problem.

- *Minimal Dependencies Construction*. For entity dependency information within the project, we leverage tree_sitter for AST parsing whenever possible to obtain "actual dependencies". For dependencies on imported third-party packages, given the potential multi-level nesting and complexity of acquiring these packages, we employ a LLM-based simulation approach to construct "virtual dependencies".
- *Issues Reproduction Test.* We focus on reproducing the original issues to establish a reliable baseline for validation. This step is primarily aimed at making the sample able to be accurately determined whether its issue has been resolved using a Fail-To-Pass test approach. Our primary goal in this step is to establish a reliable baseline for validation by accurately reproducing the original issues. We directly rerun the lint tool on the code with minimal dependencies constructed in the previous step. If the same issue is detected, the reproduction is considered successful. Otherwise, it is deemed a failure.
- *Effective Sample Selection.* We ensure the final samples meet three key criteria: accurate verification capability, reasonable difficulty distribution, and balanced knowl-

edge point coverage. Our selection process involves: (1) retaining only samples with successfully constructed minimal dependencies that compile and pass reproduction tests; (2) classifying difficulty using DeepSeek-R1 [36] by performing eight repair attempts per sample, where difficulty is determined by the number of successful repairs (samples with all eight successful attempts are excluded as too simple); and (3) selecting up to 30 samples per lint error category while maintaining balanced difficulty distribution across categories.

2) User Feedback Data Collection: Methods based on execution-based verification can achieve promising results in cold-start reinforcement learning training. However, in our approach, instead of employing test case execution, we utilize compilation and lint tool scanning as an approximation of execution-based verification. This methodology cannot truly guarantee the absolute correctness of the generated code fixes, which to some extent constrains the performance ceiling of RL training: the model tends to generate modifications that pass compilation and lint tool scanning rather than truly correct fixes. Therefore, we still need to obtain samples with "golden patches". Evidently, samples from online user feedback satisfy this requirement. In practice, following the small-scale pilot deployment of BitsAI-fix, we began collecting user feedback data and incorporating it into the RL training sample set. Over time, we continuously perform iterative refinement through this process.

3) Rule Reward Design: We follow the GRPO algorithm for RL and design rule-based rewards conditioned on error type. The GRPO algorithm has its advantage in improving the LLM's robustness when dealing with different groups of preference data. The GRPO objective is formulated as follows 1

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{\mathbf{q}\sim\mathcal{G},\,\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(O|\mathbf{q})}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\min\Bigg(r_t(\theta)\hat{A}_{i,t}, \text{clip}(r_t(\theta),1-\epsilon,1+\epsilon)\hat{A}_{i,t}\Bigg) - \beta\,\text{KL}[\pi_\theta\|\pi_{\text{ref}}]\Bigg]. \tag{1}$$

where π_θ denotes the LLM as the current policy model, $\pi_{\theta_{\text{old}}}$ is the previous policy model, and $\mathcal{G}$ includes the aforementioned three data groups. The importance sampling ratio $r_t(\theta)$ is defined as

$$r_t(\theta) = \frac{\pi_\theta(o_{i,t})}{\pi_{\theta_{\text{old}}}(o_{i,t})} \tag{2}$$

The reward is designed to be within 0 and 1 point, which is rated based on two main criteria, formatting and correctness. For formatting, the reward follows 3 to deduct points for each mismatch in a penalty manner, where C is the generated code patches, i refers to i patches not in search&replace format and j refers to j patches not applied. Since we expect one patch generation per error, the patches more than one are regarded as redundant, which is also penalized.

$$r_f(C) = \begin{cases} 0.0, & \text{if } C \text{ has correct format} \\ -0.1\cdot i - 0.1\cdot j, & \text{if } C \text{ not S\&R or not applied} \\ -0.1\cdot k, & \text{if } C \text{ has redundant k patches} \end{cases} \tag{3}$$

For correctness, the reward follows 4 and 5, where C is the generated code patches. A compilable precheck $r_p(C)$ is conducted before correctness check. If the case is not compilable, the correctness reward is zero directly. Otherwise, for cold-start data, since they do not have ground truth labeling but go through Minimal Dependencies Construction and Issues Reproduction, they are executable and thereby rewarded via fail-to-pass. For feedback data, since they have ground truth labeling, a similarity match between generated code patch and ground truth is conducted according to F_β [37].

$$r_p(C) = \begin{cases} 0.0, & \text{if } C \text{ is not compilable} \\ +0.3, & \text{if } C \text{ is compilable} \end{cases} \tag{4}$$

$$r_c(C) = \begin{cases} +0.7, & \text{if } C \text{ is cold-start data and fail-to-pass} \\ +0.7\cdot F_\beta, & \text{if } C \text{ is feedback data} \end{cases} \tag{5}$$

Finally, the total reward is calculated as 6.

$$r(C) = r_f(C) + r_p(C) + \pi[r_p(C) > 0]\cdot r_c(C) \tag{6}$$

C. The Mechanism of Assessment and Iteration

Since our approach targets code fixes within the MR scenario, the fix results are presented directly on the MR page. From a product design perspective, we initially evaluated user acceptance of our fix suggestions through click-through tracking on the "Adopt" button. However, this approach revealed a significant behavioral discrepancy: users predominantly prefer to modify code within their IDE and submit changes directly, rather than interacting with the MR interface. Consequently, this method yielded extremely limited feedback, rendering it ineffective for accurately measuring the true performance of BitsAI-Fix and insufficient to support our subsequent continuous optimization iterations.

To obtain more accurate user feedback data, we implemented a "code diff matching" methodology for evaluation. Specifically, we employ the following approach: we establish the LLM-generated diff patch as the baseline reference, then extract the actual diff after users merge or update their merge requests, and subsequently perform matching between these two artifacts. When the LLM-generated diff patch is completely encompassed within the user's actual submitted diff, we classify the fix as adopted. Conversely, any deviation or partial overlap is categorized as non-adoption.

Through training our reinforcement learning loop based on this feedback mechanism over a two-month iteration period,

we generated patches that more closely aligned with developers' authentic coding and remediation practices. Empirically, this evaluation strategy demonstrates superior persuasiveness and reliability compared to conventional copy-or-click proxies, thereby accelerating our iterative model enhancement process.

IV. Experiments

To comprehensively evaluate the effectiveness of our method, we conducted detailed offline experiments and assessments using the Go programming language as a case study, which serves as the primary programming language at ByteDance.

A. Dataset Construction

To date, we have constructed a training dataset of 20,000 samples, comprising approximately 16,000 cold start samples and 4,000 user feedback samples. These samples are evenly distributed across all 198 lint error categories, ensuring comprehensive enhancement of the model's ability to handle various lint error fix challenges. Additionally, we built a test dataset containing 2,271 samples that similarly covers all 198 lint error categories, with each category containing between 1 and 15 samples.

B. Evaluation Criteria

To validate the accuracy of repair results and ensure minimal code modifications, we adopt the following two evaluation criteria:

- **Fix Accuracy.** Defined as the proportion of model-generated patches that successfully compile and pass the original lint error scanning when applied to the problematic code. This metric serves as a key indicator for assessing repair effectiveness. The metric is formally expressed as:

$$\text{Fix Accuracy} = \frac{\sum_{i=1}^{N} \mathbb{I}(S_i)}{N} \tag{7}$$

  where N is the total number of code samples, and $\mathbb{I}(S_i)$ equals 1 if the patch for sample i is successful, and 0 otherwise.
- **Fix Redundancy.** Defined as the conciseness of the patches generated by the model. For a given problematic code, if the number of search-and-replace blocks generated by the model exceeds the number of lint errors detected, the repair is considered redundant. We measure the overall redundancy rate as the proportion of samples that result in redundant repairs, with a lower rate indicating better performance.This can be mathematically formulated as:

$$\text{Fix Redundancy} = \frac{\sum_{i=1}^{N} \mathbb{I}(P_i > E_i)}{N} \tag{8}$$

  where P_i is the number of patches generated for sample i, and E_i is the number of actual lint errors. The indicator function $\mathbb{I}(P_i > E_i)$ equals 1 if the condition $P_i > E_i$ is true, and 0 otherwise.

C. Experiment Setup

Considering effectiveness, efficiency, and internal compliance requirements, we selected Qwen2.5-Coder-32B [38] as the base model for fine-tuning. We fine-tuned it using GRPO-based RL with the verl [39] framework. The model was trained with a learning rate of 1×10^{-6}, a global batch size of 256, and a PPO mini-batch size of 128. For KL divergence control, we set the KL loss coefficient to 0.001 using a low-variance formulation. Input sequences were limited to 8,192 tokens with outputs capped at 4,096 tokens, and a rollout of 8 was employed. All training was conducted on Ascend-910B2 NPUs across 8 nodes with 16 NPUs per node.

D. Experiment Results

We conducted a systematic evaluation comparing the overall accuracy performance of models across distinct training stages, including mainstream base model, models enhanced via SFT using our dataset that was first processed with DeepSeek-R1 to fix error and remove erroneous samples, and models further refined through RL optimization. The experimental results presented in Table I support the following conclusions:

TABLE I: Performance Comparison of Base, SFT, and RL Models

	Accuracy	Redundancy	Category
DeepSeek-R1-0528	73.23%	12.15%	Base
Doubao1.6-Thinking	69.70%	20.83%	Base
DeepSeek-V3	66.18%	17.57%	Base
Qwen2.5-Coder-32B	53.76%	10.00%	Base
Qwen3-32B	51.30%	11.71%	Base
Qwen2.5-Coder-32B-SFT	65.48%	9.25%	SFT
Qwen2.5-Coder-32B-RL (Ours)	84.68%	1.72%	RL

RL Training Achieves Superior Accuracy. Post-training significantly improves base model performance on fix tasks, with both SFT and RL fine-tuning demonstrating substantial gains over the untuned Qwen2.5-Coder-32B model (53.76%). However, their practical effectiveness differs markedly. While Qwen2.5-Coder-32B-SFT increases accuracy to 65.48%, this improvement remains insufficient for real-world deployment requirements. In contrast, Qwen2.5-Coder-32B-RL achieves 84.68% accuracy through GRPO training by optimizing direct objectives and effectively leveraging reward signals to master complex code repair strategies. Remarkably, our 32B RL-trained model outperforms significantly larger models including Doubao-1.6-Thinking and DeepSeek-R1-0528 on fix tasks, demonstrating that targeted RL optimization can compensate for scale limitations while delivering substantial cost savings.

RL Training Minimizes Redundant Modifications. Untrained models are particularly prone to generating redundant modifications, ultimately resulting in intrusive code repairs that compromise code quality. This challenge persists even among sophisticated large-scale models—both Doubao-1.6-Thinking and DeepSeek-R1-0528 exhibit high redundancy rates without targeted training. Our results demonstrate that task-specific training substantially mitigates this issue: while

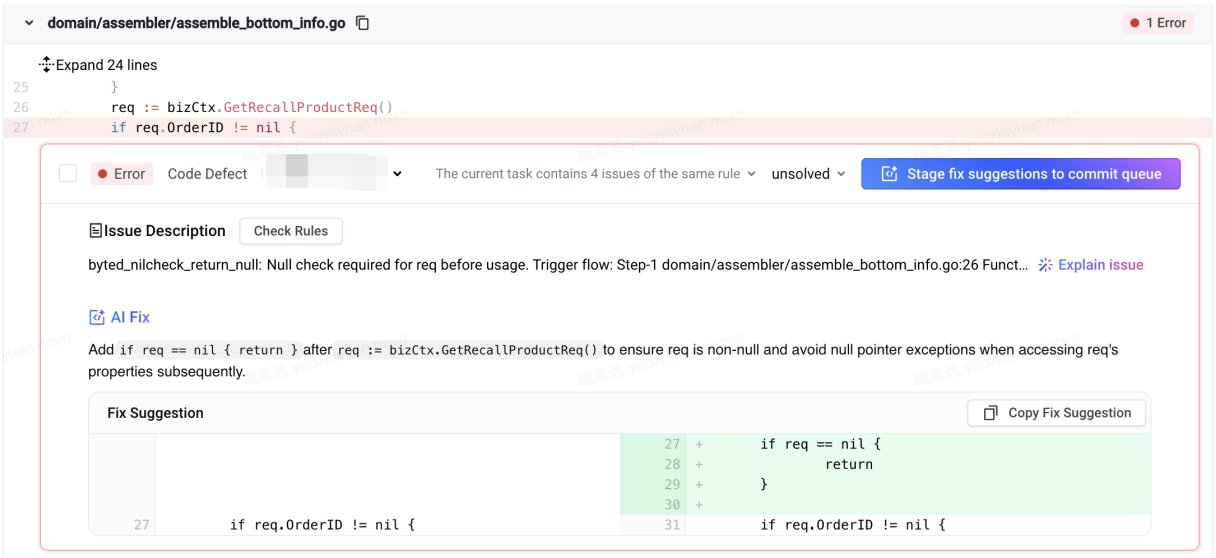

Fig. 3: Main interface of the automated code-fix tool embedded in the MR review page.

our Qwen2.5-Coder-32B-SFT model achieves relatively low redundancy compared to untrained models, Qwen2.5-Coder-32B-RL reaches an exceptionally low fix redundancy of 1.72%. This demonstrates that RL, by directly penalizing unnecessary edits, is more effective at avoiding redundant code modifications.

E. The Impact of Reward Methods

Regarding the reward design, we compared how different reward schemes affect the final outcome. Based on the comparison in Table II, we observe that the choice of reward scheme significantly affects both repair accuracy and redundancy.

TABLE II: Ablation study on reward design strategies

	Accuracy	Redundancy
Binary Pass/Fail Reward	82.78%	6.21%
Graded Reward	84.02%	7.62%
Graded Reward with Redundancy Penalty	84.68%	1.72%

Starting with a simple **Binary Pass/Fail Reward** (1 point if the issue is fixed; 0 otherwise), the model achieves 82.78% accuracy with 6.21% redundancy, serving as a solid baseline. When we introduce a **Graded Reward** system that splits into compilation and correctness-verification components (0 points for compilation failure, 0.3 points for successful compilation, 1 point for complete fix), accuracy rises to 84.02% but redundancy increases to 7.62%. This represents only a 1.24-point gain in accuracy alongside a 1.41-point rise in redundancy, suggesting that rewarding compilation success alone yields limited behavioral change without explicit penalties.

However, augmenting the **Graded Reward With Redundancy Penalty** (subtracting 0.1 points for each redundant search-and-replace block) produces markedly different results. This approach further boosts accuracy to 84.68% while sharply reducing redundancy to 1.72%. These findings demonstrate that penalizing redundant search-and-replace blocks effectively guides the model toward concise and precise fixes without sacrificing overall performance.

V. Industrial Deployment and Empirical Evaluation

This chapter presents a comprehensive evaluation of BitsAI-Fix in real-world production environments, covering the system implementation, large-scale industrial deployment, online product performance analysis, and representative case studies.

A. Implementation of BitsAI-Fix

Developers can easily enable and utilize BitsAI-Fix features as demonstrated in Figure 3. The system operates by accepting a lint error as **input**, enabling users to examine the specific rule violation details through the "View Rules" button located adjacent to the "Issue Description" tab. The system then generates two complementary **outputs**: fix recommendations that explain the rationale behind suggested changes, and concrete repair code prominently displayed on the main interface with a clear side-by-side comparison between the original and corrected versions.

Upon reviewing the system's fix results, users have two convenient options for proceeding. If they approve of the suggested corrections, they may select the **"Stage Fix Suggestion to Commit Queue"** button to seamlessly prepare the changes for subsequent version control commits. Alternatively, users can employ the **"Copy Fix Suggestion"** button to transfer the corrected code to their clipboard, allowing them to manually implement the repair within their local IDE environment. This dual-option approach accommodates different developer workflows and preferences for code integration.

B. Large-Scale Industrial Deployment

The deployment of BitsAI-Fix followed a carefully structured two-phase approach to ensure system reliability and performance optimization.

Before May: Small-Scale Pilot This phase commenced limited canary deployment to conduct initial system validation and workflow verification. It also corresponds to the cold-start phase in our progressive training approach. After training the model with cold-start data, we launched this phase to obtain the first batch of online feedback data for further model performance improvement.

After May: Full-Scale Rollout After achieving predefined performance metrics during the pilot, we expanded deployment across essentially all active repositories. The large-scale rollout enabled us to collect extensive positive and negative samples, further enriching our training dataset. This comprehensive implementation demonstrated exceptional system stability and high fix success rates in production.

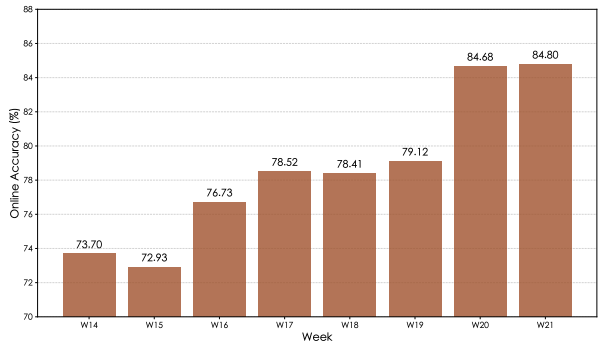

Fig. 4: Performance improvement of model optimized using user-feedback dataset across weeks

C. Online Product Performance

Online Accuracy Performance During our rapid iteration period (before May), we implemented a weekly model iteration process leveraging user feedback data. Figure 4 presents senior engineers' manual annotations of real production lint error fixes, demonstrating substantial accuracy improvements from 73.70% to 84.80%. Subsequently, we shifted focus to targeted optimizations addressing specific problem cases encountered in production environments. This significant enhancement validates the effectiveness of our approach in identifying recurring error patterns and continuously adapting to evolving developer needs.

User Adoption Metrics Since full-scale implementation, engineering teams have incorporated AI-generated code fixes for more than **12,000** distinct code issues, with participation from over **5,000** unique developers. Recent telemetry indicates sustained adoption rates exceeding **2,000** weekly fix implementations, with approximately **1,000** Weekly Active Adopters (WAA), defined as developers who implement ≥ 1 fix per week. Figure 5 presents the progressive growth trajectory of Weekly Active Adopters while also tracking the Weekly Adopter Count (WAC), representing the total number of code issues addressed each week.

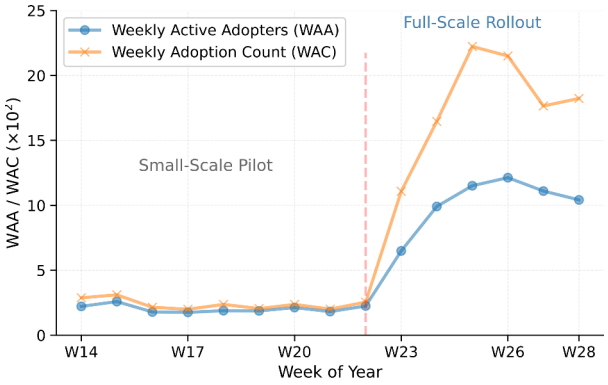

Fig. 5: Trend of Weekly Active Adopters and Weekly Adoption Count (Red line indicates Full-Scale Rollout start in May)

D. Case Study

In this subsection, we present three representative cases drawn from real-world issue repositories. Each case illustrates a typical defect pattern, the corresponding automatic fix produced by our system, and the user's feedback.

Case1: Missing recover in Goroutine In this case, the scheduler started a new goroutine to execute its main loop. However, due to the lack of a recover mechanism within the goroutine, any panic occurring in the main loop would crash the entire scheduler process, affecting system stability. To address this issue, BitsAI-Fix automatically inserted panic capture and logging logic when starting the goroutine, as shown in Figure 6. As a result, exceptions in the main loop can be promptly caught and logged without causing the scheduler process to crash, thereby improving system robustness.

Case2: Unsafe Type Assertion In this case, the *Execute* method of the finite state machine contained an unchecked type assertion. If the passed *eventCtx* is not actually of type *ShipmentContext*, the program would trigger a panic at runtime, thereby affecting system stability. To resolve this issue, BitsAI-Fix replaced the original forced type assertion with a form that includes type checking, and explicitly returns an error when the assertion fails. This way, even when types don't match, the situation can be handled gracefully through the error handling mechanism without causing exceptional crashes. The fix solution is shown in Figure 7.

Case3: Over-eager Integer-Overflow In this case, a static analysis warning regarding potential integer overflow from *strconv.Atoi* result conversion to *int16/32* triggered an automatic patch. The system replaced *strconv.Atoi* with *strconv.ParseInt* and cast the result to *int32* to address the overflow concern. However, the fix exhibited over-eagerness by modifying unrelated code regions beyond the scope of the original warning. As shown in Figure 8, the patch introduced additional changes including error message modifications and extra return statements, creating redundancy and confusion in the codebase.

```
@@ -65,7 +65,14 @@
 // Start begins the scheduler's main loop.
 func (s *SchedulerImpl) Start(ctx context.Context) {
   s.ticker = time.NewTicker(s.interval)
-  go s.run(ctx)
+  go func() {
+    defer func() {
+      if r := recover(); r != nil {
+        logs.CtxError(ctx, "Recovered from panic in SchedulerImpl.run: %v", r)
+      }
+    }()
+    s.run(ctx)
+  }()
   logs.CtxInfo(ctx, "Scheduler started with interval: %v", s.interval)
 }
```

Fig. 6: BitsAI-Fix Adding Panic Recovery Mechanism to Goroutine

```
@@ -137,7 +137,10 @@
 type FMReturnShipmentCollectionAllocatedAction struct{}

 func (a *FMReturnShipmentCollectionAllocatedAction) Execute(eventCtx EventContext) error {
-    shipmentCtx := eventCtx.(ShipmentContext)
+    shipmentCtx, ok := eventCtx.(ShipmentContext)
+    if !ok {
+        return fmt.Errorf("type assertion failed: eventCtx is not of type ShipmentContext")
+    }
     shipment := shipmentCtx.Shipment
     logger.WithContext(context.Background()).Infof(
         "Executing FMReturnShipmentCollectionAllocatedAction: %v, id: %d",
         shipmentCtx, shipment.ID)
 }
```

Fig. 7: BitsAI-Fix Converting Unsafe Type Assertion to Safe Checked Form

The developer deemed these excessive modifications unnecessary and rejected the suggestion, highlighting the importance of maintaining focused, minimal changes that directly address the identified issue without introducing unintended side effects.

Cases1 and Case2 demonstrate that targeted, minimal edits—properly scoped to the relevant code—are highly effective and readily accepted by developers. Case3 reveals the risk of over-generalizing a fix and unintentionally altering unrelated code. Based on this observation, we refined our patch-generation strategy to constrain edits strictly to the locations implicated by the reported issue.

VI. Lessons Learned and Practical Insights

In deploying BitsAI-Fix at ByteDance scale, we distilled three core lessons that can guide other organizations in building reliable, high-impact automated code-fix systems:

1) Lightweight Solutions for Lint Error Fix Scenarios: While much current research focuses on leveraging large-scale models with complex CodeAgent frameworks for code repair, our work demonstrates that for lint error scenarios—where error locations are well-defined and repair volumes are substantial—smaller LLMs combined with simplified workflows can achieve impressive results while being more practical and cost-effective. This finding offers valuable insights to the community that not all code repair tasks require heavyweight solutions.

2) Progressive Training Method for Industrial Code Fix: We developed a comprehensive RL pipeline covering the complete data lifecycle: from cold-start semi-synthetic dataset construction with executable verification, to post-deployment feedback loops, and carefully designed rule-based rewards. This approach improved our fix rate from 53.76% to 84.68% while reducing redundancy from nearly 10% to 1.72%. Both our data construction strategy and rule design methodology provide valuable references for industrial applications, demonstrating a scalable framework for deploying code fix systems in production environments.

3) Effective Evaluation Framework for Industrial Deployment: We developed an effective evaluation methodology for code fix performance that addresses real-world deployment challenges. While automated code fixes on platforms like GitHub are common, users naturally prefer performing fixes directly in their IDEs, making system effectiveness evaluation difficult. Our proposed approach using code change block matching effectively resolves this issue, providing valuable insights for industrial implementations where traditional evaluation metrics fall short due to user behavior patterns.

```
@@ -23,7 +23,9 @@
  thirdPartySticker *sticker_common.ThirdPartySticker) *sticker_common.StickerItem {
- normalImageWidth, err := strconv.Atoi(thirdPartySticker.NormalImage.Width)
+ normalImageWidth64, err := strconv.ParseInt(thirdPartySticker.NormalImage.Width, 10, 32)
+ normalImageWidth := int32(normalImageWidth64)
  if err != nil {
-  logs.CtxError(ctx, "invalid normalImageWidth", err, "Width", thirdPartySticker.NormalImage.Width)
+  logs.CtxError(ctx, "invalid normal image width", err, "Width", thirdPartySticker.NormalImage.Width)
+  return nil
  }
 normalImageHeight64, err := strconv.ParseInt(thirdPartySticker.NormalImage.Height, 10, 32)
 normalImageHeight := int32(normalImageHeight64)
```

Fig. 8: BitsAI-Fix Over-eager Patch with Excessive Modifications

VII. Conclusion and Future Work

This paper proposes an industrial-grade workflow for automatically repairing Lint errors based on large language model—BitsAI-Fix, which efficiently addresses the daily Lint scanning issues in enterprise-level codebases and significantly reduces the workload of software engineers. To tackle the limitations of existing pre-trained LLMs in this scenario, we employ reinforcement learning strategies to enhance the model's code repair capabilities. We emphasize the importance of progressive and verifiable data in model training, which includes constructing an executable and verifiable cold-start dataset as well as a continuously evolving, user-annotated feedback dataset. Different rule-based reward mechanisms are designed according to the source of the data. We validate the effectiveness of our approach in real production environments, achieving high fix accuracy, and successfully demonstrate its scalability and practical value through deployment in enterprise-scale software development setting

Although BitsAI-Fix has demonstrated satisfactory accuracy in lint error correction, the system still has certain limitations that warrant further improvement in future research.First, the current system's repair scope is primarily limited to lint errors. To build a more comprehensive code repair solution, we plan to extend the system's capabilities to a broader range of error types, thereby enhancing its applicability in real-world development scenarios.Second, the code context information provided to the model by the existing system is relatively limited, covering only single-layer function nesting structures. This limitation in context scope may adversely affect repair accuracy. To address this, we propose to introduce a dynamic context supplementation mechanism in subsequent work that adaptively adjusts the coverage of context information based on specific repair requirements.Looking ahead, with the continuous development of related technologies and deepening application practices, BitsAI-Fix and its future versions are expected to play an increasingly important role in building high-quality, highly reliable, and highly maintainable software systems, thereby enabling developers to dedicate more effort to creative core development work.

References

[1] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, Dec. 2023, ISSN: 1049-331X. DOI: 10.1145/3631974. [Online]. Available: https://doi.org/10.1145/3631974.

[2] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 298–309, ISBN: 9781450356992. DOI: 10.1145/3213846.3213871. [Online]. Available: https://doi.org/10.1145/3213846.3213871.

[3] Y. Wang, Y. Chen, B. Shen, and H. Zhong, "Crsearcher: Searching code database for repairing bugs," in *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, ser. Internetware '17, Shanghai, China: Association for Computing Machinery, 2017, ISBN: 9781450353137. DOI: 10.1145/3131704.3131720. [Online]. Available: https://doi.org/10.1145/3131704.3131720.

[4] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1482–1494. DOI: 10.1109/ICSE48619.2023.00129.

[5] X. Du, M. Liu, K. Wang, *et al.*, "Evaluating large language models in class-level code generation," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 982–994. DOI: 10.1145/3597503.3639219.

[6] B. Rozière, J. Gehring, F. Gloeckle, *et al.*, *Code llama: Open foundation models for code*, 2024. arXiv: 2308.12950 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2308.12950.

[7] E. Nijkamp, B. Pang, H. Hayashi, *et al.*, *Codegen: An open large language model for code with multi-turn program synthesis*, 2023. arXiv: 2203.13474 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2203.13474.

[8] T. Sun, J. Xu, Y. Li, *et al.*, *Bitsai-cr: Automated code review via llm in practice*, 2025. arXiv: 2501.15134 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2501.15134.

[9] Y. Duan, Y. Yu, X. Zhao, Y. Wu, and W. Liu, *Pdc & dm-sft: A road for llm sql bug-fix enhancing*, 2024. arXiv: 2411.06767 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2411.06767.

[10] Z. Shao, P. Wang, Q. Zhu, *et al.*, *Deepseekmath: Pushing the limits of mathematical reasoning in open language models*, 2024. arXiv: 2402.03300 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2402.03300.

[11] C. Lee, C. S. Xia, L. Yang, *et al.*, *A unified debugging approach via llm-based multi-agent synergy*, 2024. arXiv: 2404.17153 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2404.17153.

[12] X. Wang, Y. Chen, L. Yuan, *et al.*, *Executable code actions elicit better llm agents*, 2024. arXiv: 2402.01030 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2402.01030.

[13] H. Wen, Y. Zhu, C. Liu, X. Ren, W. Du, and M. Yan, *Fixing function-level code generation errors for foundation large language models*, 2025. arXiv: 2409.00676 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2409.00676.

[14] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 593–604, ISBN: 9781450351058. DOI: 10.1145/3106237.3106309. [Online]. Available: https://doi.org/10.1145/3106237.3106309.

[15] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 428–432. DOI: 10.1109/ICSME.2016.66.

[16] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 802–811, ISBN: 9781467330763.

[17] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224. DOI: 10.1109/SANER.2016.76.

[18] A. Koyuncu, K. Liu, T. F. Bissyandé, *et al.*, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Softw. Engg.*, vol. 25, no. 3, pp. 1980–2024, May 2020, ISSN: 1382-3256. DOI: 10.1007/s10664-019-09780-z. [Online]. Available: https://doi.org/10.1007/s10664-019-09780-z.

[19] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, Sep. 2019, ISSN: 1049-331X. DOI: 10.1145/3340544. [Online]. Available: https://doi.org/10.1145/3340544.

[20] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 479–490. DOI: 10.1109/SANER.2019.8668043.

[21] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 602–614.

[22] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "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*, ser. ISSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 101–114, ISBN: 9781450380089. DOI: 10.1145/3395363.3397369. [Online]. Available: https://doi.org/10.1145/3395363.3397369.

[23] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, May 2023, pp. 1482–1494. DOI: 10.1109/icse48619.2023.00129. [Online]. Available: http://dx.doi.org/10.1109/ICSE48619.2023.00129.

[24] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?" *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 2507–2525, 2024. DOI: 10.1109/TDSC.2023.3308897.

[25] Y. Wu, N. Jiang, H. V. Pham, *et al.*, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1282–1294, ISBN: 9798400702211. DOI: 10.1145/3597926.3598135. [Online]. Available: https://doi.org/10.1145/3597926.3598135.

[26] Y. Wei, O. Duchenne, J. Copet, *et al.*, *Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution*, 2025. arXiv: 2502.18449 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2502.18449.

[27] N. T. Islam, M. B. Karkevandi, and P. Najafirad, *Code security vulnerability repair using reinforcement learn-*

ing with large language models, 2024. arXiv: 2401.07031 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2401.07031.

[28] X. Wang, Y. Wang, Y. Wan, *et al.*, "Compilable neural code generation with compiler feedback," in *Findings of the Association for Computational Linguistics: ACL 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds., Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 9–19. DOI: 10.18653/v1/2022.findings-acl.2. [Online]. Available: https://aclanthology.org/2022.findings-acl.2/.

[29] J. Liu, Y. Zhu, K. Xiao, *et al.*, *Rltf: Reinforcement learning from unit test feedback*, 2023. arXiv: 2307.04349 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2307.04349.

[30] C. E. Jimenez, J. Yang, A. Wettig, *et al.*, "SWE-bench: Can language models resolve real-world github issues?" In *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66.

[31] J. Yang, C. E. Jimenez, A. Wettig, *et al.*, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[32] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.

[33] A. Mathai, C. Huang, S. Ma, *et al.*, *Crashfixer: A crash resolution agent for the linux kernel*, 2025. arXiv: 2504.20412 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.20412.

[34] F. Behrang, Z. Zhang, G.-V. Saioc, P. Liu, and M. Chabbi, *Dr.fix: Automatically fixing data races at industry scale*, 2025. arXiv: 2504.15637 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2504.15637.

[35] C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, *Migrating code at scale with llms at google*, Apr. 2025. DOI: 10.48550/arXiv.2504.09691.

[36] DeepSeek-AI, *Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning*, 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2501.12948.

[37] Z. Ma, C. Peng, P. Gao, X. Meng, Y. Zou, and B. Xie, *Sorft: Issue resolving with subtask-oriented reinforced fine-tuning*, 2025. arXiv: 2502.20127 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2502.20127.

[38] B. Hui, J. Yang, Z. Cui, *et al.*, *Qwen2.5-coder technical report*, 2024. arXiv: 2409.12186 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2409.12186.

[39] G. Sheng, C. Zhang, Z. Ye, *et al.*, "Hybridflow: A flexible and efficient rlhf framework," *arXiv preprint arXiv: 2409.19256*, 2024.