Agentic Reinforcement Learning for Real-World Code Repair

Siyu Zhu[*†] Anastasiya Karpovich[*†] Albert Chen Jessica Koscheka Shailesh Jannu Di Wen Yuqing Zhu Rohit Jain Alborz Geramifard[†]
LinkedIn

Abstract

We tackle the challenge of training reliable code-fixing agents in real repositories, where complex builds and shifting dependencies make evaluation unstable. We developed a verifiable pipeline with success defined as post-fix build validation and improved reproducibility across ∼1K real issues by pinning dependencies and disabling automatic upgrades. Building on this, we introduced a scalable simplified pipeline for large-scale reinforcement learning (RL). Using this setup, we supervise fine-tuned `Qwen3-32B` in the full pipeline and applied RL on top of SFT model in the simplified environment. The SFT model distilled from `GPT-4.1` trajectories performs on par while being 56× smaller, and RL added 7–20% absolute gains under matched train–test conditions. "Thinking mode" was on par or worse in our experiments. Both SFT and RL models failed to generalize across environments, highlighting the importance of matching train–test environments for building reliable real-world code-fixing agents.

Introduction

Large language models (LLMs) have transformed the landscape of code intelligence, powering systems such as GitHub Copilot (Zhang et al. 2023), ChatGPT Code Interpreter (Mutch 2025), and AlphaCode (Li et al. 2022). These models excel at code completion, bug fixing, and even multi-step development workflows, offering tangible productivity gains in both individual and collaborative programming settings. Recent research has extended their reach into more realistic environments: CodeRL (Le et al. 2022b) integrated reinforcement learning (RL) to improve correctness in competitive programming, CoRNStack (Suresh et al. 2024) advanced large-scale code retrieval and reranking, and surveys (e.g., Wang et al. 2024) highlight the breadth of RL applications for code generation, from reward shaping to execution-based optimization.

Despite these advances, applying LLMs to real-world repositories remains difficult. Production environments introduce heterogeneous build systems, shifting dependencies, and intricate project structures that often cause seemingly correct model-generated fixes to fail when executed. Proxy metrics such as code similarity to human patches correlate weakly with true functional success, and even RL approaches grounded in execution feedback, such as RLEF (Gehring et al. 2024a), focus primarily on uniform Python environments and still require stable, reproducible execution to function effectively.

In contrast, our setting involves heterogeneous repositories spanning multiple languages, dependency managers, and build systems. To ensure reliable learning signals, we develop a verifiable training and evaluation pipeline where success is defined by post-fix build validation. The pipeline ensures reproducibility by pinning dynamic dependencies and disabling automatic upgrades, providing a stable foundation for execution-grounded learning. On this platform, we train `Qwen3-32B` agents via supervised fine-tuning (SFT) followed by reinforcement learning (RL). The key contributions are summarized below.

- Formulated a realistic, large-scale agentic environment for code repair, encompassing heterogeneous problem types such as dependency issues and logical bugs across diverse file types (Java, Python, Config). The setup extends `VerL` to support multi-run execution, enabling robust orchestration of long-running, tool-driven episodes with built-in tolerance for failures and timeouts during intensive build operations. We further curated and analyzed a dataset of $1,000$ real-world problems.
- Provided strong empirical evidence for the effectiveness and necessity of realistic training environments. Applying SFT on `Qwen3-32B`, a $56\times$ smaller model, led to performance close to GPT-4.1 in the full pipeline, while reinforcement learning via GRPO (Shao et al. 2024) built on SFT yielded 7–20% absolute gains in the simplified pipeline. However, both SFT and RL models trained on the full and simplified pipelines respectively failed to generalize to their counterpart settings, underscoring the importance of matching train and test domains. Incorporating "thinking" did not lead to performance improvements and in some cases degraded results.
- Conducted a qualitative analysis of agent behavior evolution under RL, showing that initial LLMs behaved like novice developers applying recipe-based fixes, whereas RL-trained agents demonstrated behaviors akin to experienced engineers who identified root causes and invoked

[*]Equal contribution. Detailed contributions in Appendix .

[†]Corresponding authors:{`jzhu, akarpovich, agf`}`@linkedin.com`.

surgical tools. The study further highlights the importance of reward design to prevent reward hacking.

Related Work

LLMs for Automated Code Repair. LLMs have shown strong performance in program repair tasks when fine-tuned with execution-based signals. For instance, CodeRL (Le et al. 2022a) applies actor-critic RL guided by unit-test results on APPS and MBPP, improving functional correctness. Multi-objective training approaches (e.g., semantic and syntactic objectives) such as MORepair (Yang et al. 2024) improve robustness across both function-level and repository-level datasets, while SWE-Fixer (Yasunaga et al. 2024) scales repair to over 110K GitHub issues using retrieval plus generation.

Agentic RL with Execution Feedback. Agentic systems structure repair as a multi-turn decision-making process. RLEF (Gehring et al. 2024b) uses PPO-trained LLM agents receiving execution results during iterative code generation, achieving state-of-the-art on CodeContests, HumanEval+, and MBPP+. Agent-RLVR further enhances generalization using guided rewards on SWE-Bench Verified (Yu et al. 2024). RepairAgent (Bouzenia et al. 2024) executes autonomous planning and patch iteration on Defects4J, showing new bug fixes not solvable by previous methods.

Enterprise and Real-World Evaluation. Google's internal system Passerine demonstrates agentic LLM repair on 178 real bugs drawn from its internal issue tracker (78 human-reported, 100 machine-reported). With 20 sampled trajectories and Gemini 1.5 Pro, Passerine achieves plausible fix rates of 73% on machine-reported and 25.6% on human-reported bugs; semantically equivalent patch rates are 43% (machine) and 17.9% (human) (Rondon et al. 2025). Another system, BRT Agent, specializes in generating reproducible Bug Reproduction Tests (BRTs), achieving 28% plausible test generation rate (vs. 10% with prior technique LIBRO) on approximately 80 human-reported Google bugs. Supplying these BRTs to Passerine increases plausible patch generation by about 30%, and enables more efficient fix trajectory selection using the introduced Ensemble Pass Rate metric (Cheng et al. 2025).

Problem Formulation

Our system automatically repairs build and test failures by analyzing logs, retrieving relevant fixes, and synthesizing verified patches. Figure 1 shows the workflow. When a pull request fails to merge, the system enters a loop. It builds and validates the PR, then enters an automated repair loop. The *Log Analyzer* summarizes errors and selects the top one, *Fetch Potential Solutions* retrieves candidate fixes through historical data using RAG (Lewis et al. 2020), and *Solution Selector* ranks them by relevance and past success. The top solution, selected error, and repository name are provided to the *LLM*, which generates the final patch.

The following example illustrates the prompt provided to the agent during training and evaluation. It defines the agent's role, available tools, and operational constraints used to guide autonomous code-repair behavior.

Example Prompt

Role: `system`

You are a fully automated software agent tasked with independently fixing a build issue with a software project.

Build Error:
The Gradle version `5.6.4` used in the build has been deprecated. This can cause build failures or incompatibilities with newer plugins and dependencies.

Recommended Fix:
Apply the fix using the tools provided to fix the problem. Use the `validate_and_build` tool to verify the result of your work and act based on the results.

Important Instructions:

- DO NOT respond with suggestions, ask questions, or engage in a conversation.
- DO NOT ask for confirmation or approval to apply the fix or perform any actions.
- Never give up. Keep making decisions based on the information you have and keep taking action until the problem is fixed.

Available Tools:

The agent is provided with function signatures enclosed within `<tools></tools>` XML tags.

`find_files`: Find files by name or pattern.
Parameters: {`file_path` (string): glob or filename to search}

`read_file`: Read contents of a file.
Parameters: {`file_path` (string): file name or path to read}

`write_file`: Write contents to an existing file while preserving structure and comments.
Parameters: {`file_path` (string), `updated_contents` (string)}

`run_sh`: Execute shell commands and return stdout/stderr.
Parameters: {`cmd` (string): shell command to execute}

`upgrade_gradle`: Upgrade Gradle when builds fail due to deprecated versions.
Parameters: {}

`find_files_with_text`: Search for files containing a specific string.
Parameters: {`search_text` (string), `glob_file_pattern` (string, optional)}

`remove_dependency`: Remove a dependency from `product-spec.json`.
Parameters: {`dependency_name` (string)}

`ask_for_help`: Query internal knowledge base for troubleshooting advice.
Parameters: {`troubleshooting_question` (string)}

`dependency_upgrade`: Run `mint dependency update` to upgrade libraries.
Parameters: {`dependency_to_upgrade` (string), `version_to_upgrade_to` (string, optional)}

`validate_and_build`: Run a full build and return results.
Parameters: {}

Tool Call Format:
Each tool invocation returns a JSON object wrapped in `<tool_call></tool_call>` tags.

```
<tool_call>{"name": "<function-name>",
"arguments": {...}}</tool_call>
```

Role: `user`

Fix: Use the `upgrade_gradle` tool to upgrade Gradle.

Role: `assistant`

(The agent begins reasoning and tool invocation to address the issue.)

The patch is built and validated to verify the fix, with an LLM judge ensuring no loss of test coverage. If validation succeeds, a PR is created and the loop terminates. If a new, distinct error appears, the system commits the current state, resets the LLM context, and treats it as a new failure with updated fix suggestions. If the new error is similar to the original, the system discards the previous changes, retrieves an alternative fix, and retries. Each error can be retried up to three times before proceeding to the next, while the main loop continues until reaching the LangGraph recursion limit of 100.

MDP Formulation

Full Pipeline We model the code-repair process as a finite-horizon Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma)$. A state $s_t \in \mathcal{S}$ represents the system prompt, repository name, detected errors, selected solution, and reasoning context. An action $a_t \in \mathcal{A}$ emits a token that may invoke one of ten tools: `ask_for_help` (query internal knowledge base for a solution), `dependency_upgrade`, `find_files`, `find_files_with_text`, `remove_dependency`, `run_sh` (execute shell commands), `upgrade_gradle`, `validate_and_build`, `read_file`, and `write_file` (code). See Appendix **??** for details. Instead of editing diffs, the agent writes complete files, from which a utility extracts patches to form a PR. While `run_sh` subsumes all other tools, its generality increases reasoning complexity, so the remaining nine tools are explicitly exposed to simplify the decision space. The full pipeline, built on `LangChain` for tool execution and `LangSmith` for trace logging, mirrors the deployed environment and determines success from trajectory-level flags in the `LangSmith` database.

The transition function $\mathcal{T}(s_{t+1}|s_t, a_t)$ appends the emitted token to the reasoning trace and, if a tool is invoked, incorporates its response into the next state. Rewards are sparse, with $r_t = 1$ when a build succeeds and LLM judge approves the change and $r_t = 0$ otherwise. Each episode terminates upon success, after 50 tool invocations, or when wall-time exceeds 80 minutes. Although the full MDP could, in principle, continue trajectories across loops, each failed submission is treated as a new trajectory for better context management. We set $\gamma = 1$ given the finite horizon.

Simplified Pipeline Collecting on-policy data from the full pipeline was computationally prohibitive, so we implemented a simplified RL environment in `VerL` (Sheng et al. 2024). This setup treats each $\langle \texttt{repo}, \texttt{error}, \texttt{solution} \rangle$ tuple as a one-shot fix rather than running the full iterative loop. Since the error context remains static, repositories with multiple errors were split into separate examples, one per error. The agent must still resolve all errors for overall success, but each problem focuses on a different starting error. To enable efficient rollouts, we cached repositories, build states, error logs, and retrieval outputs on shared NFS storage, isolating each rollout in its own directory. We reduced the episode length from 50 to 30, retained the same toolset as in the full pipeline, and used the same reward except that the LLM judge was disabled and manual spot checks were performed to ensure code integrity.

Data

We collected a dataset of $\sim$1K real-world code-fixing problems from Linkedin, broken via time-ordered split into train (80%), validation (10%), and test (10%). Each problem consists of a failing commit with build logs, error messages, and retrieved context such as fix suggestions or code documentation. Our problems spanned heterogeneous repositories and multiple programming languages (e.g. Python, Java) and build systems, making this dataset a challenging benchmark for code automation, with failure modes ranging from missing dependencies to multi-stage compilation errors. Reproducibility was challenged by environmental non-determinism arising from (1) wildcard dependencies, (2) automated dependency updates (ADU), (3) non-deterministic retrieval, and (4) infrastructure variability. Retrospective analysis showed that 40% of past fixes no longer build, while 1% of prior failures now succeed without a fix. Pinning dependencies and disabling ADU recovered $\sim$20% of failed fixes; We also removed the 1% problems that succeeded out of the box.

To better understand the dataset, we ran all problems through the full pipeline using `Qwen3-32B`. Over 60% of tokens were devoted to internal *thinking*, while `write_file` actions accounted for about 20%, confirming that thinking mode quickly exhausts context. Successful trajectories averaged 12 turns, whereas failed ones often reached 30 and hit the 50-step cap. Each run took roughly four hours, with about half the time spent on build validation, averaging three minutes per call but occasionally exceeding one hour. Dataset analysis revealed that **81%** of all errors were dependency-related. Overall, these findings show that agentic repair is both reasoning-intensive and computationally demanding. For further analysis see Appendix .

For SFT, we used the full pipeline and constructed two complementary subsets. The *thinking dataset* comprises 101 trajectories (84 train / 17 validation) from successful `Qwen3-32B` runs, preserving multi-turn reasoning traces with explicit `<think>` annotations. The *no-thinking dataset* includes 365 trajectories (311 train / 54 validation)

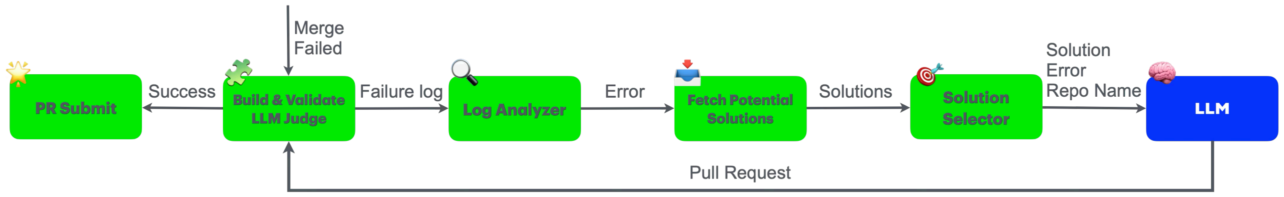

Figure 1: Automated code-fixing pipeline. Each PR passes through build and validation; on failure, the system analyzes logs, retrieves candidate patches, selects a promising fix, and synthesizes a solution before retrying the build.

from successful `GPT-4.1`[1] and `Qwen3-32B` runs, where thinking tokens were removed to provide concise tool-call supervision. Both subsets were restricted to problems reproducible under the stabilized environment. The *thinking dataset* only included `Qwen` runs, as `GPT` models do not natively support explicit reasoning traces.

For RL, we expanded the whole data into $\langle$repo, error, candidate fix$\rangle$ tuples as described in Section and then filtered training to 2469 cases with initial build times under 100 seconds, prioritizing environments that allowed high-throughput rollouts

Experimental Setup

We used a context window of 131K tokens via RoPE scaling (Su et al. 2021; Xiong et al. 2023) for both SFT and RL. For SFT, we converted successful trajectories into chat-formatted text, masking out instructions and tool responses during loss computation to focus learning on tool command generation, with or without thinking traces. Training used the `HuggingFace TRL` library in FP32 precision on `Qwen3-32B` with a multi-turn SFT pipeline. All experiments were run on a single H200 node with 8 GPUs. We determined the largest feasible per-GPU batch size (4). Learning rates were tuned over $\{6, 20, 60, 200\} \times 10^{-7}$, with 200×10^{-7} yielding the best validation results.

For RL, we experimented with both PPO (Schulman et al. 2017) and GRPO (Shao et al. 2024). Given GRPO's better preliminary performance, we adopted it for our main experiments. Within `VerL`, we enabled multiple concurrent code-fixing agents trained under GRPO. Because each agent executes `validate_and_build`, a CPU-intensive operation, high rollout concurrency can cause CPU contention. To mitigate contention, we fixed the batch size to 8 and the number of rollouts per batch to 4, resulting in up to 32 concurrent builds per node. Episodes terminate upon success or when either the turn or time limit is reached, after which incomplete runs are recorded as failures. We used $1,000$ learning steps.

For the initial RL model, we compared the `Qwen3-32B` base model with its supervised fine-tuned variant (`Qwen3-SFT`) and selected the latter due to better observed performance. We further extended `VerL` to enable large-scale agentic RL via controlled parallel rollouts, addressing concurrency issues in asynchronous execution and introducing one-hour tool timeouts that automatically emit a `tooltimeout` signal. These extensions provide reproducible and scalable rollouts across diverse environments.

We evaluated models in both full and simplified pipelines. The full pipeline was built on `LangChain` for tool execution and `LangSmith` for trace logging. It computed success from trajectory-level flags in the `LangSmith` database. The simplified pipeline provided faster, controlled evaluation on $8\times$H200 GPUs. Since `VerL` cannot perform rollouts with GPT models, we report GPT results only for the full pipeline. For efficiency, we disabled running validations during the RL training. We triggered evaluation runs separately from training: in `VerL`, setting `num_epoch`=0 triggers evaluation on the validation and test sets, storing the results without entering the training loop. Each evaluation configuration was run five times, and 95% confidence intervals were computed.

Experimental Results

Figure 2 summarizes PR success across models in the full pipeline. The asterisk indicates execution with thinking. The SFT model distilled from `GPT-4.1` trajectories (without reasoning traces) achieved 12.3% success while being $\sim$ $56\times$ smaller[2] than `GPT-4.1` (15.1%), a difference not statistically significant. RL fine-tuning in the simplified pipeline yielded only marginal gains over the base model here, likely due to train–test mismatch, highlighting the need for consistent environments. Thinking mode offered no clear benefit: it slightly improved base and RL models but degraded SFT performance, likely due to context overhead. A nine-day live A/B test between `Qwen3-SFT` and `GPT-4.1` showed a 17% relative drop, consistent with the 18.5% offline gap we observe here.

Figure 3 shows results in the simplified pipeline. RL fine-tuning yielded strong gains due to matched train–test conditions. `Qwen3-RL` outperformed its base model by 7–20 absolute points (7% $\to$ 27% on validation and 17% $\to$ 24% on test), showing clear learning. In contrast, SFT models showed no improvement, underscoring their sensitivity to environmental mismatch. "Thinking" again offered no benefit, reducing or neutralizing performance in code repair tasks.

Policy Shift Analysis

We analyzed agent behavior shifts under RL. Figure 4 shows next-tool selection given the current tool, with

[1]GPT model was accessed via Azure OpenAI Service.

[2]Estimated `GPT-4.x` size: 1.8T (Howarth 2025).

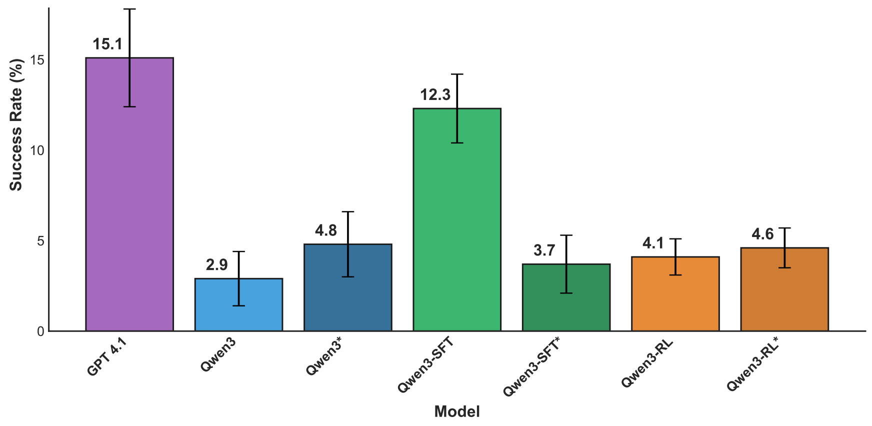

Figure 2: PR success rate with 95% confidence interval of various models on the test set for the full pipeline. The * in the name means execution with thinking.

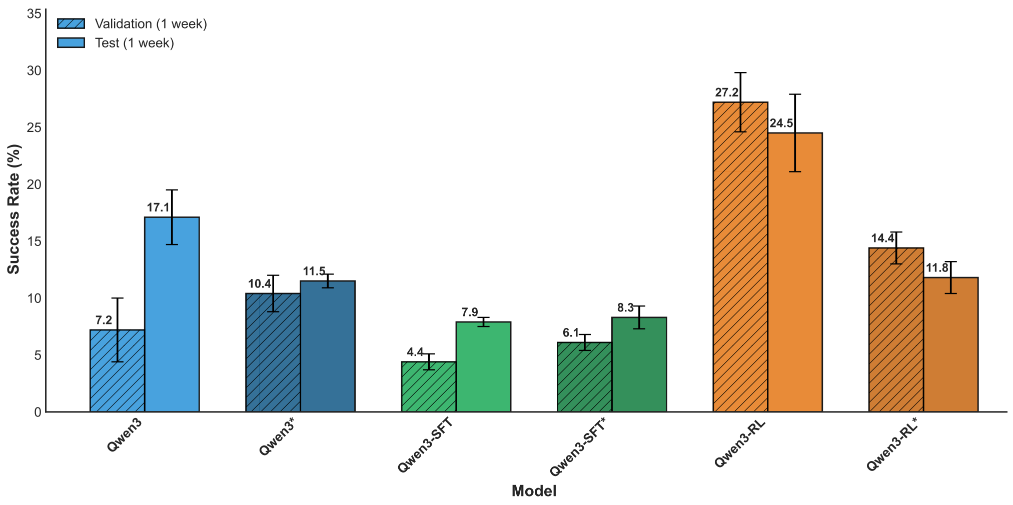

Figure 3: PR success rate with 95% confidence interval of various models on the test set for the simplified `VerL` pipeline. The * in the name means execution with thinking.

each row's percentage indicating how often that tool was used during evaluation. The base `Qwen3-32B` (left) behaved like a general developer, using many tools in a typical sequence: `find_files` → `read_files` → `write_file` → `validate_and_build` → `ask_for_help`. After RL fine-tuning (right), the agent behaved more like an expert, emphasizing high-impact actions such as `dependency_upgrade` (14%) and `validate_and_build` (71%), consistent with our finding that 80% of the training data were dependency-related (see Appendix for details). The agent also learned to rerun builds to handle infrastructure instability, revealing awareness that compilation can be non-deterministic. Extending training to 3,000 steps increased PR success to 65%, but the agent achieved this by removing validation code, exposing reward exploitation and underscoring the need for stronger reward design (e.g., LLM-based judges).

Conclusion

We developed a realistic, large-scale agentic reinforcement learning framework for automated code repair, extending `VerL` to support multi-run execution, fault tolerance, and reproducibility across heterogeneous problem types and languages. Comparing "thinking" and non-"thinking" variants revealed that the base model with thinking incurred high reasoning overhead, with most tokens spent on deliberation without corresponding performance gains, suggesting suboptimal context utilization during extended reasoning traces. Supervised fine-tuning on `Qwen3-32B`, a 56× smaller model, achieved performance close to `GPT-4.1` in the full pipeline, while reinforcement learning via GRPO (Shao et al. 2024) built on SFT yielded 7–20% absolute gains in the simplified pipeline. However, neither SFT nor RL models generalized across pipelines, highlighting the impact of train–test distribution shift. Extended RL runs exposed reward exploitation behaviors, such as agents removing val-

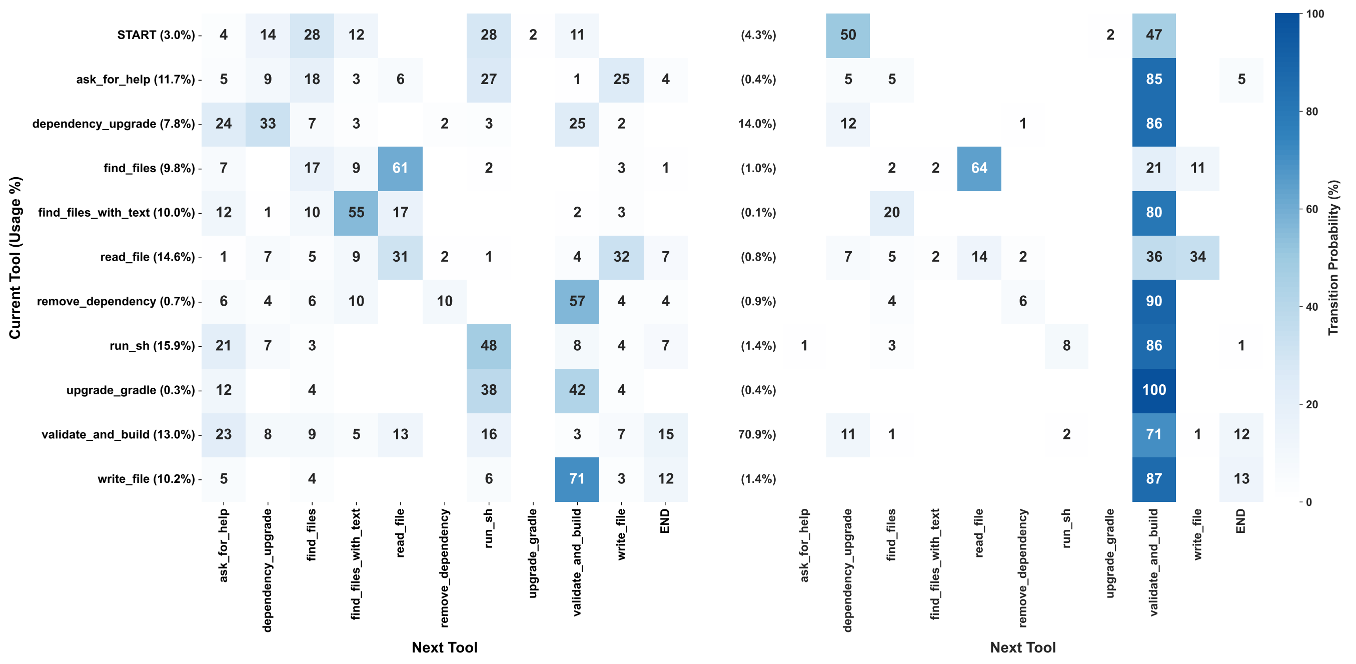

Figure 4: Policy shift between `Qwen3-32B` base (**left**) and SFT+RL-fine-tuned (**right**) using a Markov chain transition probability matrix. The base model issued diverse tool calls with low task focus, while the fine-tuned model converged to focused sequences dominated by `dependency_upgrade` and `validate_and_build`, improving build success.

idation code to inflate success, emphasizing the need for more robust reward design. Overall, RL demonstrates clear potential to evolve agent behavior from ad hoc, recipe-based fixes toward expert-like reasoning, though a substantial gap remains before such systems can operate reliably in complex, real-world production settings.

Acknowledgments

Siyu Zhu developed the sandbox environment and conducted reinforcement learning experiments. Anastasiya Karpovich contributed to paper editing, evaluations, and dataset creation. Albert Chen supported dataset creation, RL training, and evaluations. Jessica Koscheka worked on dataset creation, live testing, and analysis. Shailesh Jannu enhanced system stability and performed full pipeline evaluations. Di Wen led supervised fine-tuning training, and Yuqing Zhu carried out data analysis. Rohit Jain provided technical guidance and manuscript feedback. Alborz Geramifard oversaw the project and contributed to experimental design, data analysis, and served as the primary author of the manuscript.

We thank Bhavya Bharat Agarwal and Michaeel Kazi for their thorough reviews and valuable feedback on the paper. We are also grateful to Koushik Srinivas and Peter Phan for their early consultations that helped shape and lift the project. We further acknowledge Deepak Agarwal, Gungor Polatkan, and Balaji Srinivasan for their resourcing support and executive sponsorship that made this work possible.

References

Bouzenia, R.; et al. 2024. RepairAgent: An autonomous language agent for program repair. *arXiv:2403.17134*.

Cheng, R.; Tufano, M.; Cito, J.; Cambronero, J. P.; Rondon, P.; Wei, R.; Sun, A.; and Chandra, S. 2025. Agentic Bug Reproduction for Effective Automated Program Repair at Google. *arXiv:2502.01821*.

Gehring, J.; Zheng, K.; Copet, J.; Mella, V.; Carbonneaux, Q.; Cohen, T.; and Synnaeve, G. 2024a. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. *arXiv [cs.CL]*.

Gehring, J.; Zheng, K.; Copet, J.; Mella, V.; Carbonneaux, Q.; Cohen, T.; and Synnaeve, G. 2024b. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. *arXiv:2410.02089*.

Howarth, J. 2025. Number of Parameters in GPT-4. https://explodingtopics.com/blog/gpt-parameters?utm_source=chatgpt.com. Number of Parameters in GPT-4.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. 2022a. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv:2207.01780*.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. H. 2022b. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv [cs.LG]*.

Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.-t.; Rocktäschel, T.; Riedel, S.; and Kiela, D. 2020. Retrieval-Augmented

Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A. D.; Hubert, T.; Choy, P.; de Masson d'Autume, C.; Babuschkin, I.; Chen, X.; Huang, P.-S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D. J.; Robson, E. S.; Kohli, P.; de Freitas, N.; Kavukcuoglu, K.; and Vinyals, O. 2022. Competition-Level Code Generation with AlphaCode. *Science*.

Mutch, L. 2025. ChatGPT Code Interpreter Plus. https://chatgpt.com/g/g-2GxYeJcn6-code-interpreter-plus. Accessed: 2025-10-17.

Rondon, P.; Wei, R.; Cambronero, J. P.; Cito, J.; Sun, A.; Tufano, M.; Sanyam, S.; and Chandra, S. 2025. Evaluating Agent-based Program Repair at Google. In *ICSE Software Engineering in Practice (SEIP)*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.

Shao, Z.; Wang, P.; Zhu, Q.; Xu, R.; Song, J.; Bi, X.; Zhang, H.; Zhang, M.; Li, Y. K.; Wu, Y.; and Guo, D. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv preprint arXiv:2402.03300. V3, cs.CL / cs.AI / cs.LG.

Sheng, G.; Zhang, C.; Ye, Z.; Wu, X.; Zhang, W.; Zhang, R.; Peng, Y.; Lin, H.; and Wu, C. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv: 2409.19256*.

Su, J.; Lu, Y.; Pan, S.; Wen, B.; and Liu, Y. 2021. RoFormer: Enhanced Transformer with Rotary Position Embedding. *arXiv preprint arXiv:2104.09864*.

Suresh, T.; Reddy, R. G.; Xu, Y.; Nussbaum, Z.; Mulyar, A.; Duderstadt, B.; and Ji, H. 2024. CoRNStack: High-quality contrastive data for better code retrieval and reranking. *arXiv [cs.CL]*.

Wang, J.; Zhang, Z.; He, Y.; Zhang, Z.; Song, Y.; Shi, T.; Li, Y.; Xu, H.; Wu, K.; Yi, X.; Wan, Z.; Yuan, X.; Lu, K.; Huo, M.; Qian, G.; Li, K.; Chen, Q.; and He, L. 2024. Enhancing Code LLMs with reinforcement learning in code generation: A survey. *arXiv [cs.SE]*.

Xiong, W.; Chen, L.; Song, X.; et al. 2023. Effective Long-Context Scaling of Foundation Models. *arXiv preprint arXiv:2309.16039*.

Yang, L.; Zhang, Y.; Wang, J.; et al. 2024. MORepair: Multi-objective training for robust and generalizable program repair with LLMs. *arXiv:2404.12636*.

Yasunaga, M.; Deng, S.; Yin, P.; and Liang, P. 2024. SWE-Fixer: GitHub-scale software bug fixing with retrieval and generation. *arXiv:2501.05040*.

Yu, X.; et al. 2024. RLVR: Reinforcement Learning from Verifiable Rewards for Software Engineering Agents. *arXiv:2506.11425*.

Zhang, B.; Liang, P.; Zhou, X.; Ahmad, A.; and Waseem, M. 2023. Practices and Challenges of Using GitHub Copilot: An Empirical Study. *CoRR*, abs/2303.08733.

Appendix

Data Analysis

We analyzed runtime behavior and dataset characteristics of the baseline `Qwen3-32B` model before training. The following figures summarize token usage, trajectory length, build times, and error composition, highlighting the computational and failure patterns of real-world code repair.

Token Distribution with Thinking Enabled Figure 5 presents the token ratio per trajectory across success, failure, and total runs for the `Qwen3-32B` base model in the full pipeline. Tokens are grouped into *thinking*, `tool_call:write_file` as the coding is captured within calling the tool rather than its output. For the rest of tools we focus on their output. We also included both user prompt and system prompt (shown as content). We excluded categories contributing less than 1%. Over 60% of all tokens are devoted to internal reasoning (*thinking*), followed by `write_file` operations, reflecting the verbosity of full-file generations.

Token Distribution without Thinking Figure 6 isolates tool-related contributions by excluding assistant thinking tokens. Majority of tokens used for coding. The distribution highlights that `run_sh_response` accounts for 27% of tokens in successful runs but only 7% in failures, whereas `ask_for_help_response` remains under 2%, suggesting limited utility of this tool type.

Distribution of Turns per Trajectory Figure 7 shows the mean $\pm$ 95% confidence interval of the number of turns per trajectory. Successful repairs typically complete in 12 turns, while failed trajectories average around 30 and frequently reach the 50-step cap, illustrating the efficiency gap between effective and unsuccessful runs.

Token Utilization per LLM Call Figure 8 reports the input and output token sizes per LLM call on a log scale, capped at 131K tokens due to the `Qwen3-32B` context limit. Inputs include accumulated prompts, tool responses, and system context, while outputs consist of assistant thinking and tool call tokens. On average, 80% of output tokens are used for internal reasoning, highlighting the computational overhead of the thinking process.

Build and Validation Time Distribution Figure 9 presents the distribution of build and validation durations across. The build step averaged about $\sim$ 4 minutes with occasional outliers reaching up to 60 minutes. These long-tail distributions motivated prioritizing repositories with shorter build times during training to maintain high rollout throughput.

Error Distribution To analyze the distribution of build errors in our training dataset, we categorized 2,469 error instances from the simplified pipeline using keyword-based filters. Errors were assigned to categories based on keyword matches, following the rules listed in Table 1. Each error is assigned to the first matching rule.

Figure 10 shows the resulting category distribution. We found that **81.1%** (2,003 errors) fall under dependency-

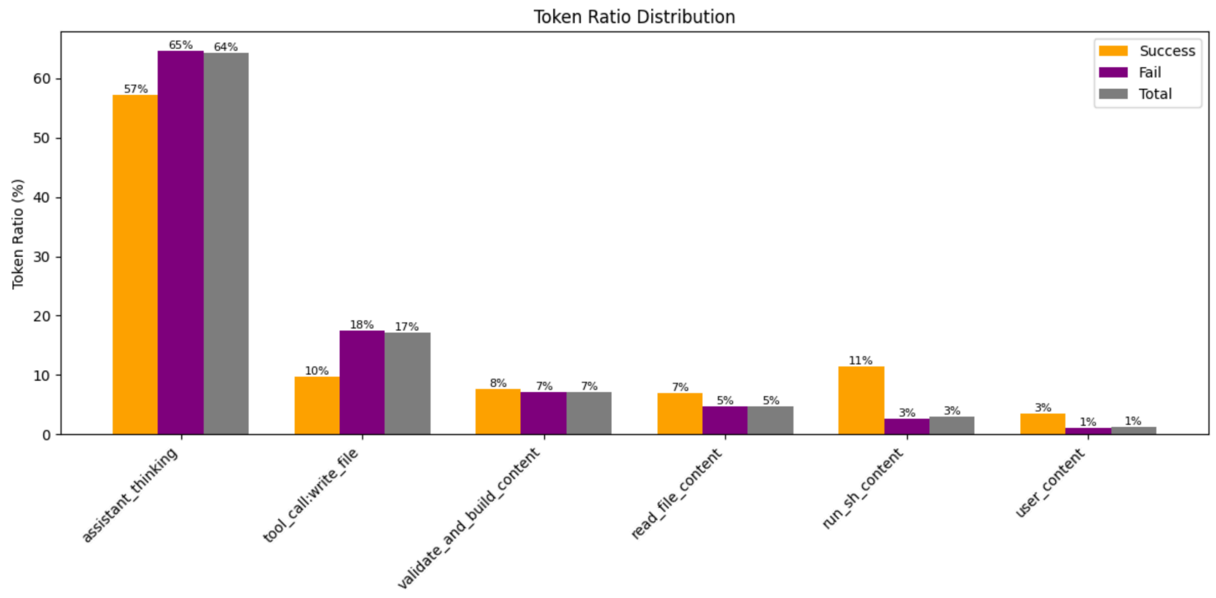

Figure 5: Token distribution across success, failure, and total runs with assistant thinking included.

Table 1: Keyword filters for error categorization. Each error is assigned to the first matching rule.

Category	**Matching Keywords / Patterns**
Dependency-Related	dependency, dependencies
Build Tool	gradle, maven, build tool, build failed, compilation failed
Test	test, unit test, integration test, test case, test failure
Configuration	configuration, config, schema, avsc, yaml, yml, json, xml
Installation	install, yarn, npm, pip, package manager
Version	version, compatibility, incompatible, mismatch
Environment	path, environment, variable, not found, cannot locate, missing
Permission	permission, access, denied, forbidden
Other	No keyword match above

related issues, including deprecated packages, missing artifacts, duplicate dependencies, and transitive resolution failures. The remaining errors are distributed across Build Tool (5.3%), Test (3.5%), Configuration (3.2%), Environment (2.6%), Version (1.8%), Other (1.5%), Installation (0.8%), and Permission (0.2%). These results indicate that dependency management accounts for the majority of build failures in our dataset, making it the most critical target for automated repair.

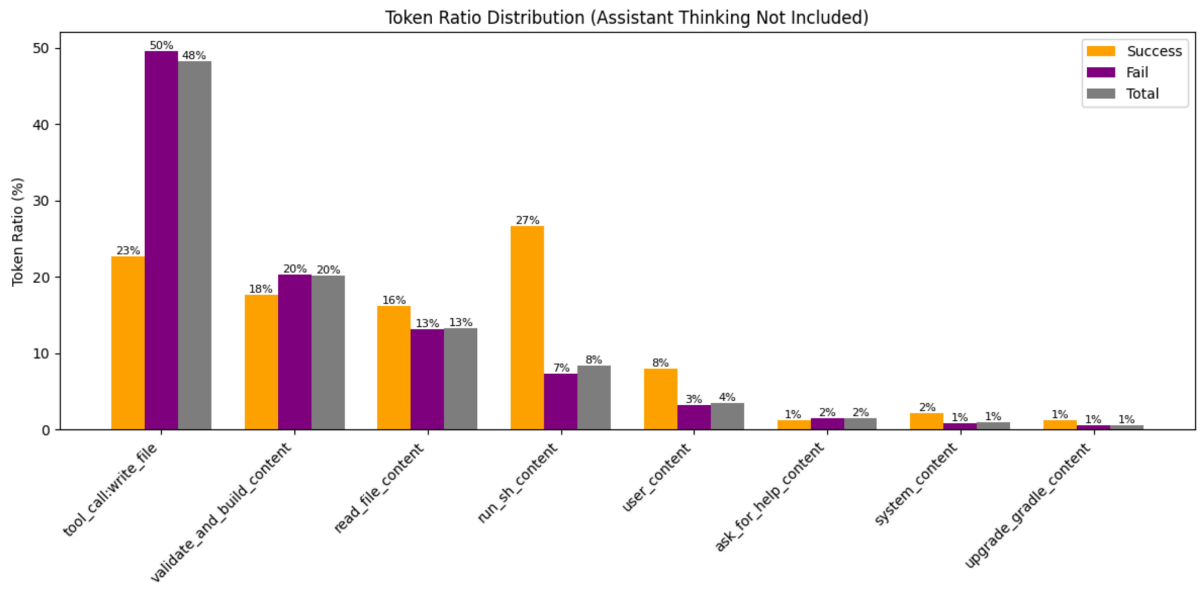

Figure 6: Token distribution excluding assistant thinking, emphasizing tool-level token contributions.

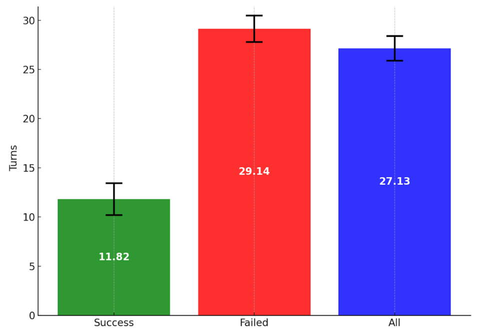

Figure 7: Distribution of trajectory lengths (turns) across successful and failed runs.

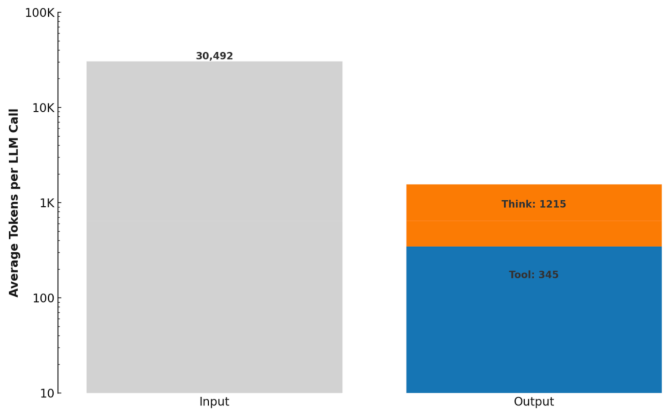

Figure 8: Token utilization per LLM call, showing input/output sizes and distribution across trajectories.

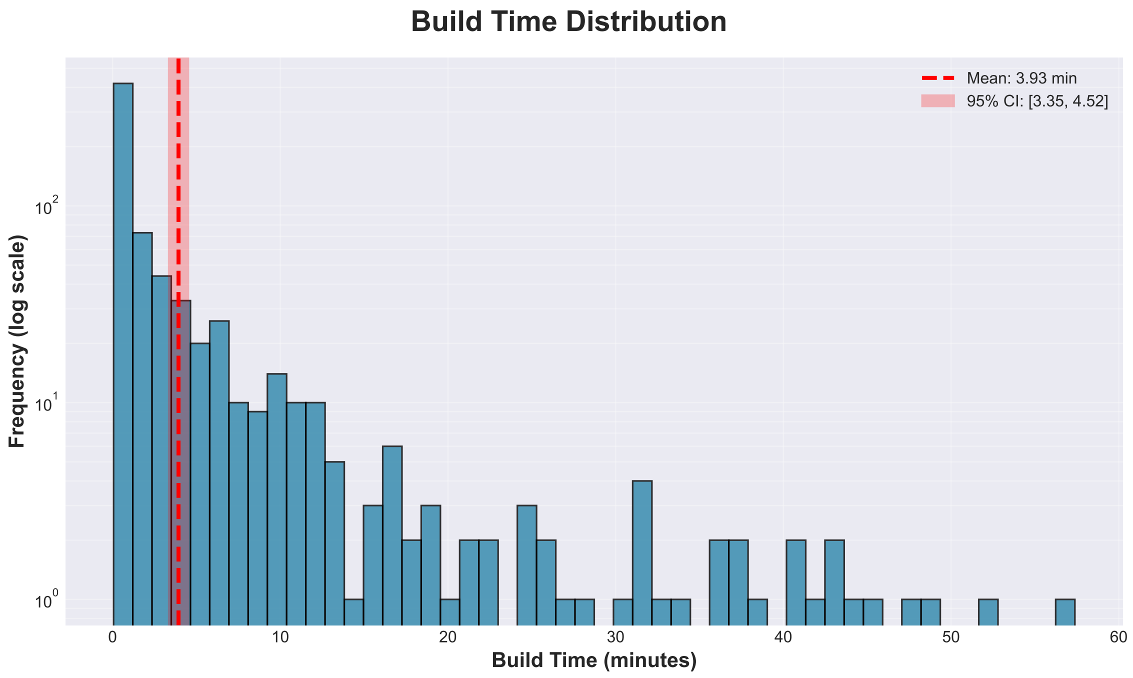

Figure 9: Distribution of build and validation times (log scale) across training problems.

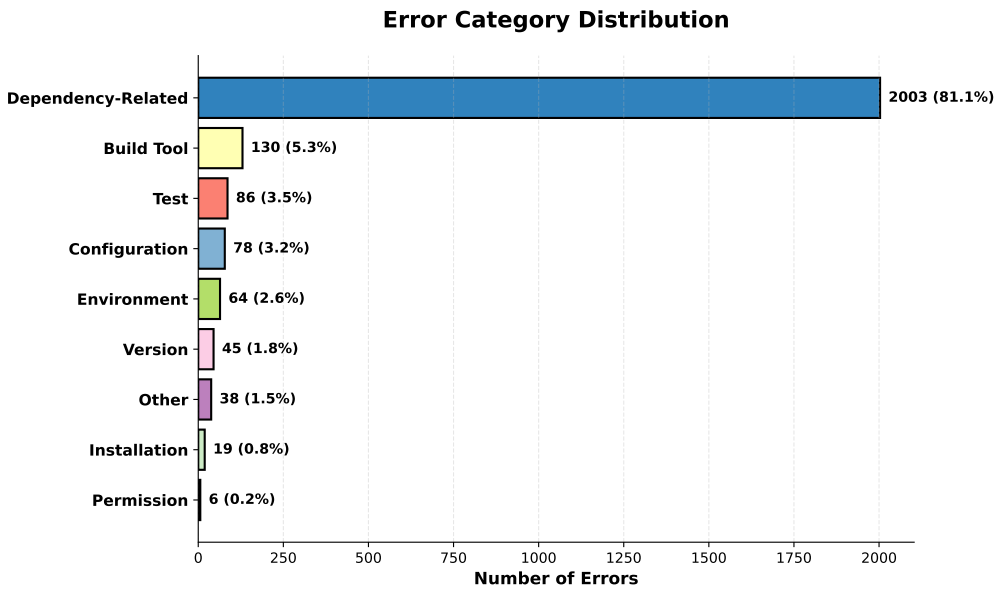

Figure 10: Distribution of error categories in the training dataset of simplified pipeline