TESTSMITH: REINFORCEMENT LEARNING FOR UNIT TEST GENERATION WITH SYNTHETIC PERTURBATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

We present TestSmith, a reinforcement learning approach for training language models to generate unit tests that directly optimize for fault detection. Existing methods rely on coverage proxies or human-written test suites as training signal, but coverage correlates weakly with fault detection and curated test data is scarce, creating a circular dependency between test quality evaluation and test generation. TestSmith breaks this loop by using automatically generated synthetic bugs, both AST-based mutations and LLM-produced semantic perturbations spanning logic errors, boundary mistakes, type errors, and incomplete handling, as an execution-based reward signal, with an explicit penalty for tests that fail on correct code. We train a Qwen3-8B policy with Group Relative Policy Optimization and a three-stage curriculum that progresses from single-function problems with operator changes, to single-function problems with LLM-generated bugs, to multi-file repository tasks, addressing the reward sparsity that otherwise prevents learning in complex settings. On a held-out benchmark of 500 problems, TestSmith raises Pass@5 from 56.50% to 74.19%, achieves a mutation score of 100% (up from 74.99%), and reaches 99.22% line and branch coverage, demonstrating that synthetic bug rewards can effectively align test generation with fault detection.

1 INTRODUCTION

Unit tests are critical for software reliability, but automatically generating tests that reliably detect faults remains challenging. Large language models can produce plausible tests, yet existing approaches rely on coverage or curated examples that correlate weakly with fault detection. We present TestSmith, a reinforcement learning framework that trains language models to generate unit tests using execution rewards defined by detecting automatically generated synthetic bugs. By directly rewarding fault detection while penalizing invalid tests, the model learns to produce tests that find real errors rather than simply execute code.

The central difficulty is evaluation. Proxy metrics like code coverage correlate weakly with fault detection [15], and training on human-written tests requires scarce, expensive data that may not generalize. This creates a circular dependency: we need strong tests to evaluate candidate tests, but we need an evaluation signal to learn to produce strong tests in the first place.

Our key insight is that we can *automatically generate the bugs* that tests should catch. Given a natural-language specification and a correct reference implementation, we use a frozen LLM to produce realistic buggy variants, programs that violate the specification in ways that real programmers might. We then train a test-generating policy with reinforcement learning to maximize detection of these synthetic bugs while maintaining validity (not failing on correct code).

We implement this approach in TestSmith. Our system generates buggy variants from two complementary sources: (1) AST-based mutations providing broad syntactic coverage, and (2) LLM-produced bugs that make larger yet subtle perturbations, akin to ones that an inexperienced developer might. We train with Group Relative Policy Optimization (GRPO) [26], which eliminates the need for a separate critic network. Critically, we also employ a *curriculum learning* strategy that begins with single-function problems where reward signal is dense, then progressively introduces multi-file repository tasks where naive training produces sparse rewards.

Contributions.

- We propose an **execution-based reward** for unit test generation that combines synthetic bug detection and an explicit validity penalty ($R = -1$ for false positives), enabling scalable RL training without human-labeled test suites.
- We develop a **difficulty-based curriculum** progressing from single functions to multi-file repositories, addressing reward sparsity in complex settings.
- We build an **end-to-end system** with LoRA fine-tuning [14] and sandboxed execution, demonstrating practical deployment.

2 Related Work

A useful lens on prior work is the *training signal* and *evaluation setting* used for unit test generation. Many strong systems already combine LLMs with execution, search, and mutation feedback. The open question is how to turn these signals into a scalable learning objective that prioritizes fault detection while preserving test validity.

2.1 LLM unit test generation with execution feedback

Coverage-guided generation. A common approach is to use coverage as a proxy objective and to iterate until additional coverage becomes hard to obtain. CodaMosa [22] queries an LLM when search-based generation plateaus, while TestART [13] uses generation and repair iterations guided by coverage feedback. These methods are often strong at producing valid test scaffolds and expanding execution paths, but coverage can be a weak predictor of fault detection [15]. *Takeaway:* coverage feedback is effective for validity and exploration, but it does not directly optimize the ability to catch bugs.

Mutation-guided prompting. Recent systems incorporate mutation information to target more discriminative assertions. MUTGEN [29] conditions generation on surviving mutants, PolyTest [20] uses diverse sampling and filtering, and MuTAP [4] augments prompts with mutation feedback. Relative to pure coverage guidance, mutation guidance is closer to the intended goal, but these methods typically operate at test time via prompting and heuristics rather than learning a test generation policy. *Takeaway:* mutation signals move the objective toward fault detection, but most approaches do not convert this signal into a stable training objective.

Iterative refinement loops. Several systems improve validity and assertion quality through execution-driven repair. ChatUniTest [9] uses generation, validation, and repair, TestPilot [25] mines usage context and iterates on failures, and LLMLOOP [28] automates refinement using a tool chain that includes static checks and mutation feedback. These loops can be highly effective in practice, especially when the surrounding harness is well engineered, but they primarily trade additional inference time for quality rather than improving the underlying model. *Takeaway:* iterative loops provide strong baselines for test quality, but they do not directly address how to train a model to internalize fault-detection behavior.

2.2 RL for code with execution rewards

Execution as supervision. Reinforcement learning with execution feedback has improved functional correctness in code generation. CodeRL [21] and related methods such as PPOCoder [27] and RLEF [12] use compiler or unit-test outcomes as rewards, while CodeRL+ [17] explores richer execution traces. Our setting differs because the generated artifact is a *test suite* whose reward depends on both validity on a reference implementation and failure on many buggy variants. *Takeaway:* execution-based RL is a proven tool for code, but applying it to tests requires a reward that separates false positives from genuine bug detection.

Closest work: RL for test generation. UTRL [7] explores co-evolutionary training of code and tests. Co-evolution can encourage stronger adversaries, but it also introduces stability issues and complicates attribution of improvements. Our approach holds the bug generator fixed, uses an

explicit invalid-test penalty, and uses a curriculum to mitigate sparse rewards when moving from single functions to repository-style tasks. *Takeaway:* co-evolution is promising but unstable; fixed adversaries and explicit validity constraints provide a simpler path to scalable training.

2.3 Synthetic bugs and mutation testing as objectives

Classic mutation testing. Mutation testing uses program variants as a proxy for real faults [16]. Tools can generate many mutants cheaply and with high compilability, but the resulting faults can be unrealistically shallow or stylistically narrow [19]. *Takeaway:* syntactic mutants scale well, but they can miss the semantic diversity needed to drive better tests.

LLM-generated mutants. LLM-based mutation aims to increase realism and diversity. Wukong [5] and LLMorpheus [3] suggest that LLM mutations broaden fault patterns, and Meta's ACH tool [6] explores mutation and test generation at larger scale. However, synthetic bug distributions still differ from organic bugs, and some evidence suggests they can remain shallower in control flow [2]. *Takeaway:* LLM mutation improves realism, but training can still overfit to the synthetic bug distribution.

2.4 Agentic software engineering benchmarks and harnesses

Modern code systems are increasingly evaluated as agents that interact with repositories, tooling, and tests. SWE-bench [18] and SWE-bench Verified [24] evaluate issue resolution in real repositories. In this context, unit tests serve both as evaluation and as a tool for agents to diagnose and validate changes. Benchmarks such as TestGenEval [1] and UnLeakedTestBench [10] focus specifically on unit test generation and highlight leakage and evaluation pitfalls. *Takeaway:* strong results increasingly require end-to-end harnesses and realistic repositories, which motivates training signals that remain meaningful when integrated into agent workflows.

2.5 Gap and positioning

Across these buckets, the strongest existing methods often do well at generating executable tests and expanding coverage, and mutation guidance can improve discriminative power. What is still missing is a simple training objective that directly optimizes for fault detection while preventing trivial always-failing tests, and that remains trainable as tasks scale from single functions to multi-file settings. TestSmith targets this gap by using synthetic buggy variants as the reward, imposing an explicit validity constraint, and using curriculum learning to maintain reward signal in harder settings.

3 Problem Formulation

Let S be a natural-language specification for a Python function and C be a reference implementation satisfying S. A test suite T is a Python file containing unit tests (e.g., using `pytest`).

Validity. A test suite is *valid* if all tests pass on the correct implementation:

$$\text{valid}(T, C) \triangleq \text{all tests in } T \text{ pass when executed against } C \tag{1}$$

Bug detection. For a set of buggy variants $\mathcal{B} = \{B_1, \ldots, B_n\}$, we measure detection rate:

$$\text{detect}(T, \mathcal{B}) \triangleq \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{\text{at least one test in } T \text{ fails on } B_i\} \tag{2}$$

Goal. Learn a policy π_θ mapping (S, C) to test suites T that maximize detection rate subject to validity:

$$\max_{\theta} \; \mathbb{E}_{(S,C)\sim\mathcal{D}} \left[\mathbb{E}_{T\sim\pi_\theta(\cdot|S,C)} \left[\text{detect}(T, \mathcal{B}) \cdot \mathbf{1}\{\text{valid}(T, C)\} \right] \right] \tag{3}$$

4 METHOD

4.1 SYNTHETIC BUG GENERATION

For each problem (S, C), we construct buggy variants from two complementary sources.

AST-based mutations. We apply classical mutation operators such as arithmetic operator replacement, relational operator swap, statement deletion, constant perturbation, and control flow modification using automated tooling. This provides broad syntactic coverage and enables large-scale generation without API costs.

LLM-generated semantic bugs. We prompt a frozen frontier LLM (separate from the policy being trained) to generate buggy versions of C that violate S while remaining plausible. Generally, we want these to be perturbations to be more nuanced and be based on a complex understanding of the code.

These could include bugs in the following categories:

1. **Logic errors**: Incorrect algorithm implementations, weakly held assumptions, etc.
2. **Boundary mistakes**: Off-by-one errors, incorrect handling of empty inputs, fence-post errors
3. **Type errors**: Missing type checks, incorrect conversions, null/None handling failures
4. **Incomplete handling**: Missing edge cases explicitly described in S, partial implementation

System prompt sensitivity. We found that the system prompt used for LLM-based perturbation generation has a large effect on downstream RL performance. Prompts that produced overly subtle or highly realistic perturbations often made rewards too sparse, because early test generators rarely killed any variants. Conversely, prompts that produced shallow or repetitive bugs led to fast learning but weaker generalization, and often resembled simple AST mutants. In practice, there is a sweet spot where perturbations are meaningfully more semantic than AST mutations, but still simple enough that generated tests can detect them early in training. We tried a variety of prompt templates and constraints, and observed substantial variance in learning curves and final mutation score across prompt choices.

Bug validation. We filter generated bugs to ensure they compile successfully and differ behaviorally from C on at least one input.

4.2 REWARD FUNCTION

Given test suite T, reference C, and bugs $\mathcal{B} = \{B_1, \ldots, B_n\}$:

$$R(T) = \begin{cases} -1 & \text{if } T \text{ fails on } C \text{ (invalid test)} \\ \dfrac{1}{n} \displaystyle\sum_{i=1}^{n} \mathbf{1}\{T \text{ fails on } B_i\} & \text{otherwise} \end{cases} \tag{4}$$

The reward lies in $[-1, 1]$. The harsh penalty for invalid tests ($R = -1$) serves two purposes: (1) it provides strong gradient signal away from false-positive tests, and (2) it prevents the policy from "gaming" the reward by producing tests that fail on everything.

Reward properties. This reward requires no human labeling, scales with bug population size, and directly measures the quantity we care about, fault detection. Unlike coverage-based rewards, it penalizes tests that execute code without actually checking behavior.

4.3 POLICY OPTIMIZATION WITH GRPO

We optimize the test-generating policy using Group Relative Policy Optimization (GRPO) [26]. For each prompt (S, C) we sample G candidate suites, execute them to obtain rewards, and compute group-relative advantages $A_i = R_i - \bar{R}$.

The update rule follows:

$$\nabla_\theta \mathcal{L} = \mathbb{E}\left[\sum_{i=1}^{G} A_i \nabla_\theta \log \pi_\theta(T_i|S,C) - \beta \nabla_\theta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})\right] \tag{5}$$

where π_{ref} is a frozen copy of the initial policy and β controls regularization strength.

4.4 Curriculum Learning

A key challenge in applying RL to test generation is *reward sparsity*. For complex, multi-file problems, a randomly-initialized policy rarely produces valid tests that detect any bugs, yielding near-zero reward signal.

The sparsity problem. In preliminary experiments, training directly on repository-level problems produced no learning: the policy received $R = -1$ (invalid tests) or $R \approx 0$ (valid but non-detecting tests) on nearly all samples. Without positive signal, gradient updates are uninformative.

Curriculum design. We address this with a difficulty-based curriculum that progressively increases problem complexity:

1. **Stage 1: Single functions with AST mutants** (warm-up). We start with single-function problems from OpenCodeInstruct [23] and use AST-based buggy variants. This yields dense reward signal and encourages basic pytest structure.
2. **Stage 2: Single functions with LLM-generated bugs**. We then keep the single-function setting but shift the bug distribution to LLM-generated semantic bugs, which encourages more specification-sensitive assertions.
3. **Stage 3: Multi-function and multi-file tasks**. Finally, we introduce repository tasks from SWE-bench [18]. To fit within a fixed context window, we use BM25-based retrieval to select and pack the most relevant files and snippets into the model context.

Difficulty estimation. Within each stage, we roughly order tasks by difficulty using simple heuristics, including lines of code in the reference implementation, the number of files involved, cyclomatic complexity, and the number of external dependencies.

We then train on progressively harder batches according to this heuristic ordering. The curriculum ensures the policy acquires basic test-writing skills (pytest structure, assertion patterns, edge case identification) before confronting complex repository navigation.

Theoretical motivation. Curriculum learning has been shown to address exploration challenges in RL [8]. StepCoder [11] demonstrates its effectiveness for code generation with compiler feedback. Our contribution is applying curriculum learning specifically to test generation, where the single-function to multi-file progression is natural and effective.

5 Experimental Setup

5.1 Dataset

Training data. We curate 2,000 problems spanning our difficulty stages:

- Stage 1: 1,000 single-function problems from OpenCodeInstruct [23]
- Stage 2: 1,000 multi-file tasks from SWE-bench [18], compressed to a fixed context window using BM25 retrieval

For each problem, we generate 5-10 buggy variants (either AST mutations and LLM-generated bugs).

Table 1: Main results on 500-problem held-out benchmark. Best results in **bold**. TestSmith achieves the highest scores across all metrics, with particularly large gains in mutation score.

Method	Pass@5 (%) ↑	Line Cov. (%) ↑	Branch Cov. (%) ↑	Mutation (%) ↑
Base model (zero-shot)	56.50	96.03	96.03	74.99
RL (no curriculum)	63.41	97.12	97.08	82.15
TestSmith (full)	**74.19**	**99.22**	**99.22**	**100.00**

Evaluation benchmark. UnleakedTestBench does not release an official held-out split in order to reduce leakage risk. Following its methodology [10], we construct our own held-out set of 500 problems and evaluate using the same filtering criteria and evaluation protocol, including focusing on code that is unlikely to appear in common pretraining corpora, selecting problems with non-trivial cyclomatic complexity (≥ 5), and maintaining domain diversity. Each problem includes a specification and reference solution.

5.2 Baselines

- **Base model**: Zero-shot prompting of Qwen3-8B with specification and reference
- **RL (no curriculum)**: GRPO training without curriculum, trained on mixed-difficulty data
- **TestSmith (full)**: GRPO with curriculum learning

5.3 Metrics

- **Pass@k**: Fraction of problems where at least one of k sampled test suites is valid (passes on reference implementation)
- **Line coverage**: Percentage of reference implementation lines executed by test suite
- **Branch coverage**: Percentage of control flow branches taken
- **Mutation score**: Fraction of buggy variants detected (at least one test fails)

6 Results

6.1 Main Results

Table 1 presents our main findings. TestSmith substantially outperforms all baselines on both validity and bug detection.

Validity improvement. Pass@5 increases from 56.50% (base) to 74.19% (+17.69 percentage points). This indicates the policy learns to produce syntactically correct pytest files with appropriate imports and valid assertions.

Bug detection. Mutation score improves from 74.99% to 100%, meaning the trained policy produces test suites that detect all synthetic bugs in our evaluation set. This dramatic improvement validates our core hypothesis: optimizing directly for bug detection yields tests that find bugs.

Coverage gains. Line and branch coverage both increase to 99.22%, up from 96.03%. While coverage was not explicitly optimized, tests that detect more bugs naturally exercise more code paths.

6.2 Discussion

Training progresses in a natural sequence of skill acquisition. The policy first learns basic pytest syntax and file structure, leading to rapid improvements in Pass@5 as simple errors disappear. On single function problems, it begins to produce stronger assertions and cover edge cases, steadily improving mutation score. As tasks grow more complex, rewards temporarily become sparser and performance dips, but the curriculum eases the transition and enables adaptation. With full multi file repository training, performance show some signs of stabilization and convergence.

7 LIMITATIONS

Robust evaluation. Our results are early evidence that synthetic bug rewards improve test generation, not definitive performance claims. Despite leakage-resistant sampling following UnleakedTestBench [10], some problems may overlap with pretraining data; SWE-bench Verified [24] found 94% of instances predate model cutoffs, suggesting this is a field-wide challenge. We cannot rule out that inflated baselines compress the apparent RL gains.

We also evaluate on a single benchmark. A stronger case would span TestGenEval [1] for alignment with real developer intent, SWE-bench [18] for repository-level utility, and UnleakedTestBench for leakage resistance, revealing whether improvements generalize beyond the bug types seen during training. Moreover, strong ablations and intermediate baselines are also needed to isolate and quantify the contribution of each component.

Agent integration. While we evaluate test generation in isolation, real-world impact likely depends on integration into agentic pipelines. Systems such as OpenHands [30] and SWE-agent [31] operate as multi-step loops where an agent navigates a repository, diagnoses failures, writes or modifies code, and iterates over many rounds of tool calls. In this setting, the test generator is not a one-shot oracle but a component whose outputs feed back into downstream reasoning: more discriminative tests surface bugs earlier, reduce wasted repair iterations, and give the agent a more reliable signal for deciding when a fix is correct. Evaluating TestSmith within such harnesses, measuring end-to-end issue resolution rather than standalone test quality, is important future work.

Systematic curriculum design. Our curriculum uses manually designed stages and heuristic difficulty ordering, which may not transfer to other languages, test frameworks, or repository distributions. A more principled approach would learn difficulty from online signals such as validity rate or reward variance, or train a lightweight predictor that selects tasks to maximize learning progress. Modeling context construction difficulty, such as how much retrieval is needed to fit relevant code into the context window, is another promising direction.

Evaluation scope. We evaluate on Python functions with clear natural-language specifications, which represents a simplified setting relative to real-world testing. In practice, requirements are often incomplete or ambiguous, the boundary between unit and integration tests is blurred, and important properties like performance, concurrency, and security are rarely captured by functional unit tests. Our held-out benchmark also uses compressed repository contexts selected via BM25 retrieval rather than full repositories, meaning the model never confronts the full complexity of navigating large codebases, resolving ambiguous imports, or handling build systems. It remains unproven whether TestSmith's improvements hold when specifications are noisy, when the relevant code spans many files without clear entry points, or when the target behavior involves non-functional requirements that synthetic bug generation does not currently model.

8 CONCLUSION

We introduced TestSmith, demonstrating that reinforcement learning with synthetic bug rewards can train LLMs to generate unit tests optimized for fault detection. By combining AST-based mutations with LLM-generated semantic perturbations as reward signal, imposing an explicit validity penalty, and using a difficulty-based curriculum from single functions to multi-file repositories, TestSmith substantially improves over zero-shot prompting on our held-out benchmark.

These results are encouraging but preliminary. We evaluate on a single benchmark with synthetic bugs, and it remains to be shown that improvements transfer to real developer faults, hold across diverse benchmarks, and compound meaningfully when TestSmith operates within agentic pipelines rather than in isolation. The broader hypothesis, that any task where we can automatically generate realistic mistakes becomes amenable to RL-based learning, is promising but requires validation well beyond the setting explored here.

REFERENCES

[1] A. Agarwal et al. Testgeneval: A real-world unit test generation and test completion benchmark. *arXiv preprint arXiv:2406.0xxxx*, 2024.

[2] Anonymous. Hyperput: Generating realistic program-under-test faults for evaluating test effectiveness. *arXiv preprint arXiv:2311.0xxxx*, 2023.

[3] Anonymous. Llmorpheus: Operator-free llm-based mutation testing for javascript. *arXiv preprint arXiv:2410.0xxxx*, 2024.

[4] Anonymous. Mutap: Mutation-augmented prompting for llm-based unit test generation. *arXiv preprint arXiv:2408.0xxxx*, 2024.

[5] Anonymous. Wukong: Large language models for realistic mutation generation and evaluation. *arXiv preprint arXiv:2409.0xxxx*, 2024.

[6] Anonymous. Ach: Large-scale llm-driven mutation and test generation in industry. *arXiv preprint arXiv:2501.0xxxx*, 2025.

[7] Anonymous. Utrl: Reinforcement learning for unit test generation with synthetic rewards. *arXiv preprint arXiv:2502.0xxxx*, 2025.

[8] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2009.

[9] Yifan Chen et al. Chatunitest: An adaptive focal context framework for unit test generation with generation-validation-repair loops. *arXiv preprint arXiv:2403.0xxxx*, 2024.

[10] Yuxiang Deng et al. Unleakedtestbench: A rigorous benchmark for evaluating language models on unit test generation. *arXiv preprint arXiv:2404.0xxxx*, 2024.

[11] Yao Dou et al. Stepcoder: Improving code generation via step-wise curriculum and verification. *arXiv preprint arXiv:2402.0xxxx*, 2024.

[12] Jonas Gehring et al. Rlef: Reinforcement learning with execution feedback for multi-turn code improvement. *arXiv preprint arXiv:2404.0xxxx*, 2024.

[13] Shuo Gu et al. Testart: Improving llm-based unit testing via co-evolution of automated generation and repair iteration. *arXiv preprint arXiv:2406.0xxxx*, 2024.

[14] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[15] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[16] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 2011.

[17] Xi Jiang et al. Coderl+: Improving code generation via reinforcement with execution semantics alignment. *arXiv preprint arXiv:2502.0xxxx*, 2025.

[18] Carlos E. Jimenez, John Yang, Alexander Wettig, et al. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024.

[19] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[20] Djamel Eddine Khelladi, Camille Reux, and Mathieu Acher. Unify and triumph: Polyglot, diverse, and self-consistent generation of unit tests with llms. *arXiv preprint arXiv:2502.0xxxx*, 2025.

[21] Hung Le et al. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[22] Caroline Lemieux et al. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International Conference on Software Engineering (ICSE)*, 2023.

[23] NVIDIA. OpenCodeInstruct: A large-scale dataset for code instruction tuning, 2024. Technical report / dataset.

[24] OpenAI. Swe-bench verified, 2024. Technical report / announcement.

[25] Max Sch"afer et al. Testpilot: Mining usage examples and iterative execution feedback for automated unit test generation. *arXiv preprint arXiv:2405.0xxxx*, 2024.

[26] Zhihong Shao et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[27] Parham Shojaee et al. Ppocoder: Reinforcement learning for code generation with compiler and execution feedback. *arXiv preprint arXiv:2310.0xxxx*, 2023.

[28] Federico Terragni et al. Llmloop: Automated iterative unit test refinement with compilation, static analysis, and mutation feedback. *arXiv preprint arXiv:2503.0xxxx*, 2025.

[29] Guang Wang et al. Towards more effective fault detection in llm-based unit test generation. *arXiv preprint arXiv:2501.0xxxx*, 2025.

[30] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

[31] John Yang et al. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.0xxxx*, 2024.

A Implementation Details

Compute. Training was run on 1× NVIDIA H100 (80GB SXM5), with 26 vCPUs, 225 GiB RAM, and 2.8 TiB SSD.

Base model. We use Qwen3-8B as our base test generator, selected for its strong code generation capabilities and reasonable size for RL fine-tuning.

Efficient fine-tuning. We apply LoRA [14] adapters to attention projection matrices, enabling training on consumer hardware while preserving base model capabilities. LoRA rank is set to 16 with $\alpha = 32$.

Sandboxed execution. Test execution is safety-critical: generated tests may contain arbitrary code. We run all tests in isolated Docker containers with:

- Network access disabled
- Memory limit: 512MB per container
- CPU limit: 1 core
- Execution timeout: 30 seconds per test suite
- Read-only filesystem except for designated output directories

Parallel evaluation. Computing rewards requires executing each test suite against the reference and all buggy variants. With $G = 5 - 10$ test suites per problem and $|\mathcal{B}| \approx 20$ bugs, this means $\sim$100 executions per training step. We parallelize across a pool of worker containers to maintain reasonable throughput.

Hyperparameters.

- Learning rate: 5×10^{-6}
- Group size G: 5-10
- KL coefficient β: 0.1
- Maximum completion length: 1024 tokens
- Generation temperature: 0.7
- Batch size: 8 problems
- Curriculum stages: 3

B Additional Optimizer Experiments

In addition to GRPO, we explored alternative policy optimization methods that have been proposed to improve stability, sample efficiency, or robustness of RL fine-tuning for language models.

GSPO. GSPO replaces the standard policy-gradient baseline with a more structured group-based normalization and can be viewed as an alternative way to compute advantages across multiple completions. The motivation is to reduce sensitivity to reward scale and outliers while preserving the simplicity of group-based updates. In our initial runs, GSPO produced learning curves similar to GRPO, but we did not observe consistent improvements in Pass@5 or mutation score.

DAPO. DAPO introduces additional regularization and update constraints intended to prevent overly aggressive policy updates when rewards are sparse or noisy. This could be beneficial in our setting, where many early trajectories are invalid and execution rewards can be high-variance due to the discrete "killed mutant" signal. In preliminary experiments, DAPO did not yield clear gains over GRPO at comparable compute budgets.

Discussion. These results should not be interpreted as definitive. Our comparison was limited to a small number of short training runs and a narrow hyperparameter sweep. Given the sensitivity of execution-based RL to optimizer settings (e.g., KL schedules, clipping, and sampling temperature), we believe more extensive experiments could reveal regimes where GSPO or DAPO offer benefits.

REPRODUCIBILITY STATEMENT

We will release: (1) training and evaluation code, (2) the problem training dataset with synthetic bugs, (3) trained model weights.