

Code-A1: Adversarial Evolving of Code LLM and Test LLM via Reinforcement Learning

Aozhe Wang^{1,*}, Yuchen Yan^{1,*}, Nan Zhou^{1,*}, Zhengxi Lu^{1,*}, Hong Zhao², Yao Fu², Weihao Jiang², Weiming Lu¹, Jun Xiao¹, Yueting Zhuang¹, Yongliang Shen^{1,†}

¹Zhejiang University

²Hikvision Research Institute, Hangzhou, China

*Equal contribution

†Corresponding author

Correspondence: waz@zju.edu.cn, yanyuchen@zju.edu.cn, syl@zju.edu.cn

Abstract

Reinforcement learning for code generation relies on verifiable rewards from unit test pass rates. Yet high-quality test suites are scarce, existing datasets offer limited coverage, and static rewards fail to adapt as models improve. Recent self-play methods unify code and test generation in a single model, but face an inherent dilemma: white-box access leads to self-collusion where the model produces trivial tests for easy rewards, yet black-box restriction yields generic tests that miss implementation-specific bugs. We introduce Code-A1, an adversarial co-evolution framework that jointly optimizes a Code LLM and a Test LLM with opposing objectives. The Code LLM is rewarded for passing more tests, while the Test LLM is rewarded for exposing more defects. This architectural separation eliminates self-collusion risks and safely enables white-box test generation, where the Test LLM can inspect candidate code to craft targeted adversarial tests. We further introduce a Mistake Book mechanism for experience replay and a composite reward balancing test validity with adversarial difficulty. Experiments on Qwen2.5-Coder models demonstrate that Code-A1 achieves code generation performance matching or exceeding models trained on human-annotated tests, while significantly improving test generation capability.

1 Introduction

Reinforcement learning has emerged as a powerful paradigm for enhancing the reasoning capabilities of large language models (Yan et al., 2026), with code generation serving as a representative task that admits precise, executable verification (Guo et al., 2025). Unlike open-ended text generation, code can be automatically validated against unit tests, providing verifiable rewards that guide policy optimization without human annotation at training time (Le et al., 2022; Shojaee et al., 2023).

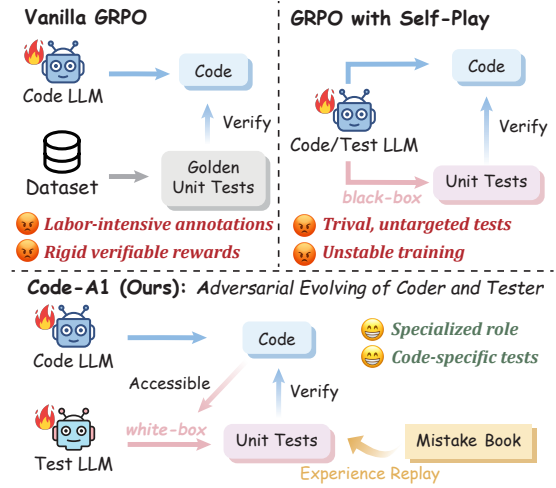


Figure 1: **Comparison of three training paradigms for code generation.** *Vanilla GRPO* relies on static golden tests with rigid rewards. *GRPO with Self-Play* unifies code and test generation but must operate in black-box mode to prevent self-collusion. **Code-A1** decouples the two tasks into models with opposing objectives, safely enabling white-box test generation and stable co-evolution via the Mistake Book.

However, the effectiveness of this paradigm hinges critically on the quality of the underlying test suites. In practice, obtaining comprehensive unit tests demands substantial human effort, and existing RL-suitable datasets remain limited in both scale and diversity (Chen et al., 2021; Austin et al., 2021). Even in carefully curated benchmarks, each question typically contains only three to five test cases, which cannot reliably distinguish genuinely correct solutions from those that happen to pass by coincidence or handle only common inputs (Liu et al., 2023b). Furthermore, these static golden tests cannot adapt to evolving model capabilities. When tests are overly simple, flawed code may receive undeserved positive rewards; when tests are overly stringent, near-correct solutions are penalized as complete failures (Jeong et al.). Both scenarios distort the learning signal and limit the

potential of RL-based training.

To address the limitations of static rewards, recent work has explored automated test generation (Chen et al.; Zeng et al., 2025) and self-play frameworks (Wang et al., 2025; Zhao et al., 2025) where a single model generates both code and tests. These approaches promise dynamic rewards that adapt to model capabilities, potentially breaking free from static constraints. Yet they fail to deliver on this promise. Direct generation methods often produce invalid or hallucinated tests (He et al., 2025). Self-play faces a more fundamental dilemma: when restricted to black-box mode (observing only question descriptions), tests remain generic and miss implementation-specific bugs; when permitted white-box access to candidate code, the model exploits this through self-collusion, generating trivial tests for easy rewards since passing code offsets penalties for weak testing within a unified model (Denison et al., 2024). To prevent collusion, self-play must restrict test generation to black-box mode, sacrificing the ability to craft targeted tests that probe implementation-specific bugs. This black-box restriction fundamentally undermines dynamic adaptation, as test difficulty becomes decoupled from the actual code being evaluated.

This analysis reveals a key insight: ***effective verifiable rewards require dynamic interplay between code robustness and test rigor, where test difficulty continuously adapts to challenge the current policy.*** A good test suite must be valid (executable and correct), sufficiently challenging (exposing real defects), yet not impossibly difficult (providing learnable gradients). These competing objectives cannot be optimized by a single model or static dataset—they require *adversarial co-evolution* (Wang et al., 2026) where two specialized agents continuously push each other toward improvement, with architectural separation that prevents self-collusion while enabling targeted adversarial optimization.

We introduce **Code-A1**, an adversarial reinforcement learning framework that jointly optimizes a Code LLM and a Test LLM with opposing objectives. Given a question, the Code LLM generates candidate solutions while the Test LLM generates challenging test cases; the two are paired and executed in a sandbox. The Code LLM receives higher rewards for passing more tests, incentivizing robust and correct solutions. The Test LLM receives higher rewards for *failing* the code, incentivizing the discovery of edge cases and subtle bugs. As both models improve, rewards dynam-

ically adapt: stronger code demands harder tests, and harder tests demand stronger code. This adversarial yet complementary setup enables continuous co-evolution beyond static performance ceiling.

To stabilize this adversarial dynamics, we introduce several key designs. First, we decouple the two tasks into separate models, eliminating the self-collusion risks inherent in self-play and enabling safe white-box adversarial optimization. Second, we design a composite reward for the Test LLM that balances *validity* (tests must execute correctly) with *adversarial difficulty* (tests should expose defects), avoiding both trivial and impossible tests. Third, we maintain a *Mistake Book*—an experience replay buffer that records historically failed tests for each question—ensuring that resolved bugs are not forgotten and providing stable baselines for reward computation.

We conduct extensive experiments on Qwen2.5-Coder models (1.5B, 3B, 7B). On code generation benchmarks, Code-A1 consistently outperforms both models trained on human-annotated golden tests and self-play baselines across all scales. On test generation, the results reveal remarkable efficiency: the 3B model achieves a Mul score of 15.29, surpassing the 7B base model (14.72), demonstrating that adversarial co-evolution discovers bug-revealing patterns more effectively than parameter scaling alone.

Our contributions can be summarized as follows:

- We introduce adversarial co-evolution into code RL, enabling dynamic and adaptive verifiable rewards that eliminate reliance on static human-annotated test suites.
- We develop Code-A1, comprising dual-policy optimization with opposing objectives, validity-aware reward shaping for the Test LLM, and a Mistake Book mechanism for stable experience replay.
- We demonstrate empirically that Code-A1 matches or exceeds the performance of RL with static golden tests on code generation benchmarks, while simultaneously producing a Test LLM capable of generating high-quality and bug-revealing tests.

2 Related Works

2.1 RL for Code Reasoning

Reinforcement Learning improves code generation by optimizing executable correctness rather than

token-level similarity. Since code can be compiled and tested, compiler feedback and unit-test outcomes naturally serve as reward signals. Early work adopted Actor-Critic training with dense rewards from test pass rates (Le et al., 2022), while later PPO-based methods incorporated compilation and test pass rates with KL constraints for stable optimization (Zhang et al., 2024a; Shojaei et al., 2023). To mitigate sparse binary rewards, RLTF introduces multi-level feedback from compilation, runtime, and logic errors (Liu et al., 2023a). Ranking-based alignment further reduces the cost and instability of online RL by selecting candidate programs according to test outcomes without an explicit reward model, though reward quality remains bounded by the coverage and adaptability of the underlying tests (Shen et al., 2023).

2.2 Unit Test Generation for Code Reasoning

LLMs increasingly synthesize unit tests to reduce manual cost and improve scalability (Yang et al., 2025; Tip et al., 2025). Prior work generate tests from problem descriptions and filter them with stronger proxy solutions (Zeng et al., 2025; Jain et al., 2025a), or construct harder tests such as “hacking inputs” and test-generator programs validated by oracles (He et al., 2025; Altmayer Pizzorno and Berger, 2025; Hossain and Dwyer, 2025). In recent co-evolutionary frameworks, models act as both code and test generators, but description-only tests may miss implementation-specific bugs while white-box unified self-play risks self-collusion (Wang et al., 2025; Zhao et al., 2025; Chen et al., 2025; Lu et al., 2025). These synthesized tests support both RL training, where pass rates provide scalable rewards (Zhang et al., 2024b), and inference-time self-verification, such as Best-of-N selection and iterative debugging or refinement (Wang et al., 2025; Lee et al., 2026).

3 Methods

In this section, we present the Code-A1 framework for adversarial co-evolution of code and test generation. We begin with problem formulation and output format constraints (Section 3.1). We then describe the adversarial rollout procedure (Section 3.2) and the Mistake Book mechanism for experience replay (Section 3.3). Finally, we detail the reward design (Section 3.4) and policy optimization (Section 3.5).

3.1 Problem Formulation

We consider function-level code generation, where a model receives a question description Q containing a function signature and natural language specification, and outputs a complete function body. Let π_C denote Code LLM and π_T denote Test LLM. Given a dataset $\mathcal{D} = \{(Q_i, C_i)\}_{i=1}^{|\mathcal{D}|}$ where C_i is the ground-truth solution, our goal is to jointly optimize both models through adversarial interaction.

The Code LLM generates candidate solutions $\hat{C} \sim \pi_C(\cdot | Q)$ that should be syntactically correct and satisfy the specification. The Test LLM generates a set of K test cases $\hat{T} = \{\hat{t}_1, \dots, \hat{t}_K\} \sim \pi_T(\cdot | Q, \hat{C})$ conditioned on both the question and a candidate solution. Each test case follows the assertion format `assert func(*args) == answer`, where `func` is the target function, `*args` are input arguments, and `answer` is the predicted output. This structured format enables reliable extraction via abstract syntax tree parsing, reducing reward noise from formatting errors. We require exactly K test cases per response to prevent trivial convergence to single-test outputs and discourage brute-force generation for reward hacking.

3.2 Adversarial Rollout

At each training step, we sample a batch of questions from $\mathcal{D}$ and perform adversarial rollout:

Step 1: Code Generation. For each question Q , the Code LLM generates M solutions $\{\hat{C}_1, \dots, \hat{C}_M\}$ via sampling.

Step 2: Test Generation. For each candidate solution $\hat{C}_m$, the Test LLM generates N test suites $\{\hat{T}_{m,1}, \dots, \hat{T}_{m,N}\}$, conditioned on both Q and $\hat{C}_m$. Conditioning on the candidate solution enables the Test LLM to craft targeted tests that probe potential weaknesses.

Step 3: Test Validation. We extract function calls from each generated test and execute them against the ground-truth solution C_{GT} . A test is deemed *valid* if: (i) the function call executes without error, (ii) the call is unique within the test suite, and (iii) the predicted answer is correct. For tests with incorrect predicted answers, we replace the prediction with the ground-truth return value, retaining the test to enrich coverage. Other Invalid tests are discarded.

Step 4: Code Evaluation. We concatenate each candidate solution $\hat{C}_m$ with the validated test suites

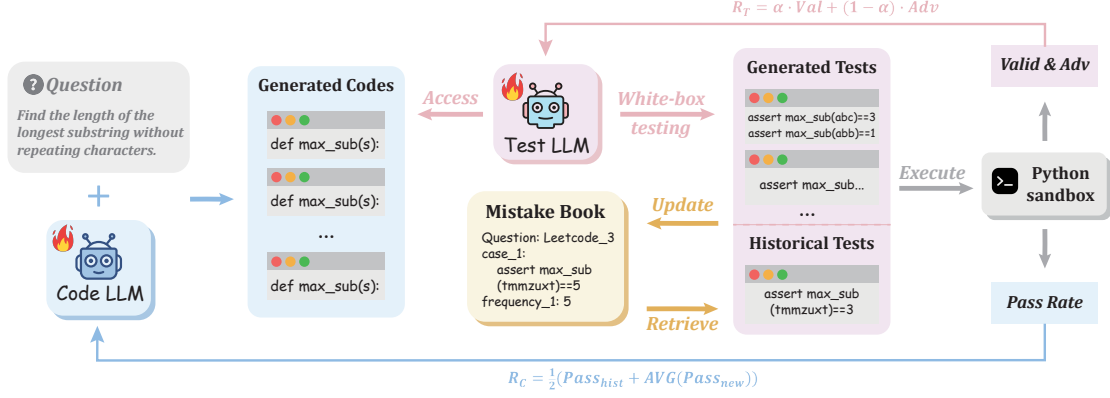


Figure 2: **Overview of the Code-A1 training framework.** The Code LLM generates candidate solutions accessible to the Test LLM for white-box test generation. Generated tests are validated and merged with historical failures from the Mistake Book. The Code LLM is rewarded for passing more tests; the Test LLM is rewarded for exposing more defects, enabling curriculum-aware adversarial co-evolution.

and execute in a sandboxed environment. The pass rate serves as the basis for reward computation.

3.3 Mistake Book

A key challenge in adversarial training is instability: a temporarily weak Test LLM may generate trivial tests, providing inflated rewards that mislead the Code LLM. Conversely, a strong Test LLM may generate tests so difficult that learning signals vanish. To stabilize training and track capability evolution, we introduce the *Mistake Book*, a per-question experience replay buffer (Zhan et al., 2025).

Structure. The Mistake Book $\mathcal{B}$ maintains a mapping from each question Q_i to a set of historically failed tests T_i^{hist} .

Update Rule. After each training step, we update $\mathcal{B}$ dynamically: newly generated tests that the current candidate solutions fail (NewFails) are added to T_i^{fail} , while historical tests that the current solutions now pass (NewPasses) are removed. This ensures that $\mathcal{B}$ reflects the frontier of model capability, containing exactly those tests that remain challenging given current Code LLM performance.

Role in Training. The Mistake Book serves three purposes. First, historical tests provide a stable baseline for reward computation, reducing variance caused by stochastic test generation. Second, the gap between historical and new test pass rates provides a curriculum signal that reveals whether the Test LLM is generating progressively harder tests. Third, re-evaluating against historical failures prevents forgetting, ensuring that previously fixed bugs are not reintroduced as training proceeds.

3.4 Reward Design

We assign rewards at the trajectory level, with opposing objectives for the two models.

Code LLM Reward. The Code LLM should produce solutions that are correct and robust. We evaluate code and compute reward $R_C(\hat{C}_m)$ based on the pass rate of two test sources: historical failures from the Mistake Book and newly generated tests from the Test LLM (see details in Appendix A.3).

Test LLM Reward. The Test LLM faces a tension: tests must be *valid* (syntactically executable, correct and unique) yet *adversarial* (capable of exposing defects). We design a composite reward $R_T(\hat{T})$ to balance these objectives (see details in Appendix A.3): The validity reward $R_T^{\text{val}}(\hat{T})$ measures the fraction of generated tests that pass validation, implicitly encouraging format compliance; the adversarial reward $R_T^{\text{adv}}(\hat{T})$ measures whether new tests are harder than historical ones.

3.5 Policy Optimization

We adopt Group Relative Policy Optimization (GRPO) (Shao et al., 2024) with token-level loss aggregation (Yu et al., 2025) for both models.

Asymmetric Sampling. The Code LLM generates M solutions per question, while the Test LLM generates $M \times N$ test suites (N suites per solution). To balance training compute between models, we select only the top- ℓ test suite groups with highest reward variance for the Test LLM update (see details in Appendix A.5.3), setting $\ell \times N = M$. This prioritizes high-learning-value samples while maintaining synchronized training steps.

Algorithm 1 Code-A1

Require: Coder π_C , Tester π_T , dataset $\mathcal{D}$, Mistake Book $\mathcal{B}$

```
1: Initialize Mistake Book  $\mathcal{B} \leftarrow \emptyset$ 
2: while not converged do
3:   for  $(Q, C_{GT}) \in \mathcal{D}$  do
4:     // Rollout
5:      $\{\hat{C}_m\}_{m=1}^M \sim \pi_C(\cdot \mid Q)$ 
6:     // Retrieve Historical Failures
7:      $T^{\text{hist}} \leftarrow \mathcal{B}(Q)$ 
8:     // Test Generation & Validation
9:     for  $m = 1$  to  $M$  do
10:       $\{\hat{T}_{m,n}\}_{n=1}^N \sim \pi_T(\cdot \mid Q, \hat{C}_m)$ 
11:       $T_{m,n} \leftarrow \text{VALIDATE}(\hat{T}_{m,n}, C_{GT})$ 
12:    end for
13:     $\text{Pass}^{(m)} \leftarrow \text{EXECUTE}(\hat{C}_m, \{T_{m,n}\}_{n=1}^N \cup T^{\text{hist}})$ 
14:    // Reward Computation
15:     $R_C^{(m)} \leftarrow f(\text{Pass}_{\text{hist}}^{(m)}, \text{Pass}_{\text{new}}^{(m)})$ 
16:     $R_T^{(m,n)} \leftarrow \alpha \cdot \text{Valid} + (1 - \alpha) \cdot \text{Adv}$ 
17:    // Update Mistake Book
18:     $\mathcal{B}(Q) \leftarrow \mathcal{B}(Q) \cup \text{NewFails} \setminus \text{NewPasses}$ 
19:    // Policy Update via GRPO
20:    Update  $\pi_C$  with  $\{R_C^{(m)}\}$ 
21:    Update  $\pi_T$  with  $\text{TOPVAR}(\{R_T^{(m,n)}\})$ 
22:  end for
23: end while
```

Training Dynamics. The adversarial setup creates a natural curriculum. In early training, both models have limited capability, and the Test LLM generates simple tests that provide achievable targets for the Code LLM without overwhelming gradients. As training progresses, the Code LLM improves and passes most historical tests, forcing the Test LLM to generate harder tests to earn rewards, which in turn raises the bar for the Code LLM. Eventually, the two models reach an equilibrium where further improvement requires genuine capability gains rather than exploitation of weak opponents. By decoupling code and test generation into separate models with opposing objectives, Code-A1 avoids the reward hacking risks inherent in white-box self-play while enabling continuous co-evolution.

4 Experiments

4.1 Experimental Setup

Implementation. We use Qwen2.5-Coder-Instruct models (Hui et al., 2024) at three scales (1.5B, 3B, 7B) as base models. The Code LLM and Test LLM are initialized from the same checkpoint and trained jointly on 9,688 hard-difficulty questions from KodCode-V1 (Xu et al., 2025). For the Test LLM, we apply supervised fine-tuning before RL to establish the assertion format. During rollout, both models sample 8 responses per question with temperature 1.0. We set $\ell = 1$ for

the Test LLM to balance training compute and $\alpha = 0.5$ for the validity-adversarial trade-off. Training runs for 111 steps with GRPO. Additional implementation details including prompt design, sandbox configuration, and Mistake Book structure are provided in Appendix A.

Baselines. We compare Code-A1 against two groups of baselines. For code generation, we consider: **Base**, the original Qwen2.5-Coder-Instruct without RL; **Golden Tests**, the Code LLM trained via GRPO using human-annotated tests as verifiable rewards; and **Self-Play**, which employs a single model for both code and test generation, with input isolation restricting test generation to question descriptions only to prevent reward hacking. For test generation, we additionally include **SFT**, which trains the Test LLM with supervised fine-tuning only. Implementation details for Self-Play and SFT are provided in Appendix A.5.4 and A.5.5.

Evaluation. We evaluate code generation on HumanEval⁺ (Liu et al., 2023b), MBPP⁺ (Liu et al., 2023b), and BigCodeBench (Zhuo et al., 2025), and test generation on UnLeakedTestBench (Huang et al., 2025). We sample 32 responses for Code LLM and 5 for Test LLM, with temperature 0.7 and top- p 0.95. We report avg@32 for code generation, and pass@ k (test accuracy) and mut@ k (mutation score) for test generation. To assess comprehensive performance, we additionally introduce Avg (the mean of code generation scores) and Mul ($\text{pass}@k \times \text{mut}@k$), a composite metric balancing test validity and adversarial power (Appendix A.7.4).

4.2 Main Results

Code Generation. Table 1 presents results across three model scales. Code-A1 consistently achieves the highest average scores, outperforming both the Golden Tests baseline trained on human annotations and the Self-Play approach. The advantage is most pronounced at smaller scales: on the 1.5B model, Code-A1 achieves 56.95% average accuracy compared to 56.23% for Golden Tests and 55.88% for Self-Play. This gap stems from the fundamental difference in testing paradigms. Self-Play must operate in black-box mode to prevent self-collusion, generating tests solely from question descriptions. In contrast, Code-A1’s decoupled architecture safely enables white-box testing, where the Test LLM inspects candidate code to craft targeted adversarial tests. This produces richer, on-

Code LLM	Method	HumanEval ⁺ avg@32	MBPP ⁺ avg@32	BigCodeBench avg@32	Avg
Qwen2.5-Coder-1.5B-Ins	/	63.42	60.87	29.34	51.21
	Golden Tests	<u>71.15</u>	63.30	<u>34.23</u>	<u>56.23</u>
	Self-Play	70.64	63.54	33.47	55.88
	Code-A1	72.69	<u>63.33</u>	34.82	56.95
Qwen2.5-Coder-3B-Ins	/	77.63	63.12	41.78	60.84
	Golden Tests	<u>81.96</u>	<u>68.05</u>	<u>45.41</u>	<u>65.14</u>
	Self-Play	81.86	67.06	45.09	64.67
	Code-A1	83.52	69.07	45.85	66.15
Qwen2.5-Coder-7B-Ins	/	83.69	71.95	49.41	68.35
	Golden Tests	84.68	74.16	<u>52.28</u>	70.37
	Self-Play	<u>84.70</u>	<u>74.23</u>	52.25	<u>70.39</u>
	Code-A1	85.21	74.50	52.46	70.72

Table 1: **Performance comparison of Code LLMs.** The best results are highlighted in bold, and the second best results are underlined.

Test LLM	Method	UnLeakedTestBench						Mul
		pass@1	pass@2	pass@5	mut@1	mut@2	mut@5	
Qwen2.5-Coder-1.5B-Ins	/	18.34	16.91	16.29	17.10	20.35	22.30	3.63
	SFT	16.01	15.16	14.76	14.53	27.57	29.45	4.35
	Self-Play	24.12	23.37	23.39	14.82	26.98	28.91	6.76
	Code-A1	27.42	25.68	27.05	17.31	23.57	26.41	7.14
Qwen2.5-Coder-3B-Ins	/	19.77	20.50	20.93	28.35	29.99	42.55	8.91
	SFT	25.40	25.71	23.51	26.77	29.08	36.29	8.53
	Self-Play	25.84	29.65	29.64	29.66	38.88	50.92	<u>15.09</u>
	Code-A1	27.83	30.31	30.86	26.44	36.96	49.56	15.29
Qwen2.5-Coder-7B-Ins	/	27.71	28.64	28.73	30.36	36.07	51.25	14.72
	SFT	29.32	29.94	28.72	31.33	41.01	50.85	14.60
	Self-Play	32.98	34.74	35.13	31.42	38.54	55.57	<u>19.52</u>
	Code-A1	36.79	37.87	37.15	35.50	36.60	53.14	19.74

Table 2: **Performance comparison of Test LLMs.** The best results are highlighted in bold, and the second best results are underlined.

policy reward signals that drive the Code LLM to develop robustness against precise vulnerabilities rather than generic edge cases. We conduct a further comparison with CURE (Wang et al., 2025) and UTRL (Lee et al., 2026) in Appendix C.

Test Generation. Table 2 reveals that Code-A1 significantly outperforms both Base and SFT models on test generation, with the Test LLM evolving from simple validity to high discriminatory power. The results demonstrate a striking efficiency gain: the 3B model trained with Code-A1 achieves a Mul score of 15.29, surpassing the 7B Base model (14.72). This indicates that adversarial co-evolution discovers bug-revealing patterns more effectively than parameter scaling alone. Unlike SFT which produces valid but generic tests, Code-A1 drives the Test LLM to synthesize targeted tests that expose implementation-specific weaknesses. The consistent improvement across all scales (1.5B: 7.14 vs 4.35, 3B: 15.29 vs 8.53, 7B: 19.74 vs 14.60

in Mul) confirms that dynamic adversarial pressure is essential for developing discriminatory test generation capability.

4.3 Ablation Studies

Trade-off Weight α . The weight α in Eq. 4 balances validity and adversarial difficulty in the Test LLM reward. As shown in Figure 3, $\alpha = 0.5$ achieves optimal performance on both Code LLM (56.95% Avg) and Test LLM (7.14 Mul) metrics. When α is too low (e.g., $\alpha = 0$), the reward overemphasizes difficulty while neglecting correctness. Although our mechanism corrects erroneous predictions using ground truth, the lack of intrinsic correctness incentives causes policy entropy to spike; the Test LLM generates challenging tests by chance rather than learning robust patterns, resulting in unstable training and poor Mul (3.94). Conversely, when α is too high (e.g., $\alpha = 1.0$), the model rapidly converges to trivial tests that provide insufficient learning gradients for the Code LLM.

Settings	HumanEval ⁺	MBPP ⁺	BigCodeBench	Avg	UnLeakedTestBench		Mul
	avg@32	avg@32	avg@32		pass@5	mut@5	
Code-A1	72.69	63.33	34.82	56.95	27.05	26.41	7.14
- w/o predicted answer	<u>70.69</u>	64.47	<u>34.60</u>	<u>56.59</u>	11.18	21.72	2.43
- w/o referable code	68.66	62.99	33.34	55.00	23.77	23.52	5.59
- w/o Mistake Book	69.66	<u>63.94</u>	33.73	55.78	25.50	27.41	<u>6.99</u>

Table 3: **Ablation study on Qwen2.5-Coder-1.5B-Instruct.** Removing referable code, predicted answer, or Mistake Book each degrades performance, confirming the contribution of white-box access, multi-objective optimization, and experience replay.

The balanced setting $\alpha = 0.5$ maintains stability while preserving adversarial pressure.

Multi-Objective Optimization. We investigate whether removing answer prediction simplifies optimization. In the “w/o predicted answer” setting (Table 3), the Test LLM generates only function calls, relying on oracles for expected outputs. While this accelerates convergence toward difficult tests, it fails to significantly improve Code LLM performance and severely degrades test generation capability (pass@5 drops to 11.18%, below the Base model). This confirms that requiring the Test LLM to predict answers serves dual purposes: preventing excessively hard tests that eliminate learning gradients, and maintaining the Test LLM’s intrinsic capability to synthesize complete unit tests.

Conditioning on Candidate Code. Removing candidate code from the Test LLM input (“w/o referable code”) causes HumanEval⁺ to drop from 72.69% to 68.66%. This ablation directly demonstrates the value of white-box access. Black-box generation yields broad tests that miss implementation-specific bugs. In Code-A1, the decoupled architecture transforms white-box access from a collusion risk into a debugging advantage: the Test LLM inspects code structure to craft on-policy tests targeting specific logical weaknesses, providing sharper curriculum-like signals that black-box methods cannot replicate.

Mistake Book. Ablating the Mistake Book degrades Code LLM performance (HumanEval⁺ drops from 72.69% to 69.66%) while minimally affecting Test LLM (Mul: 7.14 $\rightarrow$ 6.99). This asymmetry reveals the mechanism’s primary role: preventing catastrophic forgetting in the Code LLM. Without re-evaluation against historical failures, the model regresses on previously resolved bugs while overfitting to current adversarial attacks. Figure 5 further illustrates this dynamic: experience replay forces the Code LLM to maintain competence on

historical challenges while adapting to new adversarial tests (see Appendix A.4 for details).

5 Analysis

Generated Tests as Golden Tests. To directly assess test quality, we use tests generated by different models (Base, SFT, Code-A1) as static golden tests for standard RLVR training. As shown in Table 4, the Code LLM trained with Code-A1-generated tests achieves 56.75% average accuracy, surpassing both human annotations (56.23%) and SFT-generated tests (56.22%). On HumanEval⁺ and MBPP⁺, Code-A1-guided training reaches 71.67% and 64.42%, exceeding the human annotation baseline (71.15% and 63.30%). This result demonstrates that Code-A1 synthesizes tests of sufficient quality to replace expensive manual annotation, validating adversarial co-evolution as a scalable alternative to human supervision.

Training Dynamics. Figure 4 reveals the co-evolution process under different settings. For the Code LLM (Figure 4a), Code-A1 with $\alpha = 0.5$ closely tracks the Golden Tests baseline, indicating that dynamically generated tests provide learning signals equivalent to human annotations. Extreme α values lead to suboptimal trajectories: $\alpha = 0$ causes erratic learning due to unstable test generation, while $\alpha = 1.0$ results in inflated rewards from trivial tests that fail to challenge the model. For the Test LLM (Figure 4b), $\alpha = 0.5$ achieves the highest sustained reward, reflecting the emergence of tests that are both valid and adversarially effective. This dynamic equilibrium confirms that Code-A1 successfully balances the competing objectives, enabling both models to improve synchronously without manual reward calibration.

Test-Time Scaling. Beyond training, we evaluate whether Code-A1 models collaborate effectively at inference time. Table 5 shows results under parallel test-time scaling, where Code LLM generates

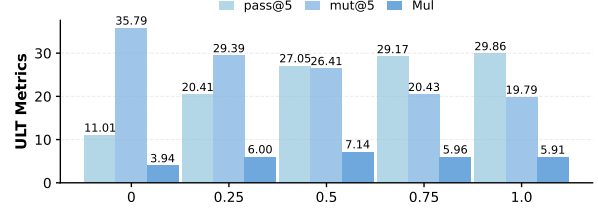
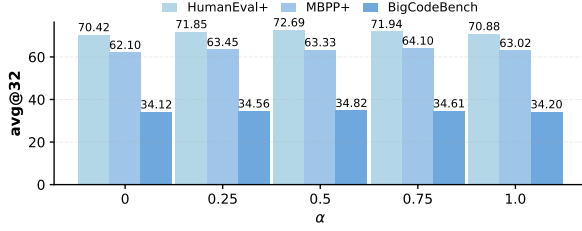


Figure 3: Impact of trade-off weight α on model capabilities using Qwen2.5-Coder-1.5B-Instruct. Left: Code generation performance (avg@32) on three benchmarks. Right: Test generation metrics on UnLeakedTestBench.

Golden Test Source	HE ⁺ avg@32	MBPP ⁺ avg@32	BCB avg@32	Avg
Human Annotation	71.15	63.30	34.23	56.23
base Test LLM	69.68	63.66	32.85	55.40
SFT Test LLM	71.49	64.24	32.93	56.22
Code-A1 Test LLM	71.67	64.42	<u>34.17</u>	56.75

Table 4: Performance comparison of Code LLMs on three benchmarks using Qwen2.5-Coder-1.5B-Instruct as the base model and employing Golden Tests from various sources for RLVR. The best results are highlighted in bold, and the second best results are underlined.

multiple candidates and Test LLM selects the best. Code-A1 Test LLM demonstrates superior discriminatory power: when verifying Base Code LLM, it achieves 65.82% compared to 65.13% with Base self-verification. The full combination (Code-A1 + Code-A1) yields 67.81%, the highest overall performance. Notably, while Base Test LLM improves Code-A1 Code LLM (66.15% to 67.45%), it cannot fully exploit its potential. This confirms that high-quality adversarial tests are essential for distinguishing subtle correctness differences in optimized solutions, ensuring verification capability scales with generation capability.

Parallel TTS		HE ⁺	MBPP ⁺	BCB	Avg
Code LLM	Test LLM				
Base	/	77.63	63.12	41.78	60.84
Code-A1	/	83.52	69.07	45.85	66.15
Base	Base	82.32	70.63	42.45	65.13
Code-A1	Base	84.76	71.69	45.91	67.45
Base	Code-A1	82.93	70.90	43.63	65.82
Code-A1	Code-A1	85.37	71.96	46.09	67.81

Table 5: Performance of different combinations under parallel test-time scaling (see details in Appendix A.6).

Test Difficulty & Diversity by Case Study. We examine the “threeSum” problem (Appendix B) to illustrate how test quality evolves. The Code LLM’s initial solution passes both human-annotated golden tests and pre-training Test LLM outputs, yet contains a subtle bug: failure to handle

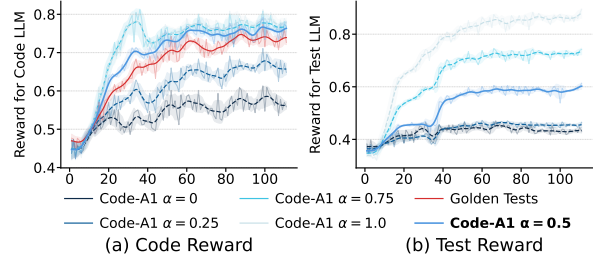


Figure 4: Training dynamics (Code Reward and Test Reward) over training steps for (a) Code LLM and (b) Test LLM.

duplicate triplets when moving pointers. As training progresses, the Test LLM learns to target this blind spot, generating edge cases with highly repetitive elements (e.g., $[-2, 1, 1, 1, 1]$) that expose the flaw. This demonstrates the curriculum-like progression: the Test LLM evolves from generic inputs to targeted adversarial attacks that probe implementation-specific vulnerabilities.

6 Conclusion

In this paper, we introduced Code-A1, a novel reinforcement learning framework that orchestrates the adversarial co-evolution of a Code LLM and a Test LLM. By decoupling code and test generation into two models with opposing objectives, Code-A1 circumvents the limitations arising from self-play to avoid self-collusion and safely enables white-box testing. A Mistake Book mechanism stabilizes training by maintaining historical failures as experience replay, preventing catastrophic forgetting while providing stable baselines for reward computation. Experiments demonstrate that Code-A1 matches or exceeds RL with human-annotated golden tests on code generation, while producing a Test LLM that generates high-quality, bug-revealing tests more effectively than supervised fine-tuning or parameter scaling. These results suggest that adversarial co-evolution offers a scalable path beyond annotation-dependent training.

Limitations

Dependence on Ground Truth Code. Code-A1 requires ground truth solutions to validate generated tests during training. This limits its applicability to domains where reference implementations are unavailable.

Test Format Constraints. The current framework requires tests to follow an assertion format for reliable extraction and execution. This excludes more complex testing scenarios, such as stateful interactions, I/O-based testing, and property-based testing.

Limited Generalization Scope. We primarily evaluate Code-A1 on Python function-level code generation, with additional evidence on LiveCodeBench and Llama-family models in Appendix A.7.2 and Appendix A.7.3. Its transferability to other programming languages, longer code contexts, or non-code reasoning tasks remains not fully established.

Ethical Considerations

This work aims to improve code generation and automated testing through reinforcement learning. By reducing reliance on expensive human-annotated test suites, Code-A1 may lower the barrier to training code models with verifiable feedback and improve software reliability. A potential risk is dual use: bug-revealing test generation could be misused to identify exploitable software weaknesses. However, this work focuses on function-level programming benchmarks and unit-test generation rather than real-world production systems, security exploits, or vulnerability exploitation workflows; generated code is evaluated in sandboxed environments, and tests are primarily used as learning signals for improving model robustness. We use public or synthetic programming benchmarks only for research and evaluation, and all third-party models, datasets, and benchmarks are cited and used under their stated terms. No human participants or annotators were recruited, and no new human-subject data were collected. Overall, we believe the primary impact of this work is positive, as stronger automated testing and more reliable code generation can help reduce software defects and support safer deployment of code models.

Acknowledgement

This work was supported by National Natural Science Foundation of China (No. 62436007), National Natural Science Foundation of China (No. 62506332) and the Key Research and Development Program of Ningbo (No. 2026Z003).

References

- Juan Altmayer Pizzorno and Emery D Berger. 2025. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, and 1 others. 2021. [Program synthesis with large language models](#). *Preprint*, arXiv:2108.07732.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, and 1 others. 2021. [Evaluating large language models trained on code](#). *Preprint*, arXiv:2107.03374.
- Yixing Chen, Yiding Wang, Siqi Zhu, Haofei Yu, Tao Feng, Muhan Zhang, Mostofa Patwary, and Jiaxuan You. 2025. Multi-agent evolve: Llm self-improve through co-evolution. *arXiv preprint arXiv:2510.23595*.
- Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, and 1 others. 2024. Fullstack bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*.
- Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, Buck Shlegeris, Samuel R. Bowman, Ethan Perez, and Evan Hubinger. 2024. [Sycophancy to subterfuge: Investigating reward-tampering in large language models](#). *Preprint*, arXiv:2406.10162.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shitong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

- Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. 2025. [Hardtests: Synthesizing high-quality test cases for LLM coding](#). In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.
- Soneya Binta Hossain and Matthew B Dwyer. 2025. [Togll: Correct and strong test oracle generation with llms](#). In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1475–1487. IEEE.
- Dong Huang, Jie M Zhang, Mark Harman, Qianru Zhang, Mingzhe Du, and See-Kiong Ng. 2025. [Benchmarking llms for unit test generation from real-world functions](#). *arXiv preprint arXiv:2508.00408*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. [Qwen2. 5-coder technical report](#). *arXiv preprint arXiv:2409.12186*.
- Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. 2025a. [Testgeneval: A real world unit test generation and test completion benchmark](#). In *The Thirteenth International Conference on Learning Representations*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025b. [Livecodebench: Holistic and contamination free evaluation of large language models for code](#). In *The Thirteenth International Conference on Learning Representations*.
- Jaewoo Jeong, Taesoo Kim, and Sangdon Park. Ensuring functional correctness of large code models with selective generation. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. [Efficient memory management for large language model serving with pagedattention](#). In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. [Coderl: Mastering code generation through pretrained models and deep reinforcement learning](#). In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328. Curran Associates, Inc.
- Dongjun Lee, Changho Hwang, and Kimin Lee. 2026. [Learning to generate unit test via adversarial reinforcement learning](#). In *The Fourteenth International Conference on Learning Representations*.
- Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. 2023a. [RLTF: Reinforcement learning from unit test feedback](#). *Transactions on Machine Learning Research*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023b. [Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation](#). In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Hongliang Lu, Yuhang Wen, Pengyu Cheng, Ruijin Ding, Haotian Xu, Jiaqi Guo, Chutian Wang, Haonan Chen, Xiaoxi Jiang, and Guanjin Jiang. 2025. [Search self-play: Pushing the frontier of agent capability without supervision](#). *arXiv preprint arXiv:2510.18821*.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, and 1 others. 2024. [Deepseekmath: Pushing the limits of mathematical reasoning in open language models](#). *Preprint*, arXiv:2402.03300.
- Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, and 1 others. 2023. [Pangu-coder2: Boosting large language models for code with ranking feedback](#). *arXiv preprint arXiv:2307.14936*.
- Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, and 1 others. 2025. [Hybridflow: A flexible and efficient rlhf framework](#). In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.
- Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. [Execution-based code generation using deep reinforcement learning](#). *Transactions on Machine Learning Research*.
- Frank Tip, Jonathan Bell, and Max Schäfer. 2025. [Ll-morpheus: Mutation testing using large language models](#). *IEEE Transactions on Software Engineering*.
- Kaiqi Wang, Wenjin Hou, Yuchen Yan, Hongrui Jia, Zhisheng Zhong, Botao Ren, Yifei Chen, Songyang Zhang, Yongliang Shen, Jun Xiao, Yi Yang, Yueting Zhuang, and Hehe Fan. 2026. [Diving into reliable self-evolving agents: A survey](#). *OpenReview.net*.
- Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025. [CURE: Co-evolving coders and unit testers via reinforcement learning](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025. [Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding](#). *arXiv preprint arXiv:2503.02951*.
- Yuchen Yan, Liang Jiang, Jin Jiang, Shuaicheng Li, zu-jie wen, Zhiqiang Zhang, JUN ZHOU, Jian Shao, Yueting Zhuang, and Yongliang Shen. 2026. [Infy-think+: Effective and efficient infinite-horizon reasoning via reinforcement learning](#). In *Forty-third International Conference on Machine Learning*.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. [SWE-smith: Scaling data for software engineering agents](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. In *The Thirty-Ninth Annual Conference on Neural Information Processing Systems*.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhui Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12023–12040.

Runzhe Zhan, Yafu Li, Zhi Wang, Xiaoye Qu, Dongrui Liu, Jing Shao, Derek F Wong, and Yu Cheng. 2025. Exgrpo: Learning to reason from experience. *arXiv preprint arXiv:2510.02245*.

Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. 2024a. [O1-coder: An o1 replication for coding](#). *Preprint*, arXiv:2412.00154.

Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. 2024b. o1-coder: an o1 replication for coding. *arXiv preprint arXiv:2412.00154*.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. 2025. [Absolute zero: Reinforced self-play reasoning with zero data](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, and 1 others. 2023. [Pytorch fsdp: Experiences on scaling fully sharded data parallel](#). *Preprint*, arXiv:2304.11277.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kadour, Ming Xu, Zhihan Zhang, and 14 others. 2025. [Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions](#). In *The Thirteenth International Conference on Learning Representations*.

A Experimental Details

A.1 Prompt Details

Code LLM

```
<lim_start>system
You are a helpful code completion assistant.
<lim_end>
<lim_start>user
Given the following Question, complete the
function. Output the complete function
inside ```python ... ``` code block,
and do not output anything else.
Question:
{question}
<lim_end>
<lim_start>assistant
```

Test LLM

```
<lim_start>system
You are a helpful test case generation
assistant.
<lim_end>
<lim_start>user
# Role
You are specializing in finding specific
inputs that cause `Buggy Code` to behave
differently from the requirements
(`Question`).

# Task
Generate 8 assertion-based test cases to
detect bugs for the function in `Buggy
Code` according to the `Question`.

# Strategy
1. Attack Logic Gaps: Analyze where the
`Buggy Code` logic might be too simple
compared to the `Question`. Construct
input `parameters` that hit these blind
spots (e.g., missing constraints,
misinterpreted rules, over-simplified logic).
2. Prioritize Complexity: Prefer complex
input `parameters` (e.g., boundary values,
nested loops, compound conditions, rare
branches) over simple ones. Ensure every
logical branch is stressed and every
potential issue is covered.
3. Zero Redundancy: Do not brute-force
generating trivial or repetitive tests. Only
the first few generated tests will be
```

evaluated, so quality and ordering matter more than quantity.

Context

Question:

{question}

Buggy Code:

```
```python
{generated_code}
```
```

Output Format

Output ALL the assert statements inside
ONE ```python ... ``` code block.

Format: assert

function_name(parameters) ==
answer

<lim_endl>

<lim_startl>assistant

A.2 Sandbox Details

Given that code training and validation heavily rely on stable sandbox environments, we deployed a sandbox service using Sandbox Fusion (Cheng et al., 2024). This service organizes code and tests for batch execution within the sandbox, retrieving execution results via the sandbox’s stdout. The code template executed in the sandbox is as follows:

Template 1: Validating and Executing in Training Phase. Template 1 is designed for efficient interaction during the training phase. It employs a high-reusability batch execution strategy, compressing all sandbox interaction requests for a single training step into one, thereby significantly reducing high concurrency pressure on the sandbox. The process follows a “calibrate-then-evaluate” mechanism: first, the Ground Truth code is introduced as an Oracle to execute and verify the test cases generated by the Test LLM, filtering out invalid cases that are malformed or inconsistent with standard outputs; subsequently, in a cleaned context, the validated new tests are combined with historical tests from the Mistake Book to comprehensively evaluate the code generated by the Code LLM, thus collecting all necessary data for reward computation in a single execution.

```
<GLOBAL_IMPORTS>
valid_asserts_list = []
```

```
try:
    <GROUND_TRUTH_CODE>
    for n in range(<NUM_SAMPLES>):
        invalid_count = <INVALID_COUNT>
        valid_asserts = []
        seen = set()
        gen_total = 0
        for stmt, call, ans in zip(<
NORMALIZED_STMTS>, <CALLS>, <ANSWERS
>):
            try:
                _r = <FUNCTION_CALL>
                stmt = stmt.replace('
__TO_BE_FILLED__', repr(_r))
                if stmt in seen:
                    invalid_count += 1
                    valid_asserts.append
(None)
            else:
                seen.add(stmt)
                valid_asserts.append
(({ 'val': _r, 'stmt': stmt})
                print('__GEN_START__
<TEST_ID>_' + str(gen_total) + ':' +
repr(stmt), flush=True)
                gen_total += 1
                if (_r) != (<
EXPECTED_ANSWER>):
                    invalid_count +=
1
            except Exception:
                invalid_count += 1
                valid_asserts.append(
None)
                print('__INVALID_TEST__<TEST_ID
>:' + str(invalid_count), flush=True)
                valid_asserts_list.append(
valid_asserts)
except Exception:
    pass

for gt_fn in <GT_FUNCTION_NAMES>:
    try:
        del <GT_FN>
    except Exception:
        pass

try:
    <GENERATED_CODE>
    print('__CODE_VALID__<CODE_ID>',
flush=True)

    for idx, test_item in enumerate(<
GT_TESTCASE_LIST>):
        try:
            <GT_TEST_CODE>
            print('__GT_PASS__<CODE_ID>_
' + str(idx), flush=True)
        except Exception as e:
            print('__GT_FAIL__<CODE_ID>_
' + str(idx) + ':' + repr(e), flush=
True)

    for idx, test_item in enumerate(<
ATTACK_TESTCASE_LIST>):
        print('__ATTACK_START__<CODE_ID>
_' + str(idx) + ':' + repr(repr(
test_item)), flush=True)
        try:
```

```

        <ATTACK_TEST_CODE>
        print('__ATTACK_PASS__<
CODE_ID>' + str(idx), flush=True)
        except Exception as e:
            print('__ATTACK_FAIL__<
CODE_ID>' + str(idx) + ':' + repr(e
), flush=True)

        for n in range(<NUM_SAMPLES>):
            valid_asserts =
            valid_asserts_list[n]
            gen_total = 0
            for j, call in enumerate(<
CALLS_LIST>):
                spec = valid_asserts[j]
                if spec is not None:
                    try:
                        assert <
FUNCTION_CALL> == spec['val']
                        print('__GEN_PASS__<
TEST_ID>' + str(gen_total), flush=
True)
                    except Exception as e:
                        print('__GEN_FAIL__<
TEST_ID>' + str(gen_total) + ':' +
repr(e), flush=True)
                        gen_total += 1
            except Exception:
                pass

```

Template 2: Evaluation in Validation Phase.

Template 2 is explicitly designed for the inference evaluation phase of the Code LLM, aiming to efficiently complete the batch validation of multiple sampled solutions for the same question via a single sandbox request. The script independently encapsulates each generated function body using a signal-based timeout protection mechanism (`run_safe`) to prevent infinite loops in individual samples from interrupting the overall evaluation task. By sequentially calling the standard test harness within the same context, it rapidly verifies the functional correctness of large-scale candidate code; this design significantly enhances the concurrency efficiency for calculating metrics such as `avg@k` while ensuring execution safety.

```

<IMPORTS>
import signal

class T0(Exception): pass
def handler(s, f): raise T0()

def run_safe(func):
    try:
        signal.signal(signal.SIGALRM,
            handler)
        signal.alarm(<TIMEOUT>)
        func()
        signal.alarm(0)
        print('__PASS__', flush=True)
    except T0:
        print('__TIMEOUT__', flush=True)
    except:

```

```

        signal.alarm(0)
        print('__FAIL__', flush=True)

<TEST_HARNESS_CODE>

def c_0():
    <GENERATED_CODE_0>
    check(<ENTRY_POINT>)

def c_1():
    <GENERATED_CODE_1>
    check(<ENTRY_POINT>)

# ... (repeated for batch size) ...

run_safe(c_0)
run_safe(c_1)
# ... (repeated for batch size) ...

```

A.3 Reward Details

Code LLM Reward. Let $\text{Pass}_{\text{hist}}(\hat{C}_m)$ denote the pass rate on historical tests T^{hist} , and $\text{AVG}(\text{Pass}_{\text{new}}(\hat{C}_m))$ denote the average pass rate on newly generated test suites. The reward for the Code LLM is defined as:

$$R_C(\hat{C}_m) = \begin{cases} \frac{1}{2} (\text{Pass}_{\text{hist}} + \text{AVG}(\text{Pass}_{\text{new}})), \\ \text{Pass}_{\text{new}}. \end{cases} \quad (1)$$

The first branch is used when historical tests are available, i.e., $T^{\text{hist}} \neq \emptyset$; otherwise, the reward reduces to the pass rate on newly generated tests. When historical failures exist, averaging the two pass rates ensures that the Code LLM cannot achieve high rewards by merely passing new tests while regressing on previously challenging cases.

Test LLM Reward. The Test LLM reward consists of a validity component and an adversarial component. The validity reward measures the fraction of generated tests that pass validation:

$$R_T^{\text{val}}(\hat{T}) = \text{Valid}(\hat{T}) \quad (2)$$

The adversarial reward compares the difficulty of newly generated tests with historical tests:

$$R_T^{\text{adv}}(\hat{T}) = \begin{cases} \frac{1}{2} (\text{Pass}_{\text{hist}} - \text{Pass}_{\text{new}} + 1), \\ 1 - \text{Pass}_{\text{new}}. \end{cases} \quad (3)$$

The first branch is used when $T^{\text{hist}} \neq \emptyset$; otherwise, only the failure rate on newly generated tests is used. When $\text{Pass}_{\text{hist}} > \text{Pass}_{\text{new}}$, the new tests expose defects that historical tests missed, yielding higher reward. When $\text{Pass}_{\text{hist}} < \text{Pass}_{\text{new}}$, the new tests are easier than historical ones, incurring

penalty. The final reward for the Test LLM is defined as:

$$R_T(\hat{T}) = \alpha \cdot R_T^{\text{val}}(\hat{T}) + (1 - \alpha) \cdot R_T^{\text{adv}}(\hat{T}) \quad (4)$$

where $\alpha \in [0, 1]$ controls the trade-off. Setting α too high encourages trivial but valid tests; setting α too low risks invalid adversarial tests. We study this trade-off in ablations.

A.4 Mistake Book Details

A.4.1 Mistake Book Design

The Mistake Book is implemented as a global variable stored in JSON format, designed to stabilize adversarial training through a persistent experience replay mechanism. The data structure utilizes a key-value pair format: the Key serves as the unique identifier for each question (`question_id`), while the Value is a list of dictionaries containing specific failed test cases and their corresponding failure frequencies (`frequency`).

$$\mathcal{B} : Q_i \mapsto T_i^{\text{hist}} = \{t \mid \hat{C} \text{ failed } t \text{ in previous iters}\} \quad (5)$$

Usage. For each question in the current training step, the system first queries the Mistake Book. If historical failure records exist, these challenging “mistakes” are retrieved and appended to the list of tests newly generated by the Test LLM, to be executed together in the sandbox.

Update. The update mechanism operates at the level of individual assertion test points. If the code fails a specific test, that test is added to the Mistake Book (or its frequency is incremented). Conversely, if the code successfully passes a known test from the Mistake Book, its frequency is decremented. When the frequency drops to zero, the test case is considered “mastered” and is removed from the Mistake Book.

Persistence. To support training interruption and resumption, the system serializes and saves the current global Mistake Book to a JSON file after every step. This ensures that historical adversarial experiences can be seamlessly reloaded when resuming training.

```
{
  "Apps_1564_I": [
    {
      "testcase": "assert
      number_of_characters(5) == 2",
      "frequency": 4
    },
  ],
}
```

```
{
  "testcase": "assert
  number_of_characters(10) == 3",
  "frequency": 5
},
...
],
"Leetcode_17190_I": [
  {
    "testcase": "assert
    can_rearrange_to_palindrome(\"aaa\",
    1) == False",
    "frequency": 10
  },
  {
    "testcase": "assert
    can_rearrange_to_palindrome(\"aa\",
    1) == False",
    "frequency": 12
  },
  ...
]
...
}
```

A.4.2 Effect of Mistake Book

Figure 5 visualizes how the Mistake Book stabilizes adversarial co-evolution. In Figure 5a, the pass rate on historical tests (dark blue) begins tracking at Step 38 after the first epoch populates the buffer. While enforcing historical correctness initially lowers the composite pass rate compared to the ablation without Mistake Book (solid vs. dashed blue), this prevents catastrophic forgetting and ultimately achieves performance exceeding the Golden Tests baseline. Figure 5b reveals the adversarial dynamics: added tests (pink) represent successful attacks exposing new vulnerabilities, while removed tests (green) indicate successful defenses fixing previous bugs. After Step 38, experience replay triggers a surge in bug fixes. As training progresses, the two curves converge toward equilibrium, with the Test LLM uncovering increasingly subtle edge cases while the Code LLM repairs specific logical flaws. This dynamic balance drives synchronized improvement of both models.

A.5 Training Details

A.5.1 General RL Details

Our RL experiments are conducted using verl (Sheng et al., 2025) as the training framework, with vLLM (Kwon et al., 2023) serving as the inference engine and FSDP (Zhao et al., 2023) as the training backend. During the rollout phase, we sequentially sample from the Code LLM and Test LLM (if exists), submitting post-processed and concatenated responses to a sandbox server

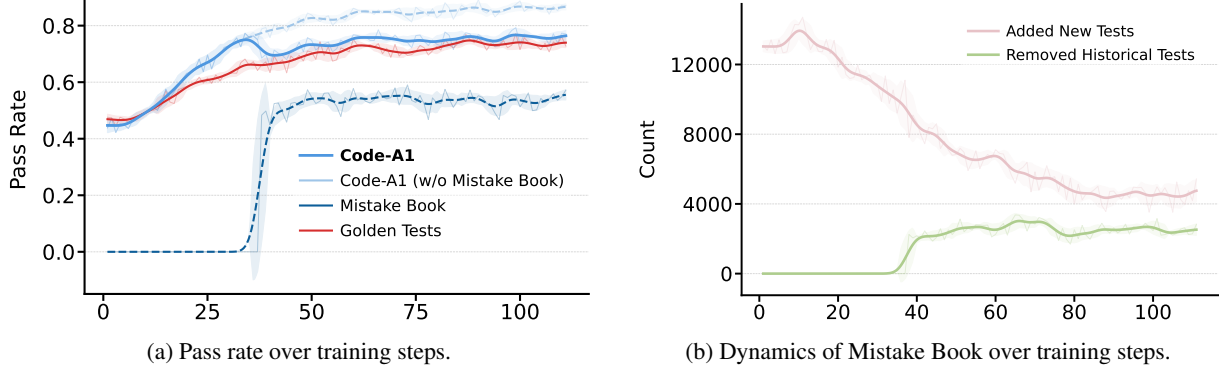


Figure 5: **Effect of Mistake Book.** (a) Pass rate on newly generated tests, historical tests, and Golden Tests baseline over training. (b) Number of tests added to and removed from the Mistake Book per step.

subject to a strict 10-second execution timeout to prevent training bottlenecks. To ensure robustness against sandbox instability, we implement a bidirectional monitoring mechanism: a supervisor script performs 30-second health checks to auto-restart Docker services upon anomalies, while execution failures trigger a retry protocol with a 60-second backoff and a maximum of five attempts. Samples that remain unresponsive are assigned a zero reward, ensuring a zero advantage estimate—since grouped samples share identical rewards—to prevent erroneous policy updates. Finally, we compute distinct rewards for each model based on execution outcomes and perform asynchronous parameter updates. All experiments are performed on 8 NVIDIA H20 GPUs with CUDA 12.8. The detailed RL hyperparameters are summarized in Table 6.

A.5.2 Cost Analysis

All experiments use the same 8-GPU budget and 111 training steps. In the 1.5B setting, excluding sandbox execution, Code-A1 requires 208.61 GPU·s per step, compared with 70.26 GPU·s per step for Golden Tests. Although Code-A1 trains two models, this cost remains below the naive $4\times$ upper bound because the Code LLM and Test LLM are each trained on 4 GPUs, whereas Golden Tests trains a single model on 8 GPUs. The CPU-side sandbox cost also remains moderate: Code-A1 takes 81.58 seconds per step, compared with 36.99 seconds per step for Golden Tests, which is far below the naive $8\times$ expectation from the number of Code LLM samples. This is mainly because sandbox latency is dominated by network requests rather than the execution speed of individual test cases. Our sandbox template therefore batches multiple generated tests for the same candidate code into a single request and returns their

results together, substantially reducing the number of sandbox calls. In addition, reward computation is performed asynchronously and can be largely overlapped with GPU-side training. Beyond model training and sandbox execution, the Mistake Book introduces negligible additional overhead, since retrieval and update are implemented as per-question hash-map operations.

A.5.3 Asymmetric Sampling and TOPVAR

For Code-A1, in the asymmetric sampling phase, we employ the TOPVAR algorithm to filter for the most informative groups based on reward variance.

Formally, let $R_{m,n}$ denote the reward for the n -th test suite generated conditioned on the m -th candidate solution $\hat{C}_m$. We calculate the intra-group reward standard deviation σ_m for each $m \in \{1, \dots, M\}$:

$$\sigma_m = \sqrt{\frac{1}{N} \sum_{n=1}^N (R_{m,n} - \bar{R}_m)^2}, \quad (6)$$

where $\bar{R}_m = \frac{1}{N} \sum_{n=1}^N R_{m,n}$.

We then derive the index set $\mathcal{I}^*$ corresponding to the groups with the highest variance:

$$\mathcal{I}^* = \arg \text{top-}\ell (\sigma_m)_{m \in \{1, \dots, M\}} \quad (7)$$

The gradient update for the Test LLM is strictly restricted to these selected groups to maximize training efficiency. For each selected test suite, we define:

$$g_{m,n} = \hat{A}_{m,n} \nabla_{\theta} \log \pi_{\theta}(\hat{T}_{m,n} | Q, \hat{C}_m). \quad (8)$$

The final Test LLM update is then:

$$\nabla \mathcal{J}_{\text{Test}} \approx \frac{1}{\ell \times N} \sum_{m \in \mathcal{I}^*} \sum_{n=1}^N g_{m,n}. \quad (9)$$

| Hyperparameter | Golden Tests (GT) | Self-Play (SP) | Code-A1 |
|---|-------------------|--------------------|-------------------|
| <i>Experiment Setup & Resources</i> | | | |
| Test LLM Source | - | Self-Play (Shared) | Independent Model |
| Reward Manager | acg_gt | acg_llm | acg_llm |
| Reward Alpha (α) | - | 0.5 | 0.5 |
| n_gpu_per_node | 8 | 8 | 4 |
| <i>Trainer</i> | | | |
| total_epochs | 3 | 3 | 3 |
| <i>Algorithm</i> | | | |
| adv_estimator | grpo | grpo | grpo |
| kl_coef | 0.001 | 0.001 | 0.001 |
| <i>Data</i> | | | |
| train_batch_size | 256 | 256 | 256 |
| max_prompt_length | 4096 | 4096 | 4096 |
| max_response_length | 4096 | 4096 | 4096 |
| <i>Model (Actor)</i> | | | |
| model.use_remove_padding | true | true | true |
| model.enable_gradient_checkpointing | true | true | true |
| actor.ppo_mini_batch_size | 128 | 128 | 128 |
| actor.use_dynamic_bsz | true | true | true |
| actor.ppo_max_token_len_per_gpu | 32768 | 32768 | 32768 |
| actor.clip_ratio_low | 0.20 | 0.20 | 0.20 |
| actor.clip_ratio_high | 0.28 | 0.28 | 0.28 |
| actor.optim.lr | 1e-6 | 1e-6 | 1e-6 |
| actor.weight_decay | 0.1 | 0.1 | 0.1 |
| <i>Rollout</i> | | | |
| rollout.name | vllm | vllm | vllm |
| rollout.mode | sync | sync | sync |
| rollout.tensor_model_parallel_size | 1 | 1 | 1 |
| rollout.gpu_memory_utilization | 0.8 | 0.8 | 0.8 |
| rollout.n | 8 | 8 | 8 |
| rollout.temperature | 1.0 | 1.0 | 1.0 |
| rollout.top_p | 1.0 | 1.0 | 1.0 |
| rollout.top_k | -1 | -1 | -1 |
| <i>Reference Model</i> | | | |
| ref.log_prob_use_dynamic_bsz | true | true | true |
| ref.log_prob_max_token_len_per_gpu | 32768 | 32768 | 32768 |

Table 6: Detailed hyperparameters for reinforcement learning experiments.

A.5.4 Self-Play Details

The Self-Play architecture employs a unified model to fulfill both Code LLM and Test LLM roles. Since the same policy in Self-Play both produces candidate solutions and constructs the tests used for reward estimation, granting code-conditioned test generation would introduce a direct self-collusion channel: the model could obtain inflated rewards by generating trivial tests aligned with its own candidate solutions rather than exposing implementation-specific failures. We therefore have to adopt black-box test generation for Self-Play, where the generated tests condition only on the problem description and are shared across candidate solutions for the same problem. By contrast, Code-A1 can safely exploit white-box testing because the Test LLM is architecturally decoupled from the Code LLM and optimized adversarially. To validate this de-

sign choice, we additionally evaluate a white-box Self-Play variant. As shown in Table 7, allowing the unified Self-Play policy to inspect its own candidate code increases test accuracy but degrades test quality and code correctness, indicating that black-box testing design for Self-Play is not an artificial restriction imposed to weaken the baseline, but a structural safeguard against self-collusion.

Apart from this architectural distinction, we keep the Self-Play baseline closely aligned with Code-A1 to avoid confounding factors. Mechanisms that are generally beneficial and can be shared across methods—including test validation, composite test reward design, and the Mistake Book mechanism—are aligned between Code-A1 and Self-Play. The comparison therefore primarily isolates the effect of role decoupling and safe white-box test generation, rather than contrasting a complex sys-

tem with a simplified baseline. In implementation, Self-Play samples $M = 8$ candidate solutions and $N = 8$ test suites for each problem. The generated solutions are cross-executed against the generated tests, after which reward construction and replay follow the same procedure as in Code-A1.

A.5.5 SFT Details

To strictly quantify the performance gains of RL training compared to Supervised Fine-Tuning (SFT) for the Test LLM under identical data distributions, we constructed the SFT dataset based on the same training source used in the RL phase. To facilitate convergence, we adopted a simplified prompt structure. Specifically, the input Buggy Code consists of pre-sampled solution candidates that failed to pass the complete set of Golden Tests, while the output labels utilize the corresponding ground truth Golden Tests. This design intends to guide the model, via imitation learning, to master the generation of standard test cases given flawed code, thereby establishing a robust SFT baseline for the comparative reinforcement learning experiments.

Prompt for Test LLM in SFT

```
<lim_start>system
You are a helpful test case generation
assistant.
<lim_end>
<lim_start>user
Given the following Question, generate 8
assert-based test cases to detect bugs in the
following python function Buggy Code.
Each test case must be in the format:
assert function_name(parameters)
== answer, and split by \n. Output all
assert statements inside ```python ...
``` code block, and do not output anything
else.

Question:
{question}

Buggy Code:
```python
{buggy_code}
```

<lim_end>
<lim_start>assistant
```

## A.6 Test-time Scaling Details

### A.6.1 Experimental Setup

To evaluate the collaborative performance of the Code LLM and Test LLM during the inference phase, we implemented a parallel scaling strategy based on the Best-of-N (BoN) selection mechanism based on Qwen2.5-Coder-3B-Instruct. For each question, the Code LLM samples  $M = 16$  candidate solutions, and the Test LLM generates  $N = 16$  corresponding unit test suites. Unlike the training phase, the Test LLM here generates test suites based solely on the question description, blind to the candidate solutions, serving as an independent verifier. We execute a full interaction matrix where every candidate solution is tested against every generated test suite. The candidate solution that passes the highest number of unique test cases across all suites is selected as the final answer.

In Table 5, the symbol ‘/’ in the **Test LLM** column denotes the baseline setting without scaling methods, reporting the direct avg@32 results. The abbreviations **HE**<sup>+</sup> denotes HumanEval<sup>+</sup> and **BCB** denotes BigCodeBench.

### A.6.2 Complete Results

Table 8 expands Table 5 with Self-Play variants. The self-verification inference setting is aligned with the unified Self-Play training setup because the same policy serves both coding and testing roles. We also include a duplicated-instance Self-Play setting for completeness. Both variants underperform Code-A1, suggesting that the gain comes from role decoupling during training rather than merely using two inference instances.

## A.7 Evaluation Details

### A.7.1 Robustness Claim

All reported evaluation results are obtained under a matched repeated-evaluation protocol with the same inference environment, decoding configuration, and sandbox for all compared methods. We repeat evaluations with 5 random seeds, compute 95% confidence intervals, and conduct two-sided paired  $t$ -tests with matched seeds using Code-A1 as the reference method. Due to space limitations, we omit detailed error bars and significance-test tables from the paper, but all values reported in the main text and appendix are based on this stable evaluation protocol rather than a single favorable random seed or sampling outcome. The consistent rankings

| Method                | HumanEval <sup>+</sup> | MBPP <sup>+</sup> | BigCodeBench | Avg          | UnLeakedTestBench |       | Mul         |
|-----------------------|------------------------|-------------------|--------------|--------------|-------------------|-------|-------------|
|                       | avg@32                 | avg@32            | avg@32       |              | pass@5            | mut@5 |             |
| Self-Play (black-box) | 70.64                  | 63.54             | 33.47        | 55.88        | 23.39             | 28.91 | 6.76        |
| Self-Play (white-box) | 67.51                  | 61.31             | 31.64        | 53.49        | 29.65             | 17.34 | 5.14        |
| Code-A1 (white-box)   | <b>72.69</b>           | 63.33             | <b>34.82</b> | <b>56.95</b> | 27.05             | 26.41 | <b>7.14</b> |

Table 7: Performance comparison between black-box and white-box Self-Play on Qwen2.5-Coder-1.5B-Instruct.

| Parallel TTS |           | HE <sup>+</sup> | MBPP <sup>+</sup> | BCB          | Avg          |
|--------------|-----------|-----------------|-------------------|--------------|--------------|
| Code LLM     | Test LLM  |                 |                   |              |              |
| Base         | /         | 77.63           | 63.12             | 41.78        | 60.84        |
| Self-Play    | /         | 81.86           | 67.06             | 45.09        | 64.67        |
| Code-A1      | /         | 83.52           | 69.07             | 45.85        | 66.15        |
| Base         | Base      | 82.32           | 70.63             | 42.45        | 65.13        |
| Self-Play    | Base      | 83.64           | 70.98             | 44.36        | 66.33        |
| Code-A1      | Base      | 84.76           | 71.69             | 45.91        | 67.45        |
| Base         | Self-Play | 82.42           | 71.05             | 43.31        | 65.59        |
| Base         | Code-A1   | 82.93           | 70.90             | 43.63        | 65.82        |
| Self-Play    | Self-Play | 84.36           | 70.37             | 44.90        | 66.54        |
| Code-A1      | Code-A1   | <b>85.37</b>    | <b>71.96</b>      | <b>46.09</b> | <b>67.81</b> |

Table 8: Complete parallel test-time scaling comparison.

across repeated evaluations support the robustness and reliability of the reported improvements.

### A.7.2 Benchmark Generalization

Although Code-A1 is designed around Python function-level tasks, we also evaluate on LiveCodeBench-v2 (Jain et al., 2025b) to test robustness under more OJ-style tasks. As shown in Table 9, Code-A1 remains consistently ahead of Golden Tests and Self-Play across all three Qwen2.5-Coder scales. The gap is smaller than on function-level benchmarks, which is expected because standardized stdin/stdout formats weaken the advantage of our training scenario.

| Code LLM               | Method       | LiveCodeBench-v2 |
|------------------------|--------------|------------------|
| Qwen2.5-Coder-1.5B-Ins | Base         | 14.89            |
|                        | Golden Tests | 20.48            |
|                        | Self-Play    | 19.78            |
|                        | Code-A1      | <b>21.23</b>     |
| Qwen2.5-Coder-3B-Ins   | Base         | 27.09            |
|                        | Golden Tests | 28.93            |
|                        | Self-Play    | 28.66            |
|                        | Code-A1      | <b>29.18</b>     |
| Qwen2.5-Coder-7B-Ins   | Base         | 38.95            |
|                        | Golden Tests | 40.26            |
|                        | Self-Play    | 39.49            |
|                        | Code-A1      | <b>40.80</b>     |

Table 9: Performance comparison of Code LLMs on LiveCodeBench-v2.

### A.7.3 Cross-Family Generalization

To check whether the improvement is specific to Qwen2.5-Coder, we evaluate Code-A1 on Llama-3.2-3B-Instruct (Grattafiori et al., 2024). Table 10 shows that the same ranking holds: Code-A1 outperforms both Golden Tests and Self-Play, which suggests that the advantages of Code-A1 are not tied to a single model family.

### A.7.4 Metrics Definition

#### A.7.4.1 Avg

**Avg** represents the macro-average of the code generation performance across all evaluated benchmarks. It is calculated as the arithmetic mean of the avg@32 scores on HumanEval<sup>+</sup>, MBPP<sup>+</sup>, and BigCodeBench, providing a single scalar to reflect the model’s overall coding capability.

#### A.7.4.2 Mul

**Mul** is designed to evaluate the *yield of high-quality tests* by combining validity and adversarial difficulty. As described in UnLeakedTestBench (Huang et al., 2025), the calculation of mutation score ( $mut@k$ ) is inherently conditional on the generated tests first being valid (i.e., compiling and passing the original code,  $pass@k$ ).

This dependency creates a trade-off between  $pass@k$  and  $mut@k$ :

- A model might achieve a high  $pass@k$  by generating trivial, low-complexity tests that lack fault-detection capability (resulting in a low  $mut@k$ ).
- Conversely, a model might achieve a high  $mut@k$  by generating highly complex tests, even if only a small fraction of them are valid (resulting in a low  $pass@k$ ). This can occur if the metric evaluation considers the “best” tests from a sparse set of valid candidates.

To address this, we define **Mul** as:

$$Mul = pass@k \times mut@k \quad (10)$$

By multiplying these two metrics, **Mul** normalizes the mutation score against the total sampling

| Method       | HumanEval <sup>+</sup> | MBPP <sup>+</sup> | BigCodeBench | Avg          | UnLeakedTestBench |       | Mul         |
|--------------|------------------------|-------------------|--------------|--------------|-------------------|-------|-------------|
|              | avg@32                 | avg@32            | avg@32       |              | pass@5            | mut@5 |             |
| Base         | 41.31                  | 41.92             | 24.90        | 36.04        | 17.13             | 40.52 | 6.94        |
| Golden Tests | 55.35                  | 60.32             | 26.80        | 47.49        | 14.57             | 43.39 | 6.32        |
| Self-Play    | 53.79                  | 59.06             | 26.15        | 46.33        | 19.52             | 43.45 | 8.48        |
| Code-A1      | <b>56.78</b>           | <b>61.50</b>      | <b>27.08</b> | <b>48.45</b> | 21.48             | 44.14 | <b>9.48</b> |

Table 10: Generalization of Code-A1 on Llama-3.2-3B-Instruct.

budget. It reflects the true proportion of generated tests that are **both** valid and adversarial, ensuring that a high score requires the model to simultaneously maintain high generation quality and strong fault-detection capability.

In our experiments, *pass@k* is evaluated on the full 3,909-task benchmark, while *mut@k* is evaluated on a 10% subset due to the cost of mutation testing, which is enough to reliably reflect overall test generation ability according to the benchmark authors.

## B Examples

### B.1 Question Example

```
from typing import List
def threeSum(nums: List[int], target:
 int) -> List[List[int]]:
 """
 Write a function that receives
 an array of integers and a target
 integer. Your task is to find all
 unique triplets in the array which
 gives the sum of the target integer.
 The solution set must not contain
 duplicate triplets.
 Example:
 - Input: `nums = [-1, 0, 1, 2, -1,
 -4], target = 0`
 - Output: `[[-1, -1, 2], [-1, 0,
 1]]`
 """
```

### B.2 Ground Truth Example

```
from typing import List
def threeSum(nums: List[int], target:
 int) -> List[List[int]]:
 nums.sort() # Sort the array to
 help avoid duplicates and use two-
 pointer strategy
 result = []

 for i in range(len(nums) - 2):
 if i > 0 and nums[i] == nums[i -
 1]: # Skip the same element to
 avoid duplicates
 continue

 left, right = i + 1, len(nums) -
 1
 while left < right:
```

```
 current_sum = nums[i] + nums
 [left] + nums[right]
 if current_sum == target:
 result.append([nums[i],
 nums[left], nums[right]])
 while left < right and
 nums[left] == nums[left + 1]: #
 Skip duplicates
 left += 1
 while left < right and
 nums[right] == nums[right - 1]: #
 Skip duplicates
 right -= 1
 left += 1
 right -= 1
 elif current_sum < target:
 left += 1
 else:
 right -= 1

 return result
```

### B.3 Golden Tests Example

```
assert threeSum([-1, 0, 1, 2, -1, -4],
 0) == [[-1, -1, 2], [-1, 0, 1]]
assert threeSum([1, 2, -2, -1], 1) ==
 [[-2, 1, 2]]
assert threeSum([0, 0, 0], 0) == [[0, 0,
 0]]
assert threeSum([], 0) == []
assert threeSum([3, 0, -2, -1, 1, 2], 0)
 == [[-2, -1, 3], [-2, 0, 2], [-1,
 0, 1]]
assert threeSum([-1, 0, 1, 2, -1, -4,
 -2, -3, 3, 0, 4], 0) == [[-4, 0, 4],
 [-4, 1, 3], [-3, -1, 4], [-3, 0,
 3], [-3, 1, 2], [-2, -1, 3], [-2, 0,
 2], [-1, -1, 2], [-1, 0, 1]]
assert threeSum([1, 2, 3, 4], 10) == []
assert threeSum([-1, 2, 1, 4], 8) == []
```

### B.4 Code LLM Response Example

```
def threeSum(nums, target):
 nums.sort()
 res = []
 n = len(nums)
 for i in range(n):
 if i > 0 and nums[i] == nums[i
 -1]:
 continue
 l, r = i + 1, n - 1
 while l < r:
 s = nums[i] + nums[l] + nums
 [r]
```

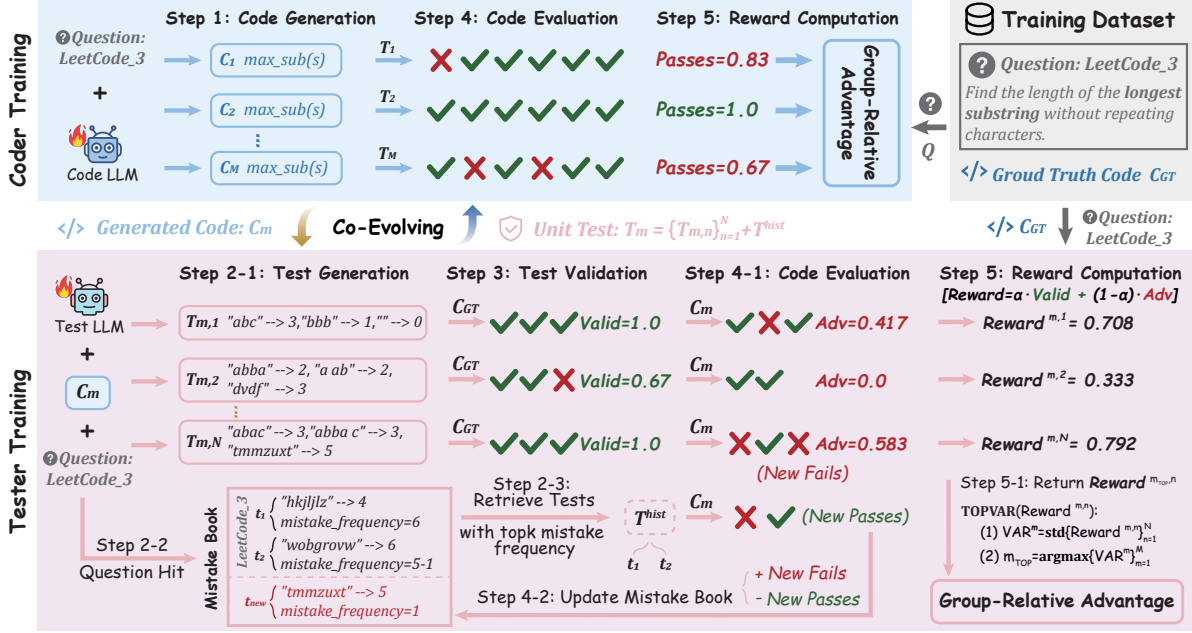


Figure 6: **Detailed pipeline of Code-A1.** Top: Code LLM training with code generation (Step 1), evaluation against combined tests (Step 4), and reward computation via group-relative advantage (Step 5). Bottom: Test LLM training with white-box test generation (Step 2-1), Mistake Book retrieval and update (Steps 2-2, 2-3, 4-2), test validation against ground truth (Step 3), and composite reward computation with TopVar selection (Step 5-1).

```

if s == target:
 res.append([nums[i],
nums[1], nums[r]])
 l += 1
 r -= 1
elif s < target:
 l += 1
else:
 r -= 1
return res

```

```

assert threeSum([-2, 0, 0, 2, 2], 0) ==
[[-2, 0, 2]]
assert threeSum([-1, -1, -1, 2, 2], 0)
== [[-1, -1, 2]]
assert threeSum([-4, 2, 2, 2, 2], 0) ==
[[-4, 2, 2]]
assert threeSum([-6, 3, 3, 3, 3, 3], 0)
== [[-6, 3, 3]]
assert threeSum([-2, 1, 1, 1, 1, 0, 2],
0) == [[-2, 0, 2], [-2, 1, 1]]
assert threeSum([-100, 50, 50, 50, 50],
0) == [[-100, 50, 50]]

```

## B.5 Test LLM Response Example

### Before Training

```

assert threeSum([-1, 0, 1], 0) == [[-1,
0, 1]]
assert threeSum([], 0) == []
assert threeSum([0, 0, 0], 0) == [[0, 0,
0]]
assert threeSum([0, 0, 0, 0], 0) == [[0,
0, 0]]
assert threeSum([-1, 0, 1, 2, -1, -4],
0) == [[-1, -1, 2], [-1, 0, 1]]
assert threeSum([1, 2, 3, 4], 0) == []
assert threeSum([-1, -1, 0, 1], 0) ==
[[-1, 0, 1]]
assert threeSum([3, 0, -2, -1, 1, 2], 0)
== [[-2, -1, 3], [-2, 0, 2], [-1,
0, 1]]

```

### After Training

```

assert threeSum([0, 0, 0, 0, 0], 0) ==
[[0, 0, 0]]
assert threeSum([-2, 1, 1, 1, 1], 0) ==
[[-2, 1, 1]]

```

## C Comparisons with Related Co-Evolution Frameworks

CURE (Wang et al., 2025), UTRL (Lee et al., 2026), and Code-A1 all explore the use of model-generated tests to improve code generation or test generation. We share the motivation that static human-written tests are limited in coverage and cannot adapt to the evolving capability of code models. However, we differ substantially in their supervision assumptions, optimization objectives, and co-evolution protocols. In particular, both CURE and UTRL mainly demonstrate that generated-test supervision can become comparable to labeled supervision, whereas Code-A1 targets a stronger outcome: removing the dependence on human-written unit tests while enabling the Code LLM to surpass the golden-test-trained baseline.

| Method  | HumanEval <sup>+</sup> | MBPP <sup>+</sup> | BigCodeBench | LiveCodeBench-v2 | Avg          | UnLeakedTestBench |       | Mul          |
|---------|------------------------|-------------------|--------------|------------------|--------------|-------------------|-------|--------------|
|         | avg@32                 | avg@32            | avg@32       |                  |              | pass@5            | mut@5 |              |
| Base    | 77.06                  | 71.78             | 46.83        | 45.60            | 60.32        | 37.91             | 52.33 | 19.84        |
| CURE    | 78.28                  | 72.57             | 46.99        | 47.16            | 61.25        | 36.90             | 57.73 | 21.30        |
| UTRL    | 74.94                  | 72.67             | 49.14        | <b>48.14</b>     | 61.22        | 52.99             | 56.72 | 30.06        |
| Code-A1 | <b>84.74</b>           | <b>75.27</b>      | <b>49.63</b> | 47.58            | <b>64.31</b> | 53.82             | 56.05 | <b>30.17</b> |

Table 11: Comparison with related co-evolution frameworks on Qwen3-4B (Long-CoT model).

### C.1 Code-A1

Code-A1 is designed for annotation-free code RL with dynamic, code-conditioned test generation. Its central objective is not only to train a stronger Test LLM, but also to make the generated tests reliable enough to serve as adaptive rewards for improving the Code LLM. To this end, Code-A1 decouples the Code LLM and Test LLM into two separately parameterized models with opposing objectives. This separation makes white-box test generation safe: the Test LLM can inspect the current candidate code and generate implementation-specific adversarial tests, while the Code LLM is optimized to pass increasingly stronger tests. The interaction is further stabilized by oracle-based test validation, composite test rewards, and Mistake Book replay. As a result, Code-A1 turns generated tests from an auxiliary signal into the primary dynamic supervision source for code RL, allowing the final Code LLM to outperform training with human-written golden tests.

### C.2 CURE

CURE also studies co-evolution between code generation and unit-test generation, but differs from Code-A1 in the role of generated tests. Although CURE removes the need for ground-truth code solutions, its code-generation reward is still based on ground-truth unit tests, while generated tests are mainly optimized on the tester side to support co-evolution. As a result, CURE demonstrates that generated-test supervision can become comparable to ground-truth labeled supervision. In contrast, Code-A1 targets a stricter annotation-free code-RL setting: human-written unit tests are not used as Code LLM training rewards, and dynamically generated and validated tests serve as the primary supervision signal for code optimization. This enables Code-A1 to surpass the golden-test-trained baseline rather than merely match it.

### C.3 UTRL

UTRL is closer to Code-A1 in that it also adopts separate models for code generation and test generation under an adversarial training formulation. Nevertheless, the two methods differ in their optimization target and co-evolution protocol. UTRL mainly aims to learn a stronger unit-test generator. In contrast, Code-A1 targets a more demanding code-RL objective: generated tests should serve as the primary reward source and improve the Code LLM beyond the golden-test-trained baseline. Methodologically, UTRL alternates the optimization of the code and test models across stages, so each model is often updated against samples produced by a previous policy. Code-A1 instead performs synchronous and adaptive adversarial co-evolution, where the Code LLM and Test LLM are optimized from their current interactions. Moreover, Code-A1 further stabilizes this interaction through code-conditioned white-box test generation and Mistake Book replay, enabling the Test LLM to expose implementation-specific failures while preventing the Code LLM from forgetting historically discovered errors. Therefore, the key improvement of Code-A1 over UTRL lies in making adversarial co-evolution more adaptive, targeted, and stable for code RL, rather than only training a stronger tester whose effect on code generation remains comparable to ground-truth-test-based supervision.

### C.4 Comparison Results

Table 11 summarizes the Qwen3-4B-scale comparison. Since the methods differ in training data, task format, and supervision assumptions, the results should be interpreted as contextual evidence rather than a fully data-identical comparison. We report HumanEval<sup>+</sup>, MBPP<sup>+</sup>, BigCodeBench, and LiveCodeBench-v2 as code-generation benchmarks, and compute Avg over all four benchmarks. Since CURE does not report LiveCodeBench-v2, its Avg is left blank to avoid mixing different averaging protocols. Under this comparison, Code-

A1 achieves the strongest reported code-generation performance among methods with complete results and remains competitive on test-generation metrics, supporting the effectiveness of using synchronous, white-box, and stabilized adversarial co-evolution as the main reward source for code RL.

## D Training Pipeline of Code-A1

Figure 6 illustrates the complete training pipeline of Code-A1. Given a question from the training dataset, the process proceeds as follows:

**Code Generation (Step 1).** The Code LLM generates  $M$  candidate solutions  $\{C_1, \dots, C_M\}$  for each question.

**Test Generation (Step 2-1).** For each candidate  $C_m$ , the Test LLM performs white-box generation: it inspects the candidate code and produces  $N$  test suites  $\{T_{m,1}, \dots, T_{m,N}\}$ , each containing multiple assertion statements.

**Mistake Book Retrieval (Step 2-2, 2-3).** The system queries the Mistake Book for historical failures  $T^{hist}$  associated with the current question, retrieving tests with the highest failure frequency to form a stable evaluation baseline.

**Test Validation (Step 3).** Each generated test suite is executed against the ground-truth code  $C_{GT}$  to compute validity scores. Tests with incorrect predictions are corrected using oracle outputs and retained to enrich coverage.

**Code Evaluation (Step 4-1).** Candidate solutions are evaluated against the combined test set (newly generated tests plus historical failures). Pass rates determine the adversarial scores for both models.

**Mistake Book Update (Step 4-2).** The Mistake Book is dynamically updated: newly failed tests are added with frequency 1, while historical tests that now pass have their frequency decremented or are removed.

**Reward Computation (Step 5).** The Code LLM receives rewards based on pass rates. The Test LLM receives composite rewards balancing validity and adversarial difficulty:  $R_T = \alpha \cdot \text{Valid} + (1 - \alpha) \cdot \text{Adv}$ . To balance training compute, only the test suite group with highest reward variance (TopVar) is selected for the Test LLM update.

## E Future Work

**Relaxing Dependence on Reference Solutions.** Future work could explore using consensus among multiple code samples or execution-based validation to reduce the reliance on ground truth implementations when validating generated tests.

**Supporting More Diverse Test Formats.** Extending Code-A1 beyond assertion-based tests to support stateful interactions, I/O-based testing, and property-based testing would broaden its applicability to more realistic programming scenarios.

**Extending to Broader Reasoning Domains.** Another promising direction is to investigate whether adversarial co-evolution can transfer to other programming languages, longer-context software engineering tasks, or non-code reasoning problems with verifiers, such as mathematical theorem proving.