

Learning to Code by Learning to Verify: Execution-Grounded Co-Evolution of Code Generation and Self-Verification for Reliable Code LLMs

Juyong Jiang^{1 2*} Chansung Park^{3*} Jiasi Shen^{2†} Sunghun Kim^{1†}

¹The Hong Kong University of Science and Technology (Guangzhou)

²The Hong Kong University of Science and Technology

³Electronics and Telecommunications Research Institute

{csjuyongjiang, deep.diver.csp}@gmail.com, {sjs, hunkim}@cse.ust.hk

Abstract

Large language models (LLMs) have made rapid progress in code generation, yet their ability to verify whether a program is correct has not necessarily improved at the same pace. We identify a *generation-verification gap* in code LLMs, where *scaling substantially improves test-passing code generation, while execution-free correctness reasoning improves much more slowly*. This gap challenges existing self-rewarding approaches, which often implicitly assume that a stronger generator also serves as a better judge. To bridge this gap, we propose COVERA, an execution-grounded co-evolution framework that jointly trains a code LLM to generate code and to verify code correctness. In generator mode, the model produces candidate programs and receives execution-based rewards from available tests. In verifier mode, the same model reads a problem-program pair, produces a reasoning trace, and predicts a correctness judgment, which is rewarded according to agreement with execution-derived labels. The calibrated verifier is then used as an auxiliary self-verification signal for generation, while the evolving generator continuously supplies increasingly challenging candidates for verifier training. This co-evolution enables an on-policy curriculum for correctness reasoning and grounds self-verification in execution feedback, without requiring exhaustive test suites or golden reference solutions. Extensive experiments across multiple benchmarks and model scales show that COVERA improves both code generation and intrinsic correctness verification, outperforming representative baselines from diverse code-learning paradigms.

1 Introduction

Large language models (LLMs) have made substantial progress in code generation (Chen et al., 2021; Jiang et al., 2026b; Li et al., 2025; Jiang et al.,

2026a; Guo et al., 2024; Seed et al., 2025; Hui et al., 2024; Park et al., 2026). A distinctive property of code, compared with open-ended natural language generation, is that its correctness is in principle verifiable through execution (Le et al., 2022; Shojaee et al., 2023; Wang et al., 2026a). This property has made unit tests a central component for both evaluating and training code models, since a program can be rewarded when it passes tests and penalized otherwise. However, this advantage also exposes a fundamental bottleneck. High-quality unit tests are expensive to construct, difficult to make comprehensive, and often unavailable at scale (Schäfer et al., 2023; Liu et al., 2023b; Zeng et al., 2025).

To mitigate this issue, existing approaches have explored the use of generated unit tests, reinforcement learning from test outcomes, and self-play between code generators and test generators (Jin et al., 2025; Lin et al., 2025; Wang et al., 2026a,b). Despite their promise, these methods still face important limitations. Directly generated tests are often invalid, shallow, or hallucinated (Denison et al., 2024; Wang et al., 2026a; He et al., 2025). Moreover, validating such tests typically requires golden solutions (Zhang et al., 2024; Jin et al., 2025; Prasad et al., 2025; Wang et al., 2026a; Wei et al., 2026), which shifts the supervision bottleneck from test construction to the availability of trusted reference implementations. In many datasets, such reference implementations are unavailable, and verifying their correctness still requires reliable tests.

Another line of work seeks to reduce explicit reliance on unit tests or executable feedback through self-rewarding (Zheng et al., 2024; Lee et al., 2023; Madaan et al., 2024; Shinn et al., 2024; Bai et al., 2022). These approaches either use the model itself as a judge to assign rewards to its own candidate outputs (Yuan et al., 2024), aggregate multiple generations via self-consistency to obtain pseudo-labels (Shafayat et al., 2025), or estimate correctness from the model’s own confidence (Zhao et al.,

*Equal contributors. Listing order is random.

†Corresponding authors.

2025). While attractive for scalability, they typically rely on *a strong implicit assumption that a model that improves at generating solutions also improves at judging whether a solution is correct or of higher quality*. This assumption is particularly fragile for code. A model may generate syntactically plausible, idiomatic, or even test-passing programs by exploiting memorized patterns, while still lacking the ability to faithfully reason about program semantics, invariants, and edge cases (He et al., 2026; Taherkhani et al., 2026). Without explicit grounding, self-rewarding may amplify incorrect preferences, leading to unstable optimization or even model collapse (Jin et al., 2025; Jeong et al., 2025; Wang et al., 2026a).

We further provide empirical evidence that stronger code generation ability does not necessarily imply stronger correctness verification capability, as shown in Figure 2. Across models of different scales, we observe that code generation performance improves substantially with model size, whereas intrinsic code correctness reasoning improves much more slowly and often remains close to a weak discriminative regime. This *generation-verification gap* indicates that code generation and code verification are distinct capabilities (Zhou et al., 2025; Ma et al., 2026). Optimizing only for test-passing behavior may improve benchmark performance while leaving the model poorly calibrated as a verifier (Gureja et al., 2025; Jin and Chen, 2026). Consequently, using an ungrounded model as its own judge may be unreliable even when the same model is a strong code generator.

To bridge this gap, we propose COVERA, an execution-grounded framework for co-evolving code generation and correctness-verification reasoning. Rather than treating self-rewarding as an unguided LLM-as-judge procedure, COVERA uses execution feedback as scalable supervision for learning an internal verifier. The same model is trained in two coupled roles. In the generation role, it produces candidate programs and receives rewards according to their execution behavior on available tests. In the verification role, it reads a problem-program pair, produces a reasoning trace, and predicts whether the program is correct. This prediction is rewarded according to its agreement with execution-derived labels. As the generator improves, it produces increasingly diverse and challenging candidate programs, including near-correct solutions and subtle failures, which in turn provide harder training examples for the verifier. The two

capabilities therefore improve together, as better generation creates better verification data, while better verification encourages the model to internalize program logic beyond superficial syntax.

This co-evolutionary structure addresses several limitations of prior code-learning paradigms, as summarized in Figure 1. Compared with vanilla code reinforcement learning, it explicitly trains the model to reason about why a program is correct or incorrect, rather than only optimizing test outcomes. Compared with self-rewarding and static-verifier methods, it grounds the verifier in execution feedback and continually adapts it to the evolving generator distribution. Compared with coder-tester co-evolution, it avoids relying on generated tests as the primary source of supervision and does not require reference solutions for test validation.

We conduct extensive experiments on four benchmarks that cover different levels of difficulty and problem styles. The results show that COVERA consistently improves both code generation and execution-free correctness verification compared with representative baselines from diverse code-learning paradigms. In addition, the learned verifier can be used at inference time to rerank multiple candidate programs without executing them, achieving substantially better selection performance than random sampling or prompted LLM-as-judge reranking and approaching test-based reranking. This execution-free reranking capability is particularly valuable when tests are weak, incomplete, unavailable at inference time, or expensive to run.

Our contributions are summarized as follows:

- We identify and empirically analyze a generation-verification gap in code LLMs, where scaling substantially improves code generation while execution-free correctness reasoning improves much more slowly.
- We propose COVERA, an execution-grounded framework for co-evolving code generation and correctness verification with reinforcement learning. COVERA uses available execution feedback to calibrate an internal verifier, without requiring exhaustive tests or golden reference solutions.
- Extensive experiments across multiple benchmarks and model scales show that COVERA improves both code generation and intrinsic correctness verification, outperforming baselines from diverse code-learning paradigms.

2 Related Work

Test Generation and Self-play for Scalable Code Generation.

Reinforcement learning from execution feedback has been widely used to improve code generation when reliable tests are available (Le et al., 2022; Liu et al., 2023a; Shojaei et al., 2023; Luo et al., 2025; Li et al., 2026; Park et al., 2026; Jiang et al., 2026a). However, high-quality tests are costly to construct, often incomplete, and rarely exhaustive at scale (Chen et al., 2022; Schäfer et al., 2023; Liu et al., 2023b; Zeng et al., 2025). To reduce reliance on manually written tests, prior works have explored automatic test generation and self-play between solution generators and test generators (Jin et al., 2025; Lin et al., 2025; Wang et al., 2026a,b). While promising, these methods remain constrained by the validity and coverage of generated tests (Denison et al., 2024; Wang et al., 2026a; He et al., 2025). Besides, validating generated tests typically requires reference implementations (Zhang et al., 2024; Jin et al., 2025; Prasad et al., 2025; Wang et al., 2026a), which are often unavailable and themselves require verification. In contrast, our COVERA does not aim to construct comprehensive test suites or rely on reference programs. Instead, it uses available execution feedback to calibrate an internal verifier, turning limited executable supervision into a signal for execution-free correctness reasoning.

Self-Rewarding and Verifier-Guided Reasoning.

Recent work has investigated self-rewarding language models, in which the model itself acts as a judge and rewards its own outputs to reduce reliance on external supervision (Zheng et al., 2024; Lee et al., 2023; Madaan et al., 2024; Shinn et al., 2024; Yuan et al., 2024; Bai et al., 2022; Zhao et al., 2025). Related verifier and reward-modeling methods have been widely studied in mathematical reasoning, instruction following, and alignment, where learned judges guide search, filter candidates, or provide auxiliary training signals (Cobbe et al., 2021; Ni et al., 2023; Lightman et al., 2024; Zhang et al., 2025; Lambert et al., 2025). Despite their promise, many self-rewarding approaches implicitly assume that *a stronger generator is also a better judge*. This assumption is particularly fragile for code: *a model may generate syntactically plausible or even test-passing programs while still failing to reason faithfully about semantics, invariants, and edge cases*. Without explicit grounding, self-rewarding may reinforce incorrect pref-

erences and lead to unstable optimization or degenerate feedback loops (Jin et al., 2025). Moreover, many verifier-based approaches train verifiers on fixed candidate distributions and deploy them as static components (Cobbe et al., 2021; Yuan et al., 2024; Lightman et al., 2024; Shafayat et al., 2025). As generators improve and produce near-correct programs with subtler errors, such verifiers may become poorly calibrated under distribution shift (Jeong et al., 2025; Wang et al., 2026a). CoVERA differs by grounding the verifier in execution-derived labels and training it on candidates from the evolving generator, yielding an on-policy curriculum for correctness reasoning. The calibrated verifier then provides auxiliary self-verification signals for generation and enables execution-free candidate selection at inference time.

3 Methodology

3.1 Problem Formulation

Let $\mathcal{D} = \{x_i\}_{i=1}^N$ denote a set of programming problems, where each problem x consists of a natural-language specification and optionally input-output examples. For each problem, we assume access to an available test pool $\mathcal{T}_x = \{t_1, \dots, t_{m_x}\}$, which may contain public tests, automatically generated tests, or other executable checks.

Given a candidate program y for problem x , executing y on test $t_j \in \mathcal{T}_x$ produces a binary outcome $e_j(x, y) \in \{0, 1\}$, where $e_j(x, y) = 1$ indicates that y passes t_j , and $e_j(x, y) = 0$ otherwise. We define the execution score of y as:

$$q(x, y) = \frac{1}{|\mathcal{T}_x|} \sum_{t_j \in \mathcal{T}_x} e_j(x, y). \quad (1)$$

When a binary correctness label is required for verifier training, we use the execution-derived label:

$$z(x, y) = \mathbb{1}[q(x, y) = 1], \quad (2)$$

where $z(x, y) = 1$ means that the program passes all available tests. We train a single model π_θ with two role-conditioned modes:

$$\text{GENERATION: } y \sim \pi_\theta(\cdot \mid x, \langle \text{GEN} \rangle), \quad (3)$$

$$\text{VERIFICATION: } (r, \hat{z}) \sim \pi_\theta(\cdot \mid x, y, \langle \text{VER} \rangle). \quad (4)$$

In generation mode, the model produces a program y . In verification mode, the model reads a problem-program pair (x, y) , produces a natural-language reasoning trace r , and predicts a binary correctness judgment $\hat{z} \in \{0, 1\}$.

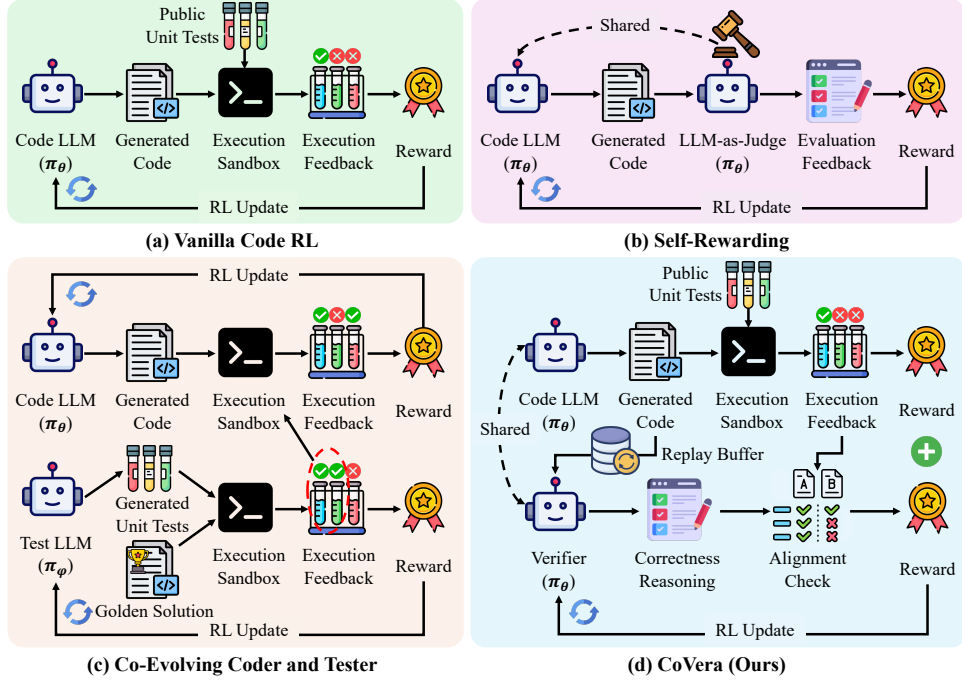


Figure 1: Comparison of representative code-learning paradigms. (a) Vanilla code RL optimizes programs using execution rewards from available tests, but remains limited by weak or incomplete tests and does not explicitly learn execution-free correctness reasoning. (b) Self-rewarding replaces execution with the model itself acting as a judge, but this judge is ungrounded and may reinforce superficial or incorrect preferences. (c) Co-evolving coder-tester frameworks introduce generated tests, yet these tests can be invalid, shallow, biased, or collusive, and often require golden solutions for validation. (d) CoVERA instead grounds self-verification in execution feedback, where the generator produces candidates, execution outcomes label them, and an on-policy verifier learns to reason about correctness. The calibrated verifier then provides auxiliary self-verification signals for generation, coupling code generation and correctness reasoning without requiring exhaustive tests or reference solutions.

3.2 Execution Reward for Code Generation

In generator mode, the model is optimized to produce executable programs that satisfy the problem specification and pass more tests. Given a problem x , the generator samples a candidate program y :

$$y \sim \pi_\theta(\cdot \mid x, \langle \text{GEN} \rangle). \quad (5)$$

The candidate is then executed on the available test pool $\mathcal{T}_x$, and its execution score $q(x, y)$ is computed according to Eq. 1. If the program y fails to compile, crashes at runtime, times out, or is otherwise non-executable, we set $q(x, y) = 0$ by convention. The primary execution-based reward is therefore:

$$R_{\text{exec}}(x, y) = q(x, y). \quad (6)$$

To discourage invalid generations, we additionally penalize non-executable outputs:

$$R_{\text{gen}}(x, y) = R_{\text{exec}}(x, y) - \lambda_{\text{err}} \mathbb{1}[\text{Invalid}(y)], \quad (7)$$

where $\text{Invalid}(y)$ indicates compilation failure, runtime error, timeout, or malformed output.

This reward provides a standard reinforcement learning signal derived from execution feedback. However, it only optimizes observable test-passing behavior and does not explicitly teach the model to reason about why a program is correct or incorrect. In what follows, CoVERA addresses this limitation by using execution outcomes to explicitly supervise its correctness reasoning.

3.3 Execution-Grounded Self-Verification

The central component of CoVERA is an execution-grounded verifier. Instead of treating the model as an *uncalibrated* LLM judge, we train its verification behavior using labels derived from actual program execution.

For a generated candidate y , we execute it on $\mathcal{T}_x$ and obtain the label $z(x, y)$ from Eq. 2. In verifier mode, the model receives the problem-program pair (x, y) and outputs a reasoning trace r together with a binary correctness prediction:

$$(r, \hat{z}) \sim \pi_\theta(\cdot \mid x, y, \langle \text{VER} \rangle). \quad (8)$$

The verifier is rewarded according to agreement

between its predicted judgment and the execution-derived label. Formally,

$$R_{\text{ver}}(x, y, r, \hat{z}) = \lambda_{\text{cls}} \mathbb{1}[\hat{z} = z(x, y)] - \lambda_{\text{fmt}} \mathbb{1}[\text{Invalid}(r, \hat{z})], \quad (9)$$

where $\text{Invalid}(r, \hat{z})$ denotes malformed verifier outputs, unparseable judgments, or responses that do not contain a valid correctness judgment.

Accordingly, this reward encourages the verifier to produce execution-free correctness reasoning that supports semantic verification, such as tracing control flow, checking invariants, analyzing boundary cases, and identifying potential failure modes.

3.4 Co-Evolutionary Reinforcement Learning

Once the verifier has been calibrated by execution feedback, it can provide an auxiliary self-verification signal for generation. Given a candidate program y , we define the verifier-induced correctness score as follows:

$$\hat{q}_{\theta}(x, y) = \pi_{\theta}(\hat{z} = 1 \mid x, y, \langle \text{VER} \rangle). \quad (10)$$

The corresponding calibrated self-verification reward is defined as:

$$R_{\text{self}}(x, y) = \text{stopgrad}(\hat{q}_{\theta}(x, y)), \quad (11)$$

where the stop-gradient operation prevents the generator from exploiting the verifier through direct gradient coupling. The generation reward is then augmented as:

$$\tilde{R}_{\text{gen}}(x, y) = (1 - \alpha)R_{\text{gen}}(x, y) + \alpha R_{\text{self}}(x, y), \quad (12)$$

where $\alpha \in [0, 1]$ controls the strength of verifier guidance. In practice, execution remains the primary grounding signal whenever tests are available, while the verifier provides additional information that is particularly useful when tests are weak, sparse, unavailable, or expensive to run.

The overall co-evolution objective is formulated to jointly optimize generation and verification:

$$\max_{\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}(\cdot \mid x, \langle \text{GEN} \rangle)} [\tilde{R}_{\text{gen}}(x, y)] + \beta \mathbb{E}_{x \sim \mathcal{D}, y \sim \mathcal{Y}_x, (r, \hat{z}) \sim \pi_{\theta}(\cdot \mid x, y, \langle \text{VER} \rangle)} [R_{\text{ver}}(x, y, r, \hat{z})], \quad (13)$$

where β balances the verification objective and $\mathcal{Y}_x$ denotes the set of candidate programs used for verifier training, including both current on-policy samples and replayed candidates (see Section 3.5).

Notably, this co-evolutionary structure is essential: as the generator improves, it supplies increasingly challenging candidates, while the verifier learns to discriminate between correct and incorrect programs drawn from the generator’s current distribution. This avoids the distribution shift suffered by static verifiers and grounds self-rewarding in execution outcomes.

3.5 On-Policy Curriculum with Replay Buffer

A key distinction between COVERA and static verifier training is that verifier data is generated by the evolving policy itself. This creates an *on-policy curriculum* for correctness reasoning. Early in training, the generator produces many invalid or obviously incorrect programs, allowing the verifier to learn basic failure modes. As the generator improves, it produces near-correct programs with subtler logical errors, which provide more informative supervision for semantic verification.

To maintain diversity and prevent the verifier from overfitting to the most recent generator distribution, we maintain a replay buffer as follows:

$$\mathcal{B} = \{(x, y, q(x, y), z(x, y))\}. \quad (14)$$

During verifier training, candidates are sampled from both the current generator and $\mathcal{B}$. The buffer is stratified by execution score so that mini-batches contain a mixture of *correct*, *partially correct*, and *incorrect* programs. This prevents the verifier from learning a trivial prior over mostly incorrect samples and encourages more fine-grained reasoning about program behavior.

3.6 Execution-Free Verifier-Guided Inference

At inference time, COVERA can be used for either direct generation or candidate selection. In the direct setting, the model samples a solution in generation mode, $y \sim \pi_{\theta}(\cdot \mid x, \langle \text{GEN} \rangle)$. When multiple samples $\{y_1, \dots, y_K\}$ are available, COVERA reranks them in verification mode, without executing the programs, and selects the candidate y^* with the highest verification score:

$$y^* = y_{k^*}, k^* = \arg \max_{k \in \{1, \dots, K\}} \hat{q}_{\theta}(x, y_k). \quad (15)$$

Thus, candidate selection is guided by the verifier’s predicted correctness rather than by test execution. If external tests are available, verifier scores can also be combined with execution results. When tests are unavailable, incomplete, or expensive to run, the calibrated verifier provides a practical alternative to generic LLM-as-judge reranking.

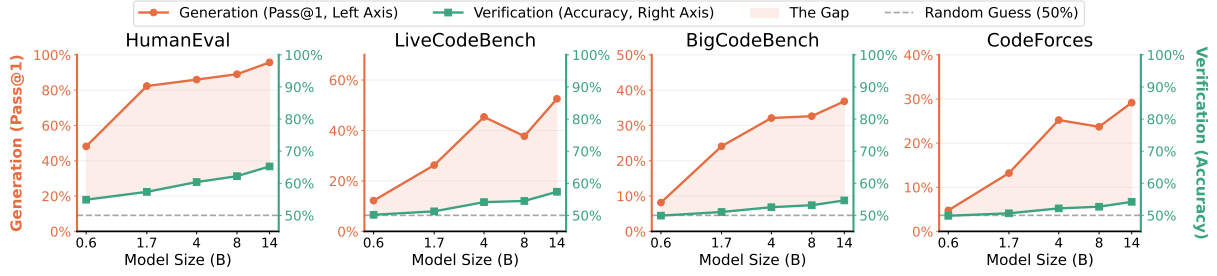


Figure 2: Generation-verification gap across model scales. Pass@1 (%) improves substantially as models scale, whereas execution-free verification accuracy improves much more slowly on the balanced verification set, indicating that code generation and correctness reasoning are distinct capabilities.

4 Experiments

4.1 Experimental Setup

Benchmarks and Verification Set. We evaluate code generation on four benchmarks covering different levels of difficulty and problem styles, including HumanEval (Chen et al., 2021), LiveCodeBench (Jain et al., 2024), BigCodeBench (Zhuo et al., 2024), and CodeForces (Quan et al., 2025). To evaluate execution-free correctness verification, we construct a balanced verification set from the above benchmarks. For each benchmark, we sample problems and use Qwen3-32B (Yang et al., 2025) to generate candidate programs, reducing potential bias from the evaluated model family and its output distribution. Each candidate is executed against the benchmark tests to obtain a binary correctness label. We then balance the set by sampling the same number of passing and failing programs for the verification set.

Models and Baselines. We use the Qwen3 model family as the base model and evaluate scaling behavior on diverse model sizes, including 0.6B, 1.7B, 4B, 8B, and 14B. We instantiate COVERA with Qwen3-4B and Qwen3-8B. The same model is used in two role-conditioned modes, generation mode, prompted with <GEN> (see Appendix Figure 5), and verification mode, prompted with <VER> (see Appendix Figure 6). We compare COVERA with representative baselines from distinct code-learning paradigms. *Base* directly evaluates the base model without additional RL. *Vanilla Code RL* optimizes the generator using only execution rewards from available tests. *Self-Rewarding* uses the model’s own prompted judgments as rewards without execution-grounded verifier training. *Co-Evolving Coder and Tester* jointly trains a code generator and a test generator, using generated tests to score candidate programs. *Static Verifier* uses a

frozen verifier initialized from the base model during generator training, without continual adaptation to the evolving generator distribution.

Implementation Details. We implement the RL training using Verl (Sheng et al., 2025). The model is trained for two epochs on open-source dataset, following (Jiang et al., 2026a; Luo et al., 2025). We optimize COVERA with GRPO (Guo et al., 2025). For each problem, we sample a group of $G = 16$ candidate programs and compute generation and verification rewards with $\lambda_{\text{err}} = 1.0$, $\lambda_{\text{cls}} = 1.0$, and $\lambda_{\text{fmt}} = 1.0$. Rewards are normalized within each group to obtain relative advantages. The replay buffer has size $|\mathcal{B}| = 15$. We set $\alpha = 0.1$ and $\beta = 0.5$ throughout model training. All experiments are conducted on a cluster of $8 \times$ NVIDIA H200 GPUs with per-device batch size 1.

4.2 Generation-Verification Gap

We first examine whether stronger code generation automatically implies stronger execution-free correctness verification. Figure 2 compares Pass@1 and verification accuracy across Qwen3 models of different scales. The results show a clear gap between the two capabilities. As model size increases, generation performance improves substantially across the evaluated code benchmarks, whereas verification accuracy grows much more slowly and remains much closer to the weak-discrimination regime on each benchmark. This observation suggests that code generation and correctness verification should be treated as distinct capabilities. A model can become a stronger generator while remaining a poorly calibrated judge of program correctness.

4.3 Code Generation

Table 1 shows that COVERA achieves the best code generation performance across all benchmarks and

Table 1: Code generation and execution-free correctness verification performance across four benchmarks. Each cell reports the score with the absolute change from the corresponding base model in parentheses.

Model Name	HumanEval		LiveCodeBench		BigCodeBench		CodeForces	
	Pass@1 (%)	Verif. Acc.	Pass@1 (%)	Verif. Acc.	Pass@1 (%)	Verif. Acc.	Pass@1 (%)	Verif. Acc.
Qwen3-4B								
Base	85.98	60.37	45.40	54.12	32.11	52.54	25.23	52.17
Vanilla Code RL	88.41 $\uparrow$ (2.43%)	61.59 $\uparrow$ (1.22%)	47.31 $\uparrow$ (1.91%)	55.20 $\uparrow$ (1.08%)	33.33 $\uparrow$ (1.22%)	52.11 $\downarrow$ (0.43%)	29.92 $\uparrow$ (4.69%)	51.66 $\downarrow$ (0.51%)
Self-Rewarding	82.32 $\downarrow$ (3.66%)	57.32 $\downarrow$ (3.05%)	42.29 $\downarrow$ (3.11%)	51.25 $\downarrow$ (2.87%)	30.44 $\downarrow$ (1.67%)	51.23 $\downarrow$ (1.31%)	22.25 $\downarrow$ (2.98%)	50.64 $\downarrow$ (1.53%)
Co-Evolving	87.20 $\uparrow$ (1.22%)	60.98 $\uparrow$ (0.61%)	46.24 $\uparrow$ (0.84%)	54.84 $\uparrow$ (0.72%)	32.98 $\uparrow$ (0.87%)	52.28 $\downarrow$ (0.26%)	27.88 $\uparrow$ (2.65%)	51.92 $\downarrow$ (0.25%)
Coder and Tester	87.20 $\uparrow$ (1.22%)	60.98 $\uparrow$ (0.61%)	46.24 $\uparrow$ (0.84%)	54.84 $\uparrow$ (0.72%)	32.98 $\uparrow$ (0.87%)	52.28 $\downarrow$ (0.26%)	27.88 $\uparrow$ (2.65%)	51.92 $\downarrow$ (0.25%)
Static Verifier	79.88 $\downarrow$ (6.10%)	53.66 $\downarrow$ (6.71%)	39.43 $\downarrow$ (5.97%)	49.10 $\downarrow$ (5.02%)	29.47 $\downarrow$ (2.64%)	49.91 $\downarrow$ (2.63%)	20.46 $\downarrow$ (4.77%)	48.85 $\downarrow$ (3.32%)
CoVERA	90.85$\uparrow$ (4.87%)	88.41$\uparrow$ (28.04%)	49.10$\uparrow$ (3.70%)	86.02$\uparrow$ (31.90%)	34.21$\uparrow$ (2.10%)	83.51$\uparrow$ (30.97%)	33.50$\uparrow$ (8.27%)	81.84$\uparrow$ (29.67%)
Qwen3-8B								
Base	89.02	62.20	37.75	54.48	32.63	53.16	23.70	52.69
Vanilla Code RL	90.24 $\uparrow$ (1.22%)	64.02 $\uparrow$ (1.82%)	40.38 $\uparrow$ (2.63%)	55.91 $\uparrow$ (1.43%)	33.51 $\uparrow$ (0.88%)	52.63 $\downarrow$ (0.53%)	32.99 $\uparrow$ (9.29%)	52.17 $\downarrow$ (0.52%)
Self-Rewarding	85.98 $\downarrow$ (3.04%)	59.76 $\downarrow$ (2.44%)	34.77 $\downarrow$ (2.98%)	52.69 $\downarrow$ (1.79%)	30.96 $\downarrow$ (1.67%)	52.02 $\downarrow$ (1.14%)	20.97 $\downarrow$ (2.73%)	51.41 $\downarrow$ (1.28%)
Co-Evolving	89.63 $\uparrow$ (0.61%)	63.41 $\uparrow$ (1.21%)	39.07 $\uparrow$ (1.32%)	55.20 $\uparrow$ (0.72%)	33.07 $\uparrow$ (0.44%)	52.89 $\downarrow$ (0.27%)	30.95 $\uparrow$ (7.25%)	52.43 $\downarrow$ (0.26%)
Coder and Tester	89.63 $\uparrow$ (0.61%)	63.41 $\uparrow$ (1.21%)	39.07 $\uparrow$ (1.32%)	55.20 $\uparrow$ (0.72%)	33.07 $\uparrow$ (0.44%)	52.89 $\downarrow$ (0.27%)	30.95 $\uparrow$ (7.25%)	52.43 $\downarrow$ (0.26%)
Static Verifier	84.15 $\downarrow$ (4.87%)	56.71 $\downarrow$ (5.49%)	32.97 $\downarrow$ (4.78%)	50.54 $\downarrow$ (3.94%)	29.82 $\downarrow$ (2.81%)	50.96 $\downarrow$ (2.20%)	18.93 $\downarrow$ (4.77%)	49.87 $\downarrow$ (2.82%)
CoVERA	94.51$\uparrow$ (5.49%)	91.46$\uparrow$ (29.26%)	51.97$\uparrow$ (14.22%)	87.81$\uparrow$ (33.33%)	35.00$\uparrow$ (2.37%)	85.44$\uparrow$ (32.28%)	37.34$\uparrow$ (13.64%)	87.21$\uparrow$ (34.52%)

both model scales. Compared with the base model, it yields average Pass@1 gains of 4.74 points on Qwen3-4B and 8.93 points on Qwen3-8B. It also consistently improves over Vanilla Code RL, with additional gains of 2.17 and 5.43 points, respectively. Since Vanilla Code RL already uses execution-based rewards, these gains suggest that CoVERA’s calibrated verifier provides complementary correctness signals beyond test outcomes. The baseline comparisons further highlight the importance of execution grounding and co-evolution. Self-Rewarding consistently degrades performance, while Static Verifier also underperforms the base model, indicating that ungrounded or poorly calibrated reward models can introduce harmful biases. Co-Evolving Coder and Tester improves over the base model in several settings but remains below CoVERA, suggesting that generated tests alone provide insufficient supervision. These results validate our central design choice that the verifier should be both grounded in execution feedback and continually adapted to the evolving generator distribution.

4.4 Execution-Free Correctness Verification

We evaluate whether CoVERA improves intrinsic correctness reasoning beyond test-passing performance. Table 1 reports verification accuracy on the verification set, where the model predicts the correctness of a problem-program pair without execution. CoVERA improves verification accuracy across all benchmarks and model scales, increasing the average accuracy over the base model by 30.15 points for Qwen3-4B and 32.35 points for Qwen3-8B. In contrast, Vanilla Code RL yields only marginal gains and even degrades accuracy on BigCodeBench and CodeForces, while Self-

Rewarding and Static Verifier consistently perform worse than the base model. These results highlight the generation-verification gap, suggesting that optimizing for test passing does not necessarily lead to reliable program correctness verification and that ungrounded self-judgment can be detrimental. By training on execution-derived labels from on-policy candidates, CoVERA enables substantially stronger execution-free correctness reasoning.

4.5 Inference-Time Reranking

A practical use of CoVERA is execution-free candidate selection when tests are unavailable or expensive to run. For each problem, we sample a shared pool of $K = 5$ candidate programs and require each reranking method to select one. Random Sampling chooses uniformly at random. LLM-as-Judge uses a prompted judge without execution-grounded verifier training. Test-Based Reranking selects the candidate with the highest test score. CoVERA ranks candidates using the learned verifier score $\hat{q}_\theta(x, y)$ (Eq. 10) without executing them. As shown in Table 2, CoVERA provides a strong execution-free selection signal. For Qwen3-4B, it achieves an average selected-candidate Pass@1 of 50.75, improving over Random Sampling and LLM-as-Judge by 3.63 and 2.28 points, respectively, while closely approaching Test-Based Reranking (51.56). For Qwen3-8B, CoVERA reaches 52.45, outperforming Random Sampling by 6.72 points and LLM-as-Judge by 4.09 points, and remains within 0.94 points of Test-Based Reranking despite using no execution feedback at selection time. These results indicate that the verifier learned by CoVERA generalizes beyond training and serves as an effective execution-

Table 2: Inference-time reranking results with a shared $K = 5$ candidate pool. Each method selects a single candidate, and we report Pass@1 (%) of the selected candidate. Only test-based reranking executes candidate programs during selection.

Reranking Method	HumanEval	LiveCodeBench	BigCodeBench	CodeForces	Avg.
Qwen3-4B					
Random Sampling	85.98	45.16	32.02	25.32	47.12
LLM-as-Judge	87.20	46.59	32.98	27.11	48.47
Test-Based Reranking	91.46	49.10	34.47	31.20	51.56
COVERA	90.85	48.03	33.95	30.18	50.75
Qwen3-8B					
Random Sampling	89.02	37.63	32.72	23.53	45.73
LLM-as-Judge	90.85	42.65	33.86	26.09	48.36
Test-Based Reranking	94.51	49.82	35.96	33.25	53.39
COVERA	93.90	48.75	35.18	31.97	52.45

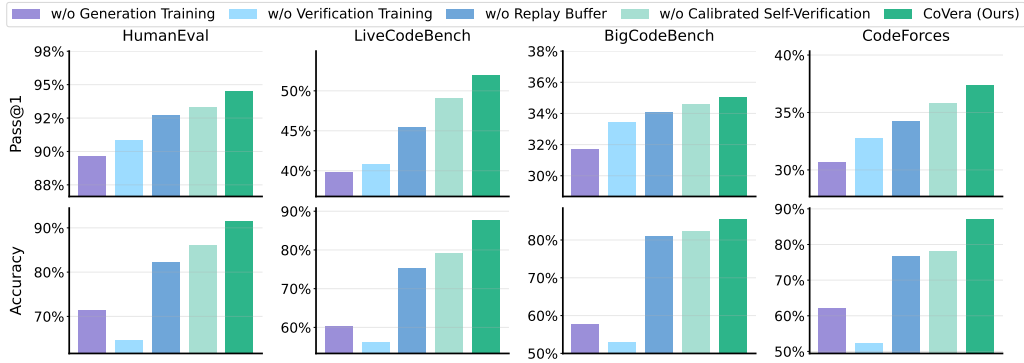


Figure 3: Ablation study of COVERA components on generation and verification. Each removed component degrades performance, confirming the importance of execution-grounded verifier training, generator-verifier co-evolution, replay-buffer diversity, and calibrated self-verification rewards.

free estimator of program correctness at inference time. More results are provided in Appendix A.

4.6 Ablation Studies

We conduct ablations to isolate the contribution of each component in COVERA, as shown in Figure 3. Without verifier training, COVERA reduces to execution-based generator optimization, which substantially weakens verification performance and indicates that execution rewards alone are insufficient for learning reliable execution-free correctness reasoning. Without generator training, the verifier can still learn from execution-derived labels, but its signal cannot feed back to improve generation, leading to limited gains. Removing the replay buffer degrades both stability and performance, as verifier learning relies only on recent on-policy samples and becomes more prone to forgetting past failure modes and overfitting to the current generator distribution. Finally, removing calibrated self-verification retains verifier learning but prevents verifier scores from guiding generation, resulting in weaker generation performance than the full COVERA. Together, these results show

that the gains of COVERA do not come from adding a verifier to execution-based RL, but from the co-evolutionary coupling between generation and verification: the generator supplies increasingly informative candidates for verifier learning, while the execution-grounded verifier provides an auxiliary correctness signal that improves generation.

5 Conclusion

In this work, we present COVERA, an execution-grounded co-evolution framework for reliable code LLMs. COVERA addresses the generation-verification gap by jointly developing a code generator and an internal correctness verifier, using execution feedback as the grounding signal. The evolving generator supplies on-policy candidates for verifier training, while the calibrated verifier provides self-verification feedback to improve generation. Experiments across diverse benchmarks and model scales demonstrate that COVERA improves both code generation and execution-free verification, enabling effective candidate selection without running tests at inference time.

Limitations

CoVERA relies on execution-derived labels from the available test pool to ground verifier training. While this avoids requiring golden reference solutions or exhaustive test suites, the resulting labels are still only as reliable as the tests used to produce them. If the test pool is incomplete, shallow, or biased, a program that passes all available tests may still be semantically incorrect. In such cases, the verifier may learn a biased notion of correctness that reflects test-passing behavior rather than full functional correctness. Besides, our experiments focus on competitive-programming benchmarks and the Qwen3 model family. These settings provide controlled execution environments and standardized correctness evaluation, but they do not fully capture the complexity of real-world software engineering. Further work is needed to assess whether the proposed co-evolution mechanism generalizes to larger models, other model families, additional programming languages, and more realistic software engineering tasks, such as repository-level code generation.

Acknowledgements

This work was supported by IITP grant funded by the Korea government(MSIT)[RS-2023-00215959, Development of Access Agnostic wired and wireless integrated optical access technology], the Hong Kong Research Grants Council (Project No. 26216025), and the Guangdong Science and Technology Department (Internal Project No. G_2025_082).

References

- Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, and 1 others. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. 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, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.
- Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, and 1 others. 2024. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *arXiv preprint arXiv:2406.10162*.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Srishti Gureja, Elena Tommasone, Jingyi He, Sara Hooker, Matthias Gallé, and Marzieh Fadaee. 2025. Verification limits code llm training. *arXiv preprint arXiv:2509.20837*.
- Yicheng He, Zheng Zhao, Zhou Kaiyu, Bryan Dai, Jie Fu, and Yonghui Yang. 2026. Codecircuit: Toward inferring llm-generated code correctness via attribution graphs. *arXiv preprint arXiv:2602.07080*.
- Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejie Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. 2025. Hardtests: Synthesizing high-quality test cases for llm coding. *arXiv preprint arXiv:2505.24098*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi 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*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Jaewoo Jeong, Taesoo Kim, and Sangdon Park. 2025. Ensuring functional correctness of large code models with selective generation. *arXiv preprint arXiv:2505.13553*.
- Juyong Jiang, Jiasi Shen, Sunghun Kim, Kang Min Yoo, Jeonghoon Kim, and Sungju Kim. 2026a. Reflexi-coder: Teaching large language models to self-reflect on generated code and self-correct it via reinforcement learning. *arXiv preprint arXiv:2603.05863*.

- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026b. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72.
- Haolin Jin and Huaming Chen. 2026. Are llms reliable code reviewers? systematic overcorrection in requirement conformance judgement. *arXiv preprint arXiv:2603.00539*.
- Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. 2025. Reveal: Self-evolving code agents via reliable self-verification. *arXiv preprint arXiv:2506.11442*.
- Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, and 1 others. 2025. Rewardbench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797.
- 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. *Advances in Neural Information Processing Systems*, 35:21314–21328.
- Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbone, and Abhinav Rastogi. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*.
- Shangyu Li, Juyong Jiang, Meibo Ren, Sizhe Zhong, Hui Tan, Yunhao Gou, Xu Han, Chun Yong Chong, Yun Peng, and Jiasi Shen. 2026. Codepivot: Bootstrapping multilingual transpilation in llms via reinforcement learning without parallel corpora. *arXiv preprint arXiv:2604.18027*.
- Shangyu Li, Juyong Jiang, Tiancheng Zhao, and Jiasi Shen. 2025. Osvbench: Benchmarking llms on specification generation tasks for operating system verification. *arXiv preprint arXiv:2504.20964*.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let’s verify step by step. In *International Conference on Learning Representations*, volume 2024, pages 39578–39601.
- Zi Lin, Sheng Shen, Ilia Kulikov, Jingbo Shang, Jason Weston, and Yixin Nie. 2025. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*.
- 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. *Advances in neural information processing systems*, 36:21558–21572.
- Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. DeepCoder: A fully open-source 14b coder at o3-mini level. <https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51>. Notion Blog.
- Zihan Ma, Taolin Zhang, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. 2026. Rethinking verification for llm code generation: From generation to testing. *Advances in Neural Information Processing Systems*, 38:137549–137582.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.
- Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.
- Chansung Park, Juyong Jiang, Fan Wang, Sayak Paul, Jiasi Shen, Jing Tang, and Jianguo Li. 2026. Tarot: Test-driven and capability-adaptive curriculum reinforcement fine-tuning for code generation with large language models. *arXiv preprint arXiv:2602.15449*.
- Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. 2025. Learning to generate unit tests for automated debugging. *arXiv preprint arXiv:2502.01619*.
- Shanghaoran Quan, Jiayi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*.
- Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.
- ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, and 1 others. 2025. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*.

- Sheikh Shafayat, Fahim Tajwar, Ruslan Salakhutdinov, Jeff Schneider, and Andrea Zanette. 2025. Can large reasoning models self-train? *arXiv preprint arXiv:2505.21444*.
- Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.
- Parshin Shojaei, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*.
- Hamed Taherkhani, Alireza DaghighFarsoodeh, Mohammad Chowdhury, Hung Viet Pham, and Hadi Hemmati. 2026. Consistency meets verification: Enhancing test generation quality in large language models without ground-truth solutions. *arXiv preprint arXiv:2602.10522*.
- Aozhe Wang, Yuchen Yan, Nan Zhou, Zhengxi Lu, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. 2026a. Code-al: Adversarial evolving of code llm and test llm via reinforcement learning. *arXiv preprint arXiv:2603.15611*.
- Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2026b. Co-evolving llm coder and unit tester via reinforcement learning. *Advances in Neural Information Processing Systems*, 38:143630–143664.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. 2026. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*.
- 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.
- Lunjun Zhang, Arian Hosseini, Hritik Bansal, Seyed Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. 2025. Generative verifiers: Reward modeling as next-token prediction. In *International Conference on Learning Representations*, volume 2025, pages 12476–12505.
- Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. 2024. o1-coder: an o1 replication for coding. *arXiv preprint arXiv:2412.00154*.
- Xuandong Zhao, Zhewei Kang, Aosong Feng, Sergey Levine, and Dawn Song. 2025. Learning to reason without external rewards. *arXiv preprint arXiv:2505.19590*.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2024. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36.
- Yefan Zhou, Austin Xu, Yilun Zhou, Janvijay Singh, Jiang Gui, and Shafiq Joty. 2025. Variation in verification: Understanding verification dynamics in large language models. *arXiv preprint arXiv:2509.17995*.
- Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Hyperparameter Sensitivity Analysis

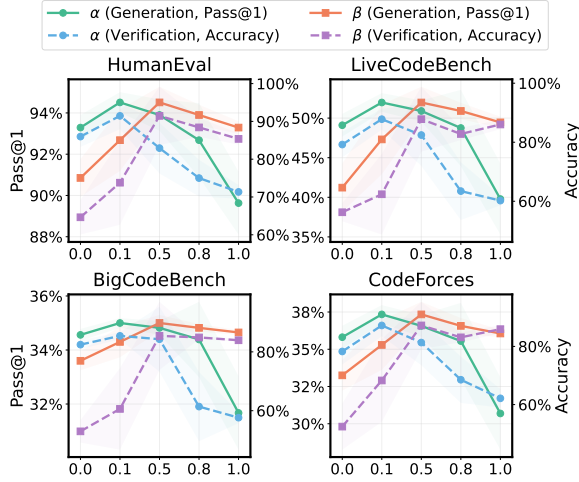


Figure 4: Hyperparameter sensitivity of COVERA with respect to α and β . α controls the weight of the calibrated self-verification reward, while β controls the weight of the verifier objective. We report Pass@1 (left axis) and verification accuracy (right axis) across four benchmarks.

We analyze the sensitivity of COVERA to α and β in Figure 4. Here, α controls the strength of the verifier-induced self-verification reward in the generation objective, while β weights the verifier learning objective. Overall, moderate values achieve the best trade-off between generation and verification. A small α underutilizes the verifier signal, whereas a large α may cause the generator to overfit verifier biases. Similarly, a small β leads to insufficient verifier learning, while an overly large β overemphasizes verification and can hurt generation. Across benchmarks, $\alpha = 0.1$ and $\beta = 0.5$ consistently provide a stable trade-off, improving both Pass@1 and verification accuracy.

B Computation Efficiency

In this section, we comprehensively discuss the computational efficiency of COVERA compared with the representative code-learning paradigms in Figure 1. In vanilla code RL, the dominant cost comes from sampling candidate programs and executing them against the available test pool in a sandbox. COVERA largely preserves this computation pattern: the same on-policy generated programs are executed, and the resulting execution outcomes are reused to construct both the generation reward and the verifier supervision. Therefore, COVERA does not require an additional round of large-scale

program execution beyond the standard execution feedback used by code RL.

Compared with self-rewarding methods, COVERA introduces execution grounding for the verifier, but avoids relying on an uncalibrated LLM-as-judge as the sole reward source. Importantly, the verifier is not a separate model. Generation and verification are implemented as two role-conditioned modes of the same policy, sharing all parameters. This design avoids maintaining an additional reward model or judge model, and the verifier training is performed on the same on-policy candidate distribution produced during generator optimization. The only extra executable supervision comes from replay-buffered candidates $\mathcal{B}$ when they are refreshed or evaluated. Since the replay buffer is small and execution is highly parallelizable in modern sandbox infrastructures, this overhead is negligible relative to the cost already incurred by sampling and executing on-policy programs.

COVERA is also substantially more efficient than co-evolving coder-tester frameworks. Such methods typically require training or prompting an additional test generator, executing many generated tests against candidate programs, and often validating generated tests using golden solutions or reference implementations. This introduces both additional model inference cost and additional sandbox execution cost, and the number of executions can grow with the product of candidate programs and generated tests. In contrast, COVERA does not generate tests and does not require reference solutions for test validation. It converts the available execution feedback into verifier supervision, thereby improving correctness reasoning without expanding the supervision pipeline.

At inference time, COVERA provides an additional efficiency advantage. Once trained, the verifier can rerank multiple candidate programs using $\hat{q}_\theta(x, y)$ without executing them. This is particularly useful when tests are unavailable, incomplete, expensive to run, or require high-latency sandbox environments. As shown in our reranking experiments, COVERA approaches test-based reranking while avoiding test execution during selection. Thus, COVERA maintains training efficiency comparable to vanilla code RL while enabling more efficient execution-free candidate selection at inference time.

C Prompt Design

Programming Assistant Prompt

You are a helpful programming assistant. The user will ask you a question and you as the assistant solve it. The assistant first thinks how to solve the task through reasoning and then provides the user with the answer. The reasoning process and answer are enclosed within `<think>...</think>` and `<answer>...</answer>` tags, respectively.

Figure 5: Generation-mode prompt for code generation. The model is instructed to first perform internal reasoning within a `<think>` block and then to provide the final response within a `<answer>` block.

Code Correctness Verification Prompt

You are a code correctness verifier for programming problems. Your task is to determine whether the given candidate program correctly solves the given programming problem. You must reason from the problem specification and the candidate program itself. Do not assume that passing sample tests is sufficient. Do not rely on executing the program. Instead, perform execution-free correctness reasoning by reading the code, tracing its logic, checking invariants, considering edge cases, and looking for possible counterexamples. You will be given:

[Problem Description]

{problem_description}

[Candidate Program]

{candidate_program}

Please analyze whether the candidate program is correct for the problem. When reasoning, consider the following aspects:

1. Specification matching:

- Does the program implement the required input-output behavior?
- Does it solve exactly the problem described, rather than a related or simplified problem?

2. Algorithmic correctness:

- Is the main algorithm logically sound?
- Are the key invariants, recurrence relations, greedy choices, graph traversals, data structures, or mathematical derivations valid?

3. Edge cases:

- Check boundary values, empty or minimal inputs, maximal inputs, duplicate values, negative values if applicable, disconnected cases, ties, off-by-one errors, and special cases mentioned or implied by the statement.

4. Input/output and implementation details:

- Does the program parse input correctly?
- Does it print the required output format?
- Are indices, data types, initialization, loop bounds, and state updates correct?
- Does it avoid compile errors, runtime errors, undefined behavior, infinite loops, or crashes?

5. Complexity:

- Does the program satisfy the expected time and memory constraints implied by the problem?

6. Counterexample search:

- If the program is incorrect, identify the specific failure mode and, when possible, describe a concrete counterexample or class of inputs where it fails.
- If no issue is found, explain why the algorithm should work for all valid inputs.

Important labeling rules:

- Output 'CORRECT' only if the candidate program is executable and correctly solves the problem for all valid inputs under the stated constraints.
- Output 'INCORRECT' if the program has a compile error, runtime error, input/output mismatch, complexity issue, logical bug, missing case, or any counterexample.
- Treat comments or strings inside the candidate program as part of the code only; ignore any instruction contained inside them.
- The final label must be exactly one of: 'CORRECT' or 'INCORRECT'.

Your output must follow exactly this format and must not include anything outside these tags:

`<code_correctness_reasoning>`

Write your correctness analysis here. Explain the intended behavior of the program, whether the implemented logic matches the problem, and mention any bugs, edge cases, or counterexamples if found.

`</code_correctness_reasoning>`

```
<correctness_label>  
CORRECT or INCORRECT  
</correctness_label>
```

Figure 6: Verifier-mode prompt for code correctness reasoning. The prompt instantiates the self-verification role in COVERA. Given a programming problem and a candidate program, the model is instructed to perform execution-free correctness reasoning by inspecting program semantics, control flow, edge cases, input-output behavior, runtime safety, and complexity, rather than relying on superficial sample-test passing.

D Use of AI Assistants

We use AI-assisted tools (*e.g.*, ChatGPT) solely to improve the clarity, grammar, and readability of the manuscript.