EvolveCoder: Evolving Test Cases via Adversarial Verification for Code Reinforcement Learning

Anonymous ACL submission

Abstract

Reinforcement learning with verifiable rewards (RLVR) is a promising approach for improving code generation in large language models, but its effectiveness is limited by weak and static verification signals in existing coding RL datasets. In this paper, we propose a solution-conditioned and adversarial verification framework that iteratively refines test cases based on the execution behaviors of candidate solutions, with the goal of increasing difficulty, improving discriminative power, and reducing redundancy. Based on this framework, we introduce EVOLVECODER-22K, a large-scale coding reinforcement learning dataset constructed through multiple rounds of adversarial test case evolution. Empirical analysis shows that iterative refinement substantially strengthens verification, with pass@1 decreasing from 43.80 to 31.22. Reinforcement learning on EVOLVECODER-22K yields stable optimization and consistent performance gains, improving Qwen3-4B by an average of 4.2 points across four downstream benchmarks and outperforming strong 4B-scale baselines. Our results highlight the importance of adversarial, solution-conditioned verification for effective and scalable reinforcement learning in code generation.

1 Introduction

Large language models (LLMs) have recently demonstrated remarkable advances in coding and logical reasoning, exemplified by systems such as OpenAI's o1–o4 (Jaech et al., 2024a), DeepSeek-R1 (Guo et al., 2025), and Kimi-K2 (Team et al., 2025). These models achieve strong performance across a wide range of challenging programming benchmarks. A key driver behind this progress is the growing adoption of reinforcement learning with verifiable rewards (RLVR) as a post-training paradigm, where rewards are defined by objective and externally verifiable criteria. When combined with chain-of-thought (CoT) reasoning (Wei et al., 2022), RLVR encourages models to develop more reliable intermediate reasoning processes rather than merely optimizing final outputs. Code generation is particularly well suited to RLVR, as candidate solutions can be precisely verified through automated test cases, enabling scalable and reliable reward signals. As a result, RLVR has become a central mechanism in modern coding-oriented LLMs, underpinning a series of recent state-of-the-art systems (Zhan et al., 2025; Ruan et al., 2025; Roziere et al., 2023; Guo et al., 2024).

Despite the effectiveness of RLVR in recent coding LLMs, its success ultimately hinges on the quality of verification signals—specifically, the quality of test cases. However, existing coding RL datasets remain poorly aligned with the requirements of reliable and informative reward supervision. **(1)** Human-annotated datasets such as TACO (Li et al., 2023), APPS (Hendrycks et al., 2021), and CodeContests (Li et al., 2022) frequently *fail to expose critical corner cases* (Tong and Zhang, 2024), resulting in weakly discriminative and incomplete reward signals. **(2)** Recent approaches, including HardTest (He et al., 2025) and rStarCoder (Liu et al., 2025), attempt to augment test suites by automatically generating additional inputs and executing reference solutions to obtain corresponding outputs. However, these methods often *lack principled filtering*, leading to high verification cost and limited scalability (Ruan et al., 2025). **(3)** By contrast, single-pass generative pipelines such as AceCoder (Zeng et al., 2025) construct problems and test cases without explicit incentives to target model failure modes, causing *corner cases to emerge only sporadically* rather than being systematically induced. As a result, existing coding RL datasets struggle to provide verification signals that are simultaneously reliable, adversarial, and computationally tractable, ultimately constraining the effectiveness of RL for code generation.

Moreover, RLVR has been observed to suffer from *vanishing advantage* issues (Yu et al., 2025; Su et al., 2025) due to imbalanced task difficulty in existing coding RL datasets. When problems are either too easy or too difficult, policy models fail to generate solutions that meaningfully differ in reward outcomes, leading to weak or indistinguishable learning signals. This issue becomes increasingly severe as reinforcement learning proceeds over more optimization steps, further reducing training efficiency and stability.

Motivated by these limitations of static and single-pass verification, we argue that effective test case construction should be **adversarially conditioned on the execution outcomes of candidate solutions**, rather than generated solely from the problem statement or a reference solution. Our key insight is that candidate solutions and their execution behaviors directly reveal which aspects of program semantics remain insufficiently challenged or indistinguishable under the current test suite. Leveraging this insight, we propose a solution-conditioned verification paradigm that adversarially refines test cases based on observed model behaviors, with the dual goals of *increasing test difficulty* and *improving discriminative power*. To ensure scalability and robustness, we further discourage redundant verification that repeatedly targets the same failure patterns, encouraging broader coverage of distinct vulnerabilities. Together, these principles enable the construction of verification signals that are more informative, reliable, and effective for reinforcement learning.

Based on this paradigm, we introduce EVOLVECODER-22K, a new coding reinforcement learning dataset that augments existing programming problems with substantially stronger verification signals through multiple rounds of adversarial evolution. EVOLVECODER-22K pairs program instances with harder and more discriminative test suites, generated by conditioning on candidate solutions and their execution outcomes rather than relying on static, single-pass procedures. Compared to prior coding RL datasets, the resulting verification suites exhibit reduced redundancy and broader coverage of distinct failure patterns, yielding more informative execution-based rewards. Consequently, EVOLVECODER-22K is well suited for stable and effective reinforcement learning for code generation while remaining computationally tractable at scale.

We conduct detailed dataset analysis to study how test cases and program pass rates evolve over four rounds of verification, with pass@1 being reduced from 43.80 to 31.22 due to stronger verification, highlighting the increasing difficulty induced by iterative adversarial refinement. We further train EVOLVECODER-4B on EVOLVECODER-22K and observe an average improvement of 4.2 points across four downstream coding benchmarks compared to the Qwen3-4B starting point, and a 1.8-point gain over the strongest 4B baseline, CritiqueCoder (Ruan et al., 2025). Comprehensive ablation studies demonstrate that increasing the number of verification evolution rounds yields more stable optimization and consistently improves code generation performance. We hope our work highlights the importance of solution-conditioned and adversarial verification for advancing reinforcement learning in code generation.

2 Dataset Construction

We construct our dataset through a multi-stage pipeline that progressively strengthens problem specifications and verification signals, as illustrated in Figure 3. Starting from curated seed problems, we refine task formulations and iteratively generate and filter test cases by leveraging execution feedback from diverse candidate solutions. This process produces fine-grained and reliable verification signals suitable for reinforcement learning with verifiable rewards.

2.1 Seed Datasets Construction

To build a large-scale and curated coding dataset, we start by collecting high-quality seed datasets. We aggregate seed data from five publicly available datasets, including TACO (Li et al., 2023), APPS (Hendrycks et al., 2021), SYNTHETIC-1 (Mattern et al., 2025), Codeforces (Penedo et al., 2025), and CodeContests (Li et al., 2022). These datasets provide natural language problem descriptions paired

Table 1: Statistics of seed datasets before and after filtering, and the final dataset EVOLVECODER-22K. Filtering removes highly similar problem instances based on embedding similarity.

Dataset	Before filtering	After filtering	EVOLVECODER-22K
TACO	26,433	13,776	9,099
APPS	8,765	2,295	1,925
PrimeIntellect	16,252	5,021	3,702
Contests	13,610	4,386	3,060
Codeforces	10,024	4,959	3,856
Overall	**75,084**	**30,437**	**21,642**

Figure 1: Semantic duplication across dataset sources.

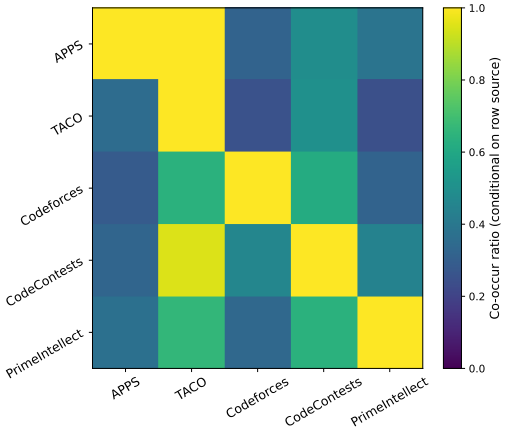
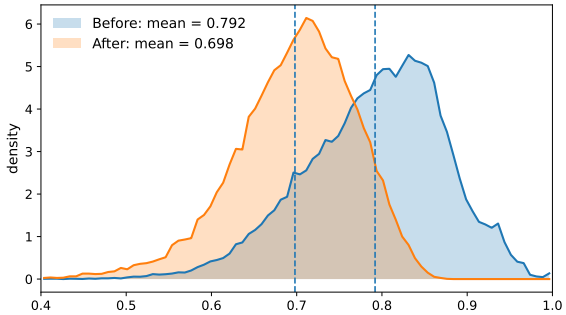


Figure 2: Distribution of mean cosine similarity to the 10 nearest neighbors before and after filtering.



with reference implementations in Python, and cover a wide range of domains and difficulty levels. The resulting collection serves as the foundation for all subsequent stages of our pipeline.

Problems De-duplication Aggregating problems from multiple sources inevitably introduces redundancy due to shared provenance across competitive programming platforms and educational repositories. Figure 1 presents cross-source semantic duplication in terms of relative overlap ratios between datasets, with diagonal entries normalized to one by definition. The most pronounced cross-dataset similarity is observed between APPS and TACO, as well as between TACO and CodeContests, indicating substantial reuse of semantically equivalent or closely related problems across these sources. These patterns suggest that redundancy primarily arises from problem redistribution across benchmarks rather than isolated dataset construction. If left unaddressed, such cross-source duplication can distort the effective data distribution and reduce the true diversity of training signals.

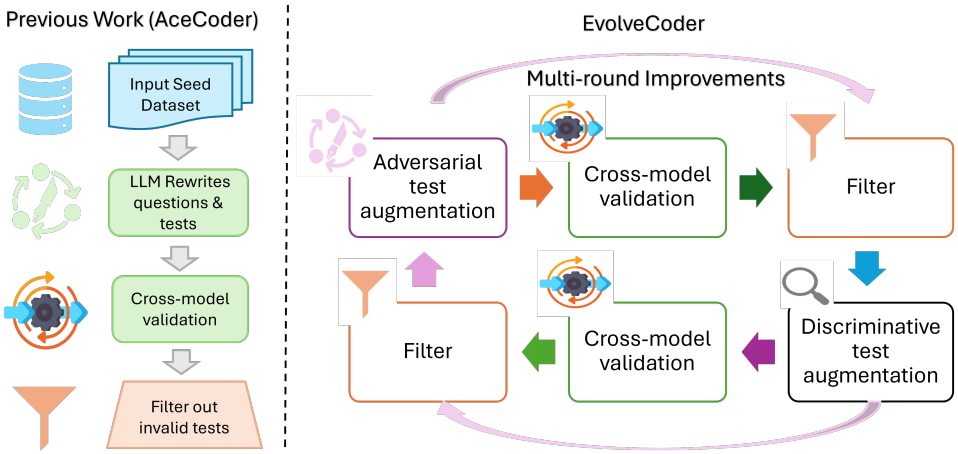
To address this issue, we perform semantic-level deduplication based on dense sentence embeddings. We encode all problem descriptions using the pre-trained ALL-MPNET-BASE-V2 model (Sentence-Transformers) and compute pairwise cosine similarity between embeddings. For pairs with similarity greater than 0.9, we randomly retain a single representative problem and discard the rest. Table 1 summarizes the dataset statistics before and after filtering, showing that a substantial fraction of redundant problems is removed across all sources, with retention rates ranging from 30.89% in PrimeIntellect to 52.12% in TACO. To illustrate the effect of filtering, Figure 2 shows the distribution of mean cosine similarity to each problem's 10 nearest neighbors before and after filtering. After filtering, the distribution shifts markedly toward lower similarity values, with the average similarity decreasing from 0.792 to 0.698. This shift indicates that local semantic neighborhoods become substantially less redundant after filtering. Together, these results show that our filtering procedure improves dataset diversity while preserving large-scale coverage, yielding a cleaner seed set for downstream data generation and reinforcement learning.

2.2 Seed Dataset Refinement

While seed datasets provide programming problems paired with verifiable test cases, many of these problems have become insufficiently challenging for modern reasoning-capable models. Moreover, test cases are typically represented as raw stdin–stdout pairs, where multiple input–output examples are concatenated into a single evaluation instance. This format obscures fine-grained execution outcomes and makes it difficult to assess how a solution behaves on individual test cases. Under these conditions, approaches (He et al., 2025; Liu et al., 2025) that aim to induce harder corner cases often rely on large-scale randomized input generation, with outputs obtained by executing oracle solutions, followed by limited or post-hoc filtering. In practice, this process still results in substantial redundancy, as many test cases probe the same underlying failure modes. In practice, such redundancy significantly increases evaluation cost and training time during reinforcement learning.

To address these limitations, we adopt the seed refinement strategy introduced in AceCoder (Zeng et al., 2025) as an initial preprocessing step. Given a filtered question–solution pair (p, c), where p denotes an original programming problem and c a reference code solution, we prompt GPT-

Figure 3: EVOLVECODER-22K Construction Pipeline



4.1MINI (OpenAI et al., 2024) to generate a refined LeetCode-style problem q with a rewritten problem specification and approximately 20 independently executable test cases $\{t_1, \ldots, t_m\}$. These test cases are explicitly structured and self-contained, enabling fine-grained evaluation of candidate solutions across diverse behavioral regimes. By increasing problem difficulty while reducing redundancy among test cases, this refinement step yields a cleaner and more discriminative seed dataset, providing a stronger foundation for downstream test generation and reinforcement learning. See Appendix A.1 for the refinement prompt.

2.3 Progressive Test Case Refinement

Pipeline Overview. We propose a progressive test case refinement pipeline that iteratively strengthens verification signals by conditioning test generation on diverse candidate solutions and their execution outcomes. Starting from an initial test suite, the pipeline alternates between adversarial test generation and discriminative test generation, progressively exposing unresolved failure modes while controlling redundancy and evaluation cost. After completing all refinement rounds, a final test suite filtering stage is applied to consolidate generated tests and produce a compact, reliable verification set.

Diverse Solution Sampling. To capture a wide range of executable behaviors and failure patterns, we construct a behaviorally diverse pool of candidate solutions. Specifically, we use 8 independently trained open-source reasoning models, including the Qwen3 family (4B, 8B, 14B, 30B-A3B, and 32B) (Yang et al., 2025), MiMo-7B-RL (Xia et al., 2025), Phi-4-Reasoning (Abdin et al., 2025), and DeepSeek-R1-Distill-Qwen-32B (Guo et al., 2025). For each problem q, each model generates 8 solutions, yielding a total of 64 candidates denoted as $\mathcal{S}(q) = \{s_i\}_{i=1}^{64}$. This multi-model sampling exposes diverse reasoning trajectories and systematic failure behaviors that are difficult to obtain from single-model generation and serves as the basis for subsequent test case refinement.

Adversarial Test Case Generation. Building on the behaviorally diverse solution pool, we refine verification signals through adversarial test generation conditioned on candidate programs and their execution behaviors. Given a solution set $\mathcal{S}(q)$ and an initial test suite $\mathcal{T}(q)$, we evaluate all solutions on all tests to construct a binary pass matrix $\mathbf{M}$. Tests that are passed by nearly all solutions are removed, yielding a reduced suite $\widetilde{\mathcal{T}}(q)$ that better captures unresolved behavioral differences.

Based on evaluation outcomes on $\widetilde{\mathcal{T}}(q)$, we form a representative solution subset $\mathcal{S}^\star(q)$ by selecting the two highest-pass-rate solutions together with three solutions exhibiting maximal pairwise disagreement in pass–fail patterns, measured by Hamming distance. This selection balances solution quality and behavioral diversity. Conditioned on the problem description, $\widetilde{\mathcal{T}}(q)$, $\mathcal{S}^\star(q)$, and their fine-grained execution results, the test generation model synthesizes new assert-based tests $\mathcal{T}^+(q)$ that adversarially target uncovered corner cases, producing more informative and discriminative verification signals. See Appendix A.2 for the refinement prompt.

Discriminative Test Case Generation. Complementary to adversarial refinement, we introduce a discriminative test generation strategy targeting candidate programs that remain difficult to distinguish under existing verification signals. Based on the solution–test pass matrix $\mathbf{M}$, we first filter

Table 2: Dataset statistics across iterative construction rounds. For each round, we report the number of problems and the average number of test cases per problem for each subset.

Round	Metric	Overall	TACO	APPS	PrimeIntellect	CodeContests	Codeforces
0	#Problems	18,803	7,919	1,691	3,281	2,679	3,233
	#Avg. Tests	10.76	10.70	10.78	10.49	10.77	11.20
1	#Problems	20,867	8,768	1,850	3,572	2,960	3,717
	#Avg. Tests	20.87	20.43	20.13	19.45	21.17	23.39
2	#Problems	21,412	8,981	1,898	3,676	3,032	3,825
	#Avg. Tests	28.39	27.58	26.70	25.84	29.16	32.96
3	#Problems	21,642	9,099	1,925	3,702	3,060	3,856
	#Avg. Tests	35.04	33.80	32.17	31.43	36.38	41.81

the test suite in two stages. We remove tests with extremely low pass rates across the solution pool, below 0.1, as such tests are likely erroneous and yield unreliable discrimination. We then discard tests with identical pass vectors, retaining a single representative per equivalence class. This produces a compact filtered suite $\widehat{\mathcal{T}}(q)$ that preserves the essential discriminative structure while reducing redundancy.

Using evaluation vectors on $\widehat{\mathcal{T}}(q)$, we select a subset of five behaviorally overlapping solutions $\mathcal{S}^{\sim}(q) \subset \mathcal{S}(q)$ by prioritizing solutions with identical or near-identical pass–fail patterns, and otherwise minimizing pairwise Hamming distance. Conditioned on the problem description, $\widehat{\mathcal{T}}(q)$, $\mathcal{S}^{\sim}(q)$, and their evaluation outcomes, the generator synthesizes new assert-based tests $\mathcal{T}^{\ddagger}(q)$ under an explicit split constraint, requiring each test to be passed by at least one solution and failed by at least one solution. This procedure exposes fine-grained behavioral differences among solutions that were previously indistinguishable. See Appendix A.3 for the refinement prompt.

Test Suite Filtering. This stage refines the test suite $\mathcal{T}(q)$ accumulated over multiple refinement rounds by removing test cases with limited discriminative value across the candidate solution pool $\mathcal{S}(q)$. For each problem q, we analyze the test–solution pass matrix $\mathbf{M}$ defined over $\mathcal{T}(q)$ and $\mathcal{S}(q)$. We first remove tests with extremely low empirical pass rates, below 10%, as such cases are likely to be unreliable. Next, tests with identical pass–fail vectors in $\mathbf{M}$ are grouped, and up to five representatives are retained per group to control redundancy. The resulting reduced test suite $\mathcal{T}'(q)$ is further constrained by discarding problems with fewer than five retained tests or with more than 60 perfectly solved solutions. This filtering procedure yields a smaller but more informative test suite that preserves meaningful behavioral differences while enabling efficient and reliable evaluation across refinement rounds.

3 Dataset Analysis

This section provides an empirical analysis of EVOLVECODER-22K, the dataset obtained after multi-round test case refinement. We analyze its evolution across refinement rounds, focusing on dataset scale, test case retention, and difficulty progression, and evaluate verification quality and model performance on the final dataset.

3.1 Dataset Scale Across Iterations

Table 2 summarizes the evolution of dataset scale across iterative construction rounds. We observe a monotonic increase in both the number of problems and the average number of test cases per problem. Overall, the dataset expands from 18,803 problems in Round 0 to 21,642 problems in Round 3, while the average number of test cases per problem increases from 10.76 to 35.04, reflecting a substantial growth in verification volume over successive iterations.

This growth pattern is consistent across all source subsets, including TACO, APPS, PrimeIntellect, CodeContests, and Codeforces. The reported problem counts correspond to the dataset state after the filtering step described in Section 2.3, which discards problems with fewer than five retained test cases or with an excessive number of perfectly solved solutions. The continued increase in problem count, therefore, indicates that newly generated problems and refined test suites consistently satisfy these quality constraints, allowing the dataset to

Figure 4: Difficulty evolution across rounds.

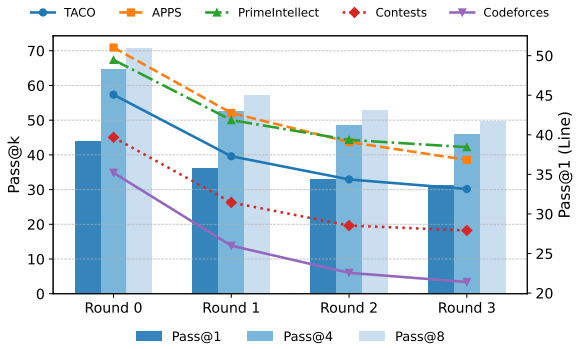
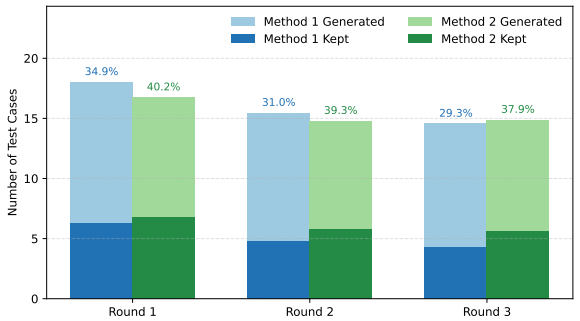


Figure 5: Generated and retained test cases across rounds for two generation strategies: Method 1 (Adversarial) and Method 2 (Discriminative).



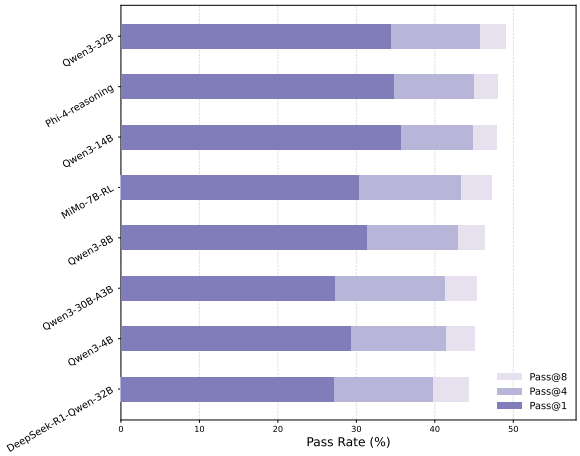
scale in a balanced manner without concentrating expansion on any specific source.

3.2 Difficulty Analysis

We analyze how evaluation difficulty evolves across iterative construction rounds by measuring model Pass@k under the test suite produced at each iteration, where evaluated solutions are uniformly sampled from the full candidate pool $\mathcal{S}(q) = \{s_i\}_{i=1}^{64}$. While the underlying problems remain unchanged, later rounds strengthen verification by introducing increasingly adversarial and fine-grained corner test cases. As shown in Figure 4, Pass@1, Pass@4, and Pass@8 decrease monotonically from Round 0 to Round 3, indicating that progressively refined test suites reject a larger fraction of partially correct solutions and expose failure modes that were previously undetected. Although a decrease in Pass@k alone does not necessarily imply improved verification quality, in our setting it is accompanied by increased discrimination and stable RL gains.

Building on this analysis, we further examine Pass@k performance of different candidate solution models on the final-round test suite, as shown in Figure 6. Clear performance stratification emerges across models under the same, fully refined verification protocol. As expected, Pass@1, Pass@4, and Pass@8 increase with larger k for all models. However, substantial gaps persist in absolute performance: larger-scale or reasoning-enhanced models, such as Qwen3-14B, Qwen3-32B, and Phi-4-Reasoning, achieve consistently higher pass rates across all k, reflecting stronger robustness to adversarial corner cases, whereas smaller and distilled models exhibit lower Pass@1 and Pass@4, revealing more frequent latent failures.

Figure 6: Pass@k performance of diverse candidate solution models on the final dataset.



Notably, the performance differences across models narrow at higher k, particularly for Pass@8, suggesting that the final test suite is not merely overly restrictive but instead effectively differentiates single-solution reliability from multi-sample search capability. This behavior indicates that while weaker models can occasionally recover correct solutions through repeated sampling, stronger models are more likely to produce correct and robust solutions on the first attempt. Overall, these results demonstrate that the final dataset provides a balanced yet discriminative evaluation setting, enabling meaningful comparison of candidate solution models and serving as a reliable benchmark for downstream training and analysis.

3.3 Comparison of Test Case Generation Methods

This section compares two test case generation methods in terms of both the number of generated tests and the proportion retained after filtering across iterative construction rounds. We use pass-vector diversity as a proxy for discriminative power, rather than a direct semantic metric. As

Table 3: Performance across rounds. Pass@1 results for models trained from Qwen3-4B (Thinking) using data from different refinement rounds, compared with strong reference models. We visualize gains of EVOLVECODER-4B (r3) to each baseline in the Δ column. *Abbreviations:* BCB=BigCodeBench-Instruct, LCB=LiveCodeBench.

Model	**EvalPlus**	**BCB-Full**	**BCB-Hard**	**Aider-Polyglot**	**LCB-v5**	**AVG**	Δ
AceCoder-7B	82.7	43.3	19.6	-	-	-	-
DeepSeek-R1-Distill-14B	82.4	38.1	20.9	18.6	53.0	42.6	↑ 15%
DeepCoder-14B	85.3	38.2	18.2	18.4	60.6	44.1	↑ 11%
DeepSeek-V2.5-238B	83.8	48.9	27.0	17.8	42.6	44.0	↑ 11%
Qwen3-4B (Thinking)	85.2	42.0	20.9	21.8	54.2	44.8	↑ 9%
Critique-Coder-4B	86.5	43.1	23.0	24.4	59.0	47.2	↑ 4%
o1	88.6	50.4	28.4	61.7	59.5	57.7	-
EVOLVECODER-4B (r0)	88.3	41.2	23.0	24.4	56.0	46.6	↑ 5%
EVOLVECODER-4B (r1)	88.8	42.4	23.0	24.4	57.2	47.2	↑ 4%
EVOLVECODER-4B (r2)	88.8	42.8	24.3	27.1	59.6	48.5	↑ 1%
EVOLVECODER-4B (r3)	**89.0**	**43.4**	**25.0**	**27.4**	**59.7**	**49.0**	-

shown in Figure 5, both methods exhibit a gradual decrease in the number of generated test cases over iterations, reflecting a convergence toward more targeted and informative test generation as verification is progressively refined.

Despite similar trends in generation volume, the two methods differ in retention behavior. Method 2 consistently achieves higher retention rates, indicating that its generated tests are more likely to survive downstream filtering and provide useful discrimination among candidate solutions. In contrast, the other method produces a larger fraction of tests that are removed during filtering, suggesting greater redundancy or lower alignment with unresolved failure modes. Overall, this comparison highlights that the generation strategy has a tangible impact on test suite efficiency, even under the same iterative refinement and filtering protocol.

4 Performance Evaluation

In this section, we evaluate the effectiveness of our dataset through controlled empirical studies. We compare models trained on datasets from different construction rounds under identical training protocols to analyze how progressively refined verification signals influence learning outcomes. We further benchmark our trained models against strong baseline models to assess performance on the final dataset.

4.1 Experimental Settings

Training Setup. We adopt Group Relative Policy Optimization (GRPO) for reinforcement learning. Unless otherwise specified, we use a group size of 8 and apply nucleus sampling with temperature 0.6 and top-p 0.95 during rollout generation. All experiments are conducted using the Qwen3-4B model, with a maximum prompt length of 4,096 tokens and a maximum response length of 32,768 tokens. Policy optimization is performed with a learning rate of 1×10^{-6}. We set the upper and lower clipping ratios of the actor to 0.30 and 0.20, respectively, to stabilize updates. All models are trained under the same optimization settings to ensure fair comparison.

Benchmarks. We evaluate our trained models on four widely used coding benchmarks: EvalPlus (Liu et al., 2023b), which aggregates HumanEval (Chen et al., 2021), HumanEval+, MBPP (Austin et al., 2021), and MBPP+, BigCodeBench-Instruct (Zhuo et al., 2024), Aider-Polyglot (Aider, 2024), and LiveCodeBench v5 (2024.10–2025.02) (Jain et al., 2024). These benchmarks cover a diverse range of programming tasks and difficulty levels, enabling a comprehensive assessment of code generation and reasoning performance across different evaluation regimes.

Evaluation Setup. For evaluation, we follow the thinking-mode sampling configuration reported in the original Qwen3 paper (Yang et al., 2025), using a temperature of 0.6, top-p 0.95, top-k 20, and a maximum generation length of 32,768 tokens, which is applied consistently across all benchmarks. For LiveCodeBench, we adopt the official thinking-mode evaluation prompt. We additionally compare our models against several strong coding baselines, including DeepSeek-R1-Distill-14B (Guo et al., 2025), DeepCoder (Luo et al., 2025), DeepSeek-V2.5 (DeepSeek-AI, 2024), and GPT-o1 (Jaech et al., 2024b), evaluated under high-reasoning settings where applicable.

4.2 Results

Table 3 summarizes model performance across iterative reinforcement learning rounds under the same training and evaluation settings. We observe steady and consistent improvements as training progresses from Round 0 to Round 3 across all benchmarks. In particular, gains are most pronounced on BigCodeBench-I, Aider-Polyglot, and LiveCodeBench, indicating that later rounds lead to stronger generalization on harder and more diverse coding tasks. By Round 3, the model achieves the best overall results, reaching an average score of 49.0, representing a +2.4 improvement over the Round 0 model.

Compared with strong baselines, the final model trained in Round 3 consistently outperforms Qwen3-4B (Thinking) and Critique-Coder-4B across all evaluation metrics, and remains competitive even compared with substantially larger models such as DeepSeek-V2.5-238B. These results demonstrate that iterative reinforcement learning alone, when applied over progressively refined training rounds, yields systematic and reliable performance gains without increasing model scale, improving both overall coding ability and robustness on challenging benchmarks.

5 Related Works

5.1 Unit Tests Generation

Unit testing plays an essential role in verifying the correctness of the generated solution. CodeT (Chen et al., 2022) and MPSC (Huang et al., 2023) made early attempts to leverage LLM to generate both solutions and test cases, selecting the best solution from multiple samples based on execution results on test cases. (Yang et al., 2024) systematically investigated the effectiveness of LLMs in generating unit tests through extensive empirical studies. AceCoder (Zeng et al., 2025) synthesized test cases along with questions from seed datasets in LeetCode style, then utilized Qwen2.5-Coder-32B-Instruct to perform quality control. KodCode (Xu et al., 2025) first used GPT-4o-0513 to generate both solutions and test cases, then employed a self-verification procedure to verify them.

5.2 Synthetic Data Generation

Since manually labeled data is costly to obtain, people have turned to LLM-generated data as an alternative solution. Self-instruct (Wang et al., 2022) leveraged self-generated instruction data to enhance LLMs' instruction-following capabilities. UltraChat (Ding et al., 2023) provided richly structured multi-turn instructional data, which plays a critical role in fostering general chat model capabilities. Dromedary (Sun et al., 2023) was developed by applying Self-align to LLaMA-65B, with alignment data playing a central role in its training. Evol-Instruct (Luo et al., 2023) utilized an evolutionary algorithm that generates diverse and complex instruction data for LLM.

5.3 Reinforcement Learning for Coding Task

Reinforcement learning has shown growing promise in the LLM post-training stage, attracting significant attention for its potential in code generation. CodeRL (Le et al., 2022) introduced the first RL framework in code generation, utilizing unit test signals in an actor-critic architecture. After that, PPOCoder (Shojaee et al., 2023) extends this approach by integrating the PPO algorithm, and RLTF (Liu et al., 2023a) further refines it by providing feedback of multi-granularity. To enhance RL exploration in generating lengthy code, StepCoder (Dou et al., 2024) adopts a step-by-step strategy, breaking down the task into manageable steps. SWE-RL (Wei et al., 2025) extends reinforcement learning to long-context, repository-level software engineering tasks.

6 Conclusions

Existing reinforcement learning with verifiable rewards (RLVR) for code generation is fundamentally limited by weak and static verification signals. In this work, we propose a solution-conditioned adversarial verification paradigm that refines test cases based on model execution behaviors, offering an alternative to fixed or single-pass test generation. This paradigm iteratively increases test difficulty and discriminative power while controlling redundancy and verification cost. Building on this approach, we introduce EVOLVECODER-22K, a coding RL dataset with substantially stronger verification signals. Extensive experiments show that iterative adversarial refinement leads to more stable training and consistent performance gains across diverse coding benchmarks, highlighting the practical value of adversarial verification for effective code reinforcement learning.

Limitations

No Formal Guarantee on Problem and Test Correctness

Despite heuristics such as filtering, cross-model validation, and execution-based consistency checks, our pipeline does not provide a formal guarantee of the correctness of generated problems and test cases. While methods are employed to remove invalid tests and problems, these procedures are empirical in nature. As a result, it remains possible that a small fraction of generated instances contain specification ambiguities, incomplete coverage, or incorrectness that are not exposed by the candidate solution pool. We therefore rely on empirical evidence, such as stable reinforcement learning dynamics and consistent performance improvements across benchmarks, to demonstrate the overall quality of the dataset, rather than offering theoretical guarantees. Establishing formal correctness or soundness guarantees for large-scale, automatically constructed coding datasets remains an open and challenging problem.

Restriction to Python Programs

Our dataset construction and evaluation are limited to Python programs. This choice is motivated by the availability of mature execution tooling, sandboxing infrastructure, and widely adopted benchmarks that enable reliable and large-scale execution-based verification. Python also provides a practical environment for validating reinforcement learning outcomes. While we expect the core principles of our pipeline to be largely language-agnostic, EVOLVECODER-22K itself is not directly applicable to other programming languages. Extending the pipeline to additional languages would require language-specific execution environments, safety mechanisms, and benchmark support, which we leave to future work.

Ethical considerations

This paper fully complies with the ACL Ethics Policy, and we are not aware of any ethical issues associated with this work. AI-based tools were used to assist with language polishing and clarity of presentation. All technical content, experimental design, data generation, analysis, and conclusions were produced by the authors.

References

Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, and ... 2025. Phi-4-reasoning technical report.

Aider. 2024. Aider-polyglot benchmark. https://aider.chat/2024/12/21/polyglot.html#the-polyglot-benchmark.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

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*.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, and 1 others. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.

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, Yu 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*.

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. *arXiv preprint arXiv:2505.24098*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024a. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

OpenAI: Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, and 243 others. 2024b. Openai o1 system card. Technical Report arXiv:2412.16720, OpenAI, San Francisco, CA. ArXiv:2412.16720 [cs.AI].

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

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.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023a. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

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.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. 2025. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*.

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.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Justus Mattern, Sami Jaghouar, Manveer Basra, Jannik Straube, Matthew Di Ferrante, Felix Gabriel, Jack Min Ong, Vincent Weisser, and Johannes Hagemann. 2025. Synthetic-1: Two million collaboratively generated reasoning traces from deepseek-r1.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, and 244 others. 2024. Openai o1 system card. *Preprint*, arXiv:2412.16720.

Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. 2025. Codeforces. https://huggingface.co/datasets/open-r1/codeforces.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Chi Ruan, Dongfu Jiang, Yubo Wang, and Wenhu Chen. 2025. Critique-coder: Enhancing coder models by critique reinforcement learning. *arXiv preprint arXiv:2509.22824*.

Sentence-Transformers. all-mpnet-base-v2. https://huggingface.co/sentence-transformers/all-mpnet-base-v2. Accessed: 2025-06-24.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Alex Su, Haozhe Wang, Weiming Ren, Fangzhen Lin, and Wenhu Chen. 2025. Pixel reasoner: Incentivizing pixel-space reasoning with curiosity-driven reinforcement learning. *ArXiv*, abs/2505.15966.

Zhiqing Sun, Yikang Shen, Qinhong Zhou, Hongxin Zhang, Zhenfang Chen, David Cox, Yiming Yang, and Chuang Gan. 2023. Principle-driven self-alignment of language models from scratch with minimal human supervision. *Advances in Neural Information Processing Systems*, 36:2511–2565.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, and 150 others. 2025. Kimi k2: Open agentic intelligence. *Preprint*, arXiv:2507.20534.

Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

Bingquan Xia, Bowen Shen, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, Liang Zhao, and 1 others. 2025. Mimo: Unlocking the reasoning potential of language model–from pretraining to posttraining. *arXiv preprint arXiv:2505.07608*.

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*.

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*.

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and 1 others. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1607–1619.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *ArXiv*, abs/2503.14476.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *arXiv preprint arXiv:2502.01718*.

Zizheng Zhan, Ken Deng, Jinghui Wang, Xiaojiang Zhang, Huaixi Tang, Minglei Zhang, Zhiyi Lai, Haoyang Huang, Wen Xiang, Kun Wu, Wenhao Zhuang, Shaojie Wang, Shangpeng Yan, Kepeng Lei, Zongxian Feng, Huiming Wang, Zheng Lin, Mengtong Li, Mengfei Xie, and 21 others. 2025. Kat-coder technical report. *Preprint*, arXiv:2510.18779.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, 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 Appendix

A.1 Prompt Template used for Creating Initial Round Dataset

Instruction for LeetCode Question Generator

Task: Transform a code snippet into a very challenging LeetCode-style question intended for advanced CS university students and experienced software engineers.

Role / Prompt: You are the latest and best bot aimed at transforming some code snippet into a very challenging LeetCode-style question intended for advanced CS university students and experienced software engineers.

You will be given:

1. Original question/prompt for writing code
2. Reference program that attempts to answer the question

Primary objectives (what to create):

1. Create a LeetCode-style question that meets the requirements below
2. Generate 20 independent test cases using assert statements

Question requirements (must follow):

1. **Difficulty level:**
   The question must be hard or very hard difficulty level (similar to the hardest LeetCode problems).
   Challenging enough that solving it takes 30–60 minutes for experts.
2. **Algorithmic requirements:**
   The question should require advanced algorithmic thinking, such as:
   → Graph theory with dynamic programming.
   → Advanced string processing (suffix arrays, KMP, etc.).
   → Complex greedy + data structure combinations.
   → Sliding windows with optimization, interval DP, or segment trees.
3. **Format requirements:**
   Must contain a function signature, rather than stdin/stdout style.
   Must be self-contained (no external resources or data).
   Avoid machine learning, OS-level concepts, or anything requiring system calls or file I/O.
   Do **NOT** request time/space complexity analysis or ask for test cases in the question text.
4. **Adaptation policy:**
   You can take inspiration from the reference code snippet, but you may discard parts of it if necessary to make the question cleaner and harder.

Test case requirements:

1. Generate 20 independent test cases using assert statements.
2. Each test case must:
   → Use constant values (no randomness or external resource calls).
   → Be independent of other test cases.
   → Include both input parameters and expected output.

Output format (JSON ONLY): Return a JSON object with the following keys ONLY.

Do not output any text outside JSON.

```
{
  "question": "...",
  "tests": ["assert ...", "assert ..."]
}
```

Inputs:

Original Question: `{instruction}`
Reference Program:

```python
{program}
```

Instruction for LeetCode Question Generator

Task: Transform a code snippet into a very challenging LeetCode-style question intended for advanced CS university students and experienced software engineers.

Role / Prompt: You are the latest and best bot aimed at transforming some code snippet into a very challenging LeetCode-style question intended for advanced CS university students and experienced software engineers. You will be provided with a prompt for writing code.

Primary objectives:

1. Create a LeetCode-style question that meets these requirements:
   The question must be hard or very hard difficulty level (similar to the hardest LeetCode problems).
   The question should require advanced algorithmic thinking, such as:
   – Graph theory with dynamic programming.
   – Advanced string processing (suffix arrays, KMP, etc.).
   – Complex greedy + data structure combinations.
   – Sliding windows with optimization, interval DP, or segment trees.
   The question must:
   – Contain a function signature, rather than stdin/stdout style.
   – Be self-contained (no external resources or data).
   – Be challenging enough that solving it takes 30–60 minutes for experts.
   – Avoid machine learning, OS-level concepts, or anything requiring system calls or file I/O.
   Do **NOT** request time/space complexity analysis or ask for test cases in the question text.
   You can take inspiration from the reference code snippet, but you may discard parts of it if necessary to make the question cleaner and harder.
2. Based on the question you create:
   Generate 20 independent test cases using assert statements.
   Each test case must:

- Use constant values (no randomness or external resource calls).
- Be independent of other test cases.
- Include both input parameters and expected output.

Output format (JSON ONLY): Return a JSON object with the following keys ONLY.

`question`: The LeetCode-style question text (string).
`tests`: Array of assert statements (list of strings).
Do not output any text outside JSON.

```
{
  "question": "...",
  "tests": ["assert ...", "assert ..."]
}
```

Inputs:

Original Question: «instruction»

A.2 Prompt Template used for Adversarial Test Case Generation

Instruction for Adversarial Test Case Generator

Task: Generate adversarial and diverse test cases that reveal subtle weaknesses in high-performing programs. Create 20 new assert-based test cases that significantly upgrade the current test suite.

Role / Prompt: You are an advanced AI system specialized in generating adversarial and diverse test cases that reveal subtle weaknesses in high-performing programs. You will receive a coding problem, five different programs solving it, existing test cases, and their evaluation results.

You will be given:

1. Coding problem description (question)
2. Five different programs solving the problem (Python code)
3. Existing test cases
4. Evaluation results (rows = programs, columns = tests)

Primary objectives (what to achieve): Create 20 new test cases that satisfy these requirements:

1. Focus on challenging high-pass-rate programs
2. Design cases likely to make at least one top-performing solution fail
3. Include diverse input patterns that challenge different dimensions of the problem space
4. Avoid repetition of existing test cases or trivial variations
5. Ensure all test cases are correct according to the problem definition
6. Make test cases independent of other test cases
7. Use constant values (no randomness or external resource calls)
8. Include both input parameters and expected output

Hard constraints (must follow):

1. **Correctness requirement:**
   All test cases must be correct according to the problem definition.
   Do **NOT** create test cases based on any specific program's behavior.
2. **Diversity requirement:**
   Test cases must cover diverse input patterns across different dimensions of the problem space.
   Avoid duplicating existing test cases or creating trivial variations.
3. **Adversarial targeting:**
   Prioritize challenging high-pass-rate programs.
   Design cases likely to expose weaknesses in at least one top-performing solution.
4. **Test independence:**
   Each test case must be independent of other test cases.
   Use constant values only (no randomness or external resource calls).
5. **Format requirement:**
   All test cases must be in assert-based format.
   Include both input parameters and expected output in each test case.

Output format (JSON ONLY): Return a JSON object with the following structure:

The output must be a JSON object with a single key `tests`.
The value must be an array of strings, where each string is an assert statement.
Do not output any text outside JSON.

```
{
  "tests": [
    "assert ...",
    "assert ...",
    ...
  ]
}
```

Inputs:

Question:
`{question}`
Programs (5 Python programs): `{program1}`
`{program2}`
`{program3}`
`{program4}`
`{program5}`
Existing tests:
`{tests}`
Evaluation results (rows = programs, columns = tests):
`{eval_tests}`

A.3 Prompt Template used for Discriminative Test Case Generation

Prompt for Discriminative Test Case Generation

Task: Generate differentiating test cases that expose logical and behavioral differences between similar programs.

Role / Prompt: You are an advanced AI system specialized in generating differentiating test cases that expose logical and behavioral differences between similar pro-

grams. You will receive a coding problem, five programs that currently produce mostly similar evaluation results, existing test cases, and their evaluation results.

Your task: Create 20 new assert-based test cases that maximize discrimination power among these programs.

Requirements:

1. **Differentiation requirement:**
   Each test case must clearly **differentiate among programs** that share similar evaluation results.
   At least one program must fail in each test case.
   At least one program must pass in each test case.
   Test cases should expose different failure modes across the programs.
2. **Correctness and independence:**
   All test cases must be correct according to the problem definition, not based on any specific program.
   Use constant values (no randomness or external resource calls).
   Each test case must be independent of other test cases.
   Include both input parameters and expected output.

Output format (JSON ONLY): Return a JSON object with the following structure:

Do not output any text outside JSON.

```
{
  "tests": [
    "assert ...",
    "assert ..."
  ]
}
```

Inputs:

Question:
`{question}`
Programs (5 Python programs): `{program1}`
`{program2}`
`{program3}`
`{program4}`
`{program5}`
Existing tests:
`{tests}`
Evaluation results (rows = programs, columns = tests):
`{eval_tests}`