HARDTESTS: Synthesizing High-Quality Test Cases for LLM Coding

Anonymous Author(s)
Affiliation
Address
email

Abstract

Verifiers play a crucial role in large language model (LLM) reasoning, needed by post-training techniques such as reinforcement learning. However, reliable verifiers are hard to get for difficult coding problems, because a well-disguised wrong solution may only be detected by carefully human-written edge cases that are difficult to synthesize. To address this issue, we propose HARDTESTGEN, a pipeline for high-quality test synthesis using LLMs. With this pipeline, we curate a comprehensive competitive programming dataset HARDTESTS with 47k problems and synthetic high-quality tests. Compared with existing tests, HARDTESTGEN tests demonstrate precision that is 11.3 percentage points higher and recall that is 17.5 percentage points higher when evaluating LLM-generated code. For harder problems, the improvement in precision can be as large as 40 points. HARDTESTS also proves to be more effective for model training, measured by downstream code generation performance.

1 Introduction

Post-training large language models (LLMs) with outcome verifiers[1] (Guo et al., 2025; Kimi Team et al., 2025) can greatly improve their reasoning ability. LLMs trained with these techniques are approaching the level of the best humans on challenging problems in math and programming olympiads (OpenAI et al., 2025). To properly assign outcome rewards in post-training, reliable verifiers are needed for both reinforcement learning and (self-) distillation.

Verification is a non-trivial process. How good are current verifiers? How to get better verifiers? How much does verifier quality matter in LLM post-training? Verification loops become increasingly less tractable as the notion of correctness increases in complexity. For math, it is relatively straightforward to determine correctness by looking at the answer, whereas verifying programs needs execution. An effective approach to verify programs is through test cases (Le et al., 2022; Singh et al., 2023). However, most datasets of coding problems and associated test cases are less than comprehensive. 60% of the programs that pass test cases in APPS (Hendrycks et al., 2021) are in fact, wrong. 46% of the programs that pass test cases in CodeContests (Li et al., 2022) are semantically correct, but too inefficient to pass human-written tests. More importantly, scraping human-written tests is unfeasible — according to our study, for 80% of the problems, human-written test cases are proprietary and impossible to scrape, demanding synthesized tests. Previous test synthesis attempts, such as TACO (Li et al., 2023), have limited reliability, with the false positive rate being more than *90%* for difficult problems in our experiments.

[1]In this paper, the term "verifier" refers to rule-based systems that attempt to check the correctness of problem solutions. It is used to differentiate from model-based rewards, such as those in RLHF. "Verifiers" are not necessarily formal and do not necessarily guarantee correctness.

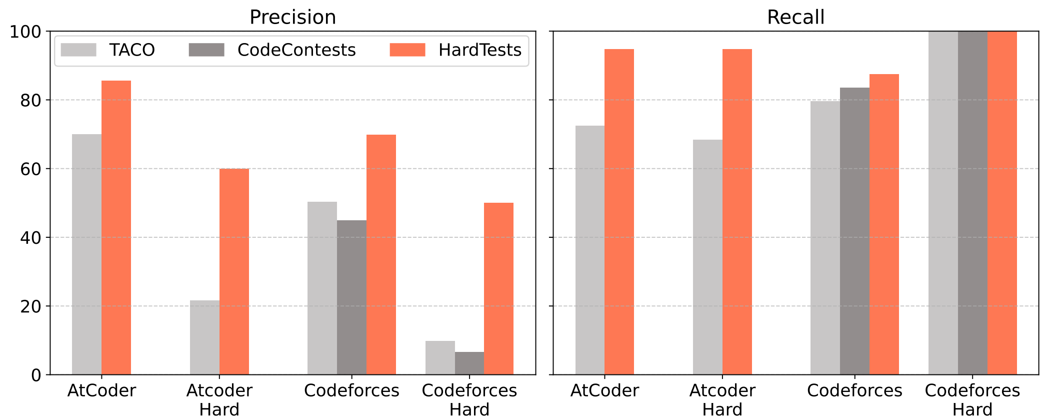

Figure 1: HARDTESTS test cases are significantly better than the baselines. The large improvement in precision indicates that our tests greatly reduce false positives and are indeed *harder*.

The low quality of synthetic tests is due to the challenging nature of coding problems. Coding competitions often require efficient solutions with advanced data structures and algorithms. A bad choice of algorithm can lead to a well-disguised wrong solution, which may easily pass most random tests but still break on human-written special cases. For example, on a random rooted tree with n nodes and depth of d, an algorithm with the time complexity of $\Theta(nd)$ can be very efficient, as $\mathbb{E}[d] = \Theta(\log n)$ for randomly generated trees (Devroye et al., 2012). For such an algorithm to time out, the test case needs to be a valid tree that is large enough (so that n is large) and special enough (so that d is large). A chain (each non-leaf node has exactly one child), whose depth $d = n$ can cause the algorithm to be as slow as $\Theta(n^2)$. We need valid, comprehensive tests that cover edge cases.

Generating valid and comprehensive tests is hard. Existing test synthesis methods, such as CodeT (Chen et al., 2023) and TACO (Li et al., 2023) rely on LLMs to directly write test inputs. While this works when the test inputs are small, it can barely keep the test inputs valid at a larger scale, let alone make them special. To alleviate these issues, we propose HARDTESTGEN, an LLM-based test synthesis pipeline. Our main insights are 1) Test case validity is better preserved when generated from LLM-produced programs rather than directly from the LLMs themselves, and 2) Each test generator has different hypotheses about the programs under test and creates tests from a different distribution. With these insights, HARDTESTGEN prompts an LLM with different aspects to consider for test cases, extracts LLM-generated test generator programs, and filters the test cases using human-written oracle programs, which widely exist for all problems in online coding competitions.

With HARDTESTGEN, we curate HARDTESTS, a comprehensive dataset for coding competitions with 47,136 problems and high-quality test cases. As shown in Figure 1, compared to existing test synthesizers, HARDTESTS tests are more reliable in terms of precision and recall when evaluating programs. The gap in precision can be as large as 40 percentage points for harder problems. Higher reliability of verification makes HARDTESTS the ideal playground for post-training research in the coding domain.

To further demonstrate the benefits of high-quality tests, we conduct post-training experiments with HARDTESTS and baseline tests. Our experiments in 3 different scenarios show that test quality matters significantly for self-distillation and reinforcement learning. Higher-quality tests can lead to improvements in downstream performance. However, our results also indicate that test quality is less important for teacher distillation.

In summary, this work provides:

- HARDTESTGEN, an LLM-based test synthesis pipeline that generates high-quality test cases for coding problems, improving precision by 11.3 points and recall by 17.5 points on average.
- HARDTESTS, a comprehensive problem set for competition-level code generation, with 47,136 problems, among which 32.5k have high-quality test cases generated by HARDTEST-GEN.
- Empirical analyses on how test quality affects LLM post-training. We show that test quality is of great importance for reinforcement learning and self-distillation.

2 Related work

RLVR. Reinforcement learning has shown great potential in improving LLM reasoning abilities in various domains, such as math (Guo et al., 2025; Zeng et al., 2025b; Ren et al., 2025) and coding (OpenAI, 2025; Liu & Zhang, 2025; Luo et al.). The resulting long-reasoning LLMs, such as OpenAI-o3 (OpenAI, 2024) and DeepSeek-R1 (Guo et al., 2025), largely outperform short-reasoning LLMs through simple RL training to improve outcome-based reward, i.e., whether the model-generated code solution passes all test cases. Although some previous works have explored heuristic rules for selecting training data to improve RL performance (Ye et al., 2025; Wang et al., 2025b; Li et al., 2025) or reward design (Hou et al., 2025; Kimi Team et al., 2025; Costello et al., 2025), the impact of test case quality on coding LLMs during RL training remains underexplored. In this work, we show that high-quality test cases, those better at detecting subtle bugs in code, can largely improve coding LLM performance after RL training.

LLM-based test synthesis. Test cases are crucial in evaluating the functional correctness and performance of LLM-generated code. Benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021) provide hand-written test cases that serve as a proxy for code correctness. However, such human-authored test cases are often only publicly available for a limited set of problems. Early approaches such as EvoSuite (Fraser & Arcuri, 2011) and Pynguin (Lukasczyk & Fraser, 2022) employ search-based heuristics methods. More recently, CodeContests (Li et al., 2022) generates additional test cases by mutating existing crawled inputs. Several efforts leverage LLMs to synthesize test inputs and (pseudo)-oracle programs for test outputs. CodeT (Chen et al., 2023) and ALGO (Zhang et al., 2023) rely on LLMs to generate both tests and reference programs for existing coding problems. EvalPlus (Liu et al., 2023) extends HumanEval with more tests by providing the reference implementation to LLMs to synthesize seed input.Similarly, TACO (Li et al., 2023) also generates test inputs with LLMs and outputs with reference implementation. STGen (Peng et al., 2025) generates stressful test cases for evaluating the time efficiency of code. KodCode (Xu et al., 2025) and AceCoder (Zeng et al., 2025a) push synthetic data even more by generating coding questions, reference solutions, and tests all with LLMs. Although existing LLM test synthesis methods prove to be useful in many scenarios, their quality is far from perfect. We present a more thorough discussion on the quality issues in LLM synthetic tests and their implications in Appendix A.1. Concurrently with our work, rStar-Coder (Liu et al., 2025) and HF-Codeforces (Penedo et al., 2025) also study more reliable test synthesis in the competition context. Comparing to them, our work highlights a thorough analysis of test quality and a unique set of post-training experiments that demonstrate the downstream effects of high-quality tests.

Datasets for competition code generation. Existing datasets for competition code generation focus on scaling the number of problems and CoTs. Luo et al. filters a high-quality 24k problemset of TACO, LiveCodeBench, and other contest programming problems. CodeForces-CoTs, the dataset of 10k Codeforces problems created by Penedo et al. (2025), contains 100k reasoning traces and solutions generated by DeepSeek R1. OpenCodeReasoning (Ahmad et al., 2025) also compiles a dataset of 28k problems, generates 735k reasoning traces, and filters them for syntactic correctness. While these efforts have shown that better models can be trained with more data and more trajectories from teacher models, they are facing a "code verifiability crisis", as described by Open-R1 (Face, 2025), and programs that pass test cases in these problem sets are not necessarily correct. In our paper, we curate HARDTESTS, the competitive coding problem set with the most number of problems (47k). More importantly, we push the scaling of training data towards higher quality of test cases and evaluate how test quality affects model training.

3 HARDTESTGEN: Synthesizing High-Quality Test Cases

3.1 Problem Setting

Coding problems. We study test generation for coding problems with natural language specifications. We denote the space of problem specifications as $\mathcal{X}$, the space of candidate programs as $\mathcal{Y}$, and the space of test suites as $\mathcal{V}$. A test suite V is a set of test cases $\{t_1, t_2 \cdots, t_{|V|}\}$. A test case is a pair (a, c), where a is an input to a program, and c is a checker for the corresponding output [2]. A candidate

[2] In most cases, the output checker is simply a comparison between golden outputs and program outputs. Others might be equivalence checkers that do not directly compare strings.

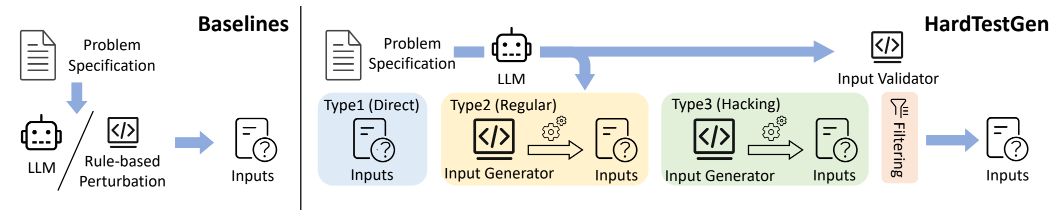

Figure 2: Comparison of the input generation process between previous test synthesizers (left) and HARDTESTGEN (right).

program $y \in \mathcal{Y}$ takes an input and generates an output $y(a)$, which is then sent to the output checker c for a boolean verdict $c(y(a)) \in \{\top, \bot\}$. When y exceeds a pre-defined runtime limit, its verdict is also $\bot$.

Oracle tests and correctness. For every coding problem $x \in \mathcal{X}$, we assume the existence of an oracle test suite $V^* \in \mathcal{V}$, which definitively tells the correctness of a program $y \in Y$, i.e.

$$\text{Correctness}(y, V^*) := \bigwedge_{(a_i, c_i) \in V^*} c_i(y(a_i)). \tag{1}$$

In practice, the oracle tests are usually carefully written and proprietary by problem authors. Only very few of them are available for downloading, which makes them infeasible for model training.

Oracle programs. Compared to rarely available oracle tests, oracle programs (y^* such that $\text{Correctness}(y^*, V^*) = \top$) are available for almost all coding problems in online competitions. Therefore, we assume the existence of oracle programs y^* in our setting.

Test synthesis. Given a problem x, and an oracle program y^*, the task of test synthesis is to create a test suite V that agrees with V^*, i.e., we want $\text{Correctness}(y, V) = \text{Correctness}(y, V^*)$ for as many ys as possible. In HARDTESTGEN, we create a set of inputs $\{a_1, a_2, \cdots, a_{|V|}\}$ and utilize the oracle program to get the outputs, i.e., $c_i = y^*(a_i)$.

3.2 Generating Inputs of Test Cases

We synthesize three types of test inputs. One is directly generated by an LLM, while the other two are generated by LLM-generated programs. Before generating inputs, we first prompt an LLM to generate an input validator in Python that checks whether a given input satisfies all the constraints in the problem specification. We subsequently prompt the LLM to generate the inputs. In the prompt, we include the input validator and an oracle program, as we find that doing so increases the likelihood of synthesizing valid inputs. Figure 2 illustrates the differences between the input generation processes of previous test synthesiziers and HARDTESTGEN.

Type 1. Directly Generated Inputs. We prompt an LLM to directly generate $n_D = 10$ inputs by imitating the sample test cases provided in the problem specification. This type of input is typically small in scale, making it easy to generate and understand, and allowing for quick testing of the candidate program's functional correctness.

Type 2. Regular Inputs. Regular inputs are generated randomly according to the constraints specified in the problem specifications. For most problems, we prompt an LLM to generate a Python function g_R with no parameters that returns a random input on each call. We call this function $n_R = 20$ times to get n_R random inputs. For some problems, there are some unusual categories of outputs that are rarely triggered by random inputs. For example, when a problem's expected output is either "Yes" or "No", the correct output for almost all random inputs might be "Yes". In such cases, random inputs can potentially lead to false positives. For these problems, we prompt an LLM to generate m_R functions, each corresponding to one output category (e.g., "Yes" and "No"). We call each function $n_R = 10$ times to obtain a total of $m_R \times n_R$ inputs with their outputs specified.

Type 3. Hacking Inputs. Some well-disguised false positives cannot be easily detected with random inputs. For example, some programs may be functionally correct but inefficient in worst-case scenarios, or some programs may fail to handle certain edge cases that require special treatment. Therefore, we first prompt an LLM to list several candidate programs for the problem in natural language. Then, we prompt it to generate m_H input generation functions, each attempting to cause one candidate program to fail. Each function is called $n_H = 10$ times, generating $m_H \times n_H$ inputs.

After generating the inputs, we filter out all inputs that fail to pass the examination of the input validator.

3.3 Generating Outputs of Test Cases

We use human-written oracle programs that exist for all online competitions to test outputs. For each problem, we use at most $n_{\text{oracle}} = 8$ oracle programs, prioritizing those from more reliable sources. Each oracle program generates outputs for all synthesized inputs. If the outputs generated by two oracle programs match for more than 90% of the cases, we consider the outputs to be acceptable and adopt the matching portion as the final outputs.

For the majority of problems, a simple string comparison between two outputs is sufficient to determine whether they match. However, some problems require a special judge. For example, a problem might require returning a set (where element order does not matter) or a sequence of operations that achieves a certain effect. In that case, we prompt an LLM to implement a special judge function. This function takes the input and two outputs as parameters, and returns a Boolean value indicating whether the two outputs are equivalent. In our dataset, 25.4% of the problems require a special judge function. In subsequent training and testing processes, this function will continue to be used to determine whether the candidate output and the reference output match.

In our dataset, we use GPT-4o to generate all of the above content. For all functions that need to be generated, we include two to three carefully crafted examples in the prompts. The implementation details of HARDTESTGEN (e.g., prompts), the number of generated test cases, the failure rate and reasons for failure, as well as a concrete example, are provided in Appendix A.2.

3.4 HARDTESTS: 47k Problems with High-Quality Test Cases

The HARDTESTS dataset comprises 47,136 competitive programming problems with high-quality test cases, aggregated from 13 major online judges (OJs) for competitive programming. The dataset is constructed from five direct data sources: Codeforces, AtCoder, Luogu, CodeContests (Li et al., 2022), and TACO (Li et al., 2023). We apply HARDTESTGEN to synthesize test cases for 32.5k problems among them. The detailed constitution and description of the data sources are described in Appendix A.3.

Cleaning, deduplication, and decontamination. For problems with only non-English descriptions, we translated them into English using GPT-4o. To handle overlapping content among the five direct data sources, we filtered out duplicated problems using problem IDs and n-gram overlaps in description, prioritizing versions from the original platforms rather than mirror sites. For correct programs, we retained all available versions and annotated them with their respective sources. We conduct decontamination by removing the problems that are in LiveCodeBench (Jain et al., 2025b) from our dataset. Since most of its problems are from Codeforces and AtCoder, we directly compare the URLs to the problems.

Labelling difficulty. We retained the difficulty labels assigned by all five data sources in our dataset. In the experiments presented in Section 4, we used the difficulty labels from Luogu, as it provides consistent and fine-grained labels for problems from both AtCoder and Codeforces. Luogu's difficulty labels are divided into seven levels, with the first level representing beginner-level problems and the seventh level corresponding to problems at the level of national competitions.

4 Direct Evaluation of Test Case Quality

4.1 Evaluation Criteria

We regard the testing of candidate programs as a binary classification process: a program is classified as positive if it passes all test cases, and negative otherwise. To directly assess the quality of test cases, we evaluate how good they are as binary classifiers. Given a problem x, an oracle test suite V^*, a synthesized test suite V, and a set of candidate programs $\{y_1 \cdots y_n\}$, we categorize the programs with their correctness according to V and V^*. When V and V^* both find a candidate program correct, it's a true positive (TP). When V finds a program correct while V^* finds it wrong, it's a false positive (FP). Similarly, we can define true negatives and false negatives. With these categories defined, we use precision and recall to measure test quality:

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}.$$

4.2 Baselines

CodeContests. CodeContests (Li et al., 2022) primarily consists of problems from Codeforces. Codeforces only provides test cases within certain length constraints. CodeContests collects these and refers to them as "private test cases." Additionally, it generates new test cases by introducing random perturbations to the private test cases; these are referred to as "generated test cases." This gives CodeContests an unfair advantage as it has access to the distribution of oracle tests. In our experiments, we only use generated test cases, which reduces the unfairness but does not eliminate it.

TACO. TACO (Li et al., 2023) integrates several existing datasets, such as APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022), while retaining their test cases. In addition to this, TACO generates several additional test cases by using GPT-4o to directly generate the inputs and using oracle programs for outputs. Furthermore, we observed that for some problems from AtCoder and Codeforces, the TACO test cases included official test cases. To ensure fair comparisons, we removed these official test cases.

Ablative Baselines. We also evaluate HARDTESTGEN with only Type 1 or Type 2 inputs to demonstrate the necessity of all 3 types. Notably, the scenario with only Type 1, LLM directly generated inputs, very much resembles many existing test synthesis methods such as KodCoder (Xu et al., 2025), except that they synthesize not only the inputs but also the oracle programs.

4.3 Evaluation Pipeline

To evaluate the accuracy of rewards that our test cases can give to model training, we evaluate the precision and recall over candidate programs generated by LLMs and written by humans on subsets of problems in HARDTESTS. Details about the evaluation protocol can be found in Appendix A.4.

Generating candidate problems. To compare our tests with other synthesizers, we choose the problems that exist in both HARDTESTS and the baseline datasets. For problems from AtCoder, we select 653 problems that exist in both HARDTESTS and TACO. For problems from Codeforces, we select 600 problems that exist in HARDTESTS, CodeContests, and TACO.

Generating candidate programs. We compare our tests with baseline tests on candidate programs generated by 3 LLMs and also by human programmers. Specifically, we use three LLMs: Qwen2.5-Coder-7B-Instruct (Yang et al., 2024), Qwen2.5-Coder-14B-Instruct, and GPT-4o. For each problem, we sample 10 candidate programs from each LLM using a temperature of 0.7 and a top-p of 0.95. We also randomly select 10 real-world human submissions for each problem.

Generating gold labels. We need gold labels to compute precision and recall. For AtCoder, we run candidate programs on official tests that have been previously made available. For Codeforces, we submit candidate programs to the website to obtain ground-truth verdicts. The human-written candidate programs are sampled from `MatrixStudio/Codeforces-Python-Submissions`, which provides official verdicts. We then use synthetic test cases to classify the correctness of these programs and compare the results against the ground-truth labels, thereby evaluating test case quality.

4.4 Results

We evaluate the correctness of programs written by three LLMs and human programmers for problems from AtCoder and Codeforces using test cases from TACO, CodeContests, and HARDTESTS. The results are in Table 1 and 2. We present qualitative analyses of the synthetic tests in Appendix A.5.

We find that HARDTESTS significantly outperforms TACO and CodeContests in terms of both precision and recall under most evaluation settings. Moreover, this advantage becomes more pronounced as problem difficulty increases. For example, for the Qwen2.5-Coder-7B-Instruct model on AtCoder problems with difficulty level 4+, TACO achieves a precision of 21.67 and a recall of 68.42, whereas HARDTESTS achieves a precision of 60.00 and a recall of 94.74. This implies that using HARDTESTS during RL training would yield more true positive rewards and much fewer false positive rewards.

Furthermore, we observe that as the source of programs becomes less "intelligent" (ranging from human-written to 7B LLM-generated), the precision advantage of HARDTESTS becomes more pronounced. We attribute this to the fact that less skilled programmers are more likely to produce functionally correct but inefficient programs. For instance, among incorrect human-written programs, 14.9% are due to TLE (Time Limit Exceeded), whereas among the incorrect programs written by

Table 1: Precision and recall of the test cases of TACO, HARDTESTS, and ablative baseline on AtCoder. HT-TYPE1 refers to the results using only the test cases of Type 1 from HARDTESTS. while TH-TYPE1+2 refers to the results using only the test cases of Type 1 and Type 2 from HARDTESTS.

	difficulty 1		**difficulty 2**		**difficulty 3**		**difficulty 4+**		**average**	
	prec.	**recall**	**prec.**	**recall**	**prec.**	**recall**	**prec.**	**recall**	**prec.**	**recall**
					Qwen2.5-Coder-7B-Instruct					
TACO	**99.48**	77.09	89.66	62.90	69.07	81.71	21.67	68.42	69.97	72.53
HT-TYPE1	94.63	**99.84**	74.70	**100.0**	42.20	**89.02**	10.40	**94.74**	55.48	**95.90**
HT-TYPE1+2	97.85	99.35	97.58	**100.0**	74.23	87.80	58.06	**94.74**	81.93	95.47
HARDTESTS	98.15	98.95	**97.64**	97.58	**86.75**	87.80	**60.00**	**94.74**	**85.64**	94.77
					Qwen2.5-Coder-14B-Instruct					
TACO	**99.82**	78.00	93.24	69.00	80.23	73.40	39.33	72.92	78.16	73.33
HT-TYPE1	96.21	**99.72**	77.22	**100.0**	58.90	**96.81**	18.50	**97.92**	62.71	**98.61**
HT-TYPE1+2	97.31	99.02	94.79	**100.0**	87.50	**96.81**	65.71	95.83	86.33	97.92
HARDTESTS	97.99	99.02	**96.95**	95.50	**93.33**	**96.81**	**67.16**	93.75	**88.86**	96.27
					GPT-4o					
TACO	**100.0**	73.06	99.75	67.29	92.74	74.08	62.07	71.05	88.64	71.37
HT-TYPE1	99.42	**99.47**	94.31	**99.32**	86.39	**99.42**	45.56	**99.67**	81.42	**99.47**
HT-TYPE1+2	99.53	99.18	99.82	97.60	**96.04**	98.45	79.00	99.01	93.60	98.56
HARDTESTS	99.53	99.18	**100.0**	97.43	**96.04**	98.45	**84.18**	98.03	**94.94**	98.27

Table 2: Precision and recall of the test cases of TACO, CodeContests, and HARDTESTS evaluated using LLM-generated programs for problems on Codeforces.

	difficulty 1		**difficulty 2**		**difficulty 3**		**difficulty 4**		**average**	
	prec.	**recall**	**prec.**	**recall**	**prec.**	**recall**	**prec.**	**recall**	**prec.**	**recall**
					Qwen2.5-Coder-7B-Instruct					
TACO	**89.64**	86.13	71.07	92.91	31.06	39.47	9.82	**100.0**	50.40	79.63
CodeContests	85.74	89.24	63.73	97.64	23.80	47.54	6.67	**100.0**	44.99	83.61
HARDTESTS	87.61	**95.45**	**93.30**	**98.82**	**48.38**	**55.61**	**50.00**	**100.0**	**69.82**	**87.47**
					Qwen2.5-Coder-14B-Instruct					
TACO	80.67	87.45	83.88	81.13	53.87	73.88	25.76	**100.0**	61.05	85.62
CodeContests	79.70	95.59	79.29	86.16	46.49	**91.84**	18.68	**100.0**	56.04	**93.40**
HARDTESTS	**83.19**	**98.64**	**88.44**	**100.0**	**67.47**	80.41	**46.58**	90.80	**71.42**	92.46
					GPT-4o					
TACO	**99.58**	80.02	**95.76**	81.72	89.64	74.83	62.64	78.17	86.91	78.69
CodeContests	99.47	94.80	95.25	89.89	86.83	87.08	58.28	**94.31**	84.96	91.52
HARDTESTS	98.80	**98.20**	95.66	**98.71**	**92.73**	**88.50**	**79.82**	**94.31**	**92.00**	**94.93**
					Human Submission					
TACO	**96.28**	88.89	**91.48**	81.59	**75.90**	78.84	62.23	73.77	**81.47**	64.62
CodeContests	94.15	90.06	87.47	89.99	73.11	85.10	56.80	79.88	77.88	69.01
HARDTESTS	93.29	**94.13**	85.15	**95.05**	73.71	**93.59**	**64.16**	**89.35**	79.08	**74.42**

the three LLMs, 30.0% are due to TLE. Consequently, the larger and more diverse test cases in HARDTESTS are more likely to catch inefficient programs than the small-scale test cases in TACO and CodeContests.

Compared with the ablative baselines in Table 1, HARDTESTS that includes Type2 (Regular) and Type3 (Hacking) test cases consistently leads to a precision improvement ranging from 2% to 48%, while the decrease in recall is always within 2.5%. This demonstrates the necessity for having different types of tests.

5 Downstream Effects of Test Case Quality in LLM Post-Training

In this section, we aim to answer two questions with HARDTESTS: when does verifier/test quality matter, and how much does it matter in post-training? We run experiments in 3 different post-training scenarios: *teacher-distillation*, *self-distillation*, and *reinforcement learning*. We examine how much verifier quality affects the training results in code generation, if any.

5.1 Experiment Setup

Teacher-distillation. Various papers, such as DeepSeek-R1 (Guo et al., 2025) suggest that fine-tuning a smaller student model with reasoning trajectories from a stronger reasoning model can greatly improve the student's performance. In this scenario, verifiers can be used to filter out the incorrect trajectories. We sample one reasoning trajectory with a C++ solution program from DeepSeek-R1 for each question in HARDTESTGEN, obtaining 46.6k trajectories in total after deduplication and decontamination against all LiveCodeBench questions. We fine-tune two models from Qwen2.5-Coder-Instruct-7B: one with all 46.6k trajectories, the other with 13k trajectories that are correct according to HARDTESTS. As a baseline, we also evaluate OlympicCoder-7B (Face, 2025), another Qwen2.5-Coder derivation fine-tuned with $\sim$100k trajectories of $\sim$10k Codeforces problems.

Self-distillation. Fine-tuning a model with its own reasoning trajectories can also improve its reasoning ability (Zelikman et al., 2022). Hence, determining which trajectories to use is a critical issue. To examine the effects of test quality, we sampled 5 traces of Qwen3-4B and used the tests generated by HARDTESTGEN for filtering. We selected 4989 questions where there is at least one Qwen3-4B generated program that passes the tests and at least one that fails the tests. We create 3 datasets for self-fine-tuning, each containing one trajectory per question. The *bad 5k* randomly samples one incorrect trajectory for each question. The *good 5k* randomly samples one correct trajectory. The *random 5k* randomly samples one trajectory, regardless of its correctness, for each question. We further fine-tune Qwen3-4B with these 3 datasets and compare the performance of the resulting models. All our fine-tuning experiments were done with Llama-factory (Zheng et al., 2024).

Reinforcement learning. Verifier feedback is an option for distillation, but it is a must for reinforcement learning. To investigate how verifier quality affects RL, we train Qwen3-4B with RL using the same problem set, the identical training setup, and different test cases. We select a problem set with $\sim$5k problems that exist in both HARDTESTS and TACO for training. We use a modified version of veRL (Sheng et al., 2024) inspired by Code-R1 (Liu & Zhang, 2025) for training with GRPO (Shao et al., 2024). When a program passes all tests, it gets a reward of 1, otherwise, it gets a reward of 0. We compare different verifiers by looking at the final performance and the validation curve.

Evaluation protocol. We use LiveCodeBench (Jain et al., 2025b) version 5 to evaluate the model performance. Since all the programs we use for tuning are in C++, we build an evaluation pipeline for evaluating C++ programs for LiveCodeBench and select a 105-problem subset where all problems have test cases of "stdin" type. We name this subset of problems we use "LiveCodeBench-105". Details about our training and evaluation procedure can be found in Appendix A.6, including the problems and hyperparameters we use for training and the sampling parameters we use for evaluation.

5.2 Results

Teacher-distillation benefits more from question scaling than test quality or sample scaling. We evaluate models fine-tuned from Qwen2.5-Coder-7B using different training sets on LiveCodeBench-105 and report the results in Table 3. Note that the difficulty labels are obtained from LiveCodeBench. The model trained with HARDTESTS with all 46.6k examples outperforms OlympicCoder-7B (trained with 100k trajectories of 10k questions), suggesting that the quality and diversity of training questions matter more than the number of training samples. Interestingly, the model trained on smaller but more curated subsets (13k filtered trajectories) does not match the performance of using larger, unfiltered data, suggesting that data scaling dominates trajectory correctness in the teacher-distillation setting. This observation aligns with the concurrent findings from OpenCodeReasoning (Ahmad et al., 2025).

Self-distillation performance is highly dependent on sample quality and needs a good verifier. We evaluated variants of Qwen3-4B models self-distilled with different 5k subsets on LiveCodeBench-105 and present the results in Table 4. Model self-distilled from incorrect samples identified by HARDTESTGEN's tests drops more significantly in pass@k. Self-distillation with randomly selected

Table 3: pass@k (%) of teacher-distilled LLMs based on Qwen2.5-Coder-7B on LiveCodeBench-105.

	Easy pass@1	Medium pass@1	Hard pass@1	All pass@1	All pass@10
QC2.5-7B-Ins	58.75	9.58	2.46	16.95	27.62
OlympicCoder-7B (100k trajectories)	65.83	41.25	2.46	25.81	46.67
QC2.5-7B-Ins + HARDTESTS (13k, filtered)	77.08	29.17	1.75	25.24	39.05
QC2.5-7B-Ins + HARDTESTS (46.6k, full)	**83.65**	**44.58**	**6.49**	**32.86**	**53.33**

Table 4: pass@k (%) self-distilled LLMs based on Qwen3-4B on LiveCodeBench-105.

	Easy pass@1	Medium pass@1	Hard pass@1	All pass@1	All pass@5	All pass@10
Qwen3-4B	88.75	53.33	11.05	**38.48**	52.04	56.19
Qwen3-4B (with *bad 5k*)	84.17	45.42	8.07	34.00	48.42	54.92
Qwen3-4B (with *random 5k*)	84.58	36.25	9.12	32.75	50.85	57.14
Qwen3-4B (with *good 5k*)	85.42	47.08	10.53	36.00	**55.15**	**60.00**

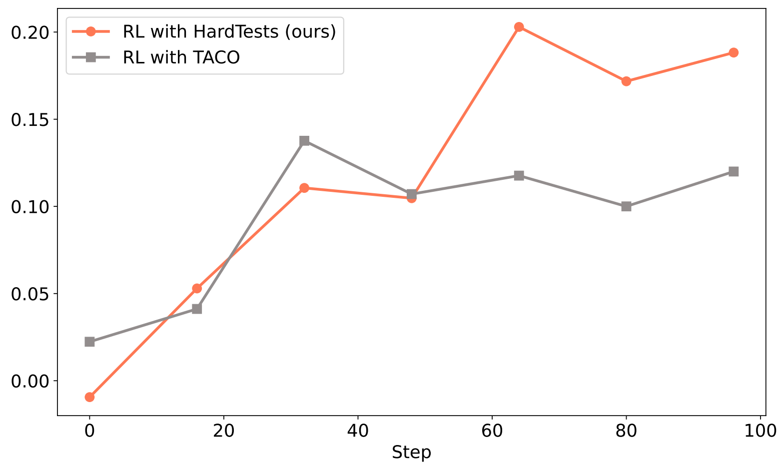

Figure 3: RL Validation Rewards Over Time. Reward from HARDTESTS makes the training better.

data could harm pass@1 even more, despite the slight improvements in pass@10. In contrast, using a 5k subset verified by HARDTESTGEN's test cases results in a smaller drop in pass@1 and a notable gain in pass@5 and pass@10, suggesting that verifiers are important to self-distillation.

Test quality matters significantly for reinforcement learning. As shown in Figure 3, the validation reward curve for HARDTESTS during RL training is generally higher than that for TACO. This indicates that for the same problems, HARDTESTS is giving better rewards. To evaluate on LiveCodeBench-105, we run the best checkpoints (according to valid reward) of both training jobs within 100 steps. As reported in Table 5, TACO tests hurt the model's overall performance, while HARDTESTS improves the model's overall performance.

Table 5: pass@k (%) for LLMs RL-trained from Qwen3-4B on LiveCodeBench-105.

	pass@1	pass@5	pass@10
Qwen3-4B	38.48	52.04	56.19
Qwen3-4B (RL with TACO)	36.95	51.01	57.14
Qwen3-4B (RL with HARDTESTS)	**39.42**	**57.89**	**64.76**

6 Conclusion

We present HARDTESTGEN, an LLM-based test synthesis pipeline, which is used to create HARDTESTS, a competitive coding dataset with 47k problems and significantly higher-quality tests. We examine when and how much test quality matters in LLM post-training, showing that harder tests generated by HARDTESTGEN can indeed help LLM post-training in many scenarios.

Limitation

Although HARDTESTS has higher-quality tests than the baselines, they are still not as good as human-written ones. Moreover, we assume the existence of oracle solutions to utilize HARDTESTGEN, which may not be true for some coding domains. To address this issue, we briefly discuss an initial idea for synthesizing tests without oracles in Appendix A.7. Another limitation of the HARDTESTGEN is that the code being tested is constrained to a single file that uses Standard I/O for input and output. However, many real-world coding problems are more complicated, *e.g.* coding problems in SWE-bench that may involve file I/O or web I/O, and we leave the exploration of applying HARDTESTGEN to these scenarios as future work.

References

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data distillation for competitive coding, 2025. URL `https://arxiv.org/abs/2504.01943`.

Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. Tdd-bench verified: Can llms generate tests for issues before they get resolved?, 2024. URL `https://arxiv.org/abs/2412.02883`.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL `https://arxiv.org/abs/2108.07732`.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *ICLR*, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL `https://arxiv.org/abs/2107.03374`.

Caia Costello, Simon Guo, Anna Goldie, and Azalia Mirhoseini. Think, prune, train, improve: Scaling reasoning without scaling models. *arXiv preprint arXiv: 2504.18116*, 2025.

Luc Devroye, Omar Fawzi, and Nicolas Fraiman. Depth properties of scaled attachment random recursive trees. *Random Structures & Algorithms*, 41(1):66–98, 2012.

Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL `https://github.com/huggingface/open-r1`.

Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pp. 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025179. URL `https://doi.org/10.1145/2025113.2025179`.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Bairu Hou, Yang Zhang, Jiabao Ji, Yujian Liu, Kaizhi Qian, Jacob Andreas, and Shiyu Chang. Thinkprune: Pruning long chain-of-thought of llms via reinforcement learning. *arXiv preprint arXiv: 2504.01296*, 2025.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark, 2025a. URL `https://arxiv.org/abs/2410.00752`.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2E: Turning any github repository into a programming agent environment. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 21196–21224. PMLR, 21–27 Jul 2024. URL `https://proceedings.mlr.press/v235/jain24c.html`.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL `https://openreview.net/forum?id=chfJJYC3iL`.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, and Zonghan Yang. Kimi k1.5: Scaling reinforcement learning with llms, 2025. URL `https://arxiv.org/abs/2501.12599`.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

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

Xuefeng Li, Haoyang Zou, and Pengfei Liu. Limr: Less is more for rl scaling. *arXiv preprint arXiv: 2502.11886*, 2025.

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, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Yanchi liu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, and Wei Cheng. Scattered forest search: Smarter code space exploration with llms, 2025. URL `https://arxiv.org/abs/2411.05010`.

Jiawei Liu and Lingming Zhang. Code-r1: Reproducing r1 for code with reliable rewards. 2025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023. URL `https://arxiv.org/abs/2305.01210`.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset, 2025. URL https://arxiv.org/abs/2505.21297.

Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22. ACM, May 2022. doi: 10.1145/3510454.3516829. URL http://dx.doi.org/10.1145/3510454.3516829.

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. Deepcoder: A fully open-source 14b coder at o3-mini level.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents, 2025. URL https://arxiv.org/abs/2406.12952.

OpenAI. Openai o1 system card. *arXiv preprint arXiv: 2412.16720*, 2024.

OpenAI. Competitive programming with large reasoning models. *arXiv preprint arXiv: 2502.06807*, 2025.

OpenAI, :, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, o3 contributors, Oleg Mürk, Rhythm Garg, Rui Shu, Szymon Sidor, Vineet Kosaraju, and Wenda Zhou. Competitive programming with large reasoning models, 2025. URL https://arxiv.org/abs/2502.06807.

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

Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. Coffe: A code efficiency benchmark for code generation, 2025. URL https://arxiv.org/abs/2502.02827.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv: 2504.21801*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation, 2025a. URL https://arxiv.org/abs/2406.04531.

Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Lucas Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, et al. Reinforcement learning for reasoning in large language models with one training example. *arXiv preprint arXiv:2504.20571*, 2025b.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*, 2024.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding, 2025. URL https://arxiv.org/abs/2503.02951.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. Limo: Less is more for reasoning. *arXiv preprint arXiv: 2502.03387*, 2025.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation, 2024. URL https://arxiv.org/abs/2305.04207.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis, 2025a. URL https://arxiv.org/abs/2502.01718.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. *arXiv preprint arXiv: 2503.18892*, 2025b.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. Algo: Synthesizing algorithmic programs with llm-generated oracle verifiers, 2023. URL https://arxiv.org/abs/2305.14591.

Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. Testbench: Evaluating class-level test case generation capability of large language models, 2024. URL https://arxiv.org/abs/2409.17561.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Our first two claims about test quality is supported by Section 3 and 4's method and experiments. Our third claim about the downstream effects of test quality is supported by Section 5.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: We discussed the limitations of HARDTESTGEN and HARDTESTS in the limitation section in the appendix.

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The overview of experiments and protocol is listed in our section 3 and 4. The details including dataset curation process, hyperparameters for training and sampling parameters for inference are described in the Appendix.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [NA]

Justification: We plan to release the full dataset with all 47k problems and all the code and model checkpoints upon publication.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.

- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: They are listed in the experiment setup sections and more details are in the appendix.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: It is conventional for people not to report error bars as the computation cost for sampling enough samples to obtain statistic significance for each problem is very high.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: These are provided in the appendix.

   Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: See our conclusion.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

Justification: No such risks.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We carefully credit and cite them in the appendix about daca curation.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: In section 3 and the appendix.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: **[TODO]**

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

A Appendix

A.1 More Related Work on Synthetic Test Quality and its Implications

Although existing LLM test synthesis methods prove to be useful in many scenarios, such as improving the quality of synthetic data (Wei et al., 2024) and software engineering(Mündler et al., 2025; Jain et al., 2024), their quality is far from perfect (Yuan et al., 2024) and are bounded in complexity, because direct generations of complicated data structures often result in inconsistency (Zhang et al., 2023). Weak verifiers can harm downstream code generation and search performance (Light et al., 2025). The quality of those synthetic tests and their implications are less discussed. Existing benchmarks for LLM test case generation abilities focus on code coverage and/or mutation scores (Wang et al., 2025a; Zhang et al., 2024; Jain et al., 2025a, 2024), the success rate for reproducing issues (Mündler et al., 2025), and the code change coverage for generated code patches (Ahmed et al., 2024; Mündler et al., 2025).

A.2 Details of the Test Cases Generation Pipeline HARDTESTGEN

As we mentioned in Section 3.2, HARDTESTGEN constructs both the input generator functions and the validator functions for verifying input correctness. In this section, we first introduce the detailed HARDTESTGEN implementation, including the coding problem filtering process, and detailed prompts for input generator/validator synthesis (Section A.2.1), followed by detailed dataset statistics for the final HARDTESTS dataset (Section A.2.2) and some examples in HARDTESTS (Section A.2.3).

A.2.1 HARDTESTGEN Implementation

Coding problem filtering. Before generating test cases, we first filter out questions not suitable for our test case generation. For example, those without oracle code solutions, and the questions that do not use standard I/O for input and output. More specifically, our question filtering process is as follows: We first remove problems that do not have any oracle programs. Next, we exclude all problems where the `starter_code` field is non-empty, as they are so-called "core logic" problems, rather than "input-output" style problems, and typically originate from online judges like LeetCode and GeeksforGeeks. In such problems, the programmer is not responsible for handling input and output logic, but only for implementing the core function based on a given function signature. Since the inputs and outputs in these problems are often not strings, they are difficult to use for test case generation. After the filtering, we are left with 32.5k unique coding problems.

Input validator prompt. We use the following LLM prompt to generate an input validator function, and a special judge function when necessary. This prompt includes the problem specification and the oracle program to help the LLM have a better understanding.

Input generator prompt. We use the following prompt to have the LLM generate inputs directly (Type 1), a regular input generator (Type 2), and a hacking input generator (Type 3). This prompt makes use of the problem specification, oracle program, and input validator to help the LLM better understand the problem requirements.

Note that in the prompts above, we provide two to three carefully crafted examples for each function that we ask the LLM to generate, enabling in-context learning. Additionally, we prompt the LLM to perform chain-of-thought reasoning. These two requirements help the LLM understand the task better and improve the data synthesis.

A.2.2 HARDTESTS Statistics

We generated test cases for all 32.5k valid questions in the HARDTESTS. The status distribution of test case generation is shown in Figure 5. While we carefully designed the test-case generation prompt, we didn't attain 100% coverage. We successfully generated test cases for 81.9% of the questions. The main failure reasons include: no valid oracle programs (i.e., compiles and runs without errors) (6.62%), all output verification failed (5.85%), and input generation failed (3.72%). The distribution of the number of Type1, Type2, and Type3 test cases, as well as the total number of test cases, is shown in Figure 4.

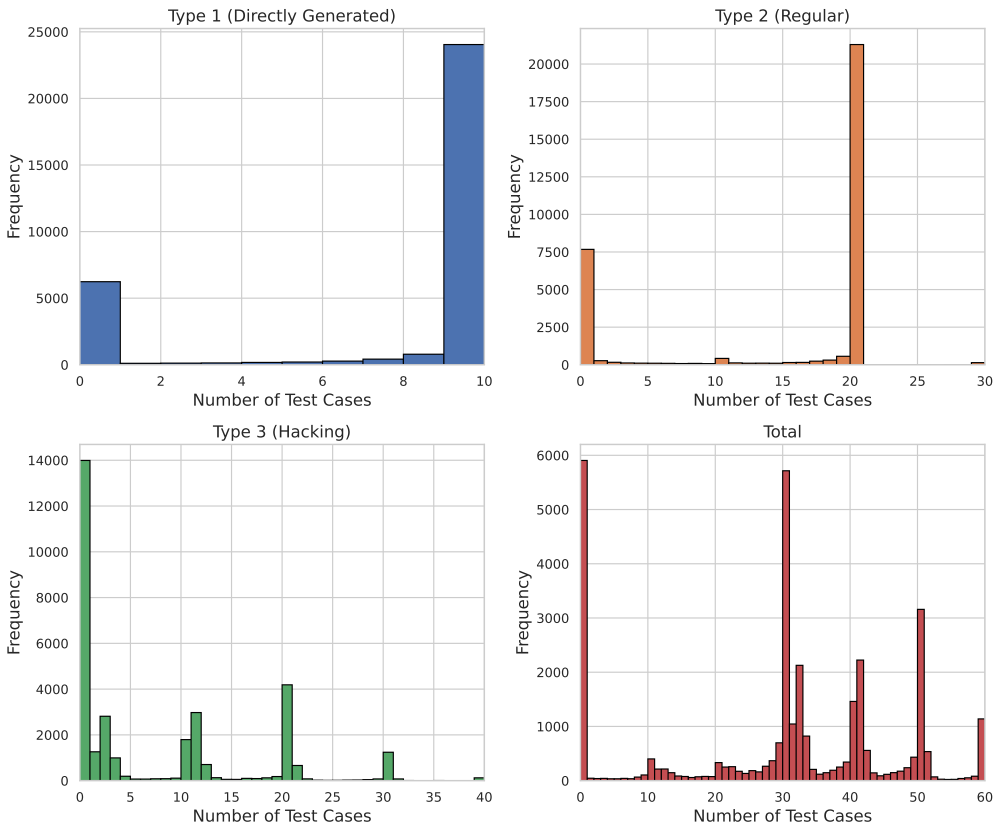

Figure 4: The distribution of the number of Type1, Type2, and Type3 test cases, as well as the total number of test cases in HARDTESTS.

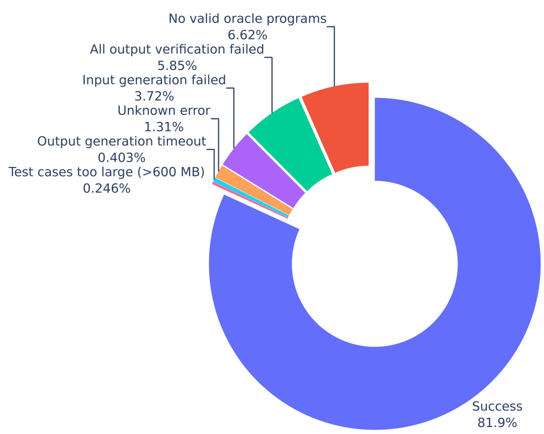

Figure 5: The result status distribution of our test case generation pipeline HARDTESTGEN.

A.2.3 HARDTESTS Examples

Example 1

This example demonstrates the input validator, Type 1 (Directly Generated) and Type 2 (Regular) test cases, as well as a custom judging function. Here's the problem description:

Codeforces 1096A: There are a total of T ($1 \leq T \leq 1000$) sub-tasks. Each sub-task gives a pair of integers l, r ($1 \leq l \leq r \leq 998244353$), and the goal is to find a pair of integers x, y such that $l \leq x, y \leq r$, $x \neq y$, and y is divisible by x. It is guaranteed that every sub-task has a valid solution.

Note: It can be mathematically proven that a sub-task has a solution if and only if $2l < r$.

The input validator is as follows. It checks whether `input_str` conforms to the required format specified in the problem specification, whether all data falls within the required ranges, and whether other constraints are satisfied (e.g., whether each sub-task has a solution).

Since this problem allows multiple correct solutions, simple string comparison is not sufficient. We need a special, customized output judging function. The output judging function is as follows.

The Type1 (Directly Generated) inputs are as follows.

The Type 2 input (Regular) generator is as follows. To ensure a solution always exists, the LLM sets $r \geq 2l$.

The LLM believes that there is no need to generate a Type 3 (Hacking) input generator for this problem.

Example 2

This example demonstrates the input validator, as well as the Type 1 (Directly Generated), Type 2 (Regular), and Type 3 (Hacking) test cases. Here's the problem description:

Codeforces 1141 A: Given two numbers n, m ($1 \leq n \leq m \leq 5 \times 10^8$), you are to determine whether it is possible to transform n into m by multiplying by 2 and 3, and if so, output the minimum number of operations. Otherwise, output -1.

The input validator is as follows. It checks whether `input_str` conforms to various format requirements and constraints.

The Type1 (Directly Generated) inputs are as follows.

The Type 2 input (Regular) generator is as follows. The output of this problem has two categories (i.e., possible and impossible), so the LLM generates two regular input generating functions, corresponding to these two categories respectively.

The Type 3 input (Hacking) generator is as follows. The LLM generates two hacking input generating functions. The first function sets a small n and a large m. This is because a brute-force approach that a candidate program might take is to use DFS, recursively trying to multiply n by 2 or 3 until it becomes greater than or equal to m. If we randomly choose n and m, the ratio between them is usually small, so this approach might still pass. Setting n to be small and m to be big creates a large gap between n and m, making the brute-force DFS approach inefficient. The second function sets $m = n$, which serves as an edge case.

For this problem, the LLM believes that a string comparison function would be enough for output judging.

A.3 Details of the Collection of Problem Specifications and Oracle Programs in HardTests

HardTests consists of 47,136 coding problems collected from 13 OJs. In practice, the dataset obtains problem specifications and oracle programs from five direct data sources: AtCoder, Codeforces, Luogu, CodeContests, and TACO.

Data sources. *Codeforces* (`https://codeforces.com/`) is one of the largest English OJs. We collected all publicly available problem specifications up to September 2024 from Codeforces. *AtCoder.* (`https://atcoder.jp/`) is a large OJ offering problems in both Japanese and English. We scraped all problem specifications available up to September 2024, along with three correct user-submitted C++ programs for each problem. We used those directly for problems with official English versions. *Luogu* (`https://www.luogu.com.cn/`) is a large Chinese OJ consisting of a main section (Luogu-Main) and four mirror sections. The main section hosts original problems authored by users and administrators, as well as problems sourced from real-world contests (e.g. USACO). The mirror sections contain problems from other OJs, including AtCoder, SPOJ, Codeforces, and UVa. We collected all available problem specifications and community-authored tutorials, which often include both correct C++ programs and corresponding natural language explanations, from Luogu. *CodeContests* (Li et al., 2022) is a dataset comprising 13,493 problems collected from five OJs. Each entry includes a problem specification and several correct programs in C++, Python 2, Python 3, and Java. Only Codeforces problems in CodeContests were used in our dataset, as only their problem IDs were explicitly provided. *TACO* (Li et al., 2023) is a large-scale English dataset containing 25.4k problems sourced from ten OJs. Each entry includes a problem specification and multiple correct Python programs. We collect all problems from TACO.

The distribution of problem counts across each OJ is shown in Figure 6. The URLs of each OJ, along with the direct data sources of their problem specifications and oracle programs, are listed in Table 6.

Note that since some problems have multiple oracle program sources, we prioritize programs from more reliable sources when generating test cases. The reliability, supported languages, and notes regarding each direct source of oracle programs are presented in Table 7. The distribution of the number of oracle programs per problem in HARDTESTS is shown in Figure 7.

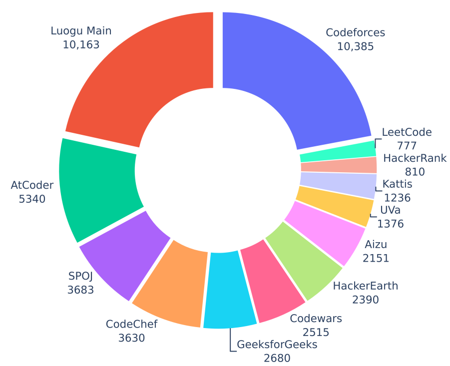

Figure 6: Number of problems from each OJs.

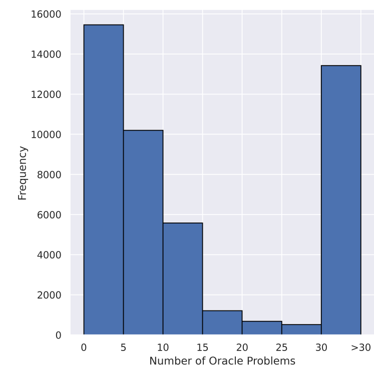

Figure 7: Distribution of the number of oracle programs in HARDTESTS.

Table 6: Problem specification sources and oracle solution sources of each OJ.

OJ	URL	Problem Specification Sources	Oracle Program Sources
Codeforces	`https://codeforces.com/`	Codeforces	TACO, CodeContests, Luogu
AtCoder	`https://atcoder.jp/contests/`	AtCoder	AtCoder, TACO, Luogu
Luogu	`https://www.luogu.com.cn/`	Luogu	Luogu
UVa	`https://onlinejudge.org/`	Luogu	Luogu
SPOJ	`https://www.spoj.com/`	Luogu	Luogu
Aizu	`https://onlinejudge.u-aizu.ac.jp/`	TACO	TACO
GeeksforGeeks	`https://www.geeksforgeeks.org/`	TACO	TACO
Codewars	`https://www.codewars.com/`	TACO	TACO
Kattis	`https://open.kattis.com/`	TACO	TACO
CodeChef	`https://www.codechef.com/`	TACO	TACO
HackerEarth	`https://www.hackerearth.com/`	TACO	TACO
LeetCode	`https://leetcode.com/`	TACO	TACO
HackerRank	`https://www.hackerrank.com/`	TACO	TACO

A.4 Direct Evaluation Details

Evaluation details for LLM-generated programs on AtCoder. AtCoder previously made its official test cases publicly available. Although this is no longer the case, we obtained a partial archive

Table 7: Oracle program sources with reliability, languages, and notes

Oracle Program Source	Reliability	Languages	Notes
User-submitted and accepted programs from AtCoder	High	Python, C++	Some code (either Python or C++) may use AtCoder's custom library.
Code solutions from CodeContests	High	Python 2/3, C++, Java	—
Community-authored editorials from Luogu	Medium	C++	Some editorials may lack complete, directly executable code. But if the code has no compilation or runtime errors, it is very likely to be completely correct.
Verified programs from TACO, i.e., programs that can pass all TACO's own test cases	Medium	Python	There's some false positives in TACO's test cases.
Other programs from TACO	Low	Python	Reliability is not zero due to some false negatives in TACO's test cases.

from the Github repository `conlacda/atcoder-testcases`. On AtCoder, we use the test cases in TACO as the baselines. We selected problems that have at least one test case in each dataset, resulting in a total of 653 problems.

Evaluation details for LLM-generated programs on Codeforces. Codeforces does not make its test cases publicly available. Therefore, we manually submit LLM-generated candidate programs to the Codeforces platform to obtain ground-truth verdicts. We use TACO and CodeContests as baselines. For problems where the results of all three datasets agree, we randomly select 5% of them for submission. For problems where the datasets produce conflicting results, we submit 50% of the candidate programs. We compute precision and recall based on the combined submission outcomes. For each difficulty level from 1 to 4, we randomly select 150 problems with at least one test case in each dataset, yielding a total of 600 problems.

Evaluation details for human-written programs on Codeforces. A dataset at Huggingface titled `MatrixStudio/Codeforces-Python-Submissions` collects 690k human-submitted programs on Codeforces along with their official verdicts. We use the verdicts as the ground-truth labels. All other settings are the same as those of evaluation using LLM-generated programs.

A.5 Qualitative Analysis of Generated Tests

A.5.1 Example 1: False Positive of TACO and HARDTESTS Type 1

In this example we show how TACO and HARDTESTS Type 1 tests cannot break a wrong program and result in a false positive, while HARDTESTS Type 2 tests succeeds in making the program fail. Here's the problem description:

AtCoder ABC117C: Given an integer N ($2 \leq N \leq 2 \times 10^5$) and an integer array A of length N ($0 \leq A_i \leq 10^9$), compute the value of $\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} A_i A_j$ modulo $10^9 + 7$.

Since $2 \leq N \leq 2 \times 10^5$, the solution to the problem needs to be relatively efficient. The correct solution employs mathematical techniques to simplify the equation into: $\frac{1}{2}\left(\left(\sum_{i=1}^{N} A_i\right)^2 - \sum_{i=1}^{N} A_i^2\right)$, which yields an $O(N)$ algorithm.

However, a candidate program generated by Qwen2.5-Coder-7B-Instruct uses a brute-force algorithm with a time complexity of $O(N^2)$. The candidate program is as follows:

Due to its inefficiency, this candidate program failed to pass the official test cases. Nevertheless, because the test cases in TACO and HARDTESTS Type 1 (Directly Generated) were relatively small (with small N), the candidate program successfully passed these cases.

Furthermore, the HardTests Type 2 (Regular) input for this problem is generated using the following Python function:

Due to the larger scale of HardTests Type 2 (Regular) inputs, the candidate program failed to pass these test cases and we have a true negative.

A.5.2 Example 2: False Positive of TACO and HardTests Type 1 + 2

In this example we show how TACO and HardTests Type 1 + 2 tests cannot break a wrong program and result in a false positive, while HardTests Type 3 tests succeeds in making the program fail. Here's the problem description:

AtCoder ABC139C: There are N *(*$1 \leq N \leq 10^5$*) squares arranged from left to right, with the height of the* i*-th square from the left being* H_i *(*$1 \leq H_i \leq 10^9$*). Starting from any square, you can move one step to the right as long as the next square's height is not greater than the current one. Find the maximum number of moves possible.*

Given that $1 \leq N \leq 10^5$, the solution needs to be relatively efficient. The correct solution uses an $O(N)$ greedy algorithm. Specifically, it traverses the array from left to right, counting how many consecutive heights satisfy $H_i \geq H_{i+1}$. Each time a condition fails, the counter is reset, and the maximum value is maintained throughout the traversal as the answer.

A candidate program generated by Qwen-Coder-14B-Instruct uses a brute-force approach, iterating over each starting position and simulating the moves. Although functionally correct, its complexity is $O(N^2)$ and too inefficient to work. The code is as follows:

Because of its inefficiency, this candidate program failed the official test cases. Nevertheless, due to the relatively small scale of the test cases in TACO and HardTests Type 1 (Directly Generated), the candidate program passed these tests.

Additionally, the HardTests Type 2 (Regular) input for this problem is generated using the following Python function:

We observe that since the H_i sequence is randomly generated, it fluctuates significantly, reducing the complexity of the "simulate moving from a certain square" procedure from $O(N)$ to approximately $O(1)$. Thus, the tests generated do not lead to the worst case complexity of the inefficient program and its overall time complexity effectively becomes $O(N)$, enabling the candidate program to pass HardTests Type 2 (Regular) test cases.

The HardTests Type 3 (Hacking) inputs for this problem are generated using the following Python functions:

There are three hacking input generation functions: monotonically decreasing, monotonically increasing, and alternating sequences. The first generated input (monotonically decreasing) successfully increased the actual runtime complexity of the candidate program to $O(N^2)$, causing a timeout and consequently a failure on this test case.

A.5.3 Example 3: False Negative of TACO

In this example, we show an example of false negative caused by the lack of special judge function in TACO tests. We also show how HardTests can correctly evaluate the candidate program. Here's the problem description:

AtCoder ABC117A: Given an integer T *and an integer* X *(*$1 \leq T \leq 100$*,* $1 \leq X \leq 100$*). Compute the value of* T/X *with an error tolerance within* 10^{-3}.

A candidate program generated by Qwen2.5-Coder-14B-Instruct is:

This is clearly correct and passes all official test cases. It also passes all test cases from HardTests, but it fails on TACO's test cases. This is because using a simple string comparison function is insufficient due to potential differences in precision between the candidate output and the reference output. TACO does not provide a special output judging function for problems, which leads to false negatives. HardTests provides a special output judging function, shown below:

A.6 Downstream Training and Evaluation Details

Teacher-distillation training and evaluation details. In the teacher-distillation experiments, our model is trained with the same training parameters used to train OlympicCoder-7B (epochs=10, learning_rate=4e-5, batch_size=128, cosine learning rate schedule with a decay to 10% of the peak learning rate and 32,768 max length). The evaluations are sampled with temperature=0.7, top_p=0.95, max_new_tokens=16384.

Self-distillation training and evaluation details. In the self-distillation experiments, our model is trained with the following training parameters (epochs=20, learning_rate=4e-5, batch_size=128, cosine learning rate schedule with a decay to 10% of the peak learning rate and 32,768 max length). The evaluations are sampled with temperature=0.6, top_p=0.95, top_k=20, min_p=0, max_new_tokens=32768 as recommended by Qwen.

RL training and evaluation details. We use verl for RL training and firejail for sandboxing code execution. The rollouts are generated with temperature=1, top_p=0.95, top_k=20, min_p=0, response_length=24000, initial learning rate 5e-7. We use a global batch size of 32 and generate 32 samples per rollout. All our experiments are run on 8 NVIDIA H100 GPUs. We do not use KL divergence in our RL loss.

A.7 Test Case Generation Without an Oracle Model

In the case that an oracle program y^*, or an oracle test suite V^* does not exist for a problem x, such as when problems are synthetically generated, we propose a method, based on ALGO (Zhang et al., 2023) that synthesizes both the oracle and tests. To start, we prompt an LLM, such as Anthropic Claude 3.5 Sonnet, to generate a brute-force solution y_{bf} to the problem. Specifically, we encourage it to use inefficient methods such as exhaustive search and enumeration of the possible output space. This is founded on the observation that it is relatively easy to generate a solution that exhaustively searches the correct output, but more difficult to optimize it within a time complexity bound.

Then, an LLM is prompted to create a validator program and 10 edge test input generators, which are used to generate one test input each, $\{a_1, \ldots, a_{10}\}$. To prevent the y_{bf} from timing out when computing their respective outputs, we explicitly prompt the LLM to keep input values small. Once these test inputs are verified for correctness using the validator, the brute-force solution is used to generate the corresponding outputs $c_i = y_{bf}(a_i)$ for each input, resulting in a total of 10 input-output pairs as test cases. Finally, the LLM is prompted to create one maximum-length test case a_{max} with inputs at the upper bounds of the problem's constraints, designed to catch solutions that are functionally correct but inefficient. This test case is considered to be passsed as long as the program produces an output before timing out. Crucially, all 11 of the generated test cases $\{a_1, \ldots, a_{10}, a_{max}\}$ are designed to cause seemingly correct programs to fail, and none are generated using random inputs.

We compare this method to the baseline method outlined in AceCoder (Zeng et al., 2025a), which uses a direct prompt to generate 20 full test cases (inputs and corresponding outputs), also using Claude 3.5. Then, after prompting a stronger model such as Qwen2.5- Coder-32B-Instruct to generate a solution, the test cases that cause the solution to fail are considered hallucinated and are filtered out. Problems with fewer than 5 test cases after filtering are discarded.

To evaluate the accuracy of rewards that our test cases can give to model training, we evaluate the precision and recall over candidate programs generated by LLMs and written by humans on subsets of problems in HARDTESTS.

The quality of the test cases are verified using 165 Atcoder problems, each with 50 sample solutions. It is clear from these experiments (shown in Table 8) that our method can also work much better than the baseline even when oracle programs are not available. The false positive rate of HARDTESTGEN is only half as high as AceCoder, showing that deliberately crafting high-quality, hard test cases is crucial for effective program verifiers.

We will show some examples of the test generation process in the following sections.

Table 8: Performance comparison of oracle-free test generation algorithms based on error rates.

	False Positive Rate (FPR)	False Negative Rate (FNR)
AceCoder	32.49	2.59
HARDTESTGEN	17.67	2.19

A.7.1 Example 1: Brute-force oracle solution generated by ALGO

In this example, we show a brute-force oracle solution y_{bf} that ALGO generates. The problem is challenging, and while the LLM is unable to correctly solve it directly, its brute-force solution works. Here's the problem description:

AtCoder ABC301C: A single-player card game is popular in AtCoder Inc.
Each card in the game has a lowercase English letter or the symbol written on it. There is plenty number of cards for each kind. The game goes as follows.

- Arrange the same number of cards in two rows.
- Replace each card with with one of the following cards: a, t, c, o, d, e, r.
- If the two rows of cards coincide, you win. Otherwise, you lose.

To win this game, you will do the following cheat.

- Freely rearrange the cards within a row whenever you want after step 1.

You are given two strings S and T ($1 \leq |S|, |T| \leq 2 \times 10^5$), representing the two rows you have after step 1. Determine whether it is possible to win with cheating allowed. If it is possible to win with cheating allowed, print Yes; otherwise, print No.

The solution generated by Claude 3.5 Sonnet is:

This solution is evidently inefficient. A full analysis yields runtime in the bound of $O(n \log n 7^n)$, where $n = |S| = |T|$, but the exponential complexity is clear from the Cartesian product function in the implementation. For inputs $n > 10$, this algorithm quickly becomes intractable. However, for inputs $n \leq 10$ it is able to generate valid test outputs, allowing it to correctly evaluate the validity of submitted solutions. The test outputs it generates achieve a 100% accuracy, compared to actual execution results from the online judge.

A.7.2 Example 2: Test cases generated by ALGO

In this example we show a contest coding problem for which ALGO effectively generates a testing suite. Here's the problem description:

AtCoder cafeteria sells meals consisting of a main dish and a side dish. There are N types of main dishes, called main dish 1, main dish 2, $\dots$, main dish N. Main dish i costs a_i yen. There are M types of side dishes, called side dish 1, side dish 2, $\dots$, side dish M. Side dish i costs b_i yen.

A set meal is composed by choosing one main dish and one side dish. The price of a set meal is the sum of the prices of the chosen main dish and side dish.

However, for L distinct pairs (c_1, d_1), $\dots$, (c_L, d_L), the set meal consisting of main dish c_i and side dish d_i is not offered because they do not go well together. That is, $NM - L$ set meals are offered. (The constraints guarantee that at least one set meal is offered.)

Find the price of the most expensive set meal offered.

The input is given from Standard Input in the following format:
$N\ M\ L$
$a_1\ a_2\ \dots\ a_N$
$b_1\ b_2\ \dots\ b_M$

c_1 d_1
c_2 d_2
$\vdots$
c_L d_L

Constraints:
- $1 \leq N, M \leq 10^5$
- $0 \leq L \leq \min(10^5, NM - 1)$
- $1 \leq a_i, b_i \leq 10^9$

The first 3 edge test input generators created by ALGO are shown below, corresponding to the following test inputs. Note that the values are at the boundaries of the input bounds and follow clearly defined structures.

Also, the generator for the maximum-length test input a_{max} is shown here. It produces a test input where $N = M = 10^5$, which is the upper bound of the problem.

This test suite effectively achieves 100% accuracy on evaluating submissions, demonstrating that precise test inputs are crucial for oracle-free verifiers.

A.7.3 Example 3: Test cases generated by AceCoder

For the same Atcoder problem as Example A.7.2, AceCoder generates the following 16 test cases with inputs and outputs after filtering. While the LLM implicitly knows to generate edge test cases, shown in the maximal values of c_i, d_i, all of the test cases have relatively similar and low values of M and N.

These test cases fail to correctly categorize solutions that exceed the problem's time limit. One such example is shown below, which AceCoder falsely categorizes as a positive solution. Compared to Example A.7.2, in which ALGO generated test inputs as large as $N = M = 10^5$, the test cases from AceCoder are no larger than $N = M = 5$, making them unable to break inefficient programs. Without a brute-force reference oracle, and constrained by the requirement of generating input-output pairs simultaneously, the LLM used by AceCoder sticks to simple test cases that it can be confident are correct. Moreover, longer test cases are likelier to contain hallucinations, and get removed by their filtering process. As a result, their test cases are relatively weaker and result in less effective verifiers.