# Themis: Automated Constraint-Aware Test Synthesis Framework for Code Reinforcement Learning

**Shengyu Ye**[1], **Qi Liu**[1, 2*], **Hao Jiang**[1], **Zheng Zhang**[1], **Heng Yu**[1], **Zhenya Huang**[1, 2]

[1]State Key Laboratory of Cognitive Intelligence, University of Science and Technology of China, China
[2]Institute of Artificial Intelligence, Hefei Comprehensive National Science Center, China
{ysy007, jianghao0728, zhangzheng, yh112358_1321}@mail.ustc.edu.cn, {qiliuql, huangzhy}@ustc.edu.cn

## Abstract

Reinforcement learning (RL) has shown promise for enhancing code generation capabilities in large language models (LLMs), yet its effectiveness critically depends on high-quality test suites for reliable reward signals. Current approaches suffer from inadequate test case quantity and quality, leading to false positives (incorrect solutions passing verification) and slow positives (valid but suboptimal implementations), which corrupt RL training dynamics. We address these challenges through three key contributions: (1) We systematically analyze how low-quality test suites degrade Code RL performance via reward misalignment; (2) We propose Themis, an automated framework that transforms test case generation into code synthesis—first extracting problem constraints via template-guided parsing, then generating executable test generators through LLM-powered code synthesis, and finally validating tests through constraint-aware filtering; (3) We develop an error-guided test case reduction method that preserves error detection efficacy while reducing test set cardinality, thereby enhancing reinforcement learning training efficiency. Evaluated on programming competition datasets, Themis achieves 95 percent error detection rates, outperforming original test suites in most of the cases. When integrated into RL pipelines, models trained with Themis-generated tests demonstrate consistent 3-5 percent improvements across HumanEval, MBPP, and LiveCodeBench compared to the baseline, matching performance levels achieved with manually curated test suites. Our constraint-aware test synthesis framework ensures full automation while preserving semantic validity—critical for scaling RL training to complex code generation tasks. The framework's modular design also enables seamless integration with existing code data synthesis frameworks.

**Code** — https://github.com/ysy-phoenix/Themis.git

## 1 Introduction

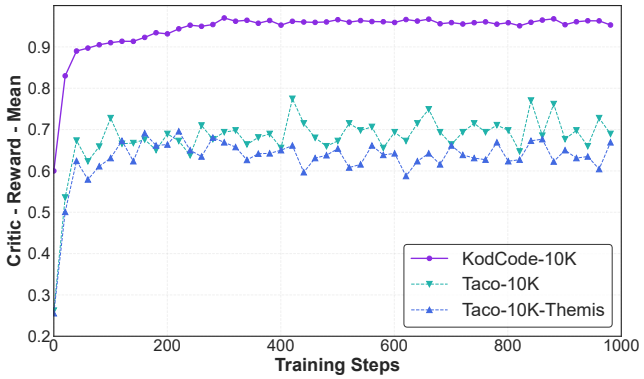
Recent advances in test-time scaling techniques (e.g., OpenAI's o1 (Jaech et al. 2024) and DeepSeek's R1 (Guo et al. 2025)) have demonstrated significant performance improvements in large language models (LLMs) for complex reasoning tasks through enhanced support for chain-of-thought (CoT) reasoning (Wei et al. 2022). These developments can be primarily attributed to the capability of reinforcement learning (RL) (Sutton, Barto et al. 1998) to systematically explore verifiable problem-solving trajectories (e.g., mathematical proofs (Xin et al. 2024) and program synthesis (Gulwani et al. 2017)). However, despite significant progress in RL-driven reasoning model scaling within mathematical domains (Luo et al. 2025b; Wen et al. 2025; Yu et al. 2025; Hu et al. 2025), the application of analogous techniques to programming tasks remains comparatively underdeveloped (Luo et al. 2025a; Zeng et al. 2025). This disparity stems from the scarcity of reliable and verifiable reward signals coupled with high-quality training datasets in code-related domains.

While synthetic datasets exist for code tasks (Huang et al. 2024; Xu et al. 2025, 2023), they suffer critical limitations for RL. Most support only supervised fine-tuning (Luo et al. 2023; Wei et al. 2023; Jiang et al. 2025, 2024) and lack adequate test cases. Even datasets with tests exhibit insufficient quantity and quality, causing high false positive rates (Li et al. 2022) that undermine RL training. As Figure 1 shows, flawed test suites misvalidate incorrect solutions, propagating reward miscalibration during optimization. This degrades model performance (Chen et al. 2021; Austin et al. 2021; Jain et al. 2024), sometimes below baselines. Code RL effectiveness depends critically on test suite quality and quantity; poor tests generate misleading rewards, derailing policy optimization. This reveals a key gap: beyond problem collection, robust methods for generating adversarially validated test suites are urgently needed to ensure precise correctness verification and stable reward signals.

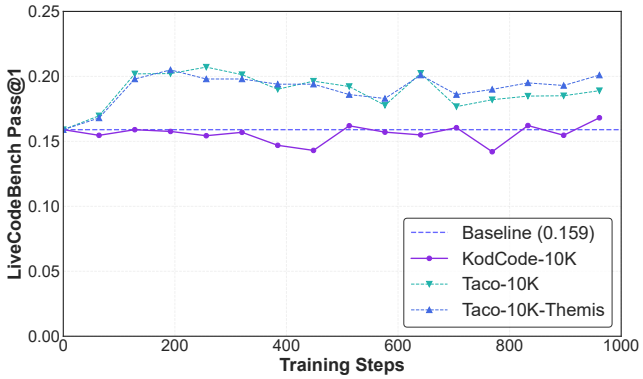
While prior work has explored automated test case generation for code evaluation (Liu et al. 2024b), existing approaches remain hindered by three core challenges that impede their applicability to RL-driven code generation:

- **Challenge 1: Inadequate Test Quantity.** Methods relying on prompting techniques (e.g., generating assert statements (Chen et al. 2022) or pytest (Xu et al. 2025)) often produce insufficiently diverse test cases. Sparse test suites elevate false positive rates, where incorrect solutions pass verification, compromising RL reward accuracy.
- **Challenge 2: Poor Test Quality.** Low-quality tests introduce slow positives (e.g., accepting inefficient yet correct code) (Zhang et al. 2024a), incentivizing reward hack-

*Corresponding Author.




(a) Rewards during training




(b) Pass@1 on LiveCodebench

Figure 1: The variations of rewards during training under three distinct dataset settings (with varying test case qualities) and the Pass@1 on LiveCodeBench.

ing (Skalse et al. 2022). While mutation testing (Budd 1980) addresses coverage gaps via generated variants, synthetically created tests frequently violate problem constraints, yielding invalid tests.

- **Challenge 3: Lack of Full Automation.** Most approaches require human intervention for quality assurance—feasible for static benchmarks (Zhuo et al. 2024; Jain et al. 2024) but prohibitively costly for RL scenarios needing large-scale corpora. This dependence creates scalability bottlenecks with standard RL training requiring thousands of automatically verifiable instances.

We propose Themis, a fully automated framework for generating high-quality test suites addressing these challenges. As shown in Figure 2, Themis adopts a three-stage pipeline: (1) **Constraint Extraction**, parsing problem descriptions into structured constraints (Liu et al. 2024c); (2) **Test Code Generation**, producing executable code adhering to constraints; (3) **Validation and Filtering**, executing tests against ground-truth solutions and filtering constraint-satisfying cases. For RL efficiency, Themis employs **error-guided test reduction**, eliminating redundant tests while maintaining coverage (Ivanković et al. 2019). This design ensures scalability, validity, automation, and efficiency, directly addressing the three challenges.

**Contributions**:

- **Method**: **Themis** automates high-quality test suite generation via constraint extraction, code-based test synthesis, and execution-based validation. Error-guided test compaction optimizes RL training. Plug-and-play for existing pipelines.
- **Insight**: We first systematically analyze how *false positives* (erroneous solutions passing tests) and *slow positives* (inefficient valid solutions) corrupt RL reward signals, degrading downstream performance.
- **Results**: Models trained with Themis-generated testcases match manual testcases performance from online judge (OJ) platforms. Augmenting test-deficient datasets yields 3%–5% average downstream improvements across code benchmarks.

## 2 Themis Framework

Current approaches in Code RL training rely on two dominant paradigms: (1) *Function-level synthesis* with predefined signatures (e.g., HumanEval (Chen et al. 2021), MBPP (Austin et al. 2021), LeetCode), and (2) *Standard I/O-driven programs* requiring self-contained code with I/O handlers (e.g., Codeforces (Codeforces Team 2025), Olympiad in Informatics (OI) contests). Conventional unit testing suffices for simple function-level tasks, but complex problems and I/O programs face significant challenges due to strict formatting and higher complexity, making automated test generation difficult.

Inspired by human experts in OI competitions formalizing problem-specific constraints and manually implementing custom generators, we propose Themis. As shown in Figure 2, Themis integrates three critical phases: (1) Problem understanding and constraint extraction, parsing specifications into invariants; (2) Test code generation, synthesizing executable test generators adhering to constraints; (3) Validation and filtering, using execution to eliminate invalid tests. By mirroring the human constraint-first methodology and leveraging LLMs, Themis addresses limitations of prior methods (Chen et al. 2022; Liu et al. 2024b; Xu et al. 2025), ensuring high quality and coverage for complex problems.

### 2.1 Constraint Extraction

The process initiates with problem constraint extraction, a critical programming challenge design step. Constraints multidimensional specifications including: (1) numerical boundaries (e.g., $0 < n < 1000$), (2) data type requirements, (3) structural specifications (e.g., specialized binary trees), and (4) semantic validations (e.g., lowercase exclusivity). Formally integrating these into test case generation achieves dual goals: enhanced model-task alignment through precise specification grounding, and improved evaluation robustness via constraint-aware sampling. This ensures test suites maintain syntactic validity and semantic consistency, strengthening automated judge reliability.

We employ a **template-guided generation** methodology

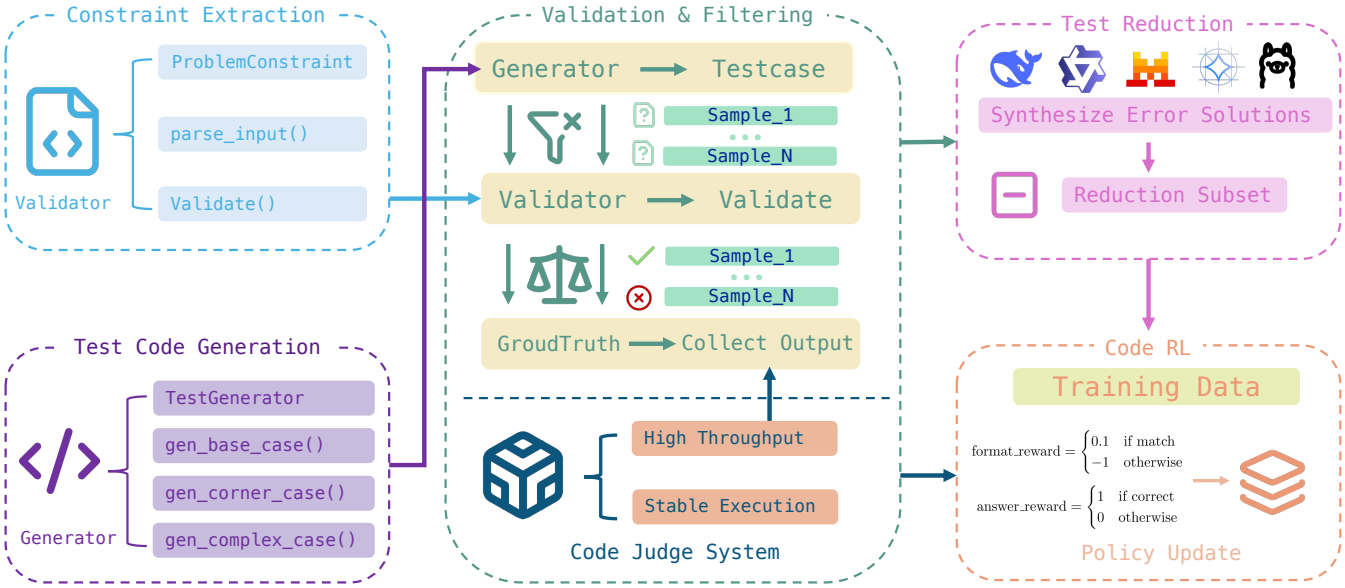



Figure 2: Pipeline of Themis. (1) Template-guided constraint extraction and validator synthesis, (2) Test input generator construction, (3) Constraint-aware test validation with ground truth execution to collect output, (4) Error-guided test reduction for RL training, all supported by a unified high-throughput code judge system.

that leverages structured prompting to facilitate constraint identification in large language models. Our approach utilizes minimalist templates to specify input format protocols, numerical boundary conditions, and structural validity requirements while intentionally avoiding over-constraining prompts. This preserves models' code comprehension through flexible scaffolding, requiring only three fixed elements as shown in Figure 3: the `ProblemConstraint` class nomenclature and mandatory `parse_input()` (parse string-based individual test case inputs into valid constituent components) and `validate()` (further verifies whether each component satisfies the specified constraint conditions) method signatures.

These core constraints ensure automated execution compatibility while maintaining an optimal exploration-exploitation balance during constraint elicitation. The same template-driven paradigm extends seamlessly to subsequent test code generation phases, demonstrating consistent effectiveness in preserving syntactic-semantic alignment across generation tasks.

## 2.2 Test Code Generation

Traditional approaches predominantly employ assertion-based testing, which suffers from two limitations: misalignment with problem specifications, especially for dynamic constraints, and prohibitive inefficiency at scale. Mutation-based methods improve partially but risk semantic integrity violations via unintended constraint breaches.

To overcome these, we propose reframing test case generation as *executable code synthesis*. Leveraging constraint-aware specifications, we utilizes large language models to synthesize parametric test generators rather than individual instances. This paradigm offers three advantages: (1) Formal logic structures inherently enforce specification completeness; (2) Parametric execution enables efficient large-scale test suite generation; (3) Deterministic execution guarantees syntactic and semantic constraint adherence.

Building on our Phase I template-guided paradigm, we instruct the LLM via structured prompting to construct the `TestGenerator` class (Fig. 3) with three core methods: **(1) basic** (typical scenarios), **(2) corner** (edge conditions), and **(3) complex** (adversarial inputs). Although not all problems require every category, this tripartite structure ensures pipeline consistency and interface standardization while maintaining flexibility for complex domains.

Systematic randomization via seeded PRNGs ensures deterministic reproducibility, achieving **scalability** through parametric sampling and **diversity-quality balance** via controlled stochasticity. This enables rigorous validation of functional correctness and performance across input scales.

## 2.3 Validation and Filtering

Stage 3 first executes Stage 2 generators to produce candidate test cases, followed by a dual-filtering mechanism. Primary-stage constraint validators perform **syntactic validation and specification-conformance verification**, retaining only inputs satisfying both structural parsability and constraint compliance. Validated inputs undergo **groundtruth execution** via reference solutions to: (1) filter ill-posed (e.g., no-solution) or intractable cases (via runtime constraints), and (2) generate outputs through deterministic execution. Critically, Stage 2 synthesizes solely inputs, while outputs derive from pre-existing oracles—preserving semantic validity and reproducibility.

For scalable dataset generation, we developed a lightweight code judge backend. Supporting function-

**Problem Description**

Tanya starts at box $s$ in a row of $n$ boxes (numbered from 1 to $n$). Each box $i$ contains $r_i$ candies of color $c_i$ (R/G/B).

- **Movement**: 1 second per move to any adjacent box
- **Eating**: Instantly consume all candies in current box
- **After eating**: Box becomes completely empty

**Constraints for eating candies:**

- No two consecutively eaten boxes can have the same color.
- Each newly eaten box must have strictly more candies than the last.

**Goal:** Eat at least $k$ candies in minimum total movement time. If impossible to collect $k$ candies, output $-1$.

**Test Format**

- $n, s, k (1 \leqslant n \leqslant 50, 1 \leqslant s \leqslant n, 1 \leqslant k \leqslant 2000)$
- $r_1 ... r_n (1 \leqslant r_i \leqslant 50)$
- String of $n$ letters (R/G/B) (colors of boxes)

**Output:** Minimum movement time or $-1$ if impossible

**Example**

**Input:**
```
5 3 10
1 2 3 4 5
RGBRR
```

**Output:** 4

**Explanation:** Move 2→3→4→5, eating 2+3+5 candies.

**ProblemConstraint**

```python
class ProblemConstraint:
    def __init__(self):
        self.max_n, self.max_k, self.max_r = 50, 2000, 50
        self.valid_colors = {'R', 'G', 'B'}
    def parse_input(self, raw_input: str) -> dict:
        try:
          lines = raw_input.strip().split('\n')
          n, s, k = map(int, lines[0].split())
          r = list(map(int, lines[1].split()))
          c = lines[2].strip()
        except Exception as e:
            print(e)
            return None
        return {'n':n, 's':s, 'k':k, 'r':r, 'c':c}
    def validate(self, raw_input: str) -> bool:
        p = self.parse_input(raw_input)
        if not p:
            return False
       n, s, k, r, c = p['n'], p['s'], p['k'], p['r'], p['c']
        return (
            1 <= n <= self.max_n and 1 <= s <= n and
            1 <= k <= self.max_k and len(r) == n and
            all(1 <= ri <= self.max_r for ri in r) and
            len(c) == n and
            all(color in self.valid_colors for color in c)
        )
```

**TestGenerator**

```python
class TestGenerator:
    def __init__(self, nbase=30, ncorner=10, ncomplex=60):
        random.seed(seed)
        self.colors = ['R', 'G', 'B']
        ...
    def gen(self, n, s, k, r, c) -> str:
        return f"{n} {s} {k}\n" + ' '.join(map(str, r)) + \
            '\n' + ''.join(c)
    def gen_base_case(self) -> List[str]:
        cases = []
        for _ in range(self.nbase):
            n = random.randint(10, 50)
            k = random.randint(1, 2000)
            s = random.randint(1, n)
            r = [random.randint(1, 50) for _ in range(n)]
          c = [random.choice(self.colors) for _ in range(n)]
            cases.append(self.gen(n, s, k, r, c))
        return cases

    def gen_corner_case(self) -> List[str]:
        cases = []
        cases.append(self.gen(1, 1, 1, [1], ['R']))
        cases.append(self.gen(50, 50, 2000, \
            [1] * 50, ['G'] * 50))
        ...
        return cases
```

Figure 3: Example of validator and generator: (1) Template-guided constraint extraction via the `ProblemConstraint` class with mandatory `parse_input()` and `validate()` methods that formalize numerical boundaries, structural requirements, and semantic rules; (2) Program synthesis of parametric test generators through the `TestGenerator` class, implementing systematic generation of canonical, corner, and complex test cases via seeded randomization.

based and stdio interfaces, it processes test cases as minimal parallelization units for high concurrency and throughput. Beyond Stage 3 validation, the architecture serves as a reusable code judge for subsequent reinforcement learning phases, enabling batch optimization and parallelized reward computation acceleration.

## 2.4 Test Reduction

Despite generating sufficient test cases, employing the full set for RL reward evaluation would be prohibitively expensive computationally. We therefore prioritize identifying minimal high-quality test suites that maximize error detection—a challenge noted in EvalPlus (Liu et al. 2024b). While we experimented with code coverage metrics and test adequacy criteria, these proved ineffective for competitive programming problems. The limitation stems from the elevated difficulty of Code RL training data (sourced from contest platforms (Codeforces Team 2025; Hendrycks et al. 2021)), where traditional software engineering approaches (Budd 1980; Serebryany 2016; King 1976) fail to capture subtle functional discrepancies in complex solutions.

To address this, we propose an error-guided reduction method. To ensure diversity, we adopt a persona-driven approach (Ge et al. 2024) for generating synthetic erroneous solutions. The core insight leverages differently-sized models to emulate programmers at varying skill levels, guided by prompts to generate diverse solutions. Notably, for a problem with optimal $O(n)$ complexity, an expert might produce $O(n)$ code, while a novice yields $O(n^2)$, and intermediates $O(n \log n)$—each passing different test cases. This enables effective identification of error-revealing tests. We

further integrate heuristics: retaining longest inputs (complexity validation) and shortest inputs (boundary cases), followed by randomized selection. This methodology efficiently obtains minimal high-quality test suites while maintaining error detection capability.

# 3 Evaluation

## 3.1 Experiment Setup

**Dataset and Metrics** To validate the effectiveness of our test case generation framework, we conduct two experiments on the taco-verified dataset (Li et al. 2023): one evaluating test case quality and the other performing end-to-end Code RL training to assess downstream task performance. For test case evaluation, we propose a novel error detection rate metric (defined as detected erroneous solutions / total erroneous solutions), with erroneous solutions synthesized following the methodology in the Test Reduction section. For RL training, we report pass@1 (Chen et al. 2021) results on downstream benchmarks including HumanEval (Chen et al. 2021), MBPP (Austin et al. 2021), LiveCodeBench(v5) (Jain et al. 2024).

**Open Source LLMs** For the test case generation phase, we evaluate various open-source models of different scales as proxies, including the Qwen2.5-Coder series (Hui et al. 2024), DeepSeek-Distil-Qwen series (Guo et al. 2025), DeepSeek-V3-0324 (Liu et al. 2024a), and DeepSeek-R1 (Guo et al. 2025). Samples for the latter two models are generated via their official APIs, while others are deployed locally using sglang (Zheng et al. 2023) for inference. For RL training, we adopt mainstream configurations and focus on end-to-end training with Qwen2.5-7B Base/Instruct and Qwen2.5-Coder-7B Base/Instruct models.

**RL Training Settings** Our implementation largely follows Code-R1 (Liu and Zhang 2025) and DeepCoder (Luo et al. 2025a), employing the verl (Sheng et al. 2024) framework for training, utilizing vllm (Kwon et al. 2023) as the rollout engine, and adopting naive GRPO (Shao et al. 2024). For hyperparameters, we use the AdamW optimizer with a constant learning rate of $5\times10^{-7}$. During rollout, the prompt batch size is set to 64 with 4 responses sampled per prompt. For training, we configure a mini-batch size of 16 and micro batch size of 4. To encourage exploration and reduce GPU memory consumption, we remove the KL penalty term (i.e., without using a reference model). All experiments are conducted on 4 Nvidia A100 GPUs.

## 3.2 Test Case Quality

**Test Case Quality Evaluation.** For cost efficiency, we cap generation attempts at five for both the verifier and generator, while allowing the model to autonomously determine test inputs per problem based on difficulty assessment, requiring only a minimum of 16 test cases during execution. Table 1 compares test case generation methodologies using three pipeline metrics: Stage 1: validator generation success rate (valid syntax/semantics), Stage 2: functional generator success rate, and Stage 3: test case retention post-verification. Overall reports the percentage of programming problems successfully producing test cases through all stages. Error detection rates are measured under four configurations: Full Suite (all generated cases), Reduced Suite (minimized subset via optimization), EvalPlus (cases reproduced via EvalPlus methodology), and Original (original dataset cases).

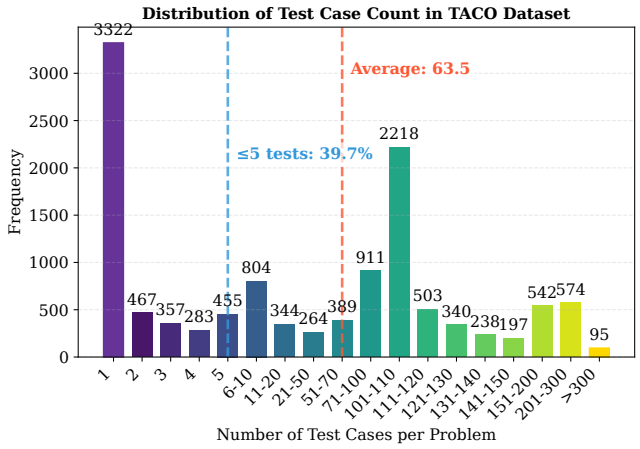
As shown in Table 1, our framework successfully generates sufficient test cases for 95% of programming problems when using the best-performing model, achieving a 93% error detection rate that surpasses the original dataset. This performance gap stems from the original dataset containing $\sim$ 4,000 problems with $\leq$ 5 test cases (as shown in Figure 4a for distribution), revealing significant test scarcity even in state-of-the-art Code RL datasets and highlighting the necessity of automated synthesis. Besides, Our test cases achieve significantly higher quality than EvalPlus, which relies heavily on manual inspection in its process. In contrast, test cases generated by Themis exhibit a more balanced distribution(see Figure 4b), ensuring a minimum of 16 cases per problem to thoroughly validate solution correctness.

**Discussion of Test Case Generation Capabilities.** Furthermore, we observe that test generation inherently depends on the model's code capabilities: stronger models consistently achieve higher success rates across all evaluation stages, while our training-free framework fundamentally relies on agent models' capabilities to seamlessly integrate future improvements in foundation models. Notably, reasoning-augmented models demonstrate markedly superior performance compared to their non-reasoning counterparts, though this advantage comes at the cost of substantially longer response lengths and significantly higher token consumption during inference.

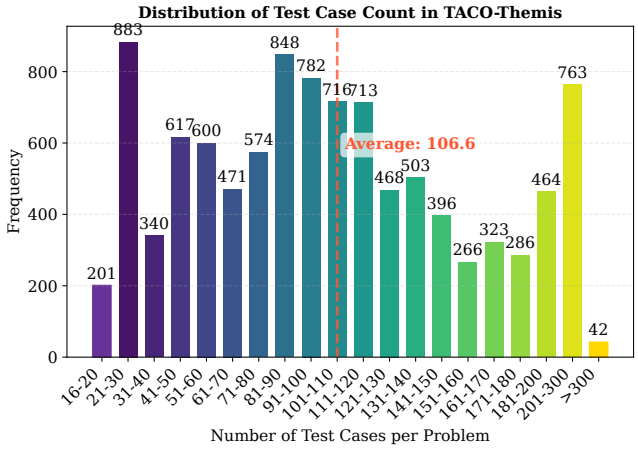
**The Necessity of Validation and Reduction**. Low retention rates (50-70%) at stage 3 further highlight the crucial role of the verification-filtering pipeline in ensuring test quality, with substantial quantities of non-compliant and semantically invalid test cases being effectively eliminated. Our reduction method achieves 5× compression of test sets on average (yielding compact subsets of $\sim$ 20 test cases) while preserving comparable error detection capability, with the reduced subsets being empirically employed in subsequent reinforcement learning experiments.

## 3.3 RL Results

**Overview of Qwen2.5-7B Series RL Results.** Table 2 presents the main results of end-to-end RL training. For each model, the first row reports baseline performance, the second row shows results using original test cases from the Taco dataset for RL training, and the third row demonstrates outcomes with test cases generated by our framework. Notably, despite the original test cases being meticulously designed by human experts, our framework-generated test cases achieve consistent performance improvements across models, matching or exceeding human-crafted cases. This enhancement stems from the higher fault detection rate of our test cases, which provides more precise reward signals during RL training. Compared to the baseline, our approach yields an average improvement of 3%–5% across three benchmarks, with a 1% additional gain over results using Taco's original test cases.




(a) Distribution of TACO test cases




(b) Distribution of Themis enhanced TACO test cass

Figure 4: Original and enhanced data distributions. Reduced number of instances post-augmentation stems from both rigorous data cleaning and unsuccessful synthesis of a small subset of problems.

| Model | Size | Generation Stages | | | Overall | Full | Reduction | Evalplus | Original |
|---|---|---|---|---|---|---|---|---|---|
| | | Stage1 | Stage2 | Stage3 | | | | | |
| DeepSeek-R1 | 671B | 93.5 | 94.3 | 75.8 | 92.1 | **95.6** | **94.7** | 66.2 | 89.3 |
| DS-V3-0324 | 671B | 90.1 | 92.1 | 68.6 | 89.2 | 92.1 | 91.3 | 47.8 | 89.3 |
| QWQ | 32B | 92.3 | 93.6 | 70.3 | 91.1 | 93.3 | 92.5 | 55.3 | 89.3 |
| Qwen2.5-Coder | 32B | 86.5 | 85.8 | 63.1 | 84.7 | 90.6 | 90.2 | 43.3 | 89.3 |
| | 14B | 83.1 | 82.3 | 60.4 | 82.0 | 90.0 | 89.7 | 42.0 | 89.3 |
| | 7B | 80.1 | 81.2 | 56.1 | 79.6 | 88.5 | 87.6 | 40.9 | 89.3 |
| | 3B | 75.5 | 74.3 | 52.1 | 73.1 | 87.3 | 86.9 | 37.6 | 89.3 |
| DeepSeek-R1-Distill-Qwen | 32B | 91.1 | 92.4 | 71.1 | 90.2 | 93.5 | 92.9 | 48.9 | 89.3 |
| | 14B | 87.1 | 86.5 | 67.7 | 85.3 | 92.3 | 91.7 | 46.7 | 89.3 |
| | 7B | 84.1 | 83.6 | 63.7 | 82.7 | 90.5 | 90.3 | 45.5 | 89.3 |
| | 1.5B | 78.7 | 77.6 | 52.3 | 77.1 | 88.4 | 88.0 | 43.8 | 89.3 |

Table 1: Quality of test cases generated by different models.

**Ablation Study.** We further conducted an ablation study to investigate how low-quality test cases impact Code RL training. Training Qwen2.5-Coder-7B-Instruct on KodCode (Xu et al. 2025) data, Table 3 presents our findings. The second row shows results from training on 10K randomly selected samples, which performed poorly due to severe false positives (see Figure 1), where rewards rapidly converged to 1, yielding near-zero advantages and minimal model updates. Subsequent refinement by filtering samples with insufficient test cases or low difficulty mitigated false positives but still underperformed the baseline. Further enhancement of the data using the Themis framework ultimately produced competitive results comparable to those trained on the Taco dataset. These results demonstrate that high-quality test cases are critical for effective Code RL training, and highlight capability of Themis to synthesize reliable test cases, which can be seamlessly integrated with existing code synthesis frameworks to construct more robust RL-driven code datasets.

## 4 Related Work

**Test Case Generation Approaches.** Test case generation can be categorized into traditional software engineering methods and large language model (LLM)-based approaches. Traditional black-box techniques like fuzz testing (Miller, Fredriksen, and So 1990) generate random inputs without understanding system internals, while white-box methods (King 1976) analyze source code to create high-quality test cases. Grey-box approaches such as coverage-guided fuzzing (Serebryany 2016) optimize input generation through coverage feedback during mutation. Recent advances demonstrate LLMs' effectiveness in test case generation. Empirical studies (Schäfer et al. 2023; Liu et al. 2021) reveal that LLMs can produce diverse and seman-

| Model | Size | LiveCodeBench v5 | | | | HumanEval | | MBPP | | Avg |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *Overall* | *Easy* | *Medium* | *Hard* | *Base* | + | *Base* | + | |
| Qwen2.5-7B | baseline | 9.9 | 29.8 | 5.2 | 1.6 | 57.9 | 50.6 | 74.9 | 62.9 | 41.1 |
| | +taco | 17.9 | 58.7 | 9.3 | 2.5 | 79.0 | 73.1 | 79.2 | 67.1 | 52.7 |
| | +Themis | **18.6** | 55.9 | 10.9 | 4.5 | 80.7 | **74.8** | 80.3 | **68.2** | **56.7** |
| Qwen2.5-7B-Instruct | baseline | 13.6 | 44.1 | 9.0 | 1.7 | 84.8 | 77.1 | 79.2 | 68.0 | 52.9 |
| | +taco | 18.6 | 58.7 | 10.5 | 3.4 | 85.1 | 78.0 | 81.2 | 69.3 | 55.3 |
| | +Themis | **19.8** | 58.7 | 12.8 | 4.2 | 85.4 | **79.9** | 84.4 | **71.7** | **57.1** |
| Qwen2.5-Coder-7B | baseline | 11.9 | 40.0 | 5.6 | 1.5 | 61.6 | 53.0 | 76.9 | 62.9 | 42.6 |
| | +taco | 18.7 | 56.2 | 10.3 | 4.9 | 83.6 | 77.6 | 80.0 | 68.4 | 54.9 |
| | +Themis | **19.3** | 57.2 | 12.4 | 4.1 | 84.4 | **78.4** | 81.2 | **69.5** | **55.7** |
| Qwen2.5-Coder-7B-Instruct | baseline | 15.9 | 53.9 | 7.2 | 2.0 | 88.4 | 84.1 | 83.5 | 71.7 | 57.2 |
| | +taco | 18.7 | 57.6 | 9.5 | 4.7 | 88.6 | 84.4 | 83.9 | 70.7 | 57.9 |
| | +Themis | **20.1** | 58.7 | 12.8 | 5.0 | 89.0 | **85.4** | 85.9 | **72.8** | **59.4** |

Table 2: Results of end-to-end RL training on the TACO dataset.

| Setting | LiveCodeBench v5 | | | | HumanEval | | MBPP | | Avg |
|---|---|---|---|---|---|---|---|---|---|
| | *Overall* | *Easy* | *Medium* | *Hard* | *Base* | + | *Base* | + | |
| Qwen2.5-Coder-7B-Instruct | 15.9 | 53.9 | 7.2 | 2.0 | 88.4 | 84.1 | 83.5 | 71.7 | 57.2 |
| +KodCode random-10k | 16.4 | 49.3 | 7.8 | 5.3 | 87.9 | 82.1 | 84.7 | 71.7 | 56.7 |
| +KodCode filter-10k | 17.5 | 53.7 | 10.5 | 3.4 | 88.3 | 83.6 | 84.0 | 72.1 | 57.7 |
| +KodCode-Themis filter-10k | **19.4** | 57.1 | 11.6 | 5.0 | 89.0 | **84.8** | 84.7 | **72.3** | **58.8** |

Table 3: Ablation experiment results of KodCode.

tically meaningful unit tests. Frameworks such as OpenCoder (Huang et al. 2024) and AceCoder (Zeng et al. 2025) utilize teacher models to strengthen reliability through unit test generation. Notably, LLM-generated test cases can also synergize with conventional techniques like type-aware mutation (Liu et al. 2024b) to extend test coverage. Moreover, considerable attention has been paid to test fairness (Zhang et al. 2025, 2024b) and cognitive diagnosis (Wang et al. 2023) in recent studies.

**LLM for Code Generation.** Driven by advancements in pre-training methodologies and the availability of large-scale code datasets, code-specialized large language models (LLMs)—notably Codex (Chen et al. 2021), DeepSeek-Coder (Guo et al. 2024), and QwenCoder (Hui et al. 2024)—have demonstrated exceptional proficiency and are widely applied in downstream coding tasks such as program translation(Liu, Li, and Zhang 2023), code understanding(Nam et al. 2024), and program repair(Xia, Wei, and Zhang 2023). Subsequent innovations, such as WizardCoder (Luo et al. 2023) and Magicoder (Wei et al. 2023), enhance code LLMs' complex instruction-tuning capabilities via adaptations of Evol-Instruct-style methods to the code domain. Recent efforts like DeepCoder (Luo et al. 2025a) and AceCoder (Zeng et al. 2025) further extend code reasoning boundaries through reinforcement learning (RL). For systematic evaluation, benchmarks span foundational assessments (HumanEval (Chen et al. 2021), MBPP (Austin et al. 2021)), enhanced metrics (EvalPlus (Liu et al. 2024b)), and complex reasoning tests (LiveCodeBench (Jain et al. 2024), BigCodeBench (Zhuo et al. 2024)).

## 5 Conclusion

In this work, we propose Themis, a lightweight multi-stage framework for automated test case synthesis. By reframing test case generation as constrained code synthesis, it first synthesizes a validator to extract constraints, then generates an executable test generator for inputs, and finally validates test cases via constraint-aware filtering and execution using ground-truth solutions to collect outputs. Extensive experiments show Themis-synthesized test suites achieve 95% error detection. When used in end-to-end RL training, they deliver 3–5% performance gains on downstream code generation benchmarks—surpassing even human-designed test cases from programming competition platforms.

Our framework establishes a novel paradigm for reliable test case generation in code-oriented reinforcement learning. Future directions include extending Themis to support broader programming paradigms beyond algorithmic challenges, such as API-driven development and tool-augmented execution. Additionally, the constraint extraction mechanism shows promise for enhancing program synthesis in mathematical reasoning and formal verification domains through verifiable specification generation, with potential applications to mathematical problem-solving tasks.

## Acknowledgments

This work was supported by the National Natural Science Foundation of China (62337001, 623B1020), the Key Technologies R & D Program of Anhui Province (No. 202423k09020039), and the Fundamental Research Funds for the Central Universities.

## References

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Budd, T. A. 1980. *Mutation analysis of program test data*. Yale University.

Chen, B.; Zhang, F.; Nguyen, A.; Zan, D.; Lin, Z.; Lou, J.-G.; and Chen, W. 2022. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Codeforces Team. 2025. Codeforces: Competitive Programming Platform. https://codeforces.com. Accessed: 2025-11-21.

Ge, T.; Chan, X.; Wang, X.; Yu, D.; Mi, H.; and Yu, D. 2024. Scaling synthetic data creation with 1,000,000,000 personas. *arXiv preprint arXiv:2406.20094*.

Gulwani, S.; Polozov, O.; Singh, R.; et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2): 1–119.

Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; and Steinhardt, J. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS*.

Hu, J.; Zhang, Y.; Han, Q.; Jiang, D.; Zhang, X.; and Shum, H.-Y. 2025. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model. *arXiv preprint arXiv:2503.24290*.

Huang, S.; Cheng, T.; Liu, J. K.; Hao, J.; Song, L.; Xu, Y.; Yang, J.; Liu, J.; Zhang, C.; Chai, L.; et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Dang, K.; et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186*.

Ivanković, M.; Petrović, G.; Just, R.; and Fraser, G. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 955–963.

Jaech, A.; Kalai, A.; Lerer, A.; Richardson, A.; El-Kishky, A.; Low, A.; Helyar, A.; Madry, A.; Beutel, A.; Carney, A.; et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Jain, N.; Han, K.; Gu, A.; Li, W.-D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; and Stoica, I. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974*.

Jiang, H.; Liu, Q.; Li, R.; Ye, S.; and Wang, S. 2024. CursorCore: Assist Programming through Aligning Anything. *CoRR*, abs/2410.07002.

Jiang, H.; Liu, Q.; Li, R.; Zhao, Y.; Ma, Y.; Ye, S.; Lu, J.; and Su, Y. 2025. VERSE: Verification-based Self-Play for Code Instructions. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(23): 24276–24284.

King, J. C. 1976. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394.

Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J. E.; Zhang, H.; and Stoica, I. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Li, R.; Fu, J.; Zhang, B.-W.; Huang, T.; Sun, Z.; Lyu, C.; Liu, G.; Jin, Z.; and Li, G. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Liu, F.; Li, J.; and Zhang, L. 2023. Syntax and domain aware model for unsupervised program translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 755–767. IEEE.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2024b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Liu, J.; and Zhang, L. 2025. Code-R1: Reproducing R1 for Code with Reliable Rewards. https://github.com/ganler/code-r1.

Liu, M. X.; Liu, F.; Fiannaca, A. J.; Koo, T.; Dixon, L.; Terry, M.; and Cai, C. J. 2024c. " We Need Structured Output": Towards User-centered Constraints on Large Language Model Output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 1–9.

Liu, Q.; Huang, Z.; Yin, Y.; Chen, E.; Xiong, H.; Su, Y.; and Hu, G. 2021. EKT: Exercise-Aware Knowledge Tracing for

Student Performance Prediction. *IEEE Trans. Knowl. Data Eng.*, 33(1): 100–115.

Luo, M.; Tan, S.; Huang, R.; Shi, X.; Xin, R.; Cai, C.; Patel, A.; Ariyak, A.; Wu, Q.; Zhang, C.; Li, L. E.; Popa, R. A.; and Stoica, I. 2025a. DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51. Notion Blog.

Luo, M.; Tan, S.; Wong, J.; Shi, X.; Tang, W. Y.; Roongta, M.; Cai, C.; Luo, J.; Li, L. E.; Popa, R. A.; and Stoica, I. 2025b. DeepScaleR: Surpassing O1-Preview with a 1.5B Model by Scaling RL. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2. Notion Blog.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *The Twelfth International Conference on Learning Representations*.

Miller, B. P.; Fredriksen, L.; and So, B. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12): 32–44.

Nam, D.; Macvean, A.; Hellendoorn, V.; Vasilescu, B.; and Myers, B. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

Schäfer, M.; Nadi, S.; Eghbali, A.; and Tip, F. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

Serebryany, K. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, 157–157. IEEE.

Shao, Z.; Wang, P.; Zhu, Q.; Xu, R.; Song, J.; Bi, X.; Zhang, H.; Zhang, M.; Li, Y.; Wu, Y.; et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Sheng, G.; Zhang, C.; Ye, Z.; Wu, X.; Zhang, W.; Zhang, R.; Peng, Y.; Lin, H.; and Wu, C. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv: 2409.19256*.

Skalse, J.; Howe, N.; Krasheninnikov, D.; and Krueger, D. 2022. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35: 9460–9471.

Sutton, R. S.; Barto, A. G.; et al. 1998. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Wang, F.; Liu, Q.; Chen, E.; Huang, Z.; Yin, Y.; Wang, S.; and Su, Y. 2023. NeuralCD: A General Framework for Cognitive Diagnosis. *IEEE Trans. Knowl. Data Eng.*, 35(8): 8312–8327.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35: 24824–24837.

Wei, Y.; Wang, Z.; Liu, J.; Ding, Y.; and Zhang, L. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Wen, L.; Cai, Y.; Xiao, F.; He, X.; An, Q.; Duan, Z.; Du, Y.; Liu, J.; Tang, L.; Lv, X.; Zou, H.; Deng, Y.; Jia, S.; and Zhang, X. 2025. Light-R1: Curriculum SFT, DPO and RL for Long COT from Scratch and Beyond. *arXiv preprint arXiv:2503.10460*.

Xia, C. S.; Wei, Y.; and Zhang, L. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 1482–1494. IEEE.

Xin, H.; Ren, Z.; Song, J.; Shao, Z.; Zhao, W.; Wang, H.; Liu, B.; Zhang, L.; Lu, X.; Du, Q.; et al. 2024. Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*.

Xu, C.; Sun, Q.; Zheng, K.; Geng, X.; Zhao, P.; Feng, J.; Tao, C.; and Jiang, D. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Xu, Z.; Liu, Y.; Yin, Y.; Zhou, M.; and Poovendran, R. 2025. KodCode: A Diverse, Challenging, and Verifiable Synthetic Dataset for Coding. *arXiv preprint arXiv: 2503.02951*.

Yu, Q.; Zhang, Z.; Zhu, R.; Yuan, Y.; Zuo, X.; Yue, Y.; Fan, T.; Liu, G.; Liu, L.; Liu, X.; et al. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Zeng, H.; Jiang, D.; Wang, H.; Nie, P.; Chen, X.; and Chen, W. 2025. ACECODER: Acing Coder RL via Automated Test-Case Synthesis. *arXiv preprint arXiv:2502.01718*.

Zhang, K.; Wang, D.; Xia, J.; Wang, W. Y.; and Li, L. 2024a. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems*, 36.

Zhang, Z.; Li, N.; Liu, Q.; Li, R.; Gao, W.; Mao, Q.; Huang, Z.; Yu, B.; and Tao, D. 2025. The other side of the coin: Exploring fairness in retrieval-augmented generation. *arXiv preprint arXiv:2504.12323*.

Zhang, Z.; Wu, L.; Liu, Q.; Liu, J.; Huang, Z.; Yin, Y.; Zhuang, Y.; Gao, W.; and Chen, E. 2024b. Understanding and improving fairness in cognitive diagnosis. *Sci. China Inf. Sci.*, 67(5).

Zheng, L.; Yin, L.; Xie, Z.; Sun, C.; Huang, J.; Yu, C. H.; Cao, S.; Kozyrakis, C.; Stoica, I.; Gonzalez, J.; Barrett, C. W.; and Sheng, Y. 2023. SGLang: Efficient Execution of Structured Language Model Programs. *Neural Information Processing Systems*.

Zhuo, T. Y.; Vu, M. C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I. N. B.; Zhan, H.; He, J.; Paul, I.; et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.