ScaleBox: Enabling High-Fidelity and Scalable Code Verification for Large Language Models

Jiasheng Zheng[1,2*], **Xin Zheng**[1*], **Boxi Cao**[1*], **Pengbo Wang**[1,2], **Zhengzhao Ma**[1,2], **Qiming Zhu**[1,2], **Jiazhen Jiang**[1,2], **Yaojie Lu**[1†], **Hongyu Lin**[1†], **Xianpei Han**[1], **Le Sun**[1]
[1]Chinese Information Processing Laboratory
Institute of Software, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
{zhengjiasheng2022,zhengxin2020,caoboxi,luyaojie,hongyu}@iscas.ac.cn
https://github.com/icip-cas/ScaleBox

Abstract

Code sandboxes have emerged as a critical infrastructure for advancing the coding capabilities of large language models, providing verifiable feedback for both RL training and evaluation. However, existing systems fail to provide accurate verification and efficiency under high-concurrency workloads. We present SCALEBOX, a high-fidelity and scalable system designed to address these limitations in large-scale code training. SCALEBOX introduces automated special-judge generation and management, fine-grained parallel execution across test cases with seamless multi-node coordination, and a configuration-driven evaluation suite for reproducible benchmarking. A series of experiments demonstrates that SCALEBOX significantly enhances code verification accuracy and efficiency. Our further RLVR experiments show that SCALEBOX substantially improves both performance on LiveCodeBench and training stability, significantly outperforming heuristic-matching baselines. By providing a reliable and high-throughput infrastructure, SCALEBOX[1] facilitates more effective research and development in large-scale code training.

1 Introduction

Large Language Models (LLMs) have achieved substantial advances in code intelligence (Zhang et al., 2024; Yang et al., 2025b), reshaping code agents and automated software engineering (Dong et al., 2025; Wang et al., 2025a; He et al., 2025; Jiang et al., 2026). A central enabler of this progress is the code sandbox (Guo et al., 2025; Kimi et al., 2025b,a), an essential infrastructure that provides executable ground-truth feedback for both evaluation and Reinforcement Learning from Verifiable Rewards (RLVR). In this paradigm, the accuracy and throughput of sandbox feedback have become key bottlenecks for evaluation reliability and training efficiency.

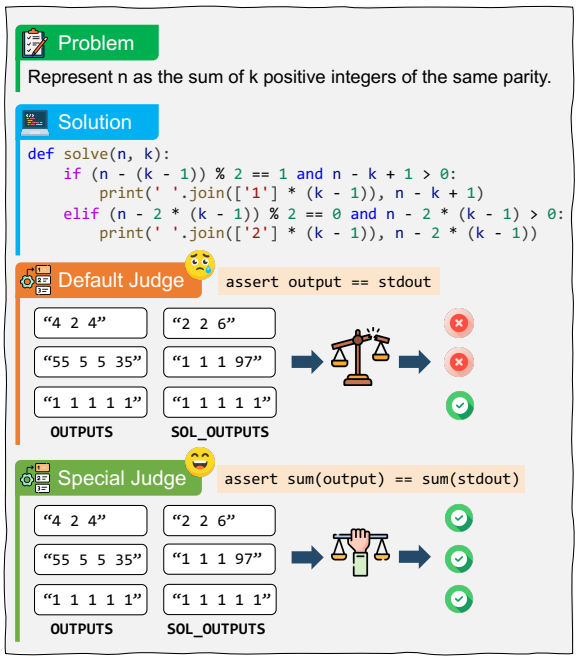


Figure 1: Comparison between Exact Match and Special Judge evaluation.

However, when scaling to large-scale code training, existing sandbox systems exhibit substantial deficiencies in both verification accuracy and efficiency. First, current systems suffer from a **verification accuracy gap** due to their excessive reliance on exact match (EM) or heuristic-based matching evaluation (Cheng et al., 2024; Dou et al., 2025; Cassano et al., 2023). Such mechanisms incorrectly assume a single canonical output, ignoring tasks with multiple valid solutions or precision requirements (Figure 1). Our analysis of 34,757 problems reveals that 14.57% of tasks require specialized verification (Special Judges), and 59.01% of their correct solutions are falsely rejected by EM. This misclassification injects significant noise into learn-

*Equal contribution.
†Corresponding authors.
[1]The demonstration video is available at: https://youtu.be/TGW-qxRFb5s.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 307–317
July 2-7, 2026

ing signals, leading to biased gradients and suboptimal convergence. Second, existing code verification pipelines suffer from a critical **efficiency bottleneck**. Most systems employ static deployment and coarse-grained task scheduling (Cheng et al., 2024; Feng et al., 2025a), hindering horizontal scaling in multi-node clusters. This leads to a severe computational asymmetry: accelerators (e.g., GPUs and NPUs) are heavily utilized while host processors remain underused for parallel code execution and testing, leading to host–accelerator imbalance that limits throughput and scalability in large-scale training.

To this end, we present SCALEBOX, a high-fidelity and scalable sandbox system engineered for large-scale code RLVR training and evaluation. First, SCALEBOX introduces an automated synthesis and management mechanism for special judges, together with a unified verification framework. This design enables adaptive judgment logic, substantially fortifying the accuracy and stability of rewards. Second, SCALEBOX implements hybrid parallelism at both the instance and unit-test levels, along with seamless multi-node deployment. Controlled via a centralized web-based dashboard, the system supports hot-update operations across distributed clusters, as well as real-time resource and log monitoring, thereby fully leveraging idle CPU resources for high-throughput execution. Third, SCALEBOX provides a configuration-driven evaluation suite supporting major code benchmarks, offering the research community a stable, efficient, and reproducible evaluation platform.

A series of experiments demonstrates that SCALEBOX significantly enhances both verification accuracy and efficiency. Accuracy evaluation on AetherCode (Wang et al., 2025b) reveals that our synthetic special judges achieve over 84% per-solution precision, indicating high-fidelity verification. Efficiency evaluation shows that SCALEBOX achieves a 59% speedup over verl's native execution and exhibits substantial throughput scalability across multi-node clusters. To further assess its practical impact, we conduct RLVR training using SCALEBOX as the verification backbone. Experimental results show that the high-fidelity feedback provided by SCALEBOX substantially reduces reward noise, leading to significant improvements in training performance and stability. Specifically, Qwen3-8B (Yang et al., 2025a) trained with SCALEBOX achieves substantial performance gains on the widely used LCB-V5 and LCB-V6 benchmarks (Jain et al., 2025), significantly outperforming heuristic-matching baselines.

Our main contributions are summarized as:

- We propose SCALEBOX, a high-fidelity and scalable system designed to enhance verification accuracy and efficiency for large-scale code training and evaluation.
- We develop a configuration-driven evaluation framework supporting major code benchmarks to streamline assessment workflows.
- Experiments show that SCALEBOX significantly improves verification accuracy and efficiency, leading to better performance and stability in RLVR training.

2 Related Works

2.1 Reinforcement Learning with Verifiable Rewards

Reinforcement Learning from Verifiable Rewards (RLVR) has emerged as a powerful paradigm for enhancing LLM reasoning via objective feedback (Guo et al., 2025). While discussions continue regarding whether RL elicits latent behaviors or expands intrinsic capabilities (Yue et al., 2025), recent empirical evidence suggests that RLVR can synthesize novel strategies surpassing distillation-based limits (Wen et al., 2025; Liu et al., 2025b; Chen et al., 2025b). To ensure stable policy scaling and prevent entropy collapse (Cui et al., 2025b), algorithms like DAPO (Yu et al., 2025), Dr. GRPO (Liu et al., 2025d), and GSPO (Zheng et al., 2025a) introduce diversity-preserving and bias-correction mechanisms. However, the efficacy of these methods is strictly bounded by the fidelity and scalability of the verification signal (Liu et al., 2025c; Chen et al., 2025a). Although tool-augmented (Feng et al., 2025b) and reasoning-augmented approaches (Zheng et al., 2025b) have been proposed, providing high-fidelity judgment for RL training remains a critical frontier.

2.2 Code Execution Environment

The emergence of RLVR in the code domain demands execution environments that provide both high-throughput scaling and high-precision reward generation. However, existing systems remain fragmented. MultiPL-E (Cassano et al., 2023) pioneered large-scale multi-language evaluation, but its offline execution is unsuitable for the iterative feedback loops required in RL. While recent

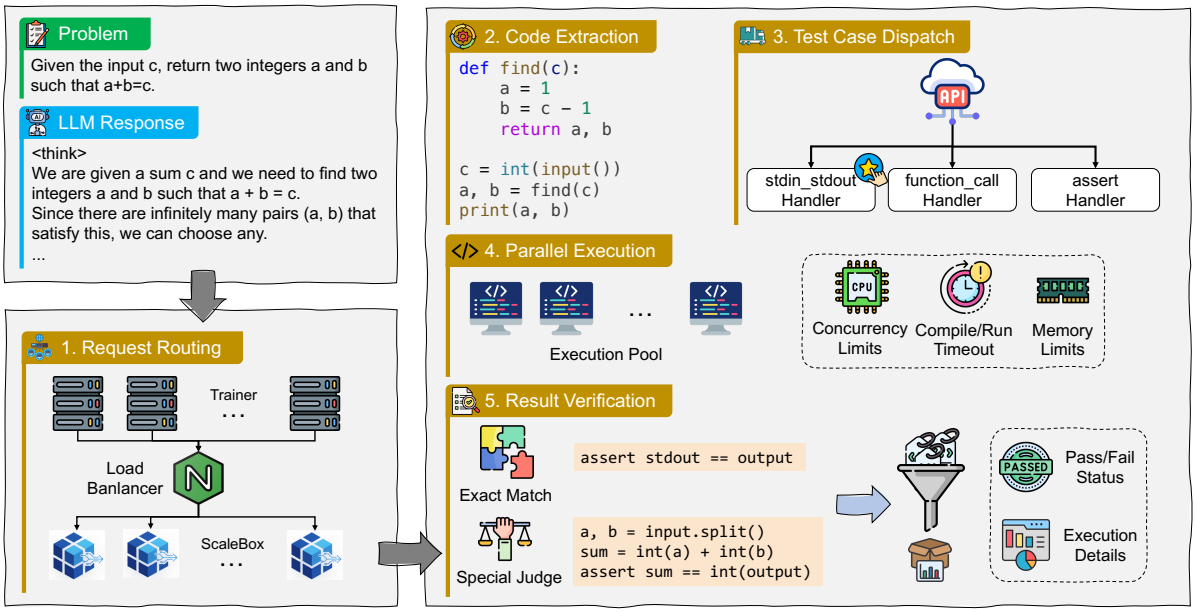


Figure 2: Overview of the SCALEBOX Architecture. A distributed architecture featuring NGINX load balancing, test-case parallelism, and unified verification with special judge support, optimized for high-throughput and reproducible code training and evaluation.

works like SandboxFusion (Cheng et al., 2024) and MPLSandbox (Dou et al., 2025) introduce RL integration for training, they lack either the distributed infrastructure for high-throughput scaling or the specialized judging logic necessary for generating the precise, verifiable rewards that RLVR depends on. Conversely, systems like Judge (Fu et al., 2025) prioritize evaluation accuracy through special judges but lack RL compatibility and scalability. ScaleBox fills this critical gap by providing a scalable sandbox system that simultaneously optimizes for efficiency and accuracy to support the full lifecycle of RLVR.

3 Preliminary Study: The Fragility of Exact Match Evaluation

Current code execution frameworks predominantly rely on Exact Match (EM) of string outputs to verify program correctness (Dou et al., 2025; Cheng et al., 2024). While computationally efficient, EM fails to capture the semantic correctness of tasks where multiple valid representations exist. For instance, problems involving multiple valid outputs (e.g., "output any valid sequence") or floating-point arithmetic (with precision tolerances) are inherently incompatible with string-based evaluation.

To quantify this misalignment, we perform a systematic study of the PrimeIntellect (Mattern et al., 2025) dataset, covering 34,757 Python problems. We use *Qwen3-235B* (Yang et al., 2025a) to determine whether each problem requires special judge evaluation beyond exact string matching (see Figure 5). **Our analysis demonstrates that EM verification is fundamentally ill-suited for a significant portion of code tasks, leading to systematic misclassification of correct solutions.** Specifically, we find that 14.57% of the problems are inherently incompatible with EM evaluation, which requires a special judge to verify correctness, a feature currently absent from most mainstream RL training sandboxes. Among them, 78.26% admit multiple valid outputs and 21.74% require floating-point comparisons with specified numerical tolerances. More critically, we observe that 59.01% of the ground-truth solutions fail EM-based unit tests, despite being logically correct.

These findings indicate that naive string-based evaluation would systematically reject a large fraction of correct solutions for problems requiring special judges. In RLVR, such false negatives introduce noisy reward signals, biasing the learning process and undermining training stability. This motivates the need for a sandbox system with native support for special judge evaluation to ensure accurate verification across diverse programming problem types.

4 ScaleBox System Architecture

SCALEBOX is a distributed, high-concurrency sandbox infrastructure designed for secure code ex-

ecution and evaluation. Built upon the foundation of SandboxFusion (Cheng et al., 2024), SCALEBOX is specifically optimized to meet the rigorous demands of RL training and large-scale code benchmarking. As illustrated in Figure 2, the architecture comprises three synergistic components: (i) **Evaluation Workflow** (§4.1): An optimized end-to-end pipeline for code extraction, execution, and verification. (ii) **Distributed Deployment** (§4.2): A scalable architecture designed for high-throughput RL training. (iii) **Special Judge Evaluation** (§4.3): A flexible verification framework for non-deterministic programming tasks.

4.1 Evaluation Workflow

The SCALEBOX evaluation pipeline is designed to minimize latency while ensuring execution fidelity. The workflow follows a five-stage process:

1. **Request Routing**: Incoming evaluation tasks are managed by an NGINX-based load balancer, which dynamically routes requests to available sandbox workers.

2. **Code Extraction**: The worker node uses robust heuristics to extract executable code from diverse LLM formats, including Markdown blocks, code fragments, and scripts.

3. **Test Case Dispatch**: Utilizing the unified API, the system identifies the test type (`stdin_stdout`, `function_call`, or `assert`) and generates language-specific test harnesses.

4. **Parallel Execution**: Test cases are executed in parallel within isolated environments, with configurable constraints on memory, CPU usage, and multi-level wall-clock time. The system maps time constraints to different lifecycle stages, covering individual compilation, per-test execution, and global session timeouts.

5. **Multi-Stage Verification**: Prioritizes optimized *exact match* (handling whitespace and float tolerance), falling back to a *special judge* for custom verification if required.

To facilitate standardized RLVR research and rapid iteration, SCALEBOX further provides a one-click evaluation workflow for widely used code benchmarks, such as LiveCodeBench (Jain et al., 2025), HumanEval (Chen et al., 2021), and AetherCode (Wang et al., 2025b).

Method	Nodes	Time (s)	Throuput (tasks/s)
verl Prime	1	331.25	24.73 (1.00×)
SandboxFusion	1	548.93	14.92 (0.56×)
SCALEBOX	1	208.38	39.31 (1.59×)
SCALEBOX	2	163.40	50.13 (2.03×)
SCALEBOX	3	131.92	62.10 (2.51×)

Table 1: Efficiency comparison of SCALEBOX against widely-adopted strong baselines.

4.2 Distributed Deployment

To accommodate the bursty and high-volume workloads characteristic of code RL, SCALEBOX implements a horizontally scalable architecture.

Load Balancing and Availability An NGINX-based ingress controller implements a round-robin strategy to distribute workloads. To ensure robustness during long-running training jobs, we utilize Docker-based health checks and automated recovery protocols, maintaining high availability even under extreme hardware utilization.

Hybrid Parallelism Unlike existing systems that parallelize only at the instance level, SCALEBOX introduces *hybrid parallelism* at both the instance and unit-test levels. Each worker node supports configurable multi-tenancy, processing multiple instances concurrently while executing individual test cases within a single instance in parallel.

Web-based Dashboard SCALEBOX provides a web-based dashboard (Appendix A) for visualized distributed deployment, resource monitoring, and log inspection. Through this interface, we can perform hot updates across multi-node clusters, dynamically reallocating and utilizing idle CPU resources to achieve efficient and scalable execution.

Efficiency Benchmarking We evaluate SCALEBOX using 8,192 Python problems from the PrimeIntellect (Mattern et al., 2025) dataset on Intel Xeon 6700P Series CPUs (64 cores per node). Our results (Table 1) show that SCALEBOX achieves a single-node throughput of 39.31 tasks/s, representing a 1.59× improvement over verl (Sheng et al., 2025) framework (24.73 tasks/s) with Prime (Cui et al., 2025a) code and 2.63× over SandboxFusion (Cheng et al., 2024) baseline (14.92 tasks/s). This throughput enhancement reflects the architectural advantages of SCALEBOX, particularly its integration of hybrid parallelism and batch processing, which are specifically engineered to mitigate the serial execution bottlenecks prevalent in existing sandboxes. Furthermore, SCALEBOX scales to

62.10 tasks/s on 3 nodes, demonstrating substantial scalability for large-scale RL clusters.

4.3 Special Judge Evaluation

Standard exact-match verification often yields false negatives in programming tasks with multiple valid solutions, such as those involving non-deterministic algorithms, multiple optimal paths, or floating-point precision (Chou et al., 2025; Wang et al., 2025b). SCALEBOX addresses this via a generalized special judge framework.

For stdin/stdout tasks, the special judge acts as a programmable verifier that reads three files: (1) `stdin.txt` (the input), (2) `stdout.txt` (the reference output), and (3) `answer.txt` (the participant output). The judge program returns a binary verdict based on task-specific logic.

To optimize performance, SCALEBOX employs a **Short-Circuit Verification** strategy: the computationally expensive special judge is only invoked if the participant output fails the initial exact-match heuristic. This two-stage approach ensures both rigorous correctness and high execution efficiency.

5 Automated Special Judge Synthesis

In this section, we present an automated pipeline to synthesize *special judges*, mitigating reward noise from exact matching. We first detail the synthesis process and then validate judge fidelity to establish a reliable foundation for downstream RL.

5.1 Synthesis Pipeline

SCALEBOX transforms rigid exact-match signals into functional verification logic through a three-stage pipeline: (1) problem taxonomy detection, (2) program synthesis, and (3) sandbox-based verification. The implementation details can be found in Appendix D.

Problem Taxonomy Detection We use an LLM to classify tasks based on their problem descriptions. The LLM identifies tasks where exact-match is insufficient and maps them into a predefined taxonomy: (1) **non-deterministic outputs**, which allow multiple valid trajectories (e.g., any valid path in a DAG), or (2) **numerical tolerance**, which require precision-based comparisons within a specific threshold ϵ (e.g., "absolute error $< 10^{-6}$").

Program Synthesis For identified problems, we use an LLM to synthesize a Python-based judge program `validate_solution`. This judge operates by reading the problem input (`stdin.txt`), reference output (`stdout.txt`), and participant output (`answer.txt`) to produce a final binary verdict based on problem-specific constraints.

Sandbox-based Verification Loop To ensure reliability, synthesized judges undergo a pre-deployment validation stage within the SCALEBOX environment. This involves a fidelity test to ensure the judge accepts ground-truth solutions and a robustness test to ensure it rejects null or known-incorrect outputs. This dual-verification filter discards malformed judges and triggers iterative LLM-based regeneration upon failure.

Model	TPR (%)	TNR (%)
GPT-5.2 (Singh et al., 2025)	96.3	86.5
Claude-3.7-Sonnet (Anthropic, 2025)	96.3	88.5
DeepSeek-V3.2 (Liu et al., 2025a)	90.0	86.5
Qwen3-235B (Yang et al., 2025a)	90.3	84.2

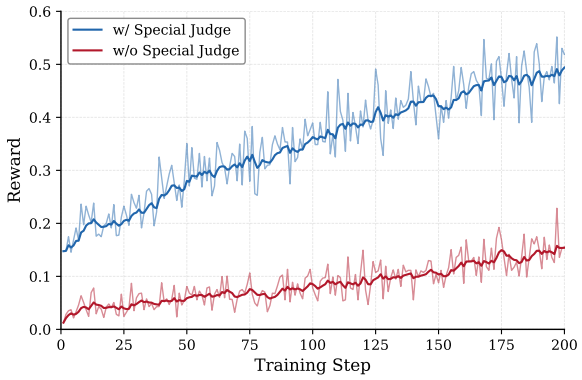
Table 2: Fidelity evaluation of synthesized special judges on 27 AetherCode instances. TPR and TNR are averaged across all submissions.

5.2 Evaluation of Judge Fidelity

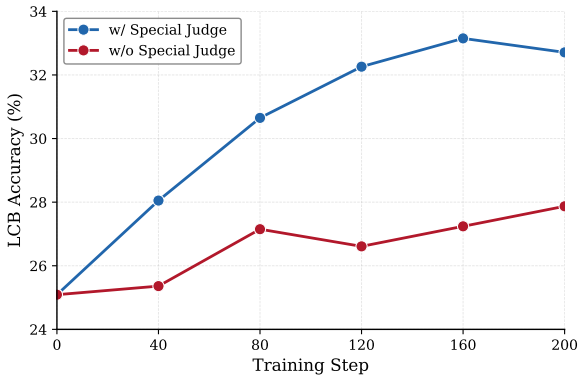
To assess the reliability of synthesized special judges, we conduct a systematic evaluation using 27 complex problems from the AetherCode dataset (Wang et al., 2025b) that require non-trivial verification logic. For each problem, we sample special judge codes up to 20 attempts. We then evaluate the valid sampled special judges against human-authored gold-standard oracles on a total of 269 correct and 260 incorrect submissions, focusing on two critical metrics:

- True Positive Rate (TPR): The proportion of correct solutions that are correctly accepted by the generated judge.
- True Negative Rate (TNR): The proportion of incorrect solutions that are correctly rejected by the generated judge.

As shown in Table 2, **our synthesis method consistently generates highly effective verification logic across all evaluated LLMs**. Every model achieves a TPR $\geq 90.0\%$ and a TNR $\geq 84.0\%$, with Claude-3.7-Sonnet reaching the highest sensitivity (96.3% TPR) and providing the strongest filtering (88.5% TNR). These results demonstrate that synthesized judges can reliably distinguish between correct and incorrect code even for elite competitive programming tasks, validating the efficiency and high fidelity of our approach.



(a) Training reward trajectories.



(b) Pass@1 Accuracy (LCB-v5).

Figure 3: Impact of reward fidelity on RL training (1.2K Subset): (a) Standard exact-match rewards are artificially suppressed due to false negatives, whereas Special Judges (SPJ) provide a more accurate and dense signal. (b) This higher-fidelity reward translates directly into superior and more stable Pass@1 performance across training steps.

6 RL Experiments

Building on SCALEBOX, we leverage RL to demonstrate that high-fidelity synthesized judges drive sustained gains by accurately verifying syntactically diverse programs where exact matching fails.

6.1 Experimental Setup

Our training data consists of 26K Python problems filtered from the PrimeIntellect (Mattern et al., 2025) ($\geq$ 5 unit tests). Within this set, our pipeline (§5.1) using *DeepSeek-V3.2* (Liu et al., 2025a) identifies 2.8K problems requiring Special Judges (SPJ). Notably, for 1.2K of these, reference solutions failed standard exact-match tests but are validated by our synthesized judges. For RL setup, based on *Qwen3-8B* (Non-thinking) (Yang et al., 2025a), we perform the GRPO (Shao et al., 2024) algorithm using the verl (Sheng et al., 2025) framework. Further details are provided in Appendix C.

To investigate the impact of special judges, we evaluate training performance across two scales: (1) the 1.2K SPJ subset where special judges are critical, and (2) the full 26K dataset which mixes standard and 2.8K SPJ problems. Performance is measured by Pass@1 on LiveCodeBench (Jain et al., 2025) (v5 and v6 subsets).

6.2 Results and Analysis

Table 3 presents the comparative performance of RL training across the 1.2K special judge subset and the full 26K dataset.

Synthesized special judges significantly improve RL performance by providing high-fidelity reward signals. On the 1.2K special judge subset, SPJ-enhanced training achieves +5.91% on LCB-v5. As illustrated in Figure 3a, this gain is directly attributable to escaping the exact-match trap, where valid programs were previously mislabeled as failures. By resolving these false negatives, our method ensures that the policy is accurately rewarded for legitimate exploration.

Training Set	Reward Type	LCB-v5	LCB-v6
(Base Model)	–	25.09	27.21
1.2K Subset	w/o SPJ (EM)	27.24	27.94
	w/ SPJ (Ours)	**33.15**	**32.35**
26K Full Dataset	w/o SPJ (EM)	37.19	34.12
	w/ SPJ (Ours)	**38.17**	**36.03**

Table 3: Comparative performance of Pass@1 (%) on LiveCodeBench using Qwen3-8B as the base policy. Results show that incorporating Special Judges (SPJ) consistently improves performance across both datasets, demonstrating the critical role of reward fidelity.

High-fidelity rewards enhance training stability and convergence. Figure 3b demonstrates that the SPJ-enhanced model consistently maintains a performance lead throughout the training steps. This gap suggests that cleaner reward signals reduce variance in credit assignment, allowing the model to converge more efficiently on robust coding patterns.

The benefits of special-judge verification generalize to large-scale datasets. Even on the full 26K dataset, where SPJ cases account for only $\sim$ 10% of the total, we observe +1.91% on LCB-v6. This disproportionate impact suggests that accurate evaluation of complex, non-deterministic problems provides a high-quality training signal that positively transfers to the model's overall reasoning capabilities, even in standard coding tasks.

7 Conclusion

In this paper, we introduce SCALEBOX, a robust and scalable sandbox system that enhances reward accuracy and verification efficiency in large-scale code training. By leveraging automated special judge synthesis, unified verification, and parallel multi-node execution, SCALEBOX mitigates the reward noise and host–accelerator imbalance issues present in existing systems. Our experiments confirm that SCALEBOX's high-fidelity rewards and efficient infrastructure yield substantial gains in model performance and training stability, providing a robust foundation for scaling training in LLMs.

Limitations

Although SCALEBOX is architecturally designed with configurations to support high scalability, its current evaluation is primarily focused on specific model scales and benchmarks. In future work, we plan to conduct larger-scale and multilingual RL experiments to fully verify the effectiveness and robustness of SCALEBOX in more diverse and extensive code training scenarios. Additionally, while the current work focuses on providing feedback for code training and evaluation, we will explore applying SCALEBOX to broader code agent scenarios, such as automated software engineering and multi-turn autonomous problem-solving, to further demonstrate its utility as a foundational infrastructure.

Acknowledgements

We sincerely thank the reviewers for their insightful comments and valuable suggestions. This work was supported by the National Key R&D Program of China (2024YFC3308000), Beijing Natural Science Foundation (L243006), the Natural Science Foundation of China (No. 62306303, 62476265). The authors would like to thank Huawei Ascend Cloud Ecological Development Project for the support of Ascend 910 processors.

References

Anthropic. 2025. Claude 3.7 sonnet system card.

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

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Trans. Software Eng.*, 49(7):3675–3691.

Ding Chen, Qingchen Yu, Pengyuan Wang, Mengting Hu, Wentao Zhang, Zhengren Wang, Bo Tang, Feiyu Xiong, Xinchi Li, Chao Wang, and 1 others. 2025a. xverify: Efficient answer verifier for reasoning model evaluations. *arXiv preprint arXiv:2504.10481*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2025b. Acereason-nemotron: Advancing math and code reasoning through reinforcement learning. *arXiv preprint arXiv:2505.16400*.

Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, and 1 others. 2024. Fullstack bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*.

Jason Chou, Ao Liu, Yuchi Deng, Zhiying Zeng, Tao Zhang, Haotian Zhu, Jianwei Cai, Yue Mao, Chenchen Zhang, Lingyun Tan, and 1 others. 2025. Autocodebench: Large language models are automatic code benchmark generators. *arXiv preprint arXiv:2508.09101*.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Yuchen Zhang, Jiacheng Chen, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, and 1 others. 2025a. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, and 1 others. 2025b. The entropy mechanism of reinforcement learning for reasoning language models. *arXiv preprint arXiv:2505.22617*.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*.

Shihan Dou, Jiazheng Zhang, Jianxiang Zang, Yunbo Tao, Weikang Zhou, Haoxiang Jia, Shichun Liu, Yuming Yang, Shenxi Wu, Zhiheng Xi, Muling Wu, Rui Zheng, Changze Lv, Limao Xiong, Shaoqing Zhang, Lin Zhang, Wenyu Zhan, Rongxiang Weng, Jingang Wang, and 8 others. 2025. Multi-programming language sandbox for LLMs. In *Proceedings of the*

63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 40–50, Vienna, Austria. Association for Computational Linguistics.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025a. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.

Ruixiang Feng, Zhenwei An, Yuntao Wen, Ran Le, Yiming Jia, Chen Yang, Zongchao Chen, Lisi Chen, Shen Gao, Shuo Shang, and 1 others. 2025b. Cosineverifier: Tool-augmented answer verification for computation-oriented scientific questions. *arXiv preprint arXiv:2512.01224*.

Jia Fu, Xinyu Yang, Hongzhi Zhang, Yahui Liu, Jingyuan Zhang, Qi Wang, Fuzheng Zhang, and Guorui Zhou. 2025. Klear-codetest: Scalable test case generation for code reinforcement learning. *arXiv preprint arXiv:2508.05710*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Junda He, Christoph Treude, and David Lo. 2025. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5):124:1–124:30.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

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

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sung Hun Kim. 2026. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, 35(2):58:1–58:72.

Jur1cek. 2022. Codeforces Dataset.

Kimi, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025a. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Kimi, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, and 1 others. 2025b. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*.

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

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

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. 2025b. ProRL: Prolonged reinforcement learning expands reasoning boundaries in large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Shudong Liu, Hongwei Liu, Junnan Liu, Linchen Xiao, Songyang Gao, Chengqi Lyu, Yuzhe Gu, Wenwei Zhang, Derek F. Wong, Songyang Zhang, and Kai Chen. 2025c. CompassVerifier: A unified and robust verifier for LLMs evaluation and outcome reward. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33454–33482, Suzhou, China. Association for Computational Linguistics.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. 2025d. Understanding r1-zero-like training: A critical perspective. In *Second Conference on Language Modeling*.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 1279–1297. ACM.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, and 1 others. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*.

Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. 2025a. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*.

Zihan Wang, Jiaze Chen, Zhicheng Liu, Markus Mak, Yidi Du, Geonsik Moon, Luoqi Xu, Aaron Tua, Kunshuo Peng, Jiayi Lu, and 1 others. 2025b. Aethercode: Evaluating llms' ability to win in premier programming competitions. *arXiv preprint arXiv:2508.16402*.

Xumeng Wen, Zihan Liu, Shun Zheng, Shengyu Ye, Zhirong Wu, Yang Wang, Zhijian Xu, Xiao Liang, Junjie Li, Ziming Miao, and 1 others. 2025. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base llms. *arXiv preprint arXiv:2506.14245*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and Julian J. McAuley. 2025b. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 2586–2616. Association for Computational Linguistics.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, and 17 others. 2025. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the perspectives of NLP and software engineering: A survey on language models for code. *Transactions on Machine Learning Research*.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, and 1 others. 2025a. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*.

Shenghe Zheng, Chenyu Huang, Fangchen Yu, Junchi Yao, Jingqi Ye, Tao Chen, Yun Luo, Ning Ding, Lei Bai, Ganqu Cui, and 1 others. 2025b. Sci-verifier: Scientific verifier with thinking. *arXiv preprint arXiv:2509.24285*.

A Screenshots of ScaleBox Dashboard

Figure 4 shows the screenshots of SCALEBOX dashboard, which includes features for distributed deployment, resource monitoring, and log monitoring.

B Supported Benchmark of ScaleBox

SCALEBOX supports diverse code benchmarks, including:

- **Assert-based**: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), HumanEval+, and MBPP+ (Liu et al., 2023)
- **Multi-language**: MultiPL-E (Cassano et al., 2023)
- **Standard I/O and function call**: LiveCodeBench (Jain et al., 2025),
- **Special Judge**: Aethercode (Wang et al., 2025b)

Table 4 presents evaluation results on standard benchmarks using the one-click evaluation workflow of SCALEBOX.

C Training Details of RLVR

For training data, we use PrimeIntellect/verifiable-coding-problems (Mattern et al., 2025) dataset as source, which curated from APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), Codeforces (Jur1cek, 2022), and TACO (Li et al., 2023). We then select 26K Python problems that contain non-empty reference solutions and ≥ 5 test cases.

For RL setup, based on *Qwen3-8B* (Non-thinking) (Yang et al., 2025a), we perform the GRPO (Shao et al., 2024) algorithm using the verl (Sheng et al., 2025) framework, with a global batch size of 128, a mini batch size of 32, 8 rollouts per question, learning rate of 1e-6, max response length of 8192, ϵ_{low} of 0.2 and ϵ_{high} of 0.28.

For evaluation, we use the widely adopted v5 (2408–2502) and v6 (2501–2504) subsets of LiveCodeBench (Jain et al., 2025), consisting of 279 and 170 problems, respectively. We sample 4 times per problem, with max output length of 32768, temperature of 0.6, and top_p of 0.95.

D More Details on Special Judge Synthesis

The automatic special judge synthesis pipeline (§5.1) uses two prompts. The classification prompt (Figure 5) detects problems requiring special judges by identifying (1) multiple valid outputs and (2) floating-point comparisons with tolerance constraints, and returns structured JSON with confidence scores for filtering. The generation prompt (Figure 6) provides an example special judge program and instructs the LLM to produce a standardized program that reads `stdin.txt`, `stdout.txt`, and `answer.txt`, and outputs correctness via exit codes. This few-shot scheme yields reliable verification programs across problem types.

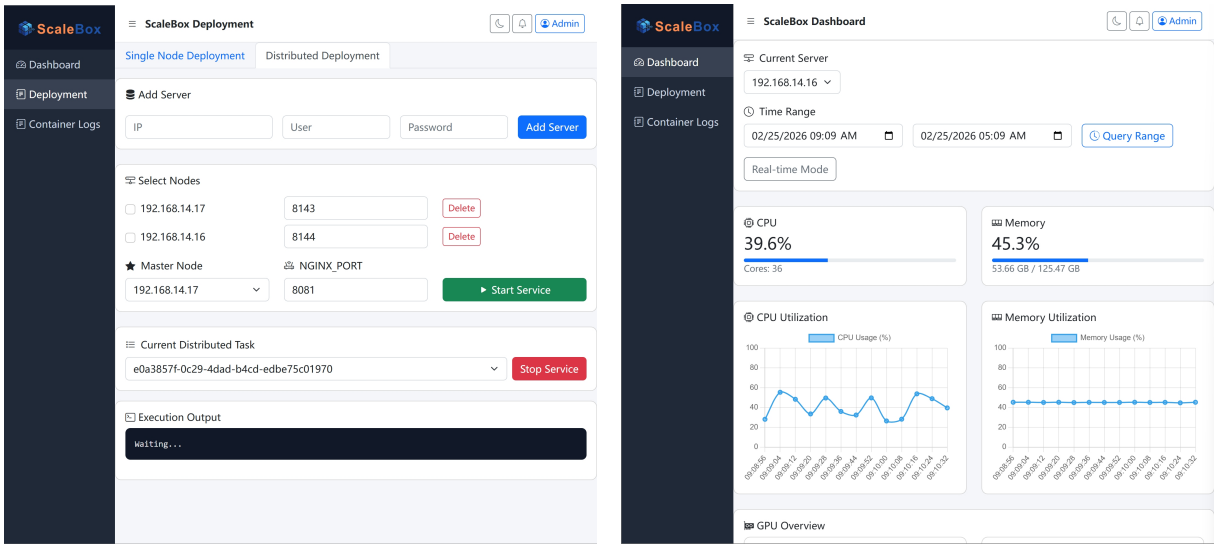


Figure 4: The screenshots of the SCALEBOX dashboard. **Left:** Distributed Deployment Panel. **Right:** Resource Monitoring Panel.

Model	HumanEval	MBPP	HumanEval+	MBPP+	LiveCodeBench	AetherCode
Llama-3-8B-Instruct	60.98	62.76	57.93	54.76	10.48	0.20
Llama-3.1-8B-Instruct	70.73	66.74	65.24	57.67	6.18	0.20
DeepSeek-R1-Distill-Qwen-1.5B	47.56	40.28	44.51	37.30	16.13	0.07
Qwen3-4B	89.63	82.67	85.37	73.81	53.92	8.07
Qwen3-8B	88.41	85.95	80.48	73.28	60.09	9.18

Table 4: Pass@1 accuracy (%) on HumanEval, MBPP, HumanEval+, MBPP+, LiveCodeBench, and AetherCode.

```
You are an assistant that classifies programming / algorithm / data processing tasks regarding SPECIAL JUDGE need.

Decide for the given task text:
1. multiple_solutions? For example:
 - If there are multiple answers, print any of them
 - If there are multiple solutions, you are allowed to print any of them.
 - If there are multiple possible solutions, print any of them.
 - If there are different possible orders with a correct answer, print any of them.
 - If there are multiple solutions, satisfying the problem condition(s), you can print any "one" solution.

2. float_comparison? (floating point answers, precision, tolerance, absolute/relative error, decimals)

Return JSON object:
{
 "reason": "<short justification <less than 160 words>",
 "needs_special_judge": <true|false>,
 "categories": [ zero or more of "multiple_solutions","float_comparison" ],
 "confidence": <float 0..1>
}

Rules:
 - needs_special_judge is true iff categories non-empty.
 - confidence: 0.9 clear indicators, 0.6 somewhat, 0.3 guess.
 - Keep output STRICT JSON. No extra keys.
```

Figure 5: Prompt Template for classifying programming problems that require special judge support.

````
Here is an example of task description and a special judge program in Python.
<problem>
{PROBLEM}
</problem>

<special_judge_program>
{SPECIAL_JUDGE}
</special_judge_program>

Given the stdin, stdout, answer and programming task, write the JUDGE python program to check if the answer is as
      valid as the stdout. You may want to leverage stdout to save computation when needed.

```python
import sys

def read_file(filepath):
    with open(filepath, 'r') as f:
        return f.read().strip().split('\n')

def validate_solution(stdin_path, stdout_path, answer_path):
    stdin_lines = read_file(stdin_path)
    stdout_lines = read_file(stdout_path)
    participant_output = read_file(answer_path)

    if participant_output == [''] and stdout_lines != ['']:
        return False
    # ...

stdin_path = "stdin.txt"
stdout_path = "stdout.txt"
answer_path = "answer.txt"

is_valid = validate_solution(stdin_path, stdout_path, answer_path)

if is_valid:
    sys.exit(0)
else:
    sys.exit(1)
```
````

Figure 6: Prompt Template for generating special judge programs.