CoSineVerifier: Agentic Verification for Computation-Oriented Scientific Questions

Anonymous ACL submission

Abstract

Answer verification methods are widely employed in language model training pipelines spanning data curation, evaluation, and reinforcement learning with verifiable rewards (RLVR). While prior work focuses on treating answer verification as a general reasoning or semantic matching problem, significant challenges remain in computation-oriented scientific domains. In this paper, we introduce CoSineVerifier, an agentic verifier that actively invokes external executors to ground its judgments in precise computations and symbolic reasoning. To train this accurate agentic verifier, we propose a novel verifier training data augmentation method to increase the correctness and efficiency of tool-invoked verifications. Extensive experiments across STEM, QA, and long-form reasoning tasks demonstrate CoSineVerifier's robust generalization abilities, achieving state-of-the-art performance on VerifyBench-Hard and SCI-Bench. Furthermore, when employed as an RLVR reward model, CoSineVerifier consistently outperforms both rubric- and model-based verifiers on AIME'24, AIME'25 and GPQA-D, highlighting its potential to advance LLM reasoning. Our code and model are available in our anonymous repository.

1 Introduction

The training methods for large language models (LLMs) have gradually shifted from using human-annotated data to model self-evolution approaches, which also imposes higher demands on how to evaluate the quality of generated content. In this context, answer verification, which compares a model output against a reference, has become a critical component throughout data curation, evaluation, and model training pipelines. In data curation, answer verification methods are employed to guarantee the consistency, fidelity, and accuracy of synthetic corpora (He et al., 2025b; Yuan et al., 2025; Toshniwal et al., 2024; Albalak et al., 2025). In evaluation, these methods provide a robust alternative to exact match (EM), enabling semantic-level comparison (Ma et al., 2025a; Li et al., 2024; Paech, 2023; Wu et al., 2025b). In model training, Reinforcement Learning with Verifiable Rewards (RLVR) relies on answer verifiers to produce scalable reward signals (He et al., 2025a; Guha et al., 2025; Guo et al., 2025).

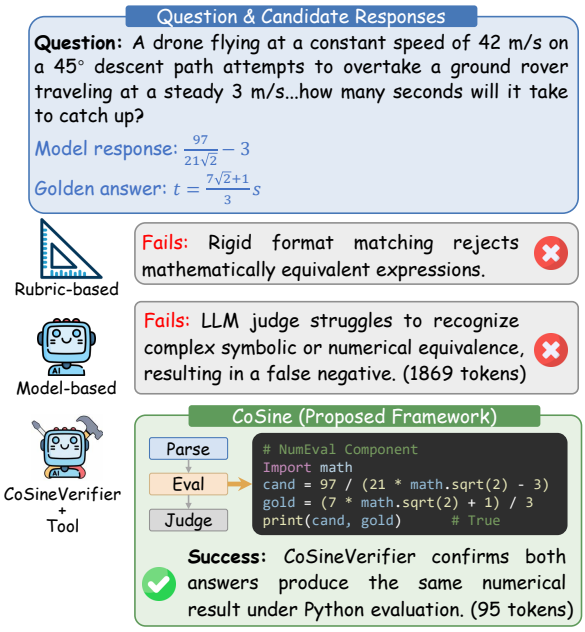

Figure 1: Comparison between CoSineVerifier and existing verifiers, where CoSineVerifier enables accurate judgment on calculation-intensive scenarios.

To this end, substantial efforts have been devoted to building robust verifiers, which can be broadly categorized into two paradigms: rubric-based verifiers and model-based verifiers. Rubric-based verifiers (Kydlíček), which depend on structured outputs (e.g., Math-Verify evaluating equivalence in Markdown), usually fail even on simple cases such as comparing *2*2* and *4*. Alternatively, recent work like Xverify (Chen et al., 2025a) and CompassVer-

ifier (Liu et al., 2025a) employs LLM-based verifiers, which judge answer correctness through pure natural-language reasoning and semantic comparison (Chen et al., 2025b). However, as illustrated in Figure 1, both paradigms remain insufficient for computation-oriented scientific questions. When verification involves non-trivial numerical computation, symbolic manipulation, or scientific units, rubric-based verifiers often fail because they lack the ability to execute and compare the underlying computations. Meanwhile, model-based verifiers may produce false negatives even after lengthy reasoning, as they can struggle to recognize complex symbolic or numerical equivalence. These limitations suggest that relying solely on surface-form matching or standard LLM reasoning is inadequate for efficient and reliable verification in calculation-intensive scientific scenarios. As a result, the inability to accurately verify computation-oriented answers may further constrain the quality of synthetic data, the reliability of model evaluation, and the effectiveness of RLVR training signals.

In this paper, we present CoSineVerifier, an agentic LLM-based verifier for general reference-based verification, with particular strengths in computation-oriented scientific scenarios. CoSineVerifier proactively invokes external tools, such as a Python interpreter, to perform precise computations on both formulas to derive aligned numerical results, thereby ensuring accurate verification.

To further diagnose the limitations of existing verifiers and motivate the need for external tool use, we construct a diverse set of verifiable question-answer pairs from standard benchmarks covering STEM, general QA, and long-form reasoning. We evaluate representative verifiers under a pass@3 setting and use a strong LLM(e.g., GPT-5) to categorize failures into 15 error types. As shown in Figure 2, the most frequent failure is calculation inaccuracy, accounting for 25.1% of all errors. This suggests that verifiers relying solely on standard reasoning often struggle to validate computation-heavy questions and formally complex answers. These findings highlight the necessity of introducing external execution tools for challenging, calculation-intensive verification. Meanwhile, verifier errors exhibit a long-tail distribution due to the heterogeneous outputs of reasoning models. Thus, focusing only on dominant errors may overlook other fine-grained failure modes and bias the assessment of verifier reliability.

To address these limitations, we propose a long-tail-aware and difficulty-oriented data augmentation strategy to generate challenging verification instances. In the long-tail augmentation stage, In the long-tail augmentation stage, we expand lower-frequency verification failures by constructing semantically similar variants, thereby increasing their coverage. This prevents the verifier from being biased toward dominant error patterns(e.g. Calucation Error) caused by the imbalanced failure distribution, enabling it to verify a broader and more diverse range of heterogeneous LLM responses. Meanwhile, to further exploit limited high-difficulty data, we introduce a self-iterative augmentation process. Starting from a small set of seed instances, we repeatedly synthesize candidate computation-heavy verification problems and filter them with the current verifier, retaining those that the verifier fails to solve. By continuously mining verifier-specific hard cases, this process strengthens the model on error-prone verification scenarios and improves its robustness to complex numerical and symbolic reasoning.

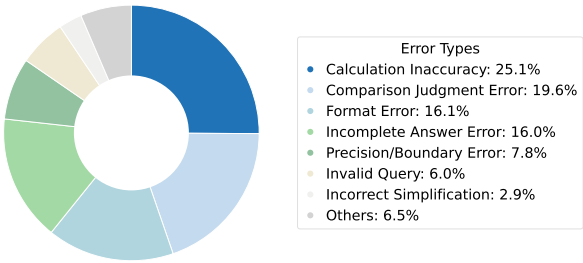

Figure 2: Error type distribution of verifiers. Detailed explanation and types can be found in 7.1

Finally, we employ a two-stage training strategy to equip the verifier with effective tool-use capabilities in computation-heavy scenarios: (i) cold-start supervised fine-tuning, followed by (ii) reinforcement learning with tool-call encouragement. Extensive experiments conducted on two verifier benchmarks demonstrate that our proposed CoSineVerifier outperforms the state-of-the-art verifiers. To validate the effectiveness of CoSineVerifier when using it as a reward model in downstream tasks, we compare the performance of LLMs trained using CoSineVerifier and other verifiers on AIME24 and AIME25 benchmarks. And the results show that the LLM using CoSineVerifier as the reward model achieves the best performance compared to the current state-of-the-art verifier.

To summarize, our contributions are threefold:

- We identify common verification failures and

propose **CoSineVerifier**, an agentic verification model designed for precise general reference-based answer verification, with particular strengths in computation-heavy scientific scenarios.

• We propose a novel verifier training data augmentation method that balances long-tail error coverage and synthesizes complex verification problems for robust and efficient agentic verification.

• Experiments demonstrate that CoSineVerifier achieves state-of-the-art performances on both verification tasks and downstream RLVR applications.

2 Related Works

2.1 Answer Verification

Answer verification is the automated process of deciding whether a candidate response is factually correct to the provided reference answer. Significant progress has been made in the development of LLM-output verification (Liu et al., 2025b; Xia et al., 2025), which can broadly be divided into two main classes: rule-based verification and model-based verification. Early rule-based methods like Math-Verify (Kydlíček) extract the final answer via heuristics and compare it against the ground truth using predefined rules. However, these approaches struggle with unstructured model outputs and tend to incur a high false-negative rate (Huang et al., 2025). To address these shortcomings, model-based verifiers exploit the flexibility of LLMs to interpret and evaluate unpredictable output. Xverify (Chen et al., 2025a) and CompasVerifier (Liu et al., 2025a) are designed to provide efficient and robust answer verification by carefully constructing training examples. While effective, they are constrained by the verifier's inherent reasoning and calculation competency—particularly in computation-intensive math and unit-sensitive STEM problems.

2.2 Agentic Reasoning

A growing line of work investigates improving LLM reasoning by augmenting models with external tools, yielding consistent gains in accuracy, calibration, and sample efficiency across math, program, and multi-turn QA tasks (Mai et al., 2025; Chen et al., 2025c; Dong et al., 2025; Zhang et al., 2025; Ma et al., 2025b; Lin and Xu, 2025). ReTool (Feng et al., 2025), ToRL (Li et al., 2025), and SimpleTIR (Xue et al., 2025) train models to interleave real-time code execution with language reasoning and to learn when/how to call tools via reward signals, yielding strong gains on math-reasoning benchmarks. VTool-R1 (Wu et al., 2025a) integrates Python-based visual editing tools into the RFT loop so the model learns to interleave intermediate visual steps with textual chain-of-thoughts. Beyond code, Search-R1 (Jin et al., 2025) casts web search as an action space and trains query generation/revision via outcome rewards. Taken together, these approaches recast reasoning over a unified tool-action space, and reflect, yielding stronger generalization and reliability than standard training beyond base models.

3 Method: CoSineVerifier

In this section, as shown in Figure 3, we outline the workflow and training procedure of CoSineVerifier, which is endowed with agentic tool-use capabilities for accurate outcome verification.

3.1 Agentic Verification

To facilitate the reliable outcome verification, we equip our verification model with external tools and dynamically execute them, enabling precise judgments on complex verification tasks. Specifically, as shown in Figure 3 (a), when facing a verification problem, our CoSineVerifier first decides whether to use an external tool. If an external tool call is needed, CoSineVerifier first selects the tool and parses the tool arguments (e.g., Python code), and then sends them to sandbox environments. After the sandbox environment completes the execution and returns the output, CoSineVerifier gives its final judgment based on the sandbox output.

3.2 Data Curation

Agentic answer verification presents a dilemma regarding model selection and tool use: large-scale LLMs (e.g., >100B parameters) are too inefficient for the pipeline, while smaller LLMs often lack the proficiency required for complex tool-augmented verification, frequently falling into repetitive tool-call loops without effectively verifying the answer(Section 5.1). To address this, we construct a diverse dataset designed to support efficient and robust agentic verification.

Seed Data Collection Answer verification in real-world applications is challenging for two reasons: (i) the diversity of questions encountered in practice, and (ii) the variability of model outputs (format, length, and reasoning style). To address these challenges, we construct a large-scale

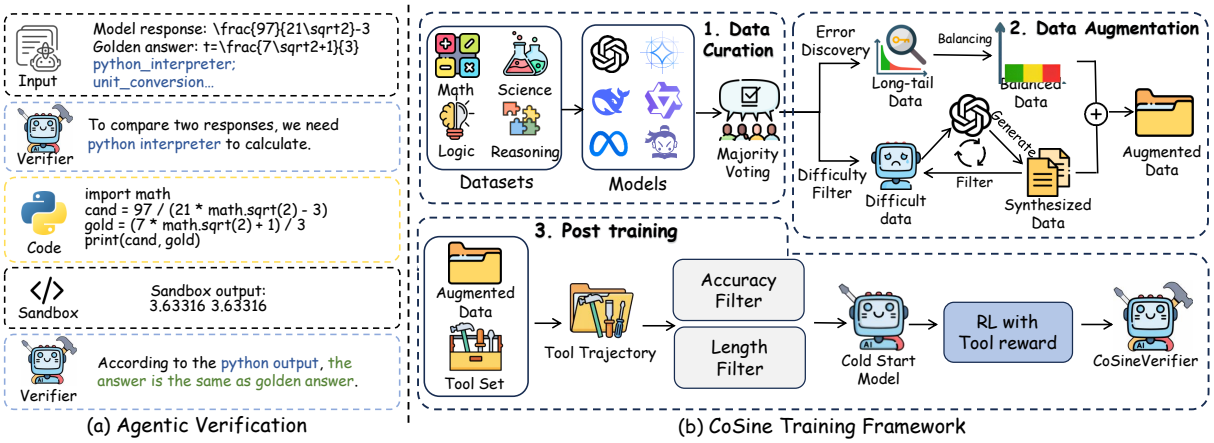

Figure 3: (a) Workflow of agentic verification. (b) The overall training framework of CoSineVerifier.

candidate training QA pairs, comprising 1.14 million samples. These samples were collected from 15 datasets using 14 LLMs, spanning mathematics, science, logical reasoning, and commonsense knowledge domains. This provides broad coverage for training robust verification models. It is notable that a substantial portion of our data is sampled from competition-level STEM questions, where verification often fails in practice. To further increase robustness, we also prepend different prompt prefixes (e.g., Let's think step by step) on a subset of training data, promoting generalization to diverse prompting styles.

Data Annotation After constructing a large and diverse dataset, we use a multi-turn annotation pipeline to obtain reliable verification labels. We first employ a set of models (Compass-Verifier-32B, qwen3-4b-2507, and Qwen3-32B) for annotaion and aggregate their predictions via majority voting. We then retain 63,714 samples that show any disagreement among these annotations. We further send these cases to a more powerful model (GPT-o3) with majority voting, ensuring high label fidelity on harder questions.

3.3 Data Augmentation

3.3.1 Long-tail data augmentation

As shown in Figure 2, all error types reveal a pronounced skew: the top five categories—dominated by Calculation Inaccuracy and Exact Match Failure, account for over 85% of errors, while the remaining ten categories are sparsely represented. This distribution makes low-frequency errors harder to learn. To improve coverage of these rare cases, we synthesize targeted long-tail verification examples using a stronger LLM(e.g.GPT-o3), under constrained generation settings. Specifically, for each lower-frequency error type, we randomly sample N existing examples from the same category and inject them into the prompt as in-context demonstrations. The generator is then asked to construct semantically similar but distinct verification instances that preserve the underlying error pattern. Formally, this process can be written as:

$$Q_{\text{new}} = G(P, Q_1, Q_2, \ldots, Q_N), \qquad (1)$$

where G denotes the LLM-based generation function, P is the constrained synthesis prompt, $Q_1, \ldots, Q_N$ are randomly sampled seed examples from the same error type, and Q_{new} is the synthesized verification instance.

After generation, we further filter the synthesized examples by retaining those that Qwen3-32B fails to verify correctly while still attempting tool use. This filtering step encourages the resulting data to focus on challenging cases that are both tool-relevant and difficult for existing models. In total, this procedure yields 10k long-tail synthesis instances and all training data statistics, prompts can be found in Appendix 7.3 and 7.5.

3.3.2 Difficulty-oriented Data Augmentation

After cold-start training on our constructed dataset, the model acquires foundational agentic verification skills. However, we observe that it still struggles with verification cases involving complex instructions, multi-step computations, or non-trivial tool-use decisions.

To further improve the model's capability in difficult verification settings, we introduce a difficulty-oriented data augmentation strategy. Starting from a small set of high-difficulty seed examples, we expand them into a larger training corpus through

multi-round self-iteration, targeting the current weaknesses of the model. Specifically, for each seed question, we first prompt an LLM to analyze its key difficulty factors, such as unit conversion, symbolic simplification or tool-selection ambiguity. We then randomly sample N difficulty factors and inject them into a constrained generation prompt, asking the LLM to synthesize new verification questions that exhibit similar challenges.

After generation, we apply a difficulty filter using the cold-start model, retaining only examples that the model fails to verify accurately. The retained hard examples are then merged back into the seed pool for the next iteration, enabling the generator to progressively explore more diverse and model-specific failure modes. Formally, the iterative augmentation process can be described as:

$$\mathcal{S}^{(t+1)} = \mathcal{S}^{(t)} \cup \mathcal{F}_{\theta_0}\left(G\left(P, d_1, d_2, \ldots, d_N\right)\right), \quad (2)$$

where $\mathcal{S}^{(t)}$ denotes the seed pool at iteration t, G is the LLM-based generator, P is the constrained generation prompt, and $d_1, \ldots, d_N$ are difficulty factors sampled from the LLM-analyzed difficulty set of $\mathcal{S}^{(t)}$. $\mathcal{F}_{\theta_0}$ denotes the filtering function based on the cold-start model θ_0, which keeps only examples that remain difficult for the current verifier.

After four scalable iterations, this procedure produces a corpus of 9,456 examples. A detailed algorithm is provided in Algorithm 1.

3.4 Post Training

To ensure verification accuracy and robustness, we equip CoSineVerifier with two executable tools: a Python interpreter and a unit-conversion utility. Specifically, we first perform supervised fine-tuning on the broad-coverage and balanced data constructed in Section 3.3.1, which enables the model to acquire foundational agentic verification skills. To maintain inference efficiency, we retain only the augmented cases that satisfy two criteria: (i) the trajectory produces a correct final answer, and (ii) contains explicit tool-call trajectories. This filtering leaves 29,339 instances and encourages the model to learn concise and effective verification, allowing its average output length to remain below 100 tokens even after reinforcement learning.

Then, to further improve the model in complex verification scenarios, we conduct reinforcement learning with DAPO (Yu et al., 2025) on the difficulty-oriented data introduced in Section 3.3.2. This stage strengthens the model on its weakness regions, including complex instructions, multi-step computation, unit conversion, and non-trivial tool-use decisions. During reinforcement learning, we combine an answer correctness reward with a tool-use encouragement term. Let $\hat{y}$ and y denote the predicted and ground-truth verification labels, and let u be the number of tool invocations in the trajectory. The final reward is defined as:

$$R = R_{\text{ans}} + \begin{cases} 0.5, & \text{if } \hat{y} = y \text{ and } u > 0, \\ -0.5, & \text{if } \hat{y} \neq y \text{ and } u = 0, \\ 0, & \text{otherwise.} \end{cases}$$

This reward keeps correctness as the primary objective while encouraging the model to use executable tools when tool-assisted verification is necessary.

Finally, we train two verifier variants for different use cases: CoSineVerifier-Tool and CoSineVerifier-Label. CoSineVerifier-Tool performs agentic reasoning with external tools before producing the final verification result, achieving high accuracy while maintaining efficient inference. Meanwhile , CoSineVerifier-Label directly outputs the verification judgment with a single token, making it suitable for latency-sensitive scenarios that require robust but extremely fast verification.

4 Experimental Setup

Datasets and Baselines We evaluate the CoSineVerifier family on three benchmarks: VerifyBench, VerifyBench-Hard (Yan et al., 2025), and SciBench (Zheng et al., 2025), which are challenging verification benchmarks spanning math, science, commonsense knowledge, and general reasoning domains with 2000, 1000, and 2500 samples. Our baselines mainly include two kinds of models: labeling verifiers and COT verifiers, with a difference in whether they contain reasoning chains for verification, and we report accuracy as mean@3 and efficiency as average output tokens per verdict on these benchmarks.

Implementation details Our CoSineVerifier-4B-Tool, CoSineVerifier-4B-Label and CoSineVerifier-32B are trained from Qwen3-4B-Instruct-2507, Qwen3-4B, and Qwen3-32B (Yang et al., 2025) respectively. We use OpenRLHF (Hu et al., 2024) for supervised fine-tuning and verl (Sheng et al., 2025) framework for reinforcement learning. All models are trained for one epoch on 32xA800 80G GPUS during supervised finetuning with 1e-5 learning rate, and 60 steps for reinforcement learning

Model	VerifyBench	VerifyBench-Hard	Sci-Bench	Avg. Tokens
CoT Verifier				
Closed-source Models				
o3(OpenAI, 2025)	96.1	88.7	87.5	206.7
GPT-4o(OpenAI, 2024)	96.0	84.6	86.0	192.4
Gemini2.5-Flash(Comanici et al., 2025)	96.0	86.0	85.9	193.0
Open-source Models				
GPT-oss-20B(Agarwal et al., 2025)	92.2	84.7	85.0	221.0
LLaMA3.3-70B-Instruct(Dubey et al., 2024)	94.8	77.2	84.8	347.3
Qwen3-4B(Yang et al., 2025)	92.6	80.3	82.0	1156.7
Qwen3-8B(Yang et al., 2025)	93.7	83.6	83.9	926.6
Qwen3-32B(Yang et al., 2025)	94.7	85.2	83.5	798.8
Qwen3-4B-Instruct-2507(Yang et al., 2025)	94.7	84.1	82.4	869.7
Qwen3-4B-Instruct-2507(w tool)(Yang et al., 2025)	92.0	75.3	81.5	2498.3
Qwen3-235B-A22B-2507(Yang et al., 2025)	94.4	87.7	82.6	1885.3
Qwen3-235B-A22B-2507(w tool)(Yang et al., 2025)	93.1	85.7	84.6	4284.3
CompassVerifier-7B (CoT)(Liu et al., 2025a)	93.5	82.6	84.2	234.7
CompassVerifier-32B (CoT)(Liu et al., 2025a)	95.9	86.5	85.5	213.0
CoSineVerifier-4B-Tool	**96.6**	**91.9**	**89.7**	**95.3**
Labeling Verifier				
XVerify-8B-I(Chen et al., 2025a)	92.5	83.3	78.1	1.0
CompassVerifier-7B(Liu et al., 2025a)	93.5	85.2	85.7	1.0
CompassVerifier-32B(Liu et al., 2025a)	**96.3**	88.9	85.3	1.0
CoSineVerifier-4B-Label	95.7	85.4	85.9	1.0
CoSineVerifier-32B-Label	95.7	**90.0**	**86.4**	**1.0**

Table 1: Main results on three verify benchmarks. We separate CoT verifier and Labeling verifier for fair comparison, where CoT verifier outputs a reasoning chain before achieving the final answer while labeling verifier directly gives its verdict. **Bold** highlights the overall best accuracy, while underline highlights the second best accuracy. We report mean@3 accuracy and average token used per question.

with 1e-6 learning rate. All training details are listed in Appendix 7.4.

5 Experimental Results

5.1 Verification Performance

As shown in Table 1, our CoSineVerifier series outperforms all verifier baselines across three verification benchmarks. Specifically, for CoT Verifier, our CoSineVerifier-Tool-4B surpasses baselines by 0.5% - 14.7% with noticeably fewer tokens, especially on harder benchmarks like VerifyBench (hard) and Sci-Bench, where complex calculation and long-context string comparing are needed to verify the LLM-generated answer. In addition, for labeling verifiers, CoSineVerifier-4B-Label and CoSineVerifier-32B-Label also surpass other labeling verifiers by 0.7%-1.1%, demonstrating stronger verification skills under single-token scenarios.

Furthermore, simply equipping stronger LLMs, such as Qwen3-235B-A22B, with external tools does not necessarily lead to better verification performance; in fact, their performance even drops on VerifyBench and Sci-Bench. This is mainly because these models fail to invoke and coordinate tools effectively during difficult verification processes. These results suggest that agentic verification is not merely a standard tool-augmented reasoning problem, but a specialized capability that conventional tool-augmented LLMs generally fail to acquire without targeted training.

Figure 4 demonstrates the performance comparison of our CoSineVerifier against various model across different domains. Among three benchmarks, even closed-source SOTA models like GPT-o3 still struggle with reliable verification on compute-intensive and scientific reasoning tasks. In contrast, both CoSineVerifier-4B-Tool and CoSineVerifier-32B-Label demonstrate exceptional proficiency in the Math and Science domains while achieving competitive or even superior performance on the remaining domains. Despite being much smaller in parameter size, the superior accuracy exhibited by CoSineVerifier-4B-Tool vali-

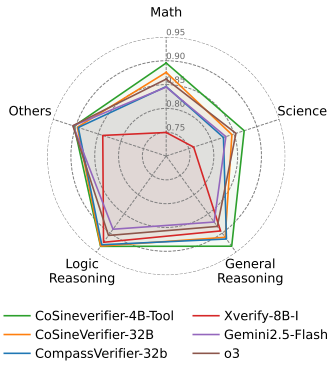

Figure 4: Analysis of CoSineVerifier-4B-Tool verification accuracy across different domains

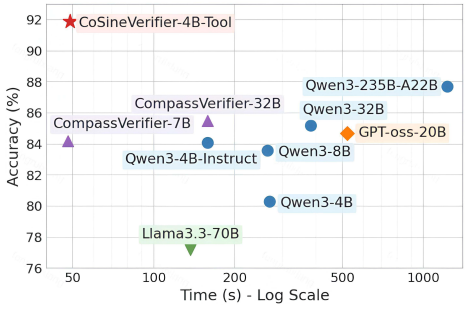

Figure 5: Analysis of CoSineVerifier-4B-Tool inference efficiency with other models.

dates the efficacy of integrating external calculation tools to enhance the model's verification capabilities. These results demonstrate the feasibility of tool-augmented, small-scale models to exceed the performance of large SOTA models in complex verification tasks.

5.2 Verification Efficiency Analysis

We further evaluate the efficiency of CoSineVerifier-4B-Tool against several baselines in Figure 5(b). To ensure a fair comparison, all models are tested on 4 NVIDIA A800 GPUs. We report the mean inference latency computed over 1000 randomly sampled prompts. While CoSineVerifier-4B-Tool achieves high accuracy, incorporating external tools does not incur significant inference latency. We attribute this efficiency to two primary factors: (1) our efficiency-driven cold-start data construction methodology, which explicitly optimizes for short and precise tool-calling trajectories, resulting in 2.4x-19x times less token usage compared to other models. (2) the computational efficiency afforded by the model's compact 4B parameter count, which minimizes the overhead of the backbone model during inference.

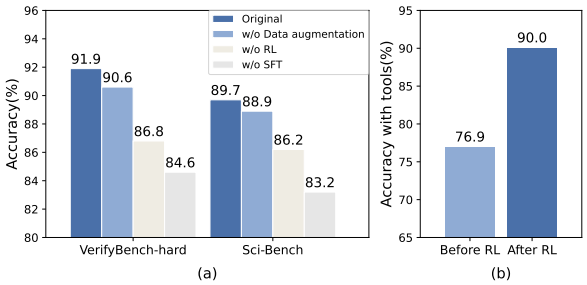

Figure 6: (a) Ablation study of CoSineVerifier-4B-Tool. (b) Average verification accuracy with tools before RL and after RL.

5.3 Ablation Study

We conduct ablation studies on each module's contribution to CoSineVerifier-4B-Tool. As shown in Figure 6(a), removing any component consistently degrades the performance of CoSineVerifier-4B-Tool. The cold-start SFT stage is particularly important: without it, accuracy drops from 91.9% to 84.6% on VerifyBench-hard and from 89.7% to 83.2% on Sci-Bench. We further find that the model without cold-start training often fails to produce valid function calls and may easily fall into repetitive tool-use loops, leading to both lower accuracy and higher inference latency. RL further improves the model's ability to use tools effectively. As shown in Figure 6(b), the accuracy on samples involving tool trajectories increases from 76.9% to 90.0% after RL, indicating that RL teaches the model not only to invoke tools, but also to decide when and how to use them for verification. Data augmentation also provides consistent gains by exposing the model to long-tail and hard verification cases, which are rare in the original training distribution but crucial for robustness. Overall, these results demonstrate that both our two-stage training strategy and data augmentation are essential for building a reliable agentic verifier.

5.4 Evaluation on RLVR Task

One primary objective of our CoSineVerifier series is to establish a reliable and efficient outcome reward verifier tailored for real-world data processing and reinforcement learning. Thus, we further evaluate multiple answer-verification methods in RLVR tasks to show the efficacy of CoSineVerifier. Specifically, we train Qwen3-4B-Instruct-2507 with an on-policy GRPO algorithm on math and science tasks in separate runs, each paired with different verifiers. For the math setting, we construct competition-math problems with 42K train-

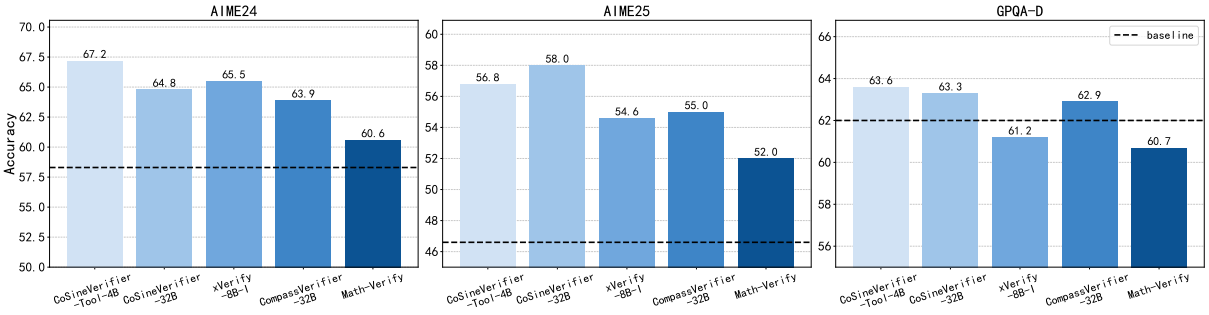

Figure 7: Results of using CoSineVerifier as a reward model on AIME24, AIME25 and GPQA-D.

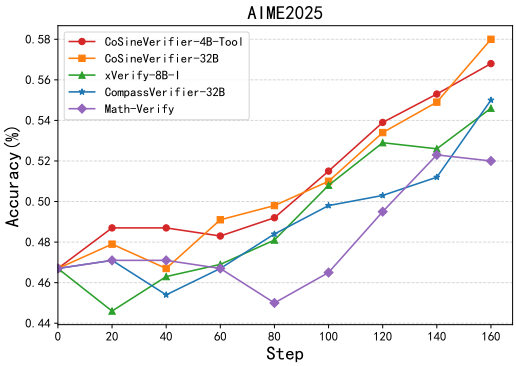

Figure 8: Accuracy of AIME 2025 during training.

ing data drawn from DAPO-Math-17k (Yu et al., 2025), OpenR1-Math-220k (Hugging Face, 2025), and DeepScaleR-Preview (Luo et al., 2025). And for the science setting, we directly leverage guru-RL(Cheng et al., 2025) as our training set. We compare our approach against various verifiers, including both rule-based and model-based methods, under identical training configurations. We report mean@32 accuracy on AIME24&25 for the math task and GPQA-D for the science task. Comprehensive training details are provided in Appendix 7.4.

CoSineVerifier enables robust reasoning improvements As shown in Figure 7, using CoSineVerifier as the reward model consistently improves the base model's reasoning performance. On mathematical benchmarks, CoSineVerifier outperforms other verifiers by 1.7%–6.6% on AIME 2024 and 1.8%–6.0% on AIME 2025. In the science domain, CoSineVerifier also achieves stable gains of 0.7%–2.9% over competing methods. We further observe a clear gap between model-based and rule-based verifiers. This gap is especially evident on GPQA-D, where Math-Verify leads to a 1.3% drop below the baseline. This suggests that noisy or inaccurate reward signals can misguide policy optimization toward undesirable reasoning trajectories, ultimately hurting performance. These results highlight the importance of precise and reliable verification for RL training, particularly when further improving already well-tuned models.

Verifier quality shapes performance and reasoning ceiling To investigate how verifier quality affects RL training dynamics, we track the base model's accuracy on AIME 2025 throughout training. Specifically, we ask: *How does the choice of verifier affect convergence and final performance under identical RL settings?* As shown in Figure 8, CoSineVerifier consistently outperforms all baselines across training steps and reaches the highest final accuracy. Notably, its advantage comes primarily from a higher performance ceiling rather than uniformly faster convergence. In contrast, weaker verifiers such as Math-Verify lead to slower improvement and lower final performance, suggesting that noisy or less reliable rewards directly limit policy optimization. These results highlight the importance of high-fidelity verification signals for RL on challenging reasoning tasks (Phan et al., 2025; Guan et al., 2025).

6 Conclusion

In this paper, we propose CoSineVerifier, an agentic LLM-based verifier for general reference-based verification, addressing the limitations of existing methods in precise verification. Experiments show that CoSineVerifier achieves state-of-the-art accuracy across multiple verification benchmarks while maintaining high efficiency. When used as a reward model in RL training, CoSineVerifier leads to higher final performance on challenging math and science benchmarks, outperforming existing verifiers. Overall, our results highlight CoSineVerifier as an efficient and general-purpose verifier that can serve as a reliable foundation for data curation, evaluation, and reward-driven model improvement.

Limitations

Despite strong empirical results, CoSineVerifier's reliability is tightly coupled to its available tools and sandboxed execution environment, which may incur extra computational burden during RLVR training. Moreover, CoSineVerifier is fundamentally a reference-based verifier: it assesses correctness by comparing model outputs against a provided reference answer. As a result, it is less applicable to settings where ground-truth references are unavailable, ambiguous, or inherently subjective (e.g., open-ended generation, preference-based tasks, or partially specified problems), limiting its direct transfer to non-verifiable domains.

Ethical Considerations

CoSineVerifier is designed to improve the faithfulness of automated verification by grounding judgments in tool-executable computations, thereby making model training and evaluation more reliable. However, CoSineVerifier can still produce incorrect judgments during data curation or RLVR training; such errors may inject spurious reward signals and, in turn, steer optimization toward unintended behaviors.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Alon Albalak, Duy Phung, Nathan Lile, Rafael Rafailov, Kanishk Gandhi, Louis Castricato, Anikait Singh, Chase Blagden, Violet Xiang, Dakota Mahan, et al. 2025. Big-math: A large-scale, high-quality math dataset for reinforcement learning in language models. *arXiv preprint arXiv:2502.17387*.

Ding Chen, Qingchen Yu, Pengyuan Wang, Wentao Zhang, Bo Tang, Feiyu Xiong, Xinchi Li, Minchuan Yang, and Zhiyu Li. 2025a. xverify: Efficient answer verifier for reasoning model evaluations. *arXiv preprint arXiv:2504.10481*.

Nuo Chen, Zhiyuan Hu, Qingyun Zou, Jiaying Wu, Qian Wang, Bryan Hooi, and Bingsheng He. 2025b. Judgelrm: Large reasoning models as a judge. *arXiv preprint arXiv:2504.00050*.

Yongchao Chen, Yueying Liu, Junwei Zhou, Yilun Hao, Jingquan Wang, Yang Zhang, Na Li, and Chuchu Fan. 2025c. R1-code-interpreter: Llms reason with code via supervised and multi-stage reinforcement learning. *arXiv preprint arXiv:2505.21668*.

Zhoujun Cheng, Shibo Hao, Tianyang Liu, Fan Zhou, Yutao Xie, Feng Yao, Yuexin Bian, Yonghao Zhuang, Nilabjo Dey, Yuheng Zha, et al. 2025. Revisiting reinforcement learning for llm reasoning from a cross-domain perspective. *arXiv preprint arXiv:2506.14965*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Guanting Dong, Yifei Chen, Xiaoxi Li, Jiajie Jin, Hongjin Qian, Yutao Zhu, Hangyu Mao, Guorui Zhou, Zhicheng Dou, and Ji-Rong Wen. 2025. Tool-star: Empowering llm-brained multi-tool reasoner via reinforcement learning. *arXiv preprint arXiv:2505.16410*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

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

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, et al. 2025. Openthoughts: Data recipes for reasoning models. *arXiv preprint arXiv:2506.04178*.

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

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, et al. 2025a. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, et al. 2025b. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *arXiv preprint arXiv:2504.11456*.

Jian Hu, Xibin Wu, Zilin Zhu, Weixun Wang, Dehao Zhang, Yu Cao, et al. 2024. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*.

Yuzhen Huang, Weihao Zeng, Xingshan Zeng, Qi Zhu, and Junxian He. 2025. Pitfalls of rule-and model-based verifiers–a case study on mathematical reasoning. *arXiv preprint arXiv:2505.22203*.

Hugging Face. 2025. Open r1: A fully open reproduction of deepseek-r1.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*.

Hynek Kydlíček. Math-Verify: Math Verification Library.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. 2024. From crowdsourced data to high-quality benchmarks: Arena-hard and benchbuilder pipeline. *arXiv preprint arXiv:2406.11939*.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.

Heng Lin and Zhongwen Xu. 2025. Understanding tool-integrated reasoning. *arXiv preprint arXiv:2508.19201*.

Shudong Liu, Hongwei Liu, Junnan Liu, Linchen Xiao, Songyang Gao, Chengqi Lyu, Yuzhe Gu, Wenwei Zhang, Derek F Wong, Songyang Zhang, et al. 2025a. Compassverifier: A unified and robust verifier for llms evaluation and outcome reward. *arXiv preprint arXiv:2508.03686*.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. 2025b. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*.

Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl. Notion Blog.

Ruotian Ma, Peisong Wang, Cheng Liu, Xingyan Liu, Jiaqi Chen, Bang Zhang, Xin Zhou, Nan Du, and Jia Li. 2025a. S^2r: Teaching llms to self-verify and self-correct via reinforcement learning. *arXiv preprint arXiv:2502.12853*.

Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. 2025b. Tool-integrated reinforcement learning for repo deep search. *arXiv preprint arXiv:2508.03012*.

Xinji Mai, Haotian Xu, Zhong-Zhi Li, Xing W, Weinong Wang, Jian Hu, Yingying Zhang, and Wenqiang Zhang. 2025. Agent rl scaling law: Agent rl with spontaneous code execution for mathematical problem solving. *arXiv preprint arXiv:2505.07773*.

OpenAI. 2024. GPT-4o system card. *arXiv preprint arXiv:2410.21276*.

OpenAI. 2025. Openai o3 system card. Technical report, OpenAI.

Samuel J. Paech. 2023. Eq-bench: An emotional intelligence benchmark for large language models. *arXiv preprint arXiv:2312.06281*.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. 2025. Humanity's last exam. *arXiv preprint arXiv:2501.14249*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.

Shubham Toshniwal, Wei Du, Ivan Moshkov, Branislav Kisacanin, Alexan Ayrapetyan, and Igor Gitman. 2024. Openmathinstruct-2: Accelerating ai for math with massive open-source instruction data. *arXiv preprint arXiv:2410.01560*.

Mingyuan Wu, Jingcheng Yang, Jize Jiang, Meitang Li, Kaizhuo Yan, Hanchao Yu, Minjia Zhang, Chengxiang Zhai, and Klara Nahrstedt. 2025a. Vtool-r1: Vlms learn to think with images via reinforcement learning on multimodal tool use. *arXiv preprint arXiv:2505.19255*.

Yuning Wu, Jiahao Mei, Ming Yan, Chenliang Li, Shaopeng Lai, Yuran Ren, Zijia Wang, Ji Zhang, Mengyue Wu, Qin Jin, et al. 2025b. Writingbench: A comprehensive benchmark for generative writing. *arXiv preprint arXiv:2503.05244*.

Yu Xia, Jingru Fan, Weize Chen, Siyu Yan, Xin Cong, Zhong Zhang, Yaxi Lu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2025. Agentrm: Enhancing agent generalization with reward modeling. *arXiv preprint arXiv:2502.18407*.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. 2025. Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. *arXiv preprint arXiv:2509.02479*.

Yuchen Yan, Jin Jiang, Zhenbang Ren, Yijun Li, Xudong Cai, Yang Liu, Xin Xu, Mengdi Zhang, Jian Shao, Yongliang Shen, et al. 2025. Verifybench: Benchmarking reference-based reward systems for large language models. *arXiv preprint arXiv:2505.15801*.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Weizhe Yuan, Jane Yu, Song Jiang, Karthik Padthe, Yang Li, Ilia Kulikov, Kyunghyun Cho, Dong Wang, Yuandong Tian, Jason E Weston, et al. 2025. Naturalreasoning: Reasoning in the wild with 2.8 m challenging questions. *arXiv preprint arXiv:2502.13124*.

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. 2025. Nemotron-research-tool-n1: Exploring tool-using language models with reinforced reasoning. *arXiv preprint arXiv:2505.00024*.

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

7 Appendix

7.1 Error Analysis

Table 2 demonstrates the detailed types and distribution of our summarized errors. We run 3 times for each verification method and leverage Qwen3-32b to identify their errors. We then send these errors to GPT-o3 to summarize these errors into 15 error types. After obtaining these error types, we use Qwen3-32b to annotate errors for each verification method and derive the statistics.

7.2 Difficulty-oriented Data Augmentation

We provide the detailed algorithm for difficulty-oriented data augmentation below:

Input: Initial hard set $\mathcal{D}_0$; questioner Q; solver S; rounds T.
Output: Reinforcement-learning corpus $\mathcal{R}$.

$\mathcal{R} \leftarrow \{\}$
$\mathcal{D} \leftarrow \mathcal{D}_0$
for $t \leftarrow 1$ **to** T **do**
  $\mathcal{Q}_t \leftarrow Q(\text{conditioned on } \mathcal{D})$
  `// Conditioned generation by Questioner`
  $\mathcal{F}_t \leftarrow \{x \in \mathcal{Q}_t \mid S(x) \text{ is incorrect}\}$
  `// Accumulate failures as next-round context`
  $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{F}_t$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{F}_t$
end
return $\mathcal{R}$

Algorithm 1: Iterative Difficult Data Synthesis

7.3 Training Data Statistics

In this section, we provide the relevant statistics of our CoSineVerifier training data.

Source	Count	Rate
Science	452661	39.70%
Math	271415	23.80%
DROP	103070	9.04%
SuperGPQA	60356	5.29%
Ceval	59501	5.21%
Cmmlu	52222	4.58%
MMLU_pro	50301	4.41%
ARC	33148	2.90%
SimpleQA	18103	1.58%
ChineseSimpleQA	12720	1.11%
BBH	11794	1.03%
Korbench	9561	0.83%
GAOKAOBench	3281	0.28%
GPQA	1913	0.16%
Total	**1,140,046**	**100.00%**

Table 3: Candidate training data distribution

Source	Count	Rate
Python interpreter data	2500	29.33%
Unit conversion data	1500	17.60%
No Tool use data	3000	35.20%
Long-tail augment data	1523	17.87%
Total	**8,523**	**100.00%**

Table 4: Cold start data distribution

Source	Count	Rate
Science	20044	31.63%
Math	8789	13.87%
SuperGPQA	8409	13.27%
MMLU_pro	6740	10.64%
Korbench	5877	9.27%
BBH	5745	9.07%
DROP	2974	4.69%
SimpleQA	1757	2.77%
ChineseSimpleQA	1215	1.92%
GAOKAOBench	849	1.34%
Ceval	484	0.76%
Cmmlu	308	0.49%
ARC	145	0.23%
GPQA	38	0.06%
Total	**63,374**	**100.00%**

Table 5: Disagreement data distribution

Error Name	Description	Percentage
Calculation Inaccuracy	The final numerical value is mathematically incorrect.	25.089%
Comparison Judgment Error	The response (e.g., a sequence) does not perfectly match the reference.	19.618%
Format Error	The answer's structure violates the required output format.	16.085%
Incomplete Answer Error	Fails to provide all required answers, e.g., missing candidates or sub-parts.	15.957%
Precision/Boundary Error	Incorrect rounding or definitions of interval endpoints, e.g., open/closed.	7.843%
Invalid Query	The error originates from a defective question or reference answer.	6.007%
Incorrect Simplification	Fails to simplify to minimal form or simplifies erroneously.	2.938%
Constraint Violation	The answer ignores a specific rule, e.g., "use each number once" .	2.166%
Missing Final Result	Provides only reasoning or code without the conclusive answer.	1.836%
Refusal or Inconclusive Response	Fails to provide a clear answer or states it is unanswerable.	1.355%
Extraneous Content Error	The response includes correct data mixed with unrelated, incorrect information.	0.677%
Self-Correction Failure	The model initially answers correctly but changes to an incorrect answer.	0.285%
Truncated Response	The response is clearly cut off and incomplete.	0.085%
Garbled or Corrupted Output	The response contains unreadable characters or significant noise.	0.046%
Meaningless Repetition Error	The response repeats text extensively, hiding any valid answer.	0.013%

Table 2: Error Types and Distributions

Model	Count	Rate
NBG-Family	13308	21.00%
DeekSeek-Qwen-7B	6337	10.00%
GPT-OSS-20B	5300	8.36%
DeepSeek-Qwen-1.5B	4874	7.69%
Qwen3-30B-A3B	4488	7.08%
Llama3.1-8B	4056	6.40%
MiMo-7b	3581	5.65%
Qwen3-4B	3405	5.37%
Qwen2.5-7B	2931	4.62%
Qwen3-32B	2917	4.60%
Qwen3-1.7B	2763	4.36%
Gemma2-9B	2474	3.90%
Gemma2-2B	2415	3.81%
InternLM-7B	2277	3.59%
Qwen3-8B	2248	3.55%
Total	**63,374**	**100.00%**

Table 6: Model responses count on disagreement data distribution (NBG-Family aggregated)

7.4 Training Details

7.4.1 Training CoSineVerifier-4B-Label and CoSineVerifier-32B-Label

For labeling verifier CoSineVerifier-4B and CoSineVerifier-32B, we conducted supervised fine-tuning on 73,714 samples, which contains 63.714 disagreement data, 2000 long-tail augmentation data and 8000 difficult-oriented augmentation data. These two verifiers are designed to output binary verification(*Correct* and *Incorrect*) without reasoning, maintaining superior efficiency while having performance comparable to CoSineVerifier-4B-Tool. Detailed training parameters are listed in Table 7:

Parameter	Value
BF16	True
Learning Rate	1×10^{-5}
LR Scheduler Type	cosine_with_min_lr
Maximum Sequence Length	16384
Training Epochs	1
Use Liger Kernel	True
Warmup Ratio	0.01

Table 7: Cold start training configurations of CoSineVerifier-4B-Tool and supervised fine tuning configurations of CoSineVerifier-4B-Label and CoSineVerifier-32B-Label

7.4.2 Ablation Study of Labeling Verifier

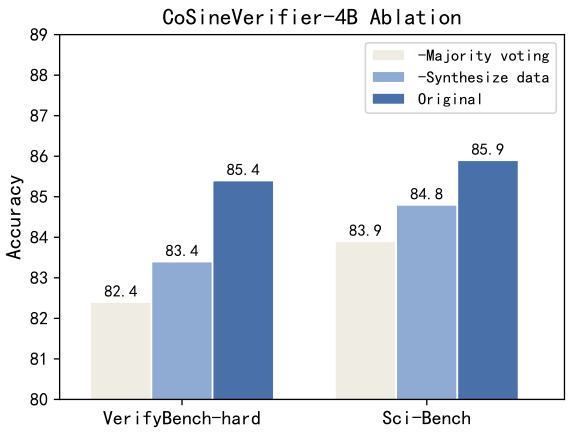

Figure 9: Ablation study of CoSineVerifier-4B.

For CoSineVerifier-4B, we investigate the impact of removing: **(1) multi-turn majority voting** and **(2) long-tail data synthesis**. As illustrated in Figure 9, removing any model from CoSineVerifier consistently leads to performance degradation, demonstrating the necessity of our holistic framework design. Specifically, excluding multi-turn majority data results in the most significant performance deterioration across two major benchmarks. This finding strongly underscores the imperative of collecting reliable "golden" answers from diverse, real-world domains. In addition, removing long-tail synthesis data also causes a marked decline, showing the effectiveness of long-tail alignment for rare but high-value cases. In conclusion, only combining two components together yields the best performance

7.4.3 Training Details of CoSineVerifier-4B-Tool

We detail the training configurations of CoSineVerifier-4B-Tool. For cold-start process, we use 8,523 samples for cold-start with long-tail augmentation data, and we then use 9,456 difficulty-oriented augmentation data for reinforcement learning. All training configurations are listed below.

Parameters	Value
BF16	True
Train batch size	128
Train epochs	10
Learning rate	1e-6
Max prompt length	16384
Max response length	2048
n	8
$\epsilon_{\min}$	0.2
$\epsilon_{\max}$	0.28
Temperature	1.0

Table 8: RL training configurations of CoSineVerifier-4B-Tool.

Parameters	Value
BF16	True
Train batch size	128
Train epochs	3
Learning rate	1e-6
Max prompt length	4096
Max response length	16384
n	8
$\epsilon_{\min}$	0.2
$\epsilon_{\max}$	0.28
Temperature	1.0

Table 9: Training configurations of RLVR application.

7.5 Prompts

7.5.1 Prompts for Error Analysis

Error Analysis Prompt

```
# ROLE: You are an expert AI assistant specializing in error analysis.

# TASK: Your goal is to analyze an "Incorrect Verification" and categorize the reasoning mistake it contains. Compare the "Incorrect Verification" to the "Correct Verification" to identify the error types described below.
```

Instructions: You should only choose the Error Categories below and do not create a new one. Focus only on the Category name, not the description in parentheses nor the labeling numbers. Wrap your answer in the final with boxed{{}}

INPUTS:
Question: The original question that was asked.
Model Answer: The answer provided by a model, which is being evaluated.
Reference Answer: The ground-truth correct answer.
Correct Verification (golden_verify): The correct reasoning used to determine if the Model Answer is right or wrong. Only in '[Correct]' or '[Incorrect]'.
Incorrect Verification (error_verify): A flawed line of reasoning that you must analyze. Only in '[Correct]' or '[Incorrect]'.

— ## Error Categories##
1. Calculation Inaccuracy (The final numerical value is mathematically incorrect.)
2. Format Compliance Error (The answer's structure violates the required output format.)
3. Constraint Violation (The answer ignores a specific rule, e.g., "use each number once".)
4. Incorrect Simplification (Fails to simplify to minimal form or simplifies erroneously.)
5. Precision and Boundary Error (Incorrect rounding or definitions of interval endpoints, e.g., open/closed.)
6. Exact Match Failure (The response (e.g., a sequence) does not perfectly match the reference.)
7. Answer Completeness Error (Fails to provide all required answers, e.g., missing candidates or sub-parts.)
8. Missing Final Result (Provides only reasoning or code without the conclusive answer.)
9. Extraneous Content Error (The response includes correct data mixed with unrelated, incorrect information.)
10. Self-Correction Failure (The model initially answers correctly but changes to an incorrect answer.)
11. Refusal or Inconclusive Response (Fails to provide a clear answer or states it is unanswerable.)
12. Garbled or Corrupted Output (The response contains unreadable characters or significant noise.)
13. Truncated Response (The response is clearly cut off and incomplete.)
14. Meaningless Repetition Error (The response repeats text extensively, hiding any valid answer.)
15. Invalid Task or Reference (The error originates from a defective question or reference answer.)

Question: {question}
Model Answer: {answer}
Reference Answer: {ref_answer}
Correct Verification: {golden_verify}
Incorrect Verification: {error_verify}

Your classified error category:

7.5.2 Prompts for data augmentation

Prompt for data augmentation

Role: You are an education expert.

Task: Your student's assignment is to check whether a model's given answer to a question is consistent with the standard (reference) answer. Your mission is to

generate a new practice problem for the student based on their past mistakes, in order to test and strengthen their verification skills.

Input: You will be given one examples of the student's incorrect exercises and its mistake type.

Instructions: First, analyze the given question, reference answer the model answer. Second, based on your analysis and the given mistake type, create a new practice question that specifically targets this weakness. Make sure the difficulty and style of this verification practice is on par with the mistake.

Output Format: Please output the new practice problem in JSON format with the following fields:
{{ "question": "...",
"ref_answer": "...",
"model_answer": "...",
"ref_judge": "Only in [Correct] or [Incorrect], represents the golden judge tag"
}}

Input Examples: Mistake types: {mistake_type} <example1>
{example1}
</example1>

Your output:

7.5.3 All prompt prefixes in training data

All prompt prefixes in training data

1. "Let's think step by step and output the final answer within \\boxed{}.\n"
2. "Solve the following problem step by step. The last line of your response should be of the form Answer: $Answer (without quotes) where $Answer is the answer to the problem.\n\n"
3. "Let's think step by step. \n"
4. "Please answer the following question. \n"
5. ""
6. "Answer the following multiple choice question. The last line of your response should be of the following format: 'Answer: $LETTER' (without quotes) where LETTER is one of ABCD. Think step by step before answering.\n\n"
7. "Answer the following multiple choice question. The last line of your response should be of the following format: 'Answer: $LETTER' (without quotes) where LETTER is one of ABCD.\n\n"

7.6 Case Study

CosineVerifier-4B-Tool for calculation

Question: A block of mass $m = 100$ g and of negligible size slides down a wedge of mass $M = 10$ kg, with angle above the horizontal $\theta = 30°$, and maximum height $h = 2$ m. The block is free to move as well, and there is a coefficient of friction $\mu = \frac{1}{2}$ between the wedge and block. Initially, the block is at rest with respect to the wedge and sits at the top of the wedge. The wedge is moving at $v_0 = 1$ m/s with the point of the wedge moving "forward."
For this problem, use $g = 10$ m/s^2. Find the exact amount of time it takes for the block to slide all the way to the bottom of the wedge. Remember to put your final answer within
boxed

---

model_answer:
"model_ans": "To solve this problem, we need to analyze the motion of the block and the wedge,

taking into account the forces acting on them and the constraints of the system.
Step 1: Analyze the forces and motion
1. **Forces on the block:** - Gravity: mg acting downward. - Normal force: N perpendicular to the wedge's surface. - Friction force: $f = \mu N$ acting up the wedge.
2. **Forces on the wedge:** - Gravity: Mg acting downward. - Normal force: N from the block. - Friction force: $f = \mu N$ acting down the wedge. - Initial velocity: $v_0 = 1$ m/s.
Step 2: Set up the equations of motion
1. **For the block:** - Along the wedge: $ma_{\text{block}} = mg\sin\theta - f$ - Perpendicular to the wedge: $N = mg\cos\theta$
2. **For the wedge:** - Horizontal direction: $Ma_{\text{wedge}} = f\cos\theta - N\sin\theta$
Step 3: Substitute the known values
1. **Normal force:**

$$N = mg\cos\theta = 0.1 \times 10 \times \cos 30^\circ$$
$$= 0.1 \times 10 \times \frac{\sqrt{3}}{2} = 0.5\sqrt{3} \text{ N}$$

2. **Friction force:**

$$a_{\text{block}} = g\sin\theta - \frac{f}{m} = 10 \times \frac{1}{2} - \frac{0.25\sqrt{3}}{0.1}$$
$$= 5 - 2.5\sqrt{3} \text{ m/s}^2$$

Step 4: Calculate the time to slide down
The block slides down a distance $s = \frac{h}{\sin\theta} = \frac{2}{\sin 30^\circ} = 4$ m along the wedge.
Using the kinematic equation:

$$s = \frac{1}{2}a_{\text{block}}t^2$$

$$4 = \frac{1}{2}(5 - 2.5\sqrt{3})t^2$$

$$t^2 = \frac{8}{5 - 2.5\sqrt{3}}$$

$$t = \sqrt{\frac{8}{5 - 2.5\sqrt{3}}}$$

Step 5: Simplify the expression
To simplify, we can rationalize the denominator:

$$t = \frac{8\sqrt{5 + 2.5\sqrt{3}}}{5}$$

Final Answer:

$$\boxed{\frac{8\sqrt{5 + 2.5\sqrt{3}}}{5}}$$

"

---

reference answer:
t = $\sqrt{((1604(2+\sqrt{3})/505)}$ s

CosineVerifier-4B-Tool verification:
<tool_call>
{"name": "python_interpreter", "arguments": {"code": "def main():\n import math\n cand = 8*math.sqrt(5 + 2.5*math.sqrt(3))/5\n gold = math.sqrt(1604*(2+math.sqrt(3))/505)\n print(cand, gold)\n\nif __name__ == \"__main__\":\n main()"}}
</tool_call>
<tool_response>
{"compile_result": null, "execution_info": {"code_length": 183, "execution_phases": ["execution_start", "execution_end"], "execution_start_time": 1760942845.014331, "language": "python", "stdin_length": 4, "stdin_provided": true, "temp_file": "/tmp/tmposhv6sxt.py", "total_execution_time": 0.028516054153442383, "warnings": []}, "run_result": {"command": ["python3", "/tmp/tmposhv6sxt.py"], "execution_time": 0.028277158737182617,

"exit_success": true, "return_code": 0, "status": "Finished", "stderr": "", "stdin_used": "None", "stdout": "4.887241058965765 3.4429464400122507\n", "working_directory": "/tmp"}, "status": "Success"}
</tool_response>

---

assistant

The candidate's final expression t = $8\sqrt{5+2.5\sqrt{3}/5}$ is numerically $\approx$ 4.89 s.

The reference expression t = $\sqrt{(1604(2+\sqrt{3})/505)}$ is numerically $\approx$ 3.44 s.

Since 4.89 $\neq$ 3.44, the candidate answer does not match the gold answer.

$\boxed{[Incorrect]}$

CosineVerifier-4B-Tool for string match

Question: "Please generate the corresponding RNA sequence directly based on the dot-bracket notation of the secondary structure: .....((((.....)))).((((((.....((((((((.....)))))))).))) .....((((((((((...)))))))))))... .....))))))"

---

model_answer:

"AUUGCUCGAAUUUAUAGGACUUUUUUCUAUAAA GAAUAGUUUGGACUUGAAAUG UAUUUAAAAACAAGAGGUUGGUAGAUUAUCAGCC UCUUUCUUGUCGUUGAAAAAG",

---

reference_answer:

ACGGGUUUCCCGGGAAACCCCAAAAUGGGGCCCC UGUCGGGUUUUAUUCCCUGUCGUCGCCCUUUUUG GGA

CosineVerifier-4B-Tool verification:

<tool_call>
{"name": "python_interpreter", "arguments": {"code": "def main():\n correct = \"AUUGCUCGAAUUUAUAGGACUUUUUUCUAUAAA GAAUAGUUUGG ACUUGAAAUGUAUUUAAAAACAAGAGGUUGGUAGAU UAUCAGCCUCUUUCUUGUCGUUGAAAAAG\"\n candidate = \"ACGGGUUUCCCGGGAAACCCCAAAAUGGGGCCCC UGUCGGGUUUUAUUCCCUG UCGUCGCCCUUUUUGGGA\"\print('match' if correct == candidate else 'no')\n\nif __name__ == \"__main__\":\n main()"}}
</tool_call>
<tool_response>
{"compile_result": null, "execution_info": {"code_length": 411, "execution_phases": ["execution_start", "execution_end"], "execution_start_time": 1761468426.650931, "language": "python", "stdin_length": 4, "stdin_provided": true, "temp_file": "/tmp/tmp4b5lvbkx.py", "total_execution_time": 0.028637170791625977, "warnings": []}, "run_result": {"command": ["python3", "/tmp/tmp4b5lvbkx.py"], "execution_time": 0.02838897705078125, "exit_success": true, "return_code": 0, "status": "Finished", "stderr": "", "stdin_used": "None", "stdout": "False", "working_directory": "/tmp"}, "status": "Success"}
</tool_response>

---

assistant

The candidate RNA sequence does not match the gold sequence at all; their nucleotide strings are completely different. $\boxed{Incorrect}$

CosineVerifier-4B-Tool for unit conversion

Question: "Please provide the LUMO energy value for this molecule.\nO=C1C=C(CO)C(=N)N1"

---

model_answer:
"-7.00~\text{kJ/mol}"

---

reference answer: "5.48e22 eV\mole"

CosineVerifier-4B-Tool verification:
<tool_call>
{"name": "unit_conversion", "arguments": {"value": 7.0, "source_unit": "kJ/mol", "target_unit": "eV/mole"}}
</tool_call>
<tool_response>
Unit parsed value: 4.369056352122534e+22 electron_volt / mole
</tool_response>

---

assistant
The gold answer is 5.48e22 eV\mole. The candidate gives -7.00 kJ mol^{-1}. The unit conversion function tells us it's roughly equal to 4.369056352122534e+22 electron_volt / mole. This is nowhere near s 5.48e22 eV\mole.
$\boxed{[Incorrect]}$

8 Prompts

CoSineVerifier-4B-Tool verification prompt

You are a diligent and precise assistant tasked with evaluating the correctness of responses. You will receive a question, an output sentence, and the correct answer. Your task is to determine if the output sentence correctly answers the question based on the provided correct answer. You can perform a short tool call and a short reasoning process. After a short reasoning process, put your response in the final with either [Correct] or [Incorrect] wrapped in □.

Evaluation Protocol:

1. Reference Standard:
   - The standard (gold) answer is definitive and always correct.
   - The question is always valid — never challenge it.
   - Allow equivalent meaning answers.
   - Do not regenerate answers; only compare candidate's final answer with the gold answer.
   - You only need to compare correct answer and output sentence, do not regenerate or judge correct answer.
2. Comparison Method:
   - Analyze the question's requirements and the gold answer's structure.
   - Determine if the question requires exact matching or allows equivalence.
   - Compare ONLY the candidate's final answer. Ignore reasoning errors.
   - Ignore differences in formatting or style.
   - For math expressions: check algebraic equivalence step by step; if uncertain, test numerically at multiple points.
   - For multiple-choice: only compare the final choice and its content.
3. Multi-part Answers:
   - All parts must match the gold answer exactly.
   - Partial matches are incorrect.

- If not specified, answer order may vary. For example, $\frac{27}{7}, -\frac{8}{7}$ and $-\frac{8}{7}, \frac{27}{7}$ are equivalent.

Special considerations:

1. **Mathematical Problems**:
   - If the formats differ but the answers are mathematically equivalent after simplfying or rounding to two decimal places (e.g. 2.909 vs $\frac{32}{11}$, $\frac{32}{11}$ vs $\frac{96}{33}$), respond with [Correct].
   - You only need to verify the correctness of the mathematical expression, not values unrelated to the overall expression, such as the domain or units (e.g., 16 vs 16km, 20 vs 20db), these cases will be considered as [Correct].
   - You may need to calculate the value or convert the value to a different unit when needed to match the reference answer.
2. **Multi-choice questions**:
   - If the question provides explicit candidate answers (e.g., multi-choice questions), the output will be considered correct if it clearly indicates the correct option's content or the correct option's code.
3. **Fact quuestions**:
   - If the question provides fact-seeking answers, the output must align with the correct answer in content to be considered [Correct].
4. **Multiple Reference Answers**:
   - If multiple reference answers are equivalent, just matching one answer will be considered [Correct].
   - If multiple reference answers are inequivalent, only matching all answers will be considered [Correct].
5. **Ohter conditions**:
   - If incomplete (cut off, unfinished sentence) $\rightarrow$ Label as [Incorrect].
   - If repetitive (looping words/phrases) $\rightarrow$ Label as [Incorrect].
   - Gives an answer but then negates it at the end. $\rightarrow$ Label as [Incorrect].
   - Numerically correct but without units. $\rightarrow$ Label as [Correct].

You can use following tools to help your verification process:

1. **Python Interpreter**: When you feel needed, you can use a python interpreter to help you determine your verification result.

2. **Unit Conversion Tool**: When faced with different physical units, you can use a unit conversion tool to convert them into the same unit.

Question: """{question}"""

Output sentence: """{pred}"""

Correct answer: {reference}

Judgement: