

Process Reward Models That Think

Anonymous authors
Paper under double-blind review

Abstract

Step-by-step verifiers—also known as process reward models (PRMs)—are a key ingredient for test-time scaling, but training them requires expensive step-level supervision. This work aims to build data-efficient PRMs as verbalized step-wise reward models that verify every step in the solution by *generating* a verification chain-of-thought (CoT). We propose THINKPRM, a long CoT verifier fine-tuned on orders of magnitude fewer process labels than those required by discriminative PRMs. Our approach capitalizes on the inherent reasoning abilities of long CoT models, and outperforms LLM-as-a-Judge and discriminative verifiers—using only 1% of the process labels in PRM800K—across several challenging benchmarks. Specifically, THINKPRM beats the baselines on ProcessBench, MATH-500, and AIME ’24 under best-of-N selection and reward-guided search. In an out-of-domain evaluation over subsets of GPQA-Diamond and LiveCodeBench, our PRM surpasses discriminative verifiers trained with the full PRM800K by 8% and 4.5%, respectively. Lastly, under the same token budget, THINKPRM scales up verification compute more effectively compared to LLM-as-a-Judge, outperforming it by 7.2% on a subset of ProcessBench. This work highlights the value of generative, long CoT PRMs that can scale test-time compute for verification while requiring minimal supervision for training.

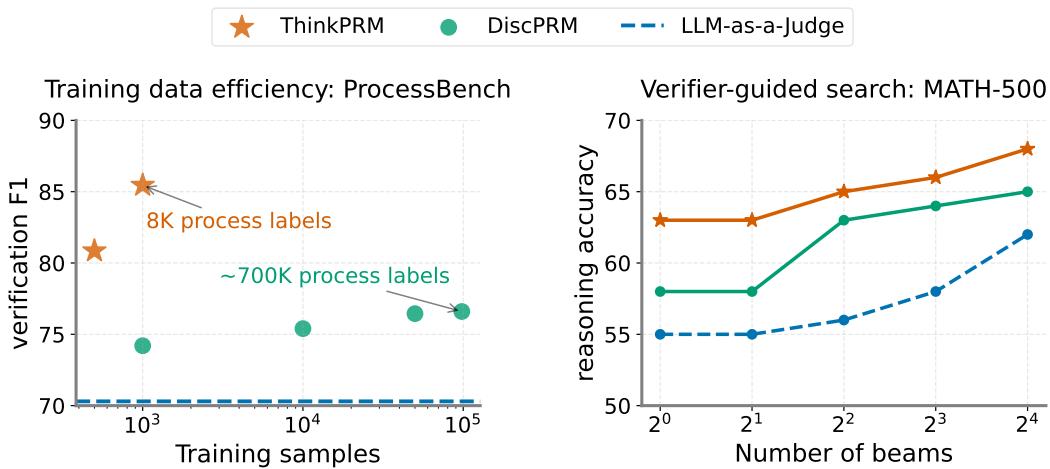


Figure 1: **Left:** Verifier F1-score on ProcessBench (Zheng et al., 2024). THINKPRM-14B, trained on 8K process labels or 1K synthetic examples, outperforms discriminative PRMs trained on about 100x more data. **Right:** Verifier-guided search accuracy on MATH-500 with Llama-3.2-3B-Instruct as generator. THINKPRM-1.5B, trained using the same 8K labels, outperforms LLM-as-a-judge and discriminative verifiers in reward-guided search on MATH-500. The LLM-as-a-judge in both figures uses the same base model as THINKPRM.

1 Introduction

Reasoning with large language models (LLMs) can substantially benefit from utilizing more test-time compute (Jaech et al., 2024; Guo et al., 2025; Akyürek et al., 2024). This typically depends on a high-quality process reward model (PRM)—also known as a process verifier—that scores (partial) solutions for selecting promising

paths for search or ranking (Cobbe et al., 2021; Li et al., 2023; Wu et al., 2024; Brown et al., 2024). PRMs have typically assumed the form of discriminative classifiers, trained to discern correct from incorrect reasoning (Uesato et al., 2022; Zhang et al., 2025). However, training discriminative PRMs requires access to process labels, i.e., step-level annotations, which either require extensive human annotation (Lightman et al., 2023; Zheng et al., 2024), gold step-by-step solutions (Khalifa et al., 2023), or compute-intensive rollouts (Luo et al., 2024; Chen et al., 2024a). For instance, training reasonably performing math PRMs requires hundreds of thousands of step-level annotations (Lightman et al., 2023; Wang et al., 2023b).

Generative verification either via LLM-as-a-judge (Wang et al., 2023a; Liu et al., 2023b; Zheng et al., 2023) or GenRM (Zhang et al., 2024a) treats verification as a generation problem of a rationale followed by a decision. However, LLM-as-a-judge is known to perform poorly compared to specialized reward models (Lambert et al., 2024; Zhang et al., 2024b; Chen et al., 2024c), as general-purpose LLMs frequently fail to recognize reasoning errors (Huang et al., 2023; Zhang et al., 2024a; Ye et al., 2024). Moreover, GenRM is limited to outcome verification via *short* chain-of-thoughts (CoTs), fundamentally limiting its ability for test-time scaling.

This paper builds on the insight that generative step-by-step verification can greatly benefit from scaling up the verifier’s inference compute—specifically, by enabling it to *think* through a CoT. Specifically, we repurpose open-weight large reasoning models (LRMs) as the foundation for generative PRMs through *lightweight* training. This training uses synthetic data (Kim et al., 2023; Zhu et al., 2023; Wang et al., 2024), utilizing as few as 8K step labels, and yielding a THINKPRM—a PRM that not only surpasses LLM-as-a-judge, but also outperforms discriminative PRMs trained on two orders of magnitude more data across a variety of test-time scaling scenarios.

We obtain THINKPRM by training four reasoning models, namely R1-Distill-Qwen{1.5B,7B,14B} (Guo et al., 2025), and QwQ-32B-Preview (Team, 2024), and extensively evaluate it both as a standalone verifier on ProcessBench (Zheng et al., 2024), and combined with a generator under Best-of-N and verifier-guided beam search. THINKPRM-14B outperforms a discriminative PRM based on the same base model in terms of accuracy while using far fewer supervision signals as in Figure 1 left. In addition, THINKPRM-1.5B demonstrates strong performance on MATH-500 (Hendrycks et al., 2021) under guided beam search, shown in Figure 1 right. Lastly, as shown in Figure 2, THINKPRM can effectively utilize more verification compute than LLM-as-a-judge, by forcing it to think for more tokens. All these results are obtained while training only on 8K step labels.

Our work highlights the promise of long CoT PRMs that *verify reasoning with reasoning*, effectively scaling both generator and verifier compute. Our main findings are as follows: THINKPRM outperforms strong PRM baselines in best-of-N and guided-search setups on two math reasoning benchmarks: MATH-500 and AIME 2024, and surpasses LLM-as-a-judge baselines under the same base model by thinking longer during verification (§4). Moreover, THINKPRM generalizes under two types of domain shift. First, it outperforms baselines on out-of-domain tasks such as scientific reasoning and code generation. Second, despite being trained only on short solutions, it generalizes to long-form reasoning without explicit step delimiters (§5.3). Third, THINKPRM outperforms self-consistency (Wang et al., 2022) when using the same compute budget, especially under high sampling regimes (§5.4). Finally, fine-grained filtering of synthetic data based on step supervision is crucial for training high-quality PRMs (§5.7).

2 Background and Related Work

Discriminative PRMs. Discriminative PRMs are trained as classifiers that directly predict numerical correctness scores for each solution step, and typically rely on extensive step-level annotations (Uesato et al., 2022; Lightman et al., 2023; Zhang et al., 2025). Given a solution prefix, discriminative PRMs encode the

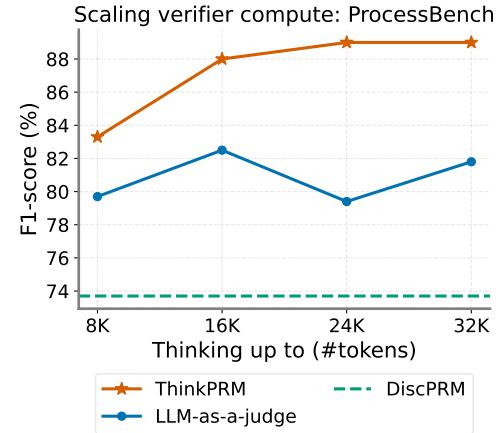


Figure 2: THINKPRM enables scaling verification compute with more CoT tokens.

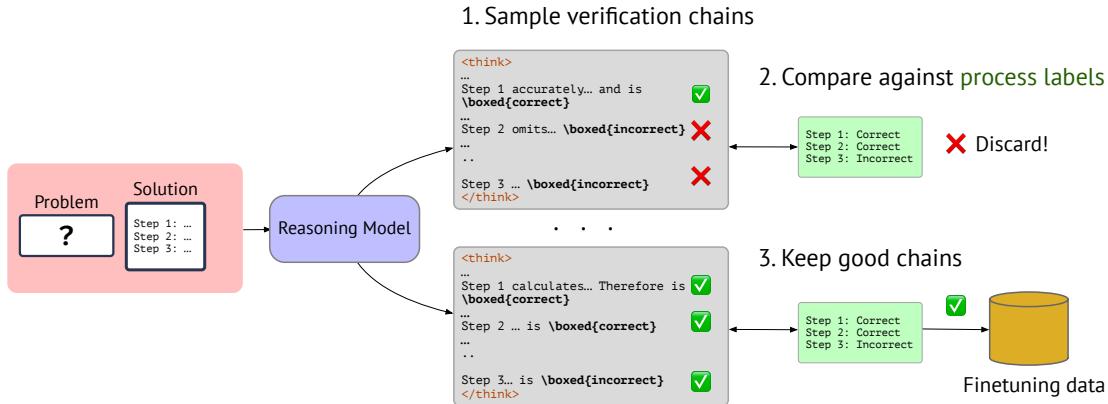


Figure 3: Collecting verification chains for finetuning. First, we prompt a reasoning model, in our case QwQ-32B-Preview to critique a given solution to a problem. Then, we sample multiple verification chains, which we judge against gold process labels from PRM800K, only keeping chains that match the gold process labels.

solution text and employ a classification head to produce step-level scores, usually optimized with binary cross-entropy. An overall correctness score for a solution is obtained by aggregating these step-level scores (Beeching et al.). PRMs are effective and straightforward but they do not utilize the language-modeling head of the base language model, making training expensive and labor-intensive (Yuan et al., 2024). Additionally, they offer limited interpretability and utilize *fixed* compute, restricting their dynamic scalability at test-time (Zhang et al., 2024a; Mahan et al., 2024). Thus, there is a need for data-efficient PRMs that can scale with more test-time compute.

Generative Verification. Generative verification (Zheng et al., 2023; Zhu et al., 2023; Zhang et al., 2024a) frames verification as a language-generation task, producing step-level decisions as tokens (e.g., “correct” or “incorrect”), typically accompanied by a chain-of-thought (CoT). One can train generative verifiers using the standard language modeling objective on verification rationales rather than on binary labels. This approach leverages the strengths of LLMs in text generation, making generative verifiers inherently interpretable and scalable (Zhang et al., 2024a; Mahan et al., 2024; Wang et al., 2023a; Ankner et al., 2024). However, prior work on generative verifiers has relied mainly on short verification CoT (e.g., few hundred tokens) (Zhang et al., 2024a), which highly limits their scalability. Thus, there is a need for verifiers that can “think” longer through verification, utilizing test-time compute effectively. While LLM-as-a-Judge has been employed for step-level verification (Zheng et al., 2024), it tends to be sensitive to prompt phrasing, and prone to invalid outputs, such as infinite looping or excessive overthinking (Bavaresco et al., 2024)—issues we further confirm in this work. Prior results with reasoning models like QwQ-32B-Preview (Team, 2024) show promise, but their practical utility in test-time scaling remains limited without additional training (Zheng et al., 2024).

Test-Time Scaling with PRMs. Test-time scaling techniques, such as Best-of-N selection (Charniak & Johnson, 2005) and tree-based search (Yao et al., 2023; Chen et al., 2024c; Wan et al., 2024), leverage additional inference-time compute to improve reasoning performance. Central to these approaches is the quality of the verifier used to score and select solutions. A major advantage of generative PRMs is that they uniquely support simultaneous scaling of both generator and verifier compute (Zhang et al., 2024a; Kalra & Tang, 2025). In particular, our work shows that generative PRMs trained based on long CoT models (Jaech et al., 2024; Guo et al., 2025) enable both parallel and sequential scaling of verifier compute.

3 ThinkPRM

Our goal is verbalized PRM that, given a problem-solution pair, verifies every step in the solution via an extended chain-of-thought (CoT) such as the one shown in Figure 29 in Appendix G. This section introduces issues with LLM-as-a-judge verification and proposes a data collection process (shown in Figure 3) to curate

high-quality synthetic verification CoTs for training such PRM. The rest of the paper addresses the following research questions:

- **RQ1:** How well do LRMAs perform under LLM-as-a-judge for process-level verification? §3.1
- **RQ2:** Can lightweight finetuning on synthetic verification CoTs improve the reliability and effectiveness of these models as process verifiers? §3.2
- **RQ3:** How does a finetuned verbalized PRM (THINKPRM) compare to discriminative PRMs and LLM-as-a-Judge baselines under different test-time scaling scenarios? §4

3.1 LLM-as-a-judge PRMs are suboptimal

This section highlights limitations we observe when using off-the-shelf reasoning models as process verifiers, suggesting the need for finetuning. For evaluation, we use ProcessBench (Zheng et al., 2024), which includes problem-solution pairs with problems sourced from existing math benchmarks, along with ground-truth correctness labels. We report the binary F1-score by instructing models to verify full solutions and judge whether there exists a mistake. We use two most challenging subsets of ProcessBench: OlympiadBench (He et al., 2024) and OmniMath (Gao et al., 2024), each comprised of 1K problem-prefix pairs. For LLM-as-a-judge, we use the same prompt template as in Zheng et al. (2024), shown in Figure 27, which we found to work best overall. Table 3 shows LLM-as-a-judge F1 scores and a sample output by QwQ-32B-Preview is displayed in Figure 26 in Appendix F.

We observe different issues with LLM-as-a-judge verification. First, the verification quality is *highly sensitive* to the instruction wording: slight change in the instruction can affect the F1-score by up to 3-4 points. First, a substantial number of the generated chains include *invalid judgments*, i.e., chains without an extractable overall label as clear in Figure 5. Such invalid judgements are caused by the following. In some cases, final decision was in the wrong format than instructed e.g., the model tries to *solve* the problem rather than verify the given solution—a behavior likely stemming from the model training. Second, we noted multiple instances of *overthinking* (Chen et al., 2024b; Cuadron et al., 2025), which prevents the model from terminating within the token budget, and *infinite looping/repetitions*, where the model gets stuck trying alternative techniques to verify the solutions.

Figure 4 (left) shows a histogram of verification CoT lengths generated by R1-Qwen-14B in the LLM-as-a-judge setting. Accurate CoTs tend to be shorter, typically under 3K tokens, while inaccurate CoTs are more evenly distributed and spike sharply around 7K-8K tokens, highlighting the prevalence of overthinking and looping in long chains. We show examples of these behaviors in Appendix B. In the next section, we mostly fix these issues via lightweight finetuning over synthetic verification CoTs.

3.2 Finetuning on synthetic data boosts LLM-as-a-judge verification

Inspired by recent work on reducing overthinking in long CoT models that by training (Yu et al., 2024; Kang et al., 2024), we aim to improve LLM-as-a-judge performance via finetuning on high-quality verification data. Collecting real data would be expensive, so we rely on filtered synthetic data (Zelikman et al., 2022; Singh et al., 2023; Dong et al., 2023; Zhang et al., 2024b; Wang et al., 2024) also known as rejection sampling finetuning. To keep our approach simple, we refrain from more expensive training techniques, such as reinforcement learning or preference-based learning.

Synthetic data collection. As training data, we sample synthetic verification CoTs from QwQ-32B-Preview, prompting it to verify each step in a solution prefix, using the instruction shown in Figure 13. The problems and corresponding step-by-step solutions come from the PRM800K dataset (Lightman et al., 2023), which provides both model-generated solutions and human-verified step-level labels.

The sampling process continues until we obtain 1K verification CoTs which correspond to 8K step labels in total. For data filtering, we use the following criteria: (i) the CoT must follow the expected format (i.e., include an extractable decision label for each step inside `\boxed{}`) as shown in Figure 12, and (ii) the generated step judgements match the gold step labels from PRM800K, and (iii) the CoT length is within a maximum budget—to avoid the excessive overthinking behavior we observed in Figure 4 (left). The filtering

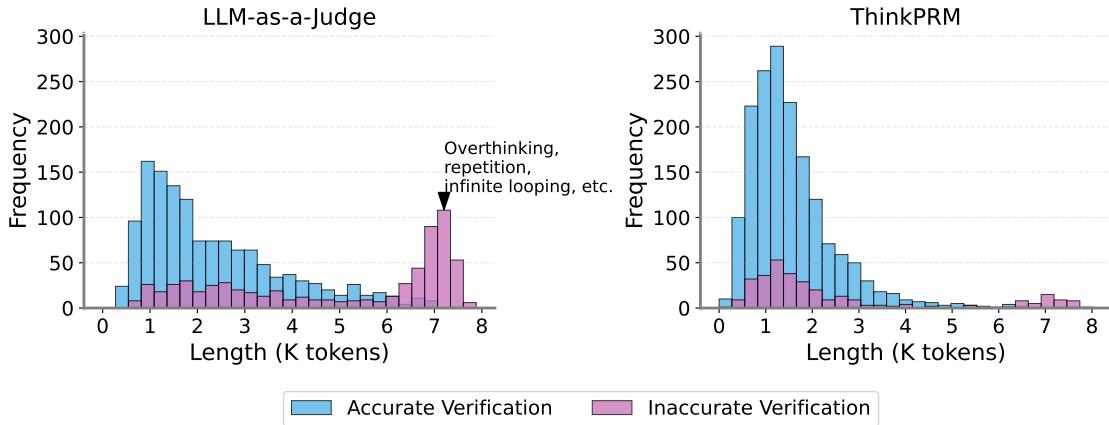


Figure 4: Verifier performance on ProcessBench in light of CoT lengths. On the left, LLM-as-a-judge produces excessively long chains including repetition, infinite looping, and overthinking, leading to worse verifier performance since the output never terminates. Training on collected synthetic data substantially reduces these issues as shown in the THINKPRM plot on the right.

process ensures our training data is of sufficient quality. Note that process-based filtering is crucial for the performance of the resulting PRM as we show in §5.7. We ensure that our 1K synthetic verification traces do not overlap with any problems used in the evaluation datasets (e.g., MATH-500, AIME ’24, GPQA, LiveCodeBench), verified via string match filtering as we detail in ???. Data collection is illustrated in Figure 3, data statistics are in §A.1 and a training example is in Figure 12.

Notably, our filtering relies only on step-level annotations, not on gold verification rationales or CoTs—making this pipeline scalable and low-overhead. In the absence of gold step-level annotations, one can obtain silver labels via Monte Carlo rollouts (Wang et al., 2023b; Chen et al., 2024a). While we train only on math data, the resulting PRM remains robust under other domains such as science QA and code generation as we show in §4.2. We then proceed to train our models on the 1K collected chains. Our training is very lightweight; finetuning QwQ-32B-Preview takes only 4.5 hours on a single A100 80GB GPU. Refer to §C.1 for training details.

Finetuning on synthetic verification CoTs substantially improves the verifier. THINKPRM trains on the 1K chains and is evaluated on ProcessBench and compared to LLM-as-a-judge under the same base model. Figure 6 shows verifier accuracy of different models before and after our finetuning. We note a substantial boost in F1 across all models, with the 1.5B model gaining most improvement by over 70 F1 points, and the 14B model performing best. Looking at the ratio of invalid judgements in Figure 5, we also note a significant reduction in invalid labels with all models, except for QwQ, where it slightly increases. Lastly, the reduction in overthinking and infinite looping behavior discussed in the last section is evident, as in Figure 4 (right), where THINKPRM generations maintain a reasonable length (1K-5K) tokens while being substantially more accurate.

4 Test-time Scaling Experiments

This section aims to answer RQ3 introduced in §3 by comparing THINKPRM to baselines under different scaling scenarios. We study how THINKPRM performs under different generation budgets (i) best-of-N selection (Wu et al., 2024; Brown et al., 2020) and (ii) guided beam search (Snell et al., 2024; Beeching et al.). We also explore how THINKPRM performs when verifier compute is scaled either in parallel by aggregating decisions over multiple verification CoTs or sequentially through longer CoTs by forcing the model to double check or self-correct its verification.

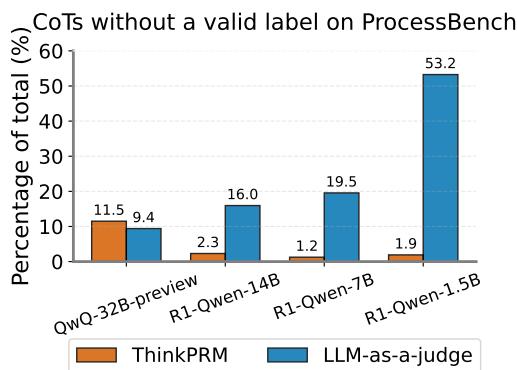


Figure 5: LLM-as-a-judge suffers from a significant ratio of verification CoTs that do not terminate with a parsable label, i.e., `\boxed{yes}` or `\boxed{no}`. Our finetuning process that yields THINKPRM, substantially mitigates this issue. Both verifiers are based on R1-Distill-Qwen-14B.

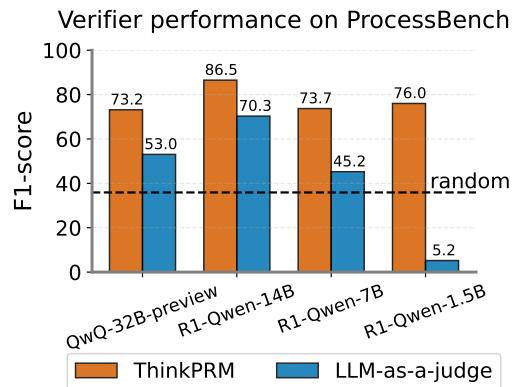


Figure 6: Verification accuracy on 2K question-solution pairs from two most challenging subsets of ProcessBench: OlympiadBench and OmniMath. THINKPRM obtained by finetuning the corresponding model over only 1K verification chains performs better.

4.1 Experimental Setup

In the remainder of the the paper, we will mainly use our finetuned verifiers based on R1-Distill-Qwen-1.5B and R1-Distill-Qwen-14B as these provide the best tradeoff between size and performance. We will refer to these as THINKPRM-1.5B and THINKPRM-14B, respectively.

Baselines. We compare THINKPRM to **DiscPRM**, which uses the same base model as THINKPRM, finetuned with binary cross-entropy on the *entire* PRM800K dataset, totaling 712K process labels, which is two orders of magnitude larger than our training data. Details on finetuning DiscPRMs are in §C.2. We also compare to **unweighted majority voting**, which merely selects the most frequent answer across the samples (Wang et al., 2022), and to LLM-as-a-Judge using the same base model as THINKPRM, prompted as in §3.1.

Tasks and Models. We show results on two math reasoning tasks, namely 100 problems from MATH-500 (Hendrycks et al., 2021) covering all difficulty levels (see §E.5 for more details), and American Invitational Mathematics Examination (AIME) problems for 2024. Since THINKPRM was finetuned only on math data, we study the out-of-domain generalization on two tasks: scientific reasoning and code generation. For scientific reasoning, we use the physics subset of GPQA-Diamond (Rein et al., 2024), consisting of 86 PhD-level multiple choice questions. For code generation, we use a 200-problem subset from the v5 release of LiveCodeBench (Jain et al., 2024).

Over MATH-500, we show results with THINKPRM-1.5B and THINKPRM-14B on two different generator models: Qwen-2.5-14B and Llama-3.2-3B-Instruct. The former model is used for best-of-N and the latter for beam search as search is compute intensive. Showing results with different generators guarantees that our conclusions are not specific to a certain model family or size. For the more challenging tasks, namely AIME '24 and GPQA, we use a more capable model, namely Qwen-2.5-32B-Instruct. For code generation, we use Qwen-2.5-Coder-7B (Hui et al., 2024). Implementation and hyperparameter details on how we select the final answer with best-of-N and beam search are in Appendix E.

Scaling verifier compute. Compared to DiscPRMs, generative reward models enable an extra dimension of scaling to squeeze more performance: scaling the verifier compute. Specifically, THINKPRM allows for two types of scaling. First, we use *parallel scaling* (Mahan et al., 2024; Brown et al., 2024), by sampling K independent CoTs and averaging their scores. We will refer to this scaling using “@K” throughout the rest of the paper. Second, and more specific to long reasoning models, we use *sequential scaling* e.g., by enabling the model to double-check its initial verification (Xiong et al., 2025; Kumar et al., 2024; Ye et al., 2024). Inspired by Muennighoff et al. (2025), we use a trigger phrase such as “*Let’s verify again*” to elicit self-correction of earlier verification. See §E.4 for more details.

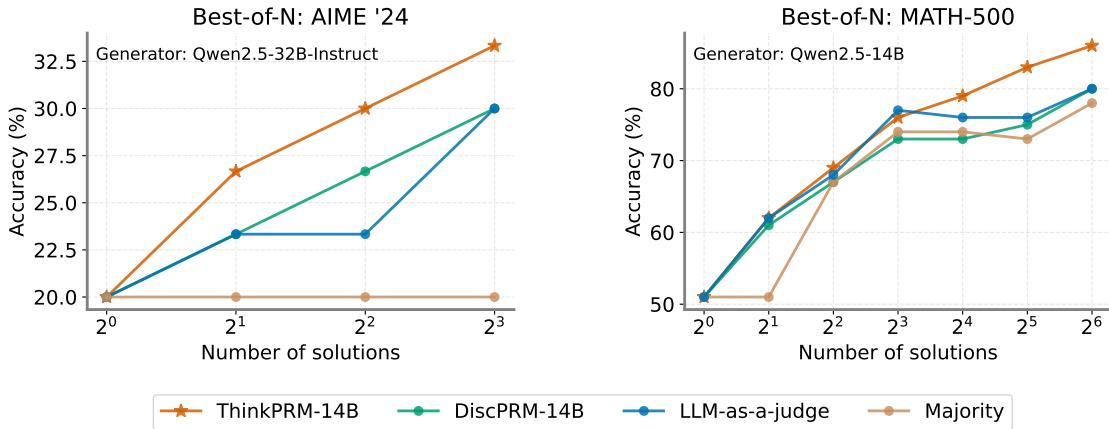


Figure 7: Best-of-N on AIME '24 and MATH-500. Compared to LLM-as-a-judge, DiscPRM, and (unweighted) majority vote, ThinkPRM-14B exhibits best accuracy scaling curve.

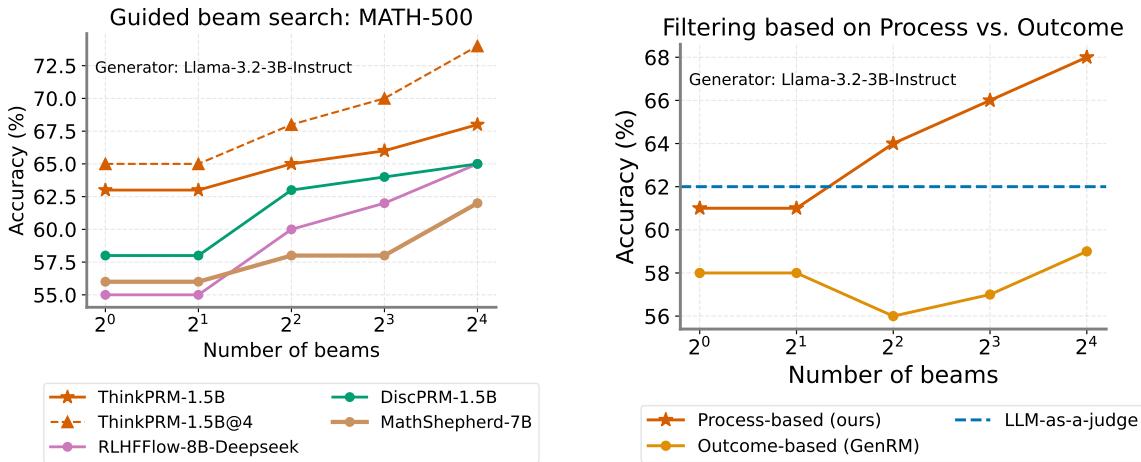


Figure 8: Comparison to off-the-shelf PRMs trained on much more step labels than ThinkPRM. $@K$ represents parallel scaling by averaging scores over K CoTs.

Figure 9: Ablating the data filtering mechanism, where our process-based filtering yields better PRMs. LLM-as-a-judge is shown with number of beams = 16.

4.2 Results

ThinkPRM outperforms DiscPRM and LLM-as-a-Judge. Under best-of-N selection with MATH-500 shown in Figure 7 (right), ThinkPRM leads to higher or comparable reasoning accuracy to DiscPRM under all sampling budgets. The trend holds on the more challenging AIME '24, shown in Figure 7 left. Additionally, Figure 1 (right) shows beam search results on MATH-500, with ThinkPRM 1.5B surpassing DiscPRM and LLM-as-a-Judge.

ThinkPRM surpasses off-the-shelf PRMs. We compare ThinkPRM-1.5B to two strong off-the-shelf PRMs, namely RLHFFlow-Deepseek-PRM (Xiong et al., 2024) and MATH-Shepherd-PRM (Wang et al., 2023b). These PRMs are trained on even more data than PRM800K and are larger than 1.5B. We show results under verifier-guided search on MATH-500 in Figure 8, with ThinkPRM-1.5B's scaling curve surpassing all baselines and outperforming RLHFFlow-Deepseek-PRM, the best off-the-shelf PRM among the ones we tested, by more than 7% across all beam sizes.

ThinkPRM excels on out-of-domain tasks. As for OOD performance on GPQA-physics (Figure 10 left), ThinkPRM scales better than DiscPRM—which drops substantially at $N=32$ —outperforming it by 8%.

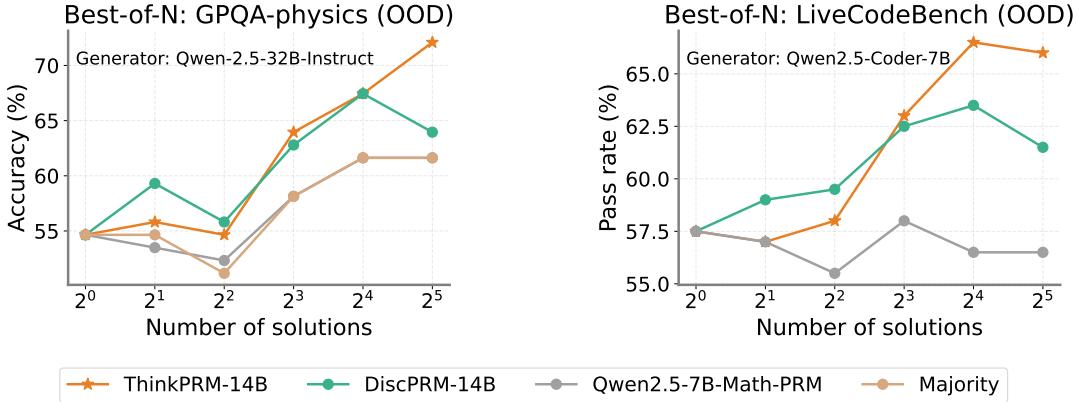


Figure 10: Best-of-N on two out-of-domain tasks: science QA (GPQA-Physics) and code generation (LiveCodeBench). Although THINKPRM was only finetuned on math, it exhibits superior OOD performance than the baselines, especially at larger sampling budgets, where the baselines fall short. Discriminative PRMs struggle despite being trained on orders of magnitude more process labels.

On LiveCodeBench (Figure 10 right), THINKPRM also outperforms DiscPRM by 4.5%. On LiveCodeBench, Qwen2.5-7B-Math-PRM (Zhang et al., 2025)—a discriminative PRM trained on substantial amount of process labels obtained from LLM-as-a-judge data and Monte Carlo rollouts—struggles when applied out-of-domain. Our results shed light on the fragility of discriminative PRMs under domain shifts in contrast with generative PRMs.

Scaling ThinkPRM compute boosts performance. Under verifier-guided search (shown in Figure 8), parallel scaling with THINKPRM-1.5B@4 boosts the accuracy by more than 5% points, and yields the best accuracy on MATH-500. In addition, parallel scaling with THINKPRM-14B@4 and THINKPRM-14B@8 boosts best-of-N performance on MATH-500 as shown in Figure 19 in §E.6. Now we move to sequential scaling of verifier compute by forcing THINKPRM to recheck its own verification. Since this can be compute-intensive, we only run this on 200 problems from OmniMath subset of ProcessBench, and observe how verification F1 improves as we force the model to think for longer as shown in Figure 2. THINKPRM exhibits better scaling behavior compared to LLM-as-a-judge, which drops after 16K tokens, and outperforms DiscPRM-14B by 15 F1 points. In summary, THINKPRM is consistently better than LLM-as-a-judge under parallel and sequential scaling.

Parallel scaling vs. sequential scaling. Is it preferable to scale verifier compute in parallel or sequentially? We investigate this by comparing the two modes of scaling under the same token budget. Figure 20 in §E.6 shows performance of best-of-N with Qwen-2.5-14B under parallel and sequential scaling with $K = 2, 4$ under both parallel scaling and sequential scaling. Overall, the performance of both methods is fairly close, but we observe a slight advantage to parallel scaling under certain budgets.

5 Analysis and Discussion

5.1 Training data efficiency

A major strength of THINKPRM is training data efficiency compared to discriminative versions. Here, we study the training scaling behavior of THINKPRM-14B by training it over 500 and 1K examples in total collected using the pipeline in §3.2, which roughly corresponds to 4K and 8K process labels from PRM800K in total. We compare that to DiscPRM-14B trained with 1K, 10K, 50K and 98K examples, where 98K corresponds to training on the full PRM800K train set that includes 712K step labels. Figure 1 (Left) contrasts the training data scaling behavior of THINKPRM-14B with that of DiscPRM-14B, where THINKPRM-14B’s performance scales substantially better with two orders of magnitude fewer process labels. This primarily stems from THINKPRM’s utilization of text generation and reasoning abilities of the underlying models.

While we train THINKPRM using only 1K data points, we investigate whether it will benefit from training on more data. Using the pipeline, we collect and filter additional verification CoTs and obtain a total of 65K chains. We then finetune R1-Distill-Qwen-1.5B and R1-Distill-Qwen-14B on these for a single epoch while keeping all other training hyperparameters fixed. We then compare the resulting models to the 1K-trained version of THINKPRM under best-of-N selection on MATH-500. Figures 21 and 22 in §E.7 show a performance boost from training on the 65K examples compared to only 1K. This suggests that THINKPRM can utilize more training data when available.

5.2 Effect of Verification CoT Length on PRM Quality

We study whether the length of verification chains of thought affects the quality of the resulting generative verifier. Specifically, we compare ThinkPRM trained on the full, long synthetic CoTs with a variant trained on short, compressed versions of the same 1K CoTs. To obtain the short CoTs, we instruct `gpt-4o-mini` to rewrite each original CoT into a concise version that preserves only the essential reasoning. We then train R1-Qwen-1.5B and R1-Qwen-14B on these short CoTs and evaluate verification F1 on ProcessBench. Table 1 reports the comparison.

Model	Long CoT (THINKPRM)		Short CoT	
	OlympiadBench	OmniMath	OlympiadBench	OmniMath
R1-Qwen-1.5B	87.3	75.7	64.8	66.7
R1-Qwen-14B	87.3	85.7	55.3	60.8

Table 1: Verification F1 when training R1 models on long versus short CoTs.

The substantial performance drop when training on short CoTs emphasizes how THINKPRM benefits from extended reasoning. Since verification is a complex task, throwing more reasoning effort at it via thinking improves performance. These results support the value of using long verification CoTs for training.

5.3 Reasoning traces without clear step boundaries

So far, we have used THINKPRM to verify short CoTs with clear steps delimiters. Here, we investigate whether THINKPRM can still verify long CoTs that involve extended reasoning, backtracking, and self-correction. As a generator, we use Qwen3-1.7B (Yang et al., 2025) with thinking mode. Although THINKPRM was only trained on short solutions from PRM800K, we find that it can still verify long CoTs and outperforms the baselines as shown in Figure 11 left. Inspecting THINKPRM’s outputs, we found that it extracts and verifies individual steps embedded in the long CoT—an example is in Figure 30.

5.4 Compute-matched comparison to self-consistency

Under a fixed test-time compute budget for best-of-N, how does THINKPRM compare to simply sampling more solutions from the generator and applying majority voting? To investigate this, we conduct a compute-matched analysis on MATH-500 and GPQA-Physics. Figure 11 mid and right plot solution accuracy as a function of sampling FLOPs for MATH-500 and GPQA-physics. At low sampling budgets, best-of-N with THINKPRM performs comparably to self-consistency, but as the compute budget increases, THINKPRM has a clear advantage. These findings agree with recent work on outcome reward models (Singhi et al., 2025).

5.5 ThinkPRM with Monte Carlo step labels

To train THINKPRM, we have relied on manual step labels from PRM800K. Since automatic labels e.g., via Monte Carlo rollouts Luo et al. (2024) are cheaper, we validate whether we can train THINKPRM using automatic labels. We train THINKPRM-1.5B using 1K synthetic chains based on labels from Math-shepherd dataset (Wang et al., 2023b). Performance on ProcessBench is shown in Table 5, where training THINKPRM with automatic labels yields very comparable performance to training with manual labels, showing that our training pipeline is agnostic to step-labeling strategy.

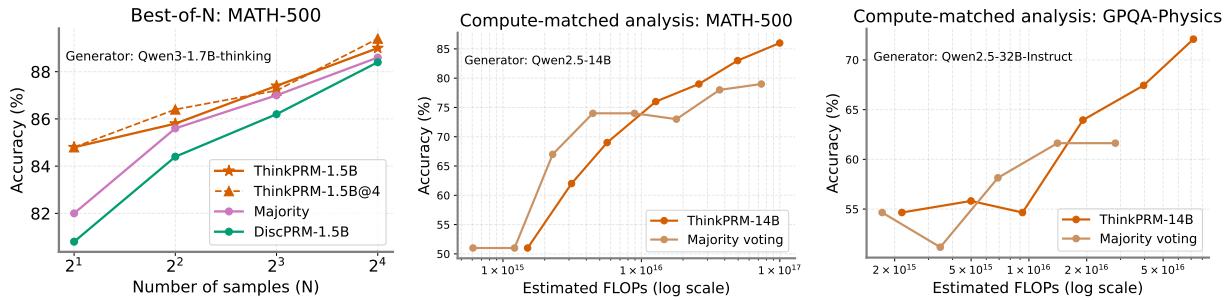


Figure 11: **Left:** Best-of-N with Qwen3-1.7B on the full MATH-500 test set, showing how THINKPRM generalizes well to verifying long reasoning traces. **Mid and Right:** Compute-matched comparison between best-of-N with THINKPRM and self-consistency or majority vote.

5.6 ThinkPRM helps with difficult reasoning problems

THINKPRM’s reasoning ability should enable it to tackle verification of hard problems. To check if this is the case, we analyze performance of THINKPRM vs. DiscPRM in light of problem difficulty over MATH-500 and GPQA-physics (how we estimate difficulty for GPQA-Physics is explained in §E.11), shown in Figure 24. The generators here are Qwen-2.5-14B for MATH-500 and Qwen-2.5-32B-Instruct for GPQA-Physics. Primarily, THINKPRM improves reasoning on the *difficult* problems (levels 3, 4, 5 in MATH-500 and 2, 3, 4 in GPQA-Physics) substantially more than DiscPRM.

5.7 Filtering based on process vs. outcome labels

In §3.2, we describe our process-based filtering strategy, which selects verification CoTs based on agreement between generated step-level decisions and gold process labels. To validate its effectiveness, we compare it to outcome-based filtering, as in GenRM (Zheng et al., 2024), which retains chains solely based on final answer correctness—keeping a CoT if its final answer is correct and the final step is `\boxed{correct}`, or if the answer is incorrect and the final step is `\boxed{incorrect}`, thereby ignoring intermediate step labels. We obtain 65K and 128K CoTs using process- and outcome-based filtering, respectively. Figure 9 shows that finetuning R1-Distill-Qwen-1.5B on process-filtered data yields significantly better verification performance, despite using fewer examples, which reflects the importance of our process-based filtering in training strong PRMs.

5.8 Limitations of Generative PRMs

While generative PRMs are more powerful and data-efficient than their discriminative counterparts, they come with some limitations that we highlight as avenues for future work. First, overconfidence is a known issue in LLMs (Liu et al., 2023a; Stechly et al., 2023; Zhou et al., 2024) and, in the case of PRMs, it can cause the predicted PRM scores to cluster near extremes: close to either 0 or 1. One reason is that we are using probabilities of certain tokens such as “yes” or “no”, which by nature will be either very high or very low. Future work should explore more reliable techniques to extract calibrated scores from generative reward models. Another limitation is due to autoregressive nature of LLMs, leading them to prematurely commit to an earlier judgment. For example, we observe a phenomenon we term *step label interference*, where verification errors for earlier steps impact verification of later steps. For example, if the PRM judges a particular step as incorrect, it becomes more likely to label subsequent steps as incorrect even if it is not.

Conclusion

We introduced THINKPRM, a generative process reward model trained with minimal synthetic supervision for scalable step-by-step verification. With just 8K process labels, THINKPRM significantly outperforms LLM-as-a-judge and even surpasses discriminative PRMs trained on orders of magnitude more data. These results highlight the benefits of generative PRMs in interpretability, scalability, and data efficiency.

References

Ekin Akyürek, Mehul Damani, Linlu Qiu, Han Guo, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for abstract reasoning. *arXiv preprint arXiv:2411.07279*, 2024.

Zachary Ankner, Mansheej Paul, Brandon Cui, Jonathan D Chang, and Prithviraj Ammanabrolu. Critique-out-loud reward models. *arXiv preprint arXiv:2408.11791*, 2024.

Anna Bavaresco, Raffaella Bernardi, Leonardo Bertolazzi, Desmond Elliott, Raquel Fernández, Albert Gatt, Esam Ghaleb, Mario Julianelli, Michael Hanna, Alexander Koller, et al. Llms instead of human judges? a large scale empirical study across 20 nlp evaluation tasks. *arXiv preprint arXiv:2406.18403*, 2024.

Edward Beeching, Lewis Tunstall, and Sasha Rush. Scaling test-time compute with open models. URL <https://huggingface.co/spaces/HuggingFaceH4/blogpost-scaling-test-time-compute>.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL '05)*, pp. 173–180, 2005.

Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. Alphamath almost zero: process supervision without process. *arXiv preprint arXiv:2405.03553*, 2024a.

Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuwei Liu, Mengfei Zhou, Zhiheng Zhang, et al. Do not think that much for $2+3=?$ on the overthinking of o1-like llms. *arXiv preprint arXiv:2412.21187*, 2024b.

Ziru Chen, Michael White, Raymond Mooney, Ali Payani, Yu Su, and Huan Sun. When is tree search useful for llm planning? it depends on the discriminator. *arXiv preprint arXiv:2402.10890*, 2024c.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Alejandro Cuadron, Dacheng Li, Wenjie Ma, Xingyao Wang, Yichuan Wang, Siyuan Zhuang, Shu Liu, Luis Gaspar Schroeder, Tian Xia, Huanzhi Mao, et al. The danger of overthinking: Examining the reasoning-action dilemma in agentic tasks. *arXiv preprint arXiv:2502.08235*, 2025.

Hanze Dong, Wei Xiong, Deepanshu Goyal, Yihan Zhang, Winnie Chow, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and Tong Zhang. Raft: Reward ranked finetuning for generative foundation model alignment. *arXiv preprint arXiv:2304.06767*, 2023.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, et al. Omni-math: A universal olympiad level mathematic benchmark for large language models. *arXiv preprint arXiv:2410.07985*, 2024.

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

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

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

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

Nimit Kalra and Leonard Tang. Verdict: A library for scaling judge-time compute. *arXiv preprint arXiv:2502.18018*, 2025.

Yu Kang, Xianghui Sun, Liangyu Chen, and Wei Zou. C3ot: Generating shorter chain-of-thought without compromising effectiveness. *arXiv preprint arXiv:2412.11664*, 2024.

Muhammad Khalifa, Lajanugen Logeswaran, Moontae Lee, Honglak Lee, and Lu Wang. GRACE: discriminator-guided chain-of-thought reasoning. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pp. 15299–15328. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-EMNLP.1022. URL <https://doi.org/10.18653/v1/2023.findings-emnlp.1022>.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, et al. Prometheus: Inducing fine-grained evaluation capability in language models. In *The Twelfth International Conference on Learning Representations*, 2023.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, et al. Training language models to self-correct via reinforcement learning. *arXiv preprint arXiv:2409.12917*, 2024.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, et al. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*, 2024.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making language models better reasoners with step-aware verifier. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5315–5333, 2023.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Xin Liu, Muhammad Khalifa, and Lu Wang. Litcab: Lightweight language model calibration over short-and long-form responses. *arXiv preprint arXiv:2310.19208*, 2023a.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*, 2023b.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, et al. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*, 2, 2024.

Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, Chelsea Finn, and Alon Albalak. Generative reward models. *arXiv preprint arXiv:2410.12832*, 2024.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

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

Nishad Singhi, Hritik Bansal, Arian Hosseini, Aditya Grover, Kai-Wei Chang, Marcus Rohrbach, and Anna Rohrbach. When to solve, when to verify: Compute-optimal problem solving and generative verification for llm reasoning. *arXiv preprint arXiv:2504.01005*, 2025.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. Gpt-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. *arXiv preprint arXiv:2310.12397*, 2023.

Zhiqing Sun, Longhui Yu, Yikang Shen, Weiyang Liu, Yiming Yang, Sean Welleck, and Chuang Gan. Easy-to-hard generalization: Scalable alignment beyond human supervision. *arXiv preprint arXiv:2403.09472*, 2024.

Qwen Team. Qwq: Reflect deeply on the boundaries of the unknown, November 2024. URL <https://qwenlm.github.io/blog/qwq-32b-preview/>.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

Ziyu Wan, Xidong Feng, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. In *Forty-first International Conference on Machine Learning*, 2024.

Jiaan Wang, Yunlong Liang, Fandong Meng, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. Is chatgpt a good nlg evaluator? a preliminary study. *arXiv preprint arXiv:2303.04048*, 2023a.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Y Wu, and Zhifang Sui. Math-shepherd: A label-free step-by-step verifier for llms in mathematical reasoning. *arXiv preprint arXiv:2312.08935*, 2023b.

Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. Self-taught evaluators. *arXiv preprint arXiv:2408.02666*, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*, 2024.

Wei Xiong, Hanning Zhang, Nan Jiang, and Tong Zhang. An implementation of generative prm. <https://github.com/RLHFflow/RLHF-Reward-Modeling>, 2024.

Wei Xiong, Hanning Zhang, Chenlu Ye, Lichang Chen, Nan Jiang, and Tong Zhang. Self-rewarding correction for mathematical reasoning. *arXiv preprint arXiv:2502.19613*, 2025.

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Tian Ye, Zicheng Xu, Yuanzhi Li, and Zeyuan Allen-Zhu. Physics of language models: Part 2.2, how to learn from mistakes on grade-school math problems. *arXiv preprint arXiv:2408.16293*, 2024.

Ping Yu, Jing Xu, Jason Weston, and Ilia Kulikov. Distilling system 2 into system 1. *arXiv preprint arXiv:2407.06023*, 2024.

Lifan Yuan, Wendi Li, Huayu Chen, Ganqu Cui, Ning Ding, Kaiyan Zhang, Bowen Zhou, Zhiyuan Liu, and Hao Peng. Free process rewards without process labels. *arXiv preprint arXiv:2412.01981*, 2024.

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

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*, 2024a.

Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. Small language models need strong verifiers to self-correct reasoning. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pp. 15637–15653. Association for Computational Linguistics, 2024b. doi: 10.18653/V1/2024.FINDINGS-ACL.924. URL <https://doi.org/10.18653/v1/2024.findings-acl.924>.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. *arXiv preprint arXiv:2501.07301*, 2025.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Processbench: Identifying process errors in mathematical reasoning. *arXiv preprint arXiv:2412.06559*, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

Kaitlyn Zhou, Jena D Hwang, Xiang Ren, and Maarten Sap. Relying on the unreliable: The impact of language models' reluctance to express uncertainty. *arXiv preprint arXiv:2401.06730*, 2024.

Lianghui Zhu, Xinggang Wang, and Xinlong Wang. Judgelm: Fine-tuned large language models are scalable judges. *arXiv preprint arXiv:2310.17631*, 2023.

Synthetic training sample

Problem: Find the area in square feet of a square with a perimeter of 32ft.

Solution:

Step 1: A square has four equal sides, so the perimeter is four times the length of one side.
 Step 2: I can write an equation to relate the perimeter and the side length: $4s = 32$, where s is the side length in feet.
 Step 3: To solve for s , I can divide both sides by 4: $s = 32/4 = 8$.
 Step 4: Now I know the side length of the square is 8 feet.
 Step 5: The area of a square is the side length squared, so the area is $s^2 = 8^2 = 64$.
 Step 6: The area is in square feet, so the units are ft^2 .
 Step 7: The answer is 64 ft^2

<think>
 Alright, let's verify step by step the solution provided for finding the area of a square with a perimeter of 32 feet.

Step 1: "A square has four equal sides, so the perimeter is four times the length of one side."
 Critique: This statement is accurate. A square indeed has four sides of equal length, and the perimeter is the sum of all sides. Therefore, perimeter (P) equals 4 times the side length (s), or $P = 4s$.
 The step is $\boxed{\text{correct}}$

Step 2: "I can write an equation to relate the perimeter and the side length: $4s = 32$, where s is the side length in feet."
 Critique: Given that the perimeter is 32 feet, setting up the equation $4s = 32$ is correct. It properly represents the relationship between the perimeter and the side length of the square.
 The step is $\boxed{\text{correct}}$

Step 3: "To solve for s , I can divide both sides by 4: $s = 32/4 = 8$."
 Critique: This step correctly solves for s by dividing both sides of the equation by 4. The calculation 32 divided by 4 equals 8 is accurate.
 The step is $\boxed{\text{correct}}$

Step 4: "Now I know the side length of the square is 8 feet."
 Critique: Based on the previous step, $s = 8$ feet, which is correctly stated here.
 The step is $\boxed{\text{correct}}$

Step 5: "The area of a square is the side length squared, so the area is $s^2 = 8^2 = 64$."
 Critique: The formula for the area of a square is indeed s squared. Calculating 8 squared equals 64 is correct.
 The step is $\boxed{\text{correct}}$

Step 6: "The area is in square feet, so the units are ft^2 ."
 Critique: Since the side length is in feet, the area will be in square feet (ft^2). This is the correct unit for area.
 The step is $\boxed{\text{correct}}$

Step 7: "The answer is 64 ft^2 "
 Critique: This is the final answer, correctly stating the area as 64 square feet.
 The step is $\boxed{\text{correct}}$

</think>
 Is the solution correct? Yes

Figure 12: Training example from our sampled 1K synthetic chains.

A Training data

A.1 Sampling

Figure 3 shows the prompt used with QwQ-32B-Preview to sample verification chains for training. We use the problem-prefix pairs from PRM800K train split (Lightman et al., 2023), which is based on MATH (Hendrycks et al., 2021). We sample 4 verification chains for each prefix in the dataset with temperature $T = 0.1$ until we have 1K verification CoTs that (1) follow the desired format i.e., the model generated an extractable label for every step in the prefix and (2) the verbalized step labels match the gold labels in the PRM800K dataset annotation, (3) at most 4096 tokens long, and are (4) roughly balanced with respect to final answer correctness.

Roughly 20% of the initial samples satisfied criteria 1 and 2 in our case, which means that about 5K samples in total are needed to obtain 1K filtered samples. Table 2 shows the statistics of the final dataset. We also show an example from the 1K training CoTs in Figure 12.

A.2 Preprocessing

From the sampled verification CoTs, we extract boxed predictions (e.g., `\boxed{correct}` or `\boxed{incorrect}`) for each reasoning step. Then we filter the data as follows:

- CoTs with missing or malformed verification decisions are discarded.
- CoTs where verification decisions don't match ground truth labels are filtered out. Precisely, we require all step-by-step verification decisions to match the ground truth labels.

Then we clean the data as follows:

- We standardize boxed notation formats.
- Content after the last verification decision is removed.
- Special tokens (e.g., `<think>` and `</think>`) are added to designate verification reasoning.

A.3 Contamination Control

To avoid test data leakage, we ensured that none of the 1K verification traces used for training THINKPRM overlap with the test problems in MATH-500, AIME '24, ProcessBench (OlympiadBench; OmniMath), GPQA-Physics, or LiveCodeBench. We applied exact-match between PRM800K problem texts and each evaluation set. We also verified that AIME '24 problems are not present in PRM800K, which only includes pre-2023 math problems.

Correctness Distribution	
# correct	486 (48.6%)
# incorrect	514 (51.4%)
Step label distribution	
# correct	7474 (92.3%)
# Incorrect	625 (7.7%)
Prefix statistics	
# unique questions	869
average steps per prefix	9.5
min # steps	1
max # steps	53
Verification Chain Statistics	
Average verification chain length (tokens)	1037.0
Min verification chain length (tokens)	207
Max verification chain length (tokens)	3669

Table 2: Statistics of the 1K finetuning dataset.

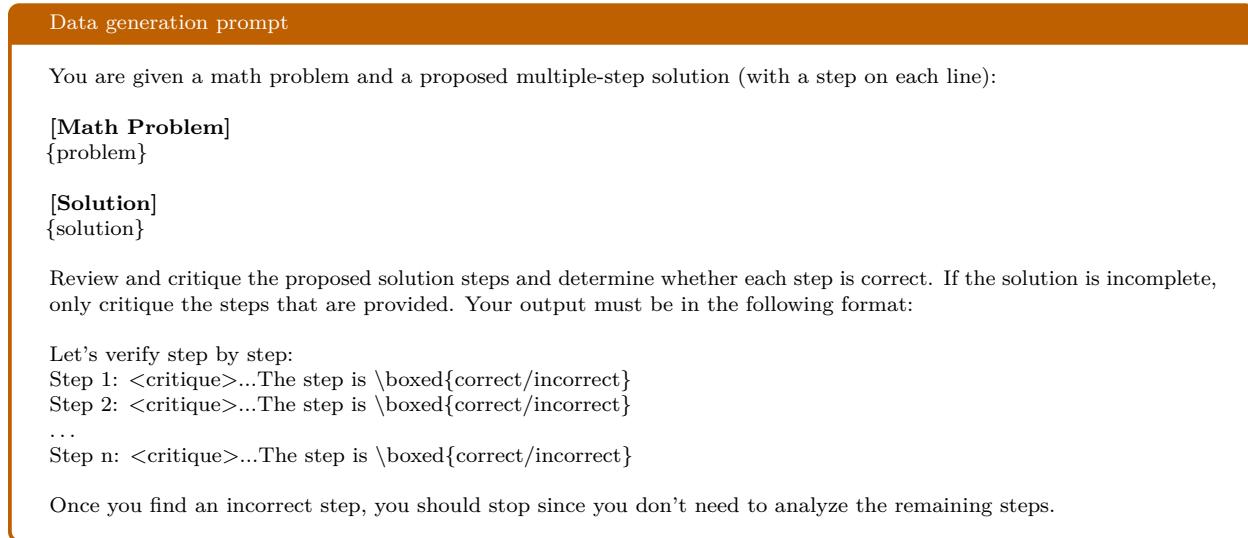


Figure 13: Prompt used to generate synthetic verification chains as described in §3.2.

B LLM-as-a-Judge Limitations

We show examples of failures from QwQ-32B-Preview and Deepseek-R1-Distill-Qwen-14B when used for LLM-as-judge verification in Figures 15 and 17.

C Training details

C.1 Training ThinkPRM

We train our models on the filtered 1K chains using LoRA (Hu et al., 2022) with rank $r = 32$ and $\alpha = 16$. Specifically, we finetune Qwen-14B-Instruct and R1-Deepseek (14B) for 3 epochs, and QwQ-32B-Preview for 5 epochs using LoRA. We use an effective batch size of 16 and a fixed learning rate of 4×10^{-4} without warmup is used. Training took 1.5 hours for the 14B models and 4.5 hours for QwQ-32B-Preview on a single A100 80GB GPU. Without particularly found QwQ to be hard to train with LoRA and still generates a relatively high percentage of invalid judgments after training. Full training of the model will likely resolve these issues but that would require more compute than we have.

The R1-Distill-Qwen{1.5B,7B} models use *full* training with the following parameters. The 1.5B model We trained for 3 epochs with an effective batch size of 32, using a constant learning rate of 6×10^{-5} without decay or warmup. We train both models using four RTX A6000 48GB GPU using data parallel. Training the 1.5B model on the 1K chains took about half an hour and the 7B model about two hours.

C.2 Training Discriminative Verifiers

We train R1-Qwen-14B for 1 epoch over the entire PRM800K dataset using two A100 80GB GPUs with a batch size of 8 and a learning rate of 6×10^{-5} . We use a constant learning rate scheduler with no warmup. Following prior work (Wang et al., 2023b; Zhang et al., 2025) We train the model using binary cross-entropy to maximize the probability of the tokens "+" and "-" for correct and incorrect steps, respectively. The R1-Qwen-1.5B model is trained with the same infrastructure with a batch size of 64 and a learning rate of 1×10^{-4} with a warm up ratio of 10%.

QwQ-32B-Preview stuck in a loop

Okay, so I need to analyze each step of the proposed solution to the math problem. The problem is: If $f(x) = \frac{3x-2}{x-2}$, what is the value of $f(-2) + f(-1) + f(0)$? The proposed solution is given, and I have to check if each step is correct.

First, let me read through the problem and the solution carefully.

The function is $f(x) = \frac{3x-2}{x-2}$. They want the sum of $f(-2) + f(-1) + f(0)$.

Looking at the solution steps:

Step 0: They plug in $x = -2$ into the function and calculate $\frac{3(-2)-2}{-2-2} = \frac{-8}{-4} = 2$.

Step 1: They mention that plugging $x = -2$ into the denominator gives zero, which would be division by zero.

Step 2: They generalize that we should avoid plugging in values that make the denominator zero.

Step 3: Similarly, plugging $x = -1$ into the denominator also gives zero.

Step 4: Plugging $x = 0$ into the denominator also results in division by zero.

...

Wait no! The original function is given as $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

No! The problem says: If $f(x) = \frac{\text{numerator}}{\text{denominator}} = ?$

...

Figure 14: Degeneration behavior from QwQ-32B-Preview with LLM-as-judge verification. Problem and solution are omitted for brevity.

Deepseek-R1-Distill-Qwen-14B stuck in a loop

Okay, so I need to review and critique the solution step by step for the given math problem. The problem is to find the exact value of the continued fraction expression:

$1 + 1/(1 + 2/(1 + 1/(1 + 2/(1 + \dots))))$

The solution provided is split into 16 steps, each enclosed in a tag. I need to go through each step, analyze it, and see if there are any errors or if the logic holds. After that, I'll determine if the solution is correct by outputting "yes" or "no" in a box.

Let me start by understanding the problem. It's a continued fraction that alternates between 1 and 2 in the numerators. The expression is:

$x = 1 + 1/(1 + 2/(1 + 1/(1 + 2/(1 + \dots))))$

So, the pattern repeats every two steps: 1, then 2, then 1, then 2, etc. The solution starts by denoting the expression as x and then tries to find a relationship by substituting the inner expression back into x .

Wait, the original expression is $x = 1 + 1/(1 + 2/(1 + 1/(1 + 2/(1 + \dots))))$. So, the part after the first 1 is $1 + 2/(1 + 1/(1 + 2/(1 + \dots)))$. But that part isn't x , because x starts with $1 + 1/(1 + \dots)$. So, maybe the substitution isn't directly x . Let me think. Wait, perhaps the inner part is similar but not exactly x . Let me denote the inner part as y . So, $y = 1 + 2/(1 + 1/(1 + 2/(1 + \dots)))$. But then, the expression for y would involve another similar structure. Hmm, maybe the solution is oversimplifying here.

...

Wait, actually, when you multiply both sides by $(1 + 2/x)$, the left side is $x*(1 + 2/x) = x + 2$.

The right side is $[1 + 1/(1 + 2/x)]*(1 + 2/x) = 1*(1 + 2/x) + [1/(1 + 2/x)]*(1 + 2/x) = (1 + 2/x) + 1 = 2 + 2/x$.

So, the correct equation after multiplying both sides by $(1 + 2/x)$ is:

$x + 2 = 2 + 2/x$.

But in step 6, they have:

$x + 2 = 1 + 1/(1 + 2/x) + 2/x + 4/x^2$

Which seems to be incorrect. They seem to have distributed the multiplication incorrectly.

Wait, let me see. If you have $x*(1 + 2/x) = [1 + 1/(1 + 2/x)]*(1 + 2/x) + [2/x*(1 + 2/x)]$

But that doesn't make sense because when you multiply both sides by $(1 + 2/x)$, you should only have:

Left side: $x*(1 + 2/x)$

Right side: $[1 + 1/(1 + 2/x)]*(1 + 2/x)$

Which simplifies to $x + 2 = 2 + 2/x$.

But in the solution, step 6 says:

$x*(1 + 2/x) = 1 + 1/(1 + 2/x) + 2/x*(1 + 2/x)$

Which seems to be adding an extra term, $2/x*(1 + 2/x)$, which is not correct. So, step 6 is incorrect.

Therefore, the solution has an error in step 6. Instead of correctly multiplying both sides, they incorrectly expanded the right-hand side, leading to an incorrect equation.

Because of this mistake, the subsequent steps are likely incorrect as well. So, the entire solution is flawed from step 6 onwards.

...

Wait, let me compute the continued fraction numerically to check.

Let me compute the continued fraction up to a certain depth.

Let me compute it step by step:

Start with the innermost part and work outwards.

...

So, it seems to converge to approximately 2.414, which is $1 + \sqrt{2} \approx 2.4142$.

Wait, but according to my earlier solving, x should be $\sqrt{2} \approx 1.414$, but the numerical computation suggests it converges to approximately 2.414, which is $1 + \sqrt{2}$. So, which one is correct?

Wait, perhaps I made a mistake in setting up the equation.

Wait, let's re-examine the original expression:

...

But when I tried to solve it earlier, I set $x = 1 + 1/(1 + 2/x)$, which led to $x = \sqrt{2}$. But the numerical computation suggests it converges to $1 + \sqrt{2}$. So, perhaps my initial substitution was incorrect.

Wait, perhaps I need to define it differently. Let me try to define the continued fraction properly.

Wait, let's solve the system:

From $x = 1 + 1/y$ and $y = 1 + 2/x$.

Substitute $y = 1 + 2/x$ into $x = 1 + 1/y$:

$x = 1 + 1/(1 + 2/x)$

Multiply both sides by $(1 + 2/x)$:

...

Alternatively, perhaps I need to define y differently. Let me try to define y as the expression starting with $1 + 1/(1 + \dots)$:

$y = 1 + 1/(1 + 2/(1 + 1/(1 + 2/(1 + \dots))))$

But then, $y = x$, which brings us back to the original substitution.

...

Which is the same equation as before, leading to $x = \sqrt{2}$. But the numerical computation suggests it's approaching $1 + \sqrt{2}$. So, perhaps my initial substitution is incorrect.

Alternatively, perhaps the continued fraction is structured such that after the first $1 + 1/(1 + \dots)$, the denominator is another continued fraction similar to x but shifted.

...

Figure 15: Degeneration behavior from Deepseek-R1-Distill-Qwen-14B with LLM-as-judge verification. Problem and solution are omitted for brevity.

Problem and solution from ProcessBench

Problem: Let a, b be positive reals with $a > b > \frac{1}{2}a$. Place two squares of side lengths a, b next to each other, such that the larger square has lower left corner at $(0, 0)$ and the smaller square has lower left corner at $(a, 0)$. Draw the line passing through $(0, a)$ and $(a + b, 0)$. The region in the two squares lying above the line has area 2013. If (a, b) is the unique pair maximizing $a + b$, compute $\frac{a}{b}$.

Solution:

We begin by finding the equation of the line through $(0, a)$ and $(a + b, 0)$. The slope is:

$$m = \frac{0 - a}{(a + b) - 0} = -\frac{a}{a + b}$$

Using point-slope form from $(0, a)$:

$$y = a - \frac{a}{a + b} x$$

Area above the line in the larger square:

At $x = a$, the height is:

$$y = a - \frac{a^2}{a + b} = \frac{ab}{a + b}$$

So the area is a trapezoid:

$$\text{Area}_1 = \frac{1}{2} \left(a + \frac{ab}{a + b} \right) a = \frac{a^3 + 2a^2b}{2(a + b)}$$

Area above the line in the smaller square:

This is a triangle with height $\frac{ab}{a + b}$ and base b :

$$\text{Area}_2 = \frac{1}{2} \cdot \frac{ab}{a + b} \cdot b = \frac{ab^2}{2(a + b)}$$

Total Area:

$$\text{Total} = \text{Area}_1 + \text{Area}_2 = \frac{a^3 + 2a^2b + ab^2}{2(a + b)} = \frac{a(a^2 + 2ab + b^2)}{2(a + b)} = \frac{a(a + b)^2}{2(a + b)} = \frac{a(a + b)}{2}$$

We are given:

$$\frac{a(a + b)}{2} = 2013 \Rightarrow a(a + b) = 4026$$

Express b in terms of a :

$$b = \frac{4026}{a} - a$$

Apply the constraints $a > b > \frac{1}{2}a$:

- $a > b \Rightarrow a > \frac{4026}{a} - a \Rightarrow a^2 > 2013 \Rightarrow a > \sqrt{2013}$
- $b > \frac{1}{2}a \Rightarrow \frac{4026}{a} - a > \frac{1}{2}a \Rightarrow a^2 < 2684 \Rightarrow a < \sqrt{2684}$

So $\sqrt{2013} < a < \sqrt{2684}$. To maximize $a + b = \frac{4026}{a}$, we minimize a , so let $a = \sqrt{2013}$. Then $b = \sqrt{2013}$, satisfying the constraints.

$$\frac{a}{b} = \boxed{2}$$

Figure 16: An example of problem and solution from OmniMath subset of ProcessBench.

Incorrect verification

Okay, so I need to review and critique the given solution step by step. Let me go through each step carefully.

Step 0: The solution starts by identifying key points and properties. It mentions that J is the excenter opposite to A , which is correct. A_1 is where the excircle touches BC , and B_1, C_1 are where it touches the extensions of AC and AB . Then, it says A_1B_1 is perpendicular to AB and intersects at D . E is the foot of the perpendicular from C_1 to DJ .

Hmm, this seems okay, but I wonder if they've correctly identified all the properties. Maybe they should have mentioned more about the excenter's properties, like the fact that it's the intersection of external angle bisectors. But maybe that's assumed knowledge.

Step 1: They note that since A_1B_1 is perpendicular to AB , D is the projection of A_1 onto AB . So, D lies on AB and A_1D is perpendicular to AB .

Wait, is that necessarily true? If A_1B_1 is perpendicular to AB , and they intersect at D , then yes, D is the foot of the perpendicular from A_1 to AB . That seems correct.

Step 2: They examine the excircle and the triangle, mentioning that the excircle is tangent to BC at A_1 , and the tangents from A_1 to B_1 and C_1 form right angles with the radii. Then, DJ is a line through the excenter and the intersection point of the perpendiculars from A_1 and B_1 to AB .

Hmm, I'm not sure about this part. The line DJ is through the excenter J and point D , which is the foot from A_1 . But does it involve perpendiculars from A_1 and B_1 ? I think the solution might be conflating different properties here. The excenter is equidistant from all sides, but DJ is just a line from D to J , not necessarily involving perpendiculars from both A_1 and B_1 . Maybe this is a misstatement.

Step 3: They state that since E is the foot of the perpendicular from C_1 to DJ , angle C_1ED is 90 degrees. Then, they argue that angles BEA_1 and AEB_1 are equal due to symmetry and properties of the cyclic quadrilateral.

Wait, why is E symmetrically placed? The solution says it's because of the properties of the excircle and right angles, but I'm not entirely convinced. Just because E is a foot of a perpendicular doesn't automatically make it symmetric with respect to A_1B_1 . Maybe there's a better reason, like cyclic quadrilaterals or similar triangles, but this step feels a bit hand-wavy.

Step 4: They conclude both angles are 45 degrees because of symmetry and right angles, forming a cyclic quadrilateral. But I'm not sure if the cyclic quadrilateral part is clearly established. Also, 45 degrees seems specific—does the problem necessarily lead to that?

Wait, maybe I should draw a diagram to visualize. If A_1B_1 is perpendicular to AB , and E is the foot from C_1 to DJ , which is connected to the excenter. Hmm, maybe the angles are indeed 45 degrees because of the right angles and the properties of the excentral triangle, but I'm not entirely certain.

Overall Thoughts: The solution seems to have some correct steps but might be making assumptions without sufficient justification. For example, in Step 3, the symmetry argument isn't fully explained. Also, in Step 2, the description of DJ might not be accurate. The conclusion of 45 degrees might be correct, but the reasoning feels a bit rushed.

Final Answer After considering the steps, I think the solution might have some gaps in justification but arrives at the correct answer. So, I'll say yes, but with some reservations.

yes

Figure 17: Incorrect verification by Deepseek-R1-Distill-Qwen-14B for the problem-solution pair in Figure 16.

D Results on ProcessBench before and after finetuning

Table 3 shows the performance numbers of LLM-as-a-Judge and THINKPRM on ProcessBench.

Model	LLM-as-a-Judge		THINKPRM	
	OlympiadBench	OmniMath	OlympiadBench	OmniMath
Random baseline	39.1	32.7	39.1	32.7
R1-Qwen-1.5B	5.0 (51.4 %)	5.4 (55.1 %)	76.3 (1.4 %)	75.7 (2.4 %)
R1-Qwen-7B	44.8 (18.2 %)	45.7 (20.9 %)	73.4 (1.1 %)	74.0 (1.4 %)
R1-Qwen-14B	72.8 (13.3 %)	67.8 (18.6 %)	87.3 (2.3 %)	85.7 (2.3 %)
QwQ-32B-preview	50.6 (7.9 %)	55.5 (10.9 %)	73.1 (15.1 %)	73.2 (7.9 %)

Table 3: Average F1-score on OlympiadBench and OmniMath subsets of ProcessBench (Zheng et al., 2024) comparing LLM-as-a-Judge to THINKPRM finetuned on 1K examples. Random baseline for OlympiadBench is 39.1% and for OmniMath is 32.7%. Percentage of bad outputs (repetitions, invalid label formatting, overthinking, etc.) are shown in red. LLM-as-a-judge with reasoning models suffer from issues that limits their utility as generative verifiers.

E Evaluation details

This section includes exact details on the test-time scaling shown in §4.2

E.1 Predicting verification labels

Following prior work (Snell et al., 2024; Beeching et al.), we aggregate scores from DiscPRM by using the score of the *last* step. For THINKPRM, we first prompt the model to generate the verification chain up to a maximum of 8192 tokens, then we force decode the string “*Is the solution correct?*” and use $\frac{P(\text{"yes"})}{P(\text{"yes"})+P(\text{"no"})}$ as the solution score.

E.2 Best-of-N selection

We sample solutions using a temperature of $T = 0.8$ for Llama-3.2-3B-Instruct and $T = 0.4$ for Qwen-2.5-14B. We instruct all models to think step by step and put the final answer in `\boxed{}`. All our Best-of-N experiments use weighted majority voting, which scores final answers based on the sum of the verifier scores of their solutions (Uesato et al., 2022; Wu et al., 2024; Sun et al., 2024) except for our experiments on AIME ’24, where we use the verifier score directly to rank the solution, as we found this to perform better for all verifiers.

E.3 Verifier-guided beam search

Under verifier-guided beam search, we sample candidate next steps and score them with the process verifier, then selects top- K out of these to further expand and so on. Our implementation is based on (Snell et al., 2024; Beeching et al.), which maintains N beams in total, and samples M candidate next steps per beam. We set $M = 4$ for all experiments and run search for a maximum of 20 steps per beam. To sample next steps, we use $T = 0.6$ and use double newlines as the step delimiter.

E.4 Sequential scaling of verifier compute

We achieve budget forcing (Muenninghoff et al., 2025) by triggering the model to think again for R rounds, where each round uses a unique trigger phrase that incites the model to revisit or double-check its earlier verification. We use different trigger phrases for each round since we found that using the same phrase causes the model to repeat what it did in the last round.

We do a maximum of $R = 4$ thinking rounds, and use the phrases “Let me double check”, “Let’s verify again”, and “Did I miss something?”, for rounds 2, 3, and 4 respectively. We do not investigate deeply into optimizing

the trigger phrase, but we note that performance may depend on these and we use the same phrases for both THINKPRM and LLM-as-a-judge to ensure fair comparison.

E.5 MATH-500 test examples

As running on all 500 examples from MATH-500 will require a lot of compute, we run all our experiments on 100 randomly sampled subsets from MATH-500 (Hendrycks et al., 2021). We pick the 100 problems such that they cover different difficulty levels, as shown in Figure 18.

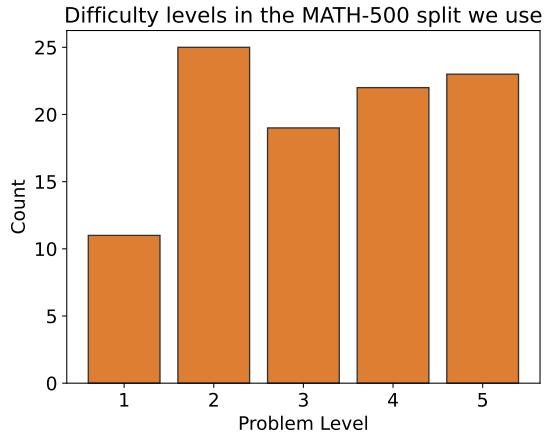


Figure 18: Histogram of difficulty levels in our 100-problem subset from MATH-500.

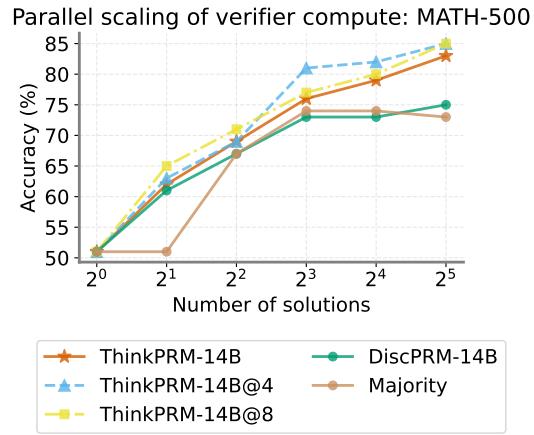


Figure 19: Scaling of verifier compute by parallel sampling of multiple verification CoTs and averaging their scores. Parallel scaling (THINKPRM-14B@4 and THINKPRM-14B@8) further boosts performance curve compared to scoring based on a single CoT (THINKPRM-14B).

E.6 Additional results on scaling verifier compute

Figure 19 shows results of THINKPRM-14B of parallel scaling verifier compute by sampling $K = 4$ and $K = 8$ CoTs with temperature $T = 0.6$ and aggregating their scores. Parallel scaling indeed lifts up the accuracy curve of THINKPRM-14B compared to standard $K = 1$ with greedy decoding. However, performance plateaus rather quickly and $K = 8$ remains comparable to $K = 4$, while slightly better at smaller sampling budgets. Figure 20 compares parallel to sequential scaling under the same token budget. While there is no clear winner, parallel scaling seems to perform slightly better at best-of-8.

E.7 Scaling training data of ThinkPRM

Here, we show results when training THINKPRM-14B and THINKPRM-1.5B using synthetic data from all PRM800K. The goal is to show that THINKPRM can still benefit from training on more synthetic data. Here, we train both R1-Distill-Qwen-1.5B and R1-Distill-Qwen-14B on a total of 65K verification CoTs we obtained by sampling and filtering as explained in §3.2. Figures 21 and 22 show best-of-N performance with THINKPRM-1.5B and THINKPRM-14B respectively when trained on 65K and compares it to training on 1K examples. Interestingly, THINKPRM benefits from additional training, and can further improve the accuracy curve compared to the 1K-trained version on MATH-500. We note, however, that while training on more math data boosts performance on MATH-500, we observe some performance drop on out-of-domain tasks due to the distribution shift.

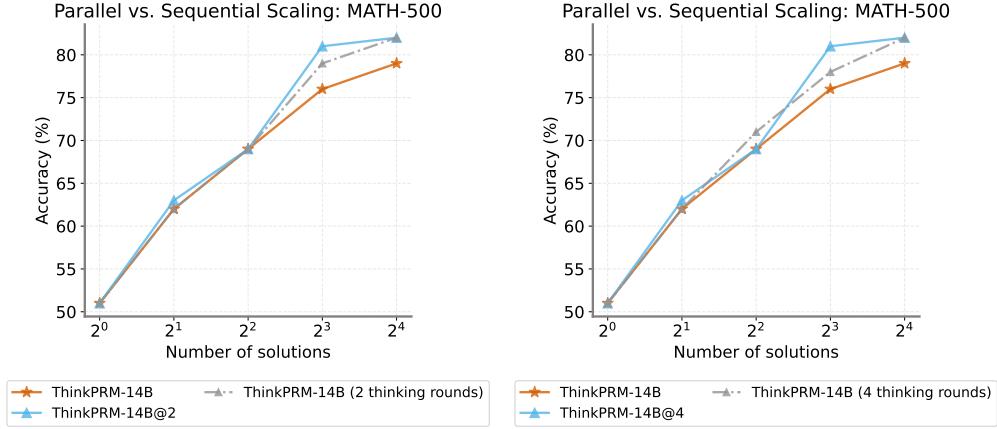


Figure 20: Parallel vs. sequential scaling of THINKPRM compute under the same generation budget with Qwen-2.5-14B generator. Parallel scaling (model@K) is done by independently sampling K verification CoTs and aggregating their scores. Sequential scaling is done by prompting the model K times to revise its own verification for K thinking rounds. Both setups generate up to 8192 tokens per generation. We do not observe a clear winner although parallel scaling seems slightly better especially at larger sampling budgets.

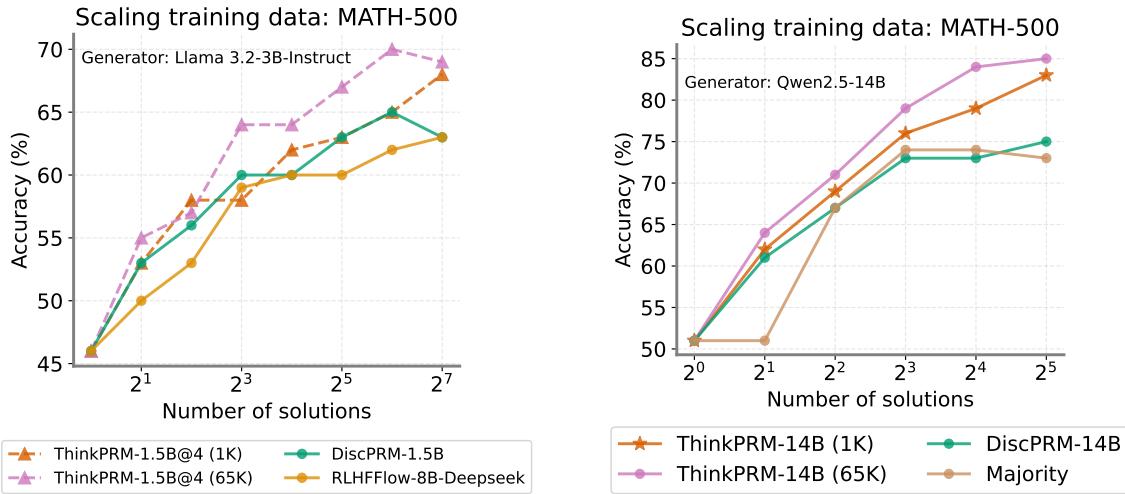


Figure 21: Best-of-N results with THINKPRM-1.5B comparing the version trained on 1K examples (used throughout the paper) and a version trained on 65K examples. THINKPRM benefits from training on more synthetic data as the performance can further improve with more training.

Figure 22: Best-of-N results with THINKPRM-14B comparing the version trained on 1K examples (used throughout the paper) and a version trained on 65K examples. THINKPRM benefits from training on more synthetic data as the performance can further improve with more training.

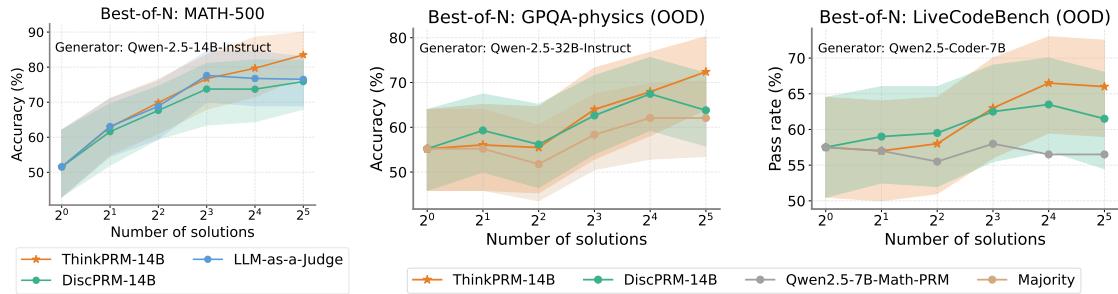


Figure 23: Best-of- N performance on MATH-500, GPQA-Physics, and LiveCodeBench with 95% confidence intervals (bootstrap $n = 100$). Despite wider intervals due to limited test set size, ThinkPRM-14B consistently outperforms DiscPRM-14B and LLM-as-a-Judge, especially at larger sampling budgets.

E.8 Results on the Full MATH-500 Dataset

In addition to the 100-problem subset used in several main-text analyses, we report best-of- N results on the **full MATH-500 test set**. All results use **Qwen-2.5-14B** as the generator, and we evaluate a range of verifiers under identical decoding and sampling budgets.

Table 4 shows that **ThinkPRM-14B consistently outperforms all baselines** across best-of-8, best-of-16, and best-of-32 settings. Notably, ThinkPRM-14B improves over strong off-the-shelf PRMs such as Math-Shepherd and Qwen-2.5-Math-PRM, despite being trained with substantially fewer step-level labels. These results confirm that the gains observed on smaller subsets extend to the full MATH-500 benchmark and are not an artifact of limited evaluation size.

Verifier	$N=8$	$N=16$	$N=32$
Majority	58.6	69.0	73.2
DiscPRM-14B	71.4	73.2	75.0
Math-Shepherd-7B	68.6	74.0	76.0
Qwen-2.5-Math-PRM-7B	72.6	76.8	78.8
ThinkPRM-14B	73.6	77.2	80.2

Table 4: Best-of- N accuracy (%) on the full MATH-500 dataset using Qwen-2.5-14B as the generator. ThinkPRM-14B achieves the strongest performance across all sampling budgets.

E.9 Bootstrapped Confidence Intervals for MATH-500, GPQA, and LiveCodeBench

To assess statistical robustness on small-scale evaluations, we report **bootstrapped 95% confidence intervals** for the best-of- N accuracy curves on three benchmarks: **MATH-500** (100 problems), **GPQA-Physics** (86 problems), and **LiveCodeBench** (200 problems). These results are visualized in Figure 23, from left to right.

Confidence intervals are computed via bootstrap resampling ($n = 100$) over test problems, with shaded regions indicating the 2.5th to 97.5th percentile range. While evaluation sets are relatively small, the observed performance trends remain consistent across tasks. We focus here on best-of- N —our primary test-time scaling setup—but note that trends align with our beam search results reported in the main text.

E.10 Results with automatic labels

Table 5 shows performance when filtering training data based on manual labels (PRM800K) vs automatic labels (Math-Shepherd) (Wang et al., 2023b). THINKPRM still performs well even with automatic labels, and comparably to manual labels.

Model	OlympiadBench	OmniMath
ThinkPRM-1.5B (PRM800K)	76.3	75.7
ThinkPRM-1.5B (Math-shepherd)	75.8	76.5

Table 5: Comparison of ThinkPRM-1.5B trained on PRM800K vs Math-shepherd step labels.

E.11 Verifier performance in terms of problem difficulty

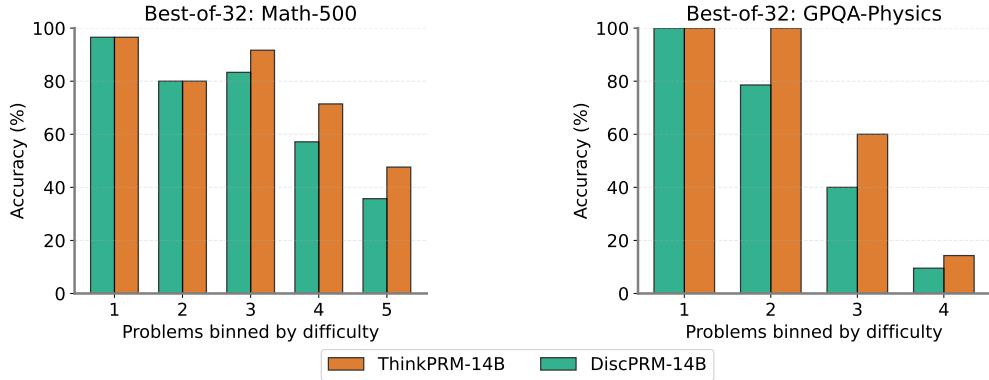


Figure 24: THINKPRM helps with challenging reasoning problems compared to DiscPRM. The generator model here is Qwen-2.5-14B for MATH-500 and Qwen-2.5-32B-Instruct for GPQA.

Each problem in MATH is annotated based on 1 of 5 difficulty levels so we just use that information. For GPQA-Physics problems, we first compute the pass@1 rate of Qwen2.5-32B-Instruct for every problem by sampling 32 solutions for each problem. Then, we bin the pass@1 rates into four different bins and use the bin assignment for each problem as the difficulty level. As discussed in §5.6, Figure 24 show how THINKPRM helps more with verifying hard problems compared to DiscPRM.

	Prompt	OlympiadBench	OmniMath
QwQ-32B	Single Yes/No	50.6 (7.9 %)	55.5 (3.2 %)
	Bad-step-index	65.4 (18.3 %)	69.0 (16.8 %)
	Decision-for-each-step	40.1 (6.4 %)	36.6 (9.2 %)
R1-Qwen-14B	Single Yes/No	72.8 (13.3 %)	67.8 (18.6 %)
	Bad step index	77.7 (17.0 %)	76.6 (17.9 %)
	Decision-for-each-step	73.3 (14.6 %)	71.4 (14.7 %)

Table 6: F1-score on OlympiadBench and OmniMath subsets from ProcessBench with different prompts. Ratio of invalid judgments is shown in red.

Example problem-solution pair from ProcessBench

Problem:
In quadrilateral $ABCD$, $m\angle B + m\angle D = 270^\circ$. The circumcircle of $\triangle ABD$ intersects \overline{CD} at point E , distinct from D . Given that $BC = 4$, $CE = 5$, and $DE = 7$, compute the diameter of the circumcircle of $\triangle ABD$.

Solution:

Step 0: To solve this problem, we will use several properties of geometry, particularly related to circles and quadrilaterals. Let's break down the steps: First, we understand the given information. We have a quadrilateral $ABCD$ where $\angle B + \angle D = 270^\circ$. The circumcircle of $\triangle ABD$ intersects \overline{CD} at point E , distinct from D . Additionally, $BC = 4$, $CE = 5$, and $DE = 7$.

Step 1: Next, we apply properties of cyclic quadrilaterals and circles. Since E lies on the circumcircle of $\triangle ABD$, E is also part of the circumcircle of $\square ABCD$ (assuming $ABCD$ is cyclic, which will be justified later). In a cyclic quadrilateral, opposite angles sum up to 180° . Given $\angle B + \angle D = 270^\circ$, it suggests an error in our initial understanding or misinterpretation, since angles cannot sum to more than 360° in a quadrilateral. However, if we consider the context, let's assume a correction was intended for the sake of explanation: Correct interpretation: If we adjust for logical consistency, perhaps the intention was to imply that one of the angles at B or D is 90° (since this would fit the property of cyclic quadrilaterals better, aligning with the geometry principles involved). For this explanation, let's proceed with an assumption that either $\angle B$ or $\angle D$ is 90° , aiming for a coherent solution path.

Step 2: We then use the Power of a Point Theorem. The Power of a Point Theorem states that for a point P outside a circle, the product of the lengths of the two segments from P to the circle equals the product of the lengths of the other two segments when considering tangents or secants from that point. However, in this case, E is not outside but on the circumference of the circle, and we apply it differently: Given $BC = 4$ and $CE = 5$, $DE = 7$, and knowing E is part of the circumcircle, we can apply the property of intersecting chords in a circle, which states that the product of the lengths of the segments of one chord is equal to the product of the lengths of the segments of the other chord. Thus, in this scenario, we consider BC and CD as intersecting chords, where $CD = BC + CE = 4 + 5 = 9$.

$$BC \cdot CD = CE \cdot DE$$

Substituting the known values:

$$4 \cdot 9 = 5 \cdot DE$$

$$36 = 5 \cdot DE$$

Thus,

$$DE = \frac{36}{5} = 7.2.$$

Step 3: This discrepancy suggests a reevaluation of the given values or a misunderstanding in their application within the context provided. However, for the purpose of demonstrating the method:

Step 4: Finally, we find the diameter of the circumcircle of $\triangle ABD$. The key to finding the diameter involves recognizing that the circumcircle's radius or diameter is crucial for calculating distances involving points A , B , and D . However, without specific details about how E relates to these points geometrically beyond being on the circle, or additional information such as the length of another segment or angle measures, directly calculating the diameter becomes complex.

Step 5: The question's focus on E and its relation to BC , CE , and DE might suggest looking at the properties of chords in a circle or applying trigonometric relationships based on the angles given initially. However, due to the discrepancy noted and the complexity introduced by the misinterpretation of the angle condition, a direct calculation for the diameter of the circumcircle of $\triangle ABD$ isn't straightforward with the information provided.

Step 6: Given the complexities and discrepancies in the original problem statement, particularly with the angles summing to 270° in a quadrilateral, the solution path becomes less clear. The explanation provided aimed to creatively interpret and apply geometric principles under the constraints of the given information. However, the direct calculation for the diameter of the circumcircle of $\triangle ABD$ requires additional, precise information about the positions and relationships of the points A , B , C , D , and E within the quadrilateral and its circumcircle.

Figure 25: An example problem and solution from the OlympiadBench (He et al., 2024) in ProcessBench (Zheng et al., 2024).

F More details on LLM-as-judge verification

The instruction used for LLM-as-judge verification in §3.1 is shown in Figure 27, which we refer to as Single-label. Table 6 shows performance of QwQ-32B-Preview and Qwen-Deepseek-R1-14B with two more instructions. **Bad-step-index** refers to instructing the model to find the index of the first incorrect step and then if a step is found generate a single yes/no label regarding the solution correctness. **Decision-for-each-step** instructs the model to critique and produce a boxed label for each step. This instruction is shown in

Example from QwQ-32B-Preview

Let's verify step by step:

Step 0: The initial step seems to correctly identify the given information in the problem. It mentions the quadrilateral $ABCD$ with angles at B and D summing to 270° , and the circumcircle of $\triangle ABD$ intersecting \overline{CD} at E , distinct from D . The lengths $BC = 4$, $CE = 5$, and $DE = 7$ are also noted. This step is accurate in summarizing the problem statement.

Step 1: This step attempts to apply properties of cyclic quadrilaterals but seems to get confused about the sum of angles in quadrilateral $ABCD$. It is mentioned that in a cyclic quadrilateral, opposite angles sum to 180° , but the given sum of angles $\angle B$ and $\angle D$ is 270° , which indeed seems inconsistent. However, the step then suggests assuming that one of the angles is 90° for the sake of explanation, which is an adjustment to proceed with the solution. This part is a bit problematic because it alters the given information, but perhaps there was a misinterpretation in the problem statement.

Step 2: Here, the Power of a Point Theorem is applied to points related to the circle. However, there is confusion in calculating the length of \overline{CD} . It states that $CD = BC + CE = 4 + 5 = 9$, but since E is a point on \overline{CD} distinct from D , and DE is given as 7, this suggests that $CD = CE + DE = 5 + 7 = 12$, not 9. This is an error in calculation. Furthermore, the application of the Power of a Point Theorem seems misapplied because E is on the circle, not outside it, so the theorem should be applied differently, possibly considering intersecting chords.

Step 3: Recognizing the discrepancy in the previous step, the solution acknowledges the need for reevaluation, which is a good practice. However, without correcting the earlier mistake in calculating \overline{CD} , the subsequent steps may be built on incorrect foundations.

Step 4: This step aims to find the diameter of the circumcircle of $\triangle ABD$, which is the main goal of the problem. It notes that the diameter is related to the circumradius, but without specific details about the triangle's sides or angles, it's challenging to compute directly. The solution hints at the need for more information, which is accurate.

Step 5: The insight here is that the relationship between E and the other points might provide the necessary information through chord properties or trigonometric relationships. However, due to the earlier discrepancies, this step does not lead to a concrete calculation.

Step 6: The conclusion reflects on the complexities and discrepancies in the problem and the solution approach. It acknowledges that the direct calculation for the diameter is not straightforward with the given information, which is a fair assessment.

Overall, while the solution attempt shows an understanding of relevant geometric principles, there are significant errors in calculation and application of theorems, particularly in Step 2. These errors propagate through the subsequent steps, leading to an inconclusive solution. Therefore, the solution is not correct.

\boxed{no}

Figure 26: Example verification chain generated by QwQ-32B-Preview in response to the problem-solution pair in Figure 25.

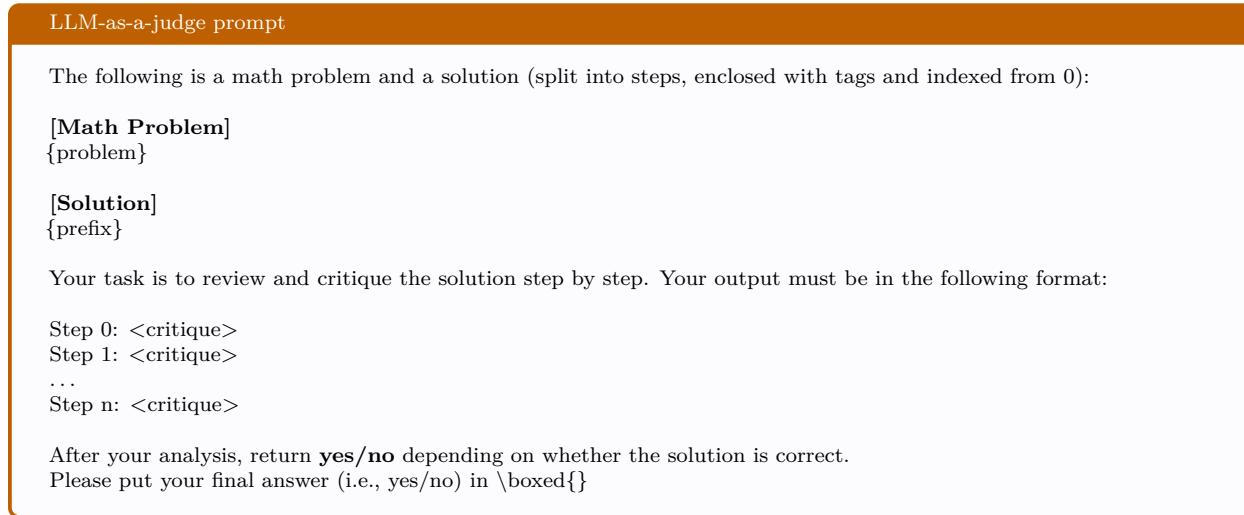


Figure 27: Prompt used for LLM-as-a-judge verification in §3.1

Figure 13. We observe high ha variance in verification performance and invalid judgment ratio depending on the prompt.

We also an example from ProcessBench in Figure 25 and the generated verification chain from QwQ-32B-Preview in Figure 26.

G Examples from ThinkPRM

Figure 29 shows an example of a verification CoT from THINKPRM-14B on a problem and solution from the OmniMath subset in ProcessBench, shown in Figure 28.

Example verification chain-of-thought

Problem: A solid rectangular prism has dimensions 4 by 2 by 2. A 1 by 1 by 1 cube is cut out of the corner creating the new solid shown. What is the surface area of the new solid?

Solution:

Step 1: To determine the surface area of the new solid after cutting a 1 by 1 by 1 cube from one corner of a 4 by 2 by 2 rectangular prism, we will follow these steps: First, calculate the surface area of the original rectangular prism. The original rectangular prism has dimensions 4 by 2 by 2. The surface area A_{original} of a rectangular prism is given by:

$$A_{\text{original}} = 2(lw + lh + wh)$$

where $l = 4$, $w = 2$, and $h = 2$. Substituting these values in, we get:

$$A_{\text{original}} = 2(4 \cdot 2 + 4 \cdot 2 + 2 \cdot 2) = 2(8 + 8 + 4) = 2 \cdot 20 = 40$$

Step 2: Second, determine the effect of cutting out the 1 by 1 by 1 cube. When we cut out a 1 by 1 by 1 cube from one corner, we remove three faces of the cube (one face for each dimension) but add three new faces of the same size. Each face of the cube is 1 by 1, so the area of each face is 1 square unit.

Step 3: Third, calculate the change in surface area. Removing the three faces of the cube reduces the surface area by $3 \times 1 = 3$ square units. Adding the three new faces increases the surface area by $3 \times 1 = 3$ square units. Therefore, the net change in surface area is:

$$3 - 3 = 0$$

Step 4: Finally, calculate the total surface area of the new solid. Since the net change in surface area is zero, the surface area of the new solid remains the same as the original surface area:

$$A_{\text{new}} = A_{\text{original}} = 40$$

Thus, the surface area of the new solid is 40.

Figure 28: Problem and solution from the OmniMath subset of ProcessBench. The verification CoT output by THINKPRM is shown in Figure 29

Example verification chain-of-thought from THINKPRM-14B

Verification: `<think>`

Alright, I've got this math problem about finding the surface area of a solid after cutting out a small cube from a larger rectangular prism. The original prism has dimensions 4 by 2 by 2, and a 1 by 1 by 1 cube is removed from one corner. The provided solution has four steps, and I need to review each one to see if it's correct.

Let's start with Step 1.

Step 1: The solution calculates the surface area of the original rectangular prism. The formula used is $A_{\text{original}} = 2(lw + lh + wh)$, which is correct for a rectangular prism. Plugging in the dimensions $l = 4$, $w = 2$, and $h = 2$, the calculation goes like this:

$$A_{\text{original}} = 2(4 \cdot 2 + 4 \cdot 2 + 2 \cdot 2) = 2(8 + 8 + 4) = 2 \cdot 20 = 40$$

This seems right. So, Step 1 is `\boxed{correct}`.

Step 2: Here, the solution talks about the effect of cutting out the 1 by 1 by 1 cube from one corner. It says that removing the cube takes away three faces but adds three new faces. Each face is 1 by 1, so each has an area of 1 square unit.

Wait a minute, is this accurate? When you cut out a cube from a corner, you are indeed removing material, but how does that affect the surface area?

Let me visualize this. Imagine a rectangular prism and cutting out a small cube from one corner. The cube has three faces exposed to the outside of the prism. When you remove the cube, those three faces are no longer part of the surface area. However, the removal also creates three new internal faces where the cube was taken out, which are now part of the surface area.

So, the solution says that removing three faces reduces the surface area by 3 square units and adding three faces increases it by 3 square units, leading to no net change. That seems logical at first glance.

But is this actually correct? Let me think deeper.

When you remove the cube, you are taking away three 1x1 faces from the original prism, which indeed reduces the surface area by 3 square units. However, the new faces created are also 1x1, and there are three of them, so that adds 3 square units back. So, the total surface area remains the same.

But I'm not entirely sure. Maybe I should consider if the new faces are actually exposed or if they are somehow internal. Wait, no. When you cut out the cube from the corner, the three new faces are now part of the exterior of the new solid, so they should be included in the surface area.

Therefore, the net change is indeed zero, and the surface area remains 40.

Hmm, but I recall that sometimes when you make a cut, the surface area can change in unexpected ways. Maybe I should double-check this.

Let me consider a simpler example. Suppose I have a cube and I cut out a smaller cube from one corner. How does the surface area change?

Original cube: surface area is $6s^2$, where s is the side length.

After cutting out a smaller cube from one corner, I remove three faces of the smaller cube and expose three new faces. So, the change in surface area is $-3s^2 + 3s^2 = 0$, similar to what the solution says.

Alternatively, perhaps I can think about the new solid's dimensions.

But I think the initial approach is correct.

Therefore, Step 2 is `\boxed{correct}`.

Step 3: This step calculates the net change in surface area as zero, which follows from Step 2.

Since the net change is zero, the surface area remains the same as the original.

So, Step 3 is `\boxed{correct}`.

Step 4: Concludes that the surface area of the new solid is 40, the same as the original.

Therefore, Step 4 is `\boxed{correct}`.

Wait, but I recall that sometimes when you make a cut, especially in certain ways, the surface area can change. But in this case, it seems to hold up.

Alternatively, maybe I should consider if the cube is cut from a corner where only two faces are on the exterior. Wait, no. In a rectangular prism, each corner is where three faces meet, so cutting out a cube from a corner would remove three faces and expose three new ones.

But perhaps I should consider if the cube is cut from a corner where one of the original faces is larger than 1x1. Wait, the original prism is 4x2x2, and the cube being removed is 1x1x1, so the faces being removed are indeed 1x1. So, the calculation seems accurate.

Alternatively, maybe I can think about it in terms of the total surface area.

Alternatively, perhaps I can think about the surface area of the new solid directly.

Let me try that.

The new solid has the original surface area minus the three 1x1 faces that were removed, plus the three new 1x1 faces that are now exposed.

So, $40 - 3 + 3 = 40$.

Alternatively, perhaps I can think about the surface area as the sum of all exterior faces.

But I think the initial approach is correct.

Therefore, the solution is correct.

But the instruction says to review and critique each step, and determine if the whole solution is correct.

So, in this case, all steps seem correct, leading to the conclusion that the surface area remains 40. [`</think>`](#)

Figure 29: Example output by THINKPRM-14B, where it verifies and labels every step in the provided prefix via a long verification CoT. The problem and solution are in Figure 28

Verification of long CoT solution

Problem:

Convert the point $(0, 3)$ in rectangular coordinates to polar coordinates. Enter your answer in the form (r, θ) , where $r > 0$ and $0 \leq \theta < 2\pi$.

Solution:

Okay, so I need to convert the rectangular coordinates $(0, 3)$ to polar coordinates. Let me recall how to do this. From what I remember, polar coordinates are given as (r, θ) , where r is the distance from the origin to the point, and θ is the angle made with the positive x -axis. The formulas to convert between rectangular (x, y) and polar (r, θ) coordinates are:

$$x = r \cos \theta, \quad y = r \sin \theta$$

To find r , we use:

$$r = \sqrt{x^2 + y^2}$$

To find θ , we use:

$$\theta = \arctan\left(\frac{y}{x}\right)$$

But I need to be careful with the quadrant since \arctan may be undefined or ambiguous when $x = 0$.

Given $x = 0$ and $y = 3$, compute:

$$r = \sqrt{0^2 + 3^2} = \sqrt{9} = 3$$

So $r = 3$. For θ , since $x = 0$, the expression $\arctan(y/x)$ is undefined due to division by zero.

However, because the point lies on the positive y -axis, $\theta = \frac{\pi}{2}$. This is the standard angle for points directly above the origin.

Therefore, the polar coordinates are $\boxed{\{3, \frac{\pi}{2}\}}$.

Wait, let me check if there's another way to get this. Since the point is $(0, 3)$, which is on the positive y -axis, the angle is indeed $\frac{\pi}{2}$.

Could θ be $\frac{3\pi}{2}$? No, $\frac{3\pi}{2}$ is the negative y -axis, which is the opposite direction. Since the point is on the positive y -axis, $\theta = \frac{\pi}{2}$.

Let me recall again that when converting from rectangular to polar, θ is calculated as $\arctan(y/x)$, but when $x = 0$, θ is either $\frac{\pi}{2}$ or $\frac{3\pi}{2}$ depending on the sign of y . Here $y > 0$, so the correct angle is $\frac{\pi}{2}$.

...

ThinkPRM-1.5B:

`<think>`

Step 1: The student correctly recalls the formulas for converting from rectangular to polar coordinates:

$$x = r \cos \theta, \quad y = r \sin \theta$$

and

$$r = \sqrt{x^2 + y^2}, \quad \theta = \arctan\left(\frac{y}{x}\right)$$

This step is `\boxed{correct}`.

Step 2: The student computes:

$$r = \sqrt{0^2 + 3^2} = 3$$

This is `\boxed{correct}`.

Step 3: The student identifies that $\theta = \arctan(3/0)$ is undefined but correctly infers that the point lies on the positive y -axis, hence $\theta = \frac{\pi}{2}$. This reasoning is also `\boxed{correct}`.

...

`</think>`

Figure 30: Example of verification of long CoT solution from Qwen3-1.7B-thinking using THINKPRM-1.5B. THINKPRM extracts and verifies individual steps embedded in the long CoT.