PROVER: PROCESS REWARD ORACLES VIA VERIFIED EQUATIONAL REASONING

James Shiffer
University of California, Los Angeles
Los Angeles, CA
jshiffer@ucla.edu

ABSTRACT

PROVER is a novel, fully-automated framework for training Process Reward Models (PRMs) in mathematics domains using verifiable silver-standard step-level annotations. We generate training data from formal problem statements by (1) using an LLM to sample potentially valid proofs in the Lean programming language; (2) annotating them based on feedback from the Comparator tool, plus some checks of our own; and (3) informalizing them into natural language while preserving step-level associations. Our process ensures that **PROVER** learns from positive and negative examples grounded in mathematical reasoning. We train our own PRM **PROVER**-7B on this dataset and evaluate it on PRMBench, where it achieves state-of-the-art *first error accuracy* for an open-source PRM and state-of-the-art *first error distance* in the 7B class. Our PRM nearly meets or exceeds its human-annotated equivalent by every measure, demonstrating that synthetic but grounded data enables model developers to fully automate PRM training without sacrificing quality.

1 INTRODUCTION

Process reward models, which provide a reward signal at every step along a reasoning trace, play a key role in reinforcement learning and test-time verification of modern LLMs. Presently, there exists a tradeoff when sourcing training data for the PRM itself: either write and annotate processes manually for the highest quality, usually by recruiting a team of human domain experts, or generate synthetic training data, which saves time and effort but sacrifices quality due to the opportunity for hallucinations. **PROVER** introduces a new paradigm that sidesteps this tradeoff by first generating synthetic proofs from off-the-shelf math problems, then annotating them with grounded feedback from the Lean 4 formal verification language—no human intervention needed. We use our pipeline to create a dataset of about 2,000 problems with high-quality step-by-step annotations, which is used to train a PRM of our own. Then, we evaluate our PRM's performance on the PRMBench benchmark.

2 BACKGROUND

Reinforcement learning. The LLMs of today undergo two major training phases: pre-training and post-training. Pre-training primarily teaches a model how to predict text, while post-training focuses on improving its alignment, correctness, and tone. Within post-training is **reinforcement learning (RL)**, a training paradigm that learns from interactive feedback. In RL, the LLM being trained acts as an **agent**, which induces **actions** within an **environment** to produce **rewards**. The agent then updates its own behavior, or **policy**, based on a learning algorithm. Some commonly-used RL algorithms include Proximal Policy Optimization (PPO) (Schulman et al. (2017)), Direct Preference Optimization (DPO) (Rafailov et al. (2024)), and Group Relative Policy Optimization (GRPO) (Shao et al. (2024)). Traditionally, in PPO, a **reward model** scores the actions taken by the agent; this is where either an outcome reward model or a process reward model comes into play.

Process reward models. The job of a **process reward model (PRM)** is to predict correctness labels for every step along a reasoning trace (Lightman et al. (2023)). This provides a much more fine-

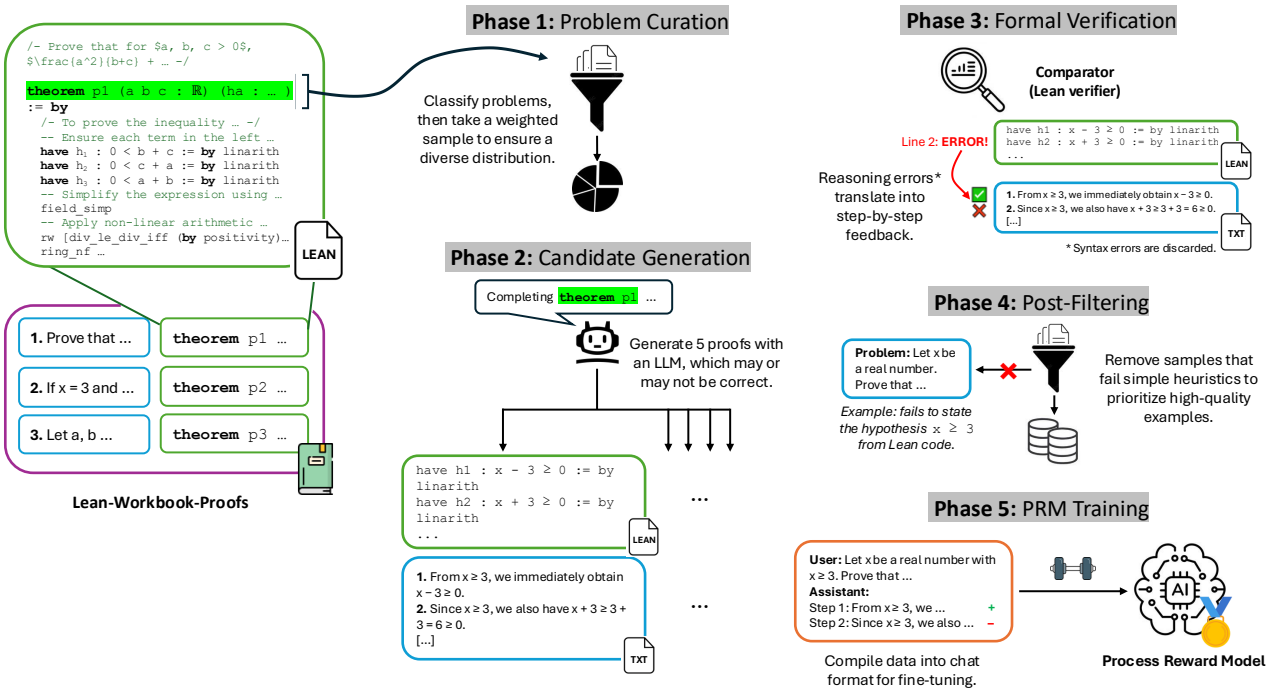

Figure 1: The **PROVER** framework.

grained reward signal compared to an outcome reward model (ORM), which only judges the final answer. PRMs are typically specialized for mathematics or coding domains. To put it more formally, suppose we use binary labels $\{0, 1\}$ to signify that a step is incorrect or correct, respectively. If we have a problem statement p and a candidate solution S of n steps, the process reward model $\mathcal{M}$ can be used to predict labels (annotations) $L = \{0, 1\}^n$. Specifically, each step's label is conditioned on the previous steps and the problem statement, such that $L_i = \mathcal{M}(S_i|p, S_{1...i-1}) = \{0, 1\}$. In practice, PRMs are almost always created by fine-tuning LLMs, due to the effectiveness of their chain-of-thought abilities (Wei et al. (2023)). PRMs are typically evaluated on benchmarks like PRMBench (Song et al. (2025)) and ProcessBench (Zheng et al. (2025)), where either (1) the PRM's prediction for *every* step is compared against a set of reference annotations, or (2) only the position of the *first* incorrect step is considered. For the purpose of this work, we are mainly interested in the latter.

Until now, the process of creating high-quality PRM training data for math has relied heavily on either manual labor or LLM-as-a-judge techniques. In OpenAI's PRM800K dataset, for instance, reasoning traces were generated synthetically and then hand-annotated by human mathematicians (Lightman et al. (2023)). In Math-Shepherd, annotations were generated without any human intervention by first generating several synthetic outputs from LLMs, then judging them based on the success rate of their continuations (Wang et al. (2024)). These methods are not without their shortcomings; human labeling requires additional effort and time, and the LLM-as-a-judge paradigm cannot guarantee correctness. Our goal is to combine the efficiency of fully automated annotation generation with the infallibility of formal verification.

Lean programming language. Lean is a functional programming language and proof assistant (de Moura et al. (2015)), mostly used by mathematicians for its formal verification capabilities. Programs in Lean represent mathematical proofs, where the programmer first states their goals and assumptions and then attempts to prove them using tactics, which may be rules-based or use numerical computing methods. Some examples of tactics are `simp`, which simplifies algebraic expressions using known lemmas, or `linarith`, which attempts to prove or disprove an inequality using linear programming. Lean guarantees that it will find any errors in one's proof so long as the provided assumptions are correct (de Moura et al. (2015)), which makes it well-suited for problems such as process reward modeling where the LLMs lack a grounded understanding of mathematics. In the

Category	Weight
Existence Construction	0.20
Iff Logic	0.15
Divisibility/Modular	0.25
Inequality	0.20
Equality/Identity	0.20

Table 1: Breakdown of proofs sampled from Lean-workbook-proofs.

Lean ecosystem, tools such as Axiom Lean Engine (Axiom Math (2026)) and Comparator (Lean Prover Community (2025)) exist for stronger correctness guarantees, if one wishes to verify that the inverse of a proof can be disproven, or limit the scope of permitted tactics, for example. To our knowledge, this work represents the first time a formal verification language has been used in the training of PRMs.

As Lean has risen in popularity, there has been a growing interest in training specialized "prover" LLMs that can write well-formed Lean programs. These models come in a couple of different forms: DeepSeek-Prover-V2 (Ren et al. (2025)) and Goedel-Prover-V2 (Lin et al. (2025b)) are fine-tuned reasoning LLMs, while Leanstral (Mistral AI (2026)) is an interactive coding agent. Usually, these models are evaluated at their ability to write compiling Lean programs after *several* attempts as single-shot performance is still lacking. The authors of Goedel-Prover ran their model on formalized (but unproven) math problems from Lean Workbook (Ying et al. (2025)), publishing a dataset of 30,000 new proofs that we use as a starting point for our own pipeline.

3 METHODS

PROVER is our fully automated multi-stage data generation pipeline, designed to generate a diverse silver-standard dataset for PRM training that has a reasonable balance of positive and negative samples, comprises several failure modes, and has problems of varying type and difficulty.

3.1 DATA CURATION AND PRE-FILTERING

The Lean-workbook-proofs dataset from Goedel-LM is used as our starting point. This dataset contains almost 30,000 formalized math problems from Lean Workbook (Ying et al. (2025)), complete with proofs generated by the Goedel-Prover model (Lin et al. (2025a)). Each proof can be thought of as a self-contained Lean program encoding the information $(p_{\mathcal{I}}, p_{\mathcal{F}}, S_{\mathcal{I}}, S_{\mathcal{F}})$, where $p_{\mathcal{I}}$ and $p_{\mathcal{F}}$ represent the informalized (natural language) and formalized (Lean code) versions of the problem statement, respectively, and S is a list of steps, represented informally (as $S_{\mathcal{I}}$, a sequence of natural language comments) and formally (as $S_{\mathcal{F}}$, a sequence of Lean tactics). An example can be seen in §A.1.

We perform additional processing on the Lean-workbook-proofs data to enhance its quality. First, we classify 1,000 randomly selected problems into one of several categories. Then, because certain categories like inequalities are highly overrepresented in the base dataset, we sample 500 problems from those 1,000, according to the weighted probabilities listed in Table 1.

The next step is to strip all natural language comments $p_{\mathcal{I}}$ and $S_{\mathcal{I}}$ from the Lean source code; upon manual inspection, we discovered that these are sometimes misleading or incomplete and can bias our own generations (see §A.2). We use a frontier model (Claude Opus 4.6) instead to generate our own high-quality informalizations.

3.2 CANDIDATE SOLUTION GENERATION

For each Lean problem statement $p_{\mathcal{F}}$, we generate four candidate proofs using Claude Opus 4.6, each taking the form $(p'_{\mathcal{I}}, S'_{\mathcal{I}}, S'_{\mathcal{F}})$. As the notation suggests, each solution produces a complete Lean program as well as a natural language explanation, to later be annotated with stepwise correct and incorrect labels. Both outputs are produced in tandem from a single LLM invocation; the exact

prompt is given in §A.3. The $p'_{\mathcal{I}}$ refers to our newly generated high-quality problem statement after deliberately discarding the original dataset's $p_{\mathcal{I}}$. Additionally, every natural language step $S'_{\mathcal{I},i}$ is mapped to one or more lines of Lean code $S'_{\mathcal{F},j\ldots k}$ (where $j \leq k$), enabling traceability when we uncover Lean compiler errors in the next pipeline stage. Furthermore, we blacklist 20 opaque tactics such as `nlinarith` and `omega` (which use linear and integer programming algorithms, respectively) to encourage the use of more easily traceable algebraic primitives like `div_eq_div_iff` (which expresses the property that if $\frac{a}{b} = \frac{c}{d}$, then $a = b$ iff $c = d$). During development, we observed that inequality problems in particular were prime targets for technically correct but unhelpful one-line `nlinarith` proofs.

Certain Lean tactics like `simp` (simplify expressions) and `linarith`—which is blacklisted for new candidates, but still present in preexisting proofs—can be broken down into intermediate steps. To ensure highly granular steps and ameliorate the risk of hallucinations, we enable the Trace module in Lean and provide the resulting logs as context for informal natural language generation, while also prompting the LLM to add more steps to accommodate these traces (§A.4). This step only applies to proofs that already compile successfully. We also monitor the proof state before and after each tactic. Once Lean reports that all goals are solved, we truncate any remaining lines from the proof to eliminate superfluous code.

Our candidate generation process is intended to introduce occasional errors in the output, but we nevertheless choose a frontier model to perform our code completions because we are chiefly interested in *reasoning* errors that stem from a misapplication of mathematical principles, not simple syntax or library errors resulting from an inability to write well-formed Lean code. The open-source models we tried, such as DeepSeek-Prover-V2, mostly resulted in the second kind of error, and we did not try coding agents like Leanstral. Contemporary frontier models still tend to take several attempts to produce a passing Lean program, especially with graduate- or Olympiad-level problems (Ren et al. (2025)), so they should produce some number of failures on their own. However, Lean-workbook-proofs tends to have easier examples, so we must also prompt the model to deliberately introduce human-like errors into its reasoning process (§A.5) to promote a healthy balance of correct and incorrect training data.

3.3 Formal Verification

Comparator as a judge. We assess the quality of each candidate $(p_{\mathcal{F}}, S'_{\mathcal{F}})$ using Comparator, a trustworthy judge for Lean proofs (Lean Prover Community (2025)). Comparator subjects proofs to more rigorous checking than the compiler alone by giving us the ability to enforce our tactic blacklist from before, and catching cases where synthetic proofs may try to "cheat" the blacklist using metaprogramming. Using pattern matching of compiler error messages, we discard candidates that have syntax errors. Compiler errors can be traced to the exact line of code $S'_{\mathcal{F},i}$, which we can map back to its natural language step in $S'_{\mathcal{I}}$ to assign correctness labels $\{0, 1\}$.

Besides our newly generated candidates, we also subject the original formal solution $(p_{\mathcal{F}}, S_{\mathcal{F}})$ from Lean-workbook-proofs to the same audit, for a total of five annotated Lean proofs per problem. There is one caveat, though: the Lean code in Lean-workbook-proofs was intended to be compiled by Lean v4.9.0, but Comparator is not fully reverse compatible with that version, so some proofs using deprecated tactics have to be thrown out. In total, we now have around 2,400 labeled proofs, barring version incompatibilities and any errors that may have occurred during LLM generation.

Cascading errors. We apply a cascading error policy to our training data, where we assign a label of "incorrect" to *every* step after the first error; future steps that could potentially rest on a false premise cannot serve as reliable rewards when every step's label is conditioned on the ones before it. Another reason is that the Lean compiler, like those of most compiled programming languages, may hide subsequent errors that are only revealed after the first is fixed, and we rely on accurate error reporting to maintain our mathematical grounding.

3.4 Post-Filtering

To account for any hallucinations or mistakes that may have happened during the annotation process, we run a final set of sanity checks on the full set of proofs. First, we remove examples where the natural language's "step 0" is labeled incorrect, as this is reserved for the problem statement which

Model Name	Overall		Simplicity		Soundness		Sensitivity*	
	Score	First	Score	First	Score	First	Score	First
Skywork-PRM-7B	65.1	56.6%	48.8	**51.4%**	60.4	72.3%	37.1	**75.3%**
Llemma-PRM800k-7B	52.0	22.2%	51.3	18.1%	50.9	23.5%	52.3	23.8%
MathShepherd-Mistral-7B	47.0	54.6%	47.1	45.0%	45.7	61.3%	47.9	51.0%
ReasonEval-7B	60.1	30.3%	55.5	25.9%	63.9	37.1%	56.9	21.0%
ReasonEval-34B	60.5	50.8%	51.5	25.8%	63.0	65.7%	61.0	46.0%
RLHFlow-PRM-Mistral-8B	54.4	22.1%	46.7	11.1%	57.5	27.8%	53.9	21.6%
RLHFlow-PRM-DeepSeek-8B	54.2	17.0%	47.6	8.6%	57.5	23.2%	52.2	13.0%
Qwen2.5-Math-PRM-7B	65.5	37.8%	52.0	13.0%	71.0	53.1%	63.3	32.3%
Qwen2.5-Math-PRM-72B	**68.2**	48.5%	54.6	19.9%	**73.9**	66.9%	**65.8**	40.4%
PROVER-7B	64.4	**61.0%**	**59.3**	36.2%	66.7	**76.1%**	64.0	55.5%

Table 2: PRMBench (Song et al. (2025)) performance, as measured by mean PRMScore (↑) and first error accuracy (↑). In each category, first place is bolded and second place is underlined.
†This model was run on a subset of 394 samples for cost savings.
*Excludes multi-solution consistency scenarios, where the ground truth has no incorrect steps.

Model Name	Overall	Category Average			Lead/Lag %
		Simplicity	Soundness	Sensitivity*	
Skywork-PRM-7B	4.121	3.825	4.195	4.268	39.8% / 26.3%
ReasonEval-7B	4.695	5.126	4.290	5.075	22.0% / 53.3%
ReasonEval-34B	3.926	5.418	3.330	3.624	36.6% / 34.8%
Qwen2.5-Math-PRM-7B	3.813	5.669	2.653	4.276	**7.2%** / 57.5%
Qwen2.5-Math-PRM-72B	**3.226**	4.966	**2.112**	3.714	13.0% / 44.1%
PROVER-7B	3.731	**3.515**	3.913	**3.582**	50.9% / **21.0%**

Table 3: Mean first error distance (↓) and lead/lag frequency of select open-source models. First place is bolded and second place is underlined.
*Excludes multi-solution consistency scenarios, where the ground truth has no incorrect steps.

should never be incorrect. Next, we use pattern-matching to cross-check the formal and informal problem statements to catch a common failure mode where some of the hypotheses (assumptions) in the Lean code fail to be captured by the informal statement. Finally, we remove any natural language descriptions containing Lean-related jargon that may have slipped through, because the PRM does not need to know the specifics of our formal verification toolkit. After post-filtering, around 2,200 proofs remain.

3.5 Training Data

The training data is formatted using the Math-Shepherd template (Wang et al. (2024)), in which the steps are written out line-by-line with a "+" or "–" at the end to signify correct or incorrect. We performed 3 epochs of supervised fine-tuning on Qwen2.5-Math-7B-Instruct, the same base model used in the creation of Qwen2.5-Math-PRM-7B (Zhang et al. (2025)). We chose a learning rate of 1.0×10^{-5}, with cosine decay and warmup ratio 0.1, and reserved 10% of the data for validation, to be run every 200 steps. The Qwen PRM, which was trained on the human-labeled PRM800K dataset, serves as a good baseline for evaluating **PROVER**-7B's performance.

4 Evaluation

We measure the performance of **PROVER**-7B on PRMBench, a challenging benchmark for process reward models (Song et al. (2025)). PRMBench categorizes problems into one of three types: simplicity, soundness, and sensitivity. While the primary metric used by the authors is the *PRM-Score*, which is derived from the F1 score taken across all annotations, we explain why this is not

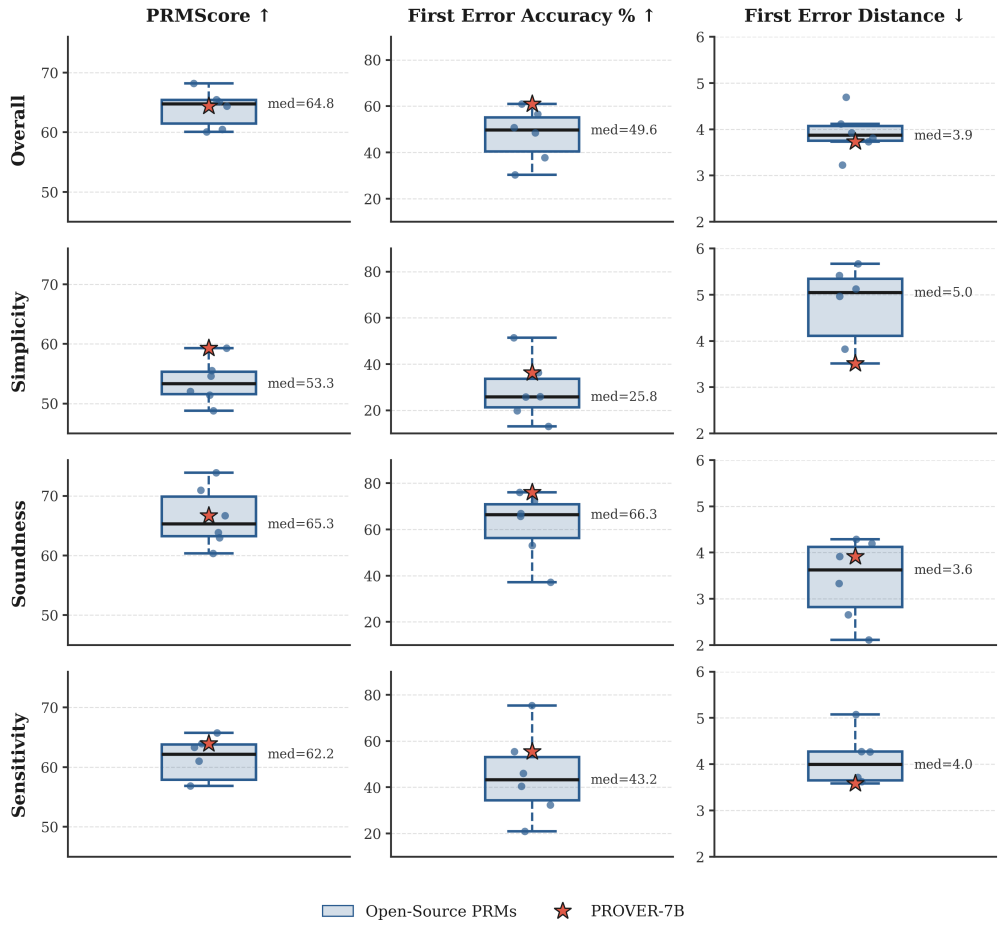

Figure 2: **PROVER**-7B's relative standing among open-source PRMs, broken down by category.

especially useful in our case in §5. Rather, the most relevant metric from PRMBench is *first error accuracy*: how often the first ground-truth error was accurately identified as a mistake. Table 2 includes a detailed performance comparison with a breakdown by problem category. **PROVER**-7B achieves a higher overall first error accuracy than any other open-source model, including the much larger Qwen2.5-Math-PRM-72B. However, it struggles to do consistently well across all categories, falling into third place for the Simplicity problems. It is also worth mentioning that even though maximizing PRMScore was not the primary goal, **PROVER**-7B still achieves above-average performance by this measure. The box plots in Figure 2 offer a visualization of our model's relative ranking.

First error distance. A naïve model which always predicts a label of "incorrect" could trivially achieve a first error accuracy of 100%, so we evaluated the most competitive open-source models on yet another metric, *first error distance*, defined as the absolute difference between the indices of the first ground-truth error and the first predicted error. We do not consider samples where either the ground-truth or predicted labels are all correct. Because the first error distance does not account for direction, we also compute the frequency of **lead** (predicted first error falls before the ground truth) and **lag** (predicted first error falls after the ground truth). Results can be seen in Table 3. **PROVER**-7B outclasses other open-source PRMs of its size, and even holds its own among far larger models like ReasonEval-34B. It only loses to Qwen2.5-Math-PRM-72B, and with far greater consistency across categories still.

Confusion matrices. To better capture the failure modes happening on a step-by-step level, confusion matrices were generated for the most competitive open-source PRMs across all problem categories. Every model appears to struggle the most at correctly identifying invalid steps as such,

with **PROVER**-7B performing the best overall. Despite its relatively good performance, it still does not perform better than a random coin flip. The complete set of matrices can be found in §A.6.

5 DISCUSSION

Effect of cascading errors on score. The formula for PRMScore as defined in Song et al. (2025) is:

$$PRMScore = w_1 * F1_{neg} + w_2 * F1$$

where $F1$ refers to the F1 score of valid labels, and $F1_{neg}$ the invalid labels. Based on the values in their data tables, we can solve for both coefficients to get $w_1 = 0.5$ and $w_2 = 0.5$. F1 score is defined as:

$$F1 = \frac{2TP}{2TP + FP + FN}$$

with TP standing for the number of true positives, FP for false positives, and FN for false negatives.

Due to our "cascading" policy of labeling all steps past the first error as erroneous in the training data, it is expected that the model will learn to predict a greater number of false negatives. In terms of calculating $F1$, this results in a smaller value. $F1_{neg}$ would also shrink because of an increased number of false positives. Therefore, it should come as no surprise that **PROVER** has a middling PRMScore while having best-in-class performance on other metrics. Note that other benchmarks like Qwen's ProcessBench (Zheng et al. (2025)) subscribe to the same belief as us and focus on identifying *only* the first error in the reasoning process.

Analysis of failure modes. The confusion matrices in §A.6 indicate that *every* PRM tested does well at identifying ground-truth correct steps in a process. Most often, problems belonging to the Soundness category have their correct steps confused for being incorrect. This category most thoroughly audits the model's ability to produce accurate rewards, lending itself to more dubious and hard-to-verify claims that an unsophisticated model would be tempted to label as wrong—it also happens to have the highest rate of true negatives. On the other hand, every PRM does poorly at labeling ground-truth incorrect steps. Most other PRMs seem to have an overall bias towards labeling things as correct (Song et al. (2025)), which partly explains why this happens. Our cascading error policy introduces a bias of the opposite kind—which might explain why **PROVER** has the second-highest overall false negative rate—but it also helps our false positive rate stay beneath that of other models. Of course, this bias is not helpful if it is merely trading one kind of mislabeling for another, and it is entirely possible that other factors are at play too; further ablation study is needed for us to draw a conclusive answer.

Comparison with Qwen2.5-Math-PRM-7B. The Qwen2.5-Math-PRM model was fine-tuned on OpenAI's gold-standard PRM800K dataset (Zhang et al. (2025)), and Qwen2.5-Math-PRM-7B and **PROVER**-7B are trained from the same base model (Qwen2.5-Math-7B-Instruct), effectively making Qwen's PRM a control for our experiment. Comparing the two head-to-head, it appears that **PROVER**-7B is dramatically better at first error accuracy in all problem categories while experiencing only a slight drop in PRMScore. This tracks with **PROVER**-7B's relatively high lead (its tendency to predict errors one or more steps before the ground-truth first error) and the cascading error policy used in its training data (meaning the steps after the first error are more likely to be considered erroneous themselves). It is also worth noting that **PROVER**-7B performs much more consistently in first error distance across problem categories compared with Qwen's PRM—we are *much more precise, but only marginally more accurate*—which suggests that while the inclusion of Lean formalization helps with consistency, there are still unresolved issues with step number alignment between the Lean code and natural language descriptions.

Ablation study. We trained **PROVER**-7B before and after performing post-filtering on the training data and ran the same set of experiments on both; the results are shown in Tables 4 and 5. Post-filtering causes first error accuracy to drop across all problem categories, which may seem like a bad thing at first, but this one metric alone does not give us the full picture. Indeed, we simultaneously see a universal rise in PRMScore and a significant drop in first error distance, which indicates that post-filtering is a net improvement for a model that is otherwise overly eager to predict negative labels. The changes in lead/lag percentages partially explain the drop in first error distance; the model

	Overall		Simplicity		Soundness		Sensitivity*	
Post-Filtering?	Score	First	Score	First	Score	First	Score	First
No	54.3	**79.1%**	55.5	**66.5%**	53.8	**88.3%**	54.6	**73.2%**
Yes	**64.4**	61.0%	**59.3**	36.2%	**66.7**	76.1%	**64.0**	55.5%

Table 4: Effect of post-filtering on PRMScore (↑) and first error accuracy (↑). First place is bolded.
*Excludes multi-solution consistency scenarios.

Post-Filtering?	Overall	Simplicity	Soundness	Sensitivity*	Lead/Lag %
No	4.729	3.563	5.498	4.356	74.0% / **8.0%**
Yes	**3.731**	**3.515**	**3.913**	**3.582**	**50.9%** / 21.0%

Table 5: Effect of post-filtering on mean first error distance (↓) and lead/lag frequency. First place is bolded.
*Excludes multi-solution consistency scenarios.

stopped predicting errors before they actually happened as often, with the side effect of somewhat increasing the frequency with which it predicted errors after they happened.

Limitations. As discussed previously, the deliberate decision to use a cascading error policy is inherently going to limit our PRMScore because it will invariably cause the model to learn some amount of false negative labels.

Additionally, step-level alignment between Lean code and natural language descriptions remains an open challenge, because not every line of Lean code maps one-to-one with a natural language step. We do our best to decompose individual lines of Lean code into smaller steps when possible using the Trace module, but this is only applicable for a select number of tactics, like `nlinarith`. The main goal is to avoid having the LLM do any guesswork, which could introduce hallucinations into the training data. We prohibit the use of `nlinarith` in new candidates to try and avoid this problem altogether, though the tactic is still ubiquitous in the Lean-workbook-proofs dataset, which remain represented to an extent in the training data for the sake of diversity.

Finally, the errors that we introduce into candidate Lean solutions via prompting (§A.5) are mostly not "organic" in the sense that the LLM hallucinated them; they come from a list of predefined categories. About 1 in 10 candidate proofs from Claude *did* have a naturally-occurring error, but this wasn't enough. Another reason for using prompting is that it was difficult to induce hallucinations in the Lean code (say, by using a weaker model) without also introducing syntax errors, rendering it unusable by Comparator. The predictable nature of these deliberate reasoning errors could have detrimental effects on the PRM's ability to recognize more subtle, out-of-distribution errors in especially complex reasoning traces.

5.1 Future Work

Investigating possible scaling laws. With the resources available to us, we were only able to train a PRM with 7B parameters. It is encouraging to see that it already performs better than other 7B and even 34B models on some metrics. Still, it would be of great interest to see if our results are reproducible on a larger scale. At the very least, we would be able to infer whether **PROVER** maintains its lead over comparable models or if it has diminishing returns as the base models get larger.

Extension to coding domains. Lean is capable of formally verifying the outputs of computer programs, too, as has been done with zlib to ensure bug-free operation (kim em (2025)). Given that coding agents are already widely adopted, and the security of AI-generated code has understandably come under scrutiny, a similar framework as ours for the coding domain would certainly be of great use.

Choice of LLMs. We are limiting our dataset's diversity by only using Claude Opus 4.6 to generate new proofs, but that was the only frontier model available to us at the time. Ideally, we would use multiple proprietary and open-source models to sample candidate proofs. It may also be worth trying an agentic setup such as Leanstral (Mistral AI (2026)) to interactively fix syntax errors as they arise,

up to a certain amount of attempts. The goal is to consistently generate noisy Lean code with the occasional reasoning error but no syntax errors.

Further ablation study. In addition to training a PRM with and without the post-filtering step, it would be worthwhile to experiment with omitting other steps of the pipeline, such as reducing the amount of candidate proofs per problem, removing the cascading error behavior, or not changing the distribution of the source dataset. This way, we would have a better idea of what pipeline stages work as intended.

Different seed dataset. Much of our pipeline was motivated by a desire to compensate for the varying quality of proofs that were found in Lean-workbook-proofs. When this dataset was created, it was assumed that any Lean proof which compiled was flawless, but this does not account for the more insidious errors that we found, like vacuously correct proofs resulting from unintended integer division in fractional exponents, or misalignment between natural language descriptions and opaque tactics like `nlinairth` (which is often the only step needed to prove an inequality). One particular case of this is described in §A.2. If another large-scale dataset with stricter quality control were to come about, it would be worthwhile to try using it as our starting point.

Test other benchmarks and downstream applications. PRMBench is not the be-all and end-all of evaluating reward models. For one, ProcessBench could have provided more relevant feedback on our model, given that it also emphasizes identification of the *first* error (Zheng et al. (2025)). Better yet, the truest evaluation is to use the model for its intended purpose: to make a larger model better at math, either by post-training the larger model through RL, or reranking candidate solutions from the larger model at test-time. Unfortunately, we did not have the resources or time to perform such an experiment, especially given the difficulty of getting RL to converge.

6 Conclusion

PROVER is a framework for training Process Reward Models for math from grounded, verifiable proofs, all while requiring zero human intervention. It works by performing LLM-based fuzzing against the Lean language's Comparator tool to generate high-quality positive and negative examples with detailed step-by-step correctness labels. It derives an informal natural language description from every step of Lean code, making it a drop-in replacement for existing datasets like PRM800K (Lightman et al. (2023)) and Math-Shepherd (Wang et al. (2024)). When tested on PRMBench, a benchmark for process reward models, **PROVER** matches or outperforms other open-source PRMs in its class in several key metrics, and is even competitive with much larger models. Ultimately, **PROVER** shows that fully automated math annotation is possible without sacrificing the quality of the reward signal, paving the way for further work into scalable oversight.

Acknowledgments

I would like to extend my thanks to Professors Pavel Izmailov and Tengyu Ma for sharing their AI and math expertise, respectively, which gave this project a clear direction. Additionally, I thank Salman Rahman for inviting me to this project, giving me guidance on the day-to-day work, proofreading the manuscript, and his general advice about grad school. I would also like to recognize Rodney Lafuente-Mercado for his mathematical knowledge, refinements to the codebase, and prompting improvements, which all played a role in the final pipeline. Finally, I thank my advisor, Professor Saadia Gabriel, for being the chair on my Capstone Committee, and providing the computing resources I needed to run both training and local inference.

REFERENCES

Axiom Math. Axiom lean engine (axle). https://axle.axiommath.ai/v1/docs/, 2026.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp (eds.), *Automated Deduction - CADE-25*, pp. 378–388, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

kim em. lean-zip: Formally verified implementation of zlib in lean 4. https://github.com/kim-em/lean-zip, 2025.

Lean Prover Community. Comparator. https://github.com/leanprover/comparator, 2025.

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, 2023. URL https://arxiv.org/abs/2305.20050.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover: A frontier model for open-source automated theorem proving, 2025a. URL https://arxiv.org/abs/2502.07640.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025b. URL https://arxiv.org/abs/2508.03613.

Mistral AI. Leanstral: Open-source foundation for trustworthy vibe-coding. https://mistral.ai/news/leanstral/, 2026.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024. URL https://arxiv.org/abs/2305.18290.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

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

Mingyang Song, Zhaochen Su, Xiaoye Qu, Jiawei Zhou, and Yu Cheng. Prmbench: A fine-grained and challenging benchmark for process-level reward models, 2025. URL https://arxiv.org/abs/2501.03124.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations, 2024. URL https://arxiv.org/abs/2312.08935.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Zheng Yuan, Dahua Lin, and Kai Chen. Lean workbook: A large-scale lean problem set formalized from natural language math problems, 2025. URL https://arxiv.org/abs/2406.03847.

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, 2025. URL https://arxiv.org/abs/2501.07301.

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, 2025. URL https://arxiv.org/abs/2412.06559.

A APPENDIX

A.1 LEAN-WORKBOOK-PROOFS SAMPLE

The following is taken from example **lean_workbook_plus_59092** in Lean-workbook-proofs.

```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

/- What is the value of $p + q + r + s$ when $(p, q, r, s) = (0, 0, 0, 0)$? -/
theorem lean_workbook_plus_59092 (p q r s : ℕ) : (p, q, r, s) = (0, 0, 0, 0) →
p + q + r + s = 0 := by
/-
Given that \( (p, q, r, s) = (0, 0, 0, 0) \), we need to show that
\( p + q + r + s = 0 \). By substituting the values \( p = 0 \), \( q = 0 \),
\( r = 0 \), and \( s = 0 \) into the expression \( p + q + r + s \), we obtain
\( 0 + 0 + 0 + 0 \), which simplifies to \( 0 \).
-/
-- Introduce the assumption that (p, q, r, s) = (0, 0, 0, 0)
rintro ⟨rfl, rfl, rfl, rfl⟩
-- Simplify the expression p + q + r + s using the fact that p = 0, q = 0, r = 0, s = 0
simp
```

Using our formal definitions:

$p_{\mathcal{I}}$ = "What is the value of $p + q + r + s$ when $(p, q, r, s) = (0, 0, 0, 0)$?"

$p_{\mathcal{F}}$ = "`theorem lean_workbook_plus_59092 (p q r s : ℕ) : (p, q, r, s) = (0, 0, 0, 0) → p + q + r + s = 0`"

$S_{\mathcal{I}}$ = {"Given that $(p, q, r, s) = (0, 0, 0, 0)$, we need to show that $p + q + r + s = 0$. By substituting the values $p = 0$, $q = 0$, $r = 0$, and $s = 0$ into the expression $p + q + r + s$, we obtain $0 + 0 + 0 + 0$, which simplifies to 0.", "Introduce the assumption that (p, q, r, s) = (0, 0, 0, 0)", "Simplify the expression p + q + r + s using the fact that p = 0, q = 0, r = 0, s = 0"}

$S_{\mathcal{F}}$ = {`rintro ⟨rfl, rfl, rfl, rfl⟩, simp`}.

A.2 Regarding Poor NL Descriptions from Lean-workbook-proofs

The following is taken from example **lean_workbook_plus_9950** in Lean-workbook-proofs.

```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

/- If $a, b, c>0$ prove that \n $\frac{a}{a+b}+\frac{b}{b+c}+\frac{c}{c+a}+
\frac{8abc}{(a+b)(b+c)(c+a)}\le\frac{5}{2}$ -/
theorem lean_workbook_plus_9950 (a b c : ℝ) (ha : 0 < a) (hb : 0 < b) (hc : 0 < c) :
(a / (a + b) + b / (b + c) + c / (c + a) + (8 * a * b * c) / ((a + b) * (b + c) *
(c + a))) ≤ 5 / 2 := by
/-
To prove the inequality \(\frac{a}{a+b}+\frac{b}{b+c}+\frac{c}{c+a}+\frac{8abc}
{(a+b)(b+c)(c+a)}\le\frac{5}{2}\) for \(a, b, c > 0\), we start by simplifying
the expression using algebraic manipulations. We then apply the inequality by
considering the non-negativity of squares and using arithmetic operations to
bound the expression.
-/
have h₀ : 0 < a * b := by positivity
have h₁ : 0 < b * c := by positivity
have h₂ : 0 < c * a := by positivity
have h₃ : 0 < (a + b) * (b + c) := by positivity
have h₄ : 0 < (c + a) * (a + b) := by positivity
have h₅ : 0 < (b + c) * (c + a) := by positivity
field_simp
rw [div_le_div_iff] <;>
nlinarith [sq_nonneg (a - b), sq_nonneg (b - c), sq_nonneg (c - a),
sq_nonneg (a * b - b * c), sq_nonneg (b * c - c * a), sq_nonneg (c * a - a * b)]
```

One issue with this proof is that it is missing line-by-line explanations, like those present in §A.1, meaning that our formal verifier would have to guess the step number alignment between the natural language and Lean tactics. More importantly, the explanation itself is inadequate; there are critical unjustified claims about the existence and form of a sum-of-squares representation without any verification. The `theorem` statement requires us to prove an upper bound of the expression $\frac{a}{a+b} + \frac{b}{b+c} + \frac{c}{c+a} + \frac{8abc}{(a+b)(b+c)(c+a)}$, yet the final step provides the expressions $(a-b)^2$, $(b-c)^2$, $(c-a)^2$, $(ab-bc)^2$, $(bc-ca)^2$, and $(ca-ab)^2$ to `nlinarith` without stating their relevance. Because of issues like this that we found in the dataset, we chose to generate our own proofs and descriptions with a stronger proprietary model instead.

A.3 CANDIDATE GENERATION PROMPT

System prompt:

```
You are an expert Lean 4 and mathematics proof writer.
Generate Lean proofs for dataset construction.  Each proof must be a sequence
of small tactic steps, and every step must have a matching mathematical natural
language explanation that does not mention Lean syntax, tactic names, formal
verification, or code.

The Lean code and the natural-language proof must be faithful to each other:
the Lean proof should implement the same mathematical argument described in the
natural language, not skip over several natural-language claims with one opaque
automation call.

Do not ban arithmetic discharge tactics outright, but do not use `linarith`,
`nlinarith`, or `omega` as opaque one-shot proofs of the main argument.  If one
of these tactics is used, the preceding steps must expose the concrete
mathematical facts, identities, cases, or inequalities it relies on.  Never use
`sorry`, `admit`, or any placeholder proof term.

Two hard requirements for the natural language:

1. Each NL description must be 1:1 with its corresponding Lean step --
   describing exactly what that step does mathematically, not a higher-level
   summary and not a vague gloss.
2. Concatenating all NL descriptions in order must read as a complete,
   standalone mathematical proof that a student could have written and a
   mathematician could follow without ever seeing the Lean code.  It must not
   read like comments about code or a list of tactic explanations.
```

User prompt:

```
Generate one independent proof attempt for this Lean theorem:

'''lean
[[[LEAN THEOREM STATEMENT]]]
'''

Requirements:
- Use Lean 4 with Mathlib.
- Keep the theorem name and statement exactly as given.
- Do not use `linarith`, `nlinarith`, or `omega` as opaque one-shot proofs.
  They are allowed only as final arithmetic discharge after the preceding steps
  have made the algebraic situation, case split, divisibility fact, modular
  computation, or inequality combination explicit enough that the NL remains
  1:1 and mathematically informative.
- Do not use `sorry`, `admit`, `by_contra!` as a placeholder, or any unfinished
  proof placeholder.
- Do not use one opaque tactic to hide the main argument.  If a proof needs
  algebra, inequalities, case splits, contradiction, divisibility facts, or
  modular arithmetic, introduce those facts as explicit intermediate `have`,
  `calc`, `rw`, or case-analysis steps before any final arithmetic discharge.
- The proof must be step-by-step: step 0 is the theorem line ending in `:= by`,
  and each later step is one Lean tactic or one tightly scoped Lean proof line.
- Each step must include a natural-language mathematical explanation matching
  that exact Lean step 1:1 (see rules and example below).
- Natural language must explain the mathematics only.  Do not mention Lean,
  tactics, automation, syntax, or the compiler.
- The natural language must not describe a stronger or more detailed argument
  than the Lean line actually proves.  If the NL says "we write k = 4q + 1",
  the Lean line must actually establish that statement in the corresponding
  case; if the NL cites an algebraic identity, the Lean line must establish that
  identity or use it directly.
- It is acceptable if the proof attempt is mathematically wrong, but avoid
```

```
  syntax errors and malformed Lean where possible.
- Return only JSON.  No Markdown fences.

Natural-language rules:
- 1:1 alignment with the Lean step - describe exactly what that step does.
- All steps concatenated in order must read as a standalone mathematical
  proof a student could have written (no gaps, no references to code).
- Do not write comments about the formal proof.  Write the mathematical proof
  itself.  If the Lean code were stripped out, the NL should still read
  naturally from beginning to end.
- Prefer concrete inequalities, computations, named lemmas - not generic filler.
- For opaque arithmetic steps, spell out the exact identity, contradiction,
  inequality combination, or modular case being used.  If you cannot state that
  exact mathematical content, split the proof into smaller explicit steps.
- Avoid vacuous or degenerate proofs unless the hypotheses are genuinely
  contradictory; when using contradiction, explicitly derive and explain the
  contradiction before concluding.

Few-shot guide for the expected level of detail and 1:1 alignment:

{
  "id": "lean_workbook_19000_1",
  "lean_statement": "theorem lean_workbook_19000 (x : ℝ) (hx : x ≥ 3) :
  (x - 3) * (x^3 + 3 * x^2 + 9 * x - 27) ≥ 0 := by",
  "proof_index": 0,
  "steps": [
    {
      "step": 0,
      "lean_code": "theorem lean_workbook_19000 (x : ℝ) (hx : x ≥ 3) :
      (x - 3) * (x^3 + 3 * x^2 + 9 * x - 27) ≥ 0 := by",
      "natural_language_description": "We must prove: for any real x with
      x ≥ 3, the quantity (x - 3)(x^3 + 3x^2 + 9x - 27) is nonnegative.",
      "label": "correct"
    },
    {
      "step": 1,
      "lean_code": "  have h1 : x - 3 ≥ 0 := by linarith",
      "natural_language_description": "From x ≥ 3, we immediately obtain
      x - 3 ≥ 0.",
      "label": "correct"
    },
    {
      "step": 2,
      "lean_code": "  have h2 : x + 3 ≥ 0 := by linarith",
      "natural_language_description": "Since x ≥ 3, we also have x + 3 ≥ 3
      + 3 = 6 ≥ 0.",
      "label": "correct"
    },
    {
      "step": 3,
      "lean_code": "  have h3 : x ^ 2 ≥ 0 := sq_nonneg x",
      "natural_language_description": "For any real x, its square is
      nonnegative: x^2 ≥ 0.",
      "label": "correct"
    },
    {
      "step": 4,
      "lean_code": "  have h4 : (x - 3) * x ^ 2 * (x + 3) ≥ 0 := mul_nonneg
      (mul_nonneg h1 h3) h2",
      "natural_language_description": "The product (x - 3)·x^2·(x + 3) is
      nonnegative because it is a product of three nonnegative factors (x - 3
      ≥ 0, x^2 ≥ 0, x + 3 ≥ 0).",
      "label": "correct"
    },
    {
```

```
      "step": 5,
      "lean_code": "  have h5 : (x - 3) ^ 2 ≥ 0 := sq_nonneg (x - 3)",
      "natural_language_description": "Also, (x - 3)^2 ≥ 0, since a square is
      always nonnegative.",
      "label": "correct"
    },
    {
      "step": 6,
      "lean_code": "  nlinarith [h4, h5, sq_nonneg x, sq_nonneg (x - 3)]",
      "natural_language_description": "Note the algebraic decomposition: x^3 +
      3x^2 + 9x - 27 = x^2(x + 3) + 9(x - 3). Hence (x - 3)(x^3 + 3x^2 + 9x - 27)
      = (x - 3)x^2(x + 3) + 9(x - 3)^2. By steps 4 and 5, each term on the right
      is ≥ 0, so their sum is ≥ 0. Therefore (x - 3)(x^3 + 3x^2 + 9x - 27) ≥ 0.",
      "label": "correct"
    },
    {
      "step": 7,
      "lean_code": "--     [linarith]  Invoking oracle\\n--       [linarith]
      found a contradiction: [(1, 9), (7, 1), (9, 1)]",
      "natural_language_description": "Equivalently, if one were to assume for
      contradiction that (x - 3)(x^3 + 3x^2 + 9x - 27) < 0 while keeping h4 ≥ 0
      and h5 ≥ 0, then combining the identity above with these nonnegativities
      would contradict that assumption.",
      "label": "correct"
    },
    {
      "step": 8,
      "lean_code": "--     [linarith]  Building final expression\\n--
      [linarith] \\n--             9 * (0 - (x - 3) ^ 2) + (0 - (x - 3) * x ^ 2 *
      (x + 3)) + ((x - 3) * (x ^ 3 + 3 * x ^ 2 + 9 * x - 27) - 0)\\n--
      should be both 0 and negative",
      "natural_language_description": "Indeed, from (x - 3)(x^3 + 3x^2 + 9x - 27)
      = (x - 3)x^2(x + 3) + 9(x - 3)^2 with both summands ≥ 0, the left-hand side
      cannot be negative. This confirms the desired inequality.",
      "label": "correct"
    }
  ]
}
```

Use this as the standard for detail: each NL line states the concrete mathematical claim behind the corresponding Lean line, and opaque algebra is expanded into the actual identity or contradiction being used. Do not write generic summaries like "algebra finishes the proof", "standard processing applied", or "the final transformation is resolved".

The final comment-only steps in this example show how trace-derived comments can be explained when they are already available. Do not fabricate trace comments in new proofs. If no trace comments are provided in the input, output only the theorem line and executable proof lines.

JSON schema:

```
{
  "steps": [
    {
      "step": 0,
      "lean_code": "[[[LEAN THEOREM STATEMENT]]] := by",
      "natural_language_description": "We prove ... "
    },
    {
      "step": 1,
      "lean_code": "  ...",
      "natural_language_description": "..."
    }
  ]
}
```

A.4 PROOF INFORMALIZATION PROMPT

User prompt:

````
You are a mathematics expert. Convert the Lean 4 proof below into a
step-by-step natural-language mathematical proof.

You are given:
- **Theorem statement** - the precise claim to prove.
- **Full Lean proof** - the complete tactic proof, possibly annotated
with tactic-state comments (`-- STATE BEFORE/AFTER`, `-- MATHLIB LEMMAS
USED`, `-- SIMP TRACE`).

Your output is a JSON array of proof steps. **You choose the granularity**:
we encourage highly granular natural-language steps. It is totally okay to
repeat the exact same line(s) of Lean code across multiple JSON steps if the
mathematical explanation for that code requires multiple steps. You are not
required to emit one step per line of Lean code.

## Rules

1. **Step 0 is the theorem statement.** Its `lean_code` is the verbatim
`theorem ...` line(s); its `natural_language_description` states precisely what
is to be proved.
2. **Every subsequent step** pairs a contiguous block of Lean code (copied
verbatim) with a rigorous mathematical explanation of what that block
accomplishes. Do not invent steps with no Lean code.
3. **Read state annotations carefully.** `-- STATE BEFORE` / `-- STATE AFTER` /
`-- SIMP TRACE` lines are machine-verified; use them to describe exactly what
the tactic proves or rewrites. Include these comment lines in the `lean_code`
of the step they annotate.
4. **No Lean jargon in natural language.** Never mention tactic names, Lean
syntax, or formal verification. Write as a mathematician would on paper.
5. **Mathematical correctness is mandatory.** Every claim must be justified;
every identity or inequality must be verified or explained. Prefer a correct
high-level argument over detailed but wrong algebra.
6. **Copy Lean code verbatim.** Do not alter, reformat, or paraphrase any Lean
code.
7. **When read in order, the natural-language descriptions must form a
complete, standalone proof** a PhD examiner would accept as rigorous.

## Output format

Return **only** a JSON array - no text before or after it.
```json
[
  {
    "step": 0,
    "lean_code": "<theorem statement, verbatim>",
    "natural_language_description": "<precise statement of what is proved>"
  },
  {
    "step": 1,
    "lean_code": "<one or more contiguous Lean lines, verbatim>",
    "natural_language_description": "<rigorous mathematical explanation>"
  }
]
```

**Theorem statement:**
```
[[[LEAN THEOREM STATEMENT]]]
```

**Full Lean proof (with tactic-state annotations):**
````

```
```lean
[[[LEAN CODE AND TRACES]]]
```

Provide the JSON output.
```

A.5 Deliberate Error Injection Prompt

The following is appended to the user prompt in §A.4 to generate incorrect examples:

```
Intentional incorrect-proof mode:
- Generate a proof attempt that is syntactically well-formed Lean, but contains
  exactly one genuine mathematical reasoning error after a plausible correct
  prefix.
- The first wrong step should be a Lean line that tries to prove a false or
  unjustified mathematical claim and therefore fails verification as a proof
  error, not as malformed syntax.
- The natural language for that step must faithfully describe the same wrong
  mathematical claim.  Do not make the NL correct while the Lean line is merely
  too weak; the negative label should correspond to a real mathematical mistake.
- Prefer plausible errors such as an invalid inequality direction, an
  unjustified divisibility/modular conclusion, a missing hypothesis in a case,
  an algebraic identity with an incorrect sign or coefficient, or applying a
  theorem outside its conditions.
- Especially prefer non-inequality reasoning errors when the theorem permits
  them: a wrong modular residue, a bad divisibility witness, an invalid parity
  case, an unjustified existential witness, or a false algebraic identity.
- Keep all steps before the first wrong step correct, explicit, and aligned.
- Do not use `sorry`, `admit`, placeholders, malformed Lean, unknown constants,
  or deliberately broken syntax.
- Avoid examples where the natural language is a valid mathematical proof but
  the Lean line fails only because the automation or formalization is missing
  an intermediate fact.
- After the wrong step, you may include a few continuation steps that depend on
  the wrong claim; their NL should remain faithful to the erroneous path.

Requested reasoning-error style: [[[ERROR TYPE]]]
- [[[ERROR INSTRUCTIONS]]]
- Use this style when it naturally fits the theorem; otherwise use the closest
  plausible student-like reasoning error.
```

The types of errors are:

- `theorem_misapplication`: Apply a true theorem while missing one of its hypotheses or using it outside its conditions.
- `wrong_rewrite_direction`: Rewrite an implication, equality, or inequality in an invalid direction.
- `missing_case_or_side_condition`: Ignore a necessary case, nonzero condition, positivity condition, or range restriction.
- `modular_or_divisibility_error`: Use a wrong residue, parity split, divisibility witness, or modular conclusion.
- `bad_witness_or_construction`: Choose an existential witness or constructed object that almost works but violates a required property.
- `algebraic_identity_error`: Use a false expansion, factorization, sign, coefficient, or algebraic identity.
- `automation_misuse`: Use arithmetic automation only after stating an explicit but false arithmetic or nonlinear premise.

A.6 Confusion Matrices