Tool-Assisted Multi-Turn Theorem Proving with LLMs

Kanan Gupta[*]
University of Pittsburgh
kanan.g@pitt.edu

Jannis Limperg
AWS Agentic AI
limperg@amazon.de

Udaya Ghai
AWS Agentic AI
ughai@amazon.com

Abstract

We present an approach for training language models to interactively prove theorems using the Lean proof assistant. Our approach enables models to propose partial proofs, receive verification feedback, and iteratively refine their proofs. We develop a synthetic data generation pipeline that converts static proof datasets into multi-turn interactive sequences, complete with incremental verification feedback. Our pipeline also incorporates error fixing examples that teach models to identify and correct mistakes based on verification results. Models fine-tuned on this synthetic data demonstrate improvements over base models, showing increased success rates in proof completion and improved ability to leverage verification feedback.

1 Introduction

Proof assistants like Lean 4 [6] offer verifiable correctness for mathematical reasoning, addressing a key weakness of large language models (LLMs). However, a common paradigm for LLM-based formal theorem proving is to generate proofs in a single shot, neglecting the proof assistant's rich feedback and diverging from the workflow of human experts. Humans generally do not write complex Lean proofs in one go; they strategically break down problems into intermediate lemmas, building an outline of the argument before formalizing their intuitions and proving all the lemmas. Our initial evaluations (Appendix B) show that specialized prover models like DeepSeekProver-v2-7B [20] excel at single-shot proof generation, but overwhelmingly fail to incorporate feedback. Meanwhile, large generalist models like Claude 3.7 Sonnet show some ability to use verifier feedback but have limited Lean capabilities. This highlights the need for specialized models explicitly trained for interactive, feedback-driven proving.

This work investigates training language models to prove theorems through an incremental, verifier-guided process. We introduce a data pipeline that transforms static proofs into training data for LLM-verifier interactions, including synthetic error-recovery scenarios. To measure our contribution, we evaluate models on miniF2F [27], a benchmark of competition math problems formalized in Lean. The evaluation is done in a multi-turn setting where the models get up to 8 attempts to complete a proof, while receiving verifier feedback for each attempt. Our fine-tuning process shows a clear progression: while a baseline trained on static proofs improves accuracy from the base model's 6% to 38.9%, it fails to utilize the interactive steps effectively. After training on data from our pipeline, the model's accuracy increases to 58.6% due to its improved ability to generate proofs incrementally and to successfully correct errors across multiple turns. Accuracy increases to 68.9% when using a hybrid strategy that runs four independent 8-turn interactive sessions in parallel for each problem, succeeding if any of the four runs finds a proof.

Recent state of the art theorem proving models like Goedel-Prover-v2 [16] and Kimina-Prover [22] achieve over 80% pass@32 accuracy on miniF2F [27] through careful curriculum curation and reinforcement learning. They, too, incorporate some proof refinement, but their approach is limited to

[*]Work done during internship at Amazon.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

correcting whole proofs. Our work provides evidence that building a proof over multiple turns and verifying incomplete states is a powerful training signal. We believe that combining this interactive paradigm with state-of-the-art curriculum learning and reinforcement learning is likely to further advance automated theorem proving. We provide an overview of related work in Appendix A.

2 Data Generation Pipeline

A key challenge for LLM-based theorem proving is the lack of suitable training data. Our core contribution is a pipeline that transforms static Lean proofs into multi-turn interactive dialogues. This process, summarized in Figure 1, involves three main stages: decomposing correct proofs, generating retrospective reasoning, and synthesizing error-recovery scenarios.

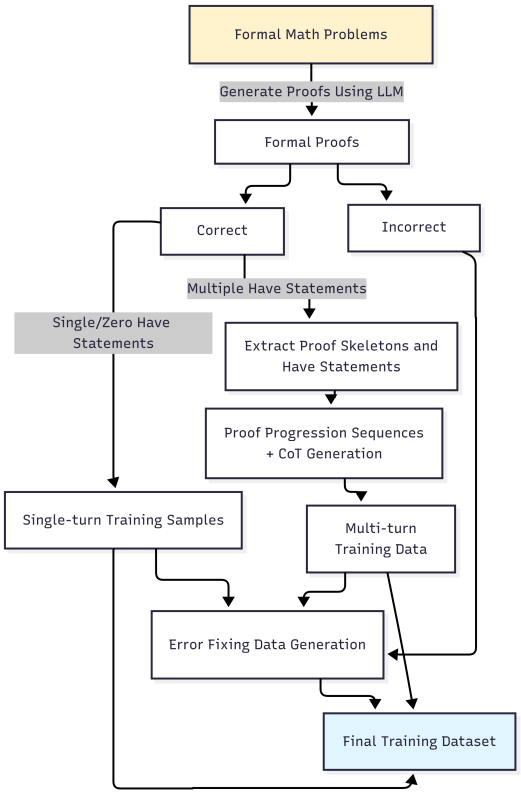


Figure 1: Multi-turn training data pipeline.

Proof Decomposition. We generated a dataset of about 120k correct proofs for problems from FineLeanCorpus [19], a dataset of mathematical problems formalized in Lean, using Goedel-Prover-v2-32B [16]. Of these proofs, 40k (35%) contain only 0 or 1 intermediate lemmas (`have` tactics). We used these as single-turn examples to maintain the model's ability to generate complete proofs directly.

The other 75k proofs, containing multiple `have` statements each, were used to create multi-turn data, yielding approximately 49k 2-step, 21k 3-step, and 5k 4+-step proofs. We decomposed these proofs into progression sequences. Each such sequence begins with a *proof skeleton* containing only the top-level lemmas (`have` statements), with their proofs replaced by `sorry`. Each subsequent step in the sequence then proves a few of these lemmas at a time, teaching the model to separate strategic planning (designing a valid skeleton) from tactical execution (proving each individual subgoal). By design, each partial proof in this sequence successfully passes the Lean verifier, confirming that there are no errors, but only the final, `sorry`-free proof is considered complete.

Retrospective Chain-of-Thought. We then generated chain-of-thought (CoT) sections simulating the model's decision-making. This was done retrospectively by providing Qwen3-235B-A22B with consecutive proof steps to produce explanatory reasoning connecting them, creating forward-looking reasoning. We created two versions: long CoT, combining Qwen3's synthetic reasoning with the original CoT from Goedel-Prover-v2-32B for the first step with the proof skeleton, and short CoT, using Qwen3 only. Both versions used exclusively Qwen3's synthetic CoT (which is much shorter than Goedel-Prover-v2's) for proof-filling steps after the first step. In comparative experiments on independently fine-tuned checkpoints, long CoT demonstrated better performance, so we use that version of the dataset.

Synthetic Error Fixing Data. To teach the model how to correct proofs based on verifier feedback, we generated a dataset of about 27k error-correction examples, using three methods.

(i) Whole proof correction: For our initial dataset of proofs, we used Goedel-Prover-v2 to generate multiple candidate solutions for each problem, then checked them for correctness. From these, we paired incorrect proof attempts with the corresponding correct proofs of the statements for about 13k samples from our single-turn data.

(ii) Skeleton correction: We further selected those incorrect proof samples where the extracted proof skeleton itself was incorrect, and combined them with the corresponding correct multi-turn

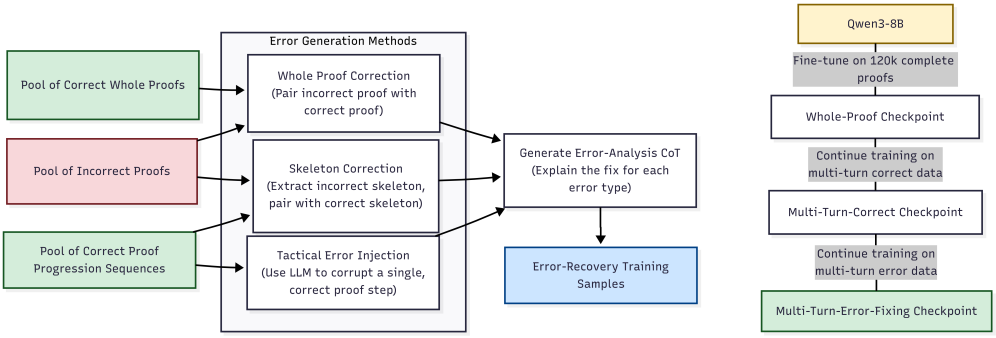


Figure 2: *Left*: Error-fixing data pipeline. *Right*: Training flow.

proofs. This yielded about 2k training examples simulating scenarios where the model generates an incorrect skeleton, receives verifier errors, then generates a correct skeleton, and fills in the missing subproofs in subsequent turns.

(iii) Tactic correction: Since localized tactical errors were scarce in our dataset, we created 12k tactical error correction examples by using Qwen3-235B-A22B to corrupt intermediate steps in multi-turn sequences. We prompted Qwen3 with comprehensive error type guidelines and instructions to avoid introducing errors in overly simple steps. We provided it consecutive proof steps from the multi-turn proof progression sequences, instructing it to inject errors only into the new tactics introduced in the later step to preserve data flow. Each corrupted proof was verified using Lean to ensure it produced errors and to capture error messages, generating approximately 12k tactical correction examples.

For each of these scenarios, we also generated CoT using Qwen3-235B-A22B as before. The CoT analyzes the verifier's error message and explains the reasoning behind the fix.

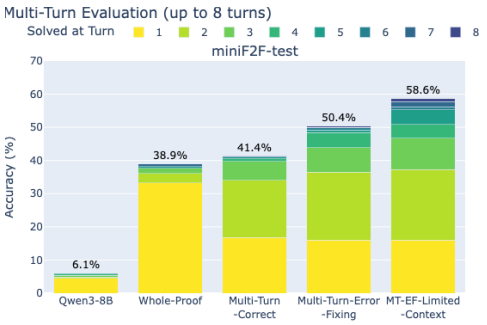
3 Experiments and Results

Experimental Setup. We fine-tuned Qwen3-8B, which is an instruction-tuned model, on the dataset we generated, using phased training to isolate the impact of each type of data. This led to three model checkpoints:

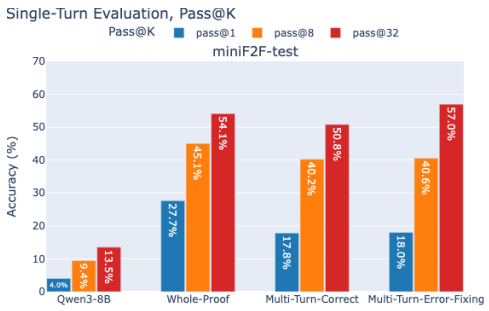
- *Whole-Proof*: We first fine-tuned Qwen3-8B directly on the original 120k Goedel-v2 proofs without modification, establishing baseline whole proof generation capabilities.
- *Multi-Turn-Correct*: Next, we trained the checkpoint from the previous step on our multi-turn dataset with only correct proofs (75k multi-turn progression sequences + 40k single-turn examples with long CoT), teaching incremental proof construction through skeleton generation followed by systematic filling.
- *Multi-Turn-Error-Fixing*: Finally, we added 27k error-correction examples while retraining on the complete multi-turn dataset. In this phase, we had 18% error-correction samples and 82% purely correct proofs.

Since much of the multi-turn data teaches skeleton generation first, there is a risk of the fine-tuned model producing excessive `sorry` statements without the ability to complete them correctly. Similarly, error-correction examples could condition the model to always generate incorrect proofs before corrections. This phased training approach allows us to evaluate whether specific data types help or hurt the model's performance. Appendix C contains the training and evaluation parameters.

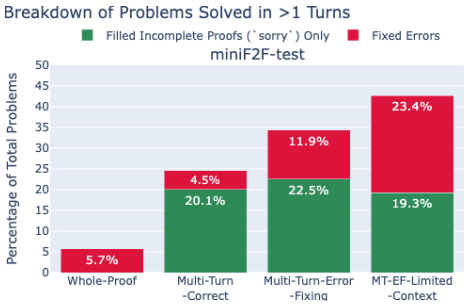
We evaluated the fine-tuned checkpoints, alongside Qwen3-8B, on miniF2F in a multi-turn interactive setting with up to 8 turns. The models generate a solution, receive any error messages from Lean, and continue until they exhaust all 8 turns or solve the problem, while maintaining the full conversation history in their context. To investigate the impact of the resulting long context, we also evaluated the Multi-Turn-Error-Fixing checkpoint in a different setting, labeled *MT-EF-Limited-Context*: Instead of the full conversation history, the model sees only the problem statement, its most recent output, and the verifier response.



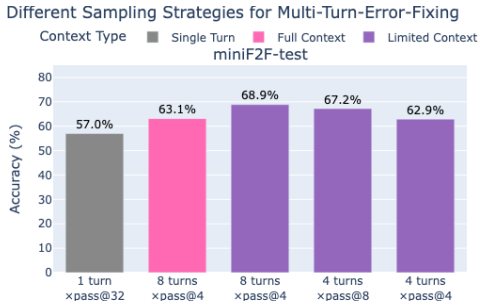
(a) Results from an 8-turn evaluation, with bars stacked by the turn of success (1 at bottom).



(b) Evaluating single-shot proof generation across k parallel attempts, without verifier feedback.



(c) Analysis of solution types for problems solved after the first turn.



(d) Different methods for spending a fixed 32-attempt budget with our best-performing model.

Figure 3: Main evaluation results on miniF2F test set. Our multi-turn and error-fixing training data significantly improves interactive theorem proving capabilities of the model **(a)**. Our final checkpoint substantially outperforms baselines in an 8-turn interactive evaluation **(b)**. This interactive performance even surpasses the strong pass@32 parallel sampling baseline **(c)**. The gain is driven by a learned ability to fix genuine errors, not just to complete partial proofs **(d)**. Combining 8-turn interaction with pass@4 further increases the accuracy. Note that MT-EF-Limited-Context is the same checkpoint as Multi-Turn-Error-Fixing with a different evaluation setting. While the full-context model has all of its previous attempts and verification feedback in its context for a problem, the limited-context model only retains its most recent attempt and verification feedback in addition to the original problem statement.

Main Results. Our results for multi-turn evaluation show a clear progression in capability through our phased training, as illustrated in Figure 3a. The Whole-Proof baseline derives little benefit from interacting with the verifier, not solving many problems beyond the first turn. The Multi-Turn-Correct checkpoint solves fewer problems in the first turn compared to Whole-Proof, since 65% of the multi-turn data trains it to first generate an incomplete proof skeleton before completing the proof in later turns. Consequently, it shows an improvement in solving problems in later turns. The key improvement comes from the Multi-Turn-Error-Fixing checkpoint, which learns to effectively use verifier feedback to solve substantially more problems from the third turn onward, increasing the accuracy to 50.4%. Surprisingly, limiting this model's context to only the most recent turn's feedback substantially boosts its ability to recover from errors, allowing it to solve even more problems after the third turn and increasing the accuracy to 58.6%. MT-EF-Limited-Context likely performs better because the most recent attempt provides all the necessary information for a fix, while the shorter context is more efficient and avoids potential long-context degradation.

To understand whether any improvements in the multi-turn setting result from genuine error feedback benefits or simply generating more samples, we also evaluate all checkpoints using pass@k in Figure 3b, with k independent, parallel single-shot proof attempts without verifier feedback. We observe that Whole-Proof performs better with 8 independent attempts (pass@8) than with 8 sequential turns. As expected, the pass@1 and pass@8 accuracy of the Multi-Turn checkpoints drop, compared to Whole-Proof, because of the training to generate proof skeletons for the first turn. Yet pass@32 performance remains surprisingly strong because a substantial minority (35%) of the training data consisted of single-turn examples, preserving the model's ability to sometimes generate complete

proofs in one shot. Remarkably, our final checkpoint in the limited-context setting with just 8 interactive turns surpasses the pass@32 performance of the other checkpoints, proving that learning to effectively use verifier feedback is more powerful than a four-fold increase in parallel attempts.

We also analyze *how* problems are solved after the first turn. Figure 3c breaks down these solutions into two types: completing a partial proof (filling a `sorry`) versus fixing a genuine compilation error. The results show a clear trend: While all multi-turn models are adept at completing proofs, the checkpoint trained on our error-recovery data shows a substantial increase in its ability to fix actual errors. We provide some examples of our model's multi-turn proof generations in Appendix D.

Another natural question is whether we can achieve additional advantages by combining parallel sampling with sequential feedback-driven attempts. We investigated hybrid strategies that use both the breadth of parallel sampling and the depth of sequential interaction, maintaining a fixed total sample budget of 32 generations per problem. We compared standard single-turn pass@32 against hybrid strategies including "8 turns×pass@4" (4 independent parallel runs, each with 8 sequential turns) and "4 turns×pass@8" (only in the limited-context setting, which consistently outperforms the full-context setting in other experiments). We also evaluated "4 turns×pass@4" as a sanity check to quantify what additional value is added by the last 4 turns, though this strategy uses only 16 total samples rather than the full 32-sample budget. The results are presented in Figure 3d.

We reach 68.9% accuracy with 8 turns×pass@4 in the limited-context setting, significantly outperforming single-turn pass@32, with 4 turns×pass@8 performing only slightly worse. Note that despite both strategies using 32 generations, the computational costs may differ substantially. Our training structure creates cost asymmetry: The model generates extensive CoT for planning the proof in the first turn, but shorter, targeted reasoning for subsequent turns. While input tokens increase with each turn, the generation cost is dominated by the number of output tokens, which is highest for the first turn. Highly parallel strategies (like pass@32 or 4 turns×pass@8) repeatedly incur this high initial cost, while more sequential approaches (like 8 turns×pass@4) require fewer expensive planning steps.

4 Conclusion

We present a systematic approach for training language models to effectively utilize verifier feedback, using a novel data pipeline that transforms static proofs into multi-turn, interactive dialogues. By fine-tuning an 8B model on our dataset of correct decomposition sequences and synthetic error-recovery scenarios, we demonstrate a significant boost in theorem-proving capabilities over the baseline. Our analysis confirms that these gains stem from the model learning to fix substantive errors through interaction, a strategy that proves more effective and computationally efficient than standard parallel sampling. Our work provides a strong foundation for future research. A natural next step is to explore reinforcement learning using the verifier to provide dense, intermediate rewards for proof-building. Systematic ablation studies will be useful to determine the optimal composition of our training data, particularly the proportion and types of error-correction examples.

References

[1] K. Baba, C. Liu, S. Kurita, and A. Sannai. Prover Agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025.

[2] I. Bouzenia, P. Devanbu, and M. Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

[3] C. Cao, L. Song, Z. Li, X. Le, X. Zhang, H. Xue, and F. Yang. Reviving DSP for advanced theorem proving in the era of reasoning models. *arXiv preprint arXiv:2506.11487*, 2025.

[4] L. Chen, J. Gu, L. Huang, W. Huang, Z. Jiang, A. Jie, X. Jin, X. Jin, C. Li, K. Ma, C. Ren, J. Shen, W. Shi, T. Sun, H. Sun, J. Wang, S. Wang, Z. Wang, C. Wei, S. Wei, Y. Wu, Y. Wu, Y. Xia, H. Xin, F. Yang, H. Ying, H. Yuan, Z. Yuan, T. Zhan, C. Zhang, Y. Zhang, G. Zhang, T. Zhao, J. Zhao, Y. Zhou, and T. H. Zhu. Seed-Prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025.

[5] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[6] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[7] K. Dong and T. Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*, 2025.

[8] E. First, M. N. Rabe, T. Ringer, and Y. Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023.

[9] F. Gloeckle, J. Limperg, G. Synnaeve, and A. Hayat. ABEL: Sample efficient online reinforcement learning for neural theorem proving. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024.

[10] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[11] A. Q. Jiang, S. Welleck, J. P. Zhou, W. Li, J. Liu, M. Jamnik, T. Lacroix, Y. Wu, and G. Lample. Draft, Sketch, and Prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022.

[12] N. Lambert, J. Morrison, V. Pyatkin, S. Huang, H. Ivison, F. Brahman, L. J. V. Miranda, A. Liu, N. Dziri, S. Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

[13] G. Lample, T. Lacroix, M.-A. Lachaux, A. Rodriguez, A. Hayat, T. Lavril, G. Ebner, and X. Martinet. HyperTree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

[14] H. Lin, Z. Sun, S. Welleck, and Y. Yang. Lean-STaR: Learning to interleave thinking and proving, 2025.

[15] Y. Lin, S. Tang, B. Lyu, J. Wu, H. Lin, K. Yang, J. Li, M. Xia, D. Chen, S. Arora, and C. Jin. Goedel-Prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, Feb 2025.

[16] Y. Lin, S. Tang, B. Lyu, Z. Yang, J. Chung, H. Zhao, L. Jiang, Y. Geng, J. Ge, J. Sun, and J. Wu. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, Aug 2025.

[17] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896*, 2023.

[18] A. Ospanov, F. Farnia, and R. Yousefzadeh. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning, 2025.

[19] Z. Peng, Y. Yao, K. Ma, S. Guo, Y. Li, Y. Zhang, C. Zhang, Y. Zhang, Z. Yu, L. Li, and M. Liu. CriticLean: Critic-guided reinforcement learning for mathematical formalization. *arXiv preprint arXiv:2507.06181*, Jul 2025.

[20] Z. Ren, Z. Shao, J. Song, H. Xin, H. Wang, W. Zhao, L. Zhang, Z. Fu, Q. Zhu, D. Yang, and Z. Wu. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, Apr 2025.

[21] S. Shang, R. Wan, Y. Peng, Y. Wu, X.-h. Chen, J. Yan, and X. Zhang. StepFun-Prover Preview: Let's think and verify step by step. *arXiv preprint arXiv:2507.20199*, 2025.

[22] H. Wang, M. Unsal, X. Lin, M. Baksys, J. Liu, M. Santos, F. Sung, M. Vinyes, Z. Ying, Z. Zhu, and J. Lu. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, Apr 2025.

[23] R. Wang, R. Pan, Y. Li, J. Zhang, Y. Jia, S. Diao, R. Pi, J. Hu, and T. Zhang. MA-LoT: Model-collaboration Lean-based long chain-of-thought reasoning enhances formal theorem proving, 2025.

[24] H. Xin, D. Guo, Z. Shao, Z. Ren, Q. Zhu, B. Liu, C. Ruan, W. Li, and X. Liang. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

[25] H. Xin, Z. Ren, J. Song, Z. Shao, W. Zhao, H. Wang, B. Liu, L. Zhang, X. Lu, Q. Du, et al. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024.

[26] R. Xin, C. Xi, J. Yang, F. Chen, H. Wu, X. Xiao, Y. Sun, S. Zheng, and K. Shen. BFS-Prover: Scalable best-first tree search for LLM-based automatic theorem proving. *arXiv preprint arXiv:2502.03438*, 2025.

[27] K. Zheng, J. Han, and S. Polu. MiniF2F: A cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, Aug 2021.

[28] Y. Zhou, J. Zhao, Y. Zhang, B. Wang, S. Wang, L. Chen, J. Wang, H. Chen, A. Jie, X. Zhang, et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025.

[29] M. Zimmer, X. Ji, R. Tutunov, A. Bordg, J. Wang, and H. B. Ammar. Bourbaki: Self-generated and goal-conditioned MDPs for theorem proving. *arXiv preprint arXiv:2507.02726*, 2025.

A Related Work

There is a rapidly growing body of research focused on applying Large Language Models (LLMs) to formal theorem proving. We situate our contribution in the context of several key paradigms: single-shot proof generation with refinement, tactic-level sequential generation, and data synthesis strategies.

Whole-proof generation. A prominent line of work treats formal proving as end-to-end sequence generation: The model emits a complete proof script in the language of, e.g., Lean or Isabelle, and the verifier checks the script once. This enables very easy scaling with parallel generation, but suffers from a less targeted search. Earlier attempts involved more direct completion of a proof, potentially with interleaved comments [24, 7, 15]. More recently, utilizing long chain of thought (CoT) and reinforcement learning with verifiable rewards (RLVR) [10, 12], foundation-style provers demonstrate that single-pass generation can already produce a large number of formally verified proofs [20, 16, 22]. Within this paradigm, authors often add *inline structure* to stabilize generation while keeping inference single-turn: producing a high-level *sketch* with `sorry` placeholders and then filling in a whole proof, roughly following the Draf-Sketch-Proof paradigm [11]. Although Lean-STaR is evaluated in stepwise regimes, its use of short thought snippets illustrates the same idea: lightweight, human-readable guidance that can be embedded in generated text without fine-grained state supervision [14]. The strength of whole-proof methods is simplicity and global consistency; their limitation is that the verifier provides no intermediate signal unless the system explicitly inserts pauses (e.g., `sorry`) or a separate repair stage.

Iterative refinement. A second line of work exploits verifier-in-the-loop training or prompting so that models improve a partially built proof across turns. At the tactic granularity, Lean-STaR interleaves a short natural-language 'thought' with a predicted tactic, then runs expert iteration to retain only successful trajectories [14]. Coarser-grained interactive systems use Lean feedback to refine lemmas or proof skeletons within the same session: StepFun-Prover performs a stepwise 'think-and-verify' cycle with real-time environment signals; Prover Agent coordinates an informal planner with a Lean-grounded executor that synthesizes on-the-fly lemmas from error messages; and Seed-Prover iteratively expands and repairs lemma-style sketches while retaining the ability to escalate search on hard instances [21, 1, 4]. Large-scale systems bring these ideas into SFT/RL pipelines: Goedel-Prover-v2 explicitly trains self-correction on scaffolded data so that the model proposes, verifies, and revises; Kimina-Prover aligns informal 'reasoning blocks' with formal code

and incorporates test-time search to discover and reuse lemmas while mitigating format collapse [16, 22]. A parallel thread augments single-pass generation with a *post-hoc* corrector that reads error messages and rewrites the *entire* proof: Baldur for Isabelle/HOL pioneered a generator+repairer architecture [8]; in Lean, multi-agent or rule-driven scaffolds such as *MA-LoT* (prover/critic with long CoT) and APOLLO (syntax refiner, 'sorrifier' to isolate subgoals, local solvers, recursive repair) follow the same regenerate-the-whole-proof template [23, 18]. Outside the formal math domain, closely related code systems use iterative refinement loops driven by compiler, runtime or test feedback, supporting the idea that intermediate verification signals are powerful supervision and control mechanisms for long-horizon reasoning [17, 2, 5].

Proof search and planning. A third line of work treats proving as *state-space search*, expanding a frontier of proof states under the checker's guidance rather than committing to one trajectory. HyperTree Proof Search (HTPS) brought AlphaZero-style planning to neural theorem proving [13]. Lean-centric planners integrate the verifier directly: DeepSeek-Prover-v1.5 couples proof assistant feedback with an MCTS-like planner; BFS-Prover scales best-first expansion with state–tactic policies trained using compiler-feedback annotations; and Bourbaki frames proving as self-generated, goal-conditioned MDPs solved by MCTS-style search [25, 26, 29]. Hybrid planner–writer systems ask a strong general LLM to draft a lemma plan or proof sketch and then rely on formal search to fill in small steps (e.g., DSP+, Delta-Prover) [28, 3]. Search generally boosts reliability on difficult theorems but is computationally expensive because many partial branches must be verified.

Like the whole-proof refinement models, we see value in error correction, but we train our model to perform this correction iteratively over multiple turns rather than as a one-shot repair. Similar to the tactic-level models, we opt for an interactive, step-by-step process, but our 'steps' are at the level of building proof skeletons and completing multiple lemmas each time, rather than individual tactics. Moreover, our data synthesis pipeline is specifically designed to generate the multi-turn, error-recovery dialogues that prove useful for teaching a model to truly interact with a proof assistant.

B Initial Evaluations

We performed initial evaluations on existing models to assess their capabilities and understand how different interaction patterns affect their accuracy on formal reasoning tasks. We evaluated Claude 3.7 (a large general-purpose LLM), Qwen3-8B (a small general-purpose LLM), and DeepSeekProver-v2-7B (a small LLM specifically trained to prove theorems in Lean) on miniF2F.

We evaluate these models under two interaction types:

1. Pass@8 (8 independent generations): Percentage of problems the model can solve when given up to 8 independent attempts at each problem. A proof is considered successful if any of the generations is correct.
2. Multi-turn interaction, 8 sequential turns: The models are allowed up to 8 attempts for each problem. They output their solution, receive any error messages from the Lean verifier, and continue based on the verifier's feedback until they exhaust all 8 attempts or solve the problem, while maintaining the full conversation history with all previous reasoning, attempts, and feedback in the context.

The purpose of this comparison is to see if having the previous attempts and verifier feedback can help a model any more than just independently retrying the proof without any context of its other tries. Figure 4 shows the accuracies with pass@8 (gray) and in the multi-turn setting. The colors represent the problems solved corresponding to different number of attempts (e.g., problems solved in the first turn are at the bottom in yellow, and problems solved after 8 attempts are at the top in blue).

From the results, we can observe that

- Claude 3.7 has a low accuracy on the benchmark overall, but the accuracy with 8 sequential attempts (with previous attempts and error feedback in context) is considerably higher compared to 8 independent generations. This makes sense since Lean data is likely a very small proportion of a general-purpose LLM's training data. However, these models excel at following instructions and at in-context learning, enabling them to use error information from Lean to some extent.

- In contrast, specialized models like DeepSeekProver-v2-7B demonstrate impressive performance on their focused domain. DSPv2-7B solves a substantial proportion of problems on the first turn. However, for problems it does not solve initially, the model struggles to utilize error feedback effectively, showing minimal improvement in subsequent attempts. Our more detailed analysis revealed that when DSPv2 receives error feedback after an initial attempt, it often fails to produce valid Lean code and instead generates incoherent text. This is likely because it was trained primarily on single-turn data, making multi-turn interactions out-of-distribution for the model.
- Qwen3-8B has minimal Lean capabilities. Pass@8 performance is marginally better than 8 sequential attempts.

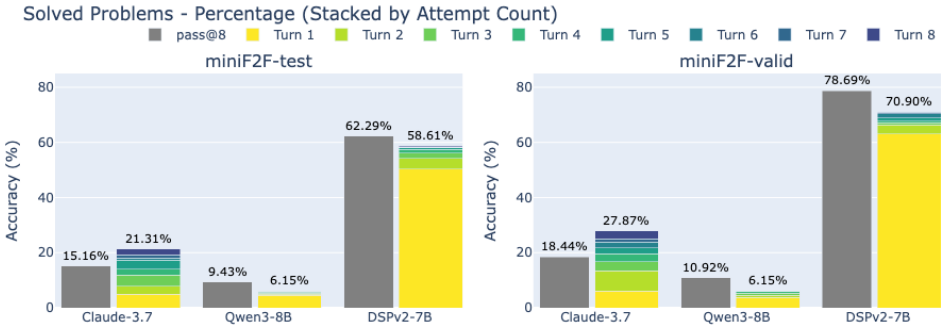


Figure 4: Comparing the performance of Claude3.7, Qwen3-8B, and DeepSeekProver-v2-7B on miniF2F test and validation sets, across pass@8 and 8 interactive turns

We also experimented with various prompting strategies for these models, including explicitly instructing them to first generate a proof skeleton with intermediate lemmas and sending it to the verifier as its first attempt before filling in the details to complete the proof. However, these different prompting approaches did not significantly affect performance.

These initial findings suggest that training a model on Lean data with appropriate interaction patterns could lead to meaningful improvements. Therefore, we focus on generating such data and evaluating its effectiveness for enhancing theorem-proving capabilities. To that end, we fine-tuned the underperforming Qwen3-8B model using our novel data generation pipeline.

C Training and Evaluation Setup

We provide the details of the training and evaluation hyperparameters in this section. For all supervised fine-tuning, we used the Adam optimizer with cosine learning rate scheduler, weight decay 0.01, gradient clipping with maximum gradient norm 1, and batch size 512. The Whole-Proof checkpoint was trained for one epoch with learning rate 5×10^{-6}, Multi-Turn-Correct was trained for 2 epochs with learning rate 2×10^{-5}, and Multi-Turn-Error-Fixing was trained for 2 epochs with learning rate 5×10^{-6}.

During evaluation, we set the maximum context length to 131,072 to accommodate the longer context required in the multi-turn setting. We used the recommended sampling parameters for Qwen3-8B in thinking mode (the underlying base model for all our checkpoints), i.e. Temperature=0.6, TopP=0.95, TopK=20, and MinP=0. We used the version of miniF2F available at `https://github.com/deepseek-ai/DeepSeek-Prover-V1.5/blob/main/datasets/minif2f.jsonl`.

Our data filtering was performed using Lean v4.21.0, whereas the evaluation server we used runs Lean v4.9.0. Due to some syntax differences between these versions, it is possible that our models' reported accuracies are slightly lower than their true capabilities. However, since all checkpoints were evaluated under these identical conditions and our source proofs were generated using Goedel-

Prover-v2-32B, which also used Lean v4.9.0, we do not expect this to have a substantial impact on the results.

Figure 5: Evaluation results on the miniF2F validation set. *Top Left:* Main results from the 8-turn interactive evaluation, showing significant improvement from our phased training. *Top Right:* Baseline pass@k performance for single-turn, parallel generation. *Bottom Left:* Breakdown of multi-turn successes, showing an increase in genuine error correction for our final model. *Bottom Right:* Comparison of hybrid sampling strategies under a fixed 32-attempt budget, showing that multi-turn methods outperform the parallel baseline.

We also present the evaluation results on the validation set of the miniF2F benchmark in Figure 5. The experiment settings are identical to the ones for the test set, described earlier. We do not use the validation set for any part of the training.

D Multi-Turn Proof Example

We present an example of a multi-turn proof generation by our model, Multi-Turn-Error-Fixing. In this example, the model first generates a proof skeleton, makes an error when completing the proof, and then fixes the error with the help of the error information received from the verifier.

User Prompt

Prove the following theorem in Lean4:
/– Find y: $\sqrt{19+3y} = 7$. Show that it is 10.-/

```
theorem mathd_algebra_263 (y : ) (h0 : 0  19 + 3 * y) (h1 : Real.sqrt
↪  (19 + 3 * y) = 7) :
    y = 10 := by
```

Model

<think>**Problem Analysis:** We need to solve for ‘y‘ in the equation ‘(19 + 3y) = 7‘ under the constraint that ‘19 + 3y 0‘.

Key Observations: 1. The square root function ‘x‘ is defined only when ‘x 0‘, and it returns the non-negative root. The hypothesis ‘0 19 + 3y‘ ensures that the square root is real. 2. The square of both sides of the equation ‘(19 + 3y) = 7‘ will eliminate the square root, but we must be careful because squaring can introduce extraneous solutions. However, since the square root is non-negative and ‘7‘ is positive, the solution we obtain will be valid if it satisfies the original equation and the constraint. 3. The square of ‘(19 + 3y)‘ is ‘19 + 3y‘, so squaring both sides gives ‘19 + 3y = 49‘.
Proof Steps: 1. Square both sides of the equation ‘(19 + 3y) = 7‘ to eliminate the square root. This gives ‘19 + 3y = 49‘. 2. Solve the resulting linear equation ‘19 + 3y = 49‘ for ‘y‘: - Subtract ‘19‘ from both sides: ‘3y = 30‘. - Divide both sides by ‘3‘: ‘y = 10‘. 3. Verify that ‘y = 10‘ satisfies the original constraint ‘19 + 3y 0‘: - ‘19 + 3 * 10 = 49 0‘, which is true. 4. Check that ‘y = 10‘ also satisfies the original equation ‘(19 + 3y) = 7‘: - ‘(49) = 7‘, which is correct.
However, in Lean, the hypothesis ‘h1 : Real.sqrt (19 + 3 * y) = 7‘ already implies that ‘19 + 3 * y 0‘ because the square root is only defined for non-negative numbers. The additional hypothesis ‘h0 : 0 19 + 3 * y‘ is redundant but does not hurt.
But we can directly use the property of square roots in Lean: if ‘Real.sqrt x = y‘, then ‘x = y2‘ provided that ‘y 0‘ (which it is in this case, since ‘7 0‘).
Lean-Specific Considerations: 1. Lean’s ‘Real.sqrt_eq_iff_sq_eq‘ lemma states that for ‘x 0‘ and ‘y 0‘, ‘Real.sqrt x = y‘ if and only if ‘x = y2‘. - Here, ‘x = 19 + 3 * y‘ and ‘y = 7‘, so we can use this lemma to deduce ‘19 + 3 * y = 49‘. - The condition ‘y 0‘ is satisfied because ‘7 0‘. 2. We can then solve ‘19 + 3 * y = 49‘ to get ‘y = 10‘.
Formal Proof Sketch: 1. Use ‘Real.sqrt_eq_iff_sq_eq‘ to deduce ‘19 + 3 * y = 49‘ from ‘h1‘, since ‘7 0‘ and ‘19 + 3 * y 0‘ (by ‘h0‘). 2. Simplify ‘19 + 3 * y = 49‘ to ‘3 * y = 30‘ and then to ‘y = 10‘.
Step-by-Step Abstract Plan
1. **Square Both Sides**: - Since ‘Real.sqrt (19 + 3 * y) = 7‘ and ‘7 0‘, we have ‘19 + 3 * y = 72 = 49‘.
2. **Solve for ‘y‘**: - Subtract ‘19‘ from both sides: ‘3 * y = 30‘. - Divide both sides by ‘3‘: ‘y = 10‘.
3. **Verification (Optional in Lean)**: - Check that ‘19 + 3 * 10 = 49 0‘ (already given by ‘h0‘). - Check that ‘Real.sqrt 49 = 7‘ (which is true, but Lean’s lemma already handles this).
Lean 4 Proof with ‘have‘ Statements

```
theorem mathd_algebra_263 (y : ) (h0 : 0  19 + 3 * y) (h1 : Real.sqrt
↪  (19 + 3 * y) = 7) :
    y = 10 := by
  have h2 : 19 + 3 * y = 49 := by sorry
  have h3 : 3 * y = 30 := by sorry
  have h4 : y = 10 := by sorry
  sorry
```

Explanation of ‘have‘ Statements: 1. ‘h2‘: We deduce ‘19 + 3 * y = 49‘ by squaring both sides of ‘h1‘ and using the property of square roots. In Lean, this can be done using ‘Real.sqrt_eq_iff_sq_eq‘. 2. ‘h3‘: Subtract ‘19‘ from both sides of ‘h2‘ to get ‘3 * y = 30‘. 3. ‘h4‘: Divide both sides of ‘h3‘ by ‘3‘ to get ‘y = 10‘.
Each step follows directly from the previous one using basic arithmetic operations, and the final result is ‘y = 10‘.</think>The problem requires proving that y = 10 given the equation (19 + 3y) = 7 and the domain constraint 0 19 + 3y. My initial strategy is to eliminate the square root by squaring both sides, then solve the resulting linear equation.
First, I recognize that squaring both sides of the equation will give 19 + 3y = 72 = 49. This forms the first subgoal h2: 19 + 3y = 49. To establish this, I need to use the property that if x = y with y 0, then x = y2. This requires verifying that 7 is non-negative (which is trivial) and that 19 + 3y is non-negative (already given by h0).
Next, solving 19 + 3y = 49 leads to 3y = 30 through simple subtraction. This becomes subgoal h3: 3y = 30. Finally, dividing both sides by 3 gives y = 10 (subgoal h4).
I structure the proof using ‘have‘ statements to isolate these steps because each transformation depends on the previous one. The final ‘apply h4‘ completes the proof once all intermediate

steps are verified. However, I need to fill in the details for each subgoal, particularly proving h2 using the square root property.

```
import Mathlib

theorem mathd_algebra_263 (y : ) (h0 : 0  19 + 3 * y) (h1 : Real.sqrt
↪  (19 + 3 * y) = 7) : y = 10 := by
  have h2 : 19 + 3 * y = 49 := by sorry
  have h3 : 3 * y = 30 := by sorry
  have h4 : y = 10 := by sorry
  apply h4
```

Lean Verifier Response

The proof outline with ‘sorry‘ placeholders is correct. The proof is incomplete. 3 goals, marked with ‘sorry‘ placeholders, remain unproven. Please fix the issues and provide an updated proof.

Model

<think>The verification feedback shows three remaining goals. I’ll address them in order of dependency:
1. For h2: 19 + 3y = 49 - Use Real.sqrt_sq_eq_iff to show (19+3y)2 = 19+3y - Apply h1 (which states (19+3y) = 7) to get 72 = 49 - Use linarith to connect these equalities
2. For h3: 3y = 30 - This follows directly from h2 (19 + 3y = 49) by subtracting 19 - linarith can handle this simple arithmetic transformation
3. For h4: y = 10 - From 3y = 30, divide both sides by 3 - Again, linarith can solve this
I notice that the square root property requires proving 0 19 + 3y (already given by h0) and 0 7 (trivial). The ‘rw [h1]‘ tactic will substitute the square root expression with 7, and ‘norm_num‘ will simplify 72 to 49. The final ‘linarith‘ calls will handle the arithmetic transformations. This approach should work since all operations are reversible and preserve equality.</think>

```
import Mathlib

theorem mathd_algebra_263 (y : ) (h0 : 0  19 + 3 * y) (h1 : Real.sqrt
↪  (19 + 3 * y) = 7) : y = 10 :=
  by
  have h2 : 19 + 3 * y = 49 := by
    have h21 : Real.sqrt (19 + 3 * y) ^ 2 = (7 : ) ^ 2 := by rw [h1]
    have h22 : (Real.sqrt (19 + 3 * y)) ^ 2 = 19 + 3 * y := by rw
    ↪  [Real.sqrt_sq_eq_iff] <;> norm_num <;> linarith
    have h23 : (7 : ) ^ 2 = 49 := by norm_num
    linarith
  have h3 : 3 * y = 30 := by linarith
  have h4 : y = 10 := by linarith
  apply h4
```

Lean Verifier Response

The proof is not correct.
Error 1: Location: Line 13, Column 68
Corresponding Code:

```
theorem mathd_algebra_263 (y : ) (h0 : 0  19 + 3 * y) (h1 : Real.sqrt
↪  (19 + 3 * y) = 7) :
```

```
    y = 10 := by
  have h2 : 19 + 3 * y = 49 := by
    have h21 : Real.sqrt (19 + 3 * y) ^ 2 = (7 : ) ^ 2 := by rw [h1]
    have h22 : (Real.sqrt (19 + 3 * y)) ^ 2 = 19 + 3 * y := by rw
    ↪  [<error>Real.sqrt_sq_eq_iff</error>] <;> norm_num <;>
    ↪  linarith
    have h23 : (7 : ) ^ 2 = 49 := by norm_num
```

Error Message: unknown constant 'Real.sqrt_sq_eq_iff'
Error 2: Location: Line 13, Column 68
Corresponding Code:

```
theorem mathd_algebra_263 (y : ) (h0 : 0  19 + 3 * y) (h1 : Real.sqrt
↪  (19 + 3 * y) = 7) :
    y = 10 := by
  have h2 : 19 + 3 * y = 49 := by
    have h21 : Real.sqrt (19 + 3 * y) ^ 2 = (7 : ) ^ 2 := by rw [h1]
    have h22 : (Real.sqrt (19 + 3 * y)) ^ 2 = 19 + 3 * y := by rw
    ↪  [<error>Real.sqrt_sq_eq_iff</error>] <;> norm_num <;>
    ↪  linarith
    have h23 : (7 : ) ^ 2 = 49 := by norm_num
```

Error Message: tactic 'rewrite' failed, equality or iff proof expected ?m.1339 y : h0 : 0 19 + 3 * y h1 : (19 + 3 * y) = 7 h21 : (19 + 3 * y) ^2 = 7 ^2 (19 + 3 * y) ^2 = 19 + 3 * y
Please fix the issues and provide an updated proof.

Model

<think>The error messages indicate that Real.sqrt_sq_eq_iff doesn't exist. I need to find the correct theorem for this identity.
Looking at Mathlib's documentation, I recall that Real.sq_sqrt (h0 : 0 x) : x ^2 = (x) ^2 is the proper theorem. This requires the non-negativity hypothesis h0, which we have.
I'll modify the proof: 1. Use Real.sq_sqrt h0 to get (19+3y)2 = ((19+3y))2 2. Apply h1 to substitute (19+3y) with 7 3. Use norm_num to simplify 72 to 49 4. Combine these with linarith to derive 19+3y = 49
This approach should work because: - Real.sq_sqrt directly gives the identity we need - h1 provides the substitution for the square root - norm_num handles numerical simplifications - linarith can then solve the resulting equations
I'll replace the invalid rewrite with Real.sq_sqrt and adjust the proof accordingly.</think>

```
import Mathlib

theorem mathd_algebra_263 (y : ) (h0 : 0  19 + 3 * y) (h1 : Real.sqrt
↪  (19 + 3 * y) = 7) : y = 10 :=
  by
  have h2 : 19 + 3 * y = 49 := by
    have h21 : Real.sqrt (19 + 3 * y) ^ 2 = (7 : ) ^ 2 := by rw [h1]
    have h22 : Real.sqrt (19 + 3 * y) ^ 2 = 19 + 3 * y := by rw
    ↪  [Real.sq_sqrt] <;> norm_num <;> linarith
    have h23 : (7 : ) ^ 2 = 49 := by norm_num
    linarith
  have h3 : 3 * y = 30 := by linarith
  have h4 : y = 10 := by linarith
  apply h4
```