LeanAssist: Open-Source & Model-Agnostic LLM Proof Assistance

Rey Riordan, Man Cao
Rutgers University - New Brunswick
01:640:495:03 SEL TOPICS IN MATH
December 10, 2025

Abstract

We gonna do this at the very end

Introduction

Contemporary large language models (LLMs) such as GPT-4, Claude, Gemini, and more are trained on massive datasets which allow them to generalize decently well across a diverse set of tasks. They have been rapidly innovated, demonstrating impressive natural language understanding and reasoning. Alongside this surge in capabilities, code assistants leveraging them such as Claude Code, GitHub Copilot, and Codex CLI have become quite popular tools in software development. Thus, when prompted with making a tool that is useful for mathematics research, we naturally gravitated towards an LLM assistant that could help with theorem proving. In order to properly ground our work, we first researched the format, methods, and other available options that are relevant for creating such a tool.

For the format, we chose to focus on Lean which is a popular programming language and interactive theorem prover widely used in formal mathematics. Lean is powerful because it contains dependent type theory, a growing mathematical library (mathlib), and a structured tactic framework for deconstructing proofs into reasoning steps. A tactic is usually a small step that changes the current goal into something simpler (e.g., introducing variables, rewriting with lemmas, or simplifying an expression). Typically, a series of tactics is used to deduce the ultimate goal. This intersection of coding, mathematics, and popularity makes it the perfect target for our proof assistant research. Although traditional coding assistants are still technically compatible with Lean, we aimed to make an assistant that is specialized in suggesting tactics.

Background

For modern LLMs, fine-tuning has become a popular strategy for adapting models to specialized domains or problems like ours. This extension of training on a specific dataset relevant to the task enables the model to internalize domain-specific patterns for a better generalization. However, one of the biggest challenges when fine-tuning these larger models is the high number of trainable parameters which increases computation and memory costs (Li et al., 2024). Hence, a method known as low-rank adaption (LoRA) can improve parameter efficiency and costs by inserting low-rank update matrices into a pretrained model. The LoRA method and its variations (e.g. QLoRA) have become the standard in the space of LLM fine-tuning due to these benefits often coming with only marginal performance tradeoffs. The only true downside is that fine-tuning is generally only widely available on open-source models, since the actual weights have to be changed. Thus, finding open-source models that are good at math becomes essential.

Much work is being done in the field of applying LLMs to mathematics and while progress is rapidly being made, the domain remains challenging when it comes to cutting edge math research (Ahn et al., 2024). Companies such as OpenAI, Google, and Harmonic have recently succeeded in achieving IMO gold medal performance with their respective models, evidencing exceptional mathematical reasoning capabilities. OpenAI and Google's results were obtained with general-purpose LLMs, making them especially impressive. Other more specialized models such as Google DeepMind's AlphaGeometry are also notable innovations. However, in regards to theorem proving assistance, nothing like an actual interactive "copilot" like what exists in the programming domain appears to be available. In addition, most state-of-the-art models remain closed-source, with the only gold medal level open-source LLM being DeepSeek-v3.2-Speciale, a specialized version of the newest offering from DeepSeek.

In terms of open-source proof assistants, options such as Isabelle and HOL Light are robust automated theorem provers but are not AI-based. They depend on large libraries like the Archive of Formal Proofs to maintain long-term stability and do not generalize as well for new theorems. Other notable open-source systems like Agda, Prover9/Mace4, and ACL2 all provide automatic reasoning environments adapted to specific domains of logic. Overall, a gap remains in AI-based proof assistance which is open-source and has state-of-the-art mathematical reasoning. In addition to these limitations, it lacks features similar to the code assistants that have gained so much popularity and use in recent years.

All of this background research was synthesized to help us make an informed decision on the direction of our work. The base model of our proof assistant would need to be open source and state-of-the-art in mathematics performance, using something like DeepSeek-v3.2. In addition, such a model would ideally be LoRA fine-tuned with data specialized for the task of suggesting Lean tactics. Lastly, the user interface should be similar to code assistants. One additional important consideration is that as new and better models are released, they should be able to be easily swapped in to the assistant. Therefore, our assistant must be model agnostic, in that it should work with any model and it should be easy for the user to add their own new or custom models.

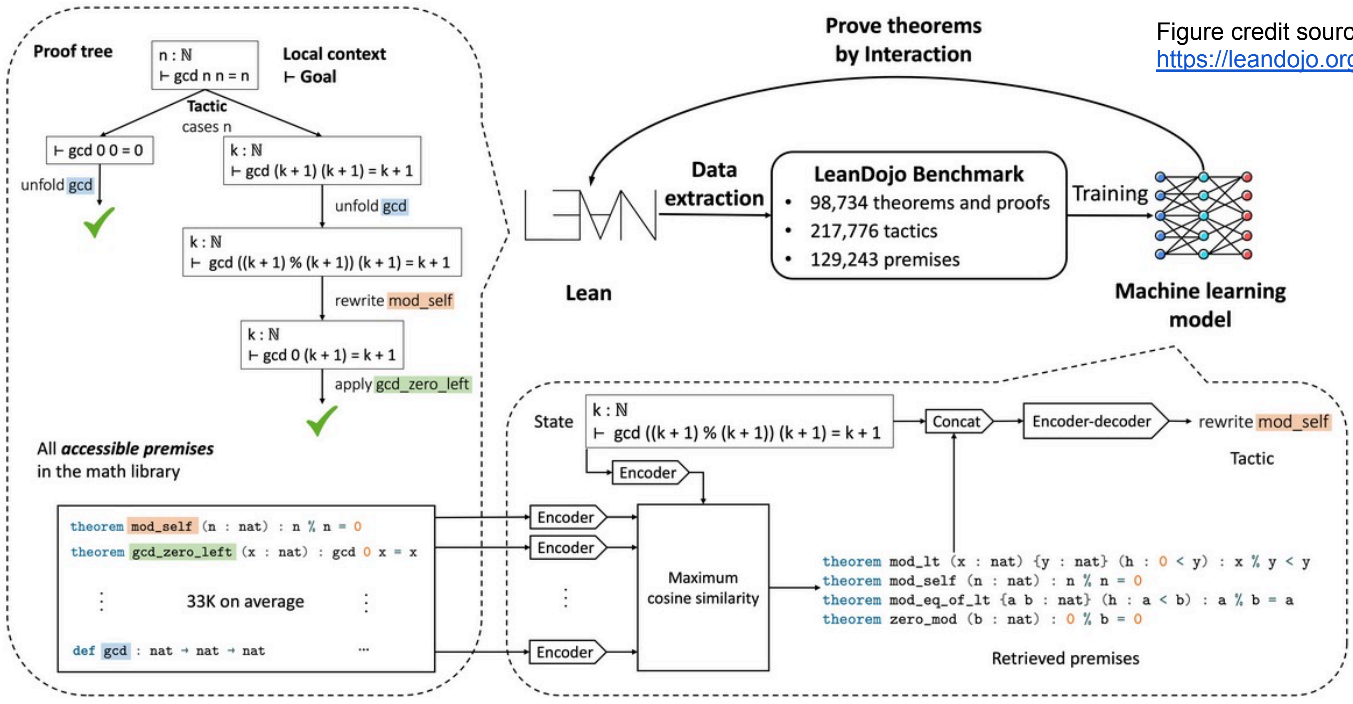
Related Work

Our work is best framed as a direct extension of the paper: “LeanDojo: Theorem Proving with Retrieval-Augmented Language Models” (Yang et al., 2023). This research is not only the most relevant work but it also provides most of the key ingredients needed for our proof assistant: a way for LLMs to easily interact with Lean, comprehensive datasets for fine-tuning and benchmarking models, and a LeanCopilot that can already suggest tactics using specific models.

Their “LeanDojo” is a framework that bridges the gap between text-based LLMs and the specialized environment of the Lean prover. This is done by converting Lean into an API-like interface in which AI agents “can programmatically observe proof states, submit tactics, and receive feedback”. This makes it very simple for LLMs to interact with the Lean environment and attempt proof steps - they just have to suggest tactics and try running them to see how the proof state changes. Another important aspect of this work is that they systematically extracted rich datasets from existing Lean repositories, using specialized techniques to obtain the exact proof states, tactics, premise annotations, etc. This resulted in two large benchmarks, one for Lean 3 and one for Lean 4. Each has hundreds of thousands of fully proven theorem examples with all steps, and are already split into training, validation, and test sets for machine learning use.

Lastly, with an interactive environment and great dataset in place, the team used LeanDojo and the generated benchmarks as a dataset to train their own model ReProver to assist in Lean theorem proving. It was a retrieval-augmented LLM, which means they used retrieval augmented generation (RAG) techniques to pull the most relevant tactics from the library to better inform the base LLM's response. The following is a comprehensive diagram detailing the full ecosystem.

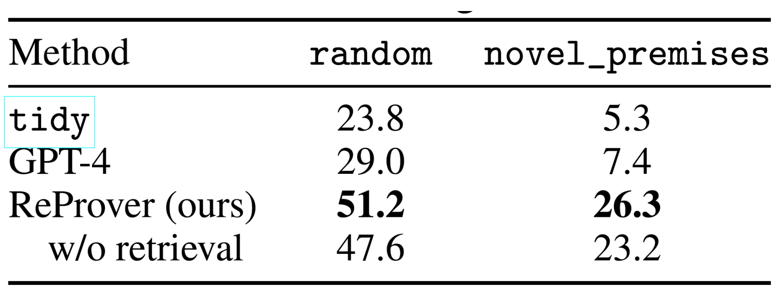
Figure credit source https://leandojo.org



```
theorem mod_self (n : nat) : n % n = 0
theorem gcd_zero_left (x : nat) : gcd 0 x = x
def gcd : nat → nat → nat
```

```
theorem mod_lt (x : nat) {y : nat} (h : 0 < y) : x % y < y
theorem mod_self (n : nat) : n % n = 0
theorem mod_eq_of_lt {a b : nat} (h : a < b) : a % b = a
theorem zero_mod (b : nat) : 0 % b = 0
```

Subsequently, the LeanDojo team benchmarked ReProver in comparison with baselines GPT-4 and tidyllm (Lean's built-in proof automation model) using a proof search algorithm that generates multiple tactics at each step, far outperforming both. Note that they used a proof search algorithm that generates multiple tactics at each step for all models, that the pass@1 refers to the probability that the model succeeds on the 1st attempt, and that they provided two different splits of the dataset where “novel_premises” is much more difficult.



Method	`random`	`novel_premises`
`tidy`	23.8	5.3
GPT-4	29.0	7.4
ReProver (ours)	**51.2**	**26.3**
w/o retrieval	47.6	23.2

Lastly, the LeanDojo team designed a “LeanCopilot” that uses their Reprover model as a base to create a theorem proving assistant that can respond to commands such as “suggest_tactics” and “search_proof” directly in VS Code and other IDEs, resulting in an AI assisted workflow similar to what is available for coding assistants.

Motivation

It is clear that Yang et al. did amazing work with LeanDojo, but there are a few limitations that prevent their assistant from modern use. Firstly, there is no easy way to benchmark the various new models that are constantly being released on their datasets. This would be helpful to measure model performance in the realm of theorem proving, but the main blocker here is that their proof search algorithm is likely buried somewhere in the repo written in Lean, making it difficult to access (people usually use Python to interact with LLMs). Secondly, the format of their dataset is not suitable for fine-tuning. Fine-tuning is the standard way to create specialized versions of models that could serve as the best possible base models for LeanCopilot, but there are specific formatting requirements. Lastly, it is difficult to add new models to their LeanCopilot. They provide a rough roadmap for adding external models, but they use different runner frameworks for each provider (OpenAI, Google, Anthropic, etc) and a lot of code has to be written from scratch to make more obscure or open-source models compatible.

We set out to remediate all of these issues and consequently achieve our main goal of creating a fully open-source and model-agnostic Lean proof assistant. First, we create a framework that allows for easy benchmarking of absolutely any model (proprietary, open-source, custom fine-tuned) on either of the LeanDojo dataset splits. To do this we include our own new proof search algorithm in Python. Second, we create a fine-tuning dataset generator that uses the LeanDojo datasets as a source to create properly formatted datasets of any size. Finally, just like with benchmarking, we make it possible to port any model of the user’s choice into LeanCopilot and use it for proof assistance. The ultimate goal would be to use all of this to benchmark and fine-tune a frontier model that can possibly outperform LeanDojo’s ReProver.

Methods

In order to enable the use of any model, we utilize OpenRouter and Fireworks. OpenRouter provides a unified API interface for almost all LLM providers, making switching between different models from different models as simple as just changing the model name in the request. However, OpenRouter itself is just a router to other inference providers, so it’s not possible to get one’s own custom fine-tuned models deployed there. This is where Fireworks comes in - it’s an inference engine that makes most open source LLMs available for on-demand deployment and fine-tuning, while also making the process of deploying/tuning models extremely intuitive and easy. Combining these two endpoints gives coverage of pretty much every single LLM that’s physically available.

Benchmarking

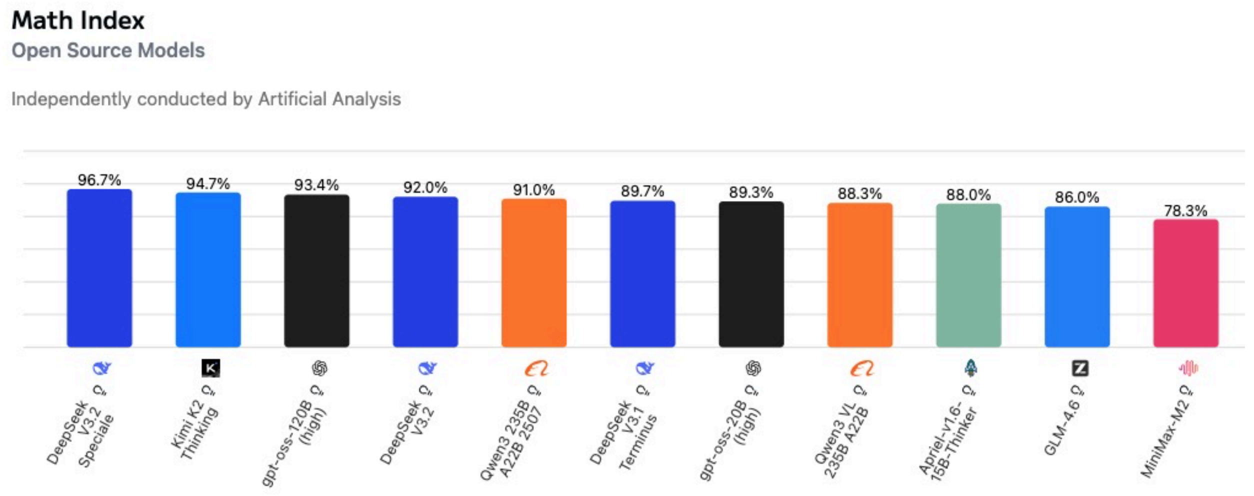
First, we created a general API client that is compatible with both OpenRouter and Fireworks. This abstracts away the specifics of calling each API, and makes it so that we can simply specify a provider and then call the generation method. Next, we implemented our custom proof search algorithm. The algorithm works as follows:

1. We configure a model provider, model name, and number of tactics to generate at each step.
2. We create a queue to store proof states, and add the original proof state.
3. We pop the queue to get a proof state, then generate N tactics for the next step by calling the LLM with 1.0 temperature in order to get more diverse outputs.
4. We run each tactic generated through LeanDojo to determine whether it leads to a valid next state, we push each valid new state into the queue.
5. We repeat until either we successfully prove the theorem, the search is exhausted (empty queue), or we hit the pre-specified search time limit.

The prompt used to generate each tactic suggestion is the following: “Help me prove a theorem in Lean 4. Here is my current proof state:\n{state}\nBased on this state, suggest the next tactic I should use in Lean 4 code. Only output one tactic step in lean code and nothing else.”

We then implemented an evaluator that iterates through all specified theorem examples and tries to prove each using this proof search. In order to make the evaluation process more efficient, we also implemented the ability to run it with a configured number of concurrent workers. Each theorem proof attempt result is written to a results.json file in a new directory as soon as it is completed, and at the end a summary.json is also created that provides statistics such as the accuracy (successfully proved / total), average proof length, and average search time.

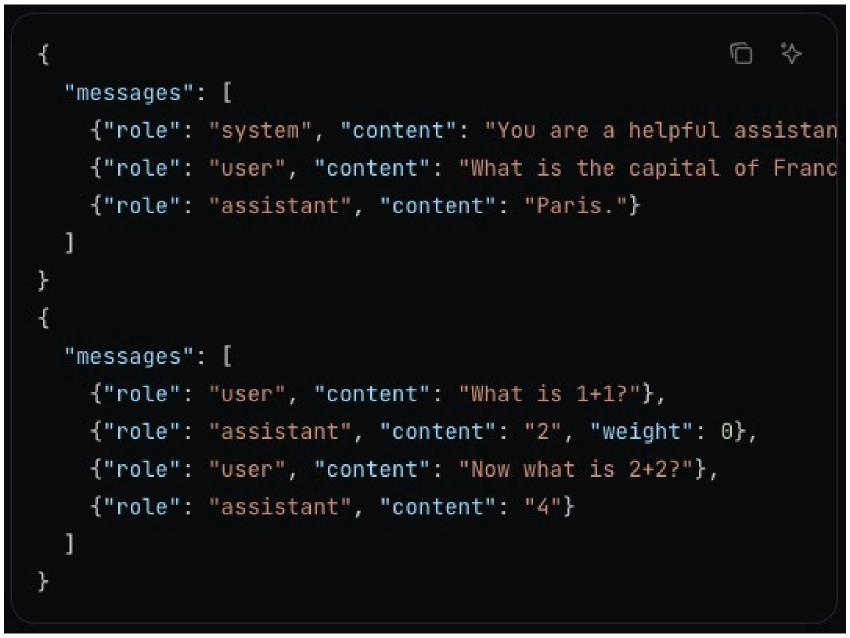
In order to benchmark the model with the most potential, we consulted one of the most reputable model performance score aggregators Artificial Analysis, and looked at the open-source models that performed the highest in their math index:



We did not initially consider proprietary models even though they are obviously available through OpenRouter because they cannot be fine-tuned. With the goal being to beat ReProver performance on the benchmark, it makes more sense to choose a model that can be fine-tuned on the LeanDojo dataset to create a theorem proving specialized LoRA, as models that are fine-tuned for a task will perform better at that task. Ultimately we chose to benchmark the newly released DeepSeek V3.2, since the top 3 did not have sufficient throughput (speed) to evaluate in a reasonable amount of time.

Fine-tuning

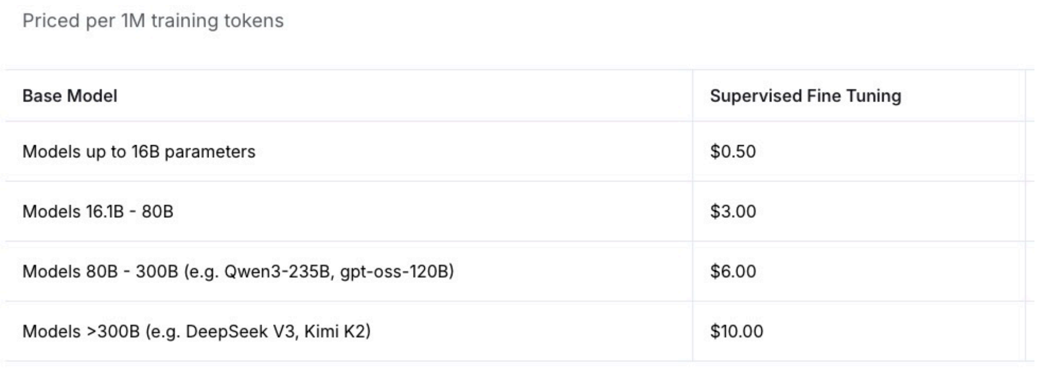
In order to do supervised fine-tuning on Fireworks and beyond, the data must be formatted how modern LLMs are used to taking in data - the “conversation history” format popularized by OpenAI. As seen in this example from the Fireworks documentation, each “conversation” must have a list of “messages”, each with a “role” (“system”, “user”, or “assistant”) and text “content”.



```
{
  "messages": [
    {"role": "system", "content": "You are a helpful assistan
    {"role": "user", "content": "What is the capital of Franc
    {"role": "assistant", "content": "Paris."}
  ]
}
{
  "messages": [
    {"role": "user", "content": "What is 1+1?"},
    {"role": "assistant", "content": "2", "weight": 0},
    {"role": "user", "content": "Now what is 2+2?"},
    {"role": "assistant", "content": "4"}
  ]
}
```

We implemented a short script that, based on some configurable parameters, automatically generates a dataset in this format with the LeanDojo datasets as the source. When running the script, the user is able to specify their dataset name, whether they want to take from “random” or “novel_premises” split, the number of theorem examples to take from the train dataset (max of 100k+), and the number of theorems to take from the validation dataset (max of 2k, used to evaluate loss/accuracy during fine-tuning). For each entry in the training datasets, we extract “pair” conversations, where the “user” message mirrors the tactic suggestion request prompt from benchmarking code, and the “assistant” message is just the correct next tactic step in the proof. Each entry can have multiple tactic steps - in these cases we simply extract multiple pairs corresponding to each tactic.

Once the new datasets have been created in the data/ folder and downloaded locally, a fine-tuning job can be created directly through the Fireworks web UI. The user can simply go to “Fine-tune a Model”, select the “Supervised” option, select the based model fine-tune, upload the train and val datasets, and optionally configure some advanced parameters before hitting the “Create Job” button. Once the job is created, the progress will be available to see in real-time through a dashboard, and upon completion the model can be deployed on-demand and used on the API through a special personal model name. For reference, this is pricing for fine-tuning on Fireworks:

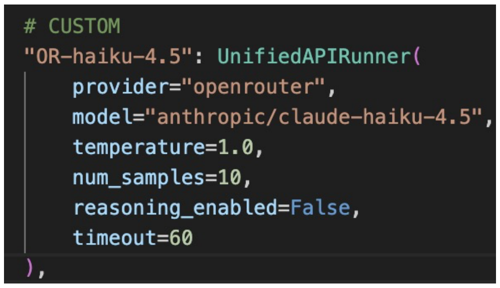


Priced per 1M training tokens

Base Model	Supervised Fine Tuning
Models up to 16B parameters	\$0.50
Models 16.1B - 80B	\$3.00
Models 80B - 300B (e.g. Qwen3-235B, gpt-oss-120B)	\$6.00
Models >300B (e.g. DeepSeek V3, Kimi K2)	\$10.00

Assistant

In a similar manner to how we handled the API calls for benchmarking, we implemented a new unified_api_runner.py in the *python/external_models/* directory of the LeanCopilot repo. After additionally adding proper parsing logic (copied over from benchmarking code) to *external_parser.py*, the assistant was made fully compatible with both OpenRouter and Fireworks. For a user to add a new model, all they have to do is go to *server.py* and add a new UnifiedAPIRunner instance; here is an example of adding Claude 4.5 Haiku through OpenRouter:



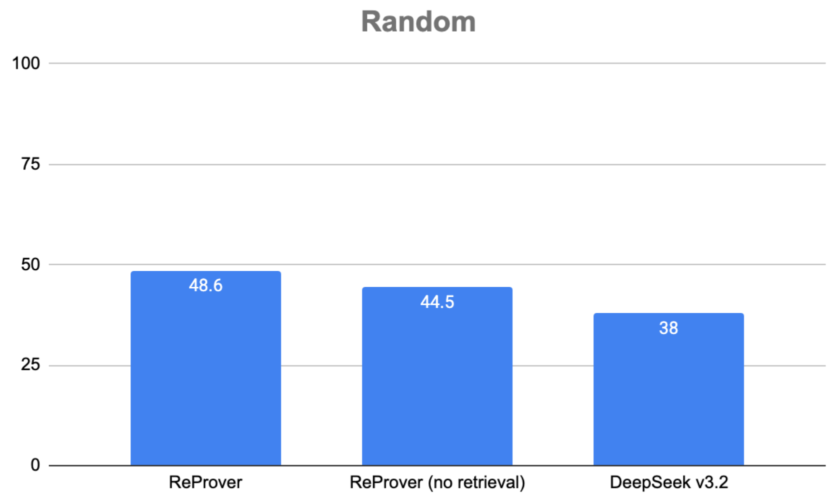
```
# CUSTOM
"OR-haiku-4.5": UnifiedAPIRunner(
    provider="openrouter",
    model="anthropic/claude-haiku-4.5",
    temperature=1.0,
    num_samples=10,
    reasoning_enabled=False,
    timeout=60
),
```

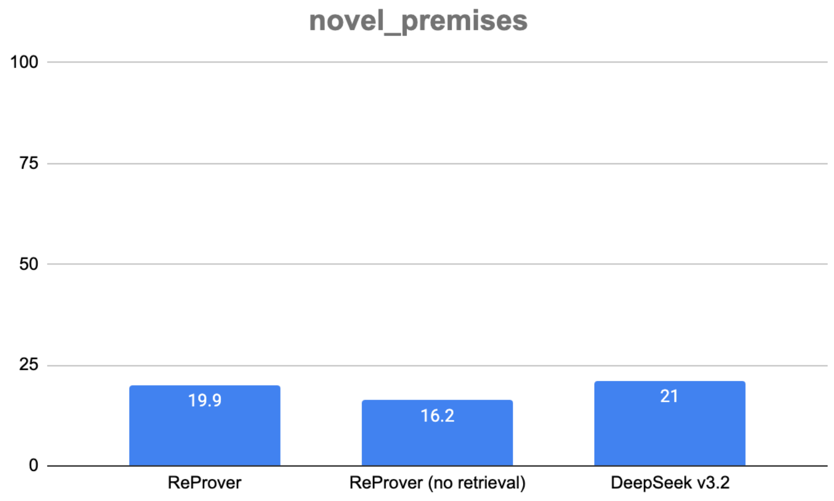
Once configured in this way, the user can just run LeanCopilot on a local server and use it with their Lean files.

Results

Benchmarking

Due to budget and time constraints, we ran the benchmark on DeepSeek V3.2 on only the first 100 entries of each test dataset, with 10 tactic samples and 4 concurrent workers. There were a total of 2,000 entries in the test sets, but a full run would have been way way over our budget of $10-20 (we also had to leave some room for fine-tuning costs). The end results were the following on the Lean 4 benchmarks:

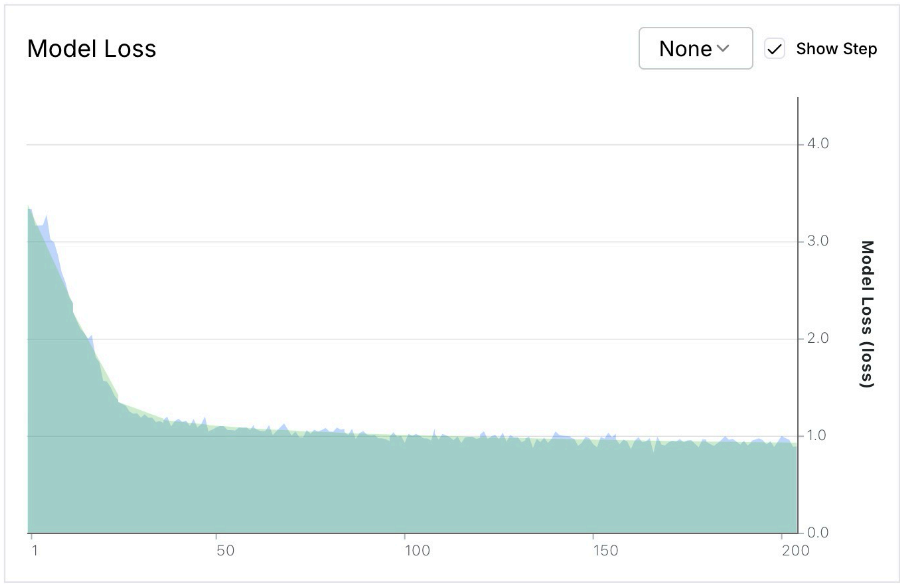




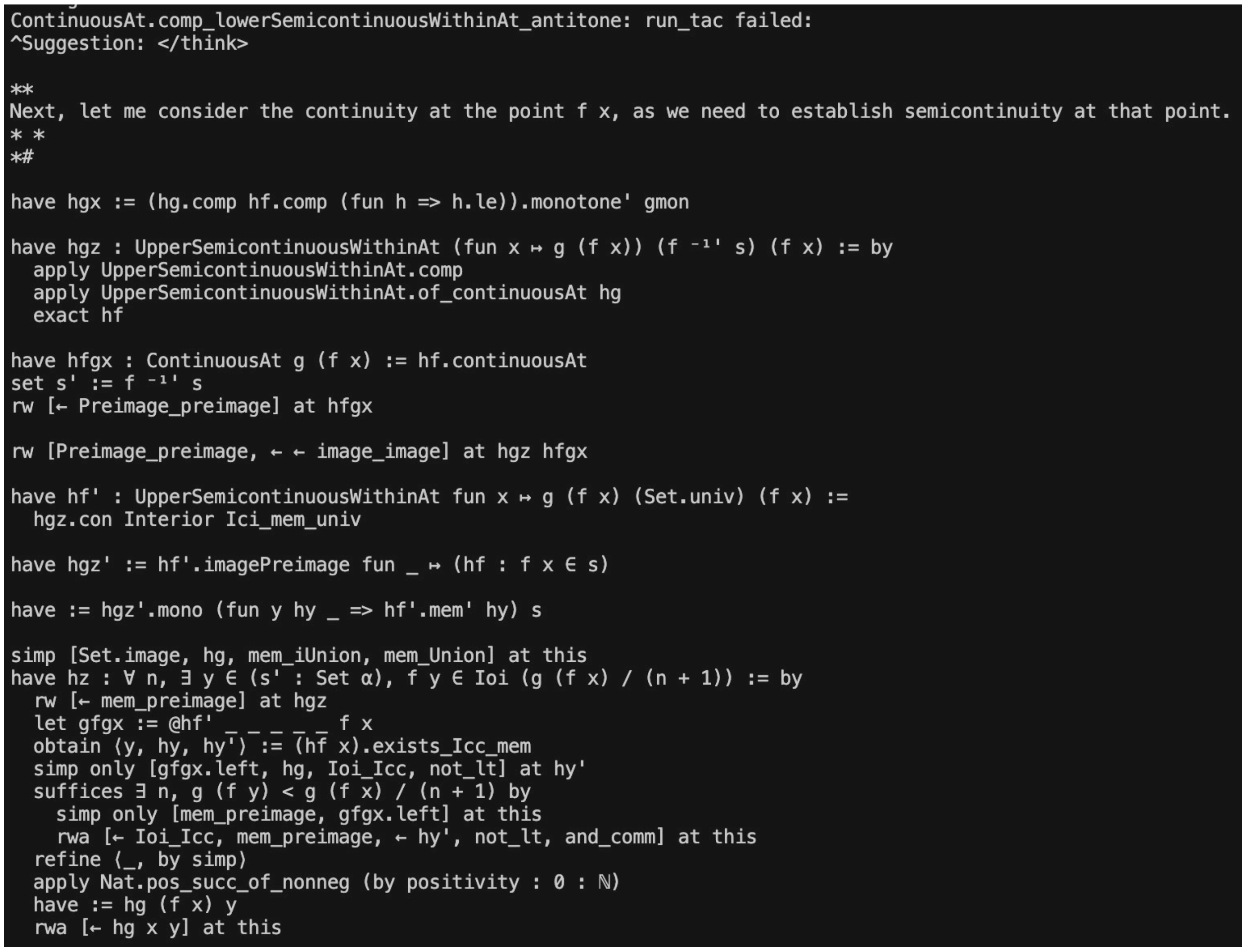
Note that the ReProver accuracies are different from earlier because the earlier results were for the Lean 3 benchmark. We chose Lean 4 because it is now the standard version of Lean. The base version of DeepSeek v3.2 underperforms ReProver in the "random" split but overperforms in the "novel_premises" split. For the "random" set, average proof length was ~1.8 steps and average search time was ~59.5 seconds. For the "novel_premises" set, the average proof length was ~2.2 steps and the average search time was ~59 seconds. Corresponding statistics for ReProver were not made available by Yang et al.

Fine-tuning

Due to budget constraints, we were not able to fine-tune a large frontier model like DeepSeek V3.2. Instead, as a demonstration and test, we fine-tuned a small model Qwen3-8B on a generated dataset from the “random” split with 25,000 training examples (52,853 pairs) and 1,000 validation examples (2,313 pairs). Here is the plot of the model loss over the course of the fine-tuning job:



It is easy to notice that gains in performance very quickly plateaued and we ultimately maintained a loss of just under 1.0. In an attempt to measure performance gains from fine-tuning, we attempted to benchmark both the base Qwen3-8B and the fine-tuned version. However, both runs had to quickly be terminated due to the model not being able to output the tactics in the requested structure. This is an example of bad output:

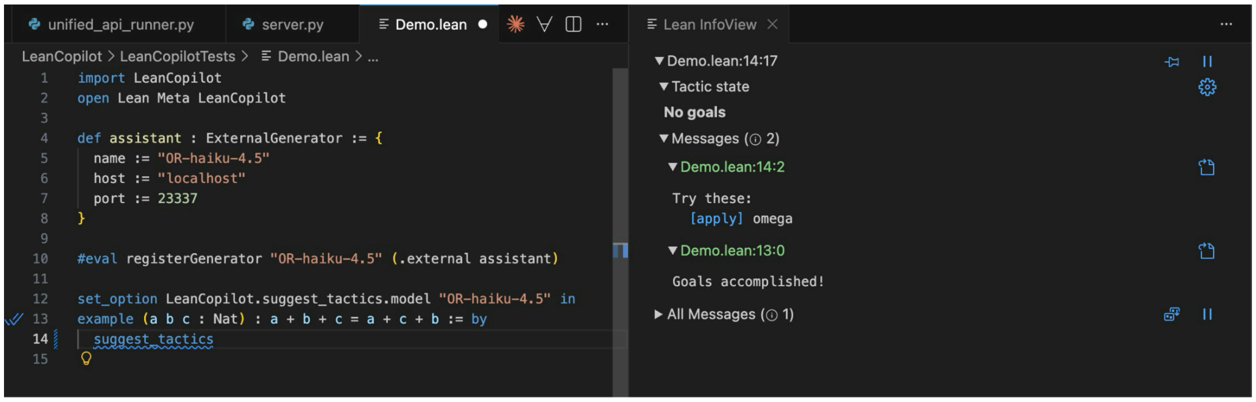


```
ContinuousAt.comp_lowerSemicontinuousWithinAt_antitone: run_tac failed:
^Suggestion: </think>

**
Next, let me consider the continuity at the point f x, as we need to establish semicontinuity at that point.
* *
*#

have hgx := (hg.comp hf.comp (fun h => h.le)).monotone' gmon

have hgz : UpperSemicontinuousWithinAt (fun x ↦ g (f x)) (f ⁻¹' s) (f x) := by
  apply UpperSemicontinuousWithinAt.comp
  apply UpperSemicontinuousWithinAt.of_continuousAt hg
  exact hf

have hfgx : ContinuousAt g (f x) := hf.continuousAt
set s' := f ⁻¹' s
rw [← Preimage_preimage] at hfgx

rw [Preimage_preimage, ← ← image_image] at hgz hfgx

have hf' : UpperSemicontinuousWithinAt fun x ↦ g (f x) (Set.univ) (f x) :=
  hgz.con Interior Ici_mem_univ

have hgz' := hf'.imagePreimage fun _ ↦ (hf : f x ∈ s)

have := hgz'.mono (fun y hy _ => hf'.mem' hy) s

simp [Set.image, hg, mem_iUnion, mem_Union] at this
have hz : ∀ n, ∃ y ∈ (s' : Set α), f y ∈ Ioi (g (f x) / (n + 1)) := by
  rw [← mem_preimage] at hgz
  let gfgx := @hf' _ _ _ _ _ f x
  obtain ⟨y, hy, hy'⟩ := (hf x).exists_Icc_mem
  simp only [gfgx.left, hg, Ioi_Icc, not_lt] at hy'
  suffices ∃ n, g (f y) < g (f x) / (n + 1) by
    simp only [mem_preimage, gfgx.left] at this
    rwa [← Ioi_Icc, mem_preimage, ← hy', not_lt, and_comm] at this
  refine ⟨_, by simp⟩
  apply Nat.pos_succ_of_nonneg (by positivity : 0 : ℕ)
  have := hg (f x) y
  rwa [← hg x y] at this
```

It appears that the “thinking” capabilities of the model interfered with the output. However, other errors included the model writing the same tactic repeated hundreds of times in the same response.

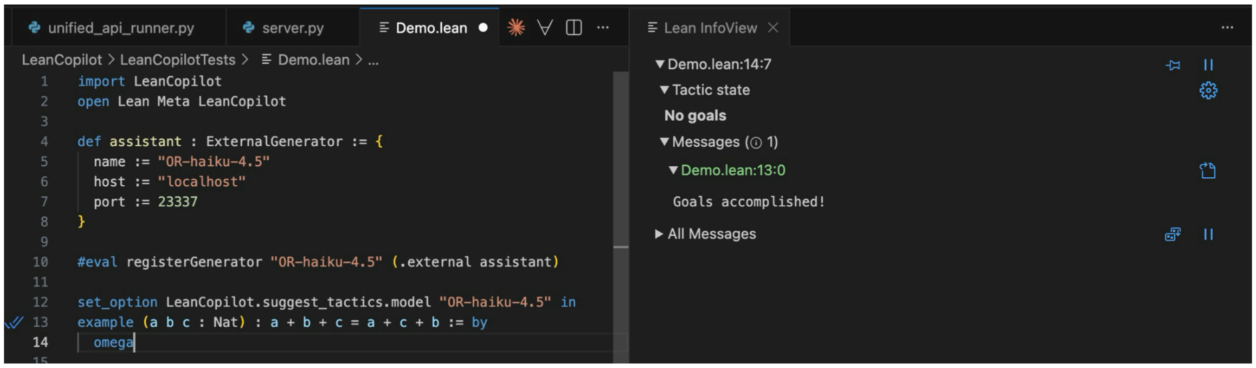
Assistant

The “suggest_tactics” command for the assistant proved successful for the following small demo example. Clicking to apply the suggested tactic resulted in the theorem being proved:



```lean
import LeanCopilot
open Lean Meta LeanCopilot

def assistant : ExternalGenerator := {
  name := "OR-haiku-4.5"
  host := "localhost"
  port := 23337
}

#eval registerGenerator "OR-haiku-4.5" (.external assistant)

set_option LeanCopilot.suggest_tactics.model "OR-haiku-4.5" in
example (a b c : Nat) : a + b + c = a + c + b := by
  suggest_tactics
```

```
▼Demo.lean:14:17
 ▼Tactic state
  No goals
 ▼Messages (ⓘ 2)
  ▼Demo.lean:14:2
   Try these:
     [apply] omega
  ▼Demo.lean:13:0
   Goals accomplished!
▶All Messages (ⓘ 1)
```



```lean
import LeanCopilot
open Lean Meta LeanCopilot

def assistant : ExternalGenerator := {
  name := "OR-haiku-4.5"
  host := "localhost"
  port := 23337
}

#eval registerGenerator "OR-haiku-4.5" (.external assistant)

set_option LeanCopilot.suggest_tactics.model "OR-haiku-4.5" in
example (a b c : Nat) : a + b + c = a + c + b := by
  omega
```

```
▼Demo.lean:14:7
 ▼Tactic state
  No goals
 ▼Messages (ⓘ 1)
  ▼Demo.lean:13:0
   Goals accomplished!
▶All Messages
```

Discussion

Conclusion

References

Yang et al., “LeanDojo: Theorem Proving with Retrieval-Augmented Language Models”, *arXiv*, 2023, https://doi.org/10.48550/arXiv.2306.15626

Harrison J., Urban J., and Wiedijk F., “History of Interactive Theorem Proving” Handbook of the History of Logic, vol. 9, Computational Logic, 2014. https://www.cl.cam.ac.uk/~jrh13/papers/joerg.pdf

Li et al., “A Survey on Deep Learning for Theorem Proving”, *arXiv*, 2024, https://doi.org/10.48550/arXiv.2404.09939

Boye J. & Moell B., “Large Language Models and Mathematical Reasoning Failures”, *arXiv*, 2025, https://arxiv.org/html/2502.11574v1

Kalisyk et al., “Reinforcement Learning of Theorem Proving”, *Advances in Neural Information Processing Systems*, 2018, https://proceedings.neurips.cc/paper_files/paper/2018/file/55acf8539596d25624059980986aaa78-Paper.pdf

Ahn et al., “Large Language Models for Mathematical Reasoning: Progresses and Challenges”, *arxiv*, 2024, https://doi.org/10.48550/arXiv.2402.00157

Zhang D., Dai Q., and Peng H., “The Best Instruction-Tuning Data are Those That Fit”, *arxiv*, 2025, https://doi.org/10.48550/arXiv.2502.04194

Kirchner et al., “Prover-Verifier Games Improve Legibility of LLM Outputs”, *arxiv*, 2024, https://doi.org/10.48550/arXiv.2407.13692

Zhao et al., “Empowering Large Language Model Agents through Action Learning”, *arxiv*, 2024, https://doi.org/10.48550/arXiv.2402.15809

Zhang et al., “DAG-Math: Graph-Guided Mathematical Reasoning in LLMs”, *arxiv*, 2025, https://doi.org/10.48550/arXiv.2510.19842