AN LLM WITH AN REPL IS A STRONG VERICODING BASELINE IN LEAN

Anonymous authors
Paper under double-blind review

ABSTRACT

LLMs are rapidly changing how software is written. However, testing alone is insufficient; formal verification is therefore needed to rigorously guarantee correctness. Given a specification as the source of truth, LLMs can synthesize both the implementation and proof of conformance, a paradigm recently termed vericoding. These proofs are machine-checkable via interactive theorem provers such as Lean. While existing LLM-based provers demonstrate remarkable performance, they often constrain how LLMs interact with Lean. We ask how far we can push LLM performance without sophisticated harnesses by enabling direct interactive Lean usage. Our baseline, Tiny Prover, combines an iterative agent loop with direct interactive access to Lean. On VERINA and VeriSoftBench, we observe that this setup saturates both benchmarks, surpassing more complex prover architectures.

1 INTRODUCTION

LLMs are rapidly changing how software is written. Given a natural-language description, models can now generate substantial programs, debug them, and iteratively improve the code. However, a fundamental question remains: how can we ensure the LLM-generated code aligns with the developer's intention? Without careful human oversight, such programs can be brittle and prone to intent misalignment, requiring extensive testing to assess correctness.

Testing is enormously useful to ensure conformance, but it can only provide guarantees for the cases covered by the tests. A correctness guarantee for every case covered by the specification requires a more involved approach: formal verification. Instead of describing what software should do only in natural language, the requirements are expressed precisely enough that a mathematical system can check them, such as Lean, Verus, Dafny, etc. Despite the additional complexity, the resulting proofs are machine-checkable, which makes proof generation a natural application for LLMs to reduce human effort. However, prior work suggests that proof generation in the context of algorithm and software engineering remains highly challenging (Ye et al., 2025; Chiosa et al., 2026; Xin et al., 2026), often requires more complex approaches, such as fine-tuning or agentic harnesses, to improve performance.

The task of interest in this work is *vericoding* (Chiosa et al., 2026), where the specification is given and assumed to be correct, and an LLM is required to generate both the implementation (code) and a proof that the code conforms to the specification. The terminology distinguishes itself from a more general and difficult setting of *verifiable code generation* (Ye et al., 2025; Sun et al., 2024; Yang et al., 2025), which requires jointly generating the specification, code, and proof. The suggested bottleneck of *vericoding* is proof generation (Ye et al., 2025; Chiosa et al., 2026), which is more difficult than code generation task alone. Therefore, improving proof generation performance significantly impacts the performance of the LLM on vericoding tasks.

Existing work for LLM-based theorem proving follows several directions, from fine-tuning to agentic harnesses. Fine-tuning approaches (Lin et al., 2025a;b; Chen et al., 2025b;a; Xin et al., 2024; Ren et al., 2025) typically involve large-scale formal datasets or synthetic data generation from interaction with the compiler, followed by supervised fine-tuning and reinforcement learning to improve performance on theorem proving tasks. However, these approaches are usually tied to specific versions of Lean, which can become outdated given Lean's fast pace of development, while also requiring substantial computational resources for training. Alternatively, agentic harness approaches

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

propose to utilize general-purpose LLMs wrapped in multiple agentic layers to improve theorem-proving performance (Kung et al., 2026; Varambally et al., 2025; Agarwal et al., 2026). These approaches scale with the performance of the base LLM and benefit from other existing capabilities such as tool calling.

However, given the complexity of these approaches and the evident generalization capability of LLMs, a fundamental question should be raised: is designing more sophisticated agentic architecture the current bottleneck for effective Lean vericoding? State-of-the-art LLMs have improved rapidly in mathematical problem solving and software engineering. Yet prior research suggests they still often struggle to produce formally verified programs. We hypothesize that a substantial part of the problems arise not from the lack of reasoning capability, but rather from the constrained interaction with Lean. For example, interactive theorem proving requires repeatedly inspecting the proof state, querying definitions and types, searching for relevant lemmas, testing intermediate lemmas, fixing syntax, etc. Many formal-proving evaluations expose only limited interaction with Lean, omitting capabilities that human experts routinely rely on during formalization. This is analogous to evaluating a programmer without an IDE, compiler, debugger, or an environment for experimentation. Some agentic harnesses imitate some of the capability, for example, by giving the agent a search capability, instead of relying on existing Lean functionalities such as `#find` or `#print`. We therefore remove this restriction and study a simple baseline: an LLM paired with a direct interactive Lean Read-Eval-Print-Loop (REPL) in an iterative workflow. This setup differs substantially from using Lean compiler feedback alone since the REPL is stateful and permits arbitrary intermediate experiments, proof-state inspection, declaration queries, native search, and suggestion tactics. Our goal is to establish how much vericoding performance can be obtained from an iterative agent with effective tool use, before introducing specialized training or more elaborate agentic architectures.

Our contributions are:

- We propose a simple Lean vericoding baseline that equips an LLM with direct access to an interactive Lean REPL, without fine-tuning or a sophisticated agentic harness.
- We evaluate the baseline on VERINA and VeriSoftBench (Aristotle subset), where it saturates both benchmarks and surpasses more complex prover architectures.
- We analyze the Lean usage patterns from the records to inform future design of a more effective agentic harness for vericoding agents.

2 Related Work

2.1 Verifiable code generation and vericoding

LLM-based code generation can produce functional programs but is prone to intent misalignment. Nonetheless, LLM-assisted development, "vibecoding", etc., are increasingly becoming the new norm in software engineering, which motivates the need for new paradigms that provide higher correctness guarantees. Verifiable code generation (Sun et al., 2024; Ye et al., 2025; Yang et al., 2025) refers to the paradigm where, given a natural language description, the LLM jointly generates the specification, code, and proof. The formal specification captures the natural language intent, while the code implements the specification and the proof certifies the code behaves as described in the specification. The proof artifacts are machine-checked, which provides an automated system for ensuring code correctness. The specification therefore plays an important role in aligning the informal intent with the formal language.

A narrower paradigm where the formal specification is given and assumed to be the ground truth, and the LLM is required to generate correct code and proof of conformance, is recently termed vericoding (Chiosa et al., 2026). Vericoding isolates the task to only the formal environment and therefore enables deterministic and automated evaluation using formal systems such as Lean, Verus, or Dafny. If the vericoding task can be automated to a high degree by LLM-based provers, the reliability of the verifiable code generation system can be moved toward specification fidelity. This specification-as-the-source-of-truth concept is also highly relevant in existing software engineering paradigms, such as spec-driven development.

2.2 LLM-BASED AUTOMATED THEOREM PROVING

Advances in LLMs have enabled the use of informal LLMs in automated theorem provers. Fine-tuning approaches produce specialized prover LLMs that directly generate the formal proofs. Early attempts at prover LLM systems include GPT-f (Polu & Sutskever, 2020) on the Metamath formalization language. With the rapid improvement in LLM performance, larger prover LLMs have also been gradually introduced, such as DeepSeekProver (Xin et al., 2024; Ren et al., 2025), Goedel-Prover (Lin et al., 2025a;b), SeedProver (Chen et al., 2025b;a) for the Lean interactive theorem prover through a mix of large-scale formal proof datasets, synthetic data generation methods, supervised fine-tuning, reinforcement learning, etc. These approaches have yielded remarkable performance given their respective size compared to general-purpose LLMs. However, formal prover systems like Lean are a moving target, as new updates may change functionalities that existing formal or synthetic datasets rely on. Since prover LLMs are usually tied to some specific versions of Lean, they can quickly become outdated. Furthermore, the synthetic data generation and fine-tuning process also incurs substantial computational cost.

Alternatively, many LLM-based provers aim to directly apply a general-purpose LLM to the task by wrapping it in an agentic harness. By using a general-purpose LLM, these provers also benefit from the existing capacities, such as instruction following and tool calling, as well as the rapid development pace of these foundation models. It has been demonstrated that general-purpose LLMs have strong mathematical reasoning capability and achieve strong informal proving results (Luong & Lockhart, 2025; OpenAI, 2026). Furthermore, coding assistants like Claude Code [1] or Codex [2] have gradually become an integral part of the software engineering workflows. Nonetheless, prior research suggests that despite these advantages, LLMs still face difficulty in generating formal proof (Tsoukalas et al., 2024; Ye et al., 2025; Xin et al., 2026), which motivates complex scaffolding to improve performance. Many agentic provers are designed for formal mathematics (Jiang et al., 2022; Ospanov et al., 2025; Varambally et al., 2025; Kung et al., 2026; Requena et al., 2026). For example, *HILBERT* (Varambally et al., 2025) proposes to employ a multi-stage scaffold with multiple steps such as theorem retrieval, sketch generation, subgoal extraction, etc. to achieve strong performance on MiniF2F (Zheng et al., 2021) and PutnamBench (Tsoukalas et al., 2024). LEAP proposes another agentic design inspired by the workflow of human experts and achieves strong results on Lean-IMO-Bench (Kung et al., 2026). A recent approach on the vericoding domain includes *Inductive Deductive Synthesis* (Agarwal et al., 2026), which leverages an iterative code refinement and proof refinement loop to achieve good results on several benchmarks, including VERINA (Ye et al., 2025).

These systems often prescribe pipelines for retrieval, decomposition, or proof refinement. However, some of these functions are already supported by general-purpose LLMs and Lean itself, for example, decomposition through model reasoning and retrieval or proof suggestions through Lean commands and tactics. This overlap also motivates studying whether direct access to a stateful Lean REPL allows the model to compose these existing capabilities without a task-specific pipeline.

3 METHODOLOGY

In this section, we detail the setup of our baseline for vericoding, which we named Tiny Prover.

3.1 ARCHITECTURE

Without a task-specific pipeline, Tiny Prover is a standard iterative agent that interleaves thinking and tool calls until a final submission is produced, as in Fig. 1. Depending on the turn, the model may output a thinking block or not, but a tool call is always produced.

While being intentionally minimal, the full capability of Lean can be exposed directly via tool calls. Therefore, we can compare Tiny Prover, which represents the baseline vericoding capability of the LLM, against existing methods and known results. An even simpler setup would remove the tools, resulting in a direct proof-generation LLM. Because prior work has already observed direct generation to perform poorly, we do not repeat that setting in this study as it provides little new insight.

[1] https://github.com/anthropics/claude-code

[2] https://github.com/openai/codex

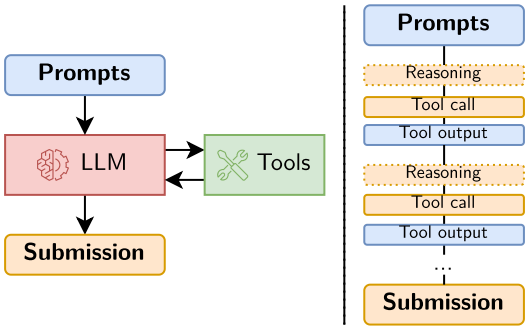


Figure 1: Overview of Tiny Prover (left) and its iterative interaction loop (right). The LLM alternates between reasoning and tool calls until a final verified submission is produced.

3.2 LEAN REPL

We expose Lean's capability via a Lean REPL [3], which returns the result for each Lean snippet together with an environment ID. The environment ID can then be passed to subsequent calls to resume existing environments or omitted to create a new one. The LLM can also insert `sorry` into an exploratory proof to inspect the current proof state and its remaining goals. Although the REPL supports continuing directly from a proof-state index, we omit this capability to simplify the tool interface because the model can instead use a temporary `example` containing the returned goal. For more information and examples, see the Lean REPL's documentation.

Unlike iterative refinement using Lean compiler feedback alone, direct REPL access lets the model use Lean as an exploratory development environment rather than only to validate candidate solutions. The REPL is stateful and accepts arbitrary Lean snippets, permitting intermediate experiments with temporary `examples`, proof-state inspection, declaration queries such as `#check` and `#print`, native search with `#find`, and suggestion tactics such as `exact?`, `apply?`, and `rw?`. By observing Lean usage patterns, we can identify limitations of existing Lean functionality for LLM-based vericoding and inform the design of customized tools or harnesses.

The REPL is only used as an interactive development tool. The final generated code and proof are checked separately by an evaluation harness to determine whether the submission is valid. Although `sorry`, `admit`, and `axiom` may be used during the development phase, they are rejected in the generated solution fields. The final submission must compile, and an additional LLM judge performs a semantic check for language loopholes.

4 EXPERIMENTS

4.1 BENCHMARKS

We evaluate the performance of Tiny Prover on 2 benchmarks, VERINA (Ye et al., 2025) and VeriSoftBench (Xin et al., 2026), as they capture complementary aspects of the *vericoding* task. VERINA directly evaluates vericoding by requiring both an implementation and a proof from a given specification, but primarily consists of self-contained algorithmic tasks. Alternatively, VeriSoftBench evaluates proof generation alone, the principal bottleneck in vericoding identified by prior work, but within repository-scale projects with substantial local context and dependencies. Together, the benchmarks evaluate both end-to-end vericoding and proof generation in a more realistic software environment.

VERINA (Ye et al., 2025) is a high-quality, modular benchmark for verifiable code generation in Lean. The benchmark supports the three core tasks: including specification generation, code generation, and proof generation, which can be evaluated individually or composed. The benchmark comprises 189 manually curated tasks divided into two subsets: Basic (108 problems) and

[3] `https://github.com/leanprover-community/repl`

Advanced (81 problems). We evaluate Tiny Prover on the vericoding setting (code generation and proof generation, specification given as source of truth). The model is provided with only an REPL tool with a budget of 100 calls. The Lean version used for the benchmark is `v4.31.0`.

```
-- VERINA: verina_advanced_1 --

def FindSingleNumber_precond (nums : List Int) : Prop :=
  /- Provided precondition (elided) -/

def FindSingleNumber_postcond (nums : List Int) (result: Int) (h_precond : FindSingleNumber_precond (nums)) : Prop :=
  /- Provided postcondition (elided) -/

def FindSingleNumber (nums : List Int) (h_precond : FindSingleNumber_precond (nums)) : Int :=
  /- Implementation generated by the model -/

theorem FindSingleNumber_postcond_satisfied (nums: List Int) (h_precond : FindSingleNumber_precond (nums)) :
    FindSingleNumber_postcond (nums) (FindSingleNumber (nums) h_precond) h_precond := by
  /- Proof generated by the model -/
```

Figure 2: An example task from the VERINA benchmark. The task prompt includes a natural language description. The specification includes the precondition and postcondition for the implementation, which are provided as the input. The system must then generate a complete code that implements the specification and a proof of code-specification alignment. The task is self-contained within a single file.

VeriSoftBench (Xin et al., 2026) is a repository-scale formal-verification benchmark for Lean. Unlike competition mathematics and Mathlib-centric benchmarks, software verification lives in definition-rich codebases. Proof success is negatively correlated with the size of the transitive dependency closure, therefore requiring significant comprehension of the dependency chain. We evaluate proof generation using the Aristotle subset (100 theorem-proving tasks), a smaller and compilable subset. Due to the repository nature of the benchmark, in addition to the REPL, the LLM also receives a POSIX shell tool for read-only code browsing. The POSIX shell provides standard utilities such as `find`, `grep`, `sed`, etc., but does not provide execution capabilities such as Python, Lean, etc. The shell tool enables the model to inspect the codebase and retrieve relevant context with minimal engineering intervention. The two tools serve complementary roles: the shell supports repository navigation, while the REPL provides interactive Lean execution. To prevent proof leakage, all ground-truth proofs for benchmark targets are removed from the repositories and replaced with `sorry`. The model is provided with a budget of 100 REPL calls and 50 POSIX shell calls. Depending on the repository, the Lean version is modified accordingly to accommodate the REPL.

```
-- VeriSoftBench (Aristotle): task 0 --

/- Provided local file context (elided) -/

theorem getBit_repr { : Nat} :  j, j < 2^  j =  k  Finset.Icc 0 (-1), (getBit k j) * 2^k := by
  /- Proof generated by the model -/
```

Figure 3: An example task from the VeriSoftBench benchmark. The system is provided with the theorem statement and the local file context before the statement and must generate the complete proof. The task requires code browsing and composing a proof from relevant local context and imports.

4.2 Main results

Table 1 demonstrates the results of Tiny Prover on VERINA and VeriSoftBench. We generate k trials of the full Tiny Prover run and report pass@k. One trial denotes a complete independent agent trajectory as in Fig. 1 from the first prompt to the final submission.

As can be observed, given direct interactive access to the Lean REPL, Tiny Prover saturates both benchmarks, achieving 100% on VERINA (pass@1) and VeriSoftBench (Aristotle subset) (pass@2) with a small k. Prior works have experimented with both `Claude-Opus-4.6` and `GPT-5.4`, but they observed unsatisfactory results from a capable model. Direct proof generation does not provide a well-calibrated baseline for vericoding, as it requires the model to generate a whole proof without any context about the local environment, especially for an interactive environment such as Lean.

Table 1: Performance of Tiny Prover against existing methods.

Model	VERINA		VeriSoftBench (Aristotle)	
VERINA[1] (Ye et al., 2025)	20.1%*	(pass@64)	-	
VeriSoftBench[2] (Xin et al., 2026)	-		65%	(pass@8)
Goedel-Code-Prover[3] (Li et al., 2026)	68.8%*	(pass@128)	-	
Inductive Deductive Synthesis [4] (Agarwal et al., 2026)	93.12%		-	
Aristotle** (Achim et al., 2025)	96.8%		69%	
Aleph Prover** (Logical Intelligence, 2026)	100%		94%	
Tiny Prover (`Claude-Opus-4.6`)	97.35%	(pass@1)	98% 100%	(pass@1) (pass@2)
Tiny Prover (`GPT-5.4`)	100%	(pass@1)	98% 100%	(pass@1) (pass@2)

[1] The best reported result uses `DeepSeek-Prover-V2-7B` with iterative refinement.
[2] The reported result excluding Aristotle uses `Gemini-3-Pro`.
[3] In the same work, the authors reported direct proof generation using `Claude-Opus-4.6` achieving less than 15% with pass@128.
[4] In the same work, the authors reported using coding harness including Claude Code (`Claude-Opus-4.6`) and Codex (`GPT-5.4`) achieving less than 80% on VERINA.
* Proof-only performance.
** Proprietary solvers.

On the other hand, existing methods provide a structured harness to simulate various capabilities, including retrieval, search, and recursive decomposition, which share certain overlaps with model capability or Lean's functionality. Giving the LLM a Lean REPL helps elicit these capabilities and demonstrate strong performance.

We evaluate Tiny Prover using `Claude-Opus-4.6` [4] and `GPT-5.4` [5], released in February and March 2026, respectively. We select these models as high-capability models from different providers with broadly similar tiers and release dates, avoiding comparison across substantially different model generations. Each model family also provides a lighter counterpart, i.e., `Claude-Sonnet-4.6` [6] and `GPT-5.4-mini` [7], which enables a robustness analysis across model capability. Both `Claude-Opus-4.6` and `GPT-5.4` predate Lean 4.31.0, which is the version used for the VERINA benchmark. However, we observe the LLMs face little difficulty adapting to a newer version when given REPL access. Furthermore, although newer and stronger models were available at the time of experiments, our objective is not to maximize benchmark performance, but rather to determine whether vericoding capabilities are already present and can be elicited.

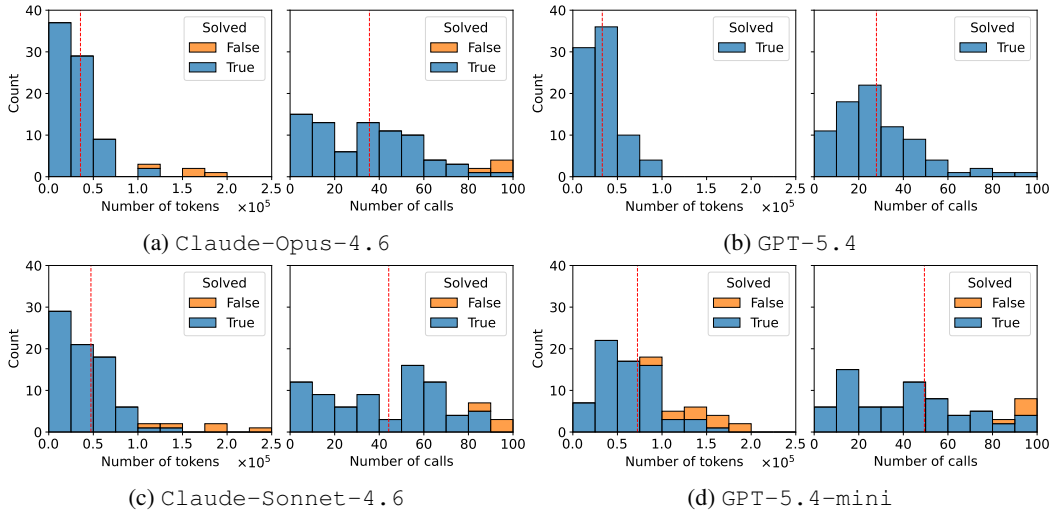
4.3 Cross-model robustness

We further extend the main result using the VERINA (Advanced subset) with less capable models, including `Claude-Sonnet-4.6` and `GPT-5.4-mini`, to verify whether our observation generalizes. Since the Advanced subset is more difficult, it is expected that the results of different LLMs will be lower for this experiment compared to the main result. We report pass@1 result for all models. As can be observed from Table 2, `Claude-Sonnet-4.6` and `GPT-5.4-mini` were able to solve the majority of the Advanced problems with an average of fewer than 50 REPL calls. `Claude-Sonnet-4.6` manages to achieve performance close to `Claude-Opus-4.6`, despite requiring significantly more REPL calls and tokens on average. Overall, all models solved more than 80% of the VERINA Advanced problems, while prior works suggest a less than 80% rate for `Claude-Opus-4.6` and `GPT-5.4` for the full VERINA benchmarks.

[4] `https://www.anthropic.com/news/claude-opus-4-6`
[5] `https://openai.com/index/introducing-gpt-5-4/`
[6] `https://www.anthropic.com/news/claude-sonnet-4-6`
[7] `https://openai.com/index/introducing-gpt-5-4-mini-and-nano/`

Table 2: Results on VERINA (Advanced subset) for Tiny Prover with different base LLMs.

Model (context window)	Rate		Avg calls	Avg tokens
`Claude-Opus-4.6` (1M)	77/81	95.06%	35.59	35554
`GPT-5.4` (1.05M)	81/81	100%	27.93	33019
`Claude-Sonnet-4.6` (1M)	76/81	93.83%	44.25	47308
`GPT-5.4-mini` (0.4M)	69/81	85.19%	49.44	72475



(a) `Claude-Opus-4.6` (b) `GPT-5.4`

(c) `Claude-Sonnet-4.6` (d) `GPT-5.4-mini`

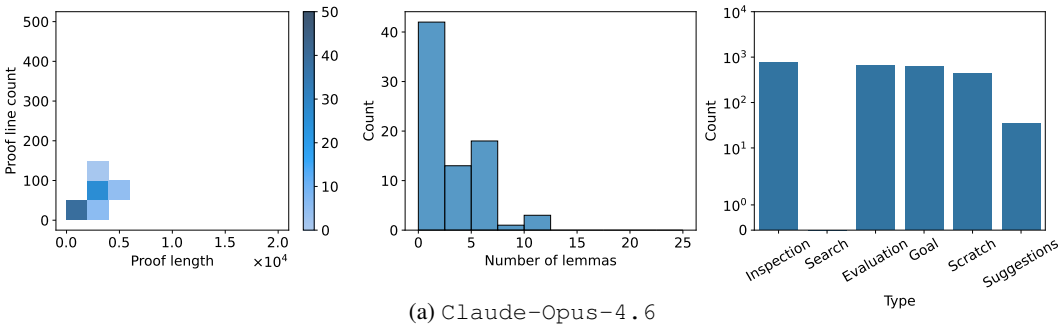
Figure 4: Number of tokens and REPL calls used across models for VERINA (Advanced subset).

Fig. 4 demonstrates the distribution of tokens and REPL calls for the four models in Table 2. Most successful problems require relatively few total conversation tokens (less than 100 000) and REPL calls (less than 60). For more difficult problems, the LLM consumes almost all REPL calls (greater than 80) but still may fail to resolve the problem. With most model context windows at around one million tokens, VERINA advanced problems typically require less than 1/10 of the window to be solved. The only exception here is `GPT-5.4-mini` with a context window of 400 000 tokens, which may partially explain the deteriorating performance compared to other models. Since most successes occur substantially before the limit, increasing the budget would primarily increase computation on a small set of difficult tasks, with uncertain marginal benefit.

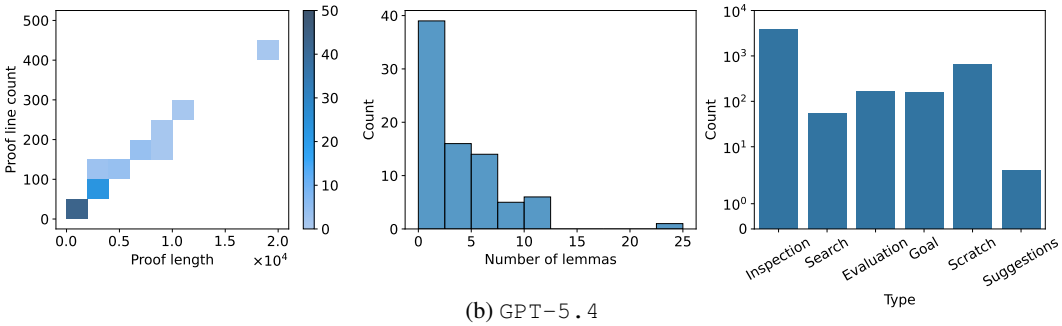
4.4 Lean usage patterns

Fig. 5 summarizes the statistics obtained from the generated proof artifact and the REPL calls from the run records. Since failed runs do not produce a valid proof artifact, they are omitted from this plot, which includes only the successful runs. The left plot demonstrates the distribution of the generated proof size by character and line count. The center plot demonstrates the number of lemmas that have been generated by the LLMs in the final solution. Together, these two plots provide rough proxies for the LLM's intrinsic capability to structure and decompose the problem to form a final solution. `Claude-Opus-4.6` is capable of generating long proofs (less than 200 lines), but most of the mass is concentrated at shorter proofs. Similarly, `Claude-Opus-4.6` can also efficiently decompose the problems into multiple lemmas to aid in the theorem-proving process. On the other hand, `GPT-5.4` generates significantly longer proofs and more lemmas, with proofs exceeding 400 lines, which may partially explain the greater success rate. Since `GPT-5.4` is more capable at decomposing the problem into lemmas, the LLM can solve more complex tasks efficiently.

The right plot of Fig. 5 counts the log-scale occurrence of Leans exploratory features during proof development, despite these features being absent from the final proof artifacts. We choose to decompose these facilities into six categories as follows: Inspection (using `#check`, `#print`, `#synth`), Search (using `#find`), Evaluation (using `#eval`, `#reduce`), Goals (inserting `sorry` to query cur-



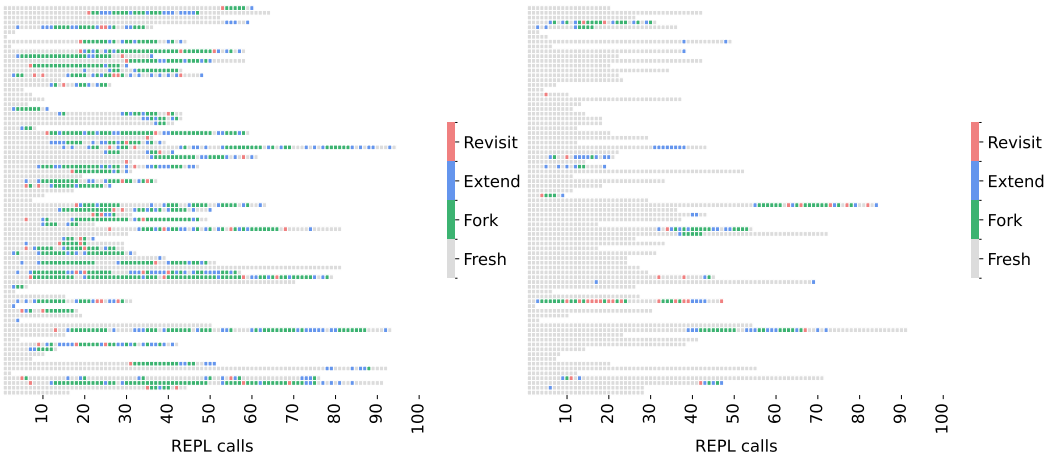
(a) `Claude-Opus-4.6`



(b) `GPT-5.4`

Figure 5: Solution and interaction statistics for 77/81 problems of VERINA (Advanced subset) solved by both `Claude-Opus-4.6` and `GPT-5.4`. **Left**: distribution of generated proof size, measured in characters and lines. **Center**: distribution of auxiliary lemmas generated for the final proof. **Right**: aggregate count of exploratory Lean feature categories in REPL calls on a log scale.

rent proof goals), Scratch (using `example` to form intermediate experiments), Suggestions (using `exact?`, `apply?`, `rw?`, `simp?`). Both LLMs heavily use these features during the development process, especially the Inspection category. These interaction statistics highlight the need for interactive Lean features beyond those used in the final proof generation, as they help the model effectively explore and construct the proof.



(a) `Claude-Opus-4.6` (b) `GPT-5.4`

Figure 6: REPL environment usage by `Claude-Opus-4.6` and `GPT-5.4` on the VERINA (Advanced subset). Each row represents a task. Colors classify the environment ID supplied to the current call. **Gray** (Fresh): no environment ID is provided, creating a fresh environment. **Green** (Fork): the provided environment ID is the same as that provided to the previous call, creating a fork from the same environment. **Blue** (Extend): the provided environment ID was returned by the previous call, extending the latest environment. **Red** (Revisit): the provided environment ID was returned by an earlier, non-immediately preceding call, revisiting an existing environment.

Fig. 6 demonstrate the REPL environment usage by `Claude-Opus-4.6` and `GPT-5.4` on the VERINA (Advanced subset). The majority of calls create a fresh environment, which can serve as either an exploratory scratchpad by constructing short snippets, a compilation sandbox by submitting self-contained snippets for local experiments, or a complete candidate-proof check. Nonetheless, both LLMs can selectively utilize the stateful capability of the REPL. The fork call begins after either a revisit or extend call, from which we can infer that the model uses a stable working environment from the previous calls to continue the experimentation process or iteratively construct the proof step-by-step.

These results may suggest that the LLM can use Lean effectively if given the right tool to do so. However, while the REPL provide access to these capabilities, it may not provide the most efficient interface for certain tasks. For example, the extensive use of the Inspection and Search category for `GPT-5.4` suggests the model would benefit from structured access to local declarations, types, relevant lemmas, etc., instead of repeatedly querying them through individual commands.

5 Dicussion and Conclusion

In this work, we explored the coding paradigm where AI generates an implementation and a proof that it satisfies a given source-of-truth specification, and a formal verification system independently checks that proof. Generating software this way is called vericoding.

Instead of building sophisticated agent harnesses for such vericoding tasks, we showed that a simple architecture (Tiny Prover) of an LLM with a REPL is a strong baseline solution that saturates VERINA and VeriSoftBench. Tiny Prover combines an iterative agent loop with direct, interactive access to Lean. Rather than generating an entire solution and submitting it only at the end, the model can try an approach, inspect Lean's response to the approach, including where it fails, revise its solution, and continue until Lean successfully verifies it. More complex architectures should be able to justify the performance gain outweighing the additional complexity. Furthermore, due to its simplicity, Tiny Prover can also serve as a component of a more structured and elaborated automated proof system. Future benchmarks will need to move closer to real software engineering: larger repositories, deeper dependencies, longer-horizon changes, implementation and proof generation together, performance constraints, and more complex specifications.

There is also a second boundary to confront. In these benchmarks, the formal specification is provided and treated as the source of truth. In real systems, proving that code conforms to a specification does not prove that the specification correctly captures human intent. As vericoding can be increasingly automated, part of the reliability problem moves upstream toward specification fidelity. But we believe the larger lesson extends beyond Lean and verified code. A large part of todays LLM stack asks probabilistic models both to generate answers and to judge whether those answers are correct. Formal systems offer another architecture. The neural model can do what it does extraordinarily well: understand intent, reason through complex problems, generate candidate solutions, explore alternatives, and recover from failure. Wherever important requirements can be expressed precisely, a formal system can independently determine whether those requirements have been satisfied and provide a stronger guarantee.

AI use statement

In this work, we study LLM models' ability to generate code and proofs in Lean. We also used generative AI tools to aid and polish writing, such as editing some sentences to improve readability and clarity. No paragraphs have substantial AI-generated content. Similarly, AI is only used to semantically check experiment code correctness, without generating any substantial part of the experimentation pipeline. We have reviewed all AI-assisted work.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: IMO-level Automated Theorem Proving, October 2025.

Shubham Agarwal, Alexander Krentsel, Shu Liu, Mert Cemri, Audrey Cheng, Rui Meng, Tomas Pfister, Chun-Liang Li, Sylvia Ratnasamy, Aditya Parameswaran, Matei Zaharia, Ion Stoica, and Mohsen Lesani. Inductive Deductive Synthesis: Enabling AI to Generate Formally Verified Systems, May 2026.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-Prover 1.5: Mastering Undergraduate-Level Theorem Proving via Learning from Experience, December 2025a.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. Seed-Prover: Deep and Broad Reasoning for Automated Theorem Proving, August 2025b.

Ionel Emilian Chiosa, Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A Benchmark for Vericoding: Formally Verified Program Synthesis. *Transactions on Machine Learning Research*, March 2026. ISSN 2835-8856.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations*, September 2022.

Po-Nien Kung, Linfeng Song, Dawsen Hwang, Jinsung Yoon, Chun-Liang Li, Simone Severini, Mirek Olšák, Edward Lockhart, Quoc V. Le, Burak Gokturk, Thang Luong, Tomas Pfister, and Nanyun Peng. LEAP: Supercharging LLMs for Formal Mathematics with Agentic Frameworks, June 2026.

Zenan Li, Ziran Yang, Mike He, Haoyu Zhao, A. Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-Code-Prover: Hierarchical Proof Search for Open State-of-the-Art Code Verification. In *Third Conference on Language Modeling*, August 2026.

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. In *Second Conference on Language Modeling*, August 2025a.

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. In *The Fourteenth International Conference on Learning Representations*, October 2025b.

Logical Intelligence. Aleph Prover. https://logicalintelligence.com/aleph-prover, 2026.

Thang Luong and Edward Lockhart. Advanced version of Gemini with Deep Think officially achieves gold-medal standard at the International Mathematical Olympiad. https://deepmind.google/blog/advanced-version-of-gemini-with-deep-think-officially-achieves-gold-medal-standard-at-the-international-mathematical-olympiad/, 2025.

OpenAI. Our First Proof submissions. https://openai.com/index/first-proof-submissions/, 2026.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and Lean Collaboration for Advanced Formal Reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Stanislas Polu and Ilya Sutskever. Generative Language Modeling for Automated Theorem Proving, September 2020.

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, July 2025.

Borja Requena, Austin Letson, Krystian Nowakowski, Izan Beltran Ferreiro, and Leopoldo Sarra. A Minimal Agent for Automated Theorem Proving. In *Forty-Third International Conference on Machine Learning*, June 2026.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings*, pp. 134–155, Berlin, Heidelberg, July 2024. Springer-Verlag. ISBN 978-3-031-65111-3. doi: 10.1007/978-3-031-65112-0_7.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating Neural Theorem-Provers on the Putnam Mathematical Competition. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, November 2024.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively Building Formal Proofs with Informal Reasoning. In *The Fourteenth International Conference on Learning Representations*, October 2025.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data, May 2024.

Yutong Xin, Jocelyn Qiaochu Chen, Greg Durrett, and Isil Dillig. VeriSoftBench: Repository-Scale Formal Verification Benchmarks for Lean. In *Third Conference on Language Modeling*, August 2026.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin E. Lauter, Swarat Chaudhuri, and Dawn Song. Position: Formal Mathematical Reasoning—A New Frontier in AI. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 82384–82398. PMLR, October 2025.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking Verifiable Code Generation. In *The Fourteenth International Conference on Learning Representations*, October 2025.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: A cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations*, October 2021.

A PROMPTS

Some parts of the prompts are adapted from VERINA (Ye et al., 2025).

For VERINA, the agent receives the following input: **Lean 4 instruction** + **Code-proof instruction** + **VERINA submit instruction** + **REPL instruction** + VERINA task template + **Message**.

For VeriSoftBench, the agent receives the following input: **Lean 4 instruction** + **Code-proof instruction** + **VeriSoftBench submit instruction** + **REPL instruction** + **Shell instruction** + VeriSoftBench task template + **Message**.

Lean 4 instruction

```
# Lean 4 Instruction
You are an expert in Lean 4 programming and theorem proving.
Your solution should:
- Be well-documented with comments if necessary.
- Follow Lean 4 best practices and use appropriate Lean 4 syntax
and features.
- DO NOT use Lean 3 syntax or features.
- DO NOT use catch-all generic imports like `import Mathlib`. Only
import what you need.

Hint:
- Use a[i]! instead of a[i] when a is an array or a list when
necessary.
- To use `lemma` instead of `theorem`, you need to use `import
Mathlib.Tactic.Lemma`.
```

Code-proof instruction

```
# Solution Instruction
You are an expert in Lean 4 programming and theorem proving.
- First, create an informal natural language step-by-step code
that addresses the problem. You MUST completely solve the problem
in natural language first.
- Then, create the formal solution using the informal solution.
You MUST complete the solution before using the `submit` tool.
- Decompose the solution into smaller units when possible.
```

REPL instruction

```
# REPL Instruction
You are provided with a Lean interactive REPL to assist with your
task.
- You MUST use the REPL to make sure that your Lean code compiles
before submission.
- The REPL does not import any libraries or include any code by
default. You need to import the library or define the code to use
it.
- You are provided with a sufficient but limited number of REPL
calls, so use it efficiently.
- You should use the REPL to check if some functions, theorems, or
imports exist.
- If the REPL only returns `{'env': _}`, it means that the code
compiles.
- You MUST use the correct `env` (returned by REPL) if you wish to
continue with some previous code.
- To enter tactic mode, issue a command containing a `sorry`,
which will return the proof goals.
- If you exceed max calls, wrap up your work and submit anyway.
You MUST explain your unfinished submission and details of the
failure.

Hint:
- Always append `set_option linter.unusedVariables false` after
import to avoid unnecessary linter warnings.
```

Semantic filtering instruction

```
# Semantic verification
```

```
You are an expert in Lean 4 programming and theorem proving.

Another LLM has solved the problem using Lean 4, and the solution
is successfully compiled.
The solution has already been checked for phrases like `sorry`,
`admit`, or `axiom`.
Your goal is to perform a semantic check to verify whether the
model successfully solves the goal without exploiting language
loopholes.
Use the `submit` tool to submit your final answer.
```

Message

```
Solve the problem using the provided template and submit the
solution.
```

VERINA submit instruction

```
{SUBMIT_INSTRUCTION}
- DO NOT import Std or Init.
- DO NOT use `set_option` in the import.
- Be careful of indentations. The first line of `code` or `proof`
submission MUST be indented.
```

VERINA semantic filtering message

```
Another LLM has filled in the following fields:
- code_imports
- proof_imports
- code_aux
- proof_aux
- code
- proof

Perform a semantic check to verify whether the LLM has solved the
problem.
Be careful of potential loopholes that an LLM may exploit, such as
`native_decide + @[implemented_by]`, metaprogramming tricks, etc.
Explain your reasoning before submission.
```

Shell instruction

```
# Shell Instruction
You are provided with a POSIX shell tool to explore the read-only
code base, which is mounted at the current directory.
- Use the shell to retrieve relevant Lean4 code. Only POSIX shell
commands exist.
- You are provided with a sufficient but limited number of shell
calls, so use them efficiently.
- Each command is independent and is executed in a fresh shell
environment.
```

VeriSoftBench submit instruction

```
{SUBMIT_INSTRUCTION}
- You can use anything available in the template, including lemmas
and proofs with `sorry`.
- DO NOT use `sorry`, `admit`, or `axiom` for the sections that
you need to fill in: {{proof}} and {{proof_aux}}.
```

```
- The templating system will only check for `sorry`-like phrases
in your provided {{proof}} and {{proof_aux}} and not the whole
template.
- You MUST correctly insert `:= by` or equivalent in {{proof}}.
```

VeriSoftBench semantic filtering message

```
Another LLM has filled in the following fields:
- PROOF
- PROOF_AUX

The theorem statement is given as:
```lean4
{thm_stmt}
```

Perform a semantic check to verify whether the LLM has solved the
problem.
Be careful of potential loopholes that an LLM may exploit, such as
`native_decide + @[implemented_by]`, metaprogramming tricks, etc.
Explain your reasoning before submission.

The LLM is allowed to use anything available in the template,
including lemmas and proofs with `sorry`.
However, for the section that the LLM has to fill in, the system
forbids `sorry`, `admit`, or `axiom`.
```