
Do Theorem Provers Need Theorem-Proving Agents?

mini-Lean-agent: A Coding-Agent Baseline for Lean

Tomas Ortega

Princeton Language and Intelligence
Princeton University
Princeton, NJ 08540
tomas.ortega@princeton.edu

Simon Park

Princeton Language and Intelligence
Princeton University
Princeton, NJ 08540
juhyunp@princeton.edu

Ayush Khaitan

Princeton Language and Intelligence
Princeton University
Princeton, NJ 08540
ayush.khaitan@princeton.edu

Liam Fowl

Princeton Language and Intelligence
Princeton University
Princeton, NJ 08540
lf2728@princeton.edu

Alex Kontorovich

Department of Mathematics
Rutgers University
Piscataway, NJ 08854
alex.kontorovich@rutgers.edu

Sanjeev Arora

Princeton Language and Intelligence
Princeton University
Princeton, NJ 08540
arora@princeton.edu

Abstract

General-purpose coding agents use standard shell and file interactions, yet formal theorem-proving agents often rely on specialized, ad-hoc tooling. This makes it difficult to determine which components of theorem-proving agents provide benchmark gains: an improved base model, a specialized harness, a specific tool, or a combination of any of them. We introduce mini-Lean-agent, a deliberately minimal agent for Lean tasks whose only model-visible tool is bash. The model must use bash to inspect and edit the repository, search local libraries, invoke Lean, and manage its own intermediate work, while a separate trusted evaluator verifies the final artifact. We compare mini-Lean-agent against AxProverBase and Numina-Lean-Agent on the Putnam 2025 benchmark. Experiments with Gemini 3.8 Flash as the base model show that mini-Lean-agent solves 7/12 problems at \$2 per problem limit, compared with AxProverBase’s 4/12. With Astra as the base model, mini-Lean-agent solves 12/12 problems at an average cost of \$1.28 per problem. This suggests that a generic coding agent with a minimal harness can already achieve strong performance on Lean benchmarks. Our results motivate using mini-Lean-agent as a standard baseline when evaluating theorem-proving infrastructure.

1 Introduction

The past few months have seen a flurry of mathematics results solved by Large Language Models (LLMs), both in natural language and in formal language (most notably Lean [Moura and Ullrich, 2021]). In practice, mathematicians and engineers who use LLMs for Lean rely on general-purpose coding agents (such as Claude Code or Codex) rather than specialized prover-specific harnesses.

However, the AI-for-math literature proposes a variety of specialized machinery to construct agents for Lean tasks: special-purpose harnesses [Lample et al., 2022, Thakur et al., 2024, Pegden, 2026, Li et al., 2026], specialized tools [Requena et al., 2026, Hu et al., 2025], or a combination of both [Yang et al., 2023, Lin et al., 2025a,b, Chung et al., 2026, Aniva et al., 2025, Liu et al., 2026, Dekoninck et al., 2026, Tsoukalas et al., 2026, Song et al., 2025]. These works report substantial performance improvements over variations of their predecessors, but it is hard to attribute the gains to an improved base model, the harness, or the specialized tools considered in isolation.

In software engineering, this problem has been addressed by the development of general-purpose coding agents with a minimal prompt (less than 100 lines) that operate through a shell and filesystem, most notably mini-SWE-agent by Yang et al. [2024]. We then ask: **Should we use a generic coding agent for evaluating Lean-specific agent machinery?** Rather than positioning our work on the spectrum of few versus many tools, we position it as *generic computer access versus Lean-specific tools*. First, this establishes a baseline for Lean-agent research that is more comparable to the workflow of current practitioners. Second, it helps isolate the contribution of the base model from the contribution of specialized tools or harnesses.

To study this, we introduce mini-Lean-agent, a deliberately minimal baseline whose only model-visible tool is `bash`. Note that a generic coding agent with a shell can naturally replicate what specialized tooling hard-codes: searching Mathlib with `rg` or `grep`, running tests and interpreting compiler errors with `lake env lean`, creating temporary scratchpads (e.g., `Scratch.lean`), and transferring successful proofs to the target file. Because the model can modify its own environment, the output artifact is treated as untrusted: a separate, trusted evaluator validates the final proof in a clean environment using Comparator [LeanFRO, 2026] to catch build failures, undeclared axioms, or statement mutations.

Using this filesystem-native baseline, we investigate two primary research questions:

- **RQ1 (Parity and Cost):** Using the same model and inference budget, how much solve rate and cost-efficiency does mini-Lean-agent recover relative to specialized theorem-proving baselines?
- **RQ2 (Minimal-Scaffold Capability):** How far can stronger frontier models go on Lean benchmarks using only a minimal coding agent scaffold?

Our code is available at <https://github.com/TomasOrtega/mini-lean-agent>.

After mini-Lean-agent was made public, a concurrent project, h1 Moalla et al. [2026], was released. Like our work, h1 argues that minimal agent harnesses achieve strong theorem-proving performance with current models. However, h1 requires specialized tools for Lean library search, whereas mini-Lean-agent relies exclusively on `bash`.

2 System Design

To separate the contribution of the model from that of the harness, we keep the agent interface to a shell and handle proof acceptance independently.

Filesystem-native agent. mini-Lean-agent is built on mini-SWE-agent, whose only model-visible tool is `bash`. Given a file containing the target theorem and a pinned Lean environment, the model can inspect and edit files, search local libraries, invoke Lean, and manage its own intermediate work. Figure 1 contrasts this generic computer access with the Lean-specific interface of AxProverBase. For example, mini-Lean-agent can create a `Scratch.lean` file to test intermediate results before incorporating successful proofs into the target file. AxProverBase instead provides two custom designed search tools, one for a fuzzy search on vectorized embeddings of Lean statements [Gao et al., 2024] and another for a web search of proof strategies. It also separates the workflow into multiple modules, including a proposer (which writes Lean proofs), a compiler (which compiles the Lean proof), and a memory module (where the compiler logs the previous errors). The main distinction is the kind of operations the model can perform and compose during proof development, not just the number of tools it has available.

Trusted evaluation. Because the agent can edit its workspace, we treat its output file as untrusted. After each run, a separate evaluator copies only the declared submission artifact into a fresh

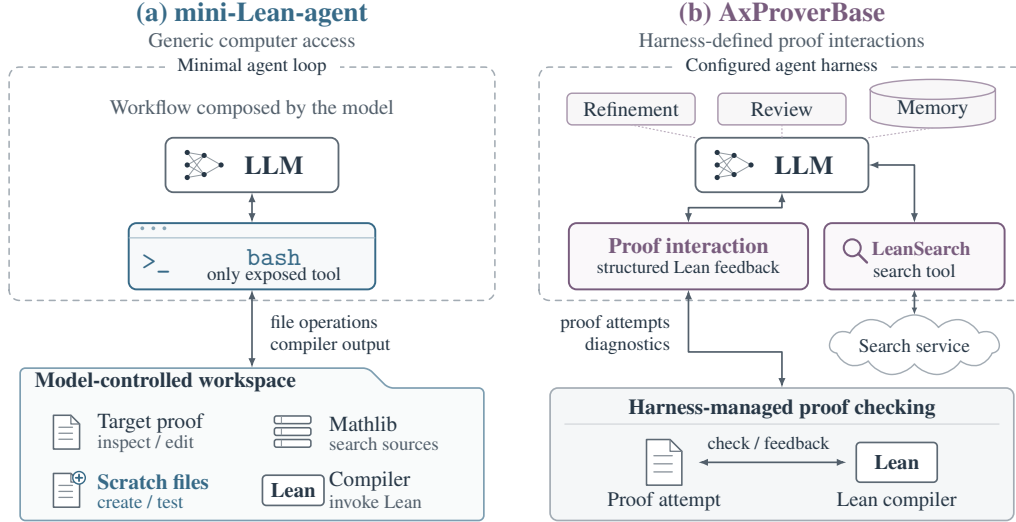


Figure 1: **A comparison between mini-Lean-agent and AxProverBase workflow.** mini-Lean-agent is only given bash tools, through which it can directly search for local Mathlib statements and edit files containing the target proof. AxProverBase instead introduces customized search tools and specialized modules in the workflow.

environment with the original theorem statement and pinned Lean and Mathlib versions. Comparator [LeanFRO, 2026] checks the candidate against the original theorem statement. This allows us to detect and reject undeclared axioms, sorry, admit, build failures, and timeouts.

3 Experimental Design

We use this procedure to evaluate mini-Lean-agent on the Putnam 2025 benchmark. For RQ1, we hold the model and budget fixed while comparing against AxProverBase, a different harness. For RQ2, we fix mini-Lean-agent and vary the model. We additionally compare against the reported results from Numina-Lean-Agent [Liu et al., 2026].

Tasks, models, and tools. We use the twelve Putnam 2025 challenge files A1–A6 and B1–B6 released with Numina-Lean-Agent, preserving the statement formalizations it attributes to Seed-Prover 1.5 [Liu et al., 2026]. We use Lean 4.28.0-rc1 and Mathlib v4.28.0-rc1 throughout. We run each problem-agent pair in its own Docker container and disable open-web access and search grounding.

Changing the harness, while keeping the model fixed. We use Gemini 3.8 Flash with high thinking as the base model for both mini-Lean-agent and AxProverBase. We report one run per problem and harness. We stop when the agent finishes or reaches one of the following limits per problem: \$2 model charge, 200 model calls, or 120 minutes of active time. We allow the final request to finish if it began below the model charge limit, and count any resulting proof as long as the call and time limits still hold. For A5 and B6, which are known to be harder problems [Liu et al., 2026], we allow up to \$50 model charge, 300 calls, and 24 active hours per problem (continuing from the model output when the limits have been reached at a lower budget).

Scaling to stronger models, while keeping harness fixed. To test how much mini-Lean-agent can achieve with stronger base models, we run it with Astra and Fable 5.1 at max reasoning effort on all twelve problems, with a maximum budget of \$50 model charge, 200 model calls (300 for A5 and B6), and 24 active hours per problem. We compare these complete systems with Numina’s reported performance and resources.

4 Results

4.1 RQ1: A minimal coding agent outperforms a minimal agent with Lean tooling

With Gemini and a \$2 limit, mini-Lean-agent and AxProverBase verify 7/12 and 4/12 runs, respectively (Table 1). The charge per successful run is \$2.09 versus \$5.24. Note that if the output is truncated at an even lower per-problem budget of \$1, AxProverBase would not have been able to solve any problem, whereas mini-Lean-agent can still solve 4.

Table 1: **Comparison between mini-Lean-agent and AxProverBase with Gemini at \$2 limit.** Each column (except last) shows the number of verified problems out of 12 when the output is truncated at an even smaller per-problem limit. The last column shows the total charge, including that for unsuccessful attempts. This does not include the charge for the higher budget run for A5/B6.

Harness	\$0.50	\$1.00	\$2.00	Charge
mini-Lean-agent	0/12	4/12	7/12	\$14.64
AxProverBase	0/12	0/12	4/12	\$20.97

For either harness, running with an extra budget on A5 and B6 does not result in a correct proof. For extended details, Tables 5 and 6 give the per-problem results and continuations in the appendix.

The bash traces show how mini-Lean-agent develops its proofs (for details, see Table 3 in the appendix). Of 1,088 requests, 602 combine writing a file with `cat` and invoking Lean, and only 73 request `rg` or `grep`. All twelve runs use scratch Lean files. In A1, the model tests a finite-set lemma with `exact?` in `Scratch.lean`, then uses Lean’s suggested lemma in the accepted proof. The excerpts with the key interactions are shown in Figure 2, in the appendix.

A possible explanation for the difference in cost is that mini-Lean-agent is more efficient in its model usage. AxProverBase makes fewer model calls (724 versus 1,091), but generates 3.64 million reasoning tokens versus 0.70 million (see Table 4 in the appendix for a full breakdown). Cache reads account for 91.5% of mini-Lean-agent’s input tokens, versus 8.7% for AxProverBase. Memory and review calls account for 11.1% of AxProverBase’s charge. All 6 unsuccessful AxProverBase runs that exhaust the allowance still leave sorry in their final candidates. On B3, two of eight proof responses truncate mid-JSON, and the final candidate leaves a modular-arithmetic case unresolved. For the excerpts, we refer the reader again to Figure 2 in the appendix.

4.2 RQ2: mini-Lean-agent with a strong model is a competitive baseline

Since Gemini fails on some problems, we naturally explore whether mini-Lean-agent can solve them with a stronger model. With Astra, we verify 12/12 problems for \$15.37 in total, or \$1.28 per verified proof (Table 2). Astra solves every problem for under \$3, including A5 for \$2.69 and B6 for \$1.62, and it uses 106 model calls across the suite. With Fable 5.1, we verify 11/12 problems for \$120.29, or \$10.94 per verified proof including the unsuccessful attempt. Fable leaves only A5 unsolved, reaching \$50.10 after 35 calls and 40.16 active minutes while developing an incomplete proof. Tables 7 and 8 give the per-problem results and cost checkpoints.

Table 2: **Performance of mini-Lean-agent with Astra and Fable at \$50 limit.** The total charge includes the charge for unsuccessful attempts.

Model	Verified	Charge	Calls	Minutes
Astra	12/12	\$15.37	106	51.3
Fable 5.1	11/12	\$120.29	133	99.9

The 12/12 result using mini-Lean-agent and Astra is on par with the result of Numina-Lean-Agent [Liu et al., 2026], the first open-source system (to the best of our knowledge) that achieved 12/12 on Putnam 2025. While the exact accounting of the cost is not offered, the authors report that Numina-Lean-Agent uses an approximate budget of \$50 per task [Liu et al., 2026, Sec. 3.1]. On the two challenging problems, they report using \$1,000 and 2,040 minutes for A5, and \$300 and 797 minutes

for B6. Instead, our mini-Lean-agent with Astra is able to solve all 12 problems with only \$15.37 in total ¹.

It is worth noting that we cannot rule out dataset contamination, since the versions of Astra, Fable, and Gemini that we use may have been trained on the answers of Putnam 2025.

5 Conclusion

We present a minimal coding-agent baseline for Lean, mini-Lean-agent, whose only model-visible tool is bash. Our experiments show that mini-Lean-agent achieves better performance than AxProverBase on the Putnam 2025 benchmark at a lower cost per verified proof, and that it can solve all twelve problems with Astra at an average cost of \$1.28 per problem. This suggests that a generic coding agent with a minimal harness can serve as a strong baseline for Lean benchmarks. An important question for future work is how the value of specialized theorem-proving scaffolding changes as base models become more capable.

Acknowledgments and Disclosure of Funding

The authors acknowledge funding from the DARPA expMath program. We thank Dr. Patrick Shafto for useful discussions.

¹Note that Numina-Lean-Agent used Opus 4.5. Because the models and budgets differ, this comparison shows what the complete systems achieve but does not isolate the effect of their harnesses. See Appendix A for details on a preliminary attempt to run mini-Lean-agent with Opus 4.5.

References

- Leni Aniva, Chuyue Sun, Brando Miranda, Clark Barrett, and Sanmi Koyejo. Pantograph: A Machine-to-Machine Interaction Interface for Advanced Theorem Proving, High Level Reasoning, and Data Extraction in Lean 4. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 104–123, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-90643-5. doi: 10.1007/978-3-031-90643-5_6.
- Jui-Hui Chung, Ziyang Cai, Zihao Li, Qishuo Yin, Rohit Agarwal, Simon Park, Rodrigo Porto, Narutatsu Ri, Ziran Yang, Shange Tang, Xingyu Dang, Hongzhou Lin, Mengdi Wang, Danqi Chen, Chi Jin, Liam H. Fowl, and Sanjeev Arora. Goedel-Architect: Streamlining Formal Theorem Proving with Blueprint Generation and Refinement, June 2026. URL <http://arxiv.org/abs/2606.06468>. arXiv:2606.06468 [cs.AI].
- Jasper Dekoninck, Nikola Jovanović, Tim Gehringer, Kári Rognvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond Benchmarks: MathArena as an Evaluation Platform for Mathematics with LLMs, May 2026. URL <http://arxiv.org/abs/2605.00674>. arXiv:2605.00674 [cs.CL].
- Guoxiong Gao, Haocheng Ju, Jiedong Jiang, Zihan Qin, and Bin Dong. A semantic search engine for mathlib4. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8001–8013, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.470. URL <https://aclanthology.org/2024.findings-emnlp.470/>.
- Jiewen Hu, Thomas Zhu, and Sean Welleck. miniCTX: Neural Theorem Proving with (Long-)Contexts, March 2025. URL <http://arxiv.org/abs/2408.03350>. arXiv:2408.03350 [cs.AI].
- Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. HyperTree Proof Search for Neural Theorem Proving, May 2022. URL <http://arxiv.org/abs/2205.11491>. arXiv:2205.11491 [cs.CL].
- LeanFRO. leanprover/comparator, August 2026. URL <https://github.com/leanprover/comparator>. original-date: 2025-06-06T11:02:28Z.
- Jinzheng Li, Zeru Zhu, and Yuanjie Ren. MerLean-Prover: A Recursive Looping Harness for Lean 4 Theorem Proving, 2026. URL <https://arxiv.org/abs/2605.26959>. Version Number: 2.
- 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, April 2025a. URL <http://arxiv.org/abs/2502.07640>. arXiv:2502.07640 [cs.LG].
- 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, August 2025b. URL <http://arxiv.org/abs/2508.03613>. arXiv:2508.03613 [cs.LG].
- Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-Lean-Agent: An Open and General Agentic Reasoning System for Formal Mathematics, January 2026. URL <http://arxiv.org/abs/2601.14027>. arXiv:2601.14027 [cs.AI].
- Skander Moalla, Ahmad Rammal, Niket Patel, Rémi Munos, and Julia Kempe. h1: A taste of the bitter lesson in automated theorem proving, 2026. URL <https://skandermoalla.com/assets/pdf/h1-arxiv-submit-09-2026.pdf>. Preprint.
- Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, page 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5. doi: 10.1007/978-3-030-79876-5_37.

- Wesley Pegden. (Auto)formalization is supposed to be easy: Trellis process semantics for spelling out rigorous proofs, June 2026. URL <http://arxiv.org/abs/2606.09674>. arXiv:2606.09674 [cs.AI].
- Borja Requena, Austin Letson, Krystian Nowakowski, Izan Beltran-Ferreiro, and Leopoldo Sarra. A Minimal Agent for Automated Theorem Proving, May 2026. URL <http://arxiv.org/abs/2602.24273>. arXiv:2602.24273 [cs.AI].
- Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Lean Copilot: Large Language Models as Copilots for Theorem Proving in Lean. In *Proceedings of the International Conference on Neuro-symbolic Systems*, pages 144–169. PMLR, July 2025. URL <https://proceedings.mlr.press/v288/song25a.html>.
- Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An In-Context Learning Agent for Formal Theorem-Proving, August 2024. URL <http://arxiv.org/abs/2310.04353>. arXiv:2310.04353 [cs.LG].
- George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, Lei Yu, Aja Huang, Miklós Z. Horváth, Andrew Ferraiuolo, Henryk Michalewski, Edward Lockhart, Codrut Grosu, Thomas Hubert, Matej Balog, Pushmeet Kohli, and Swarat Chaudhuri. Advancing Mathematics Research with AI-Driven Formal Proof Search, June 2026. URL <http://arxiv.org/abs/2605.22763>. arXiv:2605.22763 [cs.AI].
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering, November 2024. URL <http://arxiv.org/abs/2405.15793>. arXiv:2405.15793 [cs.SE].
- Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models, October 2023. URL <http://arxiv.org/abs/2306.15626>. arXiv:2306.15626 [cs.LG].

A Supplementary Results

Mini-Lean-agent with Opus 4.5. We additionally run mini-Lean-agent once with Opus 4.5 on each problem of the Putnam 2025 benchmark. We run with a \$5 per-problem budget, 120-minute time limit, and 200 call limits (300 for A5/B6). The per-problem budget is slightly higher than we allow for Gemini 3.8 Flash, but is 10x smaller than the estimated per-problem cost of Numina-Lean-Agent. Unfortunately, we are unable to solve any of the 12 problems. Due to limited resources, and since we already obtain a strong result from more powerful models (Astra, Fable), we do not further pursue this direction. Note that a head-to-head comparison against Numina-Lean-Agent is not the goal of the study; instead, we showcase that mini-Lean-agent can show strong performance, when equipped with a powerful base model.

Quantitative and qualitative analysis of model outputs. Tables 3 and 4 provide more details about the tool use and model usage in the \$2 budget runs for Gemini, excluding the extra runs for A5 and B6.

Figure 2 shows excerpts of model outputs in the mini-Lean-agent and the AxProverBase harnesses. On A1, mini-Lean-agent submits a scratch proof, to which Lean returns a lemma that the agent decides to use in the final proof. On B3, a problem on modular arithmetic, AxProverBase makes 40 requests that invoke LeanSearch (the fuzzy search on a vector embeddings database), produces eight proof responses, and makes seven memory updates. After failing to close JSON strings in two proof responses, it leaves a sorry block in the final proof proposal, leaving the non-coprime case open.

Table 3: **Number of tools requested through bash by mini-Lean-agent with Gemini and \$2 limit.** The first column (bash) includes all requests, including malformed ones. “Search” refers commands like `rg` and `grep`. “Lean” refers to commands like `lake env lean` or `lake build`. Some tools may be counted in more than one column: e.g., a request can write a file with `cat` and invoke Lean.

Problem	Bash	Search	cat	sed	Python	Lean	Find/list
A1	63	0	59	0	3	57	0
A2	75	4	62	1	2	61	1
A3	45	1	38	3	0	37	1
A4	79	0	53	1	22	54	1
A5	141	16	68	2	40	66	11
A6	86	5	31	0	49	29	1
B1	142	5	115	9	55	67	2
B2	181	33	132	4	0	132	7
B3	68	6	40	1	15	41	1
B4	92	1	47	1	38	46	3
B5	17	0	5	0	8	4	3
B6	99	2	24	0	54	23	13
Total	1,088	73	674	22	286	617	44

Table 4: **Model usage of mini-Lean-agent and AxProverBase with Gemini and \$2 limit.** Token counts sum across requests, including repeated context. Cache percentages use all input tokens as the denominator. Model calls include requests that return no bash action.

Quantity	mini-Lean-agent	AxProverBase
Model calls	1,091	724
Uncached input tokens (M)	6.47	8.11
Cached input tokens (M)	69.69	0.78
Visible output tokens (M)	0.52	0.32
Reasoning tokens (M)	0.70	3.64
Cached share of input	91.5%	8.7%
Mean charge per model call	\$0.0134	\$0.0290

mini-Lean-agent, A1. The model writes: “I’ll test how to get an upper bound K for a `Finset` $\mathbb{N}$ such that $t \subseteq \text{range } K$.” It submits:

```
cat <<'EOF' > Scratch.lean
import Mathlib

example (t : Finset ℕ) : ∃ K : ℕ, t ⊆ Finset.range K := by
  exact?
EOF
lake env lean Scratch.lean
```

Lean returns Try this: `[apply] exact Finset.exists_nat_subset_range t`. The accepted proof later uses this lemma.

AxProverBase, B3. After two truncated proof responses, the memory model writes: “Responses that exceed output token limits get truncated mid-string (EOF while parsing a string), leading to JSON parsing errors and failed builds.” The final proof proposal still contains:

```
have h_modeq : 2025 ^ Nat.totient (n + 2) ≡
  15 ^ Nat.totient (n + 2) [MOD n + 2] := by
  apply Nat.ModEq.of_dvd
  sorry
```

Figure 2: **Excerpts of Gemini’s model output and tool feedback with \$2 limit.** (Top) On A1, mini-Lean-agent sends a scratch-file query that contributes to a verified proof. (Bottom) On B3, AxProverBase submits an incomplete proof as its final proposal, after repeated retrievals and revisions.

More detailed results. Tables 5 and 6 provide the per-problem statistics for the Gemini runs, respectively the \$2 limit run and the higher budget continuation. Table 7 contains the per-problem statistics for the Astra and Fable. Table 8 shows the number of verified proofs at various thresholds, had the per-problem limit been lower for the Astra and Fable runs.

Table 5: **Per-problem statistics with Gemini and \$2 limit.** This provides a detailed breakdown of the aggregate results in Table 1. “mini” refers to mini-Lean-agent and “Ax” refers to AxProverBase. The total charge does not include the charge for the higher budget run for A5/B6. Note that AxProverBase spent slightly over the \$2 cap on A1, but we mark it correct because no other call was made after the cap was reached.

Problem	mini Verified	mini Charge	Ax Verified	Ax Charge
A1	✓	\$0.518454	✓	\$2.001509
A2	✓	\$0.658883	✗	\$0.141243
A3	✓	\$0.677093	✓	\$1.758844
A4	✓	\$1.064088	✗	\$2.014457
A5	✗	\$2.042195	✗	\$2.000157
A6	✓	\$1.192095	✗	\$1.949676
B1	✗	\$2.017090	✗	\$2.006616
B2	✗	\$2.012949	✗	\$2.063717
B3	✓	\$0.657564	✗	\$2.115382
B4	✓	\$1.622776	✓	\$1.105187
B5	✗	\$0.149430	✓	\$1.814421
B6	✗	\$2.023811	✗	\$2.003222

Table 6: **Per-problem statistics with Gemini and a higher limit on A5 and B6.** Each result includes the model charge for the first \$2 portion of the model output, where they continued from. If the final proof attempt was unsuccessful, the agent may stop before reaching any of the three limits.

Problem	Harness	Verified	Charge	Calls	Hours
A5	mini-Lean-agent	✗	\$10.773974	300	1.954
A5	AxProverBase	✗	\$8.485967	300	1.653
B6	mini-Lean-agent	✗	\$3.268421	137	0.816
B6	AxProverBase	✗	\$6.609995	173	1.254

Total model charges. Table 9 shows the total model charge for all reported experiments, including the twelve unsuccessful Opus 4.5 attempts.

B Reproducibility

To make the proof counts and charges reproducible, we pin the benchmark, harnesses, dependencies, evaluator, and Lean/Mathlib v4.28.0-rc1 and archive configurations, runs, usage records, and native LeanSearch queries and responses. We route Gemini with high thinking to Google AI Studio and Opus with a 16,384-token thinking budget and 32,768-token output limit to Anthropic through OpenRouter, without provider fallback. We route Astra (gpt-6-astra) and Fable (claude-fable-5.1) through OpenRouter to standard OpenAI and Anthropic, respectively, with max reasoning effort, a 128,000-token output limit, and no provider fallback. Astra’s recorded usage includes cache discounts, while Fable’s contains no cached tokens. We disable networking in proof and evaluator containers (the host mediates AxProverBase’s LeanSearch requests). The evaluator preserves the original source prefix, permits only plain theorem declarations, and rejects incomplete proofs, extra axioms, and metaprogramming. Fresh independent Comparator instances verify each target, allowing only `propext`, `Quot.sound`, and `Classical.choice`, with a 600-second limit per snapshot [LeanFRO, 2026].

Table 7: **Per-problem statistics of mini-Lean-agent with Astra and Fable 5.1 at \$50 limit.** This provides a detailed breakdown of the aggregate results in Table 2. Only Fable reaches its model charge limit on A5.

Problem	Astra				Fable 5.1			
	Verified	Charge	Calls	Minutes	Verified	Charge	Calls	Minutes
A1	✓	\$0.61	5	3.92	✓	\$5.09	13	9.81
A2	✓	\$1.58	9	6.16	✓	\$5.98	9	12.78
A3	✓	\$0.96	9	8.01	✓	\$3.16	5	10.33
A4	✓	\$0.98	7	6.83	✓	\$4.94	8	11.91
A5	✓	\$2.69	23	17.43	✗	\$50.10	35	40.16
A6	✓	\$0.79	5	5.06	✓	\$7.53	9	14.54
B1	✓	\$1.13	8	7.80	✓	\$10.32	11	19.12
B2	✓	\$2.06	12	11.35	✓	\$6.42	10	15.86
B3	✓	\$0.62	5	3.37	✓	\$1.56	6	5.33
B4	✓	\$1.03	6	6.78	✓	\$10.54	7	21.72
B5	✓	\$1.31	8	8.04	✓	\$5.01	6	12.65
B6	✓	\$1.62	9	10.27	✓	\$9.64	14	17.09

Table 8: **Per-problem statistics of mini-Lean-agent with Astra and Fable 5.1 at a lower budget limit.** Each column shows the number of verified problems out of 12 when the output is truncated at an even smaller per-problem limit.

Model	\$2	\$5	\$10	\$20	\$50
Astra	10/12	12/12	12/12	12/12	12/12
Fable 5.1	1/12	3/12	9/12	11/12	11/12

Table 9: **Total model charge.** Except for the second row, mini-Lean-agent is the harness used. Each row aggregates the model charge for exactly twelve problems. The first two rows include the higher budget continuations for A5 and B6.

Model	Charge
Gemini 3.8 Flash	\$24.612816
Gemini 3.8 Flash (AxProverBase)	\$32.067014
Opus 4.5	\$61.596995
Astra	\$15.367271
Fable 5.1	\$120.294990
Total	\$253.939087

C Prompts

The prompts specify the task and submission rules that the evaluator enforces. We give mini-Lean-agent and AxProverBase the same task instruction:

Complete the proof of the designated theorem in `Target.lean`. The registered project uses Lean 4.28.0-rc1 and Mathlib v4.28.0-rc1. Preserve the supplied statement, definitions, imports, and project configuration. Keep the supplied source prefix before the target theorem byte-identical. You may add proved helper theorems after that prefix, using plain theorem commands (not `lemma`, declaration attributes, or modifiers), and create scratch files while working. Submit `Target.lean` with no `sorry`, `admit`, additional axioms, or metaprogramming commands. The final artifact is checked independently in the original pinned environment. Use only the supplied tools; external models and open-web browsing are forbidden.

For mini-Lean-agent, we use the following system and initial user templates.

System message template.

```
1 You are a helpful Lean 4 theorem-proving assistant that can interact with a computer
  shell.
```

Initial user message template.

```
1 Please solve this Lean theorem-proving task: {{task}}
2
3 You can execute bash commands and edit files to complete the proof.
4 Your working directory is {{lean_project}}.
5 The fixed target file is {{lean_target}}.
6 The protected declaration is {{lean_declaration}}. Its statement must match the trusted
  challenge when your proof is verified.
7 Use the definitions, lemmas, and tactics available through the target's imports.
8
9 ## Recommended Workflow
10
11 Work step-by-step so that you can use Lean's feedback to improve the proof.
12
13 1. Read the task, fixed target, and its imports to understand the informal and formal
   statements
14 2. Identify relevant definitions, lemmas, and proof strategies. Search the project and
   dependencies, and use small scratch Lean files to check declaration names instead
   of guessing
15 3. Edit {{lean_target}} with a complete proof
16 4. Run `{{lean_check_command}}` after each proof attempt
17 5. Read every error or warning, revise the proof, and check it again
18 6. Once the check succeeds, submit your proof and finish your work by issuing the
   following command: `echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`.
19 Do not combine it with any other command. <important>After this command, you cannot
   continue working on this task.</important>
20
21 ## Correctness Rules
22
23 - Do not change or weaken the theorem statement, or replace it with an easier statement.
24 - Do not use `sorry`, `admit`, or `sorryAx`, introduce new axioms, or otherwise bypass
   the proof.
25 - Leave the complete checked proof in {{lean_target}}.
26 - Fix every compiler error. Treat any warning that a declaration uses `sorry` as a
   failed proof.
27
28 ## Command Execution Rules
29
30 You are operating in an environment where
31
32 1. You issue at least one command
33 2. The system executes the command(s) in a subshell
34 3. You see the result(s)
```

```

35 4. You write your next command(s)
36
37 Each response should include:
38
39 1. Reasoning text where you briefly explain your analysis and next step
40 2. At least one tool call with your command
41
42 CRITICAL REQUIREMENTS:
43
44 - Your response SHOULD include reasoning text explaining what you're doing
45 - Your response MUST include AT LEAST ONE bash tool call
46 - Directory or environment variable changes are not persistent. Every action is executed
  in a new subshell.
47 - Combine dependent operations with `&&`, or repeat any required `cd` and environment
  setup.
48 - Use non-interactive commands.
49 - Submit your proof with the stand-alone command `echo
  COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT` after the check succeeds.
50 The harness verifies the fixed target and its protected statement after the agent
  exits.
51 <important>After this command, you cannot continue working on this task.</important>
52
53 Example of a CORRECT response:
54 <example_response>
55 I need to inspect the target and its imports before choosing a proof strategy. Let me
  locate the Lean source files.
56
57 [Makes bash tool call with {"command": "rg --files -g '*.lean'"} as arguments]
58 </example_response>
59
60 <system_information>
61 {{system}} {{release}} {{version}} {{machine}}
62 </system_information>
63
64 ## Useful command examples
65
66 Replace example paths and declaration names with those relevant to your task.
67
68 ### Create a scratch Lean file:
69
70 Use the target's imports when exploring its libraries.
71
72 ```bash
73 cat <<'EOF' > Scratch.lean
74 import Lean
75 #check And.intro
76 #check And.left
77 EOF
78 ```
79
80 ### Find relevant definitions and lemmas:
81
82 ```bash
83 rg -n 'declaration_name' . --glob '*.lean'
84 ```
85
86 Dependency sources, when present, can be searched directly:
87
88 ```bash
89 rg -n 'declaration_name' .lake/packages --glob '*.lean'
90 ```
91
92 ### View file content:
93
94 ```bash
95 nl -ba path/to/Target.lean | sed -n '1,160p'
96 ```
97

```

```
98  ### Check the proof:
99
100  ```bash
101  {{lean_check_command}}
102  ```
```