What Helps Agentic Lean Provers? A Trace-Level Attribution Study

Pawan Sasanka Ammanamanchi [1]

Abstract

Agentic Lean provers now do much more than sample a proof: they compile candidates, read errors, inspect proof states, query libraries, retrieve premises, and decide whether to keep searching or submit. That makes pass rate hard to interpret. We study this attribution problem with trace-level ablations on a frozen 100-task Lean 4 benchmark. The clearest aggregate lesson is simple: compiler feedback helps: M1@8 solves 5–$8/100$ tasks across four Gemini runs, compared with $0/100$ for one-shot generation and $3/100$ for BoN@8. Residual proof-state feedback is useful but mixed. Richer tool and retrieval channels do not automatically become better provers; they mostly change the route through a problem: what the agent looks up, what it cites, which tasks it reaches, how it spends tool calls, and whether it ever commits. These traces expose a recurring failure mode: many unsuccessful runs keep probing instead of turning available evidence into a final checked proof. We argue that this is exactly what lower-rung ablations should reveal: when an agent should retrieve, search, restart, decompose, or commit.

1. Introduction

A Lean proving agent produces more than a pass/fail verdict. It compiles candidate fragments, reads compiler errors, queries lemma names, asks retrieval tools for premises, tries suggestion tactics, revises proofs, and sometimes keeps asking tools without ever submitting a final proof. These traces matter because two failures with the same aggregate outcome can call for different fixes. A missing lemma suggests retrieval. A repeated sequence of plausible but failing fragments suggests search. A long tool loop with no usable final submission suggests a commitment failure, not merely a knowledge failure. This paper studies the lower rungs of the agentic Lean-proving ladder: what changes when we add compiler feedback, proof-state information, tool access, Mathlib search, and premise retrieval one ingredient at a time?

Modern Lean agents combine strong base models with compiler feedback, proof-state access, retrieval, search, and decomposition (Ospanov et al., 2026; Varambally et al., 2025; Achim et al., 2025; Liu et al., 2026; Requena Pozo et al., 2026). These tool-bearing harnesses improve end-to-end performance, but they make attribution hard: an improvement may come from the base model, the feedback loop, the retriever, the tool interface, the search policy, or simply more test-time compute. Recent systems also use tool-bearing model harnesses for competition-level formal proving and textbook-scale formalization (Liu et al., 2026; Wang et al., 2026; Gloeckle et al., 2026).

Our study uses a frozen 100-task benchmark composed of 40 ProofNetSharp rows (Poiroux et al., 2024), 20 FATE-H rows, 20 FATE-X rows (Jiang et al., 2026), and 20 recent public contest rows from 2026 AIME/USAMO-style tasks that we formalized. We count a proof as solved when the assembled Lean file compiles and harness hygiene checks reject no remaining `sorry`, theorem drift, or known cheat patterns like submitting slightly modified proof statements. This is a compile-plus-hygiene criterion, not a claim of deeper independent proof validation.

We make three contributions.

1. **We isolate simple Lean feedback before tool APIs.** On the replicated Gemini block, one-shot generation solves $0/100$ and BoN@8 solves $3/100$, while compiler-feedback refinement at M1@8 solves between $5/100$ and $8/100$. Residual-goal feedback in M1.5 improves several matched rows and is a useful pre-tool feedback baseline, although the evidence does not support uniform dominance over M1.

2. **We show that tool access is not the same as tool use.** When Lean tools are available, the agent mostly turns compilation into a scratchpad. Lookup, suggestions, extra budget, and retrieval change the path through a problem, but in this campaign they do not yield a clean replicated pass-rate win.

3. **We use traces to identify the next bottleneck.** The

[1]Eleuther AI. Correspondence to: Pawan A <pawansasanka@gmail.com>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

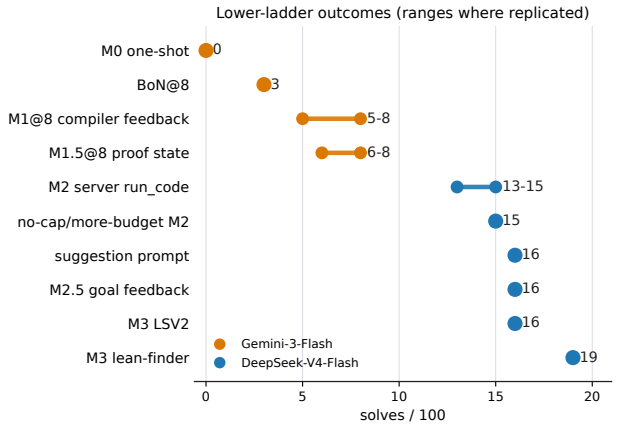

Figure 1. Lower-ladder outcomes on `standalone_mix100`. M0 and BoN are Gemini controls; M1@8 and M1.5@8 show the range across completed Gemini physical runs. Later DeepSeek-V4-Flash rows are included as single-run or small-band trace-changing configurations rather than settled aggregate comparisons.

traces show when an agent finds useful evidence, keeps probing, searches through compile calls, or fails to submit a final proof. These events are concrete targets for future search, retrieval, restart, and commit policies.

The broader implication is methodological: final Lean verdicts are not the only useful supervision signal. Traces expose whether evidence was found, used, ignored, or never converted into a submission. Future agents should therefore treat commit-vs-explore behavior as an explicit policy target.

2. Experimental Frame

Benchmark. We evaluate on a frozen 100-row Lean 4 theorem-proving benchmark assembled from four components: 40 ProofNetSharp (Azerbayev et al., 2023; Poiroux et al., 2024) rows, 20 FATE-H rows, 20 FATE-X rows (Jiang et al., 2026), and 20 recent public competition-style rows.

Capability ladder. Table 1 defines the capability ladder. M0 and BoN test generation without feedback. M1 and M1.5 add compiler-error and residual proof-state feedback. M2 introduces interactive Lean tools. M2.5 adds per-submission goal feedback and Mathlib affordances. M3 adds premise retrieval. Prompt-only suggestion/hammer instructions are reported as M2 follow-ups rather than as a separate M2.5 rung, because they change instructions without exposing a new tool or feedback channel.

Harness implementation. All runs use a lightweight Python harness around a model API and Kimina Lean Server (Dos Santos et al., 2025). The harness alternates between model responses, proof submissions or tool calls, tool execution, and JSONL trace logging. Appendix A gives wrapper details; in the main text, a "tool" means one of these harness affordances rather than an external autonomous prover.

Verifier boundary. A solve means the fixed benchmark declaration is accepted by Lean under the harness. The compiler wrapper accepts a proof only when Lean reports no errors, no remaining sorries, no harness-detected cheat pattern, and no theorem-signature drift against the benchmark task. The hygiene layer rejects patterns such as `sorry`, `admit`, user-introduced `axiom`/`constant`/`opaque`, `native_decide`, unsafe casts, and interactive placeholders such as `exact?`/`apply?` in final submitted code.

We evaluate with Gemini-3-Flash (Google, 2025) and DeepSeek-V4-Flash (DeepSeek-AI, 2026). Gemini is used for the replicated shallow-feedback block and for M2 tool-surface comparisons. DeepSeek-V4-Flash is used for most richer tool and retrieval runs because it made the larger trace campaign affordable. M2 is the main bridge where the same tool-surface structure is run across both model families.

Protocol limitations. The campaign spans several harness revisions recorded in trace metadata; the fixed object is the benchmark/hash and verifier contract, not a single repository commit. Unless otherwise stated, M2/M2.5/M3 traces use a 4×40 protocol: up to four proof submissions, with at most 40 tool calls before the harness asks for a Lean candidate. This cap creates a measurable commit point. Relaxing repeated-over-cap termination does not reveal hidden solves; it mostly produces longer scratchpad-like traces. Appendix E gives the exact cap semantics and terminal-reason counts.

3. Aggregate Ladder

We begin with a basic attribution question: is refinement useful because of feedback, or merely because it gives the model more samples? Table 2 separates these effects. M0 solves $0/100$ and BoN@8 solves $3/100$, while M1@8 solves 5–$8/100$ across four Gemini physical runs. Residual proof-state feedback is promising but mixed: it improves seed 89 at @8 and reaches $12/100$ by @16 in the seed-42B depth run, while seed 42A favors M1 at @8. We therefore treat M1.5 as a useful pre-tool baseline, not a dominance result.

The depth curve is useful mainly as a negative result about the shape of refinement. Most of the M1 gain appears early, and extending the same trajectory gives diminishing returns. This suggests that the baseline refinement loop tends to revisit a local neighborhood of proof attempts rather than opening a qualitatively new search process.

rung	ingredient added	main attribution question
M0	one-shot whole-proof generation	How often does the model solve without feedback?
BoN	independent samples, no feedback	Does sampling alone explain feedback gains?
M1	compiler-error refinement	How much does iterative Lean feedback buy over BoN?
M1.5	residual proof-state payload after failed compile	Does explicit residual-goal grounding improve refinement?
M2	Lean tool channel	Does an interactive Lean surface change behavior beyond direct compile feedback?
M2.5	submission-local feedback and Mathlib affordances	Do failure-state payloads or explicit lookup change solve paths?
M3	premise retrieval	Does retrieval change pass rate, solve set, premise use, or tool efficiency?

Table 1. Capability ladder used in this work. Each rung adds one lower-level ingredient and is read both in aggregate and through traces.

family	run	seed	@1 / M0	@4	@8	@16 / @32
M0 / BoN	one-shot	42	0	–	–	–
M0 / BoN	BoN	42	–	2	3	–
M1	seed 42A	42	–	4	8	–
M1	seed 42B depth	42	–	5	5	6 / 7
M1	seed 800	800	–	3	7	–
M1	seed 89	89	–	5	6	10 / –
M1.5	seed 42A	42	–	6	6	–
M1.5	seed 42B depth	42	–	4	7	12 / 12
M1.5	seed 800	800	–	6	7	–
M1.5	seed 89	89	–	7	8	9 / –

Table 2. Gemini-3-Flash pass counts on `standalone_mix100`. The seed-42A and seed-42B rows are independent physical re-rolls with the same nominal seed. M1 establishes the feedback-over-sampling lift; M1.5 is retained as an early proof-state-feedback baseline because it improves several matched rows, including seed 89 at @8 and the seed-42B depth run by @16, while all physical runs remain visible.

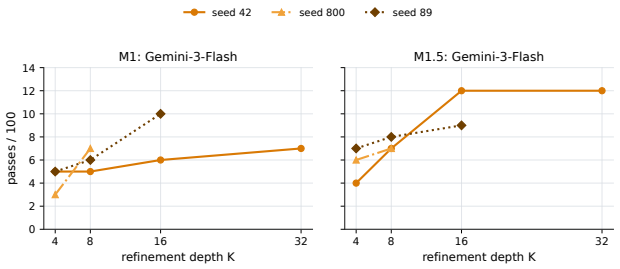

Figure 2. Pass count versus refinement depth for Gemini-3-Flash. Compiler-feedback refinement separates from same-K independent sampling. M1.5 improves several matched rows and reaches $12/100$ by @16, but the @8 rows are mixed.

4. M2: Tool Access Mostly Becomes Compile Probing

M2 asks whether an LSP/MCP-style Lean surface makes the model behave like an interactive prover. We ablate six wrappers: diagnostics, goal, term-goal, hover, completions, and whole-file run-code. The answer is mostly no. When whole-file compilation is available, the model uses it as its primary information channel; the remaining LSP tools matter less than the API shape might suggest. Thus M2 is best interpreted as an agentic delivery mechanism for compiler feedback, not as evidence that arbitrary tool access helps.

The cross-model pattern in Table 3 matters more than the exact single-seed ordering: surfaces with whole-file compilation dominate position-only surfaces. Gemini is also not uniformly stronger than DSv4 in these clean M2 cells, despite higher per-task cost (Appendix D).

The tool distribution makes the mechanism explicit: with `lean_run_code` available, calls collapse to whole-file compilation; without it, the model substitutes completions and local queries, which are weaker. The relevant abstraction is not "more LSP access" but "cheap, direct compiler feedback plus a policy for deciding what to do with it." Appendix Figure 4 gives the corresponding DeepSeek-V4-Flash tool-use chart; the same coarse pattern appears there.

4.1. Blocking and Redirection

The compile channel also becomes a crude search interface. The model frequently embeds reflection commands inside candidate files, effectively using `lean_run_code` as a slow library browser. This behavior is not just cosmetic: it

configuration	tools	Gemini	DSv4
Run-code only (LSP)	LSP `run_code` only	11	15
Run-code only (server)	HTTP `run_code`, no LSP	11	13 to 15
Full surface without diagnostics	five tools, no diagnostics	11	11 to 12
Full LSP surface	all six canonical LSP tools	9	11
Run-code + diagnostics	`run_code` + diagnostics	8	11
Diagnostics only (LSP)	diagnostics only	9	10
Diagnostics only (server)	HTTP diagnostics only	5	9
LSP surface without run-code	five tools, no `run_code`	4	8

Table 3. M2 tool-surface pass counts on 100 problems. The rows ablate the six canonical LSP-style wrappers rather than treating tool access as one binary switch; the key pattern is that surfaces with whole-file `run_code` dominate position-only surfaces.

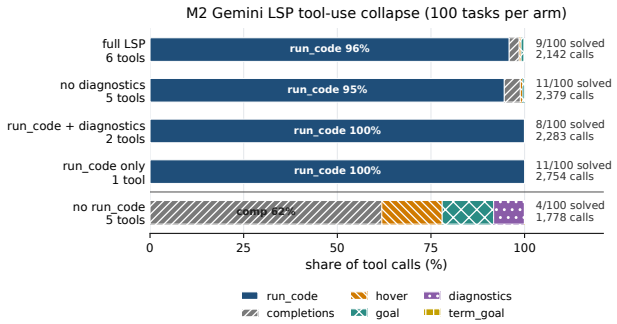

Figure 3. Tool-use summary in the M2 Gemini LSP configurations. Stacked components show each tool's share of calls. When `lean_run_code` is available, it dominates the call budget; when removed, the model substitutes completions, hover, goal, and diagnostics.

intervention	solves	read
server baseline seed 42	13/100	reference
server baseline seed 43	15/100	seed band
server baseline seed 44	15/100	seed band
no-find prompt + reject	14/100	+1, in band
no-check prompt	12/100	−1, in band
no-check prompt + reject	12/100	−1, in band
high-token slice	4/19	no new solves
no-cap/more-budget stress	15/100	in band
suggestion-tactic prompt	16/100	upper end

Table 4. M2 follow-up interventions on DSv4-Flash server-mode `run_code`. Blocking misuse and opening the repeated-over-cap termination rule do not robustly lift pass rate; redirection through a suggestion-tactic prompt is more interesting mechanistically.

reveals that the agent is missing a controlled way to search for relevant facts.

The follow-up interventions in Table 4 separate two possibilities. If the main problem were simply visible misuse, then blocking reflection probes should improve the aggregate. It does not. If the main problem were only the repeated-over-cap termination rule, then a no-cap/more-budget stress run should reveal hidden solves. It does not. The more useful intervention is redirection: suggestion-tactic prompting changes how evidence is discovered, even though its aggregate remains in the same broad range. Here "suggestion" means prompting the model to use Lean aids such as `exact?` and `apply?` during exploration. The harness rejects unresolved suggestion placeholders in final code, so the question is whether suggestion output helps the agent discover a checked proof. This illustrates a recurring attribution issue: evidence that helps search need not appear in the final artifact. We keep this prompt-only intervention in the M2 follow-up group; when combined with Mathlib lookup/search or per-submission goal feedback, it is an M2.5-plus-prompt configuration.

The no-cap/more-budget stress test changes commitment pressure, not mathematical information. Giving trajectories more freedom to keep probing leaves solve count inside the server-baseline band and makes failed traces longer and less interpretable. The capped protocol is therefore useful because it creates a measurable commit point.

5. M2.5 and M3: Information Channels Change Paths

M2 exposes two failures: whole-file compilation becomes slow library search, and failed submissions often lack local repair information. M2.5 tests direct fixes: Mathlib lookup/search, `#check`/`#find` interception, and per-submission goal feedback. None establishes a replicated pass-rate improvement, but each changes trace behavior.

5.1. M2.5: submission feedback, lookup, and interception

Table 5 should be read as behavior evidence. Per-submission goal feedback is the strongest single-seed aggregate in this group in this group, but not a new replicated rung. More informative is the trace pattern: when offered both name lookup and pattern search, the model strongly prefers pattern search, suggesting that earlier `#check`/`#find` spam was a workaround for missing search affordances.

Reflection interception reinforces the point. It does not

configuration	seed	solves	notable trace signal
Per-submission goal feedback	42	16/100	reaches two contest rows
Mathlib tools	42	13/100	tools used in 97 trajectories; `find`/`check` = 4105/769
Mathlib + suggestion prompt	42	9/100	prompt/tool stack; 71/71 model failures are cap-loop no-commit
Reflection interception	42	12/100	16,040 intercepts; reaches II-P7
Tools plus interception	42	12/100	11,461 intercepts; tools used in 95 trajectories

Table 5. M2.5 goal-feedback and tool-affordance configurations on DSv4-Flash.

configuration	seed	solves	contest rows
random control	42	12/100	none
LeanSearch-v2	89	16/100	none
BM25	89	12/100	II-P5
LeanHammer premise selection	89	10/100	none
ConcatMax	42	14/100	I-P1, II-P1
lean-finder	89	19/100	I-P1, II-P1
lean-finder + suggestion	42	18/100	I-P1, II-P5

Table 6. Completed M3 retrieval conditions on DSv4-Flash. Rows are single-seed and should be read primarily as trace-mechanism evidence rather than as settled aggregate comparisons.

cleanly improve totals, but it changes the difficult-tail solve set; similar aggregate counts can hide different mathematical coverage.

5.2. M3: premise retrieval

M3 makes the aggregate/mechanism distinction especially clear. Table 6 compares random premises, LeanSearch-v2, BM25, LeanHammer premise selection, lean-finder, a ConcatMax ensemble, and lean-finder plus suggestion prompting (Gao et al., 2026; Lu et al., 2026; Zhu et al., 2026). ConcatMax is the simple LSv2/lean-finder ensemble: for a requested top-K list, it takes the top $K/2$ results from each retriever, concatenates them, deduplicates by declaration name, and returns the merged list. Lean-finder gives the strongest single-run aggregate at $19/100$, but the main result is not a retriever ranking. Retrieval changes evidence streams and solve sets: the prompted lean-finder stack solves three tasks missed by standalone lean-finder while losing four others. A retrieved-premise-use trajectory is shown in Appendix B.

The solve-set audit gives the coverage view. Across selected completed DSv4 configurations, the union contains 24 solved tasks, including 5 recent public-contest rows. Only one selected condition has a unique solve in this audit: reflection interception reaches `AIME2026II|P7`. We treat these as coverage signals, not uncontaminated contest-performance evidence.

Retrieval should also be evaluated by whether returned premises enter final proofs and how many calls are spent before commitment. Appendix B reports retrieval-use and strict-use audits, separating usable evidence from low-confidence probing.

The LeanHammer premise-selection row is a warning about API shape. The model makes many retrieval calls and lands below the random control, suggesting that payload design and interaction pattern can matter as much as premise quality.

The strongest trace-level policy signal is failure to commit: in capped DSv4 server-style runs, every listed model failure is the repeated over-cap no-proof pattern.

6. Discussion: Traces as Supervision

The lower ladder separates three things that aggregate pass counts usually conflate: capability, interface, and policy. Compiler-feedback refinement improves the base trajectory. Tool and retrieval interfaces change what evidence becomes visible. Commitment policy determines whether that evidence becomes a final proof. This is why trace changes can be useful even without replicated pass-rate gains.

Agentic Lean failures are tool-specific. A base model may know Lean syntax without knowing how to manage a compiler loop, interpret suggestion output, choose between name lookup and pattern search, or stop probing and submit. Future tool and retriever evaluations should therefore report calls per solve, strict retrieved-premise use, compile-probe-to-submit transitions, and terminal reasons.

The commit-vs-explore pattern is the most actionable design signal. The role exploring candidates should not always be the role deciding when a proof is ready. A natural next rung is a planner/executor/committer split: the planner selects decomposition and evidence sources, the executor writes Lean, and a separate committer decides whether to submit, continue, restart, or ask for different evidence.

The solve-set audit reinforces the same point: similar aggregate counts can hide different mathematical coverage.

7. Related Work

End-to-end agentic Lean systems. APOLLO, Hilbert, Aristotle, Numina-Lean-Agent, and recent minimal-agent designs typically report end-to-end numbers from bundles that combine a base model with several of: compile feedback, proof-state access, premise retrieval, tactic search, and decomposition (Ospanov et al., 2026; Varambally et al., 2025; Achim et al., 2025; Liu et al., 2026; Requena Pozo et al., 2026). These systems answer an important engineering question: how many theorems can a complete prover

configuration	solved	model fail.	max iter.	error	budget	cap-loop / model fail.
M2 server seed 42	13	61	13	10	3	61/61
M2 no-cap/more-budget	15	0	62	14	9	–
M2 server-only M8/K40	15	66	6	8	5	66/66
M2.5 Mathlib + suggestion	9	71	10	5	5	71/71
M2.5 goal feedback M4/K40	16	60	10	9	5	60/60
M2.5 goal feedback M8/K40	14	63	5	10	8	63/63
M3 lean-finder + suggestion	18	58	14	5	5	58/58

Table 7. Commit-pressure diagnostics for capped DSv4 server-style traces. The no-cap/more-budget stress test opens repeated-over-cap `model_failure` termination, but does not remove the 40-tool boundary or forced-candidate compile behavior. In capped rows, every listed `model_failure` is the repeated-over-cap no-proof pattern.

solve? Our question is complementary: which ingredient changes the trace, and how?

Retrieval-augmented provers. LeanDojo, ReProver, and REAL-Prover established retrieval as a standard component of agentic Lean systems (Yang et al., 2023; Shen et al., 2025). DeepSeek-Prover, Goedel-Prover, BFS-Prover, HyperTree Proof Search, and Seed-Prover use synthetic data, retrieval, search, or decomposition to push formal proving performance on established benchmarks (Xin et al., 2024a;b; Ren et al., 2025; Lin et al., 2025a;b; Xin et al., 2025; Lample et al., 2022; Chen et al., 2025b;a). Our M3 configurations use LeanSearch-v2, lean-finder, LeanHammer premise selection, BM25, and a random control (Gao et al., 2026; Lu et al., 2026; Zhu et al., 2026). The random control separates "a retrieval tool is present" from "retrieval quality matters," and the trace audit asks whether retrieved premises actually enter final proofs.

8. Limitations and Reproducibility

The main limitations are the small benchmark, the public-contest component, model and harness changes across the campaign, and the compile-plus-hygiene verifier boundary. All headline numbers are Lean compile plus harness-hygiene results; we do not claim a separate independent replay verifier. Agentic proving experiments are also prone to seed and harness-path variance. We establish this variance most clearly for the Gemini feedback block, while uniformly replicating the larger tool and retrieval campaign would exceed the scope of this paper.

Appendix sections give the operational details needed to audit the claims: harness/tool definitions, a worked retrieval trace, solve-set diversity, cost accounting, and tool-cap semantics. A release should include the benchmark manifest and hash, raw JSONL traces or anonymized trace summaries, exact prompts, model identifiers and API routes, Lean/Mathlib/Kimina versions, harness source code, verifier and hygiene rules, harness configuration including caps and timeouts, exact CLI invocations or run manifests, and retrieval indexes and formatter versions. It should also include cost-accounting scripts and scripts for regenerating the main figures and selected tables. The runs reported here use Lean 4.28 and Mathlib 4.28, with the Kimina Lean Server deployment recorded in the released run metadata.

9. Future work

The next experiments should test externally managed multi-attempt search, full hammer integrations, verifier-driven candidate ranking, and planner/executor/committer architectures as separate interventions. Each should be evaluated by pass count and trace metrics: candidate count, verified-candidate count, duplicate rate, retrieval-to-use rate, terminal reasons, and spend. The LeanHammer condition here is premise selection only, not a full hammer tactic/client evaluation.

10. Conclusion

This paper studies the lower ladder of agentic Lean proving through both pass counts and traces. Compiler-feedback refinement is the clearest replicated lower-rung aggregate lever. Richer proof-state, tool, lookup, suggestion, and retrieval channels often matter first by changing trajectories: which tasks are reached, which evidence is used, how calls are spent, and whether the agent commits. The methodological lesson is that future Lean agents should not be evaluated only by final pass rate. Their traces reveal supervision targets for search, retrieval, decomposition, and commitment. For self-improving agents, the path to better proofs is not merely more tools; it is better policies for turning evidence into checked submissions.

References

Achim, T., Best, A., Bietti, A., Der, K., Fédérico, M., Gukov, S., Halpern-Leistner, D., Henningsgard, K., Kudryashov, Y., Meiburg, A., Michelsen, M., Patterson, R., Rodriguez, E., Scharff, L., Shanker, V., Sicca, V., Sowrirajan, H., Swope, A., Tamas, M., Tenev, V., Thomm, J., Williams, H., and Wu, L. Aristotle: Imo-level automated theorem proving, 2025. URL https://arxiv.org/abs/

2510.01346.

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics, 2023. URL https://arxiv.org/abs/2302.12433.

Breitner, J. Loogle: Mathlib search tool, 2026. URL https://github.com/nomeata/loogle. Accessed 2026-05-26.

Chen, J., Chen, W., Du, J., Hu, J., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Shi, W., Wang, Z., Wang, M., Wei, C., Wei, S., Xin, H., Yang, F., Gao, W., Yuan, Z., Zhan, T., Zheng, Z., Zhou, T., and Zhu, T. H. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025a. URL https://arxiv.org/abs/2512.17260.

Chen, L., Gu, J., Huang, L., Huang, W., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Ma, K., Ren, C., Shen, J., Shi, W., Sun, T., Sun, H., Wang, J., Wang, S., Wang, Z., Wei, C., Wei, S., Wu, Y., Wu, Y., Xia, Y., Xin, H., Yang, F., Ying, H., Yuan, H., Yuan, Z., Zhan, T., Zhang, C., Zhang, Y., Zhang, G., Zhao, T., Zhao, J., Zhou, Y., and Zhu, T. H. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025b. URL https://arxiv.org/abs/2507.23726.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Dos Santos, M., de Saxcé, H., Wang, H., Wang, R., Baksys, M., Unsal, M., Liu, J., Liu, Z., and Li, J. Kimina lean server: A high-performance lean server for large-scale verification, 2025.

Dressler, O. Lean LSP MCP: Tools for agentic interaction with the lean theorem prover, 3 2025. URL https://github.com/oOo0oOo/lean-lsp-mcp.

Gao, G., Sun, Z., Jiang, J., Wang, Y., Xu, J., Wu, P., Dai, B., and Dong, B. Leansearch v2: Global premise retrieval for lean 4 theorem proving. 2026.

Gloeckle, F., Rammal, A., Arnal, C., Munos, R., Cabannes, V., Synnaeve, G., and Hayat, A. Automatic textbook formalization, 2026. URL https://arxiv.org/abs/2604.03071.

Google. Gemini 3 flash preview, 2025. URL https://ai.google.dev/gemini-api/docs/models/gemini-3-flash-preview. Accessed via Google AI Studio.

Jiang, J., He, W., Wang, Y., Gao, G., Hu, Y., Wang, J., Guan, N., Wu, P., Dai, C., Xiao, L., and Dong, B. Fate: A formal benchmark series for frontier algebra of multiple difficulty levels, 2026. URL https://arxiv.org/abs/2511.02872.

Lample, G., Lachaux, M.-A., Lavril, T., Martinet, X., Hayat, A., Ebner, G., Rodriguez, A., and Lacroix, T. Hypertree proof search for neural theorem proving, 2022. URL https://arxiv.org/abs/2205.11491.

Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., and Jin, C. Goedel-prover: A frontier model for open-source automated theorem proving, 2025a. URL https://arxiv.org/abs/2502.07640.

Lin, Y., Tang, S., Lyu, B., Yang, Z., Chung, J.-H., Zhao, H., Jiang, L., Geng, Y., Ge, J., Sun, J., Wu, J., Gesi, J., Lu, X., Acuna, D., Yang, K., Lin, H., Choi, Y., Chen, D., Arora, S., and Jin, C. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025b. URL https://arxiv.org/abs/2508.03613.

Liu, J., Zhou, Z., Zhu, Z., Santos, M. D., He, W., Liu, J., Wang, R., Xie, Y., Zhao, J., Wang, Q., Zhi, L., Li, J., and Li, W. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics. *arXiv preprint arXiv:2601.14027*, 2026.

Lu, J., Emond, K., Yang, K., Chaudhuri, S., Sun, W., and Chen, W. Lean finder: Semantic search for mathlib that understands user intents, 2026. URL https://arxiv.org/abs/2510.15940.

Ospanov, A., Farnia, F., and Yousefzadeh, R. APOLLO: Automated LLM and lean collaboration for advanced formal reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=fxDCgOruk0.

Poiroux, A., Weiss, G., Kunčak, V., and Bosselut, A. Improving autoformalization using type checking, 2024. URL https://arxiv.org/abs/2406.07222.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., and Ruan, C. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Requena Pozo, B., Letson, A., Nowakowski, K., Beltran Ferreiro, I., and Sarra, L. A minimal agent for automated theorem proving. 2026. URL https://arxiv.org/abs/2602.24273.

Shen, Z., Huang, N., Yang, F., Wang, Y., Gao, G., Xu, T., Jiang, J., He, W., Yang, P., Sun, M., Ju, H., Wu, P., Dai, B., and Dong, B. Real-prover: Retrieval augmented lean prover for mathematical reasoning, 2025. URL https://arxiv.org/abs/2505.20613.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 367–381, 2020. doi: 10.1145/3372885.3373824.

Varambally, S., Voice, T., Sun, Y., Chen, Z., Yu, R., and Ye, K. Hilbert: Recursively building formal proofs with informal reasoning. *arXiv preprint arXiv:2509.22819*, 2025.

Wang, Z., Ma, W., Ming, Z., Zhang, G., Yuan, K., and Wen, Z. M2f: Automated formalization of mathematical literature at scale, 2026. URL https://arxiv.org/abs/2602.17016.

Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., and Liang, X. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data, 2024a. URL https://arxiv.org/abs/2405.14333.

Xin, H., Ren, Z. Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., Gao, W., Zhu, Q., Yang, D., Gou, Z., Wu, Z. F., Luo, F., and Ruan, C. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. 2024b. URL https://arxiv.org/abs/2408.08152.

Xin, R., Xi, C., Yang, J., Chen, F., Wu, H., Xiao, X., Sun, Y., Zheng, S., and Shen, K. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving, 2025. URL https://arxiv.org/abs/2502.03438.

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. *ArXiv*, abs/2306.15626, 2023. URL https://arxiv.org/abs/2306.15626.

Zhu, T., Clune, J., Avigad, J., Jiang, A. Q., and Welleck, S. Premise selection for a lean hammer, 2026. URL https://arxiv.org/abs/2506.07477.

A. Harness and Tool Details

The experiments use one evaluation harness rather than a collection of independent agents. The harness has four jobs: compose the prompt, parse the model response, execute the requested proof check or tool wrapper, and write a trace row with the observation and terminal verdict. If the trajectory continues, the same observation is injected into the next model turn. The important implementation details are:

- **Compilation.** All proof attempts are assembled as Lean files and sent to Kimina Lean Server. The harness records compiler messages, sorries, infotree payloads when requested, timing, and the final verdict.
- **Hygiene.** A compile pass is accepted only if the theorem signature is unchanged and the final code has no remaining `sorry`/`admit`, user-introduced axioms or constants, unsafe casts, or interactive suggestion placeholders.
- **M1/M1.5 feedback.** M1 feeds normalized compiler errors back to the model. M1.5 additionally extracts an open-goal/failure-state payload from Kimina infotree output when available and injects that text into the next refinement prompt.
- **M2 tools.** The main compile tool is `lean_run_code`. LSP-mode variants also expose diagnostics, goals, term-goals, hover, and completions. Server-mode variants keep the same agent loop but route compilation through the simpler HTTP compiler tool.
- **M2.5 tools.** `mathlib_check` is a name-lookup wrapper and `mathlib_find` is a pattern-search wrapper, backed by Mathlib declaration data and Loogle-style search when enabled (The mathlib Community, 2020; Breitner, 2026). Reflection interception catches embedded `#check`/`#find` commands inside candidate Lean files and routes them to the same lookup/search backend instead of treating them as ordinary compilation. Per-submission goal feedback injects a local failure-state payload after failed submissions when the harness can extract one.
- **M3 retrieval.** `retrieve_lemma` returns top-k premise records from the chosen backend. The M3 cells use BM25, LeanSearch-v2, lean-finder, Concat-Max (top-$K/2$ LSv2 plus top-$K/2$ lean-finder, concatenated and deduplicated), a random control, and LeanHammer premise selection (Gao et al., 2026; Lu et al., 2026; Zhu et al., 2026). MCP-backed variants call LeanSearch/Loogle/LeanFinder-style services through the harness interface (Dressler, 2025). The LeanHammer row is premise selection only; it is not a full hammer tactic or proof-search integration.

B. Worked Retrieval-to-Citation Trace

The lean-finder result in Table 6 includes a short trajectory where retrieval visibly enters the final proof. The source is

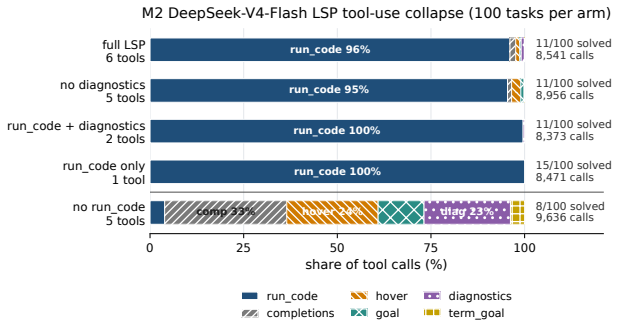

Figure 4. Tool-use summary in the M2 DeepSeek-V4-Flash LSP configurations. Stacked components show each tool's share of calls. The full-surface and no-`run_code` arms aggregate clean head/tail trace shards to cover the same 100-task benchmark.

row 1 of the corrected-format lean-finder trace from May 25, 2026, for `Herstein|exercise_2_5_43`. The first tool call queries for a group of order p^2 being abelian. The top retrieved premise is:

field	value
tool	`retrieve_lemma`
query	group of order p^2 is abelian
top premise	`IsPGroup.commGroupOfCardEqPrimeSq`
statement	`Nat.card G = p ^ 2 -> CommGroup G`
source	lean-finder dense search

The submitted proof then uses the retrieved premise directly:

```
theorem exercise_2_5_43 (G : Type*) [
    Group G] [Fintype G]
  (hG : card G = 9) :
  Nonempty (CommGroup G) := by
  have hcard : Nat.card G = 3 ^ 2 := by
    rw [Nat.card_eq_fintype_card, hG]
    norm_num
  haveI : Fact (Nat.Prime 3) := { out :=
    by decide }
  haveI : CommGroup G := IsPGroup.
    commGroupOfCardEqPrimeSq hcard
  exact Nonempty.intro inferInstance
```

This example is not used as proof that lean-finder is the best retriever. Its role is narrower: it shows the trace event the paper argues should be measured systematically, namely retrieved premise → compiled candidate → final proof using that premise.

C. Solve-Set Diversity Audit

Figure 5 aggregates the selected completed DSv4 columns used in the solve-set audit. The three M2 server seeds are collapsed into one "M2 any" column for readability; other columns correspond to the completed suggestion, no-cap/more-budget, M2.5, and M3 configurations discussed in the main text. The union contains 24 solved tasks, including 5 public-contest rows. By source family, the union contains Artin 1, Axler 2, Dummit-Foote 4, FATE-H 3, Herstein 3, Munkres 3, Rudin 3, and public-contest 5. The completed Mathlib-plus-suggestion stack solves 9 tasks, all already reached by other selected conditions. Only one selected condition has a unique solve in this audit: reflection interception uniquely solves `AIME2026II|P7`. The point is not that reflection interception is the best aggregate condition; it is that similar totals can hide different mathematical coverage.

D. Cost Accounting

We track API spend from the same trace fields used for the scientific claims. Costs are computed from each task's last cumulative `budget_used.tokens_in/out` snapshot; summing all step snapshots would double-count. The pricing table used for this accounting is the project pricing table: Gemini-3-Flash at \$0.50/M input and \$3.00/M output, and DSv4-Flash at \$0.14/M input and \$0.28/M output. The main comparison should be dollars per attempted task, because zero- and low-solve cells make dollars per solve unstable.

Active stress tests are intentionally omitted from the submitted tables until complete JSONL traces are available. This keeps the appendix as an accounting of finished evidence rather than a launch ledger.

E. Tool-Cap Semantics

The tool cap is part of the experimental protocol, not only an engineering timeout. In the capped M2-style runs, the model can use tools up to `max_tools_per_submission`; at the cap, the harness asks for a Lean proof submission rather than another tool call. Under the `retry_then_compile_candidate` policy, a recoverable over-cap candidate is compiled when one exists. After `max_cap_no_proof_events` repeated over-cap tool requests without a proof, the trajectory terminates as `model_failure` with reason `repeated_tool_call_at_cap_without_proof`.

This rule makes a real behavioral distinction measurable: whether the model can stop exploring and commit a candidate. Opening the repeated-over-cap termination moved failures into max-iteration, error, and budget-exhausted buckets and solved $15/100$, inside the server-baseline envelope. In the capped DSv4-Flash traces inspected here, model failures are overwhelmingly this over-cap no-proof pattern; for example, the per-submission goal-feedback stress tests have $60/60$ model failures in the M4 run and $63/63$ in the M8 run with this reason. We therefore use the DSv4 `model_failure` count as a trace-level commit-policy diagnostic, not as a cross-model universal failure-rate metric.

configuration	solves	rows using retrieval	retrieve calls	solves with retrieval	strict use rows
random control	12/100	94/100	2788	8/12	0
LeanSearch-v2	16/100	89/100	3170	11/16	9
BM25	12/100	93/100	3196	7/12	5
LeanHammer premise selection	10/100	89/100	2375	7/10	3
ConcatMax	14/100	88/100	2980	9/14	8
lean-finder	19/100	91/100	3100	15/19	14
lean-finder + suggestion	18/100	91/100	2837	13/18	10

Table 8. Retrieval-use audit for completed M3-style traces. Counts are mechanical trace measurements from complete JSONL files; retrieve-call counts use the final cumulative `budget_used.tool_call_counts.retrieve_lemma` counter. "Strict use" counts solved rows where a returned declaration name, or an unmistakable exact suffix, appears in the comment-stripped final proof body. These are use and pacing signals, not causal proof that a returned premise was necessary.

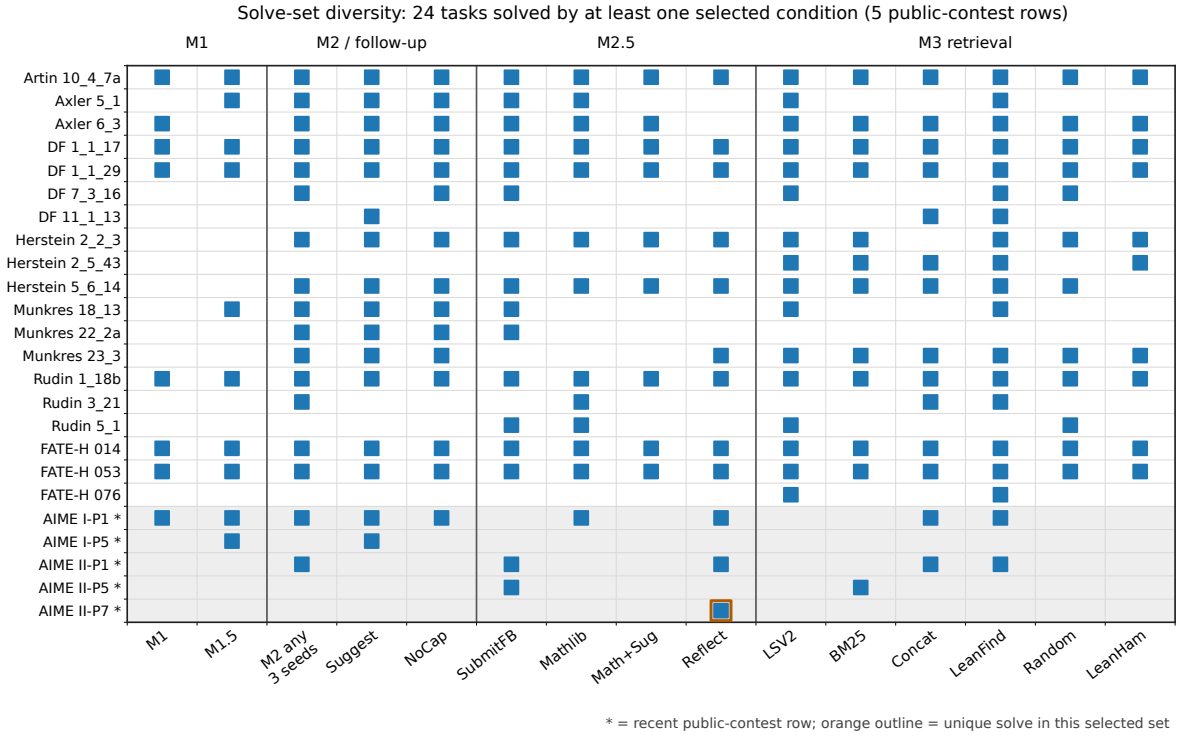

Figure 5. Solve-set matrix for selected completed DSv4 configurations. A filled cell means the configuration solved the task. Rows include only tasks solved by at least one selected configuration; an asterisk marks recent public-contest rows.

model	condition	solves	total $	$/task	$/solve
Gemini-3-Flash	M0	0/100	3.56	0.036	–
Gemini-3-Flash	BoN@4	2/100	8.63	0.086	4.32
Gemini-3-Flash	BoN@8	3/100	16.70	0.167	5.57
Gemini-3-Flash	M1@4, auxiliary native run	5/100	14.44	0.144	2.89
Gemini-3-Flash	M1@8	8/100	24.61	0.246	3.08
Gemini-3-Flash	M1@32	7/100	84.13	0.841	12.02
Gemini-3-Flash	M1.5@4, auxiliary native run	5/100	14.23	0.142	2.85
Gemini-3-Flash	M1.5@8	6/100	25.59	0.256	4.26
Gemini-3-Flash	M1.5@32	12/100	79.70	0.797	6.64
Gemini-3-Flash	M1@8, seed 800	7/100	27.06	0.271	3.87
Gemini-3-Flash	M1.5@8, seed 800	7/100	27.58	0.276	3.94
Gemini-3-Flash	M1@16, seed 89	10/100	48.70	0.487	4.87
Gemini-3-Flash	M1.5@16, seed 89	9/100	46.18	0.462	5.13
DSv4-Flash	M0	2/100	0.51	0.005	0.26
DSv4-Flash	BoN@4	3/100	2.17	0.022	0.72
DSv4-Flash	BoN@8	5/100	4.40	0.044	0.88
DSv4-Flash	M1@4	2/100	1.87	0.019	0.93
DSv4-Flash	M1@8	8/100	3.33	0.033	0.42
DSv4-Flash	M1.5@4	6/100	1.91	0.019	0.32
DSv4-Flash	M1.5@8	10/100	3.26	0.033	0.33

Table 9. Measured lower-rung API cost. The Gemini M1/M1.5 @4 auxiliary rows were paid native runs; the body table's @4 values are otherwise obtained by step-capping deeper traces and do not require additional API spend. DSv4-Flash is much cheaper per attempted task in these cells, and the available DSv4 calibration rows do not show Gemini as a uniformly stronger scratchpad-style prover.

M2 condition	Gemini solves	Gemini $	Gemini $/task	DSv4 solves	DSv4 $	DSv4 $/task
server diagnostics	5/100	15.50	0.155	9/100	9.83	0.098
LSP diagnostics only	9/100	17.78	0.178	10/100	9.72	0.097
server run-code	11/100	23.94	0.239	13/100	9.89	0.099
LSP run-code only	11/100	29.36	0.294	15/100	8.21	0.082
LSP surface without run-code	4/100	47.83	0.478	8/100	12.35	0.124
full LSP surface	9/100	49.48	0.495	11/100	8.78	0.088
full LSP surface without diagnostics	11/100	49.54	0.495	11/100	9.30	0.093
run-code plus diagnostics	8/100	51.62	0.516	11/100	8.18	0.082

Table 10. Measured M2 cost by model and tool surface, restricted to clean cells used for model-performance and spend comparisons. On these cells, Gemini spends substantially more per attempted task without a corresponding pass-rate advantage over DSv4-Flash.

group	condition	solves	total $	$/task	$/solve
M2 follow-up	server baseline, seed 42	13/100	9.88	0.099	0.76
M2 follow-up	server baseline, seed 43	15/100	10.12	0.101	0.67
M2 follow-up	server baseline, seed 44	15/100	10.54	0.105	0.70
M2 follow-up	no-find prompt plus rejection	14/100	10.01	0.100	0.72
M2 follow-up	no-check prompt	12/100	9.52	0.095	0.79
M2 follow-up	no-check prompt plus rejection	12/100	9.32	0.093	0.78
M2 follow-up	high-token slice	4/19	2.12	0.112	0.53
M2 follow-up	suggestion-tactic prompt	16/100	8.40	0.084	0.53
M2 follow-up	no-cap/more-budget stress test	15/100	13.31	0.133	0.89
M2 follow-up	server-only M8/K40	15/100	10.62	0.106	0.71
M2.5	explicit Mathlib tools	13/100	9.36	0.094	0.72
M2.5	Mathlib + suggestion prompt	9/100	10.39	0.104	1.15
M2.5	reflection interception	12/100	8.58	0.086	0.72
M2.5	tools plus interception	12/100	9.58	0.096	0.80
M2.5	per-submission goal feedback	16/100	9.92	0.099	0.62
M2.5	per-submission goal feedback, M8/K40	14/100	11.25	0.112	0.80
M3	random control	12/100	10.20	0.102	0.85
M3	LSv2, corrected formatter	16/100	12.44	0.124	0.78
M3	BM25, corrected formatter	12/100	11.45	0.115	0.95
M3	LeanHammer premise selection	10/100	10.41	0.104	1.04
M3	ConcatMax	14/100	11.97	0.120	0.86
M3	lean-finder	19/100	12.27	0.123	0.65
M3+prompt	lean-finder + suggestion, M4/K40	18/100	12.05	0.120	0.67

Table 11. Measured DSv4-Flash costs for completed M2 follow-up, M2.5, and M3 runs. Later-rung Gemini entries are absent because those experiments were not run cleanly on Gemini. The cost-per-task band is narrow for most completed DSv4 cells; the scientific question is therefore mostly attribution and solve-set diversity, not raw spend.