h1 : A Taste of the Bitter Lesson in Automated Theorem Proving

Anonymous authors
Paper under double-blind review

Abstract

Advances in automated theorem proving (ATP) have largely been driven by systems built on large language models (LLMs). Formal proof systems often wrap an LLM in a complex harness with pre-designed planning, retrieval, and multi-agent orchestration. However, it is unclear whether benchmark advances come from these inductive biases, test-time scaling, or more capable models. We build `h1`, a simple harness of generic components that allows for unbounded test-time scaling for theorem proving in the Lean proof assistant, and study how far it can go with only Lean tools, verifier feedback, and self-reflection sustaining progress. `h1` achieves state-of-the-art results and advances the cost–accuracy frontier on PutnamBench, FATE-X, and FormalConjectures100-Solved, and resolves 9 open problems in FormalConjectures. It matches or outperforms more specialized systems, including AlphaProof, Goedel-Architect, Numina-Lean-Agent, OpenGauss, Aleph, and simpler alternatives like Humanfia, AxProverBase, and Claude Code, at matched models where available. On open Erdős problems, scaling `h1`'s pass@1 can be more cost-effective than repetition within `h1` or AlphaProof Nexus (APN) basic, and surpass the full APN system. Our ablations show how tool use, prompting, self-reflection, thinking budget, and turns per iteration shape scaling efficiency.

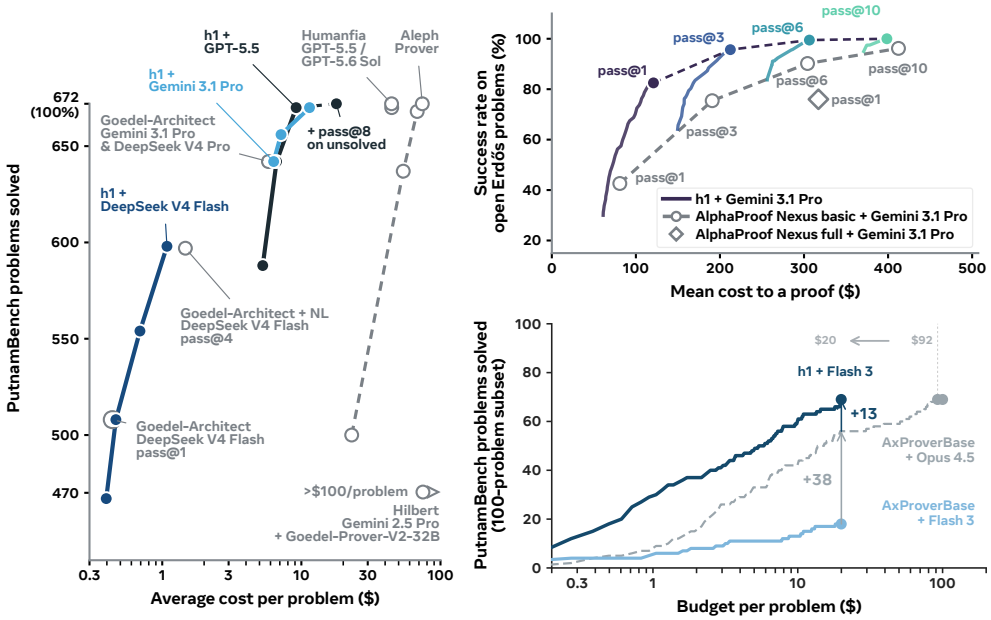

Figure 1: **(Left) Simple scaling Pareto-dominates specialized systems.** On PutnamBench (672 competition problems), `h1` with DeepSeek V4 Flash outperforms Goedel-Architect's best of four runs (pass@4) seeded with natural-language proofs. With GPT-5.5, it solves every solvable problem, as Humanfia and Aleph Prover do (Humanfia, 2026; Logical Intelligence, 2026), at lower cost. **(Top right) Longer attempts can outperform repetition.** On five open Erdős problems, a single `h1` attempt (pass@1) beats three attempts of AlphaProof Nexus's basic agent (APN basic, pass@3) at lower mean cost per success, can be cheaper than repeating `h1` itself, and Pareto-dominates the full APN system. **(Bottom right) Simple harness design is important for scaling efficiency and effectiveness.** On a 100-problem PutnamBench subset, Gemini 3 Flash trails Opus 4.5 in the AxProverBase loop but overtakes it in `h1` at matched budget through more efficient tool use and caching.

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.

1 INTRODUCTION

The bitter lesson predicts that general methods which scale with computation ultimately outperform approaches built around human domain knowledge (Sutton, 2019). However, while progress in informal mathematics has come from scaling frontier models' chain-of-thought reasoning (Guo et al., 2025; OpenAI, 2024), progress in formal proving has increasingly relied on human-designed workflows around those models. Indeed, formal proof assistants such as Lean (Moura & Ullrich, 2021) express mathematical statements and proofs in a precise language that can be checked mechanically. This includes intermediate proof steps, which naturally supports structured search that can be encoded as part of a workflow in a *harness*: the tools, context, and control loop around a model. Recent formal proving harnesses exploit this structure by orchestrating specialized provers (Varambally et al., 2026) or prescribing planning, decomposition, and proof refinement (Chung et al., 2026; Li et al., 2026a; Isenbaev & Hanin, 2026; Math Inc., 2026b). Only very recently has interest grown in *simple* loops that use the harness *only* to provide Lean feedback and sustain work (Requena et al., 2026; Tsoukalas et al., 2026; Humanfia, 2026; Kabra et al., 2026).

Formal mathematics thus raises a bitter-lesson question for harness design: with frontier models already capable of reasoning and agentic test-time scaling, can a simple test-time scaling loop match specialized workflows? And what should a simple harness provide to make that scaling effective and efficient? Do these simple methods offer better performance for the same cost? These questions matter increasingly as frontier models generate research mathematics whose informal arguments are difficult to verify. Indeed, scaling automated theorem proving (ATP) to produce mechanically checked proofs has become a necessary component of automated discovery (Tsoukalas et al., 2026; OpenAI, 2026b;a; Patel et al., 2026). Yet new workflows are often presented alongside stronger models and more compute, or compared to baselines that scale poorly. This leaves unclear whether their gains hold against an efficient simple loop given enough test-time compute.

We tackle these questions with `h1`, a simple harness with an efficient interface to Lean 4 that prescribes no proof strategy. This loop keeps a model working until success or a cost cap stops it by iterating between a multi-turn block of reasoning and tool-calling, submission to a formal validator, and a reflection that compacts the transcript into a new context for the next iteration. Each additional iteration is akin to appending "Wait" to a model's thinking in s1 (Muennighoff et al., 2025), extending the computation available at test time. We evaluate cost–success curves across benchmarks and open problems, matching models and accounting for inference cost where possible. We show that:

- **A bitter lesson for specialized harnesses: simple test-time scaling reaches frontier results without prescribed workflow** (Sections 3.1 and 3.2). A simple loop often matches or Pareto-dominates specialized systems at matched models and comparable budgets, while reaching state-of-the-art results on PutnamBench (Tsoukalas et al., 2024), FATE-X (Jiang et al., 2026), and FormalConjectures100-Solved (Firsching et al., 2026) and settling nine open Formal Conjectures problems, including the conjectured upper bound of Green 25 (Green, 2024) and variants of Erdős 138 (Bloom, 2026a) and 337 (Bloom, 2026b). At matched models, `h1` surpasses Goedel-Architect (Chung et al., 2026) at lower estimated cost; on five open Erdős problems, it Pareto-dominates full AlphaProof Nexus (APN) (Tsoukalas et al., 2026) using the same prover LLM.
- **A simple harness still needs efficient interaction, reliable feedback, and sustained progress** (Section 4). Among simple loops, harness design produces large differences in solve rates and cost. Controlled ablations and comparisons with AxProverBase (Requena et al., 2026), Claude Code (Anthropic, 2026b), and Humanfia (Humanfia, 2026) show the value of the right tools, efficient editing and cache reuse, and a loop that sustains repair and reflection.
- **Longer attempts can be more efficient than repetition** (Section 3.3). With sequential test-time scaling, pass@1 traces a budget-dependent curve that can Pareto-dominate pass@k: on five Erdős problems, extending `h1` can cost less than repeating `h1` or APN basic (Tsoukalas et al., 2026).

Our results show that a simple harness can match or outperform more complex systems, extending the bitter lesson to harness design in ATP and plausibly to other structured environments. Frontier models continue to improve their reasoning and adaptation to execution feedback through large-scale training, and harnesses should help these capabilities scale through efficient interaction with the environment, rather than impose prescribed workflows. Before designing a workflow, one should thus first try a frontier model in a simple loop at a realistic budget.

2 METHODS

We first introduce the formal-proving task, benchmarks, and baselines, and explain how we evaluate test-time scaling. We then describe `h1`'s tools and interaction loop, which let the model work efficiently and sustain progress as its inference budget grows.

2.1 EXPERIMENTAL SETUP

Formal proving. In Lean (Moura & Ullrich, 2021), a formal statement specifies a mathematical claim and its assumptions; the agent must complete its proof in a source file. Mathlib (The mathlib Community, 2020) is a shared library of formalized mathematics that provides definitions and previously proved results. Compiling the file checks the proof and returns errors. During proof development, *open goals* list what remains to be proved. Final validation checks the proof against the original statement, rejecting placeholders for unfinished steps and added assumptions.

Benchmarks. We evaluate on formal statements of increasing difficulty. PutnamBench (Tsoukalas et al., 2024) contains 672 problems from the Putnam undergraduate competition; for controlled comparisons, we use Putnam100, the same 100-problem subset used by AxProverBase (Requena et al., 2026). FATE-X (Jiang et al., 2026) contains 100 algebra problems at PhD qualifying-exam level or beyond, and FormalQualBench (Math Inc., 2026a) 23 theorems at PhD qualifying-exam level. Formal Conjectures (Firsching et al., 2026) collects formalized research conjectures; we evaluate on FC100-solved, a random sample of 100 problems with known informal solutions, and on open targets. Appendix D.1 records benchmark source revisions and Lean/Mathlib versions. We omit older benchmarks such as miniF2F (Zheng et al., 2022), which modern models essentially saturate. PutnamBench mainly tests efficiency, while FATE-X, FormalQualBench, and FC100-solved remain unsaturated at our scale. Open conjectures, such as those in Formal Conjectures (Firsching et al., 2026), test progress beyond problems with known solutions.

Models and baselines. We evaluate frontier models from OpenAI, Anthropic, and Google, alongside open-weight DeepSeek V4 Flash (DeepSeek-AI et al., 2026) and Leanstral 1.5 (Kabra et al., 2026). We select baselines representing the most competitive results on our benchmarks as of August 27, 2026, alongside simple loops that help isolate the contribution of harness design. All state-of-the-art claims use this comparison cutoff (Appendix D.2). To compare with human-designed workflows, we include systems that enforce planning, decomposition, or agent roles in code: Goedel-Architect (Chung et al., 2026), MerLean (Li et al., 2026a), and full AlphaProof Nexus (Tsoukalas et al., 2026). OpenGauss (Math Inc., 2026b) instead prescribes a plan–review workflow to a general coding agent through instruction files (skills), while Numina (Liu et al., 2026b) gives a general coding agent optional tools that consult a natural-language (informal) prover and other models, and prescribes a subagent workflow for its hardest problem. Comparisons with simple loops such as Humanfia (Humanfia, 2026) and APN basic (Tsoukalas et al., 2026), AlphaProof Nexus's single-agent loop, and our same-model reruns of AxProverBase (Requena et al., 2026) and Claude Code (Anthropic, 2026b), test interaction efficiency and persistence. Trained provers' published scores add capability comparisons; Appendix B details all baselines and models, and Appendix A discusses related work.

Measuring test-time scaling with cost per task. Prior work very rarely compares systems at matched inference compute. Earlier theorem-proving work focused on training better models, with evaluation emphasizing solve rates rather than inference cost. As improvements increasingly come from inference-time search, repair, and orchestration, these reporting conventions have persisted even as an attempt has come to mean very different amounts of computation. For example, on PutnamBench, one Goedel-Architect attempt permits up to 16 revisions of its proof plan (Chung et al., 2026), while one AxProverBase attempt permits 50 agent iterations (Requena et al., 2026). Both are reported as pass@1, but matching attempt counts does not match inference budgets (Appendix D.3). Even with the same model, output tokens can cost $100\times$ as much as cached input (Figure 16). As inference-time methods drive more of the progress, accounting for their cost should become standard practice. We compare inference cost, holding the model fixed to compare harnesses or the harness fixed to measure model progress. Throughout the paper, dollar amounts for our experiments denote API-equivalent costs computed from recorded token usage and the rates in Table 10. At budget $\$B$ per problem, solve count includes problems with a verified proof within budget. Average cost includes all problems, including failures, with each problem's cost capped at $\$B$.

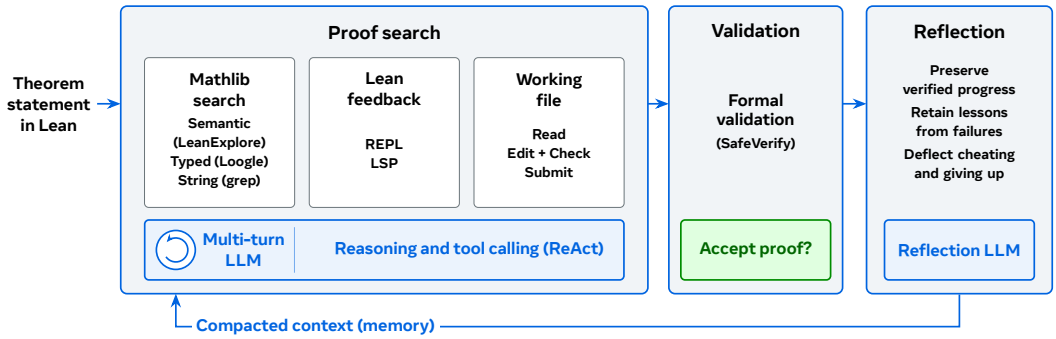

Figure 2: **`h1`: a simple harness.** An LLM agent edits a Lean file through reasoning and tool calls (ReAct; Yao et al., 2023) until it submits or reaches the T-turn limit, forcing submission. An external validator then checks the proof. On failure, a reflection LLM compacts the transcript and feedback for the next iteration. The loop provides Lean tools without prescribing a proof search structure. Search uses LeanExplore (Asher, 2025), Loogle (Breitner, 2023), and grep; proof fragments run in the Lean read–eval–print loop (REPL) (leanprover-community, 2023); the language server (LSP) reports goals and definitions; SafeVerify (GasStationManager, 2024) performs validation.

2.2 `h1`: A SIMPLE HARNESS FOR EFFICIENT TEST-TIME SCALING

`h1` keeps a model working on a Lean proof until an external validator accepts it or the budget runs out (Figure 2). In each *iteration*, the model reasons and calls tools in a ReAct loop (Yao et al., 2023) for at most T *turns*, editing a persistent working file; every edit is compiled and returns errors and open goals. The iteration ends when the model submits the file. If validation rejects it, a reflection LLM compacts the transcript and feedback into verified progress, lessons from failed attempts, and useful lemma names, and discourages giving up, as in Reflexion (Shinn et al., 2023), Self-Refine (Madaan et al., 2023), and AxProverBase (Requena et al., 2026); the model then continues from the current file with this summary. The loop builds on APN basic (Tsoukalas et al., 2026), AxProverBase, and the *Ralph* loop from software engineering (Huntley, 2025; Macedo, 2026).

A simple loop with an efficient interface. Optional tools let the model search Mathlib, inspect definitions, and test proof fragments, but it decides which to use and how to develop the proof: `h1` prescribes no decomposition, planning, or proof strategy. What the harness does provide is an efficient interface to Lean: targeted edits instead of rewritten proofs, immediate compiler feedback, and a context that grows and is cached within an iteration. Section 4 shows that these choices matter for both cost and success. The validator rejects proofs that weaken the statement or add assumptions, and accepts checked disproofs where supported (Appendix D.2); Appendix C details tools and prompts.

3 SIMPLE TEST-TIME SCALING DELIVERS FRONTIER RESULTS

We first show that frontier models in `h1` reach state-of-the-art results across competition and research mathematics. We then show that simple scaling often matches or outperforms specialized workflows at matched models and comparable inference costs, and that, even on open research problems, extending a single attempt can be more efficient than repetition.

3.1 A FRONTIER MODEL IN A SIMPLE HARNESS ACHIEVES STATE-OF-THE-ART RESULTS

State-of-the-art results on established benchmarks. With GPT-5.6 Sol, `h1` achieves state-of-the-art results: 90/100 on FATE-X, 78/100 on FC100-solved, 14/23 on FormalQualBench, and all six IMO 2026 problems (Figure 3; Table 2). On PutnamBench, `h1` with GPT-5.5 solves 670 of 672 problems. All these results use a single run per problem, without repetition or parallelism. With a simple loop that sustains work through execution feedback, verification, and reflection, reaching state-of-the-art performance with a recent frontier model required only a larger inference budget: the model works through further iterations, adjusting proofs and changing approaches without a prescribed workflow. Across our runs, individual accepted attempts reached up to 1,123 rounds of reflection, 95 active hours, or 7,124 lines of proof (Appendix H). An inexpensive open-weight model also surpasses specialized provers: DeepSeek V4 Flash in `h1` solves 43 FATE-X problems at a $14.35 per-problem cap, exceeding Leanstral 1.5's published 34 (Kabra et al., 2026). On FC100-solved, it matches the 50 solves (Firsching et al., 2026) of AlphaProof (Hubert et al., 2026) below an estimated

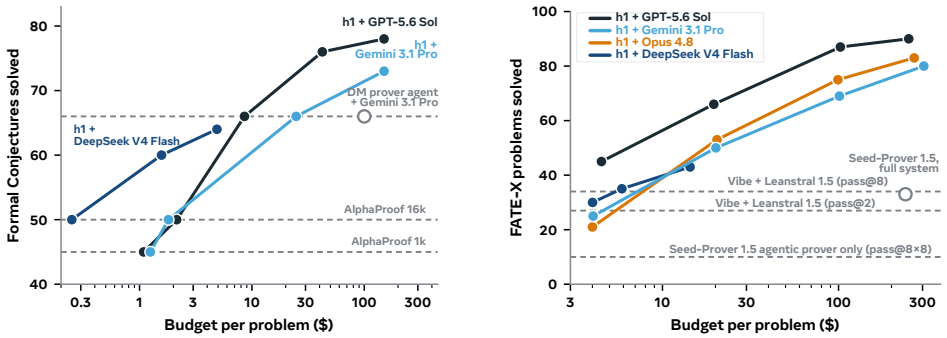

Figure 3: **Generalist models reach frontier results through simple scaling.** On FC100-solved (left) and FATE-X (right), increasing `h1`'s inference budget takes generalist models beyond published benchmark scores. On FC100-solved, `h1` with Gemini 3.1 Pro matches the DeepMind (DM) prover agent's score at a quarter of its budget cap; with GPT-5.6 Sol, it solves 78, and 90 on FATE-X. With DeepSeek V4 Flash, `h1` surpasses AlphaProof on FC100-solved and Leanstral 1.5 on FATE-X.

$0.30 cap and reaches 64 at $4.89 per problem, close to the DM prover agent's 66 (Google DeepMind, 2026). With Gemini 3.1 Pro, `h1` also surpasses the DM prover agent on FC100-solved: it matches the agent's 66 solves at a $25 rather than $100 cap per problem, then reaches 73 at $150 (Google DeepMind, 2026; Firsching et al., 2026).[1] In line with the bitter lesson, the gains frontier models draw from scaling training compute carry over to an unchanged simple loop: in `h1`, GPT-5.6 Sol matches the DM prover agent's 66 FC100-solved solves below a $9 cap, against $25 for Gemini 3.1 Pro, and reaches 90 FATE-X solves at $38 per problem, against 81 at $113 (Figure 3).

Solving open problems and exposing misformalizations. Beyond the solved subset of Formal Conjectures and benchmarks with possibly known informal solutions, we ran our system on 236 targets selected as open from Formal Conjectures (Appendix E). It settled nine of them (Table 1): variants of Erdős 138 and 337, the conjectured upper bound of Green 25, three OEIS conjectures (A108081, A211417, A022030), the existence of an AME$(9, 10)$ state (Open Quantum Problem 35), and two conjectures of Written on the Wall II. Some of them (Green 25 and the two graph theory conjectures) required repairing the statement first, so we report these as formal results about the repaired target rather than the original statement. We document concisely the newly proven theorems in Appendix E. Equally important as finding these proofs was our system's ability to find misformalizations in Formal Conjectures: formal statements that do not express the intended conjecture. Our system found several by producing short disproofs of the written statement, for instance for the Poincaré conjecture, whose formalization was missing a hypothesis requiring Hausdorff manifolds. Appendix E.2 lists all misformalizations found, some of which have since been fixed upstream.

These results establish the capability of a simple harness, and we show in the next section that simple scaling often matches specialized workflows with the same model and comparable inference cost.

3.2 The bitter lesson for specialized workflows

At matched models, simple test-time scaling recovers much of the performance of specialized workflows, often at lower inference cost, and where a gap remains, repetition (pass@k) or a newer frontier model in the same loop usually closes it (Figure 1; Table 2).

Simple test-time scaling can match specialized systems at lower cost. On PutnamBench, one extended `h1` attempt surpasses Goedel-Architect's blueprint-based search, which assigns separate agents to prove the lemmas in a dependency graph. With the same DeepSeek V4 Flash model, `h1` first matches its pass@1 score at comparable cost, then reaches 598 solves against 597 at pass@4 with informal proof seeding, at an estimated $1.08 versus $1.47 per problem (Chung et al., 2026) (Figure 1, left). Goedel-Architect's tool-integrated reasoning baseline reaches only 54.5% at pass@32 on a 200-problem subset, compared with `h1`'s 89.0% at pass@1 on the full benchmark with the same model. With Gemini 3.1 Pro alone, `h1` also matches Goedel-Architect's July leaderboard result of 642 solves, which pairs Gemini 3.1 Pro with the cheaper DeepSeek V4 Pro, at a similar cost ($6.32

[1]The DM prover agent's model is not named; we infer Gemini 3.1 Pro from its apparent connection to AlphaProof Nexus (Tsoukalas et al., 2026).

Table 1: **Nine problems marked as open in Formal Conjectures proved by `h1` with GPT-5.6 Sol.**

Problem	Statement proved
Erdős 138, variant	$W(k)/2^k \to \infty$ for the two-colour van der Waerden number
Erdős 337, variant	$\lvert(A+A)\cap[2N]\rvert/\lvert A\cap[N]\rvert \to \infty$ for thin asymptotic bases (Ruzsa–Turjányi)
Green 25, upper bound	the conjectured improvement of the upper bound (statement repaired first)
OEIS A108081	$\lvert x_n\rvert = a(n-1)$ for the inductively defined words
OEIS A211417	$a(p^k) \equiv a(p^{k-1}) \pmod{p^{3k}}$ for primes $p \geq 5$
OEIS A022030	$a(n) = 4a(n-1) - a(n-3) + a(n-4)$ for $n \geq 4$
Open Quantum Problem 35	an AME$(9, 10)$ state exists
Written on the Wall II, 100	$\alpha(G) \leq \lceil(\max_v l(v) + \frac{1}{2}\lVert \deg G^c\rVert_2)/2\rceil$ for connected G
Written on the Wall II, 314	connected triangle-free G with induced paths of at most 4 vertices is well totally dominated

versus $5.80 per problem). The much stronger performance of a simple harness with the same model illustrates how a weak simple baseline can overstate the need for specialized workflows (Figure 7).

A simple loop solves the problem that needed a hand-designed workflow. A single model in `h1` also recovers the result of Numina-Lean-Agent, which reserved a specialized workflow for the hardest problem of Putnam 2025. Numina runs Opus 4.5 in Claude Code, a general coding agent, with optional tools that call other models: an informal prover built as a Gemini 3 Pro generator–verifier loop, and a discussion partner (Liu et al., 2026b). For that problem, A5, the authors switch to a coordinator that delegates blueprint decomposition, sketching, and proving to subagents, seeded with a GPT-5.2 hint and a $1,000 budget. On the same statements, GPT-5.2 alone solves all twelve in `h1`, including A5, in a single run per problem and without such a workflow, spending $81.52 on A5 against Numina's $1,000 budget (Table 2). Also note that with Opus 4.5 alone, `h1` solves seven problems against four for Numina without its informal prover, consistent with our finding that Claude Code, on which Numina builds, scales less effectively than `h1` with the same model (Section 4).

Repetition closes most remaining gaps. Even when a single run is not enough, additional attempts can close the gap. Structured workflows also spend compute on retries and repair: MerLean retries individual statements or revises the proof plan after failures (Li et al., 2026a), while Goedel-Architect allows up to sixteen blueprint refinements within a single pass@1 attempt (Chung et al., 2026). On Putnam 2025, `h1` with Opus 4.7 solves eleven against MerLean's twelve; retrying the remaining problem recovers all twelve within six attempts. On full PutnamBench, extra attempts on the last two problems let `h1` with GPT-5.5 match Aleph Prover's (Logical Intelligence, 2026) final 672/672 coverage at an estimated $17.86 versus about $74 per problem, counting the cost of failed attempts (Appendix D.4). On FormalQualBench (Math Inc., 2026a), OpenGauss (Math Inc., 2026b) instructs Claude Code to cycle through planning, proving, review, and replanning, generating several candidate proofs per open goal and extracting helper lemmas when a goal is stuck. The best of four `h1` runs with Opus 4.6 matches its eight solves under the same four-hour limit. On these benchmarks, the only remaining exception after scaling and repetition is MerLean on FormalQualBench with Opus 4.7, which retains a one-problem advantage with or without the four-hour cutoff (Appendix F.1).

The next frontier model closes the rest. Where repetition does not close a gap, the bitter lesson does: the next frontier model in the same simple loop overtakes the specialized workflow. On FormalQualBench, replacing Opus 4.7 with its direct successor, Opus 4.8, is enough for `h1` to overtake MerLean (Li et al., 2026a): thirteen solves against ten, at lower cost per solve.

3.3 Persisting can beat repetition, even on open problems

When an attempt can keep going, pass@1 is no longer a single number but a curve that grows with budget: within one run, the model can abandon an approach and try another. The question becomes whether spending more on one attempt beats spending the same on several. AlphaProof Nexus (APN) (Tsoukalas et al., 2026) found that its basic agent, a simple loop like `h1`, can Pareto-dominate its full system, which evolves proof sketches with LLM raters, parallel provers, and AlphaProof, but only through repetition: at pass@k for large enough k. APN reports each pass@k, including pass@1, as a single point at a fixed budget, leaving open whether a single attempt given more budget can do what repetition does. We trace how each pass@k curve grows with budget, charging a group of k attempts its members' summed cost up to the first accepted proof (conditional on success, as in APN), on the five open Erdős problems that all three systems solve at least once, with Gemini 3.1 Pro.

Table 2: **Simple scaling often matches specialized systems, while model progress closes further gaps.** Row groups separate matched-model comparisons from those using other models. The 4-hour cap is FormalQualBench's official limit; $T50$/$T200$ allow 50/200 turns per iteration; `/goal` lets a small model judge completion. Table 15 adds measured times.

Benchmark	System + model	Time cap	Solved	Cost
Putnam 2025 (Numina formalization)	Numina, no informal prover, Opus 4.5	–	4/12	–
	h1 + Opus 4.5	**–**	**7/12**	$1,429 total
	Numina, full system, Opus 4.5 + Gemini 3 Pro + GPT-5.2	**–**	**12/12**	$1,800 cap
	h1 + GPT-5.2	**–**	**12/12**	$565 total
PutnamBench's 2025 slice	h1 + Opus 4.7	–	11/12	$25.97/prob.
	MerLean + Opus 4.7	**–**	**12/12**	not reported
	h1 + Opus 4.7, pass@4	**–**	**12/12**	$105.82/prob.
	h1 + Opus 4.7, best@6 (cost excludes failures)	**–**	**12/12**	$29.01/prob.
	h1 + Opus 4.8 / Gemini 3.1 Pro / GPT-5.5	**–**	**12/12**	**$7.87 / $7.12 / $6.84/prob.**
IMO 2026	AxiomProver	–	6/6	not reported
	h1 + Gemini 3.1 Pro	–	6/6	$54.37/prob.
	Humanfia + GPT-5.6 Sol	–	6/6	$21.23/prob.
	h1 + GPT-5.6 Sol ($T50$ / $T200$)	**–**	**6/6**	**$8.33 / $8.07/prob.**
FormalQualBench	Claude Code (math.inc run, Opus 4.6)	4 h	4/23	$23.16
	Claude Code (our rerun, Opus 4.6, `/goal`)	4 h	7/23	$17.28
	OpenGauss (math.inc run, Opus 4.6)	**4 h**	**8/23**	**$24.93**
	h1 + Opus 4.6 (best seed of 4)	**4 h**	**8/23**	**$23.40**
	h1 + Opus 4.7	4 h	8/23	$18.14
	MerLean + Opus 4.7	4 h	9/23	$101.09
	h1 + Opus 4.7	–	9/23	$43.92
	MerLean + Opus 4.7	**–**	**10/23**	$118.35
	h1 + GPT-5.6 Sol	4 h	11/23	$8.97
	h1 + Opus 4.8	–	13/23	$49.46
	h1 + GPT-5.6 Sol	**–**	**14/23**	**$26.38**

Figure 1 (top right) shows h1's pass@1 as a curve over budget rather than a single point. As the budget grows, it matches APN basic's pass@1 at lower cost, then exceeds both APN basic's pass@3 (75.4% at $191) and full APN (76% at $317), reaching 82.5% at $121. A single h1 attempt thus reaches what APN basic needs repetition for, without parallel provers or evolutionary search. Within h1, a longer attempt can Pareto-dominate repetition at lower budgets, but pass@1 stays below 100%: a single run can fail, so repetition is needed to solve a problem reliably; Figure 9 shows these curves for each problem, and Appendix F.6 details their construction. The right balance between depth (longer attempts) and breadth (more attempts) thus varies across problems and is hard to set in advance. The bitter lesson suggests leaving this choice to the model rather than to hand-set budgets, and our results show an early sign of it: within one h1 attempt, the model already seems to recognize when to change course, and this alone can surpass hand-designed parallel search on open problems.

4 What a harness must provide for efficient test-time scaling

Leaving the proof workflow to the model does not make the harness irrelevant: with the same model, simple loops differ several-fold in solve rate and cost. We explain these differences, and what makes h1 a state-of-the-art harness, by pairing comparisons with other simple loops with ablations on h1 that vary its tools, prompts, reflection, turns between reflections, and reasoning effort. The resulting lessons concern the interface, persistence, and context management, and should transfer beyond the automated theorem proving domain, to agents that work against a verifier in rich environments allowing structured development and retrieval, such as large libraries or codebases.

An efficient, reliable interface lets the model act in small steps and look things up. In h1, the model edits a persistent working file over many turns, sees compiler feedback after every edit, and can search the library or test a proof fragment at any point. AxProverBase (Requena et al., 2026) instead makes one round of tool calls, then regenerates a complete proof after each round of feedback. With the same Gemini 3 Flash model and a $20 budget, it solves 18 Putnam100 problems against h1's 69, and trails in all six settings we compare (Figure 4, left). A reliable interface also removes avoidable failures by construction: 42.2% of AxProverBase's failed compilations stop at an invalid library import, which cannot occur in h1 because its file already imports all of Mathlib (Figure 18). Likewise, h1 retries transient API failures and restores the file after a Lean timeout, whereas some Claude Code runs failed through API errors and repeated timeouts even after restarts (Appendix F.9). Library

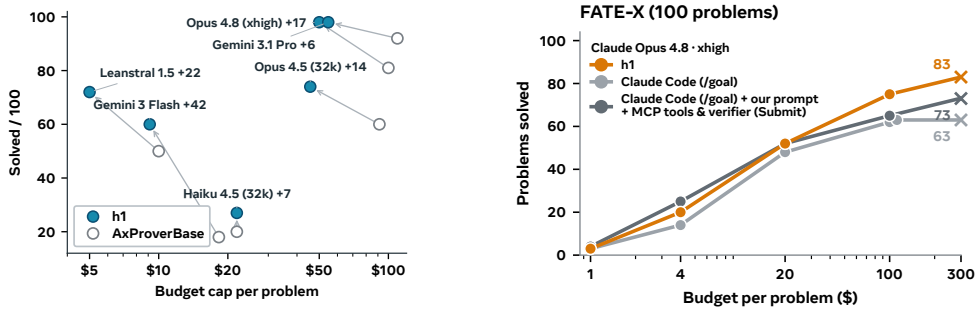

Figure 4: **(Left) `h1` Pareto-dominates AxProverBase on Putnam100. (Right) Claude Code benefits from external verification but still falls short on FATE-X.** Our prompt and tools, including validation, through the Model Context Protocol (MCP) help, but `h1` also enforces validation, repair, and reflection (Putnam100: Figure 6).

search, which AxProverBase found the least useful of its components, matters much more when the model can act on its results over many turns: without search tools, the REPL, or the language server, `h1` solves 49 Putnam100 and 49 FATE-X problems, against 69 and 63 with all tools (Figure 20). Any single search tool or the REPL recovers most of this gain, and removing any one tool from the full set costs at most four solves: what matters is the ability to look up and check, not a particular tool. Other simple loops with less efficient interfaces reach `h1`'s solve rates only at higher cost. APN basic (Tsoukalas et al., 2026) lacks dedicated search tools and needs repetition to match a single `h1` attempt on open Erdős problems (Section 3.3); Humanfia (Humanfia, 2026) uses only a shell, with a fresh session each round, and spends five times more than `h1` for the same 670 PutnamBench solves with GPT-5.5. Beyond Lean, a harness should thus offer a persistent workspace edited in small steps, immediate feedback, ways to retrieve and check what the model needs, and robust handling of predictable failures, leaving their use to the model.

The loop, not the model, decides when to stop. Test-time compute helps only if the harness turns a larger budget into continued work. With Opus 4.8, native Claude Code (Anthropic, 2026b) solves 63 FATE-X problems; giving it our prompt and tools, including a validator through the Model Context Protocol (MCP), raises this to 73, while `h1` reaches 83 (Figure 4, right; 81, 95, and 99 on Putnam100). Access to verification helps, but the model can still stop without calling the validator or give up after a rejection; in `h1`, a rejection starts another iteration with the working file, the feedback, and a reflection summary (Appendix F.9). Within `h1`, the system prompts matter most on open problems. Unlike one-line prompts, our detailed prompts tell the prover not to give up because a problem is open and the reflection LLM not to endorse such a decision, alongside general advice (Appendix C.2). On six Erdős problems, a single attempt with them outperforms six attempts with one-line prompts at the same $150 cap per attempt (Figure 5, left). Reflection, which carries verified progress and lessons into the next iteration, raises `h1` with DeepSeek V4 Flash from 554 to 598 PutnamBench solves, consistent with AxProverBase's findings (Figure 7). None of these tells the model how to prove: the harness decides whether the model keeps going, not how.

Reuse context, reset it before it degrades, and think more per turn. Each turn in a continuing context rereads it from cache, typically ten times cheaper than uncached input and a hundred times cheaper than output, whereas each restart discards it. AxProverBase sits at one extreme: output accounts for 89% of its $17.39 per problem, against 34% of `h1`'s $9.00, and it reuses at most 0.5% of its input from cache, against 74–94% for `h1` (Figure 16). Varying the number of turns between reflections in `h1` exposes the same trade-off (Figure 5, right). With one turn, `h1` solves 50 Putnam100 and 39 FATE-X problems, against 72 and 65 with ten turns, and pays roughly twice as much per solve. With 200 turns instead of one, the cached share of input rises from 0.2% to 86%, yet solves drop to 60 and 52 (Figure 27): caching makes long contexts cheap but not productive, consistent with context rot (Hong et al., 2025). Thinking more per turn is a better use of budget. AxProverBase found that raising the thinking budget from 10k to 32k tokens gains as much as doubling the number of iterations, at lower cost. In `h1`, higher reasoning effort solves more at lower total cost: with Gemini 3.1 Pro, it solves 89 Putnam100 problems at $20.63 each with low effort and 99 at $5.34 with high effort, needing far fewer compilations (Figure 26). In short, a harness for long-horizon agents should let the model work in one context long enough to reuse it, compact that context before it degrades, and favor reasoning over repeated failed attempts.

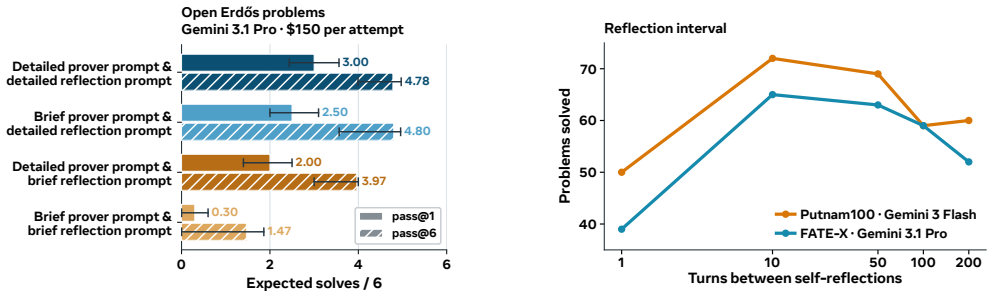

Figure 5: **(Left) Prompts that discourage giving up support persistence on open problems.** Detailed prompts at pass@1 outperform one-line prompts at pass@6 on the Erdős problems. **(Right) Moderate numbers of turns between reflections outperform both extremes.**

5 Conclusion, Limitations, and Discussion

Conclusion. We asked whether harnesses for frontier models should prescribe hand-designed workflows or give the model a simple loop and a larger budget. With `h1`, a simple loop with an efficient interface to Lean, test-time compute alone reaches state-of-the-art results and matches or surpasses hand-designed workflows at the same model and comparable cost. We find that a single longer attempt in this loop can even outperform repetition on open problems. Getting the details of the harness right still matters: its interface, persistence, and context management determine how efficiently compute turns into progress. Before designing a workflow, one should thus carefully try a frontier model in a simple loop at a realistic budget.

Limitations and discussion: the bitter lesson ahead. We acknowledge that frontier models are black boxes, and that we cannot measure the extent to which benchmark exposure during training could narrow the gap between specialized workflows and a simple loop, but where possible we used models that predate the problems or their solutions, such as Opus 4.5 and GPT-5.2 for Putnam 2025 and GPT-5.6 Sol for IMO 2026 (Appendix F.10).

Across the 707 benchmark problems on which we compare with specialized workflows, two remain exceptions: full APN solves Erdős 125, where all sixteen `h1` attempts end in Lean timeouts, and MerLean with Opus 4.7 proves `BurnsidePrimeDegree`, which `h1` with Opus 4.8 does not prove in our run (Appendix F.1). These workflows may benefit from search diversity, which we do not isolate. Rather than keeping such workflows in the inference path, one can use the traces they produce as training data, so that later models apply such strategies within a simple loop. Human design is then not eliminated but moved into training, as it already is for the model progress we credit to the bitter lesson: today's frontier LLMs acquire most of their capabilities by imitating massive amounts of human-generated data, such as human-written proofs and reasoning traces, possibly including traces of specialized workflows. The bitter lesson may eventually apply here too: models that imitate humans inherit human inductive biases and may plateau, whereas agents that learn predominantly from their own interaction with an environment could surpass them (Silver & Sutton, 2025).

Regarding scale, `h1` works on a single file, whereas formalizations that exceed a context window, such as those of Fermat's Last Theorem (Anthropic, 2026a) or entire textbooks (Rammal et al., 2026), require partial views of a library, dependency management across files, and codebase-level validation, as well as parallel agents that share intermediate results: in OpenAI's Navier–Stokes resolution, about 10,000 agents exchanged millions of messages over 88 hours (OpenAI, 2026a). Our tools also cover only Lean, not numerical computation, simulation, visualization, or web search. What makes a simple loop efficient and effective at that scale is the natural next question; following the bitter lesson, parallel agents, codebase management, and richer tools should be added without prescribing their use, for instance by letting agents communicate freely rather than through a fixed hierarchy.

`h1`'s interface and tools are themselves hand-designed, which the bitter lesson might discourage. Yet human design and leveraging computation need not run counter to each other (Sutton, 2019); the question is where each helps most first. In formal mathematics, spending compute on the reasoning of frontier models and human effort on the interface currently works best. Further ahead, a model could build and deploy its own harness for a problem, as a tool within a meta-harness, and human-designed interfaces, like specialized workflows, would be distilled into the model as data.

AI USE STATEMENT

In this work, we used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in writing proofs, propose or refine hypotheses, design or provide feedback on research methodology and experiments, implement methods, assist with translation, clean and reformat datasets, support qualitative and thematic data analysis, and interpret results. These disclosures concern AI assistance in developing the research and manuscript. We did not generate synthetic datasets using AI.

Additionally, we used generative AI tools to create or modify scientific figures or images, suggest experimental parameters, create or edit software code, create artifacts, draft parts of the paper, summarize or analyze existing literature, discover research topics or identify gaps, brainstorm, source and search for information, improve the paper's readability, identify relevant literature, format references, suggest the paper's structure, and propose a title or keywords.

We have reviewed all AI-assisted work. Generated code was reviewed and tested by the authors. The authors checked claims and citations against their sources. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have included as much detail as possible in the paper and appendices to support reproduction of both the harness and the procedures used to plot the figures. Section 2.2 and Appendix C describe the harness, including its tools, prompts, and reflection protocol. Appendix D documents formal validation, inference budgets, pricing, and cost-capped curve construction; Appendix F.6 details aggregation and uncertainty estimates; and Appendix G specifies the ablation settings. The code has not yet been released; we hope to release the harness and analysis code at the paper's final decision.

ETHICS STATEMENT

This work involves no human subjects and no private data, and all benchmarks are public. Reported costs for our experiments are API-equivalent estimates computed from token usage using the reference prices in Table 10, following the procedure in Appendix D.3; they do not disclose actual expenditure. We intend these dollar-denominated estimates only as a common basis for comparing model inference costs against our runs, not as a reference for broader economic comparisons.

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, 2025. URL https://arxiv.org/abs/2510.01346.

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 (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 104–123, Cham, 2025. Springer Nature Switzerland.

Anthropic. Formalizing Fermat's Last Theorem. Research blog post, 2026a. URL https://www.anthropic.com/research/formalizing-fermats-last-theorem.

Anthropic. Claude Code, 2026b. URL https://code.claude.com/docs.

Justin Asher. LeanExplore: A search engine for Lean 4 declarations, 2025. URL https://arxiv.org/abs/2506.11085.

Axiom Math. AxiomProver at Putnam 2025, 2025. URL https://github.com/AxiomMath/Putnam2025. Putnam 2025 results as reproduced in Liu et al. (2026b).

Axiom Math. AxiomProver at IMO 2026, 2026a. URL https://github.com/AxiomMath/IMO2026.

Axiom Math. verify_proof – AXLE (Axiom Lean Engine) documentation, 2026b. URL https://axle.axiommath.ai/v1/docs/tools/verify_proof/.

Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. Prover Agent: An agent-based framework for formal mathematical proofs, 2026. URL https://arxiv.org/abs/2506.19923.

T. F. Bloom. Erdős problem #138. https://www.erdosproblems.com/138, 2026a. Accessed: 2026-09-25.

T. F. Bloom. Erdős problem #337. https://www.erdosproblems.com/337, 2026b. Accessed: 2026-09-25.

Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M Taylor, Frank H L Koppens, and Dirk Englund. Ax-Prover: a deep reasoning agentic framework for theorem proving in mathematics and quantum physics. *Machine Learning: Science and Technology*, 7(4):045073, August 2026. doi: 10.1088/2632-2153/ae9452. URL http://dx.doi.org/10.1088/2632-2153/ae9452.

Joachim Breitner. Loogle, 2023. URL https://github.com/nomeata/loogle.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

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, 2025a. URL https://arxiv.org/abs/2512.17260.

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, 2025b. URL https://arxiv.org/abs/2507.23726.

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, 2026. URL https://arxiv.org/abs/2606.06468.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang,

Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Deicyde. fix(Millenium/Poincare): require Hausdorff in SmoothConjectureFor. Formal Conjectures pull request 5462, merged September 10, 2026. URL https://github.com/google-deepmind/formal-conjectures/pull/5462.

Yifeng Ding and Lingming Zhang. SWE-Replay: Efficient test-time scaling for software engineering agents. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=zXbem3FJR0.

Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 14114–14136. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/dong25h.html.

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

Moritz Firsching, Paul Lezeau, Salvatore Mercuri, Miklós Z. Horváth, Yaël Dillies, Calle Sönne, Eric Wieser, Fred Zhang, Thomas Hubert, Blaise Agüera y Arcas, and Pushmeet Kohli. Formal Conjectures: An open and evolving benchmark for verified discovery in mathematics, 2026. URL https://arxiv.org/abs/2605.13171.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '23, pp. 1229–1241. ACM, November 2023. doi: 10.1145/3611643.3616243. URL http://dx.doi.org/10.1145/3611643.3616243.

GasStationManager. SafeVerify, 2024. URL https://github.com/GasStationManager/SafeVerify.

Robert Joseph George, Suozhi Huang, Peiyang Song, and Anima Anandkumar. LeanProgress: Guiding search for neural theorem proving via proof progress prediction. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=eTmOwvvRu9.

Google DeepMind. DeepMind prover agent, 2026. In preparation; its solve rate on the Formal Conjectures solved subset is reported by Firsching et al. (2026).

Ben Green. 100 open problems. https://people.maths.ox.ac.uk/greenbj/papers/open-problems.pdf, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance. Technical report, Chroma, July 2025. URL https://trychroma.com/research/context-rot.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 32808–32824, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/8b4add8b0aa8749d80a34ca5d941c355-Paper-Conference.pdf.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical

reasoning with reinforcement learning. *Nature*, 651(8106):607–613, March 2026. doi: 10.1038/s41586-025-09833-y. URL https://doi.org/10.1038/s41586-025-09833-y.

Humanfia. PutnamBench solver and IMO 2026 repositories, 2026. URL https://github.com/humanfia/putnambench. See also https://github.com/humanfia/imo2026.

Geoffrey Huntley. Ralph Wiggum as a "software engineer". Blog post, July 2025. URL https://ghuntley.com/ralph/.

Vlad Isenbaev and Boris Hanin. Logical Intelligence's Aleph solves PutnamBench, January 2026. URL https://logicalintelligence.com/blog/aleph-solves-putnambench. Logical Intelligence blog.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=SMa9EAovKMC.

Jiedong Jiang, Wanyi He, Yuefeng Wang, Guoxiong Gao, Yongle Hu, Jingting Wang, Nailin Guan, Peihao Wu, Chunbo Dai, Liang Xiao, and Bin Dong. FATE: A formal benchmark series for frontier algebra of multiple difficulty levels. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 10543–10580, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/11c0318fc7b0315fa9542449b1c9ba32-Paper-Conference.pdf.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world Github issues? In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Aditi Kabra, Albert Q. Jiang, Andrew Zhao, Dhia Garbaya, Indraneel Mukherjee, Jason Rute, Mert Unsal, Roman Soletskyi, and Simon Sorg. Leanstral, 2026. URL https://github.com/mistralai/LeanstralSafeVerify/blob/main/LeanstralReport.pdf.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 22895–22907. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/kambhampati24a.html.

Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=Zy4uFzMviZ.

Joongwon Kim, Wannan Yang, Kelvin Niu, Hongming Zhang, Yun Zhu, Eryk Helenowski, Ruan Silva, Zhengxing Chen, Srini Iyer, Manzil Zaheer, Daniel Fried, Hannaneh Hajishirzi, Sanjeev Arora, Ruslan Salakhutdinov, Gabriel Synnaeve, and Anirudh Goyal. Scaling test-time compute for agentic coding. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=J9kWWkRzUM.

Adarsh Kumarappan, Mo Tiwari, Peiyang Song, Robert Joseph George, Chaowei Xiao, and Anima Anandkumar. LeanAgent: Lifelong learning for formal theorem proving. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 73525–73564, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/b67c77f8db8b991d73d6f2e16f491840-Paper-Conference.pdf.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurélien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. HyperTree Proof Search for neural theorem proving. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 26337–26349. Curran Associates, Inc., 2022. doi: 10.52202/068431-1910. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/a8901c5e85fb8e1823bbf0f755053672-Paper-Conference.pdf.

Lean FRO. Comparator, 2025. URL https://github.com/leanprover/comparator.

leanprover-community. REPL: A simple REPL for Lean 4, 2023. URL https://github.com/leanprover-community/repl. Software repository.

Austin Letson, Leopoldo Sarra, Auguste Poiroux, Oliver Dressler, Paul Lezeau, Dhyan Aranha, Frederick Pu, Aaron Hill, Miguel Corredera Hidalgo, Julian Berman, George Tsoukalas, and Lenny Taelman. SorryDB: Can AI provers complete real-world Lean theorems? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=hcO7EwDTL5.

Jinzheng Li, Zeru Zhu, and Yuanjie Ren. MerLean-Prover: A recursive looping harness for Lean 4 theorem proving, 2026a. URL https://arxiv.org/abs/2605.26959.

Xiaochuan Li, Ryan Ming, Pranav Setlur, Abhijay Paladugu, Andy Tang, Hao Kang, Shuai Shao, Rong Jin, and Chenyan Xiong. Benchmark test-time scaling of general LLM agents, 2026b. URL https://arxiv.org/abs/2602.18998.

Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-STaR: Learning to interleave thinking and proving. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 66041–66062, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a5357781c204d4412e44ed9cbcdb08d5-Paper-Conference.pdf.

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*, 2025b. URL https://openreview.net/forum?id=x2y9i2HDjD.

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 C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 11793–11818, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/13e8be77982beb73d7ed0bbf122f9f3c-Paper-Conference.pdf.

Chengwu Liu, Yichun Yin, Ye Yuan, Jiaxuan Xie, Botao Li, Siqi Li, Jianhao Shen, Yan Xu, Lifeng Shang, and Ming Zhang. Discover and prove: An open-source agentic framework for hard mode automated theorem proving in Lean 4. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 117–133, San Diego, California, United States, July 2026a. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.3. URL https://aclanthology.org/2026.acl-long.3/.

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, 2026b. URL https://arxiv.org/abs/2601.14027v1.

Logical Intelligence. Aleph prover, 2026. URL https://logicalintelligence.com/aleph-prover-final.html. PutnamBench results: 672/672 as of August 25, 2026. Earlier results: 500/660 as of November 30, 2025 (https://

logicalintelligence.com/aleph-prover.html), 637/672 as of December 22, 2025 (https://logicalintelligence.com/aleph-prover-300.html), and 668/672 as of January 6, 2026 (https://logicalintelligence.com/aleph-prover-1000.html).

David Ma, Kaijing Ma, Shawn Guo, Yunfeng Shi, Enduo Zhao, Jiajun Shi, Zhaoxiang Zhang, Gavin Cheung, Jiaheng Liu, and Zili Wang. OProver: A unified framework for agentic formal theorem proving, 2026. URL https://arxiv.org/abs/2605.17283.

Sandeco Macedo. Stop hand-holding your coding agent: Engineering the loops that replace step-by-step prompting, 2026. URL https://arxiv.org/abs/2607.00038.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Math Inc. FormalQualBench, 2026a. URL https://www.math.inc/formalqualbench.

Math Inc. OpenGauss, 2026b. URL https://github.com/math-inc/OpenGauss.

Arshia Soltani Moakhar, Iman Gholami, Max Springer, Mahdi JafariRaviz, and MohammadTaghi Hajiaghayi. Beyond the library: An agentic framework for autoformalizing research mathematics, 2026. URL https://arxiv.org/abs/2606.31134.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. doi: 10.1007/978-3-030-79876-5_37.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20275–20321, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1025. URL https://aclanthology.org/2025.emnlp-main.1025/.

OpenAI. OpenAI o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

OpenAI. On the Navier–Stokes Millennium Prize Problem. OpenAI, September 8, September 2026a. URL https://openai.com/index/navier-stokes-solution/.

OpenAI. Ten advances in mathematics and theoretical computer science. OpenAI, August 1, August 2026b. URL https://openai.com/index/ten-advances-in-mathematics/.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 41599–41633. Curran Associates, Inc., 2025. doi: 10.52202/085713-1389. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/3b77109ad4dd4ba82d07cacd4b24207e-Paper-Conference.pdf.

Niket Patel, Ahmad Rammal, Amaury Hayat, Remi Munos, and Julia Kempe. Learning to discover interesting mathematics, 2026. URL https://arxiv.org/abs/2609.28603.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. URL https://arxiv.org/abs/2009.03393.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=-P7G-8dmSh4.

Ahmad Rammal, Niket Patel, Fabian Gloeckle, Amaury Hayat, Julia Kempe, Remi Munos, Charles Arnal, and Vivien Cabannes. Formalizing mathematics at scale, 2026. URL https://arxiv.org/abs/2605.29955.

Nikil Ravi, Kexing Ying, Vasilii Nesterov, Rayan Krishnan, Elif Uskuplu, Bingyu Xia, Janitha Aswedige, and Langston Nashold. FormalProofBench: Can models write graduate level math proofs that are formally verified?, 2026. URL https://arxiv.org/abs/2603.26996.

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, 2025. URL https://arxiv.org/abs/2504.21801.

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*, 2026. URL https://openreview.net/forum?id=QGAOswfVvD.

Kári Rögnvaldsson, Chenhao Sun, Jasper Dekoninck, and Martin Vechev. Optimizing the cost-quality tradeoff of agentic theorem provers in Lean, 2026. URL https://arxiv.org/abs/2606.04883.

Marco Dos Santos, Hugues de Saxcé, Haiming Wang, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. Kimina Lean Server: A high-performance Lean server for large-scale verification, 2025. URL https://arxiv.org/abs/2504.21230.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

David Silver and Richard S. Sutton. Welcome to the era of experience. Preprint of a chapter in *Designing an Intelligence*, MIT Press, 2025. URL https://storage.googleapis.com/deepmind-media/Era-of-Experience%20/The%20Era%20of%20Experience%20Paper.pdf.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 10131–10165, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/1b623663fd9b874366f3ce019fdfdd44-Paper-Conference.pdf.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Lean Copilot: Large language models as copilots for theorem proving in Lean. In George Pappas, Pradeep Ravikumar, and Sanjit A. Seshia (eds.), *Proceedings of the International Conference on Neuro-symbolic Systems*, volume 288 of *Proceedings of Machine Learning Research*, pp. 144–169. PMLR, 28–30 May 2025. URL https://proceedings.mlr.press/v288/song25a.html.

Richard S. Sutton. The Bitter Lesson. Essay, March 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

SWE-agent Team. mini-SWE-agent, 2025. URL https://mini-swe-agent.com/.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=V7HRrxXUhN.

The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, New Orleans, LA, USA, January 2020. ACM. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

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 A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 11545–11569. Curran Associates, Inc., 2024. doi: 10.52202/079017-0368. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/1582eaf9e0cf349e1e5a6ee453100aa1-Paper-Datasets_and_Benchmarks_Track.pdf.

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, 2026. URL https://arxiv.org/abs/2605.22763.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 38998–39031, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/41b95e8ee0e054dab145a7564d9cd58a-Paper-Conference.pdf.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning, 2025. URL https://arxiv.org/abs/2504.11354.

Jianing Wang, Jianfei Zhang, Qi Guo, Linsen Guo, Rumei Li, Chao Zhang, Chong Peng, Cunguang Wang, Dengchang Zhao, Jiarong Shi, Jingang Wang, Liulin Feng, Mengxia Shen, Qi Li, Shengnan An, Shun Wang, Wei Shi, Xiangyu Xi, Xiaoyu Li, Xuezhi Cao, Yi Lu, Yunke Zhao, Zhengyu Chen, Zhimin Lin, Wei Wang, Peng Pei, and Xunliang Cai. LongCat-Flash-Prover: Advancing native formal reasoning via agentic tool-integrated reinforcement learning, 2026. URL https://arxiv.org/abs/2603.21065.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, June 2025. doi: 10.1145/3715754. URL http://dx.doi.org/10.1145/3715754.

Huajian Xin, Z.Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Haowei Zhang, Qihao Zhu, Dejian Yang, Zhibin Gou, Z.F. Wu, Fuli Luo, and Chong Ruan. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 72274–72303, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/b3b55c366d641c07180c40e4f978f311-Paper-Conference.pdf.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Kai Shen. BFS-Prover: Scalable best-first tree search for LLM-based automatic theorem proving. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32588–32599, Vienna, Austria, July 2025b. Association for Computational

Linguistics. doi: 10.18653/v1/2025.acl-long.1565. URL https://aclanthology.org/2025.acl-long.1565/.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 21573–21612. Curran Associates, Inc., 2023. doi: 10.52202/075280-0944. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/4441469427094f8873d0fecb0c4e1cee-Paper-Datasets_and_Benchmarks.pdf.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. Lean Workbook: A large-scale Lean problem set formalized from natural language math problems. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 105848–105863. Curran Associates, Inc., 2024. doi: 10.52202/079017-3357. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/bf236666a2cc5f3ae05d2e08485efc4c-Paper-Datasets_and_Benchmarks_Track.pdf.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, Yandong Wen, Ge Zhang, and Weiyang Liu. FormalMATH: Benchmarking formal mathematical reasoning of large language models, 2025. URL https://arxiv.org/abs/2505.02735.

Yuanhe Zhang, Yuekai Sun, Taiji Suzuki, Jason D. Lee, and Fanghui Liu. LeanMarathon: Toward reliable AI co-mathematicians through long-horizon Lean autoformalization, 2026. URL https://arxiv.org/abs/2606.05400.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

Tong Zheng, Haolin Liu, Chengsong Huang, Huiwen Bao, Sheng Zhang, Rui Liu, Runpeng Dai, Ruibo Chen, Chenxi Liu, Tianyi Xiong, Xidong Wu, Hongming Zhang, and Heng Huang. LLMs improving LLMs: Agentic discovery for test-time scaling, 2026. URL https://arxiv.org/abs/2605.08083.

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, Haocheng Wang, Luong Trung, Rong Ye, Phan Nhat Hoang, Huishuai Zhang, Peng Sun, and Hang Li. Solving formal math problems by decomposition and iterative reflection, 2025. URL https://arxiv.org/abs/2507.15225.

King Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, Changwang Zhang, Chenghua Lin, Jun Wang, Ge Zhang, and Wangchunshu Zhou. Scaling test-time compute for LLM agents, 2025. URL https://arxiv.org/abs/2506.12928.

APPENDIX CONTENTS

1 Introduction **2**

A Related Work **21**

B Baselines and the Structure They Add **22**

C Tools, Prompts, and Error Handling in `h1` **25**

C.1 Tools 25

C.2 Prompts for proof search and reflection 25

C.3 Lean feedback and error handling 31

D Proof Validation, Evaluation, and Cost Accounting **33**

D.1 Benchmark revisions and Lean environments 33

D.2 Formal validation 33

D.3 Inference budgets and cost curves 34

D.4 PutnamBench results and published cost conventions 35

D.5 Estimated DeepSeek API costs 36

D.6 Statistical significance and uncertainty 37

D.7 Proof screening 37

E Open-Problem Proofs and Statement Corrections **38**

E.1 Statements proved 38

E.2 Misformalizations and editable answers 38

F Additional Results for Section 3 **40**

F.1 Remaining advantages of specialized systems 40

F.2 PutnamBench: reflection and specialized provers 41

F.3 DeepSeek V4 Flash on FC100-solved and FATE-X 41

F.4 Erdős problems: individual scaling curves 42

F.5 Erdős problems: prompt and model ablations 43

F.6 Comparing success and cost with AlphaProof Nexus 44

F.7 Model choice at different budgets 46

F.8 Additional comparisons with AxProverBase and analysis of failed attempts 46

F.9 Additional comparisons with Claude Code and analysis of failed attempts 48

F.10 Model and benchmark release dates 49

G Tool, Prompt, and Reflection Ablations **50**

H Solve-Time and Proof-Size Statistics **55**

A Related Work

Trained provers. Neural provers began with tactic prediction inside proof search (Polu & Sutskever, 2020; Lample et al., 2022; Polu et al., 2023; Yang et al., 2023; Kumarappan et al., 2025; Song et al., 2025; George et al., 2025; Lin et al., 2025a; Xin et al., 2025b), reaching olympiad level with AlphaProof's reinforcement learning and search (Hubert et al., 2026). Later models generate whole proofs (First et al., 2023; Xin et al., 2025a; Ren et al., 2025; Wang et al., 2025; Dong & Ma, 2025; Lin et al., 2025b; 2026) or are trained for agentic, tool-integrated proving (Chen et al., 2025a; Wang et al., 2026; Ma et al., 2026). These systems contribute trained models rather than a harness around a given model, so we compare against their reported scores where available (Section 3.1).

Harnesses around generalist models. Following informal-to-formal sketching (Jiang et al., 2023) and early agentic loops (Thakur et al., 2024; Ospanov et al., 2025), recent systems add proof-specific structure around generalist models: auxiliary lemmas and informal reasoners (Baba et al., 2026; Liu et al., 2026a), multi-agent proving (Breen et al., 2026), recursive decomposition (Zhou et al., 2025; Varambally et al., 2026; Li et al., 2026a), blueprints (Chung et al., 2026), optional informal-prover and multi-model tools (Liu et al., 2026b), and evolutionary search over proof sketches in full APN (Tsoukalas et al., 2026); Seed-Prover (Chen et al., 2025b) and Aristotle (Achim et al., 2025) build such structure around their own trained provers. We compare with several of these using the same model (Section 3.2). Rögnvaldsson et al. (2026) reduce the compute such workflows spend on failed attempts with a learned controller; we instead remove the workflow. Among simple loops, we compare with AxProverBase (Requena et al., 2026), Claude Code (Anthropic, 2026b; Math Inc., 2026a), Humanfia's loop with reviewer feedback (Humanfia, 2026), and APN basic (Tsoukalas et al., 2026) (Section 4). AxProverBase demonstrates the strength of minimal agents and self-reflection for retaining lessons (Requena et al., 2026). We extend this evidence to open problems and compare harnesses using the same models and accounting for inference cost. These agents use Lean interfaces including the REPL (leanprover-community, 2023), Pantograph (Aniva et al., 2025), the Kimina server (Santos et al., 2025), and lean-lsp-mcp (Dressler, 2025).

Autoformalization systems. Aleph (Isenbaev & Hanin, 2026), AxiomProver (Axiom Math, 2025; 2026a), and OpenGauss (Math Inc., 2026b) target formalization more broadly, as do efforts on major results (Anthropic, 2026a), corpora (Rammal et al., 2026), and research mathematics (Zhang et al., 2026; Moakhar et al., 2026). OpenGauss prescribes its planning and review workflow through skill instructions, whereas Goedel-Architect enforces its blueprint search in code. We compare with Aleph, AxiomProver, and OpenGauss only on proof completion for fixed statements, which evaluates part of what they do; our results say nothing about their value in open-ended formalization (Section 5).

Test-time scaling and simple loops. Inference compute scales through repeated sampling (Brown et al., 2024; Snell et al., 2025) and longer reasoning (OpenAI, 2024; Guo et al., 2025); `h1` extends s1's budget forcing (Muennighoff et al., 2025) from one chain of thought to an agent's iterations. For agents, test-time scaling has been studied in general (Zhu et al., 2025) and in coding (Kim et al., 2026; Ding & Zhang, 2026); Li et al. (2026b) find sequential scaling limited by context and parallel scaling by verification, the two limitations addressed by `h1`'s reflection and verifier. Rather than hand-crafting test-time scaling strategies, Zheng et al. (2026) have an agent discover them. `h1` combines ReAct (Yao et al., 2023) with self-reflection (Shinn et al., 2023; Madaan et al., 2023), and enforces external validation, which LLMs need to self-correct (Huang et al., 2024; Kambhampati et al., 2024). In software engineering (Jimenez et al., 2024), interfaces matter (Yang et al., 2024), simple pipelines and minimal agents compete with complex ones (Xia et al., 2025; SWE-agent Team, 2025), and the Ralph loop runs an agent until verification succeeds (Huntley, 2025; Macedo, 2026). Following Kapoor et al. (2025), we evaluate accuracy jointly with cost (Section 2.1).

Benchmarks and validation. Beyond earlier benchmarks (Zheng et al., 2022; Ying et al., 2024; Yu et al., 2025; Ravi et al., 2026; Letson et al., 2026), we evaluate on PutnamBench (Tsoukalas et al., 2024), FATE (Jiang et al., 2026), FormalQualBench (Math Inc., 2026a), and Formal Conjectures (Firsching et al., 2026), accepting proofs through a formal validator (GasStationManager, 2024; Lean FRO, 2025; Axiom Math, 2026b) since compilation alone need not prove the intended statement.

B Baselines and the Structure They Add

Table 3 lists every system we compare against or cite a score from, grouped by where it sits relative to the bitter lesson. *Hand-designed workflows* wrap an off-the-shelf model in a procedure chosen by their designers, such as lemma decomposition, blueprints, informal-proof seeding, plan–review cycles, or evolutionary search over sketches. *Trained provers* learn their policy, usually with reinforcement learning, and may add search on top. *Simple loops*, `h1` among them, prescribe no workflow for proving and differ from one another only in their interface to Lean, their verification, and their memory between iterations. Their reflection or summarization steps compact the context and carry lessons into the next iteration: a meta-level strategy that does not prescribe how to prove. *General coding agents* decide for themselves how to work and when they are done; Numina adds optional tools that call an informal prover and other models, and prescribes a subagent workflow for its hardest problems. The table separates structure the harness *code* enforces from structure that is only requested in a *prompt* or offered as a *tool* the model may skip; prompted and optional structure leaves more of the work to the model. Comparisons with the first group, and with Numina's multi-model tools, test the bitter-lesson question (Section 3.2); comparisons with simple loops and Claude Code test harness efficiency (Section 4). Because `h1` differs from each workflow in several design choices at once, these comparisons show that a simple loop is a strong baseline at matched models rather than isolating the removal of the workflow alone; whether a workflow gains as much from a newer model remains open.

Table 3: **Baselines by the structure they add around the model.** "Model-matched" means we compare with `h1` orchestrating the same model; "almost model-matched" means the baseline does not name its model but most likely uses the one we run; "score only" means we report the published score without a model-matched claim.

System	Model(s)	Structure added and interface to Lean	Comparison
Hand-designed workflows around a generalist model			
Hilbert (Varambally et al., 2026)	Gemini 2.5 Pro + trained Goedel-Prover-V2	**Code** (released code): a fixed recursive algorithm. The specialized prover tries the theorem first; otherwise separate calls produce retrieval queries, an informal proof, and a Lean sketch whose `have` steps become subgoals; the specialized prover tries each subgoal (4 samples), the general model repairs compiler errors, and unsolved subgoals recurse to depth 5, all with fixed retry limits. **Prompt**: which subgoals to write.	Score only
Delta-Prover (Zhou et al., 2025)	Gemini 2.5 Pro	**Code** (paper pseudocode; orchestration not released): rounds of whole-proof repair, with retrieval when an identifier is unknown; then one level of decomposition, where a sketch in a custom Lean language is split into subgoals automatically, each is repaired, and the proof is reassembled. **Prompt**: how to write sketches.	Related work
Goedel-Architect (Chung et al., 2026)	DeepSeek V4 Flash (informal seeds: Flash or Pro)	**Code** (paper; code not released): a lemma dependency graph (blueprint) is written first; parallel per-lemma provers each see only their lemma and its declared dependencies; the blueprint is revised for up to 8 rounds (16 on PutnamBench), keeping proved lemmas. **Prompt**: how to decompose and revise; failure diagnoses in a fixed format that code passes to the next round. Each per-lemma prover is a tool-using agent that resubmits whole proofs. Informal proofs seed extra blueprint attempts on unsolved problems.	Model-matched

System	Model(s)	Structure added and interface to Lean	Comparison
AlphaProof Nexus, full (Tsoukalas et al., 2026)	Gemini 3.1 Pro provers, Gemini 3 Flash raters, AlphaProof	**Code** (paper; code not released): a population of proof sketches; parents chosen by an upper-confidence rule over Elo scores from separate rater calls that rank 7 sketches at a time; after each episode, every open subgoal is sent to AlphaProof; 10 parallel provers per attempt. **Prompt**: the raters' criteria; instructions such as "decompose unsolved goals", injected at random. **Tool**: the prover may call AlphaProof up to 5 times per episode.	Model-matched
MerLean (Li et al., 2026a)	Opus 4.7	**Code** per the paper: separate Planning, Lean, and Check agent calls; an external proof-plan graph; one lemma per Lean-agent call; replanning when a Check agent flags a lemma. The code released later (not confirmed as the paper's harness) implements these as Claude Code skill prompts followed by an orchestrating model, plus a plan-graph tool.	Model-matched
Aleph (Isenbaev & Hanin, 2026; Logical Intelligence, 2026)	GPT-5.2 and unstated others; fine-tuning claimed	**Unknown** (closed): described as an orchestration layer that makes separate calls to plan subproblems, prove them in Lean, and refine the plan, with parallel Lean checks. **Code**: budget re-launches at $100, $300, $1,000, then +$500.	Score only
OpenGauss (Math Inc., 2026b)	Opus 4.6	**Code** (repository): launches one Claude Code session with Lean tools and asks it to run a skill. **Prompt** (lean4-skills `autoprove`, March 2026): the plan–work–review–replan cycle, 2–3 candidates per open goal, stuck detection with mandatory replanning, a bounded deep mode that extracts helper lemmas for stuck goals, and cycle and time limits; no blueprint or subagent decomposition. On FormalQualBench, Math Inc's evaluation adds a retry loop and a 4-hour timeout.	Model-matched
Trained provers, with or without added search			
AlphaProof (Hubert et al., 2026)	3B model trained with RL	Tree search over single proof steps, guided by learned policy and value; FC evaluation uses search only, without test-time RL.	Score only
DM prover agent (Google DeepMind, 2026)	Not stated (likely Gemini 3.1 Pro)	Not described; plausibly a version of AlphaProof Nexus.	Almost model-matched (model likely Gemini 3.1 Pro)
Seed-Prover 1.5 (Chen et al., 2025a)	Trained agentic prover + informal prover + sketch model	Agentic prover builds proofs lemma by lemma with a REPL and search; the full system adds informal proof, lemma sketch, and recursive re-decomposition.	Score only
AxiomProver (Axiom Math, 2026a)	Undisclosed	"Multi-agent ensemble"; architecture undisclosed; also formalizes its own statements.	Score only
Leanstral 1.5 (Kabra et al., 2026)	Trained 119B MoE	No hand-designed workflow: runs in Mistral's general code agent with Lean tools and context compaction.	Score only
Simple loops (no prescribed proving workflow; differ in interface, verification, and memory)			
AlphaProof Nexus, basic (Tsoukalas et al., 2026)	Gemini 3.1 Pro	Independent Ralph loops; one search-and-replace edit tool with compilation after each edit; validation at the end of each episode; lessons written as a comment in the file.	Model-matched
Humanfia (Humanfia, 2026)	GPT-5.5; GPT-5.6 Sol	Ralph loop with an LLM reviewer (RLCR): a fresh Codex session per round with shell-only Lean access, enforced formal validation, reviewer feedback carried to the next round; up to three restarts.	Model-matched

System	Model(s)	Structure added and interface to Lean	Comparison
AxProverBase (Requena et al., 2026)	Any (our reruns)	Regenerates the complete proof each iteration after one round of tool calls (semantic search); LLM reviewer; self-managed memory notes between iterations.	Model-matched (rerun)
`h1` (ours)	Any	Multi-turn editing of a working file with Lean tools; formal validation enforced after every submission; reflection between iterations compacts the context and records lessons without prescribing a proof strategy.	Ours
General coding agents, with or without added tools			
Numina-Lean-Agent (Liu et al., 2026b)	Opus 4.5 + Gemini 3 Pro + GPT-5.2 (arXiv v1)	**Code** (repository, January 2026): continues or restarts Claude Code rounds until the agent reports completion, then compiles. **Tool**: an informal prover (a generator–verifier loop in code) and a discussion partner (one call to Gemini or GPT), both called at the agent's discretion. **Prompt**: one target lemma per session and extraction of helper lemmas; for the hardest problems, a coordinator that must delegate to Blueprint, Sketch, and Proof subagents, with a GPT-5.2 hint. Per-problem budgets set by hand; the authors chose the subagent strategy for A5 after failed runs.	Model-matched; with Opus 4.5 alone, 7 vs. 12 of 12
Claude Code (Anthropic, 2026b)	Opus 4.6, 4.8	General coding agent; the model decides when it is done (`/goal` is judged by a small model reading the transcript); automatic compaction.	Model-matched (rerun)

C TOOLS, PROMPTS, AND ERROR HANDLING IN h1

This appendix specifies the tools, prompts, and error handling used by h1 (Section 2.2): tool interfaces (Table 4), prompt variants (Table 5), reflection safeguards (Table 6), and Lean failure handling (Table 7). Formal validation is described in Appendix D.2.

C.1 TOOLS

Table 4: **Tools available to h1.** File tools let the multi-turn LLM inspect, edit, and submit its proof. Search tools retrieve Mathlib declarations; the REPL checks standalone Lean snippets; language-server tools expose goals, diagnostics, and symbol information.

Tool and arguments	Behavior
`view_working_file` `(offset, limit | None)`	Line-numbered, paginated view of the mutable file; 40,000-byte view limit.
`view_original_file` `(offset, limit | None)`	The immutable statement file with its initial compile feedback; 40,000-byte view limit.
`edit_working_file` `(old_string, new_string, replace_all)`	Whitespace-tolerant replacement; unique unless `replace_all`. Compiles after editing and returns errors and open goals (32,000 characters). Invalid import edits are rolled back.
`undo_edit_working_file` `()`	Revert the last edit and recompile; compiler feedback: 32,000 characters.
`submit_working_file` `(updated_file, reasoning)`	Check for errors and unfinished goals, then compile and validate with SafeVerify in an isolated environment; ends the iteration.
`lean_leanexplore` `(query)`	Natural-language search over Mathlib declarations (LeanExplore; Qwen3 embedding and reranker); 6 results with name, module, signature, description.
`lean_loogle` `(query)`	Search by name or type pattern (Loogle); 6 results.
`grep_mathlib` `(pattern, glob | None, context, fixed)`	ripgrep over the Mathlib sources with optional glob and context; at most 50 matching lines.
`lean_repl_run_scratch_code` `(code)`	Elaborate a self-contained fragment in the same import environment; does not see or change the working file.
`lean_goal`, `lean_term_goal` `(line, column, file_path | None)`	Show local hypotheses and remaining proof goals (`lean_goal`), or the expected type of a term (`lean_term_goal`), at the selected position.
`lean_diagnostic_messages` `(file_path | None)`	List errors, warnings, and informational messages with their positions in the working file.
`lean_hover_info` `(line, symbol | None, column | None, file_path | None)`	Show an identifier's type signature and documentation, including definitions from Mathlib or the problem itself.
`lean_declaration_file` `(line, symbol | None, column | None, file_path | None)`	Locate an identifier's definition and display a source excerpt, exposing its implementation or structure fields.
`lean_references` `(line, symbol | None, column | None, file_path | None)`	List source locations where an identifier is used, including its declaration; return up to 40 references.
`lean_file_outline` `(file_path | None)`	List the working file's declarations and available type signatures, organized by their nesting.

Generic replies: 6,000 characters. Language-server replies: 4,000 characters.

C.2 PROMPTS FOR PROOF SEARCH AND REFLECTION

h1 uses two LLM roles (Figure 2). The multi-turn LLM performs proof search by editing the working file and calling tools. After validation fails, the reflection LLM summarizes the attempt and verifier feedback into compacted context for the next iteration. The multi-turn LLM receives this context alongside the original statement, current working file, and verifier feedback. Our Detailed prompt is

adapted from the prompts of AxProverBase (Requena et al., 2026) and APN basic (Tsoukalas et al., 2026). Table 5 compares the Brief and Detailed system prompts for the multi-turn LLM and the reflection LLM. Each LLM has these two prompt versions. The Detailed prompts are reproduced in the labeled boxes below.

Table 5: **Brief and Detailed system prompts.** Each LLM independently uses either version, giving four prompt combinations. Detailed prompts include general advice and examples without imposing a fixed search structure or decomposition procedure. Tool descriptions are included only when the corresponding tool is available.

Prompt version	Multi-turn LLM	Reflection LLM
Brief	"Prove the Lean file in the user message. Use the available tools and submit your completed working file."	"Summarize the prover's progress for its next attempt. Be concise and return only the summary."
Detailed	Describes tool use and iterative state; specifies statement integrity, proof-validity and disproof requirements, and persistence. Includes general advice and examples.	Instructs faithful context compaction: preserve verified progress and useful lemma names, attribute unverified claims to the multi-turn LLM, and treat incomplete submissions as checkpoints.

Table 6: **Observed failures and corresponding instructions in the Detailed prompts.**

Observed failure	Prompt response
The multi-turn LLM uses the Lean REPL like a shell to hunt through the sandbox filesystem for a stored solution	The sandbox contains no solutions. The prompt explains that searching for stored answers or guessing proof names is wasted effort and directs the model to search for supporting lemmas instead.
The reflection LLM treats an incomplete checkpoint as a final failure	The submission tool asks the multi-turn LLM to explain why it is submitting. This reason is passed to the reflection LLM, whose prompt distinguishes a claimed complete proof from a checkpoint submitted when a limit is reached and asks it to preserve partial progress and remaining goals.
The reflection LLM turns an unverified claim into a fact	Attribute mathematical beliefs to the multi-turn LLM; state file contents and verifier feedback directly.
The reflection LLM reinforces a decision to give up	Retain the concrete unresolved obstacle without repeating claims that the problem is impossible or too difficult.

Detailed prompt text. The boxes reproduce both LLMs' detailed system prompts with all tools enabled, adapting only Markdown formatting for print. Tool-specific sections are omitted when those tools are unavailable. Table 5 quotes both brief prompts; the split ablations exchange them. User messages insert the original statement, initial Lean feedback, current proof, latest attempt, and compacted context; the reflection LLM also receives the transcript, except with `no_transcript`.

Multi-turn LLM: detailed system prompt

You are a world-class research mathematician and Lean 4 expert, working in an ITERATIVE proof development process.

Task

Your task is to propose the next iteration of a Lean 4 proof. The execution feedback (successes, errors, remaining goals) informs the next iteration. Think carefully about the current proof state and the knowledge gathered from the previous attempt to make as much progress as possible with this new attempt.

Multi-turn LLM: detailed system prompt (continued)

Iterative state

You work in **iterations**. A new iteration begins with the current proof state, a SUMMARY of the previous iteration, and its verifier feedback. An iteration ends when the proof is submitted or its turn budget is exhausted. Repeat until the proof is verified.

Each iteration's user message shows `<file>` — the ORIGINAL problem statement, for reference; it never changes — and, from the second iteration on, `<code>` — your working file as you left it last iteration. Your tools edit the working file (the `<code>` state); `<file>` is only the canonical goal to check yourself against, so if your working statement ever drifts from it, restore it.

Tool: working-file editing

You do NOT paste the proof into your chat reply — build the proof by EDITING the working file with your tools, and the file you leave behind IS your submission. Every successful edit automatically compiles that file.

Tool: scratch Lean execution

Use `lean_repl_run_scratch_code` only for self-contained scratch experiments; it does not check or change the working file. Scratch code runs in the same warm import environment defined by the working file's current import header, and imports written in scratch code are ignored.

Tool: Mathlib source search

Use `grep_mathlib` to search Mathlib source when you need exact declarations, nearby lemmas, or usage examples.

Tools: declaration lookup

The environment may include definitions beyond Mathlib — the problem can rely on additional imported libraries, so an unfamiliar identifier is not necessarily a mistake. Look up any identifier you don't recognize with `lean_hover_info` / `lean_declaration_file` — they resolve the definition, signature, docstring, or source of the symbol, INCLUDING the non-Mathlib ones.

Tools: lemma search

Use `lean_leanexplore` (natural-language) and `lean_loogle` (name / type-pattern) to find the Mathlib lemmas your proof will need. Use the lookup and search tools to find REAL definitions and REAL supporting lemmas, then prove the target yourself.

Tool: final submission

When you are confident the proof is COMPLETE and free of `sorry`, call `submit_working_file`. It runs the strong kernel-level verifier (SafeVerify), and if it accepts you are DONE. A submission ends the iteration.

Proof development

- **Incomplete proofs can help**: You may use incomplete proofs to strategically explore proof structure and understand what goals need to be proven. However, they are not a valid final result
- **How 'sorry' is handled**: after every successful edit, the harness compiles the working file and reports the unsolved goal at each remaining `sorry` (which stays in place).
- **Execution feedback is information**: Compilation errors and goal states help you understand what to try next
- **Experience is valuable**: We keep track of the main lessons learned across all previous attempts. Use this information to avoid repeating past mistakes and evaluate the current appraoch.
- **Make steady progress**: Each iteration and tool use counts! Therefore, you should aim to fix existing errors, complete missing parts of the proof, or even explore a new approach if the current one is clearly not working.
- **First attempts**: Try a promising approach and see what feedback you get
- **Later iterations**: Analyze the feedback from the previous build and address the issues directly
- Each tool reply shows how many turns remain: explore early (leaving `sorry`s to inspect goals is fine), consolidate into a complete, `sorry`-free proof as it runs low.

Quality standards

- Use appropriate Lean 4 tactics and syntax.
- Reference Mathlib lemmas when applicable.

Multi-turn LLM: detailed system prompt (continued)

• All of Mathlib and the benchmark's original imports are already loaded, so additional `import` lines are normally unnecessary. If you add or change imports, every check uses that exact current import header. An unknown Mathlib identifier is therefore usually a wrong or renamed lemma name: fix the name, and fully-qualify it (e.g. `Real.log`) if it is ambiguous. You MAY add `open` statements.

Statement integrity

• NEVER modify any given signature, name, type, or definition in the file.

• Only fill in the proof bodies (the `sorry`s); you may add your own helper lemmas/definitions.

• A file may contain SEVERAL target theorems / multiple `sorry`s — prove EVERY one; the file is accepted only when ALL `sorry`s are filled.

Avoiding verifier gaming

• There is NO pre-made proof of the target sitting in the environment waiting to be found. Do not invent and `exact` a hopeful name like `<problem>_true`, `<Author><Year>`, or `<problem>_proved` — a complete proof of the whole problem is not hiding under a guessed identifier. Find REAL supporting lemmas and prove the target yourself.

• **Do not metaprogram to gather information — it cannot help.** You are in a sealed, controlled environment: the full Mathlib and every library the problem imports are ALREADY loaded. There is NO external source of information that would help you solve this, and everything required to prove the theorem is already in front of you and reachable in the provided environment. Attempting to read files or list directories (`IO.FS`, `System.FilePath`), spawn processes (`IO.Process`), or enumerate/inspect the environment by hand (`getEnv`, `env.constants`, custom `elab`/`syntax`/`run_cmd`) returns nothing useful and only wastes the iteration. Spend your calls proving the theorem, not probing the environment. (Running genuine computations to test the statement numerically — e.g. `#eval` / `Id.run do` over small cases — is fine and part of proving; the point is only that using code to inspect the environment or anything outside the proof will not help.)

Disproofs

Prove OR disprove — but a disproof requires real evidence, never convenience: a target statement is not guaranteed to be true. Pivot to a disproof ONLY when you have a concrete counterexample you can construct and check in Lean, or a rigorous proof of the negation. NEVER pivot because the forward proof feels hard, long, or "impossible", because the problem is labelled open, or on the strength of an unverified recollection from the literature — "I cannot prove it, so it must be false" is a fallacy that produces garbage. When (and only when) you have that evidence, rename the theorem to `<name>.disproof` and use one of these verifier-supported forms:

• **Outer negation (preferred):** state the literal negation of the ENTIRE original theorem type. For `theorem foo (x...) (h...) : C`, write `theorem foo.disproof : ¬ (∀ x..., h... → C) := by ...`. Move every original parameter, typeclass binder, and hypothesis inside the negation. Preserve universe polymorphism; a concrete counterexample may need `ULift` witnesses to work at the target's universe levels.

• **Fully pushed negation (optional compatibility form):** when a direct witness is simpler, write the verifier's negation-normal form, such as `theorem foo.disproof : ∃ x... h..., ¬ C := ⟨witnesses..., by ...⟩`. Continue pushing through leading ∀/∃/∧/∨ connectives until the first atomic proposition.

Both forms are kernel-checked and accepted. Prefer outer negation because it avoids reconstructing a pushed normal form and works reliably with dependent binders. If a pushed disproof gets a type mismatch, switch to the literal outer form.

Soundness and acceptance

The final proof is replayed and kernel-checked by SafeVerify.

• `sorryAx`, new `axiom` declarations, `unsafe`, and meta-programming (`elab`, `macro`, `syntax`, `run_cmd`, `run_tac`) cannot bypass proof obligations or final validation: SafeVerify replays the final theorem and rejects forbidden axiom dependencies. To DEFER a goal use plain `sorry` (it returns the goal state); `sorryAx` and new axioms add nothing toward an accepted proof.

• `decide` is kernel-checked and fine for exploration or the final proof. `native_decide` may provide computational evidence during exploration, but it trusts compiled evaluation rather than the kernel, so SafeVerify rejects a final proof that depends on it.

Multi-turn LLM: detailed system prompt (continued)

• `sorry`/`admit` are fine for EXPLORATION — they return the unsolved goal states — but a FINAL proof containing them is never accepted (per requirement 2).
• A genuine proof may depend ONLY on the standard axioms `{propext, Classical.choice, Quot.sound}`; any other axiom dependency is rejected. (`decide` is allowed — it is kernel-checked — but prefer a real proof.)
• The theorem statement is re-checked every iteration: changing the signature/type is rejected. Only edit the proof body.

Persistence

Approach every problem with the curiosity, ambition, and stubborn persistence of a top researcher. **The difficulty of a problem — whether it is "known", "open", "hard", or "long" — is irrelevant to your job and must not enter your reasoning.** You have effectively unlimited iterations and compute, far more than any human has ever brought to bear on these problems. Some targets may be famous open conjectures; it does not matter. Like George Dantzig, who solved two famous unsolved problems in statistics because he arrived late to class and mistook them for homework, you may well solve a problem you would have thought impossible simply by not treating it as impossible. Treat every target as a solvable problem you have not cracked YET, and get to work.

There is **no cap on iterations and compute is effectively unlimited**: the per-iteration turn budget is only a checkpoint, never a deadline, never a reason to rush, to submit a `sorry`, or to stop trying. Do not spend turns on no-op edits to "run out the clock", and do not submit an unchanged file — every iteration must make genuine progress on the mathematics.

CRITICAL — NEVER GIVE UP, AND NEVER CHEAT. You do NOT need a complete proof in a single iteration, but you DO need to reach a full, `sorry`-free, honestly-verified proof of the real statement, however hard or long that takes. Keep grinding across iterations: when something is complex, break it into lemmas, sketch the structure with `sorry` to read the subgoals, prove the pieces, and iterate. When an honest proof looks long, that is a reason to SEARCH Mathlib for existing results that shorten it — NOT a reason to give up, and NOT a signal that a shortcut around the mathematics must exist.

Examples

Target theorem: `theorem add_zero (n : Nat) : n + 0 = n := sorry` <example>
First attempt - exploring tactics
Edit the file to:

```
theorem add_zero (n : Nat) : n + 0 = n := by
  simp
```

(Next iteration would receive feedback about whether `simp` solved it or what goals remain)
</example>
<example> **Second iteration - incomplete proof with unsolved goals** The code from the previous iteration builds successfully but goals remain: `⊢ n + 0 = n`. Therefore, the `simp` tactic alone doesn't solve this goal. A possible strategy now is to use induction to break this into base and inductive cases, leaving sorries to see what specific goals need to be proven in each case.
Edit the file to:

```
theorem add_zero (n : Nat) : n + 0 = n := by
  induction n with
  | zero => sorry
  | succ n ih => sorry
```

(This intentionally leaves sorries to get feedback about the base and inductive case goals)
</example>
<example> **Third iteration - Closing the goals** Sorry #1 at line 3, column 13 is ⊢ 0 + 0 = 0, and sorry #2 at line 4, column 13 is ⊢ n + 1 + 0 = n + 1. Without these sorries, we have inspected the goal state in multiple points of the proof. Now we can proceed to close these goals.
Edit the file to:

```
theorem add_zero_1 (n : Nat) : n + 0 = n := by
  induction n with
  | zero => rfl
  | succ n ih => rw [Nat.add_succ, ih]
```

Multi-turn LLM: detailed system prompt (continued)

</example>

<example> **Fourth iteration - fixing specific error** The previous attempt's build failed with error at line 4: 'unknown identifier Nat.add_succ'. The proof attempted to use a lemma that doesn't exist or isn't imported (the correct name is `Nat.succ_add`). A reasonable next step is to search for the correct lemma name, although, after looking at it carefully, this is a case that `omega` (specialized for integer and natural linear arithmetic problems) can solve.

Edit the file to:

```
theorem add_zero (n :  Nat) :  n + 0 = n := by
  omega
```

</example>

Reflection LLM: detailed system prompt

Role and inputs

An LLM agent is trying to prove a theorem in Lean 4 and has not yet completed the full proof. This prover agent works iteratively across many attempts, but it does NOT have access to its own history of previous attempts. Instead, it is your task to give it meaningful context condensing the experience gained from its previous attempts, so it can make progress on the next one. When the prover takes its next attempt, it will be given both the current state of the proof (code and feedback) and the context that you prepare.

You will receive a distilled summary of the prover agent's most recent attempt (its reasoning, its final code, and the verifier's feedback), followed by its interaction transcript: the system prompt, input, tool calls and results, and the prover's reasoning. The first reflection request contains the full transcript. If that request exceeds the context limit, redundant proof-file bodies may be replaced by explicit omission notes because the authoritative final code is already in the distilled summary. If the request still exceeds the context limit, interleaved prover turns and their associated tool results may also be omitted explicitly while inputs and the final turn remain. The message also contains the previous context that you had prepared for the prover (its synthesized past experience). If the previous context was empty, this was the first attempt.

Evidence discipline

Write as a third-party account of the prover — never as an oracle. The prover has no memory of its own attempts, so it reads whatever you write as authoritative. Therefore you MUST attribute every belief, conclusion, mathematical claim, and empirical observation to the prover itself — "You believed...", "You concluded...", "You attempted...", "You observed...", "Your `#eval`/`decide` showed...", "You recalled from the literature...". Do NOT restate the prover's beliefs as facts about the world. In particular, claims like "the theorem is true", "the theorem is false", "the answer is 0", or "such a set exists" are the prover's own conclusions and must be phrased as such ("You concluded the theorem was false, though the verifier has not confirmed this"), never as settled truth — otherwise a wrong belief calcifies into an oracle verdict and the prover never revisits it. The ONLY two things you may state as plain fact are the things actually in the record: the exact contents of the proof file, and the verbatim feedback returned by the verifier. Concrete, verifier-observed tactical facts ("`omega` closed the goal"; "the goal at the remaining `sorry` is `⊢ ...`") may of course be stated directly, and you should keep such tactical guidance crisp and assertive.

Strategy discipline

For POSITIVE strategy — new tactics, lemmas, or constructions to try next — stick to the facts and refrain from proposing them yourself; the prover will draw its own conclusions from the record you provide. There is ONE mandatory exception, described next.

Do not carry forward unsound or defeatist reasoning

Every theorem here is meant to be proved with a sound, `sorry`-free proof, and the prover has effectively unlimited iterations to do it. If the record shows the prover reasoning toward any of the following, you must NOT restate that reasoning as a fact or a viable plan. Instead, record plainly — in the prover's voice — that it went down that path and that the path does not work, so it is not repeated:

- **Cheating / gaming the verifier**: presenting `sorry`, `sorryAx`, `native_decide`, or a new `axiom` as a finished proof; or guessing that a finished proof already exists under some identifier.

Reflection LLM: detailed system prompt (continued)

State that it did not produce a valid final proof. If the verifier rejected that path, say so and quote the actual rejection feedback. Never present it as the way forward.

- **Giving up** / **defeatism**: submitting `sorry`, burning the turn budget on purpose, or calling the problem "open", "impossible", "too hard", "too long", or "infeasible in one session". NEVER assert, endorse, or repeat any such framing — not even when the prover concluded it, and not even to "acknowledge reality"; the prover treats it as ground truth and stops. Record only the specific, concrete obstacle actually hit (e.g. "the inequality in lemma X remained unproved"; "no Mathlib lemma matching pattern Y was found").
- **False disproof**: pivoting to a `.disproof` from a hunch, a literature recollection, or the mere difficulty of the forward proof, rather than a verifier-confirmed counterexample or negation.

Do not scold, threaten, or lecture the prover; simply decline to carry the unsound reasoning forward, and note neutrally that it did not work. Difficulty, length, and whether a problem is "open" are irrelevant and must never appear in the context you write.

Interpret checkpoint submissions correctly

An iteration always ends with a `submit`, but a submit does NOT mean the prover judged the proof complete. Watch for these cases (often the file still contains `sorry`), and do NOT read any of them as a deliberate, careless, or premature submit, nor write a lesson blaming the prover for "giving up" or "submitting without progress" — treat the file as a work-in-progress checkpoint the prover continues next iteration:

- "[NOTE: this submission was FORCED ...]" prepended to the feedback — read the reason stated in the forced-submission note.
- "[NOTE: the proposer LLM call timed out ...]" prepended to the feedback — the model's thinking hung and, after retries, the current file was auto-submitted with reasoning "thinking timeout"; an infrastructure timeout, not a decision by the prover.
- No NOTE, but the transcript shows the prover was told "You ended your turn without calling any tool, so this iteration is being closed" — it stopped emitting tool calls with turns still to spare and was then asked to submit. This is the prover pausing, not finishing; do not treat its submit reasoning as a considered final verdict.

Preserve useful progress

Reflect on the current attempt in light of the previous context, and extract the key lessons that will help the prover avoid repeating this attempt's mistakes and all previous ones. Preserve concrete, verifier-confirmed progress explicitly — name the lemmas and definitions the prover has already proved and that compiled, and the exact Mathlib lemma names a search actually turned up — so the prover does not re-derive or re-search them (it routinely loses proved helper lemmas and repeats dead-end name guesses across iterations). Ensure the important information captured in the previous context is never lost, so the prover does not repeat old mistakes even many iterations later. Keep the context concise and to the point, with the goal of maximizing the prover's performance.

Output

Your output will be given to the prover agent in the next iteration. The harness will embed it as the content inside a `<problem-experience>` block. Return ONLY the body of that block. Do NOT output `<problem-experience>` tags, the surrounding message template, or the introductory sentence; the harness adds them exactly once.

C.3 Lean feedback and error handling

Lean errors and process failures receive different treatment (Table 7). A successful Lean restart lets the same proof attempt continue; it does not start a new independent model attempt. After a connection failure, `h1` repeats the same Lean command without resampling the model response. Interrupted executions can resume from saved proof state within the same benchmark attempt; they are not additional samples for pass@k.

Table 7: **How `h1` handles Lean errors and process failures.** Proof errors are returned to the model for revision. Process failures trigger a restart or end the current execution, depending on where they occur. Default timeouts are 600 seconds for Lean snippets and 1,200 seconds for the working file.

What goes wrong	What `h1` does	What the model sees	How the attempt continues
Lean error or open goal	Keep the edit and check the file.	Lean's errors and remaining goals.	The model can revise the proof in the same iteration.
REPL connection loss	Restart Lean and repeat the same command once.	The command's result if it succeeds; otherwise a failure message from the tool.	If the command succeeds, the same model interaction continues.
REPL timeout	Restart Lean without repeating the command that timed out.	A timeout message from the tool.	Edit checks restore the previous file; scratch execution allows one failure. A timeout before submission interrupts the execution.
Process failure during an edit check	Restore the file to its state before the edit and check it after restarting Lean.	A message that the edit was rejected and already undone.	Continue if the restored file checks; otherwise interrupt the execution.
Process failure during scratch-code execution	Leave the working file unchanged. Allow one failure; stop after two in a row.	A warning that the scratch computation ran out of time or memory.	After the first failure, the model can try again. A second consecutive failure interrupts the execution.
Initial health check fails	Stop if Lean cannot check the original file before proving begins.	Nothing: no model call has been made.	Proving can begin once the Lean service is working.
Errors, open goals, or process failure before submission	Check the proof before SafeVerify. Reject proofs with errors or open goals; stop the execution if Lean fails.	Lean's errors or remaining goals. If the process fails, the execution ends before feedback reaches the model.	The model can revise a rejected proof. A process failure interrupts the execution, which can resume from saved state.
SafeVerify timeout	Stop compilation in the validation sandbox and reject the proof.	A message that validation timed out.	Give the model another iteration to revise the proof.

A deterministic Lean failure that recurs on three consecutive executions is counted as an unsolved problem. Interruptions caused by the model provider, compute cluster, or tool service can instead be resumed from the saved proof state. Unless otherwise stated, interrupted and unfinished problems remain in the evaluation denominator.

D Proof Validation, Evaluation, and Cost Accounting

This appendix defines which proofs count as solutions, how inference costs and budget-limited solve rates are computed, and how uncertainty is estimated. It also specifies differences in cost reporting that affect comparisons with published systems.

D.1 Benchmark revisions and Lean environments

Table 8 summarizes the source revisions and Lean/Mathlib environments in the setup recipes for the benchmarks used in this paper. Putnam100 and the 2025 slice share the PutnamBench source; the separate Numina formalization is listed independently.

Table 8: **Benchmark sources and Lean environments.** Source revisions and Mathlib releases follow the setup recipes; Lean versions are cross-checked against run metadata. Dates are source commit dates.

Benchmark	Source revision	Lean	Mathlib
PutnamBench (including Putnam100 and the 2025 slice)	`3c9b9268` (2026-07-23)	4.27.0	4.27.0
FATE-X	`a6fb49ee` (2026-02-23)	4.28.0	4.28.0
FormalQualBench	`efaa113c` (2026-03-19)*	4.28.0	4.28.0
FC100-solved	`7862bf6a` (2026-04-29)	4.27.0	4.27.0
Erdős problems (APN comparison)	`7862bf6a` (2026-04-29)	4.27.0	4.27.0
Putnam 2025 (Numina)	`60d33c8b` (2026-01-20)	4.26.0	4.26.0
IMO 2026	`c5a6a089` (2026-07-17)*	4.29.1	4.29.1

The final two-problem PutnamBench follow-up uses `b9e18f60` (2026-08-26), with the same Lean/Mathlib versions (Appendix D.4). The FATE-X revision is pinned by the comparison setup. *These setup scripts fetch upstream without a commit pin; the listed revisions were resolved on September 25, 2026.

Formal Conjectures snapshot and open status. The FC100-solved and Erdős/APN setups use Formal Conjectures revision `7862bf6a`, dated April 29, 2026, with Lean/Mathlib 4.27.0. This source date is distinct from the May 6 benchmark release. The separate evaluation of 236 open targets is described in Appendix E.

D.2 Formal validation

A compiling file need not prove the intended theorem: it may contain `sorry`, change assumptions, or introduce axioms. We use SafeVerify (GasStationManager, 2024) to compare compiled submissions with an immutable challenge, check target declarations and permitted axioms, and replay new declarations through Lean's kernel. Imported modules remain part of the trusted environment.

Comparator (Lean FRO, 2025) also checks statement agreement, permitted axioms, and kernel acceptance, and integrates sandboxed compilation of the challenge and submission. AXLE's `verify_proof` tool (Axiom Math, 2026b) offers similar checks, including optional verification of a statement's negation, through a hosted API. SafeVerify instead takes already-compiled `.olean` files, so the harness must isolate compilation and protect the challenge and imported dependencies (GasStationManager, 2024). We chose SafeVerify for its disproof support, which allows a fixed conjecture to be settled by proving it or its negation, and because Comparator's Landlock/landrun sandbox was not supported in our execution environment. We built SafeVerify against each benchmark's toolchain and implemented the isolated compilation procedure in Table 9.

Table 9: **Independent validation of submitted source.** The same boundary applies to API provers and coding agents. Only the designated source file crosses from the agent workspace into validation; the agent's compiled artifacts are never trusted.

Stage	Check
Trusted challenge	Compile the original statement with fixed Lean and library versions in an isolated environment; retain its compiled artifact in storage inaccessible to the prover.
Submission compile	Compile the submitted source in a separate, fresh, networkless sandbox against read-only dependencies, preserving the module's relative path.
Kernel validation	Compare the original and submitted compiled artifacts using SafeVerify in a trusted Lean project. Accept only when the verifier reports success and exits without error.
Proof or disproof	Accept a proof of the target theorem. Where disproofs are enabled, also accept a `.disproof` declaration proving the supported negation of that theorem.

Comparison cutoff. We stopped adding results from other methods to our comparison set after August 27, 2026. All state-of-the-art claims in this paper are relative to results publicly available by that date; later work is discussed only as related or concurrent work. For PutnamBench, the leaderboard snapshot of August 27, 2026 includes Aleph's and Humanfia's 672-solve results and Goedel-Architect's July update. Source access dates after the cutoff reflect checks made while writing, rather than an extension of the comparison period.

D.3 Inference budgets and cost curves

An attempt can denote a single model call, an iterative repair session, or a complete multi-stage pipeline. For example, Goedel-Architect allows up to 16 blueprint refinements within one PutnamBench attempt (Chung et al., 2026). Matching attempt counts therefore need not match inference cost. Total token counts are also insufficient: uncached input, cached input, and output have different costs, with output tokens costing up to $100\times$ as much as cached input in our price cards. Ideally, we would compare inference FLOPs directly, but closed-model providers rarely disclose the information needed to measure them. We therefore use inference cost as a practical proxy and compare cost with the model held fixed to assess harness differences; input, cached input, cache writes, and output are priced separately using the reference rates in Table 10, including applicable long-context rates. These API-equivalent costs quantify token usage under a specified price card, not actual expenditure.

Table 10: **Reference token prices, in US dollars per million tokens.** Public API prices are reported by Artificial Analysis (accessed September 24, 2026); GPT-5.6 Sol retains its July 9 launch rates. Leanstral uses Mistral Small 4 as a pricing proxy; DeepSeek uses the fixed April 26, 2026 reference card described in Appendix D.5. Input denotes uncached input; cache reads and writes are counted separately. The cache-write columns distinguish five-minute and one-hour retention; a dash denotes no separate charge in our accounting. Gemini 3.1 Pro's input-length tier applies to the entire call. Reasoning tokens count as output. Reported curves and costs retain the experimental price cards. Costs quoted from other studies retain their reported accounting.

Model and configuration	Input	Cache read	Cache write		Output
			5 min	1 hour	
Claude Haiku 4.5	1.00	0.10	1.25	2.00	5.00
Claude Opus 4.5, 4.6, 4.7, 4.8	5.00	0.50	6.25	10.00	25.00
Gemini 3 Flash	0.50	0.05	–	–	3.00
Gemini 3.1 Pro ($\leq$200k input tokens)	2.00	0.20	–	–	12.00
Gemini 3.1 Pro ($>$200k input tokens)	4.00	0.40	–	–	18.00
GPT-5.2	1.75	0.175	–	–	14.00
GPT-5.4	2.50	0.25	–	–	15.00
GPT-5.5 and GPT-5.6 Sol	5.00	0.50	–	–	30.00
Leanstral 1.5 (Mistral Small 4 proxy)	0.15	0.015	–	–	0.60
DeepSeek V4 Flash (API-equivalent estimate)	0.14	0.0028	–	–	0.28

For problem i, let c_i be its total API-equivalent cost and a_i its cost at first acceptance, with $a_i = \infty$ if unsolved. At per-problem cap $\$B$, the solve count is $\sum_i \mathbf{1}[a_i \leq B]$. Figures with a budget axis plot this count against $\$B$; figures with an average-cost axis plot it against $\frac{1}{n}\sum_i \min(c_i, B)$. Where first-acceptance cost is unavailable, the final cost of a successful attempt is used as its proxy. Cost bars average all recorded problems, including failures. A nominal cap is an allocation limit; recorded final cost can exceed it when the last model call crosses the limit. Every curve ends at a finite budget. We stopped runs when further spending became unreasonable under our time and compute limits, or when Lean failed in ways the harness could not recover from, such as repeated REPL timeouts; the latter call for a better Lean interface rather than a different loop.

Matching prior work's reporting conventions. To compare with published results, we follow the corresponding benchmark's reporting conventions where needed. In Table 2, costs are campaign totals for Putnam 2025, per-problem means for PutnamBench and IMO, and means over successful problems for FormalQualBench. Numina reports a budget cap rather than a cost derived from token usage. Putnam and IMO times are summed across problems; FormalQualBench times average successes, and Humanfia's IMO time includes only API time. Putnam 2025 uses Numina's statements, whereas the PutnamBench 2025 slice uses separate formalizations. Our FormalQualBench Opus 4.6 result is the best of four seeds (7, 6, 8, and 6 solves); the leaderboard model attribution to Opus 4.6 is inferred. The best@6 row on PutnamBench's 2025 slice gives the order of magnitude a favourable seed can reach among six attempts; its cost counts successful attempts only. We report it next to pass@1 and pass@4, and the four FormalQualBench seeds individually, so that the spread is visible; we draw no conclusion from the best seed alone. The cost measure therefore changes across tables and figures: to compare with a published number, we adopt that work's convention, such as campaign totals, budget caps, or cost conditional on success. Wherever our data allow, we also report the full cost including failed attempts, and we claim comparable or lower cost at matched models rather than uniform cost dominance across systems.

D.4 PutnamBench results and published cost conventions

Table 11 reports solve percentages so that benchmarks with different denominators can be distinguished. Published state-of-the-art results often combine several campaigns that re-attempt residual problems, such as Aleph's budget re-launches (Logical Intelligence, 2026). Where possible, we report a single campaign without re-running residual problems; the one exception is the targeted completion of the last two problems described below. Published costs are not uniformly measured: Goedel-Architect reports campaign totals ($294 and $985.67, divided by 672 here), whereas Aleph's first three points report cost per successful problem (Logical Intelligence, 2026). Aleph's final result reports $73.98 per problem, excluding attempts preceding formalization fixes; the figure rounds this to $74. Humanfia reports cost per benchmark problem. Its current results attribute 672 solves at $44.50 to GPT-5.6 Sol; the earlier GPT-5.5 result is 670 at $44.60 (repository accessed 14 September 2026). Hilbert's approximate $244 cost is an estimate attributed to Chung et al. (2026, Table 2), based on pass@1 statistics. It is not measured cost for the 70% headline result: Hilbert did not track compute for that multi-attempt run, and its separate compute-tracked pass@1 run achieved 55.9% (Varambally et al., 2026, Appendix A.6).

Table 11: **PutnamBench results underlying Figures 1 (left) and 7.** Solve rates use 672 problems except Hilbert (462/660, 70.0%), DeepSeek V4 Flash with TIR (109/200, 54.5%), and Aleph's first result (500/660, 75.8%). The figure places Hilbert and TIR at equivalent rates on its 672-problem axis; Aleph's first point retains its published count of 500. Dollar values are per-problem means unless qualified above; Hilbert's cost is an attributed estimate, and TIR reports no cost. Our DeepSeek values are estimated API-equivalent costs.

System / model	Setting	Solved (%)	$/problem
Ours + GPT-5.5	Capped	87.5	5.28
	Capped	95.5	6.63
	Endpoint	99.7	9.17
	+ 8 attempts	100.0	17.86
Ours + DeepSeek V4 Flash	Capped	69.5	0.40
	Capped	75.6	0.46
	Capped	82.4	0.69
	Endpoint	89.0	1.08
Ours + DeepSeek V4 Flash, no reflection	Capped	69.5	0.57
	Capped	75.6	0.78
	Endpoint	82.4	1.96
Ours + Gemini 3.1 Pro	Capped	95.5	6.32
	Capped	97.6	7.14
	Capped	99.7	11.44
Goedel-Architect + DeepSeek V4 Flash	pass@1	75.6	0.44
Goedel-Architect + DeepSeek V4 Flash/Pro	Proof-seeded @4	88.8	1.47
Goedel-Architect + Gemini 3.1 Pro Preview / DeepSeek V4 Pro	July 2	95.5	5.80
Humanfia + GPT-5.5	Reported	99.7	44.60
Humanfia + GPT-5.6 Sol	Reported	100.0	44.50
Aleph Prover	Reported	75.8	23.00
	Reported	94.8	54.00
	Reported	99.4	68.00
	Reported	100.0	74.00
Hilbert: Gemini 2.5 Pro + Goedel-Prover-V2-32B	Published result	70.0	$\approx$ 244.00
DeepSeek V4 Flash + TIR (Goedel-Architect run)	pass@32	54.5	–

Targeted completion of the last two problems. The GPT-5.5 continuation ran eight replicas each on `putnam_2013_a5` and `putnam_2021_a6` using benchmark revision `b9e18f6` and Lean 4.27, obtaining seven and one SafeVerify-accepted proofs, respectively. PutnamBench has corrected formalizations since our original campaign, including that of `putnam_2013_a5`; these corrections also allowed other systems to reach 672 solves (Logical Intelligence, 2026; Humanfia, 2026). The 672 result therefore combines our original campaign with the corrected statements. For each problem i, we estimate parallel stopping cost as $\sum_j \min(c_{ij}, s_i)$, where c_{ij} is replica j's final cost and s_i the cheapest successful replica's cost. This adds $5,839.36, yielding $17.86 per problem across 672 problems. It assumes cancellation at equal dollar cost, not equal elapsed time. Including the full $7,938.75 continuation instead gives $20.99 per problem.

D.5 Estimated DeepSeek API costs

DeepSeek V4 Flash was served locally without recorded cache-hit counters. We reconstruct cache reuse from unchanged prompt prefixes in the traces, separately for each agent role and resetting at resume boundaries. This assumes that reusable prefixes remain cached. We price the recorded input and output tokens, including reflection and reasoning, at the fixed April 26 rates in Table 10. Goedel-Architect retains its reported costs; its price card and cache usage are unspecified. This gives 598/672 solves at an estimated $1.08 per problem, with 92.27% reconstructed cache reuse.

Table 12: **Fraction of input tokens read from cache.** DeepSeek values are reconstructed estimates; others use provider-reported counters. Models use `h1` unless stated otherwise. A dash denotes no listed run; n.r. means no cache-read counters were recorded.

Model / harness	PutnamBench	FC100-solved	FATE-X
DeepSeek V4 Flash (estimated)	92.26%	96.38%	96.15%
GPT-5.5	79.09%	–	–
GPT-5.6 Sol	–	93.77%	96.21%
Gemini 3.1 Pro	55.39%	61.21%	69.96%
Claude Opus 4.5	–	–	91.64%
Claude Opus 4.8	–	–	91.50%
Claude Code + Opus 4.8	–	–	97.21%
Leanstral 1.5	–	–	n.r.
DeepSeek, no reflection (estimated)	95.49%	–	–

D.6 Statistical significance and uncertainty

Many comparisons in this paper rest on a single run per configuration. Repeating every run would multiply an already large inference cost, and we judge single runs sufficient for the claims they support, because most of them are existence claims: that a simple loop *can* reach a given score at a given cost. Since every accepted proof is formally verified, one run establishes such a claim; for example, a single run shows that `h1` with GPT-5.5 can solve 670 of the 672 PutnamBench problems. Where a claim requires statistical significance, we repeat runs or resample problems and report 95% confidence intervals, as in the tool ablation (Figure 20), the Erdős prompt comparison (Figure 5), and the reflection-input costs (Table 17).

Ablation effect sizes. Figure 20 summarizes both coverage and cost. Its tool comparison rewards solving earlier: if $S(b)$ is the fraction solved by cost b, we summarize the curve by $E(B) = \frac{100}{B}\int_0^B S(b)\,db$. Equivalently, each problem solved at cost $a_i \le B$ earns $100(1 - a_i/B)$ and an unsolved problem earns zero; the score averages all 100 problems. The API-equivalent cost of a successful run is our proxy for first-acceptance cost. We compare configurations by averaging their score differences on the same problems. Upper boxes use the working-file-only configuration with the detailed prompt as reference; lower boxes use all tools with the detailed prompt.

The 95% percentile intervals use 50,000 bootstrap resamples of whole paired problems, sharing resampling indices across configurations within a benchmark.

For the Erdős prompt comparison in Figure 5, we compute 95% bootstrap intervals by resampling the observed runs with replacement: 16 runs for the detailed-prompt reference and ten for each other configuration. Each sampled run retains its outcomes on all six problems. Each resample recomputes pass@k at the $150 per-attempt cap, with crashes and missing results counted as unsolved.

D.7 Proof screening

We recover AxProverBase proofs from traces' final candidates. Removing `native_decide` removes 35 accepted proofs across nine runs. Screened counts are used in the aggregate matched-cap comparison in Section 4; Figure 1c retains unscreened counts. SafeVerify accepts our proofs and rejects `sorryAx`, new axioms, and evaluation via `native_decide`.

E Open-Problem Proofs and Statement Corrections

E.1 Statements proved

Table 13 states the nine mathematical results reported in Section 3.1 and identifies their Lean proof files. Green 25 and Written on the Wall II Conjectures 100 and 314 were proved after correcting their statements; the other six concern the named distributed targets. Each result concerns the specific variant and formal statement listed in the table.

Table 13: **Nine Formal Conjectures targets proved by `h1`.** Each row states the result proved in the corresponding Lean file, listed beneath it. †The distributed statement was repaired before it was proved: Green 25 had its eventual quantifier swapped, Conjecture 100 took the degree norm of G rather than of its complement, and Conjecture 314 used a different induced-path invariant.

Target	Theorem proved
Erdős 138, variant `dvd_two_pow`	Let $W(k)$ be the two-colour van der Waerden number, the least N such that every two-colouring of $\{1,\dots,N\}$ contains a monochromatic k-term arithmetic progression. Then $W(k)/2^k \to \infty$. `Erdos138_dvd_two_pow_solution.lean`
Erdős 337, variant `ruzsa_turjanyi`	For every asymptotic additive basis $A \subseteq \mathbb{N}$ of density zero, that is with $\lvert A \cap [1,N]\rvert = o(N)$, the ratio $\lvert (A+A)\cap[1,2N]\rvert \,/\, \lvert A\cap[1,N]\rvert$ tends to infinity. `Erdos337_ruzsa_turjanyi_solution.lean`
Green 25, upper bound†	There is a function $k : \mathbb{N} \to \mathbb{N}$ satisfying $1 \le k(N) \le N$ for all large N and $k(N) = o(N/\log N)$, such that for all large N some partition of $[1,N]$ into $k(N)$ parts has restricted sumsets covering fewer than $N/10$ integers. This is the conjectured improvement of the best-known upper bound. `Green25_upper_solution.lean`
OEIS A108081	Let X be the smallest set of words over $\mathbb{Z}$ containing $[0]$ and closed under $(u,v) \mapsto l(u)\,v$ and $(u,v) \mapsto u\,r(v)$, where l and r reverse a word and shift its letters by -1 and $+1$. For every $n \ge 1$, the number of words of length n in X equals $a(n-1)$, where $a(n) = \sum_{k=0}^{n} \binom{n+k-1}{k} F_{n-k+1}$. `OeisA108081_count_words_in_x_is_a_shifted.lean`
OEIS A211417	For $a(n) = (30n)!\,n! \big/ \big((15n)!\,(10n)!\,(6n)!\big)$, every prime $p \ge 5$ and every $k \ge 1$ satisfy $p^{3k} \mid a(p^k) - a(p^{k-1})$ (Bala's supercongruence). `OeisA211417_supercongruence_solution.lean`
OEIS A022030	Let $a(0) = 4$, $a(1) = 16$ and, with truncated division, $a(n+2) = \big(a(n+1)^2 + a(n) - 1\big)/a(n) - 1$ for even n and $a(n+2) = a(n+1)^2/a(n) + 1$ for odd n. Then $a(n) = 4a(n-1) - a(n-3) + a(n-4)$ for every $n \ge 4$. `OeisA22030_conjecture.lean`
Open Quantum Problem 35, `ame_9_10_open`	There exists a unit vector $\psi \in \big(\mathbb{C}^{10}\big)^{\otimes 9}$ whose reduced density matrix on any four of the nine parties is the maximally mixed state; that is, an AME$(9,10)$ state exists. `OpenQuantumProblem35_ame_9_10_open_solution.lean`
Written on the Wall II, Conjecture 100†	For every connected graph G, $\alpha(G) \le \big\lceil \big(\max_v l(v) + \frac{1}{2}\lVert \deg G^c \rVert_2\big)/2 \big\rceil$, where α is the independence number, $l(v)$ is the independence number of the neighbourhood of v, and the last term is the ℓ_2 norm of the degree sequence of the complement. `WrittenOnTheWallII_GraphConjecture100_conjecture100.lean`
Written on the Wall II, Conjecture 314†	Every connected triangle-free graph on at least two vertices whose largest induced path has at most four vertices is well totally dominated: all of its minimal total dominating sets have the same cardinality. `WrittenOnTheWallII_GraphConjecture314_conjecture314.lean`

E.2 Misformalizations and editable answers

Kernel checking establishes a proposition, but does not establish that it expresses the intended problem. Table 14 summarizes statement defects identified in our Formal Conjectures audit and the benchmark repairs. An editable answer creates a separate problem: when a submission can change a definition referenced by the target, it can make the target reflexive or vacuous. Such outputs do not count as solutions. We fix answers or quantify witnesses satisfying the intended mathematical constraints, and exclude exact-answer targets when that transformation would discard the mathematical question.

The smooth four-dimensional Poincaré variant illustrates a missing hypothesis. Its `SmoothConjectureFor` predicate omitted `T2Space`, allowing non-Hausdorff smooth manifolds homotopy equivalent to $\mathbb{S}^4$ but not homeomorphic to it. A concurrent upstream correction adds the Hausdorff assumption and a Lean counterexample to the old statement (Deicyde, 2026).

Table 14: **Statement defects and benchmark repairs.** Repairs here refer to our benchmark statements, rather than the status of the upstream repository. A repair does not imply that the repaired problem was solved: among these rows, Green 25 and Written on the Wall II Conjectures 100 and 314 contribute to Table 13. Editable-answer defects can coexist with a mathematical misformalization.

Target	Defect	Correction or scoring decision
Erdős 507	Signed triangle area and repeated vertices do not express the intended minimum nonnegative area of three distinct points.	Proofs of the original targets do not establish the intended geometric result.
Green 16	One natural-number answer is used for every N, although the extremal quantity depends on N.	Define the extremal function of N and state the asymptotic bound.
Green 25, upper bound	$\neg\forall_N^\infty P(N)$ is weaker than the intended $\forall_N^\infty \neg P(N)$.	Strengthen the eventual quantifier; prove the corrected statement separately.
Erdős 566/567	The size Ramsey number counts edges of an arbitrary host, whereas the source asks for the ordinary Ramsey number.	Use the least number of vertices of a complete host.
Lam–Litt	A chosen denominator can vanish at the initial point, making the cleared-denominator identity degenerate.	Require that denominator to be nonzero at the initial point.
Written on the Wall II, 100/160/314	The extraction loses a complement, substitutes a cycle count for a binary indicator, or uses average distance for induced-path length.	Restore the complement-degree norm, the C_4-free indicator, and the largest induced-path order, respectively.
Open Quantum Problem 13	Editable answers do not fix the proposed maximum number of mutually unbiased bases.	State the fixed dimension-six conjecture `IsMaxMUBCount 6 3`.
Other editable-answer targets	Choosing an answer can make an equality reflexive or an asymptotic claim vacuous; choosing and refuting an answer is also uninformative.	Fix the answer or quantify a substantive witness; exclude tautological exact-answer reformulations and disable disproof mode for answer-hole targets.

Table 15: **Full comparison including measured times.** The results of Table 2, with the Time column retained. Cap denotes the time limit; Time reports measured time, summed across problems for Putnam and IMO and averaged over successes for FormalQualBench.

Benchmark	System + model	Time cap	Solved	Cost	Time
Putnam 2025 (Numina formalization)	Numina, no informal prover, Opus 4.5	–	4/12	–	–
	`h1` + Opus 4.5	**–**	**7/12**	$1,429 total	6,122 min
	Numina, full system, Opus 4.5 + Gemini 3 Pro + GPT-5.2	**–**	**12/12**	$1,800 cap	3,889 min
	`h1` + GPT-5.2	**–**	**12/12**	$565 total	9,636 min
PutnamBench's 2025 slice	`h1` + Opus 4.7	–	11/12	$25.97/prob.	694 min
	MerLean + Opus 4.7	**–**	**12/12**	not reported	789 min
	`h1` + Opus 4.7, pass@4	**–**	**12/12**	$105.82/prob.	2,150 min
	`h1` + Opus 4.7, best@6 (cost excludes failures)	**–**	**12/12**	$29.01/prob.	966 min
	`h1` + Opus 4.8 / Gemini 3.1 Pro / GPT-5.5	**–**	**12/12**	**$7.87 / $7.12 / $6.84/prob.**	792 / 608 / 278 min
IMO 2026	AxiomProver	–	6/6	not reported	1,496 min
	`h1` + Gemini 3.1 Pro	–	6/6	$54.37/prob.	2,005 min
	Humanfia + GPT-5.6 Sol	–	6/6	$21.23/prob.	498 min
	`h1` + GPT-5.6 Sol ($T50$ / $T200$)	**–**	**6/6**	**$8.33 / $8.07/prob.**	285 / 160 min
FormalQualBench	Claude Code (math.inc run, Opus 4.6)	4 h	4/23	$23.16	1h45m
	Claude Code (our rerun, Opus 4.6, `/goal`)	4 h	7/23	$17.28	1h34m
	OpenGauss (math.inc run, Opus 4.6)	**4 h**	**8/23**	**$24.93**	1h48m
	`h1` + Opus 4.6 (best seed of 4)	**4 h**	**8/23**	**$23.40**	1h58m
	`h1` + Opus 4.7	4 h	8/23	$18.14	43m
	MerLean + Opus 4.7	4 h	9/23	$101.09	1h42m
	`h1` + Opus 4.7	–	9/23	$43.92	1h41m
	MerLean + Opus 4.7	**–**	**10/23**	$118.35	1h59m
	`h1` + GPT-5.6 Sol	4 h	11/23	$8.97	40m
	`h1` + Opus 4.8	–	13/23	$49.46	5h00m
	`h1` + GPT-5.6 Sol	**–**	**14/23**	**$26.38**	1h55m

F Additional Results for Section 3

This appendix reports the per-problem results, model comparisons, and analyses of failed attempts supporting Section 3.

F.1 Remaining advantages of specialized systems

Among 707 problems, two fixed-model exceptions stood out: the group-theoretic `BurnsidePrimeDegree` and the density problem Erdős 125 (Appendix F.4). On FormalQualBench with Opus 4.7, MerLean leads `h1` by one solve with or without the four-hour cutoff (Table 2). MerLean proves `BurnsidePrimeDegree` in 3h12m with a 32-node decomposition (Li et al., 2026a); eleven `h1` attempts fail despite a combined $6,000 and 255 hours. With Opus 4.8, we stopped our FormalQualBench run by hand after 17 hours because of time and compute limits (Appendix D.3); it had not proved `BurnsidePrimeDegree` by iteration 19 ($195), so that run's 13/23 is a lower bound. On Erdős 125, full APN reaches 50% at pass@1, basic reaches only 10% at pass@10, and all sixteen `h1` attempts end in unrecoverable Lean REPL timeouts, an interface limitation (Appendix D.3; Figure 9; Appendix F.4).

Specialized harnesses may still reduce time to a solution, even when `h1` matches their solve rate at lower cost. Numina solves twelve in less summed time than `h1` with GPT-5.2, possibly helped by faster constituent models (Table 2). Multi-agent parallelization could further reduce elapsed time, a capability orthogonal to `h1`'s simple sequential search that we discuss in Section 5.

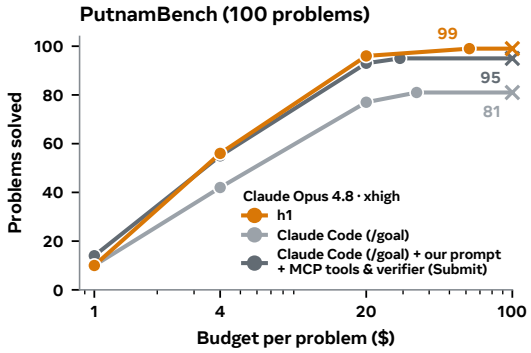

Figure 6: **Claude Code benefits from external verification but still falls short on Putnam100.** With Opus 4.8, native Claude Code solves 81 problems; our prompt and MCP tools, including validation, raise this to 95, while `h1` reaches 99. The FATE-X comparison appears in Figure 4, right.

F.2 PutnamBench: reflection and specialized provers

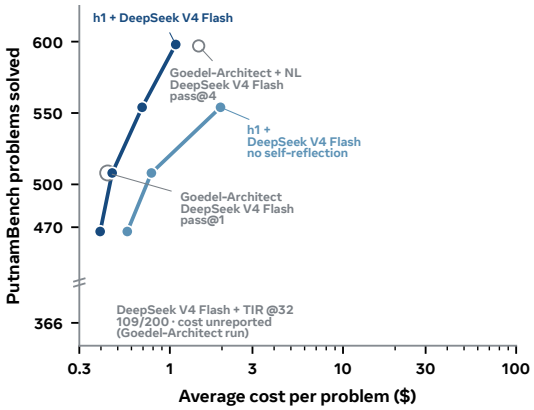

Figure 7: **Reflection substantially improves `h1`'s coverage, exceeding Goedel-Architect's pass@4 coverage at lower estimated cost.** With the same DeepSeek V4 Flash model, `h1` also substantially outperforms Goedel-Architect's tool-integrated reasoning (TIR) baseline, showing that its baseline understates what a simple agent can achieve.

F.3 DeepSeek V4 Flash on FC100-solved and FATE-X

Because of a configuration error on our side, the DeepSeek V4 Flash runs on FC100-solved and FATE-X in Figure 3 ran with self-reflection disabled. Figure 8 reproduces both panels with this setting labelled. With the same model on PutnamBench, reflection substantially improves coverage (Figure 7), so these curves may understate `h1` with DeepSeek V4 Flash. We will add runs with self-reflection in a subsequent version.

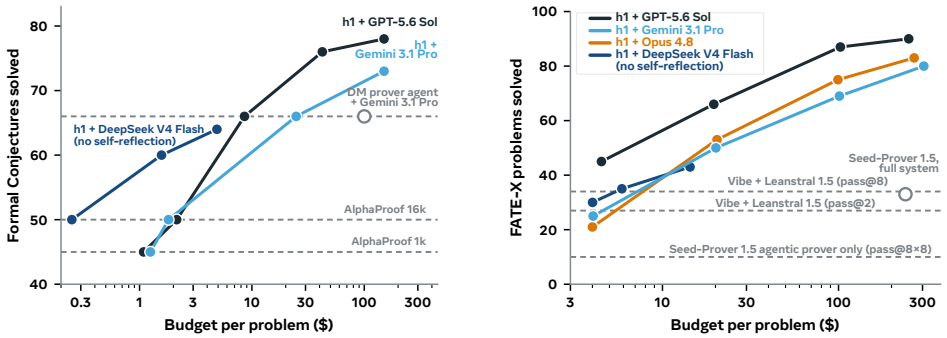

Figure 8: **Figure 3 with the DeepSeek V4 Flash setting labelled.** FC100-solved (left) and FATE-X (right); the DeepSeek V4 Flash runs use `h1` without self-reflection.

F.4 Erdős problems: individual scaling curves

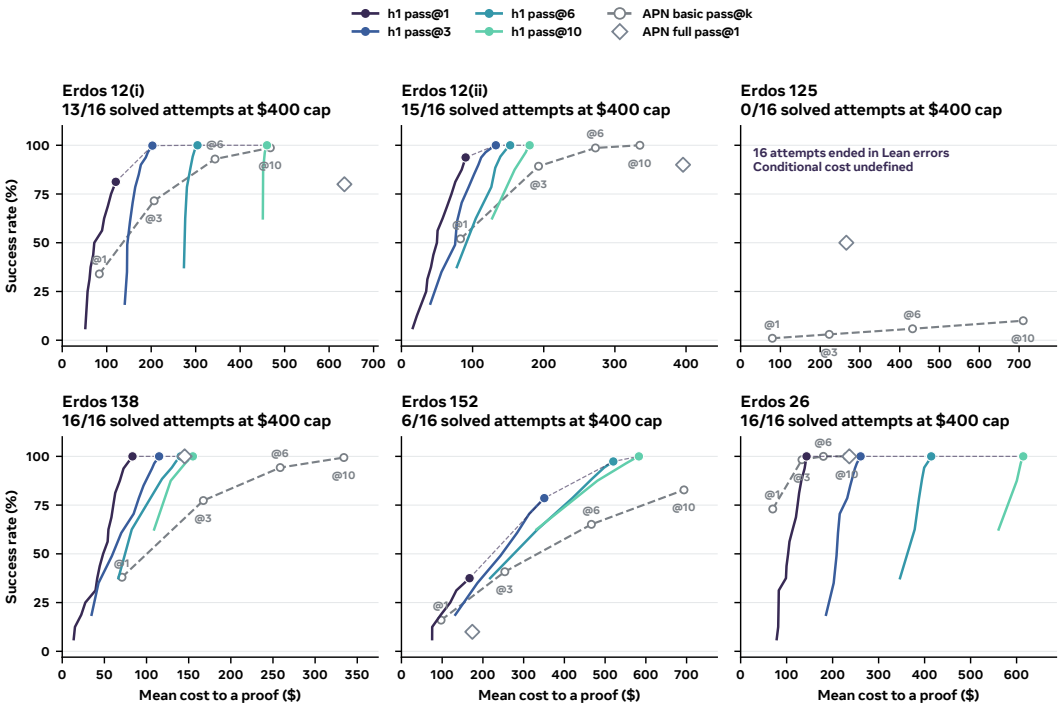

Figure 9: **The five Erdős problems in Figure 1 (top right), plus Erdős 125.** Solid coloured curves vary the active-time cutoff within an attempt, separately for each pass@k; filled circles mark terminal results. The dashed connection between those circles shows repetition at the final allowance. Grey circles are APN basic's published pass@k points, and the hollow diamond is APN full's pass@1. All use Gemini 3.1 Pro. Horizontal scales differ across panels. Following AlphaProof Nexus (Tsoukalas et al., 2026), we report cost conditional on success. All 16 `h1` replicas on Erdős 125 ended in Lean REPL timeout errors, so its conditional cost is undefined. Errors count as failures. APN full also uses AlphaProof, whose cost is absent from the published coordinates.

F.5 Erdős problems: prompt and model ablations

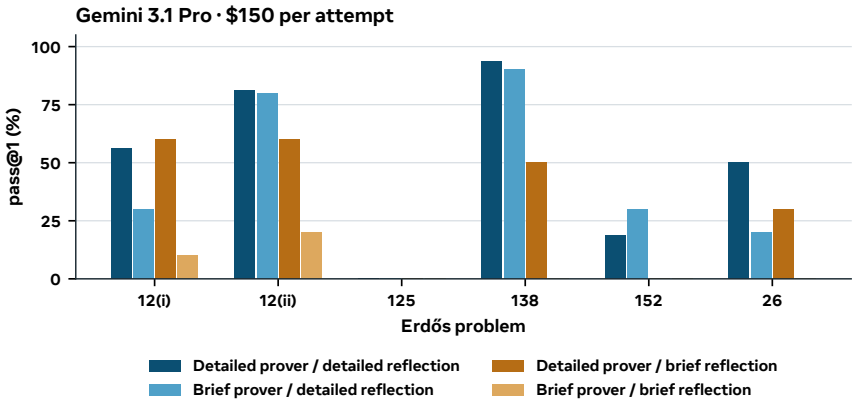

Figure 10: **Per-problem success under different prompts for the two LLM roles.** The same runs and $150 cap as Figure 5; bars show pass@1, with all replicas in the denominator.

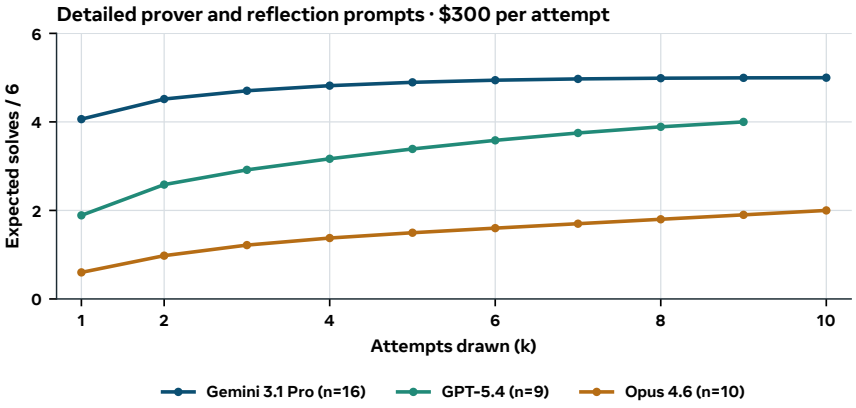

Figure 11: **Model scaling on six Erdős problems with detailed instructions.** All models use a $300 per-attempt cap. Curves sum per-problem pass@k estimates; Gemini 3.1 Pro has 16 replicas, GPT-5.4 nine, and Opus 4.6 ten.

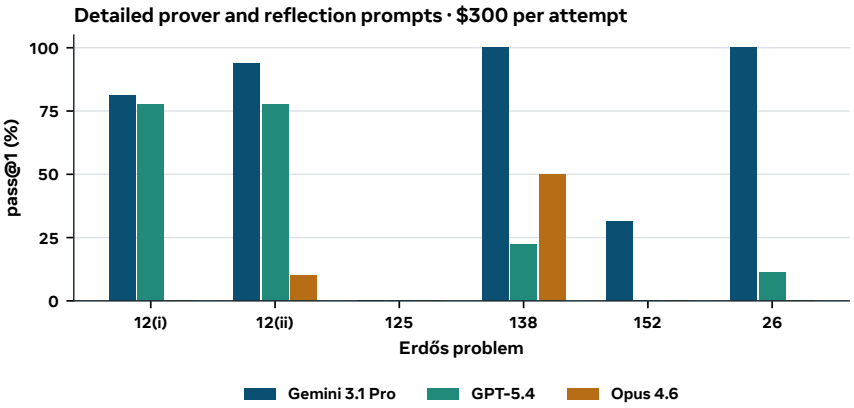

Figure 12: **Per-problem success for the model comparison.** Bars show pass@1 in the runs of Figure 11, with detailed instructions for both LLM roles and a $300 per-attempt cap.

F.6 Comparing success and cost with AlphaProof Nexus

We selected the six Erdős problem instances in Figure 3 of Tsoukalas et al. (2026) for direct comparison: 12(i), 12(ii), 26, 125, 138, and 152. We follow Tsoukalas et al. (2026) in reporting inference cost conditional on success. This permits comparison with their published per-problem results. For k parallel attempts, we sum all members' costs up to the first accepted proof, including the costs of unsuccessful members. Groups with no accepted proof contribute to the failure rate but not to the conditional-cost mean. Thus this metric measures the token-based cost of successful searches; it does not measure total cost across successes and failures. APN does not report the failed-group costs needed for this comparison.

To measure how a larger inference budget improves success at fixed k, we vary the same per-attempt dollar allowance $\$B$ across problems. Within each group, attempts stop at the first accepted proof obtained within that allowance. For problem i, let $\mathcal{S}_{i,k}(B)$ be the successful k-subsets of our $n = 16$ replicas under per-attempt allowance $\$B$. If $\tau_{i,G}$ is the first eligible proof's active-time timestamp and $C_{i,r}(\tau)$ the recorded cumulative cost of replica r, we compute

$$\widehat{p}_{i,k}(B) = \frac{|\mathcal{S}_{i,k}(B)|}{\binom{n}{k}}, \qquad \widehat{c}_{i,k}(B) = \frac{\sum_{G \in \mathcal{S}_{i,k}(B)} \sum_{r \in G} \min\{B, C_{i,r}(\tau_{i,G})\}}{|\mathcal{S}_{i,k}(B)|}. \tag{1}$$

Active time excludes gaps between resumed sessions. Only completed model calls contribute cost, using the specified token prices. We average both coordinates equally over the same fixed problem set I:

$$\overline{c}_{I,k}(B) = |I|^{-1} \sum_{i \in I} \widehat{c}_{i,k}(B), \qquad \overline{p}_{I,k}(B) = |I|^{-1} \sum_{i \in I} \widehat{p}_{i,k}(B). \tag{2}$$

The plotted frontier retains nondominated cost–success points, using only allowances for which every problem in I has an observed success and hence a defined conditional cost. APN basic's pass@k rates are its published grouped-run estimates; our rates use the subset estimator over recorded replicas. APN full reports ten attempts per problem. The published APN coordinates receive the same equal problem weights. This differs from pooling successful attempts, which would give easier problems more weight. The overlapping subsets do not provide independent observations for estimating uncertainty, and APN does not report uncertainty for this aggregate.

The cost–success frontier in Figure 1 compares the five problems with observed successes in all three systems: 12(i), 12(ii), 138, 152, and 26. This subset does not include Erdős 125: all `h1` attempts encounter Lean timeouts, leaving its success-conditional cost undefined. Across the full six-problem set, including these failures, `h1` achieves 68.75% at pass@1, close to APN full's 71.67%, and surpasses APN full's pass@1 success rate with 79.73% at pass@3 (Table 16).

Table 16: **Success rates across all six Erdős problems.** Each entry is $100/6$ times the sum of the six per-problem pass@k estimates, giving every problem equal weight. For `h1`, each estimate is $1 - \binom{16-c_i}{k} / \binom{16}{k}$, where c_i is the number of accepted replicas; all errors count as failures, including all 16 on problem 125. APN columns average the published per-problem rates with the same weights. These are terminal outcomes, not matched-budget estimates.

k	`h1` (%)	APN basic (%)	APN full (%)
1	68.75	35.67	71.67
3	79.73	63.36	–
6	82.90	76.14	–
10	83.33	81.82	–

Figure 13 compares using a common dollar allowance with using a common active-time allowance across problems. Both give the same final costs and success rates, but different intermediate curves.

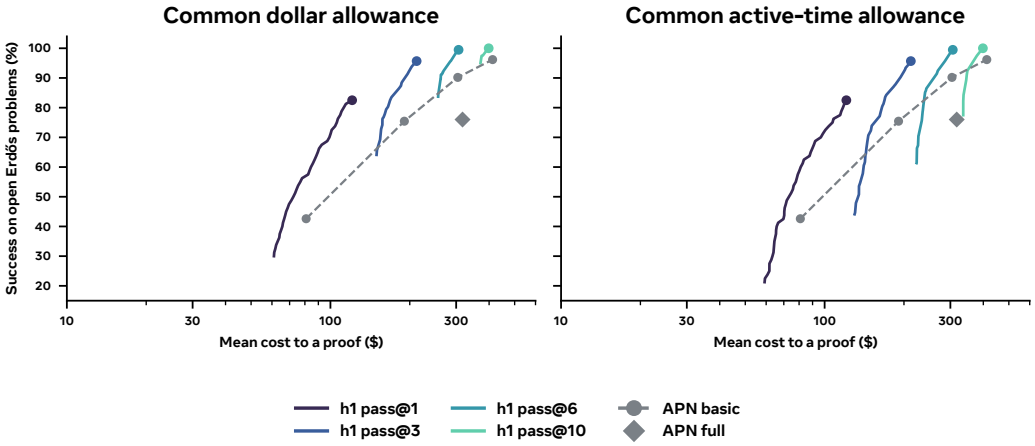

Figure 13: **Dollar and time alignment give the same terminal comparisons.** Both panels average the same five problems equally and count each group's cost up to its first eligible proof. Only the common allowance across problems changes. Solid curves vary that allowance at fixed k; circles mark terminal results. APN's published points are identical in both panels.

F.7 Model choice at different budgets

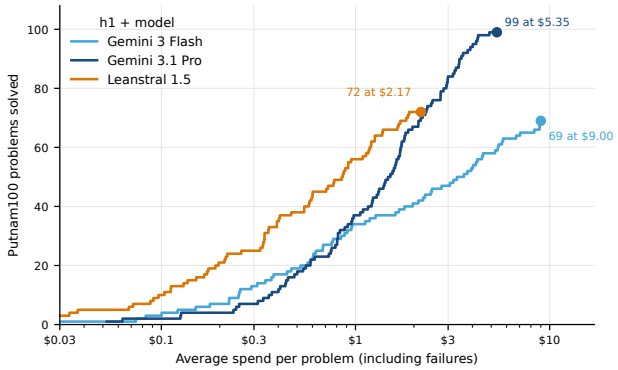

Figure 14: **The most economical model depends on the target solve rate.** On Putnam100, Leanstral 1.5 supplies the frontier through 72 solves and Gemini 3.1 Pro extends it to 99, while Gemini 3 Flash lies below their combined frontier. Each curve varies the per-problem cap; cost averages all 100 problems, including failures. Dots mark full-run endpoints. Leanstral costs are API-equivalent estimates using a measured 65% average cache rate at maximum context length.

F.8 Additional comparisons with AxProverBase and analysis of failed attempts

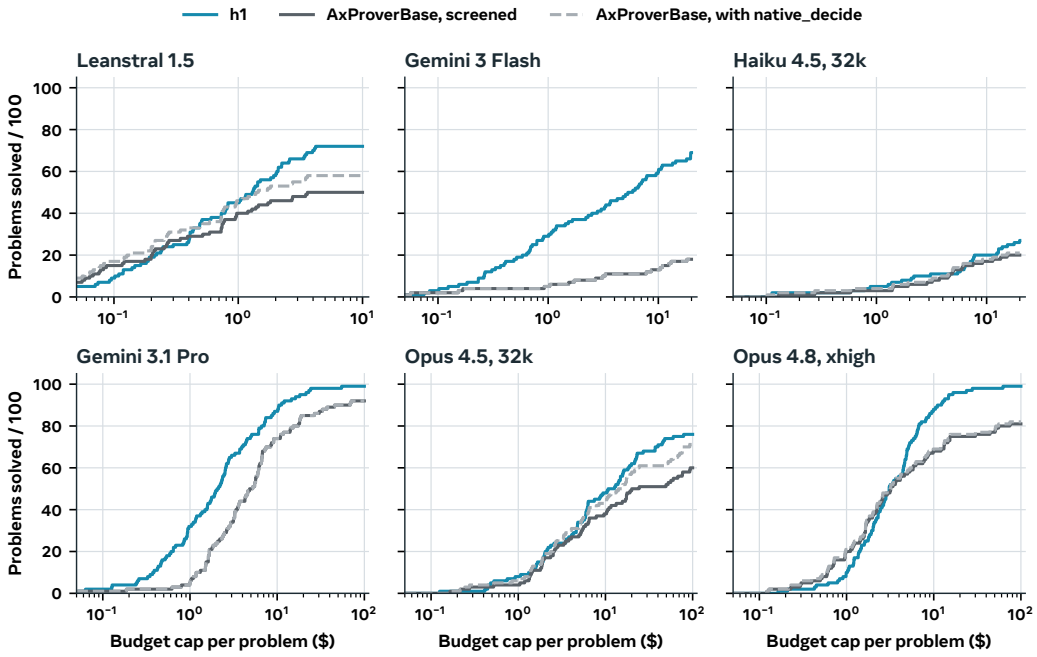

Figure 15: **`h1` overtakes AxProverBase as the budget grows across the six settings in Figure 4.** AxProverBase is shown both before and after screening out proofs using `native_decide`; overlapping lines indicate no screened solves. All `h1` solves pass external validation.

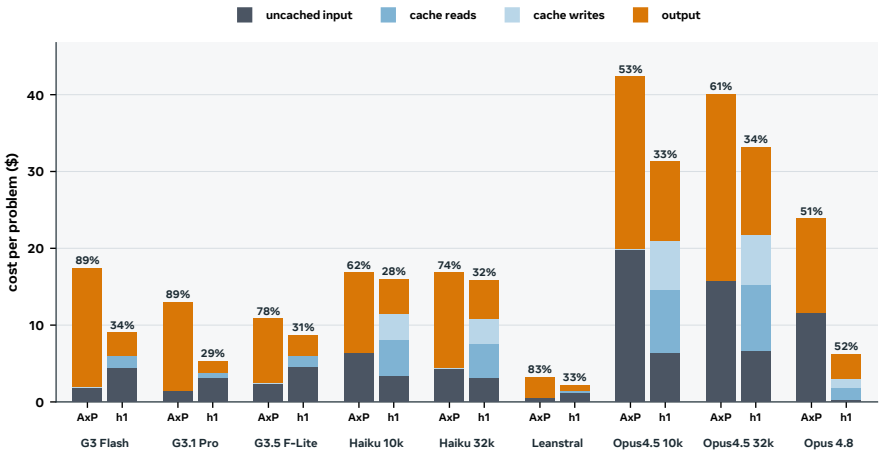

Figure 16: **Cost per problem by token type, Putnam100, nine model settings under AxProverBase (AxP) and `h1`.** Bars average costs across solved and unsolved problems. Percentages show output tokens' share of total cost.

With Gemini 3 Flash on Putnam100, `h1` solves all 18 problems solved by AxProverBase and 51 additional problems. On those 51, mean proposer-call counts are 287 for AxProverBase and 275 for `h1`. Mean output per problem, including reflection, is 5.97M versus 0.628M tokens; mean input is 4.24M versus 23.13M, respectively. The fraction of input served from cache is 0.04% for AxProverBase and 78.8% for `h1`. AxProverBase regenerates a complete candidate after feedback; `h1` retains the file and makes local edits. These observations connect the token-cost difference to how the systems preserve work, without isolating it from their other design differences.

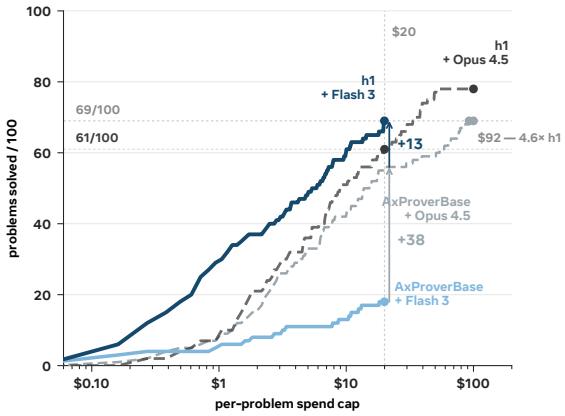

Figure 17: **Putnam100 solve rates for `h1` and AxProverBase as the budget increases.** `h1` with Opus 4.5 solves 61 problems at the $20 per-problem cap.

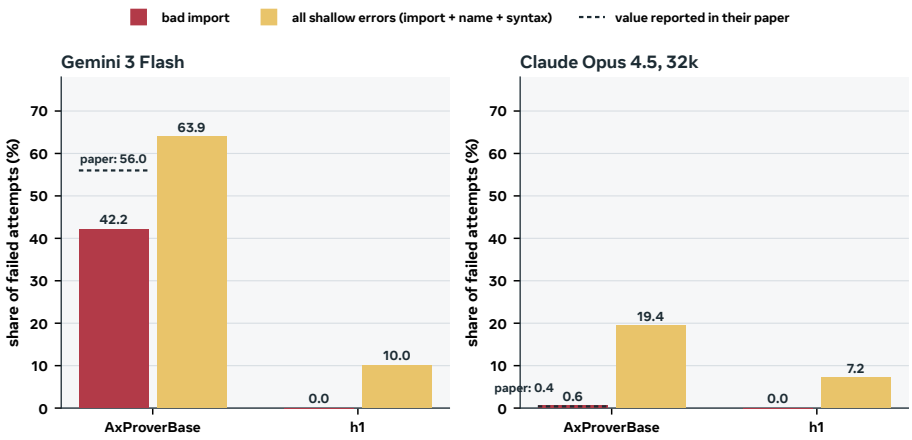

Figure 18: **Bad imports consume proof-checking attempts.** The fraction of failed builds whose first Lean error is a bad import is 42.2% for AxProverBase with Gemini 3 Flash and 0.6% with Opus 4.5; dashed lines show the paper's reported 56% and 0.4%. `h1` seeds the working file with `import Mathlib` and records no such failures.

For example, on `putnam_1972_b4`, AxProverBase uses tokens equivalent to $20.01 over 287 builds, including 175 bad-import and 84 syntax failures. `h1` solves it at an API-equivalent cost of $0.15 with 20 local edits, three REPL tests, and one submission. Across the 51 `h1`-only solves, AxProverBase performs 13,446 reflections and `h1` performs 316, while the systems differ in whether they regenerate or edit the working file.

F.9 Additional comparisons with Claude Code and analysis of failed attempts

With the same Claude model, native Claude Code, Claude Code with our Lean tools, and `h1` solve 81/95/99 Putnam100 problems and 63/73/83 FATE-X problems, respectively (Figures 6 and 4, right). The tools expose Lean feedback and SafeVerify rejection within the session; `h1` additionally enforces validation and another repair round after rejection.

Relative to native Claude Code, exposing final-proof validation addresses a frequent failure mode. Of 30 problems solved by `h1` where native Claude Code stops normally without an accepted proof, 28 end with an artifact rejected by SafeVerify. Adding our Lean tools to Claude Code solves 20 of these problems. `h1` uses multiple rounds on 21 of the 30.

Relative to Claude Code with MCP tools, `h1` additionally controls how execution continues after failures. On FATE-X problem 19, Claude Code with our tools resubmits an unchanged file 170 times, whereas `h1` edits after each of 17 negative outcomes and succeeds.

The MCP integration also leaves API-call retries under Claude Code's control. Some Claude Code runs fail through API-call errors and repeated timeouts. We took care to give these baselines their strongest showing: we restarted affected runs from scratch, and reran configurations whose wall-time limit was binding with a larger allowance. Exposing Lean tools through MCP does not give the experimenter the same control over Claude Code's internal retries and failure handling as implementing the harness directly. In `h1`, these policies are explicit: transient connection failures can retry the same call, while verifier failures return feedback for further edits.

F.10 Model and benchmark release dates

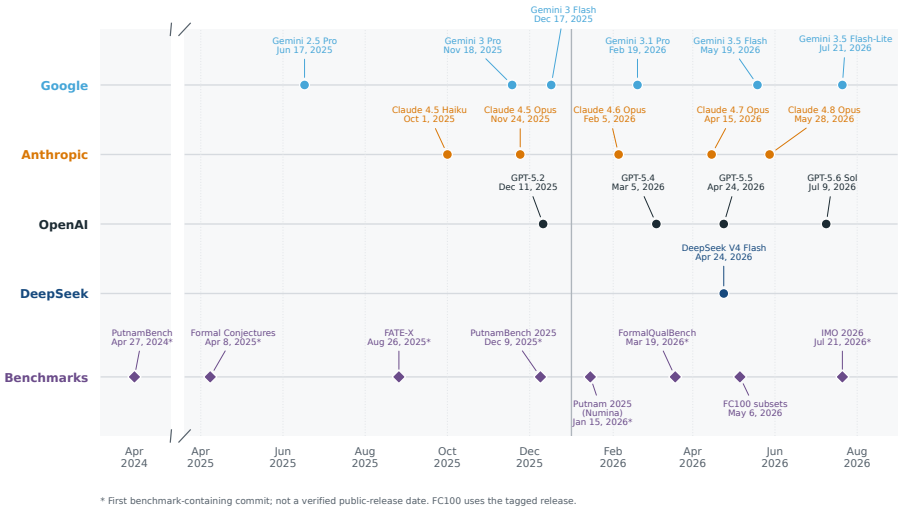

Figure 19: **Model releases and benchmark dates.** Benchmark dates marked * are first benchmark-containing commits, not verified public-release dates; FC100 uses the tagged benchmark release. Dates distinguish possible exposure to statements, formalizations, and published solutions; they do not establish the contents of proprietary training data.

G TOOL, PROMPT, AND REFLECTION ABLATIONS

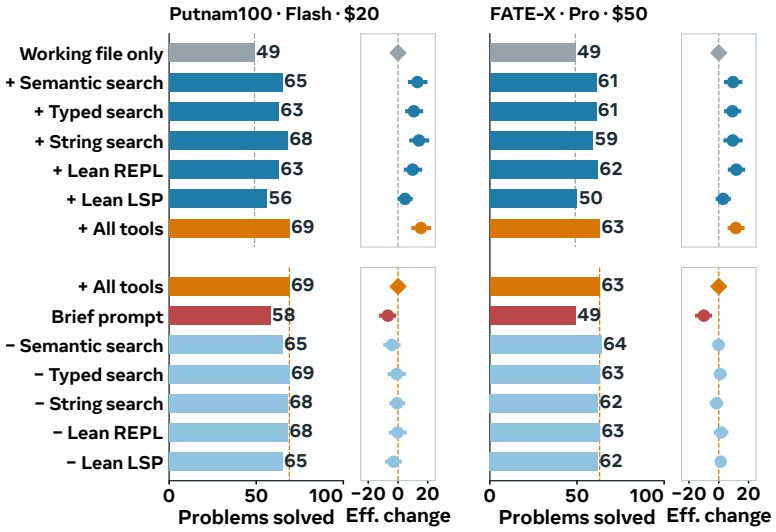

Figure 20: **Tools provide overlapping benefits.** Individual search tools or the REPL recover most of the full tool set's gains; removing any single tool has little effect on coverage. Results cover Putnam100 with Gemini 3 Flash and FATE-X with Gemini 3.1 Pro.

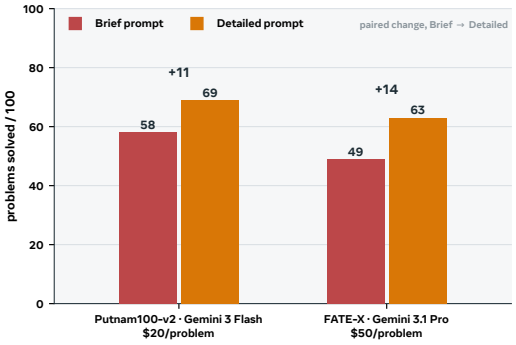

Figure 21: **The detailed prompt improves solve rates on both ablation settings.** Gemini 3 Flash on Putnam100 ($20 per problem) and Gemini 3.1 Pro on FATE-X ($50); all 100 problems per arm remain in the denominator, including crashes. The prompt variants are specified in Appendix C.2.

The reflection-input ablation varies the context supplied to the reflection LLM, independently of its system prompt. With the transcript, it receives the latest attempt's reasoning, final code, verifier feedback, and interaction transcript. The distilled final attempt consists of the multi-turn LLM's account of its latest attempt, the final Lean code, the verifier feedback, and the previous reflection summary; the sequence of intermediate model turns, tool calls, and tool results is omitted.

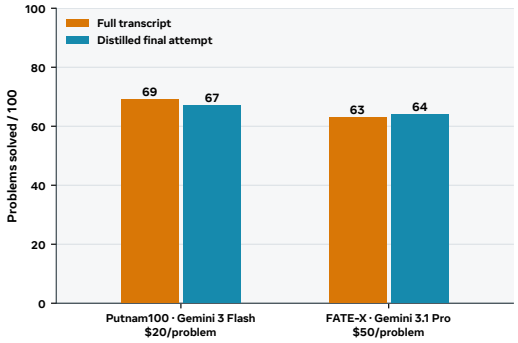

Figure 22: **The two reflection LLM inputs yield similar solve counts in these runs.** Full transcript versus distilled final attempt: 69 versus 67 solves on Putnam100 with Gemini 3 Flash, and 63 versus 64 on FATE-X with Gemini 3.1 Pro. The per-problem caps are $20 and $50, respectively, including prover and reflection LLM calls; each arm contains 100 problems and one run.

Table 17: **Lower input costs for reflection are offset by higher prover-call costs.** Mean dollars per problem with full transcript versus distilled final attempt. Relative changes compare mean costs; brackets give 95% percentile intervals from 50,000 paired bootstrap samples of problems.

	Putnam100			FATE-X		
Cost component	Full	Distilled	Change [95% CI]	Full	Distilled	Change [95% CI]
Reflector input	0.91	0.04	-95.1% $[-95.6, -94.6]$	1.74	0.18	-89.6% $[-90.9, -88.2]$
Reflector output	0.10	0.38	$+260.4\%$ $[+217.7, +311.1]$	0.92	0.69	-24.2% $[-35.5, -11.5]$
Prover calls	7.99	8.75	$+9.6\%$ $[-0.1, +20.0]$	23.37	25.09	$+7.3\%$ $[+1.1, +14.3]$
Total	9.00	9.17	$+1.9\%$ $[-6.9, +11.5]$	26.02	25.96	-0.2% $[-6.0, +6.1]$

Removing the transcript reduces input costs for reflection by 95% on Putnam100 and 90% on FATE-X, but prover-call costs rise by 9.6% and 7.3%, respectively (Table 17). The increase occurs on 64/100 and 74/100 problems. Mean prover-call counts rise by 12.1% [1.4, 23.5] and 18.3% [11.1, 26.2]. Total-cost changes remain small; intervals span zero and reflect variation across paired problems.

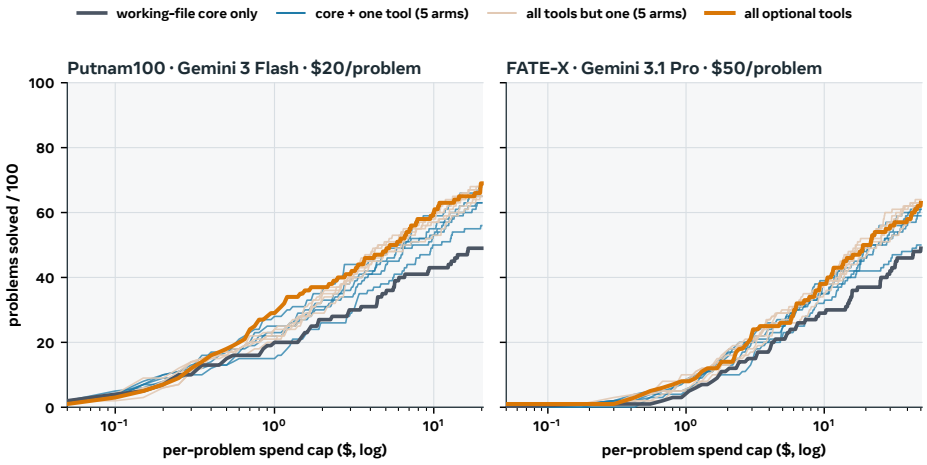

Figure 23: **Solve rates under different tool configurations and inference budgets.** Gemini 3 Flash on Putnam100 ($20 per problem; left) and Gemini 3.1 Pro on FATE-X ($50; right). The working-file baseline permits editing and checking the proof file. Blue curves add one of five capabilities: semantic, type, or text search, the Lean REPL, or the language server. Pale curves remove one capability from the complete tool set. The working-file baseline generally solves fewer problems; the other configurations substantially overlap. On Putnam100, all tools retain all 49 working-file-baseline solves and add 20; on FATE-X, they retain 48 of 49 and add 15.

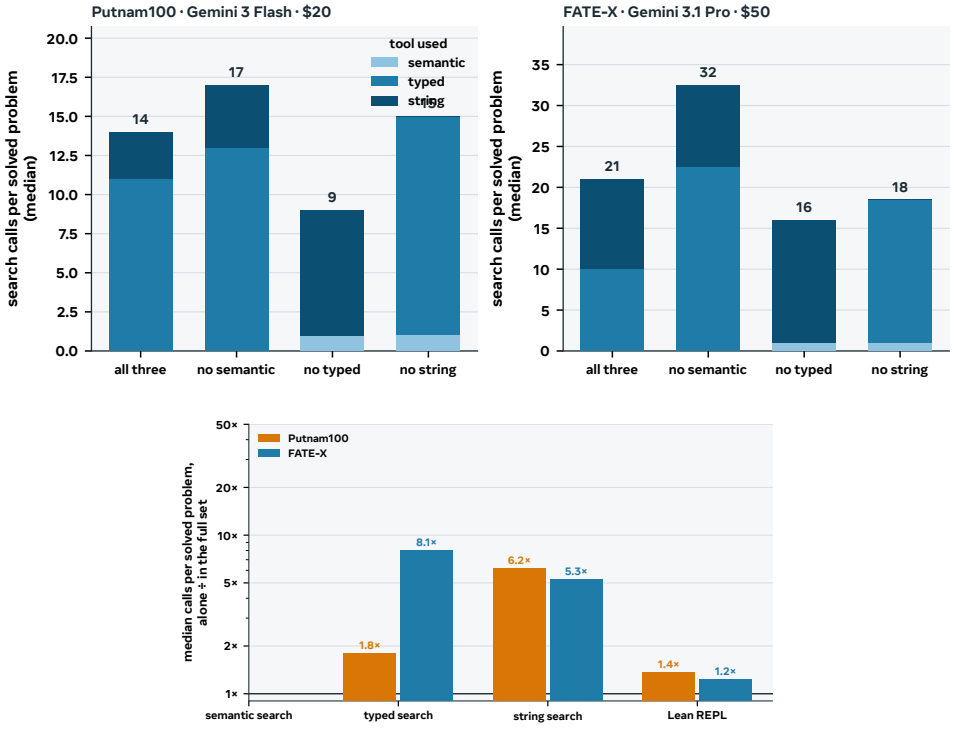

Figure 24: **Tool use when search and execution capabilities change.** Top: median calls to each search tool among problems solved by each configuration; stacked heights sum these tool-specific medians. Bottom: each tool's median call count when it is the only additional capability, divided by its median with all tools available. These comparisons use each configuration's own solved problems.

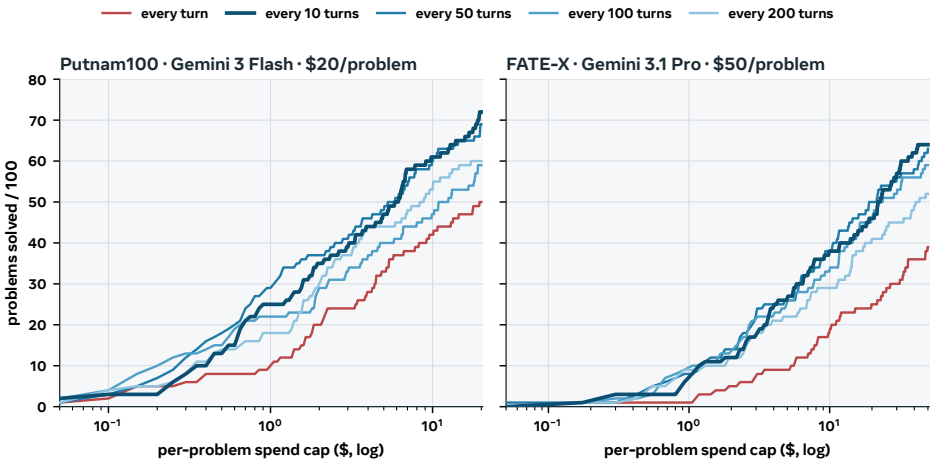

Figure 25: **Solve rates at different budgets and reflection intervals.** Reflection follows every 1, 10, 50, 100, or 200 proof-search turns ($T1$–$T200$). Gemini 3 Flash is evaluated on Putnam100 at up to \$20 per problem (left), and Gemini 3.1 Pro on FATE-X at up to \$50 (right). Reflection every turn produces fewer solves over most observed budgets; intervals of 10 and 50 turns give similar curves.

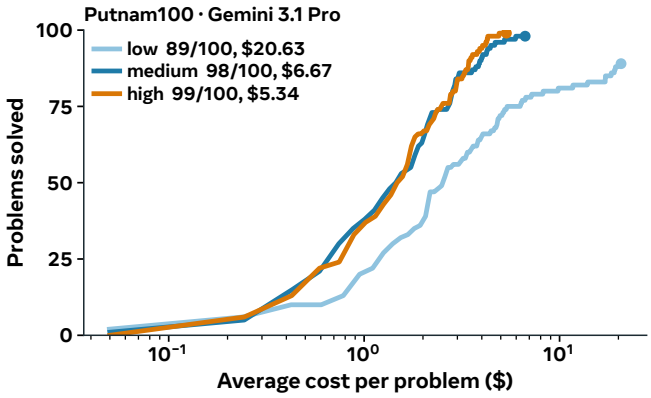

Figure 26: **More reasoning per turn can make test-time scaling more efficient.** Gemini 3.1 Pro on Putnam100 with low, medium, and high reasoning effort. Curves plot solves against mean cost, including failed attempts; legends report final solve counts and mean costs.

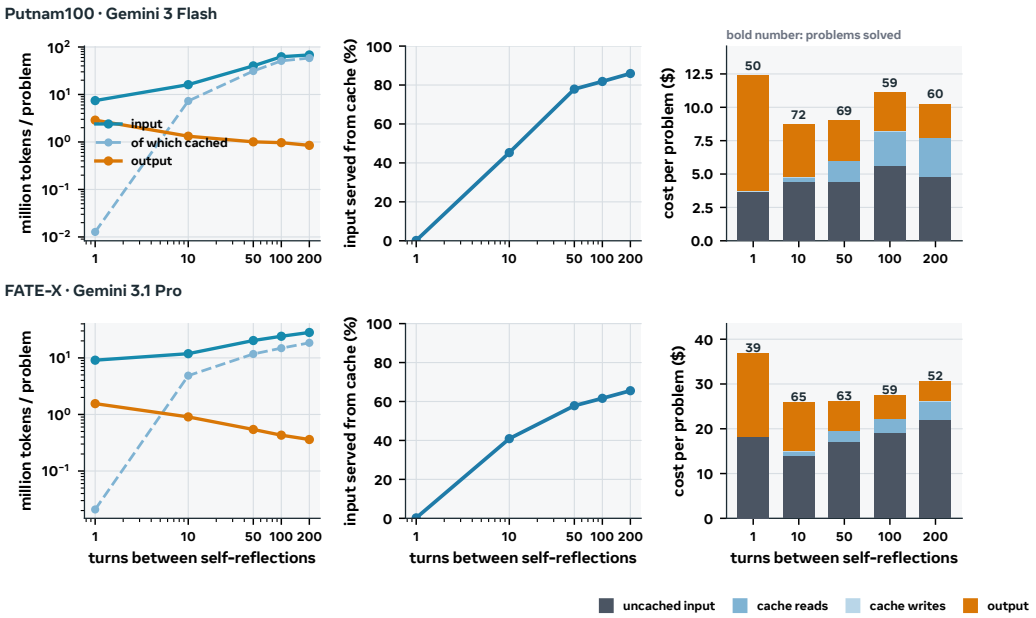

Figure 27: **Token use and cost at different reflection intervals.** Gemini 3 Flash on Putnam100 (top) and Gemini 3.1 Pro on FATE-X (bottom). Left: input, cached input, and output tokens averaged over recorded problems. Middle: the fraction of all input tokens served from cache. Right: mean inference cost per problem, including failed attempts, split by token type; numbers above bars give solve counts. Calls to both the multi-turn LLM and reflection LLM are included. The horizontal axis gives the maximum number of proof-search turns between reflections.

H Solve-Time and Proof-Size Statistics

The figures below cover the four state-of-the-art runs. Across all our runs, the maxima reported in Section 3.1 come from other runs: 1,123 iterations before acceptance on PutnamBench problem `putnam_1992_b3`; 95 active hours for DeepSeek V4 Flash on FATE-X problem 38, accepted at iteration 546; and an accepted file of 7,124 lines for the Borsuk–Ulam theorem.

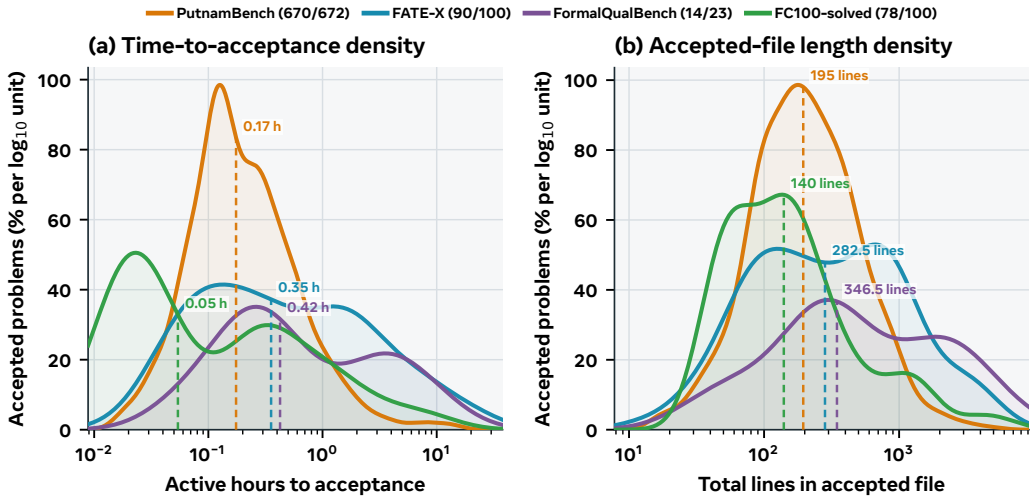

Figure 28: **Time to acceptance and accepted-file length.** The selected runs contain 670/672 accepted PutnamBench problems with GPT-5.5, and 90/100 on FATE-X, 14/23 on FormalQualBench, and 78/100 on FC100-solved with GPT-5.6 Sol: 852 accepted proofs in 895 attempts. Left: active hours to first acceptance. Right: total lines in the accepted Lean file. Densities are estimated in $\log_{10}$ space and normalized so their area in that space equals the benchmark's acceptance percentage. Dashed lines mark medians among accepted proofs.

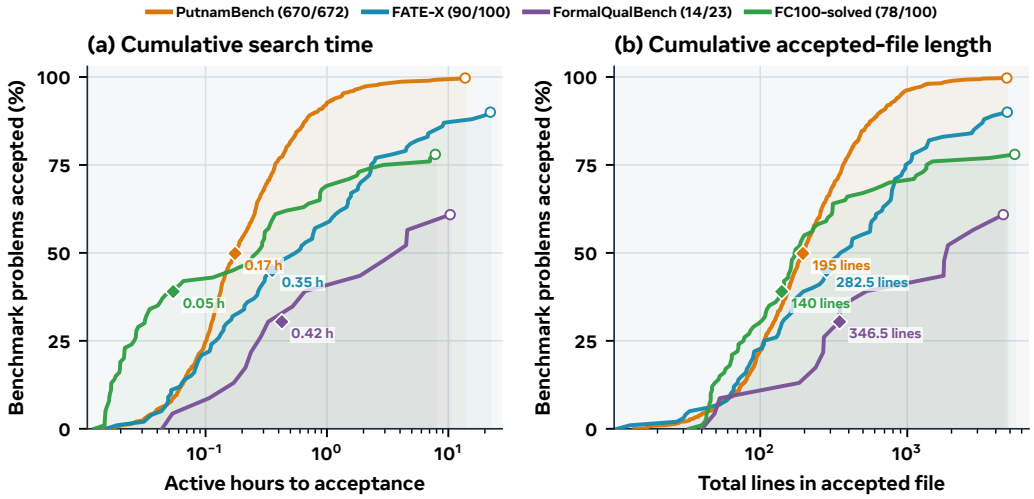

Figure 29: **Cumulative distributions of time to acceptance and accepted-file length.** The runs are those in Figure 28. Counts are divided by all attempted problems, so each curve ends at the run's acceptance percentage. Diamonds mark medians among accepted proofs.

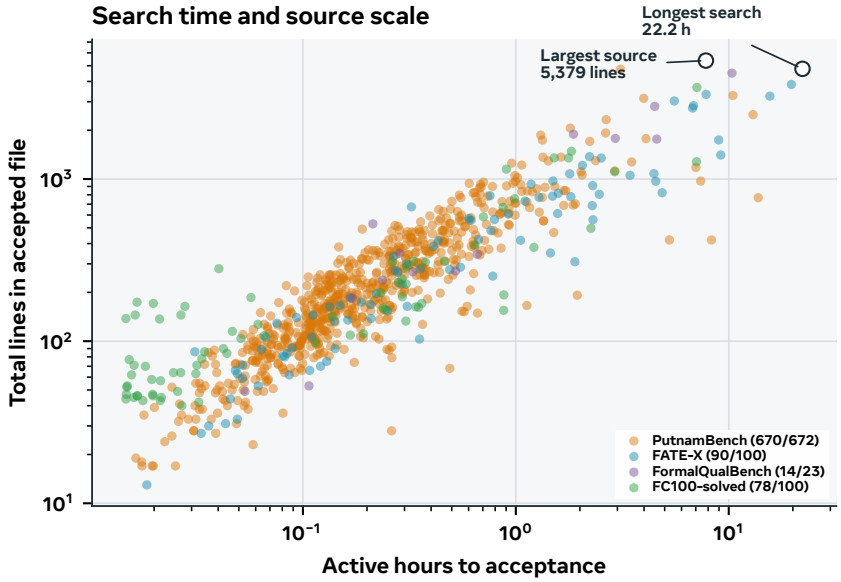

Figure 30: **Search time and total file length across 852 accepted proofs.** GPT-5.5 on PutnamBench; GPT-5.6 Sol on FATE-X, FormalQualBench, and FC100-solved.