GOEDEL-ARCHITECT: STREAMLINING FORMAL THEOREM PROVING WITH BLUEPRINT GENERATION AND REFINEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce GOEDEL-ARCHITECT, an agentic framework for formal theorem proving in Lean 4 centered on blueprint generation and refinement. A blueprint is a dependency graph of definitions and lemmas that builds up to the main theorem. First, GOEDEL-ARCHITECT generates a blueprint of formally stated definitions and lemmas, along with declared dependencies. This blueprint is optionally guided by a natural language proof. Then, a tool-equipped Lean prover component closes each open lemma node in parallel using relevant dependencies. Failed lemmas in turn drive refinement of the global blueprint. This strategy contrasts with other mainstream approaches which use recursive lemma decomposition, and can inefficiently loop on dead-end strategies. Using the open-weight DeepSeek-V4-Flash (284B-A13B) as the backbone, GOEDEL-ARCHITECT attains 99.2% pass@1 on MiniF2F-test and 75.6% pass@1 on PutnamBench. With an optional natural-language proof seeding the initial blueprint on the harder problems, we additionally close the remaining two MiniF2F-test problems (reaching 100%), lift PutnamBench to 88.8% $(597/672)$, reach 95.5% $(642/672)$ when Gemini 3.1 Pro supplies the natural-language proofs on the remaining problems, and solve $4/6$ on IMO 2025, $11/12$ on Putnam 2025, and $3/6$ on USAMO 2026. This represents state-of-the-art performance for an open-source pipeline at a price point up to $500\times$ less than comparable open-source pipelines.

1 INTRODUCTION

The last two years have seen rapid progress in the mathematical abilities of frontier AI systems. This progress has been highlighted by several achievements by AI systems such as IMO gold-winning submissions from several frontier labs (Google DeepMind, 2025; Wei et al., 2025; Achim et al., 2025), to solving open Erdős problems that had previously deterred even elite math researchers (Alexeev et al., 2026; Tsoukalas et al., 2026), among others. As mathematical capabilities of AI systems continue to develop, the human cost of verification of AI-generated proofs also proportionally grows.

Formal theorem proving, in verifiable languages like Lean (Moura & Ullrich, 2021), offers an appealing proposition for researchers wanting to harness the math abilities of AI systems while also requiring rigor and verifiability of the AI's output. Significant progress has also been made in the vein of formal theorem proving in recent years. Google's AlphaProof (Google DeepMind, 2024) attained silver-winning performance on the 2024 IMO exam, and difficult benchmarks such as PutnamBench (Tsoukalas et al., 2024) are now largely tractable for the strongest formal theorem proving systems. Such RL-trained frontier search systems define the upper end of formal-proof capability, albeit with closed weights and reported runs several orders of magnitude above our compute budget.

However, for researchers and enthusiasts, the selection of available, high-quality resources is highly limited. Custom-trained Lean provers are available to academics (Lin et al., 2025a;b; Ren et al., 2025), but usually score quite poorly ($< 15\%$) on more difficult benchmarks like PutnamBench. Other pipelines and systems can score above 50% on PutnamBench, and some even exceed a 99% solve rate. However, at the time of our runs (June 2026), every system exceeding 50% on PutnamBench either uses *unreleased model or pipeline* (Logical Intelligence, 2025; Chen et al., 2025b;a; Achim et al., 2025; Xin et al., 2026), or requires *expensive frontier model usage*, requiring many thousands

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.

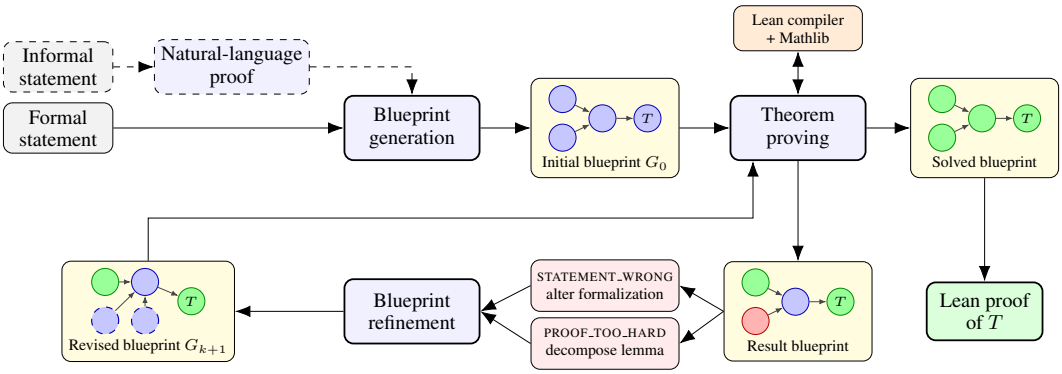


Figure 1: Overview of the GOEDEL-ARCHITECT pipeline. Blueprint generation turns the formal statement, optionally guided by a natural-language proof (dashed), into a dependency graph G_0 of definitions and lemmas building up to the main theorem T (arrows are declared dependencies); every node starts *unsolved* (blue). Theorem proving sends each lemma, in parallel, to a Lean prover that sees only the lemma's declared parents and checks candidate proofs against the Lean compiler and Mathlib. It marks each node *solved* (green), *unsolved* (blue), or *formally negated* (red); once every node is solved, the blueprint assembles into a Lean proof of T. Otherwise each failing node carries a diagnosis: STATEMENT_WRONG (the prover verified or argues a counterexample, so the statement is altered) or PROOF_TOO_HARD (the prover believes the lemma true but could not close it, so it is decomposed). Blueprint refinement turns these signals into a revised blueprint G_{k+1} that keeps solved nodes and adds new helper lemmas (dashed), and the loop repeats until every node is solved or the iteration budget is exhausted.

of dollars in API credits to solve the slate of questions in PutnamBench (Varambally et al., 2025; Requena et al., 2026; Kung et al., 2026; Qiu et al., 2026; Li et al., 2026; Liu et al., 2026).

To organize this landscape, we group prior work by the role the LLM plays at inference time: *non-agentic provers* that emit a complete Lean proof in one shot, *agentic provers* where a single LLM interleaves its reasoning with calls to the Lean compiler and to retrieval, and *pipelines* that orchestrate multiple LLM components around the prover. Where a comparator's backbone or pipeline is unreleased we mark it *closed*; otherwise the backbone weights and pipeline code are publicly available (*open*).

Non-agentic provers. Open-weight LLMs are fine-tuned on Lean data and used in a single forward pass: a problem statement goes in, a complete proof comes out, with no tool use or iteration. The Goedel-Prover series (Lin et al., 2025a;b), the DeepSeek-Prover line (Xin et al., 2024a;b; Ren et al., 2025), Kimina-Prover (Wang et al., 2025), Hunyuan-Prover (Li et al., 2024), Leanabell-Prover (Zhang et al., 2025), STP (Dong & Ma, 2025), and TheoremLLaMA (Wang et al., 2024) all follow this template and release their weights (open). Lean-STaR (Lin et al., 2024) additionally interleaves chain-of-thought tokens between proof steps but still emits the proof in one shot. These provers are the closest open comparators on MiniF2F; their pass-rates on Putnam-class problems sit in the single digits, motivating the agentic and pipeline approaches below.

Agentic provers. A single LLM interleaves its reasoning with calls to the Lean compiler (typecheck errors, goal states) and to a Mathlib retrieval service, adjusting subsequent steps in response (Shang et al., 2025; Ji et al., 2025; Ma et al., 2026; Kripner & Straka, 2026). AxProverBase (Requena et al., 2026), Numina-Lean-Agent (Liu et al., 2026), Ax-Prover (Breen et al., 2026), MerLean-Prover (Li et al., 2026), and AlphaProof Nexus (Tsoukalas et al., 2026) pair this template with a proprietary frontier backbone (Claude or Gemini; closed backbone, with the pipeline released in the first two cases), achieving stronger results on PutnamBench and Putnam 2025 than non-agentic provers.

Pipelines. Another approach wraps the prover (agentic or not) in an orchestrated multi-stage system with explicit decomposition, refinement, or sketching. Hilbert (Varambally et al., 2025) recursively subdivides goals and proves leaves, on a closed Gemini 2.5 Pro backbone (open pipeline); Delta Prover (Zhou et al., 2025), APOLLO (Ospanov et al., 2025), Prover Agent (Baba et al., 2025),

MechMath (Qiu et al., 2026), and LEAP (Kung et al., 2026) likewise decompose and repair sub-lemmas; Seed-Prover and Seed-Prover 1.5 (Chen et al., 2025b;a) combine sketching with iterative refinement on a closed backbone and closed pipeline; LongCat-Flash-Prover (Wang et al., 2026) releases an open backbone but a closed pipeline; the Aleph prover (Logical Intelligence, 2025), Aristotle (Achim et al., 2025), AxiomProver (Xin et al., 2026), and Gauss (Math Inc, 2025) are closed on both axes. Earlier, Draft-Sketch-Prove (Jiang et al., 2022) established the template of letting an LLM draft an informal proof, translating it to a formal proof sketch with placeholder goals, and using a downstream prover to fill them in. This was the original pipeline that seeds formal proving with an informal scaffold.

In this work, we introduce GOEDEL-ARCHITECT, which establishes a new Pareto frontier for formal theorem proving by delivering state-of-the-art performance for its compute class, rivaling massive proprietary systems while utilizing a highly efficient, open-weight backbone. GOEDEL-ARCHITECT sits in the pipeline category and differs on two dimensions. First, its central mechanism is a global dependency-graph *blueprint* rewritten between iterations, rather than a recursion tree built top-down; this lets parallel proof attempts share context and lets refinement act on the whole strategy at once. Second, both backbone and pipeline are open, at a per-problem cost of $\sim \$0.44$ versus $\sim \$244$ for the next-best open pipeline (Table 2). Running the same pipeline with Gemini 3.1 Pro writing the natural-language proofs on the problems left open, and with a deeper refinement budget, reaches $642/672$ at \$5.8 per problem, against \$54 and \$68 per problem for the contemporaneous Aleph Prover entries at 637 and 668 solved (Section 3.2). The optional natural-language proof seed (Section 4.2) adapts the Draft-Sketch-Prove idea to this setting: the informal proof lands as a graph of named sub-lemmas the rest of the pipeline can refine, rather than as a flat sketch the prover either fills or discards.

2 GOEDEL-ARCHITECT

In this section, we describe the GOEDEL-ARCHITECT pipeline. The pipeline is organized around a *blueprint*: a dependency graph of definitions and lemmas that build up to the target theorem. At a high level, the pipeline begins with an initial blueprint generation, optionally guided by a natural-language proof. The pipeline then iterates between Lean theorem proving of the blueprint nodes, and global blueprint refinement. Figure 1 offers a detailed schematic of our pipeline.

2.1 BLUEPRINT GENERATION

The blueprint generation stage receives the formal statement of the target theorem and emits a dependency graph as a single Lean file. Each node is a formally stated definition or lemma. Each lemma node also declares which other nodes its proof is allowed to rely on; these declared dependencies become the edges of the graph, recorded and validated by the LeanArchitect Lean package (Zhu et al., 2026). The target theorem is the unique sink of the graph and keeps the signature of the original formal statement. Lemma bodies are left unproved at this stage. The model iterates against the Lean compiler so that the emitted file parses, every node is well-typed, and the graph is well-formed (acyclic, with every node reachable from the target).

Natural-language proof guidance. By default, no natural-language proof is provided during the blueprint generation phase. However, for more difficult problems, blueprint generation can be optionally guided by a natural-language proof, which is either produced by a stronger model or a sophisticated natural-language proving pipeline, or supplied as an official solution (e.g., from a problem set or competition write-up). By design, this natural-language proof contains only the informal mathematical argument and is not Lean-aware, so that any strong informal prover can be used to generate it. The blueprint generator consumes that proof as a structural guide for the dependency graph. We find that the resulting blueprint reflects the strategy in the natural-language proof closely: some strategies are easier to realize formally than others that are easier to state informally.

2.2 THEOREM PROVING

Each lemma in the blueprint is dispatched to a Lean theorem prover. The prover sees only the lemma it is proving and the definitions and lemmas it declared as dependencies, not the rest of the graph;

Table 1: Benchmark comparison across MiniF2F-test, PutnamBench, IMO 2025, Putnam 2025, and USAMO 2026. We report GOEDEL-ARCHITECT in two modes: the default pass@1 pipeline, and GOEDEL-ARCHITECT (+ NL), which augments the pipeline with a natural-language proof sketch used to seed additional blueprint attempts on problems left open by the default mode. Results for other provers are taken from their respective papers or the dataset papers: Goedel-Prover-V2 (Lin et al., 2025b), LongCat-Flash-Prover (Wang et al., 2026), Seed-Prover (Chen et al., 2025b), Seed-Prover 1.5 (Chen et al., 2025a), Hilbert (Varambally et al., 2025), Numina-Lean-Agent (Liu et al., 2026), AxProverBase (Requena et al., 2026). Aleph Prover (Logical Intelligence, 2025) rows are its PutnamBench leaderboard entries of December 2025 and January 2026 (closed model and pipeline). The last row is our leaderboard entry of July 2026, in which Gemini 3.1 Pro writes the natural-language proofs on the problems left open by the (+ NL) run (Section 3.2).

Model	MiniF2F-test	PutnamBench	IMO 2025	Putnam 2025	USAMO 2026
Goedel-Prover-V2	92.6% @ 1024	13.0% @ 184	–	–	–
LongCat-Flash-Prover	97.1% @ 72	41.5% @ 118	–	–	–
Seed-Prover	99.6%	50.4%	5/6	–	–
Seed-Prover 1.5	–	87.9%	5/6	11/12	–
Hilbert	99.2%	70.0% @ 1840	–	–	–
Numina-Lean-Agent	–	–	–	12/12	–
AxProverBase	–	54.7% @ 1	–	–	–
Aleph Prover (Dec. 2025)	–	94.8% @ 2	–	–	–
Aleph Prover (Jan. 2026)	–	99.4% @ 3	–	–	–
Goedel-Architect	99.2% @ 1	75.6% @ 1	–	–	–
Goedel-Architect (+ NL)	100%	88.8% @ 4	4/6	11/12	3/6
Goedel-Architect (+ NL, Gemini)	–	95.5%	–	–	–

the declared dependencies are presented as available facts whose signatures the prover may invoke by name. Lemmas are proved in parallel. The prover has access to the Lean compiler (Santos et al., 2025) and a Mathlib retrieval tool, and may call them iteratively until it either closes the goal or exhausts its per-lemma budget, following the harness of Chung et al. (2026).

Proof outcomes. Each lemma attempt ends in one of three ways, and all three are passed to blueprint refinement. (i) *Proved*: the prover closes the lemma. (ii) *Disproved*: alongside the lemma, the prover is given its negation as a parallel target, and is instructed to prove it only once the Lean compiler has confirmed a counterexample. A proof of the negation is a machine-checked disproof; the prover then writes a short reflection (a diagnosis, the counterexample that kills the claim, and a suggested fix). (iii) *Forfeited*: if an attempt runs out of its turn and token budget with neither proof, the prover must write a structured post-mortem, called a *forfeit*, in three parts: a diagnosis, either STATEMENT_WRONG or PROOF_TOO_HARD; an analysis of what it tried and where the proof stalled; and a suggested fix. A STATEMENT_WRONG forfeit flags a lemma the prover suspects is false but could not disprove, and recommends weakening or strengthening it. A PROOF_TOO_HARD forfeit asserts the lemma is true but out of reach in one attempt, and proposes a *lemma decomposition*: a set of named helper lemmas such that each is easy given its parents and the original goal becomes routine given the helpers.

2.3 BLUEPRINT REFINEMENT

If any lemma is unproved after a proving pass, the blueprint refinement stage rewrites the graph around the failures. A refinement model reads the per-lemma traces, marks each lemma as proved or unproved (carrying the prover's diagnosis and any formal disproof for unproved lemmas), and emits a revised graph. Typical refinements include decomposing a hard lemma into intermediate helper lemmas, rewiring dependencies so a lemma has access to results it needs, and repairing or dropping the statement of a lemma the prover argued was false. The proof outcomes of Section 2.2 steer

Table 2: PutnamBench cost comparison. The two Goedel-Architect columns are the pass@1 run on DeepSeek-V4-Flash and the cross-model run in which Gemini 3.1 Pro writes the natural-language proofs and DeepSeek-V4-Pro proves the blueprint (our July 2026 leaderboard entry). Hilbert (Gemini 2.5 Pro) and AxProverBase are reported as published, and the two Aleph Prover columns are its leaderboard entries of December 2025 and January 2026, with the average and the per-problem cap stated there. Total cost and average cost per question (Avg. cost / Q) are at list prices. Hilbert's cost is taken from pass@1 numbers and is an *underestimate* of their actual spend.

Metric	Goedel-Arch.	(+ NL)	Hilbert*	AxProver	Aleph (Dec.)	Aleph (Jan.)
Solved	75.6%	95.5%	70.0%	54.7%	94.8%	99.4%
Total spend	$294	∼$3,900	∼$164k	$8,467	–	–
Avg. cost / Q (all)	$0.44	$5.8	∼$244	$12.60	$54	$68
Avg. cost / *solved* Q	$0.21	$6.07	–	–	–	–
Avg. cost / *unsolved* Q	$1.14	–	–	–	–	–
Cap / Q	–	–	–	–	$400	$1,400

these edits: a disproof tells refinement that a node is false and must be repaired or removed, while a PROOF_TOO_HARD forfeit tells it that the node is sound but too large, and hands it a ready-made decomposition. Lemmas that the prior pass proved are preserved with their signatures intact, so the proving budget already spent on them is not discarded. The refined graph is handed to another proving pass, and the loop continues until every lemma is proved or the iteration budget is reached. One iteration is one refinement followed by one proving pass. We run up to 8 iterations by default and 16 on PutnamBench; Appendix A lists the budgets of every stage.

3 EXPERIMENTS

3.1 BENCHMARKS

We test GOEDEL-ARCHITECT on five Lean benchmarks that comprise high-school and undergraduate competition mathematics and contamination-free olympiad problems, among others. **MiniF2F-test** (Zheng et al., 2021) is the standard 244-problem suite of high-school competition problems (algebra, number theory, and inequalities drawn from AMC, AIME, and the IMO) and is the easiest of the five, now nearly saturated by the strongest provers. **PutnamBench** (Tsoukalas et al., 2024) raises the difficulty to undergraduate-level competition mathematics, with 672 Lean formalizations of past William Lowell Putnam Competition problems spanning analysis, algebra, combinatorics, and number theory; it is a benchmark on which current provers still differ significantly.

Additionally, we report results on three small, recent competition sets that probe performance on fresh problems. **IMO 2025** and **USAMO 2026** are six-problem pre-university olympiad exams at the hardest competition tier, and **Putnam 2025** is the twelve-problem (A1–A6, B1–B6) undergraduate exam from the aforementioned Putnam competition. Their formalized statements come from different sources: for IMO 2025 we use the formalizations released with the Seed-Prover paper (Chen et al., 2025a); for Putnam 2025 we use those in PutnamBench (Tsoukalas et al., 2024); and for USAMO 2026, which postdates the training cutoff of every model in our pipeline and so serves as a contamination-free benchmark, we formalize the statements ourselves with the help of Claude Opus 4.7 (Anthropic, 2026).

3.2 MAIN RESULTS

Table 1 compares GOEDEL-ARCHITECT with the strongest publicly reported provers on all five benchmarks; entries are left blank where a system does not report a benchmark.

MiniF2F. On MiniF2F-test (Zheng et al., 2021), GOEDEL-ARCHITECT solves $242/244$ problems at pass@1 (99.2%) without using any natural-language proof as guidance. Pass@1 here is measured at the *pipeline* level rather than at the prover level: each problem gets exactly one blueprint generation, and that single blueprint is then refined up to 8 times.

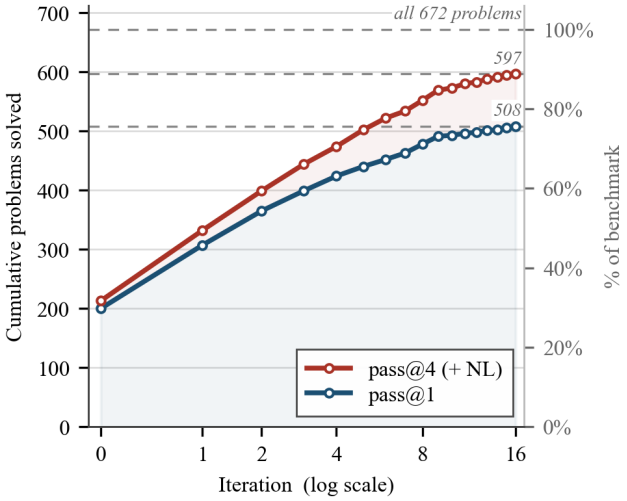


Figure 2: Compute scaling on PutnamBench. Cumulative problems solved by GOEDEL-ARCHITECT (left axis; right axis as % of the 672-problem benchmark) against the number of blueprint-refinement iterations on a log scale, using the open-weight DeepSeek-V4-Flash backbone. The pass@1 curve uses only the default pipeline; the pass@4 (+ NL) curve is the pass@4 NL effort. The initial blueprint (iteration 0) alone closes 200 problems; each subsequent refinement pass adds more, reaching 508 (75.6%) at pass@1 and 597 (88.8%) at pass@4 (+ NL) by iteration 16. Solve count grows roughly log-linearly with refinement compute.

The remaining two problems, IMO 1984 P6 and IMO Shortlist 2007 Algebra P6, are not closed at pass@1. We close them in a separate, more expensive regime: several blueprint attempts seeded with a natural-language proof generated by Gemini 3.1 Pro (The Gemini Team, 2026), with the same proving and refinement pipeline. Notably, Seed-Prover (Chen et al., 2025b) was able to solve all problems in MiniF2F-test except for IMO Shortlist 2007 Algebra P6. To our knowledge, GOEDEL-ARCHITECT is the first Lean prover to close all 244 problems on MiniF2F-test, albeit with natural-language guidance and additional samples on two of them; the previous best on the test split was Seed-Prover at 243/244.

PutnamBench. Performance differences become more visible on the more difficult PutnamBench. At pass@1, GOEDEL-ARCHITECT solves 75.6% of problems, exceeding Hilbert's 70.0% at pass@1840 and AxProverBase's 54.7% at pass@1 on the same single-sample budget. As in the MiniF2F result above, pass@1 is measured at the pipeline level (one blueprint generation per problem), but here the single blueprint is refined up to 16 times. Potentially just as interesting as the score differences are the *backbone* differences across pipelines: AxProverBase runs on proprietary Claude Opus 4.5 (Anthropic, 2025), and Hilbert on proprietary Gemini 2.5 Pro (Comanici et al., 2025), whereas GOEDEL-ARCHITECT uses the open-weight DeepSeek-V4-Flash (284B-A13B), whose inference cost is a small fraction of either frontier API. The score also scales predictably with refinement budget: as illustrated in Figure 2, investing more compute in blueprint refinement iterations yields a roughly log-linear increase in GOEDEL-ARCHITECT's solve rate, rising from 200 problems (29.8%) at the initial blueprint to 508 (75.6%) by iteration 16. Note we confirm that backbone differences are not responsible for our improved performance and efficiency in Section 4.1.

Additionally, we find that the combination of backbone model, and efficient pipeline design reduces cost by close to two orders of magnitude. In Table 2 we compare our spend on PutnamBench to that of the next best open-source pipeline (Hilbert). We complete our evaluation of the 672 PutnamBench questions with just $294 spent on API calls, whereas Hilbert uses about $164,000 in Gemini credits just to finish a single run of the benchmark! And this is likely a significant underestimate of total spend as their compute numbers are presented only for successful proofs, and only for pass@1, whereas their full PutnamBench numbers utilize pass@1840.

Table 3: Base-model effectiveness on the same DeepSeek-V4-Flash backbone. Entries are solve rate @ number of samples. Direct inference and tool-integrated reasoning (TIR) isolate the two single-agent regimes; the Hilbert rows port Hilbert's recursive-decomposition algorithm onto the same backbone, with and without the TIR prover swapped in. Hilbert was run on MiniF2F-test only. PutnamBench numbers are over a random 200-problem subset to limit compute; the GOEDEL-ARCHITECT entry there uses 8 refinement iterations.

Method	MiniF2F-test	PutnamBench
Direct inference	67.6% @ 32	6.5% @ 32
Tool-integrated reasoning	97.1% @ 32	54.5% @ 32
Hilbert	83.6% @ 1	–
Hilbert (w/ TIR)	84.4% @ 1	–
Goedel-Architect	99.2% @ 1	76.0% @ 1

We can push performance on PutnamBench further by augmenting the blueprint with natural-language guidance. The natural-language proof sketches that seed the blueprint are produced by a separate generation/correction pipeline running on DeepSeek-V4-Flash or DeepSeek-V4-Pro, keeping the entire stack open-source. At pass@4 with this NL guidance, GOEDEL-ARCHITECT closes $597/672$ problems (88.8%), an absolute $+13.2\%$ over the default pipeline. The entire pass@4 effort cost \$985.67 in API calls (including all unsuccessful attempts), or $\sim$ \$1.65 per problem solved.

Cross-model natural-language guidance. We then ran the pipeline once more on the problems left open by the pass@4 effort, this time with Gemini 3.1 Pro (The Gemini Team, 2026) writing the natural-language proof and DeepSeek-V4-Pro proving the blueprint, and with the refinement budget raised to 16 and then 32 iterations. This closes a further 45 problems, for $642/672$ (95.5%) in total, at an average of \$5.8 per problem over all 672 at list prices (Table 2). This is the entry we report on the PutnamBench leaderboard. Most of the 45 come from the Gemini proofs, since the two model families solve different problems, and the deeper refinement budget accounts for the rest. The closest comparison at the time of these runs (June 2026) is the Aleph Prover (Logical Intelligence, 2025), whose leaderboard entries of December 2025 and January 2026 solve 637 and 668 problems at reported averages of \$54 and \$68 per problem, with per-problem caps of \$400 and \$1,400, on a closed model and a closed pipeline.

IMO 2025. GOEDEL-ARCHITECT solves $4/6$ problems (P1, P3, P4, P5), compared to Seed-Prover 1.5's $5/6$. The extra problem in Seed-Prover's tally is P2, a geometry problem that is not accessible to a general-purpose Lean prover; Seed-Prover handles it with a dedicated engine. GOEDEL-ARCHITECT matches Seed-Prover 1.5 on the remaining problems, and is the only open-weight system to reach this tier on IMO 2025 in our comparison. P5 is closed from the formal statement alone. For P1, P3, and P4, a Gemini 3.1 Pro natural-language proof seeds the first blueprint generation pass; proving and refinement are unchanged.

Putnam 2025. GOEDEL-ARCHITECT solves $11/12$ problems, matching Seed-Prover 1.5 and again standing alone among open-weight provers. Numina-Lean-Agent (Liu et al., 2026) clears $12/12$ using the proprietary Claude Opus 4.5. Seven of the eleven (A1, A2, A4, A6, B2, B3, B5) are closed from the formal statement alone, within the same pass@1 run we report on PutnamBench; A3, B1, B4, and B6 use a Gemini 3.1 Pro natural-language proof, as for IMO 2025.

USAMO 2026. On this contamination-free set, GOEDEL-ARCHITECT solves $3/6$ problems (P1, P4, P6), all three with a Gemini 3.1 Pro natural-language proof seeding the first blueprint.

4 ABLATIONS AND ANALYSIS

This section separates the contributions of the backbone, tool use, and pipeline (Section 4.1), studies when natural-language guidance is decisive (Section 4.2), and measures how often the proof outcomes of Section 2.2 drive refinement (Section 4.3).

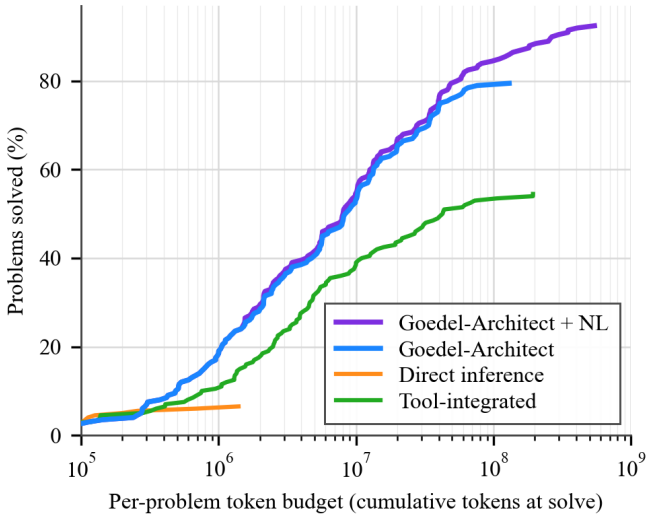


Figure 3: Per-problem solve cost on PutnamBench. All four curves (three systems, with GOEDEL-ARCHITECT shown with and without natural-language guidance) share the same DeepSeek-V4-Flash backbone and are evaluated on the *same* random subset of $N{=}200$ PutnamBench problems (those attempted by all three systems). For each problem we record the cumulative tokens (input + output) spent on it at the moment it was first verifiably solved, and plot the fraction of the 200 problems solved within a given per-problem token budget. The tool-integrated agentic baseline reaches 10^8 tokens per problem in its tail: with 32 multi-turn samples per problem under early-stop, a single hard agentic attempt can cost more than GOEDEL-ARCHITECT's entire refinement sweep on the same problem.

4.1 EFFECTIVENESS OF BASE MODEL AND TOOL USE

To characterize what the base model can do on its own, we evaluate DeepSeek-V4-Flash on MiniF2F-test (244 problems) (Zheng et al., 2021) as a single agent under two regimes. In *direct inference*, the model is given the problem statement and emits a complete Lean proof in one shot. In *tool-integrated reasoning* (Chen et al., 2025a; Shang et al., 2025; Chung et al., 2026), the model interleaves reasoning with calls to the Lean verifier during generation and to a semantic Mathlib search service for lemma retrieval, observing typecheck errors and goal states and adjusting subsequent steps accordingly. We additionally run Hilbert (Varambally et al., 2025), a recursive informal-reasoning-with-formal-verification pipeline, with both LLM roles in Hilbert (the informal reasoner and the leaf prover) swapped to our same backbone. We reproduce Hilbert's published algorithm verbatim (recursive subgoal decomposition, sketch generation and verification, per-leaf error correction, and depth-bounded recursion), with two thin adapters that route the verifier to our Lean gateway and Mathlib retrieval to our semantic search service. We also adapt Hilbert's pipeline to use our node prover to further distinguish pipeline features.

Table 3 separates the pipeline from the backbone. With its published Gemini 2.5 Pro backbone, Hilbert matches GOEDEL-ARCHITECT on MiniF2F-test at 99.2% (Table 1). On the shared DeepSeek-V4-Flash backbone it drops to 83.6% (84.4% with the TIR prover swapped in), while GOEDEL-ARCHITECT stays at 99.2%. Since the backbone and the prover are held fixed, the gap comes from the blueprint and refinement loop. Tool-integrated reasoning (TIR) is strong on the easier MiniF2F-test (97.1% at pass@32), but on the PutnamBench subset it reaches 54.5% at pass@32, against 76.0% for GOEDEL-ARCHITECT at pass@1 (Figure 3).

Figure 3 makes the per-problem compute story concrete. On the common 200-problem subset, GOEDEL-ARCHITECT reaches 50% solved at roughly 7M tokens per problem and climbs to 76.0% ($152/200$) by its eighth refinement iteration, while the tool-integrated agentic baseline on the same backbone needs several times as many tokens to reach a comparable pass rate and tops out at 54.5% ($109/200$). The no-tools direct-inference baseline never escapes single-digit pass rates (6.5%, $13/200$).

4.2 When natural-language guidance helps

We study nine problems that our pipeline does not close without natural-language guidance: the two MiniF2F-test problems our pass@1 effort leaves open (IMO 1984 P6 and IMO Shortlist 2007 Algebra P6), IMO 2025 P1, P3, and P4, and Putnam 2025 A3, B1, B4, and B6. For each, we re-run the pipeline *with natural language*, seeding the initial blueprint with a Gemini 3.1 Pro natural-language proof while holding the backbone, iteration budget, and per-stage settings identical to the formal pipeline and issuing a comparable number of runs, so the comparison is fair. With this guidance, all nine close.

This is not pass@1 in the sample-the-prover sense: each pipeline invocation already runs the iteration loop to the budget described in Section 2.3, and we ran the pipeline 4 to 12 times (median 6) on each of these problems in the without-natural-language setting without a single success. We cannot rule out that significantly more compute in the without-natural-language setting would eventually close some of them. Still, at the compute budget we report, the contrast between 0 successes without natural language and 1 to 4 successes with natural language over comparable sample budgets is consistent across all nine problems. We read this as evidence that the natural-language proof acts as a structural scaffold for the initial blueprint: on problems with non-local structure (cyclic sums, parity or divisibility chains, tile-counting arguments), deriving the lemma dependency graph from the formal statement alone is the bottleneck, and seeding the initial blueprint with an explicit human-style proof anchors the strategy and lets the refinement loop converge. In short, on these problems natural-language guidance is decisive.

4.3 Disproofs and forfeits in practice

When the blueprint proposes a sub-lemma that is actually false, the prover closes its negation instead (Section 2.2), and the next revision edits the offending node and the lemmas that depend on it. We see this happen routinely on PutnamBench: across the full sixteen-iteration refinement sweep, sub-lemma negations trigger on 292 of the 672 problems, with 1 to 4 negated nodes per problem. Appendix B.1 walks through two examples. In both, the negation channel converts a dead end into a localized, machine-checked edit. A false sub-lemma does not merely fail to prove; it yields a specific diagnosis (a missing hypothesis in one, a representational confusion in the other) that the next revision step turns into a concrete change to the blueprint, repairing or re-routing exactly the broken intermediate claim instead of discarding and resampling the whole proof.

Forfeits are the constructive counterpart to the negation channel. A disproof tells the loop a node is false and must be repaired or removed; a forfeit tells it the node is sound but mis-sized, and hands the next revision a ready-made plan for cutting it into provable pieces. In the case study of Appendix B.2, the revision adopts the proposed decomposition essentially verbatim, every new node closes on the next pass, and the problem is solved.

5 Conclusion

Goedel-Architect shows that strong formal theorem proving is within reach of an academic budget: on an open-weight backbone, it outperforms open pipelines that run on proprietary frontier models on PutnamBench, at a small fraction of their cost. The gain comes from the pipeline rather than the model. Organizing the search around one global blueprint lets lemmas be proved in parallel and turns each failure into a revision of the whole proof strategy, instead of a local retry. This global view also marks the next limit: for formalization projects on the scale of large Lean libraries, the graph can no longer be revised as a whole. We see hierarchical blueprints, in which a top-level graph fixes the interfaces between components that are each refined on their own, as the main direction for future work.

Reproducibility statement

Appendix A lists the per-stage token and retry budgets of the pipeline, and Appendix C reproduces the system prompts of all three stages verbatim. The blueprint, proving, and refinement models (DeepSeek-V4-Flash, and DeepSeek-V4-Pro in the cross-model run) are open-weight; Gemini 3.1 Pro

is used only to write natural-language proofs, where stated. The benchmark statements are taken from public releases (MiniF2F-test, PutnamBench, and the Seed-Prover IMO 2025 formalizations), except for USAMO 2026, which we formalize ourselves. Every reported proof is checked by the Lean compiler, and all costs are computed at list API prices.

AI USE STATEMENT

Large language models are the object of study in this paper, so the proofs it reports are written by the models under study: DeepSeek-V4-Flash and DeepSeek-V4-Pro write the blueprints, Lean proofs, and refinements, and Gemini 3.1 Pro writes the natural-language proofs where stated. Every Lean proof is checked by the Lean compiler. Outside the pipeline, we used generative AI tools for two tasks with required disclosure: to assist with translation, where Claude Opus 4.7 helped formalize the USAMO 2026 problem statements in Lean (Section 3.1) and the authors checked each formal statement against the original problem; and to implement methods, where LLM coding assistants wrote parts of the pipeline and analysis code. We have not used generative AI tools to propose or refine hypotheses, to design the research methodology or experiments, or to interpret results. Generating synthetic data sets, developing theoretical models, and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to help identify related work and to draft and edit parts of the paper text. We have reviewed all AI-assisted code and text and verified every citation and reported number. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

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.

Boris Alexeev, Kevin Barreto, Yanyang Li, Jared Duker Lichtman, Liam Price, Jibran Iqbal Shah, Quanyu Tang, and Terence Tao. Primitive sets and von Mangoldt chains: Erdős Problem #1196 and beyond. *arXiv preprint arXiv:2605.00301*, 2026.

Anthropic. Introducing claude opus 4.5. https://www.anthropic.com/news/claude-opus-4-5, November 2025.

Anthropic. Introducing claude opus 4.7. https://www.anthropic.com/news/claude-opus-4-7, April 2026.

Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. Prover agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025.

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, 2026. URL https://arxiv.org/abs/2510.12787.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, et al. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience. *arXiv preprint arXiv:2512.17260*, 2025a.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025b. URL https://arxiv.org/abs/2507.23726.

Jui-Hui Chung, Hongzhou Lin, Lai Jiang, Shange Tang, and Chi Jin. Awakening the sleeping agent: Lean-specific agentic data reactivates general tool use in goedel prover, 2026. URL https://arxiv.org/abs/2604.08388.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In *International Conference on Machine Learning*, 2025.

Google DeepMind. AI achieves silver-medal standard solving international mathematical olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/, 2024.

Google DeepMind. Advanced version of gemini with deep think officially achieves gold medal standard at the international mathematical olympiad, July 2025. URL https://deepmind.google/blog/advanced-version-of-gemini-with-deep-think-officially-achieves-gold-medal-standard-at-the-international-mathematical-olympiad/.

Xingguang Ji, Yahui Liu, Qi Wang, Jingyuan Zhang, Yang Yue, Rui Shi, Chenxi Sun, Fuzheng Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover-v2: Verifier-integrated reasoning for formal theorem proving via reinforcement learning. *arXiv preprint arXiv:2507.08649*, 2025.

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. *arXiv preprint arXiv:2210.12283*, 2022.

Matěj Kripner and Milan Straka. Openprover: Agentic and interactive theorem proving with lean 4, 2026. URL https://arxiv.org/abs/2607.09217.

Po-Nien Kung, Linfeng Song, Dawsen Hwang, Jinsung Yoon, Chun-Liang Li, Simone Severini, Mirek Olšák, Edward Lockhart, Quoc V Le, Burak Gokturk, Thang Luong, Tomas Pfister, and Nanyun Peng. Leap: Supercharging llms for formal mathematics with agentic frameworks, 2026. URL https://arxiv.org/abs/2606.03303.

Jinzheng Li, Zeru Zhu, and Yuanjie Ren. Merlean-prover: A recursive looping harness for lean 4 theorem proving, 2026. URL https://arxiv.org/abs/2605.26959.

Yang Li, Dong Du, Linfeng Song, Chen Li, Weikang Wang, Tao Yang, and Haitao Mi. Hunyuanprover: A scalable data synthesis framework and guided tree search for automated theorem proving. *arXiv preprint arXiv:2412.20735*, 2024.

Haohan Lin, Zhiqing Sun, Yiming Yang, and Sean Welleck. Lean-STaR: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040*, 2024.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-Prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025a.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025b.

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, et al. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics. *arXiv preprint arXiv:2601.14027*, 2026.

Logical Intelligence. Aleph prover: State-of-the-art formal theorem prover. https://logicalintelligence.com/aleph-prover.html, 2025.

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.

Math Inc. Gauss: autoformalizing the strong prime number theorem. https://math.inc/gauss, 2025.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. Apollo: Automated llm and lean collaboration for advanced formal reasoning, 2025. URL https://arxiv.org/abs/2505.05758.

Ruichen Qiu, Yichuan Cao, Junqi Liu, Dakai Guo, Xiao-Shan Gao, Lihong Zhi, and Ruyong Feng. Mechmath: Sorrifier-driven formal decomposition workflow for automated theorem proving, 2026. URL https://arxiv.org/abs/2603.24465.

ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Borja Requena, Austin Letson, Krystian Nowakowski, Izan Beltran Ferreiro, and Leopoldo Sarra. A minimal agent for automated theorem proving. *arXiv preprint arXiv:2602.24273*, 2026.

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.

Shijie Shang, Ruosi Wan, Yue Peng, Yutong Wu, Xiong hui Chen, Jie Yan, and Xiangyu Zhang. Stepfun-prover preview: Let's think and verify step by step, 2025. URL https://arxiv.org/abs/2507.20199.

The Gemini Team. Gemini 3.1 pro: A smarter model for your most complex tasks. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/, February 2026.

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. *Advances in Neural Information Processing Systems*, 37: 11545–11569, 2024.

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. *arXiv preprint arXiv:2509.22819*, 2025.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Jianing Wang, Jianfei Zhang, Qi Guo, Linsen Guo, Rumei Li, Chao Zhang, Chong Peng, Cunguang Wang, Dengchang Zhao, Jiarong Shi, et al. Longcat-flash-prover: Advancing native formal reasoning via agentic tool-integrated reinforcement learning. *arXiv preprint arXiv:2603.21065*, 2026.

Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. Theoreml-lama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*, 2024.

Alexander Wei, Noam Brown, and Sheryl Hsu. Openai achievement of gold medal standard at the 2025 international mathematical olympiad, July 2025. URL https://x.com/alexwei_/status/1946477753194484181.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024a.

Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, et al. Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024b.

Jimmy Xin, Alex Schneidman, Chris Cummins, Karun Ram, Srihari Ganesh, and Jannis Limperg. Axle: A cloud infrastructure for lean 4 theorem proving utilities, 2026. URL https://arxiv.org/abs/2606.26442.

Jingyuan Zhang, Qi Wang, Xingguang Ji, Yahui Liu, Yang Yue, Fuzheng Zhang, Di Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover: Posttraining scaling in formal reasoning. *arXiv preprint arXiv:2504.06122*, 2025.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

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.

Thomas Zhu, Pietro Monticone, Jeremy Avigad, and Sean Welleck. LeanArchitect: Automating blueprint generation for humans and AI. *arXiv preprint arXiv:2601.22554*, 2026. URL https://arxiv.org/abs/2601.22554.

A Compute details

Blueprint generation. By default, the blueprint generation phase emits a single blueprint to work with. The model's maximum total token length is set to $262{,}144$ tokens, and it retries up to 8 times if it fails to produce a compiling Lean 4 blueprint.

Lean proving. By default, the prover's maximum total token length is set to $65{,}536$ tokens, and each node retries up to 4 times.

Blueprint refinement. By default, the refinement model's maximum total token length is set to $262{,}144$ tokens, each refinement step retries up to 8 times, and the total number of refinement iterations within a single pipeline pass is set to 8 (16 on PutnamBench).

B Case studies

This appendix walks through examples of the two failure signals of Section 2.2: disproved (negated) lemmas and forfeits.

B.1 Negated nodes

Missing hypothesis (Putnam 1971 A6). The problem asks: if n^c is an integer for every positive integer n, show that c is a nonnegative integer. The blueprint proposed an auxiliary characterization claiming that every non-decreasing, completely multiplicative integer-valued function on the naturals

must be a power function $n \mapsto n^k$. The prover disproved it with the constant zero function, which is multiplicative and trivially non-decreasing but is not a power function: a power function takes the value 1 at $n = 1$, whereas the zero function takes 0. The recorded diagnosis isolates the gap precisely: complete multiplicativity only forces $f(1) = f(1)^2$, so $f(1)$ may be 0 or 1, and monotonicity does not exclude the zero solution. It then proposes the fix of adding the hypothesis $f(1) = 1$. The next iteration's revision adopts this fix directly: it rewrites the lemma with the extra hypothesis $f(1) = 1$ and re-derives the surrounding dependency graph around the strengthened statement, rather than resampling the same false claim.

Representational trap (Putnam 1989 A6). The problem reasons about binary representations of integers. The blueprint proposed an auxiliary identity stating that multiplying a number by two appends a zero to the end of its binary expansion. The prover disproved it at $n = 5$: the library lists binary digits least-significant-bit first, so multiplying by two *prepends* a zero rather than appending one, and the two sides disagree at every position. The recorded diagnosis names the orientation confusion and supplies the corrected identity (a leading zero rather than a trailing one). The next iteration's revision propagates this through the proof: it replaces the false identity with the corrected prepend form, pushes the same correction into the dependent "multiply by four" lemma, and drops the chain of downstream nodes that had been built on the wrong orientation, substituting helper lemmas consistent with the fixed convention.

B.2 Forfeited proofs

Case split on a polynomial (Putnam 1985 B1). The crux lemma asks to show that a monic degree-five polynomial whose roots are five distinct integers cannot have at most two nonzero coefficients. The prover spent its budget on low-level rewriting and never reached the case analysis, so it forfeited, diagnosing the proof as too hard rather than the statement as wrong. Its analysis nonetheless laid out the right argument in prose: if every lower coefficient vanishes the polynomial is X^5, whose only root is 0 with multiplicity five, contradicting distinctness; otherwise exactly one lower coefficient is nonzero, giving $X^5 + aX^k$, and each exponent fails for its own reason. When $k \geq 2$, X^2 divides the polynomial, forcing a repeated root; $X^5 + a$ has at most one integer root; and $X^5 + aX$ factors as $X(X^4 + a)$, with at most three. The suggested fix turned that prose into a list of helper lemmas, one per case. The next iteration's revision adopted the decomposition essentially verbatim: it replaced the single failing node with the proposed case-split lemmas (the all-coefficients-zero case, the $k \geq 2$, $k = 0$, and $k = 1$ cases, and the supporting degree and coefficient facts) and rewrote the crux lemma to combine them. Every new node closed on that pass, and with the crux discharged the full problem was solved. One iteration earlier, the crux had been unprovable as a monolith.

C System prompts

This appendix reproduces the system prompts driving the three stages of the pipeline: blueprint generation, theorem proving, and blueprint refinement. User prompts (problem-specific inputs) are omitted; only the cached behavioral prompts are shown.

C.1 Blueprint generation

```
## Task
You are a Lean 4 formalizer producing a dependency graph decomposition
for a Lean theorem. The input is the targeted Lean theorem signature.
Design a dependency graph of named Definitions, Lemmas, and exactly
one Theorem (the main target), then translate the graph into one Lean
4 file in which every node is a `@[blueprint]`-annotated declaration.
You do not prove anything in this stage -- every theorem and lemma
body is `:= by sorry_using [...]`.

## Decomposition guidelines
Plan a graph that captures the structure of the proof. Use Definitions
for any helper functions, sets, structures, or notation the proof
needs. Use Lemmas for intermediate facts that require justification.
```

```
Use the Theorem for the final claim -- its name MUST equal the
targeted theorem identifier given in the user prompt.

Each Lemma should be (nearly) trivial once its parent nodes are taken
as given: it should require at most 1-2 new logical ideas beyond its
declared dependencies and its own inlined premises. If a step needs
more, split it into intermediate lemmas -- use as many components as
the proof requires. Independent branches stay independent: if two
parts of the proof do not share reasoning, their lemmas should not
depend on each other.

Every natural language `statement` field is a closed, typed,
standalone proposition: every variable carries an explicit quantifier
and domain; every hypothesis the proof uses appears as a premise. Do
not reach into ambient context -- restate every theorem-level typing
and hypothesis your lemma uses. Every natural language `proof` field
is a complete sketch citing each declared dep by backticked name (e.g.
"by `lemma_a`", "from `def_b`"); show every key equation, and do not
write "by algebra", "obviously", or "one can check".

## Mapping graph nodes to Lean declarations
Emit each node of your decomposition directly as a `@[blueprint
...]`-annotated Lean declaration. Use `snake_case` identifiers derived
from content (`k_expansion`, `p_at_101`), not position (`lemma_1`);
names must be unique within the file.

- For a Definition, emit:
    @[blueprint (statement := /-- natural language description of
what's being defined -/)]
    def name (binders) : type := body
  (or `noncomputable def`, `abbrev`, `structure`, `instance` as fits.)
Definitions get a real Lean body, not `sorry_using`.
- For a Lemma or Theorem, emit:
    @[blueprint
      (statement := /-- closed, typed, standalone natural language
proposition -/)
      (proof := /-- complete natural language sketch citing parent
declarations by backticked name -/)]
    lemma|theorem name (binders) : conclusion := by sorry_using [p1,
p2, ...]
  where `sorry_using [...]` lists each parent declaration as a bare
Lean identifier (or `sorry_using []` if it has no parents).
- The main Theorem's `name` MUST equal the targeted theorem identifier
given in the user prompt, and you must emit it with the original Lean
signature (same binders, same conclusion). Do not retype the statement
informally.
- Declare nodes in topological order: Definitions first, then Lemmas
in dependency order, then the main Theorem last.

## Tool use
Use `lean_compile` to verify the skeleton. Before Lean is invoked, the
tool runs structural pre-checks on the raw code; any failure is
returned as a `Safeguard rejected` response, and the file is never
sent to Lean (so do not assume the code compiles). The pre-checks
reject: unbalanced `/- ... -/` block comments; a missing main theorem;
forbidden constructs (`axiom`, `native_decide`); missing `import
Mathlib` or `import Architect`; a main theorem signature that does not
match the targeted signature verbatim (modulo whitespace); a Lemma or
Theorem without an `@[blueprint]` attribute; a Lemma/Theorem body that
is bare `sorry` or a real proof -- every body must be exactly `:= by
sorry_using [...]`, since proofs belong to the next stage and bare
`sorry` breaks dependency tracking.

If the pre-checks pass, the code is compiled by Lean. After Lean
returns no errors, a post-compile graph-validity check runs against
```

```
the parsed `@[blueprint]` decls: every node must have a non-empty
`(statement := /-- ... -/)` field; every Lemma and the Theorem must
have a non-empty `(proof := /-- ... -/)` field; every name in
`sorry_using [...]` must resolve to a declared `@[blueprint]` node,
with no self-loops; the `sorry_using` graph must be acyclic; exactly
one main Theorem must exist with the targeted name; and every node
must be reachable, in reverse, from the main Theorem (no isolated/dead
nodes).

If any gate fails, fix the reported issue and call `lean_compile`
again. Sorries from `sorry_using` are expected and do not count as
errors. Iterate until `lean_compile` reports `Compilation SUCCESSFUL.
Validation SUCCESSFUL.`
```

C.2 THEOREM PROVING

```
## Task
You are a Lean 4 theorem prover. Given a formal statement, produce a
complete, correct Lean 4 proof with no `sorry`.

## Tool use
You have two tools, `lean_compile` and `mathlib_search`. Commit to a
concrete proof plan up front and execute it against the Lean compiler
-- iterating on compiler feedback is how proofs get done, not silent
reasoning or repeated searching. The compiler is a stronger signal
source than search.

Use `lean_compile` to compile Lean 4 code. Call it early, even with a
partial proof: use `sorry` as a placeholder for sub-goals you cannot
yet discharge, and iterate (compile -> read errors / open goals ->
patch -> compile). The system handles two cases automatically based on
what you submit:
- If your code includes the MAIN theorem with the canonical statement
followed by `:= by ...`, the system rebuilds under the original
theorem statement: only your `:= by` proof body is kept from your
submission; the imports, `set_option`, and `open` lines come from the
canonical formal statement, and any other top-level declarations are
dropped. Only this case can register a solve. Do not use `axiom` or
`native_decide`; use `have` for helper lemmas inside your proof, not
top-level declarations; and do not add `import` or `open` lines that
are not already in the canonical formal statement -- any extras will
be flagged as a safeguard violation, not silently kept.
- If your code does NOT include the main theorem (e.g. `#check`,
`example`, `#print`, helper-lemma prototypes), the system compiles the
snippet as-given and returns the raw feedback. This is exploration
only -- it cannot register a solve, so resubmit with the main theorem
once you have a full proof. Use this sparingly: every turn against the
compiler costs budget, and the only way to finish is to submit the
main theorem.

Use `mathlib_search` as a lookup helper for *specific* Mathlib lemmas
you need while executing your plan -- for example a name, signature,
or hypothesis pattern like "monotonicity of natural number addition"
or "Cauchy-Schwarz inequality", or to recover the correct name after
an "Unknown constant" / "Unknown identifier" error. Mathlib does NOT
contain the solution to your problem directly, so do not use this tool
to "find the proof" or to search for an exact bound stated in the goal
-- such queries return nothing useful and waste turns.
```

C.3 BLUEPRINT REFINEMENT

```
## Task
```

```
You are revising a Lean 4 dependency graph for a single mathematical
problem. The input is a sequence of `@[blueprint ...]`-annotated
declarations -- definitions, lemmas, and one main theorem -- each
lemma or theorem with body `:= by sorry_using [deps]`. Your job is to
emit a revised dependency graph -- again all `sorry_using`
declarations -- that, when handed back to the same Lean 4 theorem
prover, is more likely to close the previously-unsolved nodes while
still proving the same main theorem.

## Input format
Each lemma or theorem in the input carries a one-line marker recording
the previous prover pass's verdict on that node, and -- when the
prover failed -- a follow-up review block describing what went wrong.
There are two markers.

A `-- PROVED` marker means the prover proved the node.

A `-- UNPROVED` marker indicates that the prover failed on the node,
and is followed by exactly one `/- Diagnosis ... -/` review block. The
block has three sections. `## Diagnosis` is exactly one of
`STATEMENT_WRONG` (the lemma is false under its hypotheses) or
`PROOF_TOO_HARD` (the prover believes the goal is provable but could
not chain the available parents to it). `## Analysis` is a forensic
account of what the prover tried, what compiled, what errors remained,
and where the gap is. `## Suggested Fix` is conditional on the
diagnosis: for `STATEMENT_WRONG`, why the statement is false and how
to repair it; for `PROOF_TOO_HARD`, a helper-lemma decomposition.

These markers and review blocks are input-only -- do NOT copy them
into your revised dependency graph.

## Guidance
Each `-- UNPROVED` node falls into one of two buckets, decided by the
`## Diagnosis` label.

When the diagnosis is `STATEMENT_WRONG`, the lemma's formal statement
is false under its hypotheses. Fix the statement (strengthen
hypotheses, weaken the conclusion, fix a quantifier or coercion, etc.)
and re-emit it. If the lemma is structurally unfixable, drop it and
re-route the nodes that depended on it.

When the diagnosis is `PROOF_TOO_HARD`, the prover believes the goal
is provable but could not chain the available parents to it. Read the
`## Suggested Fix` for the prover's proposed helper-lemma
decomposition and add new parent lemmas (each as a fresh `@[blueprint
...]` declaration with body `:= by sorry_using [...]`) that bridge the
gap. Wire the failing node's `sorry_using [...]` to include the new
helpers. If the analysis instead reads as though the statement itself
is suspect, treat it as `STATEMENT_WRONG` instead -- fix or drop the
statement.

Leave `-- PROVED` nodes untouched unless a downstream revision forces
a signature change: their proof bodies will carry forward
automatically as long as the signature stays byte-identical.

After every edit, call `lean_compile`. The tool reports pre-compile
safeguard violations, real Lean compile errors, the skeleton-out
invariant (every theorem/lemma body must remain `:= by sorry_using
[...]`), graph-validity issues (cycles, missing fields, dead nodes,
etc.), and on a clean compile a per-declaration proof-reuse check.
Iterate until `lean_compile` reports `Compilation SUCCESSFUL.
Validation SUCCESSFUL.`

## Output
```

```
Emit a revised dependency graph. Every theorem and lemma is
`@[blueprint (statement := /-- ... -/) (proof := /-- ...
-/)]`-annotated and ends in `:= by sorry_using [deps]`. Definitions
are `@[blueprint (statement := /-- ... -/)]`-annotated with a real
Lean body. Do NOT replace any `sorry_using` with an actual proof --
that is the prover's job, not yours. Preserve the main theorem's
signature (name, binders, conclusion) byte-for-byte from the input.
```