Goedel-Architect: Streamlining Formal Theorem Proving with Blueprint Generation and Refinement

Jui-Hui Chung [*1] **Ziyang Cai** [*1] **Zihao Li** [1] **Qishuo Yin** [1] **Rohit Agarwal** [1] **Simon Park** [1] **Rodrigo Porto** [1] **Narutatsu Ri** [1] **Ziran Yang** [1] **Shange Tang** [1] **Xingyu Dang** [1] **Hongzhou Lin** [2] **Mengdi Wang** [1] **Danqi Chen** [1] **Chi Jin** [1] **Liam Fowl** [*†1] **Sanjeev Arora** [†1]

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% ($597/672$), 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.

[*]Equal contribution [†]Joint last authors. [1]Princeton Language and Intelligence, Princeton University [2]Amazon. This work is independent of and outside of the work at Amazon. Correspondence to: Jui-Hui Chung <jc1220@princeton.edu>, Ziyang Cai <zc5794@princeton.edu>, Liam Fowl <lf2728@princeton.edu>.

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

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), to solving open Erdős problems that had previously deterred even elite math researchers (Alexeev 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 submission, every system exceeding 50% on PutnamBench either uses *unreleased model or pipeline* (Logical Intelligence, 2025; Chen et al., 2025b;a), or requires *expensive frontier model usage* - requiring many thousands of dollars in API credits to solve the slate of questions in PutnamBench (Varambally et al., 2025; Requena et al., 2026).

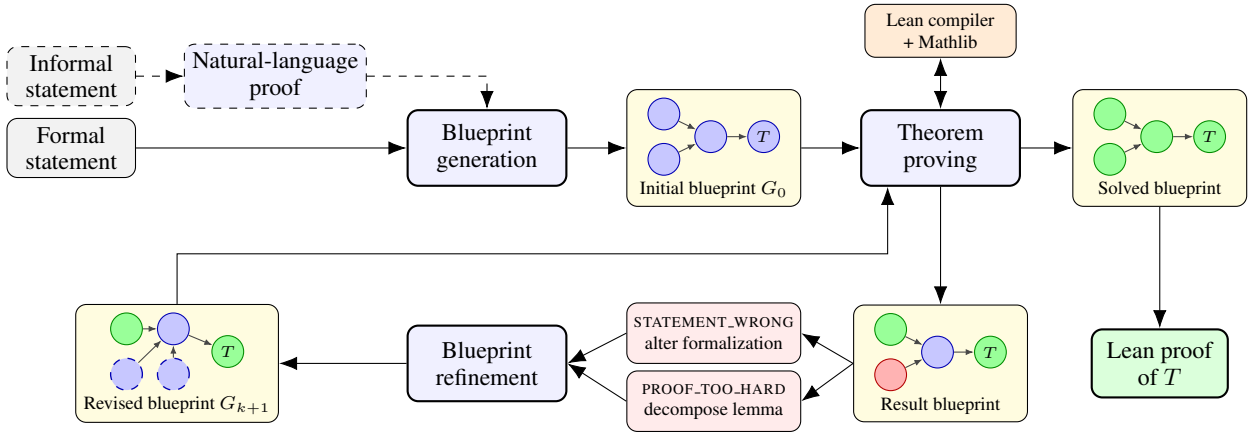

Figure 1. Overview of the GOEDEL-ARCHITECT pipeline. **Inputs.** The formal statement always seeds blueprint generation; an informal statement and a natural-language proof (dashed) form an optional structural guide. **Blueprint generation.** It emits an initial dependency graph G_0 of definitions and lemmas building up to the main theorem T (arrows are declared dependencies); every node starts *unsolved* (blue). Theorem proving dispatches each lemma to a Lean prover in parallel, restricted to its declared parents, with every candidate proof checked against the Lean compiler and Mathlib, and emits a result blueprint marking each node *solved* (green), *unsolved* (blue), or *formally negated* (red). When every node is solved, the solved blueprint assembles into a Lean proof of T. **Refinement loop.** Otherwise, each failing node carries one of two diagnoses: STATEMENT_WRONG — the statement is false under its hypotheses, so the formalization is altered — raised either by a formally negated (red) node, where the prover verified a counterexample, or by an unsolved (blue) node the prover argues is false but could not disprove; or PROOF_TOO_HARD — an unsolved (blue) node the prover believes provable but could not chain its parents to, so the lemma is decomposed into helper lemmas. Blueprint refinement consumes these signals and emits the revised blueprint G_{k+1}, preserving already-solved nodes as green (their proofs reused while signatures and parents are unchanged, though an upstream edit can revert a dependent to blue) and adding new helper lemmas (dashed). The revised blueprint is re-proven, and the loop continues until every node is solved or the iteration budget is exhausted.

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), 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. AxProverBase (Requena et al., 2026) and Numina-Lean-Agent (Liu et al., 2026) pair this template with a proprietary frontier backbone — Claude Opus 4.5 in both cases (closed backbone, open pipeline) — achieving strong 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); 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) is 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 — 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 com-

pute 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). 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 core innovation of our 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 — 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; 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 and a Mathlib retrieval tool, and may call them iteratively until it either closes the goal or exhausts its per-lemma budget. When the prover gives up on a lemma, it returns a structured diagnosis recording what it attempted and where it believes the gap lies; when it produces a compiler-corroborated counterexample, it can register a proof of the negated statement in lieu of the original. These per-lemma signals are the input to blueprint refinement.

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. 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.

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

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).

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	–	–	–
Goedel-Architect	99.2% @ 1	75.6% @ 1	–	–	–
Goedel-Architect (+ NL)	100%	88.8% @ 4	4/6	11/12	3/6

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 reports GOEDEL-ARCHITECT against the strongest publicly reported provers across the five benchmarks: MiniF2F-test, PutnamBench, IMO 2025, Putnam 2025, and USAMO 2026. Not every pipeline tested on every benchmark, and accordingly, corresponding results are left blank.

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.

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 $243/244$.

Table 2. PutnamBench cost comparison between Goedel-Architect (DeepSeek-V4-Flash), Hilbert (Gemini 2.5 Pro), and AxProverBase. Total cost and average cost per question (denoted Avg. cost / Q) are reported. Note that the Hilbert's cost is taken from pass@1 numbers, and is an *underestimate* of their actual spend.

Metric	Goedel-Arch.	Hilbert*	AxProver
Total spend	\$294	∼\$163k	\$8,467
Avg. cost / Q (all)	\$0.44	∼\$244	\$12.60
Avg. cost / *solved* Q	\$0.21	—	—
Avg. cost / *unsolved* Q	\$1.14	—	—

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. Po-

tentially 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 close to $170,000 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.

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 ∼ $1.65 per problem solved.

IMO 2025. We take the formalized problem statements for IMO 2025 from the Seed-Prover paper (Chen et al., 2025a). 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.

Of the four solved problems, P5 is closed without any natural-language proof as input: blueprint generation runs directly from the formal statement. For P1, P3, and P4 we use Gemini 3.1 Pro to supply an initial natural-language proof, which seeds the first blueprint generation pass as a structural guide; the rest of the pipeline (proving and refinement) is unchanged.

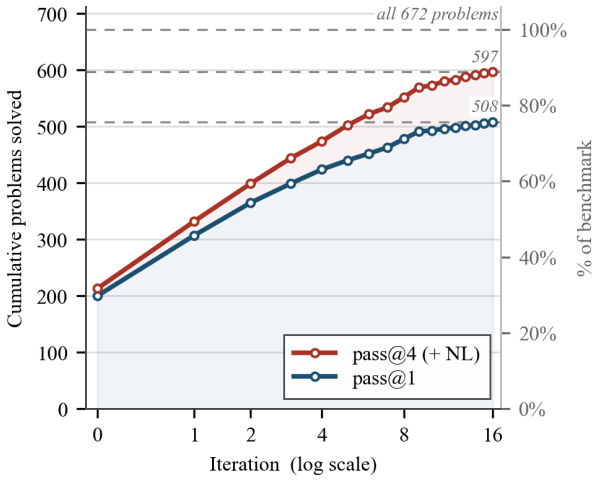

Figure 2. Compute scaling on PutnamBench. Cumulative problems solved by GOEDEL-ARCHITECT (left axis; right axis as % of the 672-problem benchmark) against the number of blueprint-refinement iterations on a log scale, using the open-weight DeepSeek-V4-Flash backbone. The pass@1 curve uses only the default 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.

Putnam 2025. The formalized problem statements for Putnam 2025 are taken from PutnamBench (Tsoukalas et al., 2024). 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.

Of the eleven solved problems, A1, A2, A4, A6, B2, B3, and B5 are closed without any natural-language proof as input: blueprint generation runs directly from the formal statement, and these are Gsolved as part of the same pass@1 run we report on PutnamBench. For A3, B1, B4, and B6 we use Gemini 3.1 Pro to supply an initial natural-language proof, which seeds the first blueprint generation pass as a structural guide; the rest of the pipeline (proving and refinement) is unchanged.

USAMO 2026. USAMO 2026 is recent enough to post-date the training cutoff of every model in our pipeline, so it serves as a contamination-free benchmark: neither the problem statements nor their solutions could have been memorized during training. We formalize the USAMO 2026 problem statements ourselves, with the help of Claude Opus 4.7. GOEDEL-ARCHITECT solves 3/6 problems (P1, P4, P6). For all three we use Gemini 3.1 Pro to supply an

initial natural-language proof, which seeds the first blueprint generation pass as a structural guide; the rest of the pipeline (proving and refinement) is unchanged.

4. Other Pipeline Features

In this section, we describe some of auxiliary pipeline features, and their impact on the entire pipeline, as well as studying some cases that highlight their utility.

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*, 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.

In Table 3, we see that our pipeline design provides an even bigger boost in performance compared to Hilbert when we control for backbone model. We also observe that on easier problems, tool-integrated reasoning (TIR) naturally yields relatively high performance on minif2f. However, Figure 2 shows that TIR is alone insufficient to achieve our performance numbers on the more difficult PutnamBench.

Table 3. Base-model effectiveness on the same DeepSeek-V4-Flash backbone. 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. PutnamBench numbers are over a random 200-problem subset due to control compute.

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	–

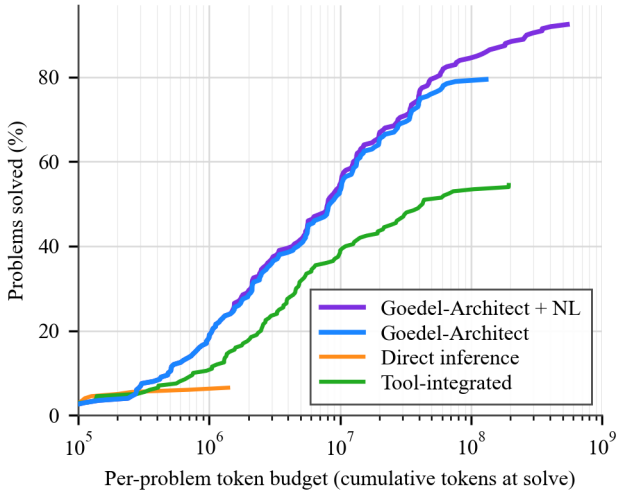

Figure 3. Per-problem solve cost on PutnamBench. All three systems share the same DeepSeek-V4-Flash backbone and are evaluated on the *same* random subset of N=200 PutnamBench problems — those attempted by all three systems. For each problem we record the cumulative tokens (input + output) spent on it 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.

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 — but 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. Negated sub-lemmas as proof-strategy diagnostics

When the blueprint proposes a sub-lemma that is actually false, our per-lemma prover closes a proof of its *negation* instead of the lemma itself (Section 2.2). The refinement loop treats this as a diagnostic signal: alongside the disproof it records a short reflection — a diagnosis, the counterexample that kills the claim, and a suggested fix — and feeds that reflection into the next iteration's blueprint-revision step, which 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. Two examples show how the revision consumes the signal.

We discuss illustrative examples of this phenomenon in Appendix B.1. In both cases here, 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 here, a representational confusion there — 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.

4.4. Forfeited proofs as decomposition proposals

The negation channel above fires only when the prover can verify a counterexample. A more common outcome is that a node simply runs out of budget: the prover can neither close the lemma nor disprove it within its turn and token limit. Rather than report a bare failure, the prover is then required to write a structured post-mortem — a *forfeit* — in three parts: a diagnosis of either the statement being wrong or the proof being too hard; a forensic analysis of what it tried and where the proof stalled; and a suggested fix. A statement-is-wrong forfeit flags a lemma the prover suspects is false but could not actually disprove, and recommends weakening or strengthening it. A proof-is-too-hard forfeit asserts the lemma is true but out of reach in one shot, and proposes a *lemma decomposition*: a set of named helper lemmas arranged so that each is easy given its parents and the original goal becomes routine given the helpers. As with the disproof reflection, this text is injected into the next iteration's blueprint-revision step, which uses it to split or repair the node — turning a goal that failed as a single monolith into a sub-graph the prover can actually discharge.

A case study for this functionality is discussed in Appendix B.2. This is the constructive counterpart to the negation channel. A disproof tells the loop a node is dead and must be 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.

References

Alexeev, B., Barreto, K., Li, Y., Lichtman, J. D., Price, L., Shah, J. I., Tang, Q., and Tao, T. 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.

Chen, J., Chen, W., Du, J., Hu, J., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Shi, W., et al. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience. *arXiv preprint arXiv:2512.17260*, 2025a.

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

Comanici, G., Bieber, E., Schaekermann, M., Pasupat, I., Sachdeva, N., Dhillon, I., Blistein, M., Ram, O., Zhang, D., Rosen, E., 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.

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/.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022.

Li, Y., Du, D., Song, L., Li, C., Wang, W., Yang, T., and Mi, H. Hunyuanprover: A scalable data synthesis framework and guided tree search for automated theorem proving. *arXiv preprint arXiv:2412.20735*, 2024.

Lin, H., Sun, Z., Yang, Y., and Welleck, S. Lean-STaR: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040*, 2024.

Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., et al. Goedel-Prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025a.

Lin, Y., Tang, S., Lyu, B., Yang, Z., Chung, J.-H., Zhao, H., Jiang, L., Geng, Y., Ge, J., Sun, J., et al. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025b.

Liu, J., Zhou, Z., Zhu, Z., Santos, M. D., He, W., Liu, J., Wang, R., Xie, Y., Zhao, J., Wang, Q., 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.

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

Ren, Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., et al. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Requena, B., Letson, A., Nowakowski, K., Ferreiro, I. B., and Sarra, L. A minimal agent for automated theorem proving. *arXiv preprint arXiv:2602.24273*, 2026.

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.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *Advances in Neural Information Processing Systems*, 37:11545–11569, 2024.

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

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M. D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Wang, J., Zhang, J., Guo, Q., Guo, L., Li, R., Zhang, C., Peng, C., Wang, C., Zhao, D., Shi, J., et al. Longcat-flash-prover: Advancing native formal reasoning via agentic tool-integrated reinforcement learning. *arXiv preprint arXiv:2603.21065*, 2026.

Wang, R., Zhang, J., Jia, Y., Pan, R., Diao, S., Pi, R., and Zhang, T. Theoremllama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*, 2024.

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

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

Xin, H., Ren, Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., et al. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024b.

Zheng, K., Han, J. M., and Polu, S. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

Zhu, T., Monticone, P., Avigad, J., and Welleck, S. Lean-Architect: 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 revising 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.

B. Case studies

Here we include a few interesting case studies to illustrate the functionality/operation of different parts of our pipeline.

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 — and 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. Forefeited 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 — X^2 divides the polynomial when $k \geq 2$, 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 — a goal that had been unprovable as a monolith one iteration earlier.

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.
```