Lean Refactor: Multi-Objective Controllable Proof Optimization via Agentic Strategy Search

Jialin Lu [1] **Soonho Kong** [2] **Rodrigo Stehling** [1] **Kaiyu Yang** [3] **Zhangyang Wang** [4] **Weiran Sun** [1] **Wuyang Chen** [1]

Abstract

We present **Lean Refactor**, a plug-and-play retrieval-augmented agentic framework for *multi-objective, controllable, and version-robust* refactoring of Lean proofs. LLM-generated proofs are notoriously correct-but-verbose and brittle across library versions, yet existing refactoring works overlook **three practical challenges**: **1)** Lean refactoring is natively multi-objective (proof length, compilation cost, and version compatibility are often in tension); **2)** Lean repositories have fragile compatibility, whereas LLM releases are unaware of Lean/Mathlib versions; **3)** Training-based pipelines require repeated fine-tuning with each new LLM release, scaling neither with model churn nor with Lean's release cycle. Lean Refactor steers a **frozen** agentic LLM with retrievals from a curated database of **multi-objective refactoring strategies**, each densely annotated with metadata such as supported Lean/-Mathlib versions and expected compilation-cost reduction. Experiments show over **70% token-level compression** on competition benchmarks, over **20%** on research repositories, and **up to 60% compilation-time reduction**, outperforming prior work and Claude Code. Version-filtered retrieval further improves compression on the target Lean version, and refactored miniF2F proofs exhibit stronger **zero-shot version transfer** to future Lean releases than their unrefactored counterparts. We will release our code, model, and data upon acceptance.

[1]Simon Fraser University [2]Amazon Web Services [3]MiroMind [4]University of Texas at Austin. Correspondence to: Wuyang Chen <wuyang@sfu.ca>.

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

1. Introduction

Lean and Mathlib (De Moura et al., 2015; Moura & Ullrich, 2021) enable collaborative, machine-verified mathematics and provide a strong reinforcement learning testbed that grounds large language model (LLM) reasoning in formal proofs, supporting gold-medal International Mathematical Olympiad (IMO) performance (Trinh et al., 2024; Achim et al., 2025) and rapid progress in LLM-powered provers (AlphaProof & teams, 2024; Xin et al., 2024a;b; Ren et al., 2025; Lin et al., 2025a;b). However, LLMs trained with reinforcement learning (RL) often produce Lean proofs that are correct but **notoriously long and hard to read** (Ahuja et al., 2024; Gu et al., 2025). Because RL rewards target proof correctness, models are insensitive to redundant steps or heavy automation where simple steps suffice. Such proofs are difficult for humans to understand and computationally expensive to compile, posing particular problems for large libraries like Mathlib.

Recent work on LLM-based code refactoring (Gautam et al., 2025; Gong et al., 2025b) pursues either fine-tuning LLMs (Gu et al., 2025; Yang et al., 2025; Gong et al., 2025a) or agentic frameworks built on pretrained general-reasoning LLMs (Ahuja et al., 2024; DePalma et al., 2024; Ospanov et al., 2025; Piao et al., 2025; Oueslati et al., 2025; Karabiyik, 2025; He et al., 2025; Cui et al., 2025; Bai et al., 2025). However, LLMs have seen limited adoption for refactoring Lean proofs, and we point out **three key limitations**. ***First, Lean refactoring is natively multi-objective.*** Lean code is often simultaneously optimized along several axes, such as proof length and compilation cost. These objectives are entangled and frequently in tension: shorter proofs may invoke heavier tactics that inflate compilation cost, while cheaper proofs may sprawl in length. Standard pretraining, RLHF, and fine-tuning (Ouyang et al., 2022; Achiam et al., 2023) offer no principled mechanism to balance such trade-offs, let alone the **unseen objectives** users routinely introduce at inference time.

Second, frontier LLMs are misaligned with the Lean/Mathlib release cycle. Lean and Mathlib evolve on a near-weekly basis, with lemmas renamed, APIs restructured, and new automation landed, while LLMs are pinned to a knowledge cutoff. Even with web-search augmentation, **no principled**

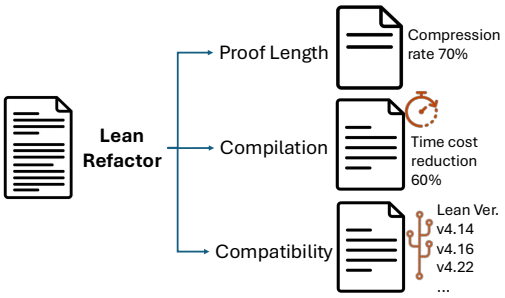


Figure 1. Our Lean Refactor can be controlled to refactor Lean proofs for multiple objectives: proof length, compilation time cost, and compatibility across Lean versions.

way exists for maintainers to pair a codebase version with a compatible model, leading to pervasive hallucination of stale tactics and removed APIs.

Third, paired refactoring data is scarce. Compared with mainstream languages (C/C++, Java, Python), Lean has a far smaller codebase, and existing Mathlib and community-aggregated competition corpora (Zheng et al., 2022; Tsoukalas et al., 2024; Li et al., 2024) provide neither refactoring trajectories nor noisy-clean code pairs.

To address these fundamental bottlenecks, we investigate our core question:

> *Can a frozen LLM be steered at inference time to balance competing refactoring objectives while reliably refactoring code in fast-evolving Lean/Mathlib environments?*

In this paper, we introduce **Lean Refactor**, a retrieval-augmented agentic framework that externalizes refactoring knowledge into a structured database. Lean Refactor grounds an agentic LLM with a curated corpus of **multi-objective, maintainable refactoring strategies**. Each retrieved strategy supplies the LLM with aligned structural context and explicit execution instructions (*when to apply*, *how to apply*, and *examples*). Crucially, these strategies are **densely annotated with refactoring metadata**, tracking signals such as expected compilation-cost reduction and validated Lean/Mathlib version compatibility. By dynamically filtering and reranking these annotated strategies at inference time, our framework steers proof optimization toward user-specified objectives without model fine-tuning, and remains version-robust as Lean/Mathlib evolves beyond the LLM's knowledge cutoff.

We summarize our contributions below:

1. We release the largest Lean refactoring dataset to date: over 200K long-short proof pairs and 9K refactoring strategies, each annotated with compilation-cost reduction and compatible Lean/Mathlib versions.

2. Lean Refactor achieves over 70% compression on competition proofs (miniF2F (Zheng et al., 2022), PutnamBench (Tsoukalas et al., 2024), Putnam2025) and over 20% on research repositories (PFR (Tao, 2025), PhysLean (Tooby-Smith, 2025), Analysis (Tao, 2024), FLT (Buzzard & Taylor, 2025)), outperforming prior work and Claude Code while remaining compatible with Gemini, Claude, and GPT series backbones.

3. Lean Refactor supports multi-objective, controllable refactoring: beyond length, it cuts compilation cost by over 30%, holds compression steady across Lean versions, and produces miniF2F proofs that type-check across more future Lean/Mathlib releases than their unrefactored counterparts.

2. Lean Refactoring is Challenging for LLMs

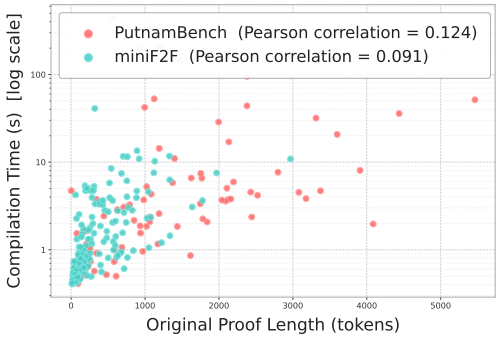


Figure 2. **Weak correlation** between proof length and compilation time. Token length fails to predict compilation cost because of short, heavy tactics, a factor often overlooked in previous Lean refactoring research.

Beyond the redundancy in code length demonstrated in previous papers (Ahuja et al., 2024; Gu et al., 2025), in this section, we motivate our work with more practical challenges of Lean code refactoring with LLMs.

Proof Optimization is Multi-Objective. While proof length is the most convenient refactoring objective, as it can be computed during LLM inference, compilation cost is an equally important complexity metric, and the two are not always aligned (Figure 2; see Appendix D for an example). Prior Lean refactoring work either ignores compilation cost or swaps it in as a substitute inference-time objective, rather than jointly optimizing it with length (Ahuja et al., 2024; Gu et al., 2025). More broadly, while recent formal-math post-training increasingly relies on verifiable rewards (e.g., compiler-judged correctness), continuous runtime signals such as **compilation cost remain underused** (Lin et al., 2025b; Gu et al., 2025).

Lean Repository Compatibility is Fragile. Version sensitivity is a long-standing bottleneck of Lean/Mathlib: Lean and Mathlib evolve rapidly; weekly releases frequently rename declarations and break imports even across patch ver-

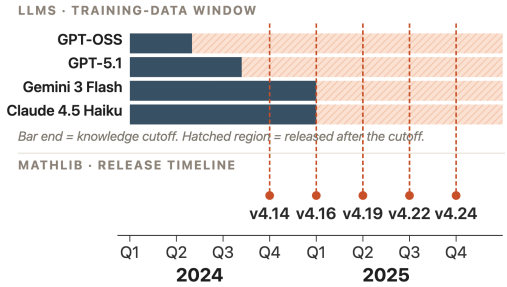


Figure 3. Knowledge cutoff lag in LLMs: Most LLMs misalign with Lean/Mathlib versions due to static knowledge cutoff.

sions (Baanen et al., 2025), forcing downstream projects to pin to Mathlib's exact toolchain (mathlib4, 2025)[1]. Agentic LLMs, however, are **never aligned** with a specific version: as Figure 3 shows, every recent frontier model's training cutoff lands *before* recent Mathlib releases, leaving newer APIs unseen and the model's coverage of older versions opaque. This pushes maintainers into continuous, costly upgrades. **Lean Refactor** mitigates this misalignment by grounding the agent in a strategy bank whose strategies carry explicit Mathlib-version metadata: retrieval is filtered to the user's target toolchain, and the bank itself can be re-profiled as new Mathlib versions ship.

Competitions Fail to Reflect Research-Level Proofs. Current LLM provers are primarily trained on competition mathematics, such as NuminaMath (Li et al., 2024) and Goedel-Pset (Lin et al., 2025a). However, competition proofs do not reflect the structural complexity of research-level Lean code. As Table 9 shows, research proofs rely far more heavily on Mathlib and intra-project dependencies. This dense network of external and internal lemmas makes research-level code harder to refactor, forcing the model to reason over a much larger and less familiar context to find valid simplifications. Consequently, despite their practical importance, research-level proofs across diverse domains have been largely overlooked in prior Lean proof optimization work (Gu et al., 2025; Ahuja et al., 2024).

3. Methods

Lean Refactor is a retrieval-augmented agentic framework built around three design choices that directly address the bottlenecks in Section 2. We externalize refactoring knowledge into a densely annotated strategy bank whose metadata, including Lean/Mathlib version compatibility and expected compilation cost reduction, mitigates the misalignment between the LLM's training cutoff and Lean/Mathlib release cycles (Section 3.1). At inference, the target proof is semantically grounded via dependency annotations [2], while a multi-objective retrieve-and-rerank pipeline steers the agent toward user-specified trade-offs between proof length, compilation cost, and version compatibility (Section 3.2). We wrap these components in a plug-and-play agent loop whose planner, refactorer, and debugger are instantiated by any frozen frontier LLM, so the framework transfers across Gemini, Claude, and GPT series without any retraining (Section 3.3). Figure 4 illustrates the pipeline; pseudocode is in Algorithm 1.

3.1. A Densely Annotated, Version-Aware Strategy Bank

Strategy structure. Each entry is a reusable refactoring pattern distilled from a raw long–short proof pair, organized into six fields: *title*, a natural-language *description*, a precondition pattern (*when to apply*), a step-by-step *application guide*, a *before/after example*, and a categorical *potential length reduction* tag. The schema preserves enough structural context for the LLM to generalize, promoting transfer to unseen proofs. Examples appear in Appendix M.

Refactoring metadata. Each strategy is additionally annotated along two axes that the LLM cannot infer from its training data:

- *Compilation-time reduction.* We profile both proofs in every long–short pair with `lake env lean --profile`, isolating proof execution from import overhead, and record the relative speedup. Since each strategy can be instantiated by many long–short pairs, we keep the median speedup across all pairs associated with the strategy as the strategy-level estimate.
- *Version compatibility.* For all non-Mathlib pairs (Mathlib is locked to its toolchain), we recompile each shortened proof under three additional Lean toolchains beyond the original v4.24.0: v4.14.0, v4.16.0, and v4.22.0. A strategy's compatibility set is the intersection of toolchains under which all of its shortened proofs continue to compile, certifying that the transformation is robust across the interval rather than merely syntactically plausible in a single snapshot. This metadata is what makes the bank version-aware at inference time (Section 3.2) and is the primary lever by which we mitigate the misalignment between the LLM's training cutoff and Lean/Mathlib release cycles.

Bank construction. We aggregate statements from NuminaMath-1.5 (Li et al., 2024), FineLeanCorpus (Peng et al., 2025), Mathlib, and ATLAS (Liu et al., 2025), and synthesize proofs with Goedel-Prover-V2-32B (Lin et al.,

[1] In a Zulip thread on mathlib-dependent projects, the requirement is stated bluntly: "It needs to be the exact same version." See this discussion.

[2] We use adapted versions of jixia and ntp-toolkit for dependencies extraction.

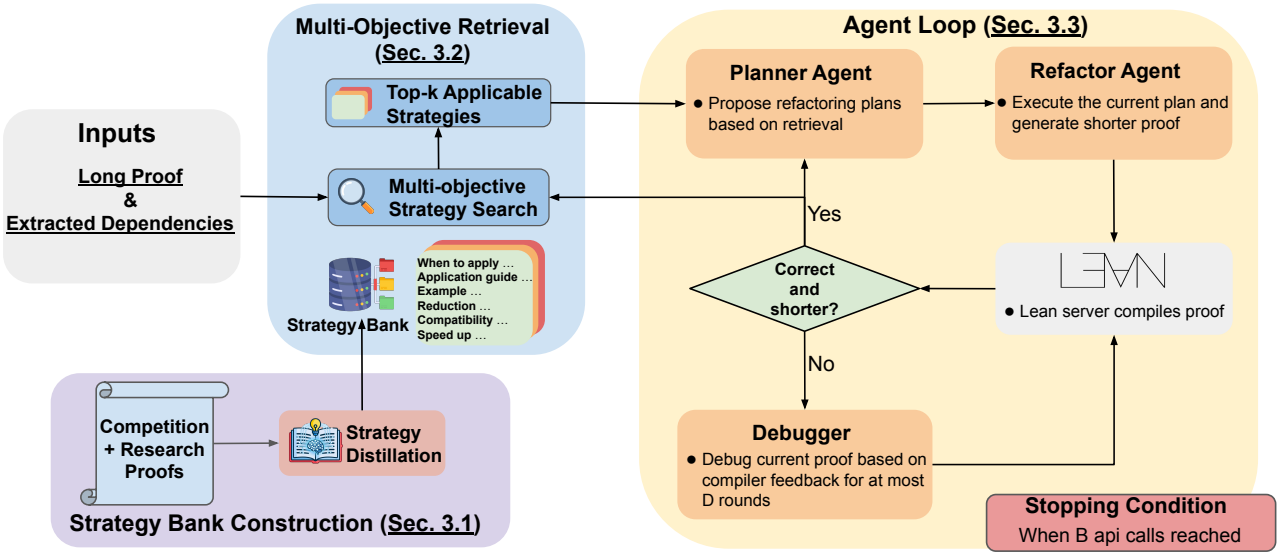


Figure 4. Overview of Lean Refactor framework. 1. We first summarize raw long-short proof pairs sourced from diverse Lean code sources into well-structured reusable strategies, and more importantly, we annotate each strategy with code refactoring metadata, including compilation time cost and version compatibility (Section 3.1). 2. We then train a strategy retrieval model that maps Lean code (that can be potentially optimized) to one or more refactoring strategies, with multi-objective re-rank and filtering (Section 3.2). 3. Finally, we design an agent workflow that intelligently plans refactoring, generates shortened proofs, and debugs (Section 3.3). Components in our agent workflow (planner, refactor, debugger) are frontier LLMs.

2025b) and GPT-OSS-120B (OpenAI et al., 2025). We decontaminate the aggregated theorems against our held-out evaluation set (defined in Section 4) via exact-match and embedding-based filtering to preclude leakage. Pairs are built by shortening long proofs and synthetically expanding short ones. GPT-OSS-120B distills each pair into one or more location-grounded strategies; a second LLM-as-judge pipeline filters incorrect extractions; and a Qwen3-Embedding-8B (Zhang et al., 2025) based iterative de-duplication pipeline clusters semantically equivalent strategies. The final corpus contains **9237 unique strategies** distilled from 481567 raw strategy extractions over 200K long–short proof pairs. Full details are in Appendix B.

3.2. Multi-Objective Retrieval for Controllable Refactoring

Retrieval model training. We fine-tune Qwen3-Embedding-8B (Zhang et al., 2025) on 639090 (query, strategy) pairs. Our training procedure largely follows prior Lean code search work (Lu et al., 2025), with several adaptations tailored to our refactoring setting. During strategy distillation from long–short proof pairs, we record the code segment in the long proof on which the strategy operates (Appendix B.4); we use this segment as the query and the strategy's *when-to-apply* clause as its positive target. Let B denote the mini-batch size, q_i the i-th query, c_i^+ its ground-truth positive strategy, and $\mathcal{C}_B = \{c_j^+\}_{j=1}^B$ the set of positives in the batch, where each c_j^+ with $j \neq i$ serves as an in-batch negative for query q_i. We write $\mathrm{Sim}(\cdot,\cdot)$ for cosine similarity, τ for the softmax temperature, and m for a false-negative margin. The indicator $\mathbb{I}(q_i, c)$ equals 0 when $\mathrm{Sim}(q_i, c) > \mathrm{Sim}(q_i, c_i^+) + m$, masking out candidates that score higher than the positive by more than m, and 1 otherwise. The training objective is

$$\mathcal{L} = -\frac{1}{B}\sum_{i=1}^{B} \log \frac{\exp(\mathrm{Sim}(q_i, c_i^+)/\tau)}{\sum_{c\in\mathcal{C}_B} \mathbb{I}(q_i, c)\, \exp(\mathrm{Sim}(q_i, c)/\tau)}. \quad (1)$$

We additionally apply boundary augmentation, jittering segment boundaries to bridge the gap between clean training spans and noisy runtime inputs. Training details are in Appendix E.

Objective-conditioned retrieval at inference (Figure 5). The user specifies an objective at inference time, and we reshape the retrieval rule accordingly while operating over the *same* underlying bank:

- *Shortest proof.* If the user prioritizes proof length, we use pure cosine-similarity top-K, exposing the agent to the most structurally relevant strategies.
- *Compilation-time aware.* If the user prioritizes compile-time reduction, we apply a two-stage retrieve-and-rerank procedure: we fetch a broad pool of semantically similar candidates, then rerank in descending order of annotated compile-cost reduction. Because the context window is bounded, top-K by similarity alone might exclude high-efficiency strategies; reranking surfaces them without sacrificing applicability.
- *Version-specific.* If the user targets a particular Lean version, we filter the candidate pool to strategies whose

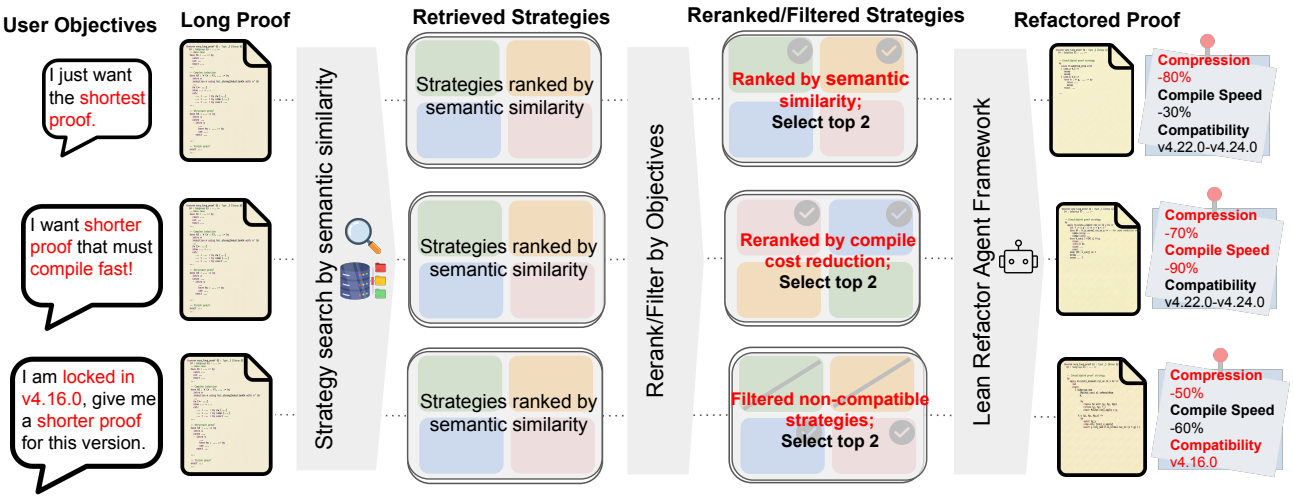


Figure 5. **Multi-objective Lean proof refactoring.** A single strategy bank adapts to diverse user objectives at inference without retraining. *Top:* for shortest proofs, top-K strategies by cosine similarity are used directly. *Middle:* to balance faster compilation, candidates are reranked by annotated compile-cost reduction. *Bottom:* for a target toolchain (e.g., v4.16.0), incompatible strategies are filtered out. The agent outputs (right) reflect these distinct trade-offs. Metadata-driven objectives are fully composable and easily extensible to new criteria.

compatibility set includes that version, removing version-incompatible strategies.

Each objective is realized as a different retrieval rule over the same shared bank. This design composes cleanly: version filtering can be stacked on top of compile-time reranking to jointly target both objectives. It also admits *new* objectives at inference time by introducing new metadata fields without touching the model.

3.3. A Plug-and-Play, LLM-Agnostic Agent Loop

Iterative agent loop (Algorithm 1, Figure 4; see Appendix N for prompts). Given a long proof, we annotate it with the signatures of all invoked dependencies to provide semantic grounding. The annotated proof then enters an agentic loop that coordinates four distinct stages:

1. *Retrieval.* We segment the annotated proof at multiple granularities to enable refactorings of varying scope. Each segment is embedded and used to query the strategy bank; the top-K strategies are retrieved and processed under the user-specified objective (Section 3.2), then passed forward with their target locations.

2. *Planner.* A frozen LLM consumes the current proof, the retrieved strategies, and execution/planning history, and decomposes the task into a sequence of refactoring steps mapped to specific segments (or the entire proof).

3. *Refactorer.* The same LLM, given one planned step, rewrites the targeted segment (or the entire proof). The candidate is handed to the Lean 4 compiler for validation.

4. *Debugger.* On compile failure, compiler feedback is routed back to the LLM for up to D rounds of localized repair, constrained to the planner's original intent.

On successful compilation *and* length reduction, the updated proof and trace are fed back to the planner for dynamic replanning; on persistent failure, the step is logged and skipped. The loop terminates at a budget of B LLM calls or when proof length falls below target T.

Adaptability and Sustainability. Our design choices keep the framework agnostic to both LLM backbones and Lean versions. The planner, refactorer, and debugger are model-agnostic agent roles that operate over any frozen frontier LLM, so the framework can transfer to a new backbone with no fine-tuning or adaptation step. Refactoring knowledge, including strategies together with their version and compilation metadata, lives in an external bank that is retrieved at inference time and can be updated as Lean/-Mathlib evolves. New optimization objectives are supported by acquiring the corresponding metadata, with no retraining required. As a result, new frontier LLMs inherit the bank's full coverage immediately, and new Lean versions are supported by re-profiling the bank. Section 4 empirically validates interchangeable deployment across Gemini, Claude, and GPT-OSS.

4. Experiments

Benchmarks. We evaluate Lean Refactor on seven benchmarks across two regimes. *Competition:* miniF2F (Zheng et al., 2022) (194 theorems), PutnamBench (Tsoukalas et al., 2024) (75 theorems), and Putnam2025 (66 theorems from problems B1/B2 of Seed-Prover 1.5 (Chen et al., 2025a)). For miniF2F and PutnamBench, we reuse the long proofs released by ProofOptimizer (Gu et al., 2025) so both methods

refactor identical inputs. *Research:* 45 theorems sampled from each of Analysis (Tao, 2024), FLT (Buzzard & Taylor, 2025), PFR (Tao, 2025), and PhysLean (Tooby-Smith, 2025), spanning diverse proof lengths with all internal and external dependencies retained.

Setup and metrics. Unless noted otherwise, all runs use Gemini 3 Flash (The Gemini Team, 2025), Lean v4.24.0, an API call budget of 30, and the syntax-aware tokenizer from ProofOptimizer (code in Appendix J). We report three primary metrics: **proof length reduction** (Section 4.1), **compilation-time reduction** (Section 4.2), and **cross-version compatibility** (Section 4.3). Heartbeat reduction and additional compilation-cost results are deferred to Appendices K and L; dataset statistics, length distributions, and full experimental details to Appendix I.

Baselines. On miniF2F and PutnamBench, we compare against **ProofOptimizer** (Gu et al., 2025) using their released optimized proofs[3]; their closed-source checkpoints preclude evaluation on the remaining benchmarks. We additionally isolate each component of our framework via four internal ablations: **Base Agent** (refactorer + debugger + retrieval, planner removed), **Base Agent w/o Retrieval** (planner and retrieval removed), **Lean Refactor w/o Retrieval** (planner + refactorer + debugger, retrieval removed), and **Lean Refactor w/ Random Retrieval** (retrieval replaced by sampling over the bank).

To test LLM-agnosticism, we re-run our framework with Claude Haiku 4.5 (Anthropic, 2025) and compare against **Claude Code** (also powered by Haiku 4.5), giving Claude Code a dollar budget matched to our framework's Haiku 4.5 runs. We evaluate Claude Code under three tool configurations of increasing tool access: (i) Lean MCP[4] only, providing compiler feedback; (ii) Lean MCP + Lean skills[5], adding general Lean 4 refactoring and golfing guidance; and (iii) the above plus our strategy retrieval tool, which exposes the strategy bank as an on-demand resource the agent can query during refactoring. Due to budget, this comparison is restricted to one representative benchmark per regime: PutnamBench (competition) and Analysis (research-level). See Appendix I for further details.

[3]ProofOptimizer's released proofs are produced via an 8-iteration shortening process with sampling budgets of $6\times64 + 2\times1024$ per theorem (64 samples per iteration for the first 6, 1024 for the last 2) from a fine-tuned 7B model, while our system uses 30 API calls per theorem orchestrating planner, refactorer, and debugger steps over a frozen frontier LLM. The two paradigms differ in model scale and sampling-vs-agentic control flow, and since ProofOptimizer's checkpoints are not publicly available, a matched-backbone or matched-budget comparison is not possible; we therefore evaluate each system under its released configuration. Our ablations (Table 1) isolate framework contributions on a shared backbone, and we additionally compare against Claude Code under matched budget and backbone (Table 2).

[4]https://github.com/oOo0oOo/lean-lsp-mcp

[5]https://github.com/cameronfreer/lean4-skills

4.1. Proof Length Reduction

Table 1 reports length reductions with Gemini 3 Flash (test-time scaling curves in Appendix O); Table 2 reports the LLM-agnostic check with Haiku 4.5 and comparison with Claude Code. **Lean Refactor achieves state-of-the-art performance across both regimes:** on competition benchmarks it surpasses ProofOptimizer, with the gap widening on PutnamBench, where longer proofs benefit most from retrieval-augmented planning, and on human-written research-level proofs it achieves substantial additional compression and beats every ablation. **Planning and retrieval are synergistic.** The planner is critical because without it, the Base Agent becomes overwhelmed by long proofs and stalls in persistent debug loops. Conversely, retrieval only pays off when the planner explicitly sequences the provided strategies. Results show that random retrieval yields inconsistent improvements, whereas targeted retrieval delivers consistent and substantially larger gains. This confirms that the framework's success stems from *accurate strategy selection* rather than mere context augmentation.

The framework is LLM-agnostic and the strategy bank improves the performance of general-purpose agents. The same ranking holds when the backbone is switched to Claude Haiku 4.5 (Table 2), and under a matched budget, Claude Code itself improves on both benchmarks once given access to our strategy retrieval tool. Even so, our full framework outperforms the strongest Claude Code configuration, demonstrating that our tailored planner and compiler-in-the-loop architecture provides targeted guidance that a general-purpose agent lacks, and more effectively leverages strategy retrieval. Beyond raw length, refactored proofs also reduce external and intra-project dependencies in 42–67% of cases across all benchmarks, evidence that the compression reflects genuine logical simplification rather than offloading work onto heavy library lemmas, which further improves readability and version resilience (Appendix F).

4.2. Multi-Objective Control: Compilation Time

We redirect the framework from length to compilation time by swapping only the retrieval rule (Section 3.2): **Lean Refactor w/ Reranking** reranks a semantically-similar candidate pool by each strategy's annotated compile-cost-reduction metadata. Controls are **w/o Retrieval**, **w/ Random Retrieval**, and the cosine-similarity-only **Lean Refactor**; ProofOptimizer is omitted on Putnam2025 (unreleased model weights). Compile time is measured via `lake env lean --profile`, excluding library import time. Table 3 reports average relative and absolute reductions on

Table 1. **Average relative proof-length reduction (%) on competition and research-level benchmarks using Gemini 3 Flash.** "–" indicates the baseline could not be evaluated. Best results in bold.

	Competition			Research-level			
Method	**miniF2F**	**PutnamBench**	**Putnam2025**	**Analysis**	**FLT**	**PFR**	**PhysLean**
ProofOptimizer (Gu et al., 2025)	87.90	57.20	–	–	–	–	–
Base Agent	79.75	41.15	68.18	17.22	12.36	8.72	10.82
Base Agent w/o Retrieval	79.37	37.02	63.91	12.10	11.39	7.93	10.90
Lean Refactor w/o Retrieval	83.44	62.03	75.22	26.54	19.72	18.83	18.74
Lean Refactor w/ Random Retrieval	83.72	59.41	74.29	24.46	21.47	18.91	23.21
Lean Refactor	**88.40**	**70.38**	**79.27**	**33.54**	**23.55**	**20.37**	**27.69**

Table 2. **Average relative proof-length reduction (%) on PutnamBench and Analysis with Claude Haiku 4.5 as the backbone for both systems.** Claude Code is evaluated under three tool configurations of increasing access; budget is matched by dollar cost. Best result in bold.

Agent	**Configuration**	**PutnamBench**	**Analysis**
Claude Code	Lean MCP only	33.71	7.86
	+ Lean skills	33.29	8.83
	+ Our strategy retrieval tool	34.91	10.05
Ours	Base Agent	39.21	15.45
	Base Agent w/o Retrieval	36.89	10.88
	Lean Refactor w/o Retrieval	48.76	18.33
	Lean Refactor w/ Random Retrieval	46.94	18.59
	Lean Refactor	**51.35**	**21.09**

competition benchmarks; for research-level proofs, whose multi-file structure precludes clean per-proof wall-clock attribution, we instead measure heartbeats, Lean 4's deterministic proxy for elaboration work (Appendix L).

Lean Refactor w/ Reranking achieves the largest compilation-cost reduction on every competition and research-level benchmark (see Table 11 for research-level results), confirming that objective-aligned retrieval translates compile-cost metadata into wall-clock and heartbeat savings. miniF2F exposes the cost of objective *misalignment*: cosine-only retrieval yields a positive absolute reduction but a strongly negative relative reduction (-103.98%, i.e., compile time doubles on average in relative terms), driven by a few proofs whose compile time blows up by large factors, evidence that cosine-similarity-only retrieval can surface token-compressing yet compile-heavy strategies. Since ProofOptimizer's heartbeat-optimized proof variants are unreleased, we cannot measure their compilation time; therefore, Table 3 compares against their length-optimized proofs. However, under the heartbeat metric, **Lean Refactor w/ Reranking also beats ProofOptimizer's heartbeat-optimized numbers on miniF2F and PutnamBench** (Appendix K). More broadly, these results confirm the framework's adaptability: unlike models fine-tuned for a single objective, it is redirected across objectives by adjusting retrieval criteria over densely annotated metadata, with no retraining required.

4.3. Cross-Version Compatibility

The rapid release cycles of Lean and Mathlib make toolchain drift a recurring concern. We test (i) whether version-tagged retrieval metadata enables refactoring against arbitrary target Lean versions and transfers across LLM backbones, and (ii) whether refactored proofs remain type-checkable as Mathlib evolves past their source version.

Version-aware retrieval across Lean releases. We restrict cross-version evaluation to PutnamBench: research-level projects are pinned to specific Lean/Mathlib toolchains through deep intra-project dependencies, so changing versions breaks compilation regardless of refactoring quality, whereas PutnamBench proofs are self-contained. The lack of version-portable research benchmarks is a notable limitation for evaluating compatibility. We evaluate three Lean versions (v4.14.0, v4.16.0, v4.22.0) under three retrieval settings: *No Retrieval*, *Full Retrieval* (all strategies), and *Filtered Retrieval* (strategies whose compatibility set covers the target version). Mathlib-derived strategies are excluded since Mathlib is locked to specific toolchains. We run both Gemini 3 Flash and GPT-OSS-20B; the latter is chosen because its knowledge cutoff predates *every* version we test (Figure 3), making it a clean test of whether version-filtered retrieval can supply toolchain-specific knowledge the backbone lacks. Results are reported in Table 4.

Three findings emerge. **(1)** Filtered Retrieval attains the best length reduction on four of six (version, model) cells, and the pattern holds across both backbones. **(2)** Full Retrieval

Table 3. **Compilation-time reduction on miniF2F, PutnamBench, and Putnam2025.** *Rel.*: average relative % decrease in compile time. *Abs.*: average absolute decrease in seconds (higher is better for both). Each proof is compiled five times; subscripts denote standard deviations. ProofOptimizer is not evaluated on Putnam2025 (model weights not publicly released). Best per dataset in bold.

	miniF2F		PutnamBench		Putnam2025	
Method	**Rel.** ↑	**Abs.** ↑	**Rel.** ↑	**Abs.** ↑	**Rel.** ↑	**Abs.** ↑
ProofOptimizer (Gu et al., 2025)	$-12.89_{\pm 0.55}$	$0.97_{\pm 0.05}$	$21.14_{\pm 0.43}$	$8.16_{\pm 0.07}$	–	–
Lean Refactor w/o Retrieval	$26.56_{\pm 0.38}$	$3.02_{\pm 0.04}$	$36.32_{\pm 0.18}$	$7.05_{\pm 0.08}$	$49.53_{\pm 0.32}$	$4.98_{\pm 0.08}$
Lean Refactor w/ Random Retrieval	$28.42_{\pm 0.48}$	$3.11_{\pm 0.05}$	$35.10_{\pm 0.35}$	$6.77_{\pm 0.07}$	$50.13_{\pm 0.41}$	$4.99_{\pm 0.08}$
Lean Refactor	$-103.98_{\pm 2.64}$	$1.46_{\pm 0.06}$	$37.11_{\pm 0.75}$	$8.97_{\pm 0.11}$	$56.21_{\pm 0.34}$	$5.58_{\pm 0.06}$
Lean Refactor w/ Reranking	$\mathbf{30.17}_{\pm 0.64}$	$\mathbf{3.14}_{\pm 0.04}$	$\mathbf{42.62}_{\pm 0.27}$	$\mathbf{10.01}_{\pm 0.10}$	$\mathbf{60.14}_{\pm 0.45}$	$\mathbf{6.05}_{\pm 0.06}$

Table 4. **Cross-version compatibility on PutnamBench.** Average relative length reduction (%, higher is better). Best per (version, model) in bold. Budget is 30.

Version	Model	No Retrieval	Full Retrieval	Filtered Retrieval
v4.22.0	Gemini 3 Flash	55.99	55.43	**57.36**
	GPT-OSS-20B	27.56	27.80	**30.33**
v4.16.0	Gemini 3 Flash	60.61	60.41	**62.24**
	GPT-OSS-20B	27.11	28.37	**28.98**
v4.14.0	Gemini 3 Flash	58.51	**62.85**	59.54
	GPT-OSS-20B	**29.37**	29.14	27.58

Table 5. **Zero-shot version transfer on miniF2F.** Count of v4.19.0 proofs that still type-check at later releases without further refactoring. Best per row in bold. See Table 8 for the full results.

Version	Lean Refactor	ProofOptimizer	Original
v4.20.1	194	194	194
v4.22.0	**189**	188	187
v4.26.0	**168**	162	141
v4.29.0	**166**	161	140

shows no consistent gain over No Retrieval, suggesting that unfiltered semantic neighbors can inject incompatible artifacts that disrupt refactoring. **(3)** On GPT-OSS-20B, Filtered Retrieval still beats both baselines at v4.22.0 and v4.16.0 despite the lack of pretraining exposure to those toolchains, providing direct evidence that the strategy bank compensates for missing in-weights knowledge. The exception is v4.14.0, where gains are less consistent, which we attribute to a significantly shrunken pool of strictly backward-compatible strategies.

Zero-shot version transfer of refactored proofs. We further evaluate zero-shot version transfer of refactored proofs by recompiling v4.19.0 outputs from Lean Refactor, ProofOptimizer, and the original verbose proofs against Mathlib releases through v4.29.0, without further refactoring. On miniF2F, Lean Refactor leads from v4.22.0 onward and the gap widens at later releases (Table 5), showing that its refactored proofs continue to type-check across a longer span of future versions than both ProofOptimizer's proofs and the originals. On PutnamBench, Lean Refactor remains the strongest among proof optimization systems, though both shortened variants trail the originals, reflecting corpus-level differences in how proof length and version transfer interact. See Appendix G for full per-version results on both benchmarks.

5. Related Work

LLM-based provers (Ren et al., 2025; Lin et al., 2025b; Chen et al., 2025a) optimize correctness via RL but leave length, compile cost, and version compatibility unsupervised. Prior Lean refactoring (Ahuja et al., 2024; Gu et al., 2025) targets a single axis: ImProver needs per-objective example corpora, ProofOptimizer fine-tunes for length with compile cost demonstrated only as an alternative inference-time objective, and neither handles Lean/Mathlib version drift. General-purpose LLM code refactoring (Gautam et al., 2025; Gong et al., 2025b; Karabiyik, 2025; Bai et al., 2025; He et al., 2025; Gong et al., 2025a) assumes paired data, stable APIs, and test-suite validity, conditions Lean inverts. Lean Refactor externalizes refactoring knowledge into a shared strategy bank with compile-time and version metadata, turning new objectives into retrieval rules rather than training runs, with a model-agnostic, training-free agent loop. Extended related work is provided in Appendix A.

6. Conclusion

We present Lean Refactor, an LLM-agnostic framework that decouples Lean proof refactoring logic from the underlying model via a densely annotated strategy bank, steering frozen agentic LLMs to balance proof conciseness, compilation efficiency, and toolchain compatibility. Comprehensive evaluations across competition suites and research repositories show that metadata-aware retrieval is critical, and that the distilled strategies transfer across LLM backbones and even improve the general-purpose agent Claude Code. Ultimately, combining this retrieval mechanism with our specialized planner and compiler-in-the-loop architecture yields advanced refactoring capabilities that general-purpose coding agents struggle to match, laying the groundwork for more maintainable and version-resilient formal mathematical proofs. See Appendix C for discussions on limitations.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Achim, T., Best, A., Bietti, A., Der, K., Fédérico, M., Gukov, S., Halpern-Leistner, D., Henningsgard, K., Kudryashov, Y., Meiburg, A., et al. Aristotle: Imo-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025.

Ahuja, R., Avigad, J., Tetali, P., and Welleck, S. Improver: Agent-based automated proof optimization. *arXiv preprint arXiv:2410.04753*, 2024.

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

Anthropic. Claude 4.5 haiku model card, October 2025. URL https://www-cdn.anthropic.com/7aad69bf12627d42234e01ee7c36305dc2f6a970.pdf. Accessed: 2026-04-22.

Baanen, A., Ballard, M. R., Commelin, J., Chen, B. G.-g., Rothgang, M., and Testa, D. Growing mathlib: maintenance of a large scale mathematical library. In *International Conference on Intelligent Computer Mathematics*, pp. 51–70. Springer, 2025.

Bai, J., Xu, R., Wu, S., Yang, D., Zhao, J., and Chen, G. Polo: An llm-powered project-level code performance optimization framework. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI). IJCAI*, pp. 7319–7328, 2025.

Buzzard, K. and Taylor, R. FLT. https://github.com/ImperialCollegeLondon/FLT, 2025. URL https://github.com/ImperialCollegeLondon/FLT.

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

Chen, L., Gu, J., Huang, L., Huang, W., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Ma, K., et al. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025b.

Cui, B., Ramesh, T., Hernandez, O., and Zhou, K. Do large language models understand performance optimization? *arXiv preprint arXiv:2503.13772*, 2025.

De Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pp. 378–388. Springer, 2015.

DePalma, K., Miminoshvili, I., Henselder, C., Moss, K., and AlOmar, E. A. Exploring chatgpt's code refactoring capabilities: An empirical study. *Expert Systems with Applications*, 249:123602, 2024.

Gautam, D., Garg, S., Jang, J., Sundaresan, N., and Moghaddam, R. Z. Refactorbench: Evaluating stateful reasoning in language agents through code. *arXiv preprint arXiv:2503.07832*, 2025.

Gong, J., Giavrimis, R., Brookes, P., Voskanyan, V., Wu, F., Ashiga, M., Truscott, M., Basios, M., Kanthan, L., Xu, J., et al. Tuning llm-based code optimization via meta-prompting: An industrial perspective. *arXiv preprint arXiv:2508.01443*, 2025a.

Gong, J., Voskanyan, V., Brookes, P., Wu, F., Jie, W., Xu, J., Giavrimis, R., Basios, M., Kanthan, L., and Wang, Z. Language models for code optimization: Survey, challenges and future directions. *arXiv preprint arXiv:2501.01277*, 2025b.

Gu, A., Piotrowski, B., Gloeckle, F., Yang, K., and Markosyan, A. H. Proofoptimizer: Training language models to simplify proofs without human demonstrations. *arXiv preprint arXiv:2510.15700*, 2025.

He, X., Liu, Q., Du, M., Yan, L., Fan, Z., Huang, Y., Yuan, Z., and Ma, Z. Swe-perf: Can language models optimize code performance on real-world repositories? *arXiv preprint arXiv:2507.12415*, 2025.

Karabiyik, M. A. Refactorgpt: a chatgpt-based multi-agent framework for automated code refactoring. *PeerJ Computer Science*, 11:e3257, 2025.

Li, J., Beeching, E., Tunstall, L., Lipkin, B., Soletskyi, R., Huang, S., Rasul, K., Yu, L., Jiang, A. Q., Shen, Z., Qin, Z., Dong, B., Zhou, L., Fleureau, Y., Lample, G., and Polu, S. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. https://github.com/

project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf, 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., Wu, J., Gesi, J., Lu, X., Acuna, D., Yang, K., Lin, H., Choi, Y., Chen, D., Arora, S., and Jin, C. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025b. URL https://arxiv.org/abs/2508.03613.

Liu, X., Bao, K., Zhang, J., Liu, Y., Chen, Y., Liu, Y., Jiao, Y., and Luo, T. Atlas: Autoformalizing theorems through lifting, augmentation, and synthesis of data, 2025. URL https://arxiv.org/abs/2502.05567.

Lu, J., Emond, K., Yang, K., Chaudhuri, S., Sun, W., and Chen, W. Lean finder: Semantic search for mathlib that understands user intents. *arXiv preprint arXiv:2510.15940*, 2025.

mathlib4. Dealing with breakages from updating. https://github.com/leanprover-community/mathlib4/wiki/Using-mathlib4-as-a-dependency#dealing-with-breakages-from-updating, 2025. GitHub repository.

Moura, L. d. and Ullrich, S. The Lean 4 theorem prover and programming language. 2021.

OpenAI, :, Agarwal, S., Ahmad, L., Ai, J., Altman, S., Applebaum, A., Arbus, E., Arora, R. K., Bai, Y., Baker, B., Bao, H., Barak, B., Bennett, A., Bertao, T., Brett, N., Brevdo, E., Brockman, G., Bubeck, S., Chang, C., Chen, K., Chen, M., Cheung, E., Clark, A., Cook, D., Dukhan, M., Dvorak, C., Fives, K., Fomenko, V., Garipov, T., Georgiev, K., Glaese, M., Gogineni, T., Goucher, A., Gross, L., Guzman, K. G., Hallman, J., Hehir, J., Heidecke, J., Helyar, A., Hu, H., Huet, R., Huh, J., Jain, S., Johnson, Z., Koch, C., Kofman, I., Kundel, D., Kwon, J., Kyrylov, V., Le, E. Y., Leclerc, G., Lennon, J. P., Lessans, S., Lezcano-Casado, M., Li, Y., Li, Z., Lin, J., Liss, J., Lily, Liu, Liu, J., Lu, K., Lu, C., Martinovic, Z., McCallum, L., McGrath, J., McKinney, S., McLaughlin, A., Mei, S., Mostovoy, S., Mu, T., Myles, G., Neitz, A., Nichol, A., Pachocki, J., Paino, A., Palmie, D., Pantuliano, A., Parascandolo, G., Park, J., Pathak, L., Paz, C., Peran, L., Pimenov, D., Pokrass, M., Proehl, E., Qiu, H., Raila, G., Raso, F., Ren, H., Richardson, K., Robinson, D., Rotsted, B., Salman, H., Sanjeev, S., Schwarzer, M., Sculley, D., Sikchi, H., Simon, K., Singhal, K., Song, Y., Stuckey, D., Sun, Z., Tillet, P., Toizer, S., Tsimpourlas, F., Vyas, N., Wallace, E., Wang, X., Wang, M., Watkins, O., Weil, K., Wendling, A., Whinnery, K., Whitney, C., Wong, H., Yang, L., Yang, Y., Yasunaga, M., Ying, K., Zaremba, W., Zhan, W., Zhang, C., Zhang, B., Zhang, E., and Zhao, S. gpt-oss-120b gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Ospanov, A., Farnia, F., and Yousefzadeh, R. Apollo: Automated llm and lean collaboration for advanced formal reasoning. *arXiv preprint arXiv:2505.05758*, 2025.

Oueslati, K., Lamothe, M., and Khomh, F. Refagent: A multi-agent llm-based framework for automatic software refactoring. *arXiv preprint arXiv:2511.03153*, 2025.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Peng, Z., Yao, Y., Ma, K., Guo, S., Li, Y., Zhang, Y., Zhang, C., Zhang, Y., Yu, Z., Li, L., Liu, M., Xia, Y., Shen, J., Wu, Y., Cao, Y., Zhang, Z., Huang, W., Liu, J., and Zhang, G. Criticlean: Critic-guided reinforcement learning for mathematical formalization, 2025. URL https://arxiv.org/abs/2507.06181.

Piao, Y. C. K., Paul, J. C., Da Silva, L., Dakhel, A. M., Hamdaqa, M., and Khomh, F. Refactoring with llms: Bridging human expertise and machine understanding. *arXiv preprint arXiv:2510.03914*, 2025.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

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.

Tao, T. A Lean companion to Analysis I. https://github.com/teorth/analysis, 2024. URL https://github.com/teorth/analysis.

Tao, T. Formalization of the Polynomial Freiman-Ruzsa conjecture of Marton. https://github.com/teorth/pfr, 2025. URL https://github.com/teorth/pfr.

The Gemini Team. Gemini 3 flash: frontier intelligence built for speed. https://blog.google/products-and-platforms/products/gemini/gemini-3-flash/, December 2025. Accessed: 2026-03-16.

Tooby-Smith, J. physlib: A project to digitalise results from physics into Lean. https://github.com/leanprover-community/physlib, 2025. URL https://github.com/leanprover-community/physlib.

Trinh, T. H., Wu, Y., Le, Q. V., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

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, 2024. URL https://arxiv.org/abs/2407.11214.

Wu, Y., Han, M., Li, R., Liang, P., Tahir, A., Li, Z., Feng, Q., and Shahin, M. Fasterpy: An llm-based code execution efficiency optimization framework. *arXiv preprint arXiv:2512.22827*, 2025.

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.

Yang, J., Lu, S., Liu, H., Bagi, S. S. G., Fazel, Z., Czajkowski, T., and Niu, D. Perfcoder: Large language models for interpretable code performance optimization. *arXiv preprint arXiv:2512.14018*, 2025.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Zhang, Y., Li, M., Long, D., Zhang, X., Lin, H., Yang, B., Xie, P., Yang, A., Liu, D., Lin, J., Huang, F., and Zhou, J. Qwen3 embedding: Advancing text embedding and reranking through foundation models, 2025. URL https://arxiv.org/abs/2506.05176.

Zhao, Y., Xiao, Y.-A., Xiao, Q., Zhang, Z., and Xiong, Y. Semopt: Llm-driven code optimization via rule-based analysis. *arXiv preprint arXiv:2510.16384*, 2025.

Zheng, K., Han, J. M., and Polu, S. Minif2f: a cross-system benchmark for formal olympiad-level mathematics, 2022. URL https://arxiv.org/abs/2109.00110.

A. Extended Related Work

LLM-based theorem proving in Lean. A rapid line of work trains LLMs to generate Lean proofs end-to-end with reinforcement learning from compiler-verified rewards, including DeepSeek-Prover (Xin et al., 2024a;b; Ren et al., 2025), Goedel-Prover (Lin et al., 2025a;b), Seed-Prover (Chen et al., 2025b;a), building on earlier tactic-level and retrieval-augmented systems (Polu & Sutskever, 2020; Yang et al., 2023). These rewards, however, primarily target proof correctness, leaving length, compilation cost, and version compatibility unsupervised. The resulting proofs are correct but verbose, slow to compile, and brittle across Mathlib releases (Ahuja et al., 2024; Gu et al., 2025). Our framework operates downstream of these provers, refactoring their outputs under multiple cost objectives without retraining the underlying model.

Lean proof optimization. Two prior systems directly target Lean 4 proof simplification. ImProver (Ahuja et al., 2024) wraps a black-box LLM in an agentic loop combining Chain-of-States prompting, best-of-n sampling with iterative refinement, and retrieval over Lean/Mathlib documentation and per-metric example databases. ProofOptimizer (Gu et al., 2025) instead fine-tunes a dedicated model on synthesized long-short pairs to directly shorten proofs. While both achieve length reductions, each restricts refactoring to a single-objective paradigm. ProofOptimizer's objective is frozen at training time. ImProver accommodates arbitrary user-defined metrics by using them to drive both example retrieval and candidate selection; however, scaling to new objectives requires constructing an entirely new, metric-specific example corpus. Crucially, neither system addresses the broader bottlenecks of Lean 4 proof maintenance identified in Section 2. Compilation cost is treated as a secondary objective swapped in at inference rather than jointly optimized (Gu et al., 2025), and continuous Lean 4 and Mathlib version drift remains invisible to both frozen fine-tuned models and standard example-based retrievers. Lean Refactor overcomes these limitations by retrieving *refactoring strategies* densely annotated with multi-objective execution metadata, such as expected compilation-time reductions and validated Lean 4 version compatibility. This enables Lean Refactor to turn complex, competing objectives into configurable retrieval rules over a single, shared strategy bank, bypassing the need for training-time commitments or fractured, per-metric datasets.

LLM-based code refactoring and performance optimization. A broader literature studies LLM-driven refactoring (Gautam et al., 2025; Gong et al., 2025b) and code performance optimization for general-purpose languages. Agentic and prompting-based systems target maintainability and code quality (DePalma et al., 2024; Piao et al., 2025; Oueslati et al., 2025; Karabiyik, 2025), while a parallel thread pursues runtime speedup (Bai et al., 2025; He et al., 2025; Cui et al., 2025; Zhao et al., 2025; Wu et al., 2025; Yang et al., 2025; Gong et al., 2025a). These methods operate on settings such as Python, C++, and Java, where paired refactoring data is plentiful, APIs are comparatively stable, and correctness is verified by test suites rather than a strict type checker. Lean inverts each of these conditions: paired data is scarce, the toolchain breaks compilation weekly, and validity is a hard binary signal. Lean Refactor's agentic system is distinguished by the densely annotated strategy bank and a compiler-in-the-loop validation tailored to Lean's release dynamics.

B. Details about Strategy Bank Construction

This section provides a complete description of the data pipeline summarized in Section 3.1. We detail (i) the source repositories from which formal statements are aggregated, (ii) the dual-strategy procedure used to synthesize aligned long–short proof pairs, (iii) the profiling and cross-version recompilation pipeline that produces the refactoring metadata, (iv) the prompting protocol used to distill location-grounded refactoring strategies from each pair, (v) the quality-control filter, (vi) the iterative embedding-based deduplication pipeline, and (vii) the procedure by which pair-level metadata is aggregated into strategy-level annotations.

B.1. Source Repositories of Lean Statements

To ensure that our strategy bank generalizes across both competition mathematics and a broad spectrum of undergraduate- and research-level domains, we aggregate formal statements from four complementary repositories.

- **NuminaMath-1.5 (Li et al., 2024).** A large-scale corpus of formalized competition problems. NuminaMath provides a high density of challenging Olympiad-style Lean statements that exercise the full range of tactical machinery typically employed in contest proofs.
- **FineLeanCorpus (Peng et al., 2025).** A second large-scale source of competition-level statements, complementary to NuminaMath in style and topical coverage.

- **Mathlib4.** The standard library for the Lean 4 ecosystem. We extract theorems from `Mathlib4` to represent broader mathematical disciplines that are underrepresented in competition-oriented corpora.
- **ATLAS (Liu et al., 2025).** A synthetic dataset of theorem statements lifted from `Mathlib4` contexts. ATLAS broadens the stylistic distribution of statements beyond what is naturally present in human-curated corpora and serves as a useful augmentation for diversity.

This hybrid sourcing strategy exposes the downstream pipeline to a wide range of Lean coding styles, tactic preferences, and proof-architecture patterns. The held-out evaluation set used to assess generalization is described separately in the experiments section of the main paper.

Decontamination against the held-out evaluation set. To preclude leakage from our held-out evaluation set into the long–short proof pairs used to build the strategy bank, we apply a two-stage decontamination filter to every theorem statement before proof synthesis. First, we perform exact-match deduplication on normalized statements. Second, we perform embedding-based near-duplicate detection: each candidate training statement is encoded with Qwen3-Embedding-8B (Zhang et al., 2025) and compared against the embeddings of all evaluation statements. If the top-1 cosine similarity is below 0.8, we treat the candidate as non-duplicate; if it is at or above 0.8, we invoke GPT-OSS-120B as an LLM-as-judge with the candidate and the matched evaluation statement to make the final determination, and discard the candidate if the judge confirms duplication. This filter is applied upstream of proof synthesis and strategy distillation, so the resulting 200K long–short pairs, the 481,567 raw strategy extractions, and the final corpus of 9,237 unique strategies are all derived exclusively from theorems that survive both stages.

B.2. Proof Synthesis and Long–Short Pair Construction

Proof generation. With the exception of `Mathlib4`, the sourced repositories provide only theorem statements without ground-truth proofs. We therefore synthesize formal proofs using two prover models: Goedel-Prover-V2-32B (Lin et al., 2025b) and GPT-OSS-120B (OpenAI et al., 2025). Using two provers rather than one is a deliberate design choice: it broadens the distribution of tactic patterns, naming conventions, and structural choices that subsequently appear in the long–short pairs and, by extension, in the distilled strategies.

For each theorem statement we sample between one and three candidate proofs per model, with the exact count bounded by our compute budget. To guarantee environmental consistency during verification, every candidate is prepended with a standardized header containing the necessary `import` directives and `namespace` declarations before being submitted to the Lean 4 compiler. When multiple candidate proofs for the same theorem successfully verify, we retain the *longest* valid proof, since verbose proofs provide the richest foundation for refactoring and yield the most informative long–short pairs in the subsequent stage.

Aligned long–short pair construction. Given a corpus of verified proofs, we construct aligned pairs through a length-conditioned, dual-strategy prompting pipeline:

- **Forward simplification (long $\rightarrow$ short).** For proofs exceeding 50 tokens, we prompt GPT-OSS-120B to optimize the trajectory and produce a shorter counterpart.
- **Reverse complexification (short $\rightarrow$ long).** For proofs under 50 tokens, we prompt GPT-OSS-120B to synthetically expand the proof into a more verbose, less idiomatic counterpart, yielding the "long" element of the pair.

The reverse-complexification branch ensures that compact, idiomatic proofs, which would otherwise be excluded because no naturally occurring "long" counterpart exists, still contribute to the bank by surfacing the simplifications they implicitly embody.

To improve structural fidelity in both directions, we augment the prompts with explicit dependency information that enumerates the lemmas and premises invoked in the input proof. Empirically, this dependency context reduces hallucinated lemma references and produces transformations that are more faithful to the underlying proof structure. Each candidate transformed proof is recompiled to confirm that the produced pair consists of two independently verified Lean proofs of the same theorem.

In total this pipeline yields **200K verified long–short proof pairs**, which form the foundation for all subsequent strategy distillation.

B.3. Refactoring Metadata Annotation

Each long–short pair is annotated along two axes that the LLM cannot infer from its training data alone: compilation-time reduction and Lean-toolchain compatibility.

Compilation-time profiling. We measure the compilation time of both the long and the short proof in each pair using the `lake env lean --profile` command, which exposes fine-grained per-declaration timing. We explicitly isolate proof execution from library import overhead, since import time is dominated by caching effects unrelated to the proof content itself. Each measurement is taken in a consistent environment to ensure stable measurements, and we record the relative reduction in execution time achieved by the shortened proof. This per-pair speedup is the unit from which strategy-level estimates are later aggregated (Appendix B.8).

Cross-version compatibility testing. To assess whether a refactoring remains valid under future Lean releases, we recompile each shortened proof under multiple Lean and `Mathlib` toolchains beyond the original v4.24.0: `v4.14.0`, `v4.16.0`, and `v4.22.0`. We restrict version-compatibility analysis to non-`Mathlib` theorems, because `Mathlib` itself is tightly coupled to a specific Lean release via `lean-toolchain` and routinely fails to build under manual version overrides; this coupling makes any cross-version signal collected on `Mathlib` theorems an artifact of infrastructure rather than of the refactoring strategy under study.

For each eligible pair we track the compilation status of both the long and short proof under each toolchain. A refactored proof is deemed compatible with a target version if the short proof compiles successfully in that version. The strategy is considered robust on the interval spanned by the toolchains under which all of its shortened proofs continue to compile, a notion that is formalized at the strategy level in Appendix B.8.

B.4. Strategy Summarization with Location Grounding

A single long–short proof pair typically encompasses several distinct refactoring operations applied to different regions of the code. To capture this granularity, we prompt GPT-OSS-120B with both proofs and ask it to extract a set of applied refactoring strategies. Crucially, we instruct the model to ground each extracted strategy to a specific component of the original long proof, identified by start and end line numbers.

This grounding requirement serves two purposes:

1. *Hallucination control.* In preliminary experiments we observed that without an explicit anchoring constraint, the model frequently fabricated strategies that did not correspond to any concrete change between the long and short proofs. Forcing each strategy to point to a specific code region eliminates a large class of these hallucinations and improves extraction quality.
2. *Retrieval supervision.* The proof-component-to-strategy mapping is the supervision signal used to fine-tune the strategy-retrieval model employed by our agent at inference time. Given a proof component currently being refactored, the retriever returns the strategies historically associated with similar components, providing contextually relevant guidance.

Importantly, location grounding does not bias the extracted strategies toward purely localized fixes. Global structural refactorings, such as restructuring an entire case split or replacing a multi-step induction with a single library lemma, are accommodated by grounding the strategy to a correspondingly larger contiguous chunk of the proof.

B.5. Strategy Schema

To ensure clarity, applicability, and reusability, each distilled strategy is organized into the following fields. Together they form a self-contained record sufficient for the agent to reason about *when*, *why*, and *how* to apply the refactoring at inference time. Examples are provided in Appendix M.

- **Title.** A short natural-language summary of the high-level idea of the refactoring strategy.

- **Description.** A natural-language statement of the core conceptual idea behind the refactoring. It includes information like the semantic shift that the strategy brings (e.g. replacing a manual step-by-step existence construction or a redundant case analysis with a more powerful library lemma or automated tactic).
- **When to Apply.** A generalized description of the Lean code pattern or logical situation that serves as a precondition for the strategy. This field explicitly identifies the structural "anti-patterns" that warrant the proposed refactoring and enables the agent to recognize applicable contexts in unseen proofs.
- **Application Guide.** A step-by-step, actionable refactoring guide. It provides the agent with deterministic, sequential instructions for executing the transformation, minimizing ambiguity at the point of application.
- **Abstract Example.** A generalized exemplar that supplies a concrete "before" and "after" snippet of the transformation. Specific variable names and domain artifacts are abstracted away, so that the example highlights the underlying syntactic and tactical shift rather than the surface details of any one theorem.
- **Potential Reduction.** A categorical estimate (*high*, *medium*, or *low*) of the expected impact on proof length.

B.6. Quality Filtering via LLM-as-Judge

To guarantee both the correctness of the distilled strategies and the fidelity of their location grounding, we apply a rigorous filtering stage. Each extracted strategy is evaluated by an LLM-as-judge alongside its corresponding long–short proof pair and the identified proof component. The judge assesses two criteria:

1. *Transformation correctness.* Whether the proposed strategy correctly and logically transforms the identified component of the long proof into its optimized counterpart in the short proof.
2. *Schema fidelity.* Whether all six fields of the strategy schema (Appendix B.5) are accurately and consistently populated, with no contradictions between, e.g., the *When to Apply* precondition and the *Abstract Example*.

Only strategies passing both checks are retained for the next stage. Strategies that fail either check are discarded; we do not attempt to repair them, as our preliminary experiments indicated that repair attempts frequently introduce subtler errors that propagate downstream.

B.7. Clustering and Iterative Deduplication

The filtered set still contains substantial redundancy: many distinct long–short pairs surface the same underlying refactoring idea, expressed with surface-level variation. To prevent redundant strategies from consuming valuable prompt context at inference time, and to keep retrieval efficient, we curate a compact, semantics-aware subset of unique strategies via the following iterative pipeline.

Initialization. We seed the unique-strategy set by prompting GPT-OSS-120B to identify a diverse initial collection of canonical strategies from a sampled subset of the pool. This produces a high-coverage starting point against which subsequent strategies can be compared.

Iterative deduplication. Each remaining strategy is then processed sequentially against the dynamically growing unique set via a four-stage cascade:

1. *Embedding and candidate retrieval.* We encode the strategy's *Description* field with the Qwen3-Embedding-8B (Zhang et al., 2025) model and retrieve the top-10 most semantically similar entries from the current unique set by cosine similarity.
2. *High-confidence shortcut.* If the top-1 cosine similarity is ≥ 0.9, the strategy is treated as a duplicate of the matched entry without further inspection. We adopt this shortcut because manual inspection of borderline cases indicated that pairs above this threshold are essentially always paraphrases of the same underlying refactoring, and skipping the judge call at this end of the distribution yields a substantial reduction in pipeline cost without a measurable loss in cluster purity.

3. *Judge-based duplicate detection.* If the top-1 cosine similarity is < 0.9, we prompt GPT-OSS-120B as an LLM-as-judge with the current strategy together with the 10 retrieved candidates. The judge decides whether the current strategy is a semantic duplicate of any existing entry. This stage handles the long tail of paraphrastic variation that pure embedding similarity tends to underestimate (e.g., strategies that share an underlying tactical idea but use different terminology or describe it at different levels of abstraction).

4. *Update.* If either the shortcut or the judge identifies a duplicate, the current strategy is *clustered* under the matched existing entry, contributing its pair-level metadata to that cluster's aggregate (Appendix B.8). Otherwise, the strategy is appended to the unique set as a new canonical entry and becomes a candidate for matching against subsequent strategies.

By combining embedding-based shortlisting, a high-similarity shortcut, and an LLM judge fallback, this cascade keeps the procedure tractable on the full filtered pool while still benefiting from the judge's ability to resolve paraphrastic variation.

Final corpus. Applying this pipeline to the filtered pool of 481,567 raw extractions yields a refined corpus of **9,237 unique refactoring strategies**, each backed by a cluster of constituent long–short pairs that share its semantics.

B.8. Strategy-Level Metadata Aggregation

The pair-level metadata collected in Appendix B.3 is lifted to the strategy level by aggregating across the cluster of long–short pairs assigned to each unique strategy.

Compilation-time reduction. For each strategy, we keep the median relative execution-time reduction across all long–short pairs in its cluster to get a robust strategy-level estimate.

Version compatibility set. Following Appendix B.3, we restrict attention to non-`Mathlib` pairs. A strategy's *compatibility set* is defined as the intersection of toolchains under which the shortened proofs of all its constituent pairs compile successfully. This intersection-based definition is conservative by design: a strategy is certified for a toolchain only if every observed instantiation of the strategy survives recompilation under that toolchain. Aggregating across all constituent pairs, rather than relying on a single example, ensures that a toolchain enters the compatibility set only when the strategy has been shown to transfer reliably across diverse instantiations, rather than happening to compile in one isolated case.

Downstream use. The aggregated metadata is the lever that makes the bank *version-aware* and *efficiency-aware* at inference time: the multi-objective retrieval procedure (Section 3.2) uses the compatibility set to filter strategies incompatible with the user's target toolchain and the median speedup to rank surviving candidates. This is the mechanism by which we decouple refactoring reliability from the LLM's training cutoff.

C. Limitations

Benchmark coverage for cross-version evaluation. Our cross-version compatibility experiments (Section 4.3) are confined to PutnamBench because, to our knowledge, no existing Lean benchmark supports compatibility evaluation beyond self-contained competition problems. Research-level projects pin to specific Lean/Mathlib toolchains through deep intra-project dependencies, so changing the toolchain breaks compilation regardless of refactoring quality. Constructing version-portable research-level benchmarks, e.g., proofs whose dependencies are at a granularity that decouples them from the project's pinned toolchain, is an important direction for the community.

Joint-effect attribution. Strategy-level compile-time annotations are computed as the cluster median of pair-level reductions, where each pair contributes the joint speedup of its refactorings; analogously, version-compatibility sets are intersected across constituent pairs, so a strategy inherits the toolchain coverage that all its instantiations jointly satisfy. Both choices fit the role the metadata plays at inference: the retrieval module reranks candidates by *relative* compile-time benefit and filters by guaranteed version coverage, for which the median and intersection are robust, conservative estimators. Disentangling marginal per-strategy contributions, via Shapley-style attribution, or regression over co-occurrence indicators, would further enrich the schema along a new axis and is a natural extension we leave to future work.

D. An Example of Proof Length and Compilation Time Tradeoff

To illustrate the misalignment of proof length and compilation time, consider the two alternative Lean proofs for the theorem `amc12_2001_p21` from miniF2F (Zheng et al., 2022) below.

```
theorem amc12_2001_p21
  (a b c d : N)
  (h0 : a * b * c * d = Nat.factorial 8)
  (h1 : a * b + a + b = 524)
  (h2 : b * c + b + c = 146)
  (h3 : c * d + c + d = 104) :
  a - d = (10 : Z) := by
  norm_num [Nat.factorial] at h0
  have : b <= 525 := by nlinarith
  interval_cases b <;> simp_all
  <;> nlinarith
```

```
theorem amc12_2001_p21
  (a b c d : N)
  (h0 : a * b * c * d = Nat.factorial 8)
  (h1 : a * b + a + b = 524)
  (h2 : b * c + b + c = 146)
  (h3 : c * d + c + d = 104) :
  a - d = (10 : Z) := by
  -- Factor hypotheses into (x+1)(y+1) = xy +
     x + y + 1
  have h4 : (a+1)*(b+1) = 525 := by
    -- proof omitted
  have h5 : (b+1)*(c+1) = 147 := by
    -- proof omitted
  have h6 : (c+1)*(d+1) = 105 := by
    -- proof omitted
  -- Pin down each variable via gcd +
    interval_cases
  have h7  : b = 20 := by -- proof omitted
  have h8  : a = 24 := by -- proof omitted
  have h9  : c = 6  := by -- proof omitted
  have h10 : d = 14 := by -- proof omitted
  -- Cast to integers and conclude
  have h11 : (a:Z) - (d:Z) = 10 := by
    -- proof omitted
  exact_mod_cast h11
```

The short proof is concise but relies on a single heavy cascade `interval_cases, simp_all, nlinarith` over the loose bound $b \leq 525$, so Lean's most expensive tactics fire on hundreds of branches. Conversely, the second proof spans over 130 lines of explicit, step-by-step arithmetic deductions. Despite its much longer text, it significantly reduces the compiler's search burden and compiles more than $20\times$ faster than its shorter counterpart. This stark contrast highlights that optimizing LLMs solely for shorter text lengths can drastically penalize underlying runtime metrics.

E. Retrieval Model Training Details

We initialize our strategy search model using Qwen3-Embedding-8B (Zhang et al., 2025). We fine-tune the model with a contrastive learning objective, using in-batch negative sampling and a margin-based false-negative filtering mechanism. We train the model for 1 epoch, a per-device batch size of 16, gradient accumulation steps of 2, and 4 NVIDIA 6000Ada GPUs. We use the AdamW optimizer with a learning rate of 6×10^{-6} and $\epsilon = 1 \times 10^{-8}$. The learning rate is linearly warmed up over the first 300 steps, then decayed to 0 using a cosine schedule. Temperature τ is set to 0.01, and the margin m is set to 0.1.

F. Proof Dependency Reduction

An essential measure of formal proof elegance and maintainability is a minimal reliance on external lemmas. Analyzing the length-optimized proofs from Section 4.1, we find that Lean Refactor consistently reduces dependencies, defined as statements invoked from external libraries (e.g., Mathlib) or distinct intra-project modules. As shown in Tables 6 and 7, dependencies decreased in 64.0% of PutnamBench and 51.0% of miniF2F proofs. Remarkably, this trend holds even in complex research-level mathematics typically entangled with extensive library architectures, yielding reductions in 66.7% of the Analysis and 60.0% of the FLT datasets. Across all benchmarks, cases where the agent increases dependencies remain rare.

This systematic reduction reveals that Lean Refactor achieves brevity through genuine logical simplification rather than merely offloading complexity to heavy, specialized library calls. By discovering direct pathways from local premises, the

agent naturally produces highly self-contained proofs. This structural shift provides a significant practical advantage by eliminating the burden of memorizing or invoking vast numbers of external lemma names. Consequently, these optimized proofs are significantly more readable for human mathematicians, easier to maintain, and inherently more resilient to the cross-version library breakages discussed in Section 4.3.

Table 6. Comparison of Proof Dependency Counts across miniF2F, PutnamBench, and Putnam 2025 datasets. Reduced Dependencies indicate a reduction in the number of external (e.g., Mathlib) or intra-project dependencies required for the proof.

Comparison Metric	**miniF2F**	**PutnamBench**	**Putnam 2025**
Total Theorems Compared	194	75	66
Reduced Dependencies	99 (51.0%)	48 (64.0%)	29 (43.9%)
Increased Dependencies	11 (5.7%)	11 (14.7%)	12 (18.2%)
No Change	84 (43.3%)	16 (21.3%)	25 (37.9%)

Table 7. Comparison of Proof Dependency Counts across Analysis, FLT, PFR, and Physlean datasets. Reduced Dependencies indicate a reduction in the number of external (e.g., Mathlib) or intra-project dependencies required for the proof.

Comparison Metric	**Analysis**	**FLT**	**PFR**	**Physlean**
Total Theorems Compared	45	45	45	45
Reduced Dependencies	30 (66.7%)	27 (60.0%)	24 (53.3%)	19 (42.2%)
Increased Dependencies	8 (17.7%)	11 (24.4%)	7 (15.6%)	11 (24.5%)
No Change	7 (15.6%)	7 (15.6%)	14 (31.1%)	15 (33.3%)

G. Zero-shot Version Transfer Details

We complement the retrieval-side analysis with a passive audit of zero-shot version transfer: we take the proofs produced on a single source version (v4.19.0) and compile them against every subsequent Mathlib release through v4.29.0, recording how many still type-check without further refactoring. We compare our system (LeanRefactor) against the original human-written proofs and against ProofOptimizer, a shortening baseline run from the same source version. This probes whether the act of refactoring itself alters how well a proof transfers across Lean/Mathlib releases, independent of any retrieval-time filtering. Table 8 reports the results.

Table 8. **Zero-shot version transfer of refactored proofs.** Number of v4.19.0 proofs from each system that still type-check at each later Mathlib release without further refactoring. Totals: 75 (PutnamBench), 194 (MiniF2F).

Mathlib Version	**PutnamBench**			**MiniF2F**		
	LeanRefactor	ProofOptimizer	Original	LeanRefactor	ProofOptimizer	Original
v4.20.1	71	70	71	194	194	194
v4.21.0	69	70	**71**	193	194	194
v4.22.0	64	63	64	**189**	188	187
v4.23.0	51	50	**61**	**175**	170	151
v4.24.0	50	50	**60**	**172**	167	148
v4.25.0	49	48	**58**	**168**	162	141
v4.26.0	48	48	**58**	**168**	162	141
v4.27.0	45	45	**53**	**167**	162	140
v4.28.0	45	45	**53**	**166**	161	140
v4.29.0	43	42	**52**	**166**	161	140

The comparison against original long proofs is benchmark-dependent: on MiniF2F, LeanRefactor transfers furthest among all baselines, whereas on PutnamBench it falls short of the originals and tracks ProofOptimizer closely. We therefore conclude that whether shortening substitutes tactics that transfer better or worse across releases depends on corpus composition.

H. Proof Dependencies in Research Repositories and Competition Problems

Table 9. Competition-type Lean sources rely on much fewer dependencies (Mathlib and intra-project) than Research-type Lean sources.

Type	Lean Source	Average Number of Dependencies Used
Competition	miniF2F (Zheng et al., 2022)	2.66
	PutnamBench (Tsoukalas et al., 2024)	10.74
Research	FLT (Buzzard & Taylor, 2025)	39.07
	PFR (Tao, 2025)	40.64
	PhysLean (Tooby-Smith, 2025)	38.53
	Analysis (Tao, 2024)	35.38

I. Experimental Settings

This section provides full details for the experiments summarized in Section 4. We describe (i) the seven evaluation benchmarks and how their initial long proofs are obtained, (ii) the metrics and the rationale for the benchmarks each metric is computed on, (iii) the baseline configurations used for the controlled ablation and the Claude Code comparison, and (iv) the implementation and hyperparameter settings shared across all runs.

I.1. Benchmarks

Competition problems. We evaluate on three competition benchmarks.

- **miniF2F** (Zheng et al., 2022) (194 theorems). Initial long proofs are taken directly from the public release of our primary baseline ProofOptimizer (Gu et al., 2025), ensuring identical inputs across both methods.
- **PutnamBench** (Tsoukalas et al., 2024) (75 theorems). Initial long proofs are likewise taken from the ProofOptimizer release for direct comparability.
- **Putnam2025** (66 theorems). We construct this benchmark from the official Lean 4 solutions to problems B1 and B2 of the 2025 Putnam Competition produced by Seed-Prover 1.5 (Chen et al., 2025a). We restrict attention to B1/B2 due to budget. Each official solution is provided as a self-contained Lean 4 file; we extract the individual theorems and track the dependencies invoked in the original proof, yielding 66 theorems.

The length distribution of the initial proofs across all three competition benchmarks is shown in Figure 6.

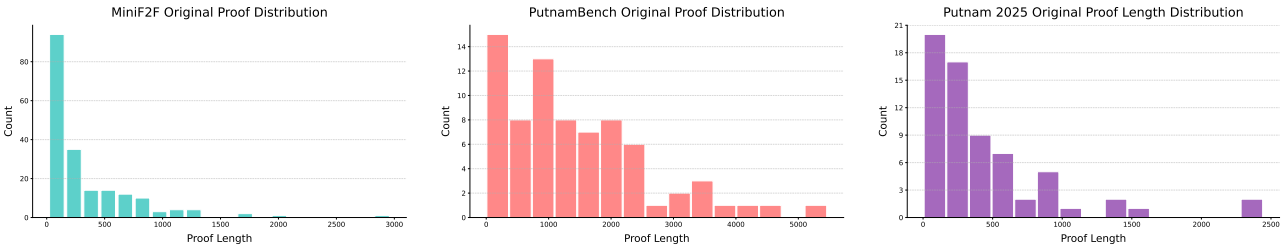


Figure 6. Distribution of original proof lengths across three competition style evaluation sets (miniF2F, PutnamBench, Putnam 2025).

Research-level problems. To assess refactoring on advanced mathematics, we draw theorems from four research-level Lean 4 repositories: Analysis (Tao, 2024), Fermat's Last Theorem (FLT) (Buzzard & Taylor, 2025), the Polynomial Freiman–Ruzsa Conjecture (PFR) (Tao, 2025), and PhysLean (Tooby-Smith, 2025). From each repository we sample 45 theorems, subject to compute budget. To evaluate refactoring effectiveness across a wide range of initial proof complexities, we stratify the sample by proof length:

- 15 proofs exceeding 1000 tokens,
- 15 proofs between 500 and 1000 tokens,

- 10 proofs between 100 and 500 tokens,
- 5 proofs under 100 tokens.

When a repository contains insufficient theorems in a given length bucket, the unfilled quota is reallocated to the nearest non-empty bucket. The extraction pipeline retains all internal and external dependencies, ensuring that the agent operates on a properly contextualized, dependency-aware proof environment. The length distribution of the initial proofs across the four research-level evaluation sets is shown in Figure 7.

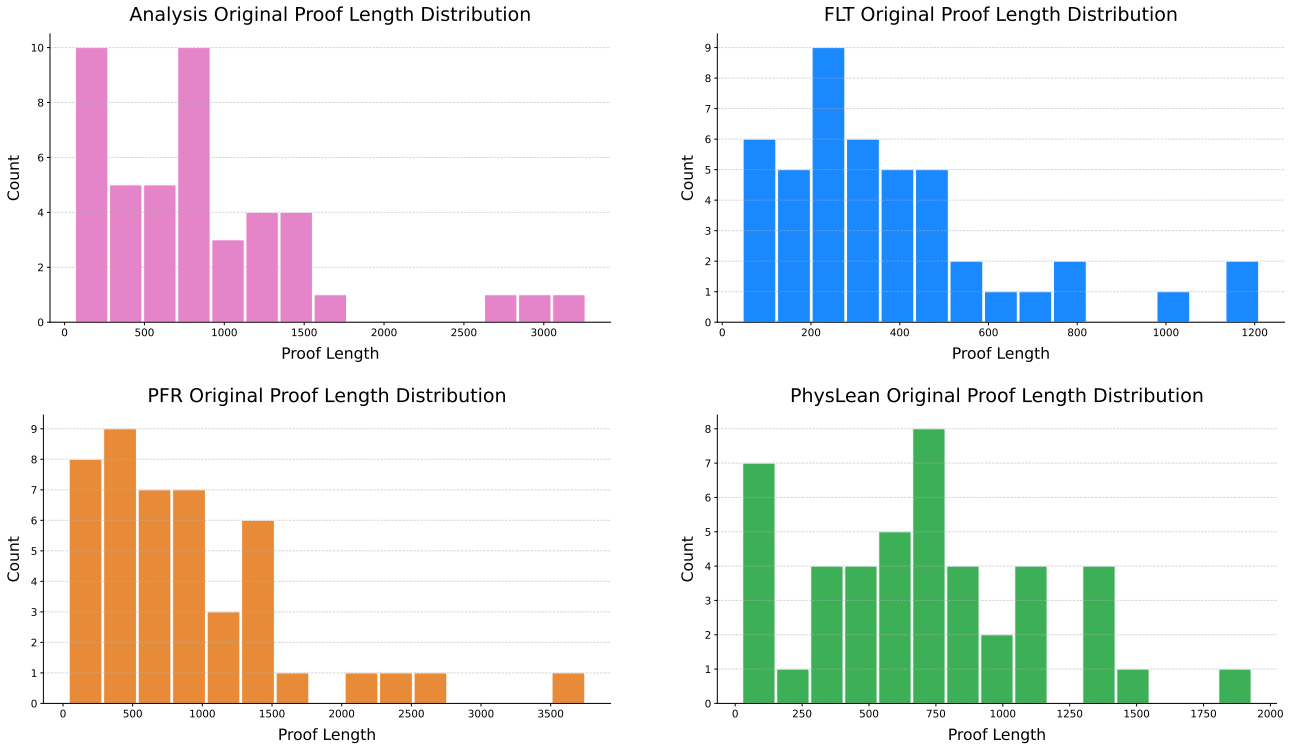


Figure 7. Distribution of original proof lengths across four research-level evaluation sets (Analysis, FLT, PFR, PhysLean).

I.2. Metrics

We report three metrics across our benchmarks.

Proof length. We measure proof brevity as the average relative percentage decrease in token count between the initial long proof and the refactored proof. To ensure consistency with prior work, we use the syntax-aware tokenizer released by ProofOptimizer (Gu et al., 2025); code is provided in Appendix J. This metric is reported on all seven benchmarks.

Compilation cost. We assess computational efficiency using `lake env lean --profile`, which exposes per-declaration timing. We isolate proof execution time from library import overhead, since import time is dominated by caching effects unrelated to proof content. Each measurement is taken in a consistent environment to ensure stable, deterministic timings, and we report the mean over five runs on identical hardware. On miniF2F and PutnamBench we follow the per-proof setting as each proof is a standalone `.lean` file, compiling each refactored proof in isolation. Research repositories do not admit clean per-proof wall-clock attribution, so we instead measure heartbeats, Lean 4's deterministic proxy for elaboration work; see Appendix L for full details. This metric is reported on all seven benchmarks.

Cross-version compatibility. To evaluate the impact of our version-tagged metadata, we compare the agent's performance under *version-filtered* strategy retrieval against an unfiltered baseline. Refactoring is run across three Lean and `Mathlib` toolchains (`v4.14.0`, `v4.16.0`, and `v4.22.0`); we then measure proof length reduction in each environment to

determine whether retrieval tailored to a target version yields larger gains than version-agnostic retrieval. We additionally assess zero-shot version transfer by recompiling `v4.19.0` PutnamBench outputs from Lean Refactor, ProofOptimizer, and the original verbose proofs against later `Mathlib` releases through `v4.29.0` without further refactoring, measuring how many proofs continue to type-check across future toolchains.

I.3. Baselines

ProofOptimizer. On miniF2F and PutnamBench, we benchmark against ProofOptimizer (Gu et al., 2025) using their officially released shortest proofs, obtained via an 8-iteration shortening process with sampling budget 64 in the first six iterations and 1024 in the final two (denoted $6\times64 + 2\times1024$). Because the ProofOptimizer model checkpoints are closed-source, we cannot run their model on the remaining benchmarks (Putnam2025, Analysis, FLT, PFR, PhysLean). On all benchmarks, we conduct a controlled ablation over variants of our own framework.

Internal ablations. We define four internal baselines, each ablating a different component of Lean Refactor:

- **Base Agent.** The framework stripped to its core execution loop. The Planner module is removed; only the Refactorer and the verifier-guided Debugger remain. Strategy retrieval is applied as in our full method (Section 3.3), and the retrieved strategies are passed directly to the Refactorer. *This isolates the contribution of the Planner.*
- **Base Agent w/o Retrieval.** The Base Agent, with strategy retrieval (Section 3.2) additionally disabled. *This isolates the contribution of retrieval within the simplified loop.*
- **Lean Refactor w/o Retrieval.** The complete agent architecture (Planner, Refactorer, and Debugger) with the retrieval module disabled. *This isolates the gains attributable purely to the Planner's task decomposition, in the absence of external strategy context.*
- **Lean Refactor w/ Random Retrieval.** The full agent framework, but with our semantic and metadata-aware retrieval replaced by uniform random sampling from the strategy bank. This serves as a *control for the embedding-and-rerank pipeline itself*, distinguishing the value of *retrieving the right* strategies from the value of *having any* strategies in context.

LLM-agnosticism and Claude Code comparison. We additionally run our framework with Claude Haiku 4.5 (Anthropic, 2025) to test that gains are not specific to Gemini 3 Flash. Using the same Claude Haiku 4.5 backbone, we compare against **Claude Code**. We prompt Claude Code with the same proof-shortening objective as our Lean Refactor agent, but adapted to its native abstractions (skills and MCP servers) rather than forcing it into our planner–refactorer–debugger loop. This lets Claude Code exercise its own agentic control flow on the task our framework is designed for, isolating the contribution of our architecture from the contribution of simply prompting a capable general-purpose agent to shorten Lean proofs. We evaluate Claude Code under three tool configurations of increasing support:

- **Lean MCP only.** The agent is provided only with the Lean MCP server[6], which exposes compiler feedback for correctness checking but no refactoring guidance.
- **Lean MCP + Lean skills.** The previous configuration is augmented with the Lean skills package[7], which supplies general Lean 4 refactoring and golfing guidance.
- **Lean MCP + Lean skills + our strategy retrieval tool.** The previous configuration is further augmented with a retrieval tool that exposes our strategy bank as an on-demand resource the agent can query during refactoring.

This comparison isolates the marginal value contributed by our strategy bank when added to a strong off-the-shelf coding agent. Due to compute budget, we run the Claude Code comparison on one representative benchmark per regime: PutnamBench (competition) and Analysis (research-level).

I.4. Implementation Details

Lean environment. All data collection and all experiments use Lean `v4.24.0` as the default environment, except where the cross-version compatibility study explicitly requires `v4.14.0`, `v4.16.0`, or `v4.22.0`.

[6] https://github.com/oOo0oOo/lean-lsp-mcp
[7] https://github.com/cameronfreer/lean4-skills

Model serving. All open-weight models used during data collection (in particular GPT-OSS and the Qwen3-Embedding-8B) are primarily served via vLLM with default sampling parameters. Additionally, we utilized the OpenRouter API for GPT-OSS. For all GPT-OSS deployments, we configured the model using the medium reasoning-strength setting. Total compute for the reported experiments was approximately 10,000 GPU-hours on NVIDIA 6000 Ada GPUs, with strategy-bank construction (proof synthesis, strategy distillation, deduplication) accounting for the majority of this cost. API-served models (GPT-OSS-120B via OpenRouter, Gemini 3 Flash, and Claude Haiku 4.5) incurred an additional cost of approximately $5,000 USD.

Agent configuration. For our agent framework, we use:

- a maximum of 3 debugging cycles per refactoring episode ($D = 3$),
- a total API call budget of 30 per theorem ($B = 30$),
- the minimum proof length during optimization is 5 ($T = 5$),
- proof chunking with chunk sizes of 5, 10, and 20 lines. The multi-granularity chunking ensures that retrieval queries target proof regions of different lengths and consequently surface strategies appropriate to refactoring at different granularities.
- experiments were run on servers equipped with Intel Xeon w9-3495X CPUs and NVIDIA 6000 Ada GPUs.

Backbone LLMs. Unless stated otherwise, all main-paper experiments use Gemini 3 Flash (The Gemini Team, 2025) as the agent backbone. Claude Haiku 4.5 (Anthropic, 2025) is used both for the LLM-agnosticism check and as the shared backbone in the Claude Code comparison.

J. Code for Getting Proof Length

```python
def proof_length(statement_and_proof):
    """
    Compute the token count of a proof from a full statement string.
    Extracts the proof by finding where the signature ends, then tokenizes and counts.
    """
    lean_operators = [
        ":=", "!=", "&&", "-.", "->", "←", "..", "...", "::", ":>",
        "<;>", ";;", "==", "||", "=>", "<=", ">=", "ᵢ", "?_"
    ]
    lean_operators_spaced = [" ".join(conn) for conn in lean_operators]
    lean_operators_dict = dict(zip(lean_operators_spaced, lean_operators, strict=False))

    def lexer(lean_snippet):
        tokenized_lines = []
        for line in lean_snippet.splitlines():
            tokens = []
            token = ""
            for ch in line:
                if ch == " ":
                    if token:
                        tokens.append(token)
                        token = ""
                elif str.isalnum(ch) or (ch in "._'"):
                    token += ch
                else:
                    if token:
                        tokens.append(token)
                        token = ""
                    tokens.append(ch)
            if token:
                tokens.append(token)
            tokenized_line = " ".join(tokens)
            for conn in lean_operators_spaced:
                if conn in tokenized_line:
```

```
                    tokenized_line = tokenized_line.replace(conn, lean_operators_dict[conn])
            tokenized_lines.append(tokenized_line)
        return "\n".join(tokenized_lines)

    try:
        statement_and_proof = remove_comments(statement_and_proof) # Remove all comments in the proof
        decl_start, decl_end = return_theorem_to_prove(statement_and_proof) # Gets the declaration start and end
        proof = statement_and_proof[decl_end:] # Isolate the proof from the declaration
        proof_tokenized = lexer(proof)
        return sum([len(l.split(" ")) for l in proof_tokenized.splitlines()])
    except:
        return 10**9
```

K. Comparison with ProofOptimizer's Heartbeat-Optimized Proofs

The wall-clock results in Table 3 compare against ProofOptimizer's *length-optimized* proofs because their heartbeat-optimized model weights are not released, putting their best-case compilation numbers out of reach of direct re-execution. We close this gap using Lean's `#count_heartbeats` (under `set_option Elab.async false`) as a deterministic proxy of compilation cost, the same metric ProofOptimizer adopts when optimizing for execution efficiency. This permits a direct comparison against the compilation cost numbers that ProofOptimizer (Gu et al., 2025) publish: the values produced by their heartbeat-optimized inference-time variant after eight iterations with sampling budget $6\times64 + 2\times1024$, i.e. their strongest reported heartbeat results. We compute the heartbeats for proofs refactored under the *Lean Refactor w/ Reranking* setting from Section 4.2, and report the comparison in Table 10.

Table 10. **Heartbeat-level comparison against ProofOptimizer's heartbeat-optimized variant.** Heartbeats reports the average heartbeat count (in thousands) across theorems. ProofOptimizer values are taken from their heartbeat-optimized proofs; "Original" refers to the unrefactored proofs from the same source. Lower is better; best per column in bold.

Method	miniF2F Heartbeats (K) ↓	PutnamBench Heartbeats (K) ↓
Original (no refactoring)	36.3	221
ProofOptimizer, Heartbeat-optimized (Gu et al., 2025)	10.4	111
Lean Refactor w/ Reranking (Ours)	**9.2**	**82**

Objective-conditioned reranking yields the lowest heartbeat counts. Across both benchmarks, Lean Refactor w/ Reranking attains the lowest average heartbeat count among all evaluated methods, improving over ProofOptimizer's heartbeat-optimized proofs from 10.4K to 9.2K on miniF2F and from 111K to 82K on PutnamBench. We emphasize that ProofOptimizer's underlying model is trained to generate shorter proofs; its heartbeat-optimized variant achieves the published numbers by reusing the same length-trained model and selecting low-heartbeat candidates at inference. This exposes a train–inference objective mismatch: the prior is shaped by token length, while the selection rule targets execution cost. Our framework avoids this mismatch by carrying the objective in the retrieval rule rather than in model weights, reranking a shared, semantically-retrieved candidate pool by per-strategy heartbeat metadata (Section 3.2). Because the metadata is annotated directly on strategies, the same bank can be redirected to new cost metrics without retraining.

L. Compilation Cost for Research-Level Proofs

For competition benchmarks, each proof lives in a self-contained `.lean` file, so we can isolate proof execution from import overhead and report wall-clock time directly. Research repositories invert this setup: proofs reside inside large multi-file Lean projects whose elaboration is interleaved with import resolution, neighboring declarations, and kernel cache state, and there is no stable way to attribute wall-clock time to a single proof in this regime[8]. We therefore switch to *heartbeats*, Lean 4's deterministic proxy for elaboration work. We prepend each proof with `set_option Elab.async false in` and `#count_heartbeats in` to pin the count to the declaration under measurement, consistent with prior work (Gu et al., 2025). We report dataset-level means of the per-proof absolute and relative reductions in Table 11.

[8] https://github.com/leanprover/lean4/issues/8038

Table 11. **Heartbeat reduction on Analysis (Tao, 2024), FLT (Buzzard & Taylor, 2025), pfr (Tao, 2025), and PhysLean (Tooby-Smith, 2025).** *Avg. Rel. Reduc.*: average per-proof relative % decrease in heartbeats. *Avg. Abs. Reduc.*: average per-proof absolute decrease in heartbeats. Higher is better for both. Each dataset contains $n = 45$ proofs.

Dataset	Method	Avg. Rel. Reduc. ↑ (%)	Avg. Abs. Reduc. ↑ (heartbeats)
Analysis (Tao, 2024)	Lean Refactor w/o Retrieval	−6.72	2324
	Lean Refactor w/ Random Retrieval	−4.18	2759
	Lean Refactor	−14.43	3049
	Lean Refactor w/ Reranking	**3.19**	**3809**
FLT (Buzzard & Taylor, 2025)	Lean Refactor w/o Retrieval	0.37	−946
	Lean Refactor w/ Random Retrieval	0.70	222
	Lean Refactor	3.00	311
	Lean Refactor w/ Reranking	**11.33**	**1235**
PFR (Tao, 2025)	Lean Refactor w/o Retrieval	−6.81	1332
	Lean Refactor w/ Random Retrieval	9.21	3223
	Lean Refactor	3.13	2184
	Lean Refactor w/ Reranking	**9.89**	**3309**
PhysLean (Tooby-Smith, 2025)	Lean Refactor w/o Retrieval	1.41	1152
	Lean Refactor w/ Random Retrieval	3.73	2166
	Lean Refactor	-21.97	2609
	Lean Refactor w/ Reranking	**7.47**	**4101**

Lean Refactor with reranking is the strongest method by both metrics on every dataset, indicating that objective-aligned retrieval, not structural rewriting alone, is what consistently delivers heartbeat savings. The two columns can disagree in sign (notably for Analysis, PFR, and PhysLean under non-reranking variants) because the relative mean is sensitive to outliers: a few refactored proofs exhibit large heartbeat increases that pull the average relative reduction negative, even when the absolute mean remains positive and the majority of proofs improve.

M. Examples of Refactoring Strategies

Strategy 1: Eliminate exhaustive numeric case analysis; construct new witness via arithmetic shift

Description. A wasteful proof pattern: extract concrete numeric values for an existential witness (and any auxiliary constants) by evaluating a universal hypothesis at many points and discharging the resulting system with `norm_num`, `nlinarith`, and `linarith`. This is unnecessary when the goal differs from the hypothesis only by a shift in the bound variable — the witness can be obtained by translating the existing one algebraically, leaving the numeric values abstract.

When to apply. The hypothesis has the form `∃ x, ∀ y, P x y` and the goal has the form `∃ x, ∀ y, P x (y - k)` (or any other affine reparametrisation of `y`). **Note:** The remaining goal after providing the shifted witness must close by specializing the hypothesis at the shifted point — i.e. it must not depend on the specific numeric value of the witness. If it does, the original numeric derivation is needed.

Application guide.

1. Delete every block that pins the witness or auxiliary constants to specific numeric values — the long chains of `have h_i := hyp v_i` followed by `norm_num`, `nlinarith`, `linarith`.
2. Destructure the existential: `rcases h with ⟨a, ha⟩`.
3. Provide the shifted witness: `refine ⟨a + k, ?_⟩`, where k is the shift between the hypothesis's bound variable and the goal's (read off the goal: if it mentions `y - k`, use `a + k`).
4. Close the remaining goal by specializing `ha` at the shifted point, typically `intro y; simpa using ha (y - k)`, or a short `linarith`/`ring` after substitution.

Example. Schematic shape: from a hypothesis `h : ∃ x, ∀ y, P x y` prove the shifted goal `∃ x, ∀ y, P x (y`

– `k)`. The before-proof wastefully derives the concrete value of the witness a (and a secondary constant c) by instantiating `h` at many integers.

Before

```
-- pin the witness to a concrete value
have h_a : a = a₀ := by
  have e₁ := h v₁
  have e₂ := h v₂
  -- ... many more instantiations ...
  have e_n := h v_n
  norm_num at e₁ e₂ ... e_n
  nlinarith
-- pin an auxiliary constant
have h_c : c = c₀ := by
  have e₁ := h v₁'
  -- ... many more instantiations ...
  have e_m := h v_m'
  norm_num [h_a] at e₁ ... e_m ⊢
  <;> linarith
use a₀ + k
-- ... finish using h_a, h_c ...
```

After

```
rcases h with ⟨a, ha⟩
refine ⟨a + k, ?_⟩
intro y
simpa using ha (y - k)
```

Reduction: large.

Strategy 2: Replace per-entry `have` lemmas with a single `ext` and `fin_cases` proof

Description. The long proof proves each matrix entry separately using `have` statements and then glues them together with `Matrix.ext` inside a final `h_main`. For matrices indexed by a finite type, equality can be reduced to pointwise equality by `ext`, after which a case split on the indices (`fin_cases`) makes each entry goal trivial. A single `simp` (with the right lemmas) then closes all cases, removing the need for separate per-entry lemmas.

When to apply. Use this pattern whenever you are proving equality of two matrices (or similar objects) indexed by `Fin n` or another finite type, and you have written individual lemmas for each entry. **Note:** Requires that the target matrix is built via `Matrix.of` / `!![...]` notation (so `Matrix.of_apply` reduces it) and that the per-entry goals reduce to something a unified finisher can solve. If the original per-entry proofs required nontrivial work, that work must still be expressible as part of the unified finisher.

Application guide.

1. Delete all `have` statements that compute each entry individually, including the final `h_main` that wraps `apply Matrix.ext`.
2. After `intro X`, write `ext i j`.
3. Immediately follow with `fin_cases i <;> fin_cases j` to enumerate all index pairs.
4. Finish with

   ```
   simp [Matrix.mul_apply, Matrix.of_apply, Fin.sum_univ_two]
   ```

   optionally followed by `<;> ring` or `<;> norm_num` if arithmetic remains.
5. Remove the final `exact h_main` as the goal is already solved.

Example. *Before*

```
intro X
have h₁ : ((M : Matrix (Fin 2) (Fin 2) ℝ) * X) 0 0 = a := by
  simp [Matrix.mul_apply]
  -- many try tactics
have h₂ : ((M : Matrix (Fin 2) (Fin 2) ℝ) * X) 0 1 = b := by
  simp [Matrix.mul_apply]
  -- many try tactics
have h₃ : ((M : Matrix (Fin 2) (Fin 2) ℝ) * X) 1 0 = c := by
  simp [Matrix.mul_apply]
  -- many try tactics
have h₄ : ((M : Matrix (Fin 2) (Fin 2) ℝ) * X) 1 1 = d := by
  simp [Matrix.mul_apply]
  -- many try tactics
have h_main : (M : Matrix (Fin 2) (Fin 2) ℝ) * X = M2 := by
  apply Matrix.ext
  intro i j
  fin_cases i <;> fin_cases j
  simp [h₁, h₂, h₃, h₄]
exact h_main
```

After

```
intro X
ext i j
fin_cases i <;> fin_cases j <;>
  simp [Matrix.mul_apply, Matrix.of_apply, Fin.sum_univ_two]
```

Reduction: medium.

N. Prompts

N.1. Planner Prompt

```
Your task is to analyze a given Lean 4 proof and create a structured plan for
optimizing and shortening it, while maintaining the correctness.

## Instructions

Analyze the proof and identify regions (or entire proof) that can be optimized and
simplified. For each optimization opportunity:

1. **Identify the line range**: ...
2. **Provide a title**: ...
3. **Provide potential reduction**: ...
4. **Determine an optimization strategy**: ...

## Inputs

1. A correct Lean 4 statement and proof source code.
2. Elaborated signature.
3. Doc string from the source if it exists.
4. All the dependencies used in the statement and proof...
5. A list of retrieved optimization strategies from a strategy index...

## Retrieved Optimization Strategies
...

## Output Format

Output the plans using the following JSON format, wrapped in a single ```json ``` tags.
 Ensure the output is valid JSON.

[
  {
    line_start: X,
    line_end: Y,
    title: the strategy name,
    reduction: high, medium, or low,
    description: Detailed description of the optimization strategy ...
  }
]

Rules about the ordering of the output plans:
1) Sort plans primarily from top to bottom of the proof (increasing line numbers).
2) Overlaps of optimization regions are allowed. If two plans overlap in location, put
 the plan with the MOST significant potential reduction first.
3) If two plans have the same overlap and same potential impact, keep top-to-bottom
ordering.

Examples:
...

Here is the proof to optimize:
...

Theorem's elaborated signature:
...

Theorem's documentation:
...

Information about dependencies used in the proof:
```

```
...

Now analyze the proof and generate your optimization plans.
```

N.2. Refactor Prompt

```
You are an expert Lean 4 proof optimizer. Your task is to execute a specific
optimization plan on a proof, modifying the targeted section while ensuring the proof
remains correct.

## Current Proof
...

## Theorem's Elaborated Signature
...

## Documentation
...

## Dependencies
...

## Optimization Plan

**Target Lines:**: ...
**Title:**: ...
**Description:**: ...

## Instructions

1. **Focus on the targeted section**: ...
2. **Preserve correctness**: ...
3. **Do NOT modify the theorem statement**: ...
4. **Aim for shorter proof**: ...
5. **Output the complete theorem and proof**: ...

Before providing the code, briefly explain what changes you're making and why.

Then output the complete optimized Lean 4 code wrapped in tags:

```lean4
<your optimized complete theorem and proof here>
```
```

N.3. Debugger Prompt

```
The proof (Round {{ prev_round_num }}) is not correct. Following is the compilation
error message, where we use <error></error> to signal the position of the error.

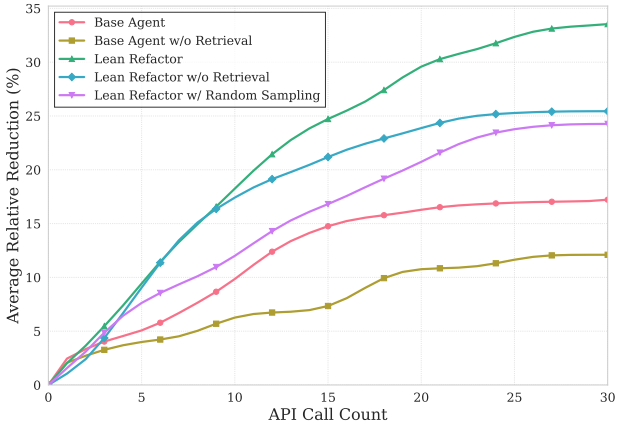
Error message:
...

Before producing the Lean 4 code that fixes the error, provide a detailed analysis of
the error message. Focus on fixing the compilation errors, and try your best to
preserve the proof optimization that was previously done. Do not revert to the
original unoptimized proof.

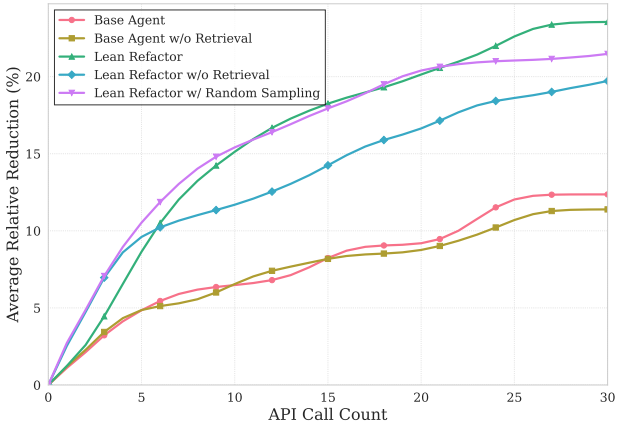
You must wrap the entire Lean 4 theorem and proof in tags like:
```

```
```lean4
<your corrected proof here>
```
```

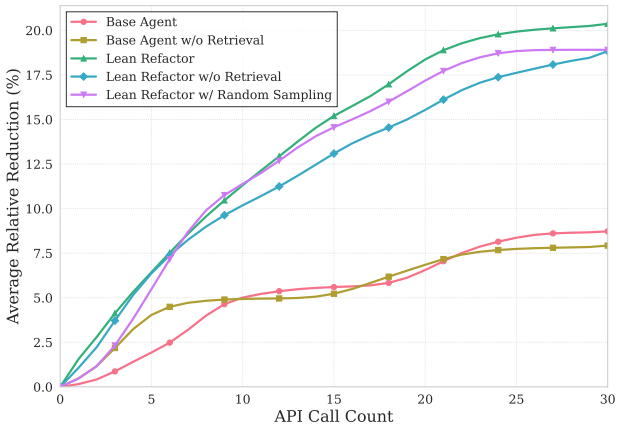
O. Test Time Scaling Curves



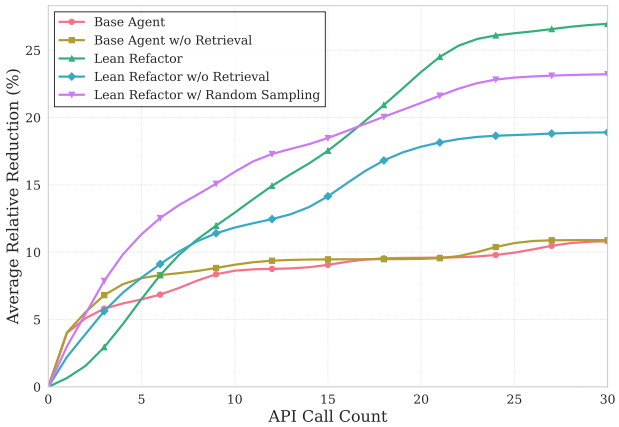
(a) Improvements on Analysis Dataset



(b) Improvements on FLT Dataset

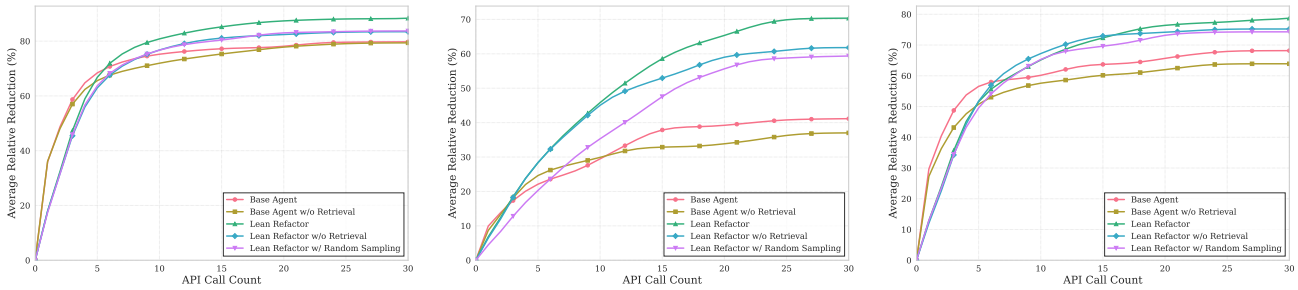


(c) Improvements on PFR Datasets



(d) PhysLean Improvements

Figure 8. **Average relative length reduction across research-level datasets over successive API calls.** We compare the base agent against our proposed Lean Refactor agent, alongside ablation variants (without retrieval and with random sampling). The Lean Refactor agent converges to a higher relative reduction compared to all other variants.



(a) miniF2F Improvements *(b)* PutnamBench Improvements *(c)* Putnam 2025 Improvements

Figure 9. **Average relative length reduction across competition-style datasets over successive API calls.** We compare the base agent against our proposed Lean Refactor agent, alongside ablation variants (without retrieval and with random sampling). The Lean Refactor agent converges to a higher relative reduction compared to all other variants.

Algorithm 1 Retrieval-Augmented Agent Framework for Proof Refactoring

Require: Initial unoptimized proof P, Strategy DB $\mathcal{S}$, Budget B, Target length threshold T, Max debug rounds D, Retrieval budget K
Ensure: Optimized refactored proof P^*

```
P_curr ← P
History ← ∅
steps ← 0
while steps < B and LENGTH(P_curr) > T do
    Chunks ← SEGMENTPROOF(P_curr)
    C_retrieved ← ∅
    for c ∈ Chunks do
        E_c ← EMBED(c)
        S_c ← RETRIEVEANDRERANK(S, E_c, K)                    ▷ Multi-objective alignment
        C_retrieved ← C_retrieved ∪ {(S_c, LOCATION(c))}
    end for
    Plan ← PLANNER(P_curr, C_retrieved, History)
    if Plan = ∅ then
        break                                                 ▷ Terminate if no refactoring steps are viable
    end if
    step_applied ← False
    for step ∈ Plan do
        P_cand ← REFACTORER(P_curr, step)
        success, ϵ ← COMPILELEAN(P_cand)
        d ← 0
                                                              ▷ Verifier-guided self-debugging loop
        while not success and d < D do
            P_cand ← DEBUGGER(P_cand, ϵ, step)
            success, ϵ ← COMPILELEAN(P_cand)
            d ← d + 1
        end while
                                                              ▷ Check for successful compilation and length reduction
        if success and LENGTH(P_cand) < LENGTH(P_curr) then
            P_curr ← P_cand
            History ← History ∪ {(step, Success)}
            step_applied ← True
            break                                             ▷ Break inner loop to dynamically replan with updated proof
        end if
    end for
    if not step_applied then
        History ← History ∪ {(Plan, Failed)}
    end if
    steps ← steps + 1
end while
return P_curr
```