# LEANLEAN: BENCHMARKING REPOSITORY-SCALE LEAN PROOF COMPRESSION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

As large language models (LLMs) formalize increasingly advanced mathematics, their proofs can span millions of lines of code. Producing more concise formalizations requires models to discover simpler arguments and extract reusable lemmas across large codebases. To measure this important capability, we introduce LEANLEAN, a benchmark consisting of 64 large-scale, real-world Lean repositories containing up to 174 thousand lines of code. Within a 12-hour time limit, models are tasked with reducing repository size while preserving the mathematical validity of their main theorems. To achieve maximal compression, models need to optimize at three levels of abstraction: (1) simplifying the underlying mathematical arguments, (2) restructuring proof dependencies by deleting and adding new declarations, and (3) applying syntactic optimizations. The best model, Opus 5, achieves a very high average compression score of $48.3\%$, while smaller models such as Gemini 3.8 Flash and GPT-5.6 Luna reach less than $25\%$. We find that most compression comes from restructuring proof dependencies by making previously used declarations unnecessary, while Opus 5 is the only model to achieve substantial gains through syntactic optimizations. LEANLEAN provides a strong testbed for evaluating models across a combination of capabilities, including long-context handling, mathematical theorem proving, and coding proficiency in Lean.

## 1 INTRODUCTION

Large language models (LLMs) have demonstrated impressive capabilities, for instance by formalizing a proof of Fermat's Last Theorem (FLT) and finding a counterexample to the Navier-Stokes existence and smoothness problem (Anthropic, 2026a; OpenAI, 2026c). These rapid improvements call for extremely challenging benchmarks that can differentiate the capabilities of frontier LLMs. Formal mathematics offers an ideal domain for such benchmarks: problems can be made almost arbitrarily difficult by focusing on recent research, and correctness can be verified automatically.

**Compression of formalized mathematics.** AI agents are already generating formalized mathematics at a large scale, with the formalization of the FLT alone comprising over 13 million lines of code (Anthropic, 2026a), over five times larger than the entire Mathlib library (The mathlib Community, 2020). This growing scale motivates repository-wide proof compression as an evaluation task: can models prove the same main results using less Lean code? Beyond saving space, compression has been identified as one of the key ideas behind human mathematics (Aksenov et al., 2026), and is important for the reusability of formalized mathematics. However, existing benchmarks only evaluate theorem proving (Jiang et al., 2025; Poiroux et al., 2025) or the optimization of individual proofs (Lu et al., 2026; Gu et al., 2025; Fu et al., 2026), leaving repository-scale compression unexplored.

**Compression as a capability.** Compressing formalized mathematics at repository scale requires a combination of important capabilities. Specifically, compression can be achieved by changing proofs on three levels: (1) simplifying mathematical arguments, (2) reorganizing proof dependencies and extracting useful intermediate results, and (3) introducing more concise syntactic representations. These changes require both high-level mathematical reasoning and low-level code manipulation, as well as a good understanding of the Lean language. Due to the large size of formalization repositories, compression also requires long-term planning and long-context reasoning. However, current benchmarks focus on individual proofs and therefore miss many of these capabilities, including reasoning about dependencies across large codebases and identifying reusable lemmas.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

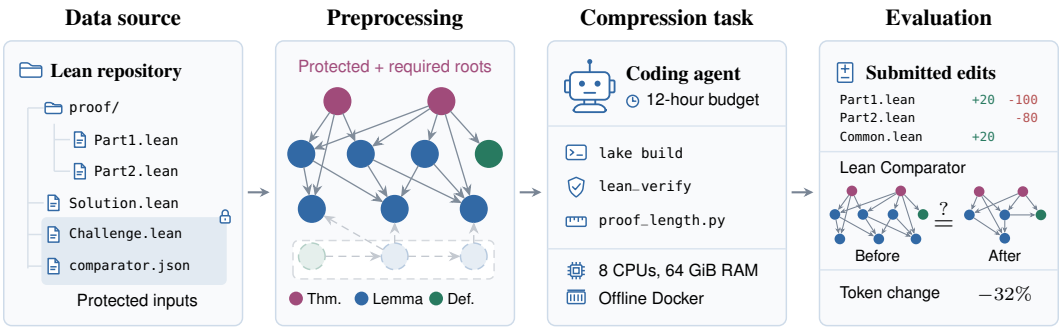



Figure 1: We construct LEANLEAN by collecting 64 Lean repositories from the Palomar Registry and removing declarations outside the dependency closure of the protected results and other required roots. An agent then has 12 hours to reduce the remaining Lean token count, with access to build, verification, and measurement tools. We rebuild the submitted repository and use the Lean Comparator to check that the protected results still hold under the permitted axioms.

**This work: LEANLEAN.** We introduce LEANLEAN, a benchmark for repository-scale Lean compression. As shown in Figure 1, LEANLEAN contains 64 real-world repositories curated from the Palomar Registry (Palomar, 2026), a recently introduced centralized registry of Lean codebases. We clean each repository by removing most dead code that the protected results do not rely on. Agents are then tasked with compressing the Lean codebase while preserving the protected results, i.e., the mathematical validity of the formalization. Compression is measured as the relative reduction of *Lean tokens* (Lu et al., 2026), which are syntax tokens excluding comments, imports, and whitespace. After compression, Lean Comparator (leanprover, 2026) is used to check that the protected results still hold under the permitted axioms. Submissions that fail this check receive a score of zero.

**Main results.** Figure 2 summarizes the compression scores across the evaluated models. Opus 5 achieves the best compression rate (48.3%), followed by GPT-5.6 Sol (37.8%). Compression rates drop significantly for the sixteen largest repositories, with Opus 5 only achieving 21% compression. By categorizing the compression strategies used, we also find that the largest gains come from restructuring proof dependencies so that previously used declarations become unnecessary. Only Opus 5 achieves a significant amount of additional compression through syntactic optimizations.

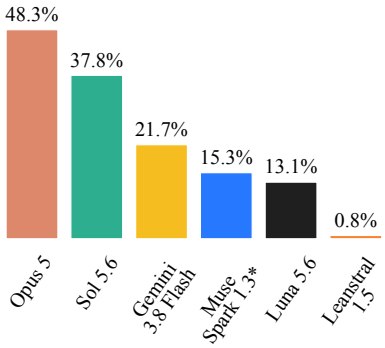



Figure 2: Performance on LEANLEAN.

**Key contributions.** Our key contributions are:

- We identify repository-scale Lean compression as an important task for LLM evaluation.
- We create LEANLEAN, a Lean compression benchmark with 64 real-world repositories (§3).
- We evaluate six AI agents and analyze their compression strategies in depth (§4).

## 2 RELATED WORK

**Formal theorem proving.** LLM-based theorem proving in formal proof assistants such as Lean Moura & Ullrich (2021) has attracted significant research interest. For instance, recent work has trained specialized models to generate Lean proofs (Lin et al., 2025; Ren et al., 2025; Leang et al., 2026) and to accurately translate natural language statements into formal Lean statements (Wu et al., 2026; Gao et al., 2025). Alongside these advances, systems such as Hilbert (Varambally et al., 2025) and Seed-Prover (Chen et al., 2025) combine proof generation with tool use and verifier feedback in agentic systems. More recently, these capabilities have been applied to large-scale research formalization through collaborative platforms such as Prove2Me (Chen et al., 2026) and agentic systems such as LeanMarathon (Zhang et al., 2026).

**Benchmarks in formal theorem proving.** Established benchmarks, including miniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), and PutnamBench (Tsoukalas et al., 2024), initially focused on competition mathematics and undergraduate theorem proving. More recent benchmarks evaluate context-dependent theorem proving in real-world research-level Lean projects (Hu et al., 2025; Poiroux et al., 2025; Letson et al., 2026; Xin et al., 2026). Beyond theorem proving, APE-Bench (Xin et al., 2025) evaluates tasks extracted from Mathlib maintenance commits, for instance, including refactoring according to specific instructions. However, these benchmarks focus on individual theorems or engineering tasks within a single file. LEANLEAN instead evaluates a long-running compression task for an entire repository, allowing for more complex reasoning about dependencies and redundancies across the codebase.

**Compression and readability in Lean.** Formal theorem proving requires attention to reusability and readability beyond successful verification (Baanen et al., 2025), and several works explore the use of LLMs to address these concerns. ImProver 2 (Ahuja et al., 2026) and ProofOptimizer (Gu et al., 2025) train models for structural proof optimization and proof shortening, respectively. Similarly, Lean Refactor (Lu et al., 2026) targets proof length, compilation cost, and version compatibility, while Proof-Refactor (Fu et al., 2026) seeks to extract helper declarations from proofs to improve readability. LEANLEAN builds on these motivations by providing a benchmark with a measurable compression objective over large repositories. In contrast to these works, LEANLEAN optimizes entire repositories rather than individual proofs.

**Proof quality in mathematics.** Several works have explored the importance of proof quality beyond correctness in mathematical research (Tao, 2026; Weatherall & Wolfson, 2026; Avigad, 2026). For instance, Klowden & Tao (2026) argue that a proof's value also lies in the conceptual understanding it provides, its motivation, and its broader mathematical context. Even before the rise of LLMs, mathematicians emphasized the importance of developing and communicating understanding (Thurston, 1994), and argued about the difficulty of maintaining large-scale formalized mathematics repositories (Bourke et al., 2012; Vyskocil et al., 2010). These perspectives motivate evaluating mathematical artifacts beyond correctness.

**Code refactoring and optimization.** Creating benchmarks for alternative metrics has been explored in the context of code repositories. CodeTaste (Thillen et al., 2026), RefactorBench (Gautam et al., 2025), and SWE-Refactor (Xu et al., 2026) evaluate multi-file code refactoring in various programming languages. SWE-fficiency (Ma et al., 2025) and GSO (Shetty et al., 2025) require LLMs to optimize the performance of codebases, similar to LEANLEAN's focus on optimization of compression. In contrast to these works, LEANLEAN focuses on formalized mathematics in Lean, where models need strong mathematical reasoning capabilities in addition to code manipulation skills.

## 3 BENCHMARKING LEAN COMPRESSION

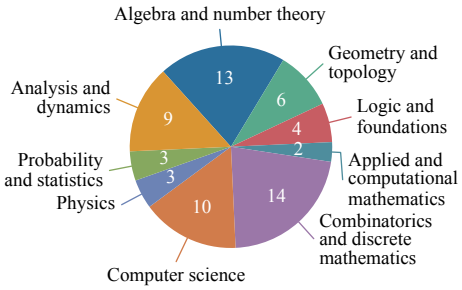
In this section, we describe the repository collection (§3.1) and preprocessing steps (§3.2) used to construct the LEANLEAN benchmark (§3.3).

### 3.1 REPOSITORY COLLECTION

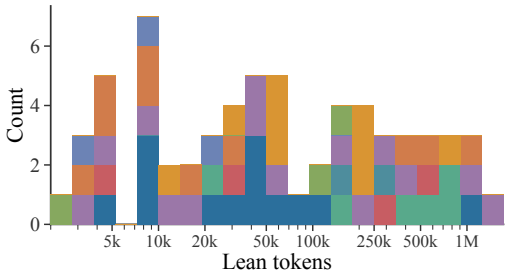
**Data source.** We collect repositories from the Palomar Registry (Palomar, 2026), which brings together Lean proofs in a standardized format. The registry contains[1] 200 GitHub repositories covering areas such as number theory, mathematical physics, and formal languages. Each repository must contain a `Challenge.lean` file stating the theorems to be proved and a `Solution.lean` file containing their proofs. This separation is used by Comparator (leanprover, 2026) to check that the submitted proofs establish the required statements. Palomar also uses LLM-based verification to check that the theorems stated in `Challenge.lean` match their natural language descriptions.

**Data selection.** We manually curated 64 benchmark instances from the eligible Palomar repositories to cover diverse mathematical subjects without overrepresenting any one area, while keeping the number of instances manageable for evaluation. We also sought variation in repository size, the number of protected declarations, and dependency-graph depth. Figure 3(a) shows the distribution of primary subject areas across the selected instances.

[1] As of September 23, 2026.



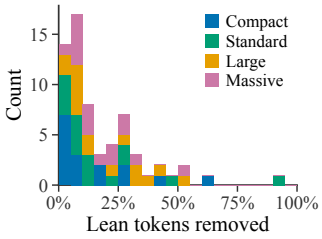

(a) Repositories by subject family.




(b) Repository sizes.

| Scale | Size | $n$ | Protected Thms. Med. | Protected Thms. Max. | LOC Med. | LOC Max. | Lean tokens Med. | Lean tokens Max. |
|---|---|---|---|---|---|---|---|---|
| Compact | $\leq$ 10k | 16 | 3.5 | 54 | 0.5k | 1.0k | 5.0k | 9.9k |
| Standard | 10k–50k | 16 | 4.5 | 84 | 2.2k | 4.9k | 27.9k | 49.5k |
| Large | 50k–250k | 16 | 2 | 24 | 9.8k | 25.4k | 123k | 218k |
| Massive | $>$ 250k | 16 | 3 | 109 | 59.4k | 174k | 619k | 1696k |
| All | — | 64 | 3 | 109 | 4.4k | 174k | 51.9k | 1696k |

(c) Repository statistics after preprocessing.




(d) Preprocessing removal.

Figure 3: Composition and preprocessing of LEANLEAN. The 64 repositories cover diverse mathematical topics (a) and are balanced across repository sizes (b). The four size groups span a wide range of protected theorem counts, lines of code (LOC), and Lean tokens after preprocessing (c). Preprocessing removes a median of 10.7% unused Lean tokens across all four size groups (d).

### 3.2 PREPROCESSING

Several Palomar repositories contain a substantial amount of code that is not needed to prove the main results in `Challenge.lean`. To evaluate proof compression beyond trivial dead code removal, we clean and standardize the repositories using a five-stage preprocessing procedure. Figure 3(b) shows the resulting repository sizes. We measure sizes in Lean tokens using tokenization that excludes comments, imports, and whitespace (Lu et al., 2026). The five stages are as follows:

**Cleaning.** We remove files that are not reachable from `Solution.lean`, such as Markdown files or Python scripts. We then build the repository to check that it remains valid.

**Dependency mapping.** From the build artifacts, we construct a directed graph of declarations. Each node represents a theorem, lemma, definition, macro, or other declaration. If declaration $X$ depends on declaration $Y$, we add an edge from $X$ to $Y$. Elaboration and code generation preserve different information from the source code, so no single source captures all dependencies reliably. We therefore construct the graph from three sources: `.olean` artifacts, `.ilean` artifacts, and the results of a string search algorithm. Further details are given in Appendix A.

**Grind dependencies.** Lean's `grind` tactic can automatically apply many lemmas to prove a goal. For instance, calls to `grind +suggestions` and `grind +locals` may select rules that are not explicitly named in the source. We temporarily replace these calls with their `grind?` equivalents to obtain Lean's suggestions for explicit calls of the form `grind [lemma1, lemma2, ...]`. We insert the suggested calls, update the dependency graph, and rebuild. If a call cannot be normalized and replayed safely, we restore the original file and conservatively retain all its declarations. After dependency analysis, we restore the original `grind` syntax.

**Graph roots.** We designate a subset of declarations as graph roots that must be retained. These include the main results defined in `Challenge.lean`, declarations with attributes such as `@[simp]` or `@[grind]`, instances, macros, syntax, notation, and elaborators. We retain these additional declarations because dependencies involving them are difficult to infer reliably. This procedure makes the process sound but incomplete: some dead code may remain.

Table 1: Frontier agents achieve average compression rates of below 50% at considerable cost.

| Model | Score (%) | Cost ($) | $t$ (h) |
|---|---|---|---|
| Opus 5 (high) | **48.30** | 63.52 | 8.79 |
| GPT-5.6 Sol (xhigh) | 37.77 | 135.51 | 4.45 |
| Gemini 3.8 Flash (high) | 21.73 | 25.23 | 3.93 |
| Muse Spark 1.3[2] (max) | 15.27 | 9.32 | 4.87 |
| GPT-5.6 Luna (xhigh) | 13.08 | **2.32** | 1.70 |
| Leanstral 1.5 (high) | 0.77 | 7.35 | **1.15** |

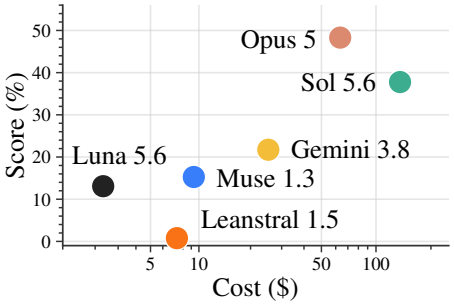



Figure 4: Compression increases roughly logarithmically with cost.

**Stripping.** Starting from the root set, we compute the transitive closure of the graph. We remove declarations outside this closure while preserving required imports. We then rebuild the stripped repository and run Lean Comparator to verify that the protected results and their permitted axioms remain intact. Each resulting repository serves as a benchmark instance. Preprocessing removes a median of 10.7% of each repository's Lean tokens, and more than half in some cases (Figure 3(d)).

### 3.3 The LeanLean Benchmark

The 64 processed repositories, summarized in Figure 3(c), form LeanLean. Models must compress each repository while preserving the correctness of its protected results. Good performance requires both high-level mathematical reasoning through proof simplification and refactoring, and low-level code manipulation through syntactic optimizations. Models must also thoroughly understand Lean's syntax and semantics, and reason across large codebases with long-term dependencies.

**Execution environment.** Each model runs in an isolated container without internet access and is given a 12-hour time limit, a CPU quota equivalent to 8 cores, and 64 GiB of RAM. Agents can read and edit files, compile Lean code with the repository-specific Lean toolchain, and run Python scripts. We also provide scripts to verify that the protected results are proven exactly and to measure repository size in Lean tokens. The prompt informs the model of the time limits (see Appendix G).

**Model execution.** To reflect real-world usage, we run each model in its recommended execution harness, such as Claude Code for Opus 5 and Codex for GPT-5.6 Sol. To improve reproducibility, we remove all skills, MCP access, and subagent tools. We record each model's actions and token usage to analyze its behavior and performance. All hyperparameters are given in Appendix B.

**Model evaluation.** After compression, we revert any changes to `Challenge.lean` and `comparator.json` before verifying that the protected results are still proven exactly using Comparator. Our main metric is Lean token compression, as defined by Lu et al. (2026). We count the total number of Lean tokens in the repository's `.lean` files, excluding `.lake/` build artifacts, `lakefile.lean`, `Challenge.lean`, comments, and imports. For each repository $r$ and model $M$, we compute the relative reduction in Lean tokens as $S_{M,r} = 1 - T_{\text{after}}/T_{\text{before}}$. If the build or Comparator check fails, we set $S_{M,r} = 0$. The model's overall score is the average across all 64 instances.

## 4 Experiments

We now perform an analysis of model performance on LeanLean. We evaluate OpenAI's GPT-5.6 Sol (OpenAI, 2026a), Anthropic's Opus 5 (Anthropic, 2026c), Google's Gemini 3.8 Flash (Google, 2026), Mistral's Leanstral 1.5 (Mistral, 2026), OpenAI's GPT-5.6 Luna (OpenAI, 2026a), and Meta's Muse Spark 1.3 (Meta, 2026)[2] on the full benchmark, and Anthropic's Fable 5.1 (Anthropic, 2026b) and OpenAI's GPT-6 Astra (OpenAI, 2026b) on a subset (§4.1). Reasoning-effort settings for the full-benchmark models are detailed in Table 6. We then analyze compression strategies quantitatively and qualitatively (§4.2). Further experiments are outlined in Appendix D.

[2]Due to a configuration issue, Muse Spark 1.3 ran with lower output token and context-compaction threshold than intended. Its results should therefore be interpreted as a lower bound on its performance. A partial rerun with the corrected configuration is reported in Appendix F.

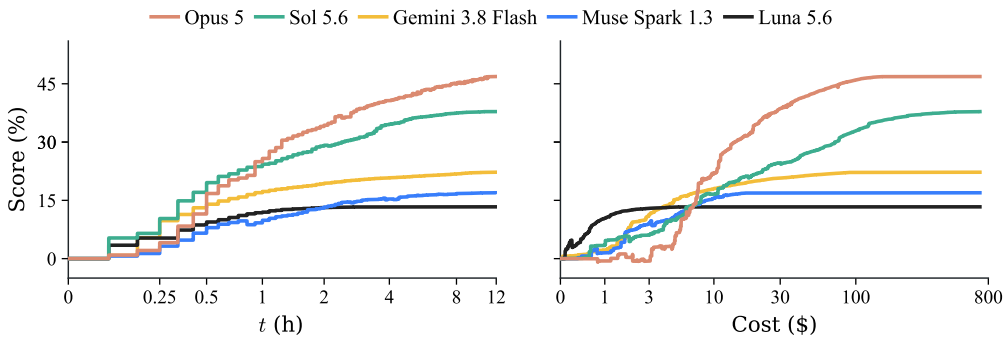



Figure 5: Compression increase is almost saturated after at most 2 h and $30 but continues to rise for GPT-5.6 Sol and Opus 5.

### 4.1 Model Performance

**Main results.** Table 1 and Figure 4 show the results for models evaluated on the full benchmark. Opus 5 achieves the highest average compression score (Score) at 48.3%, outperforming its closest competitor, GPT-5.6 Sol, by more than 10 percentage points. Importantly, even a single repository can distinguish between models: rankings computed on *individual* repositories achieve a mean Spearman correlation of 0.89 with the overall ranking. Due to Leanstral 1.5's poor performance, we exclude it from further analysis.

**Cost and time analysis.** Table 1 reports the average cost ($) and duration ($t$, in hours) per repository. Despite taking roughly twice as long, Opus 5 costs less than half as much as GPT-5.6 Sol. This shows that time and cost are two distinct aspects of efficiency. Interestingly, as shown in Appendix D, Opus 5's durations cluster near the 12-hour limit, while other models do not consistently use all time available. By far the cheapest model is GPT-5.6 Luna, which achieves a reasonable compression score for just $2 per repository.

**Fable 5.1 and GPT-6 Astra.** We evaluate Fable 5.1 and GPT-6 Astra on a subset of 12 repositories, three from each size category (listed in Table 11), and compare their results with those of Opus 5 and GPT-5.6 Sol on the same subset (Table 2). GPT-6 Astra achieves similar compression to GPT-5.6 Sol at lower cost, while Fable 5.1 both improves on Opus 5's score and reduces its cost.

Table 2: GPT-6 Astra (medium) and Fable 5.1 (high) dominate the respective benchmark leaders on a subset of 12 repositories.

| Model | Score (%) | $ | $t$ (h) |
|---|---|---|---|
| Fable 5.1 | 53.17 | 39.08 | 4.49 |
| Opus 5 | 46.10 | 60.15 | 8.14 |
| GPT-6 Astra | 37.41 | 38.53 | 1.53 |
| GPT-5.6 Sol | 35.02 | 68.75 | 3.31 |

**Compression by repository size.** Figure 6 compares each model's compression scores across the four size categories introduced in §3. Compression scores are significantly lower for larger repositories, pointing to long-context reasoning as a shared difficulty. Further, the ranking between models is similar across repository sizes, with Opus 5 leading in all categories.

**Progress through time.** We investigate how models progress through time and what a cost budget would imply for their performance. To do so, we take a snapshot of the repository after every tool call and record the elapsed time and accumulated cost. We then compute the compression for one snapshot in every 5-minute window, discarding those that do not produce a valid build. Importantly, this procedure only returns an approximate leaderboard, since the model was not given the restricted budget during execution.

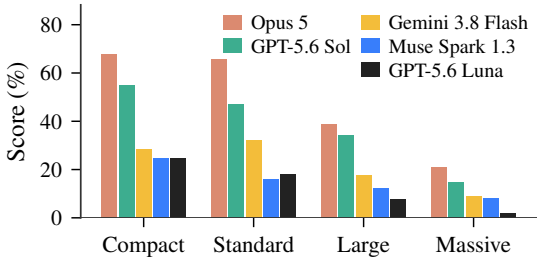
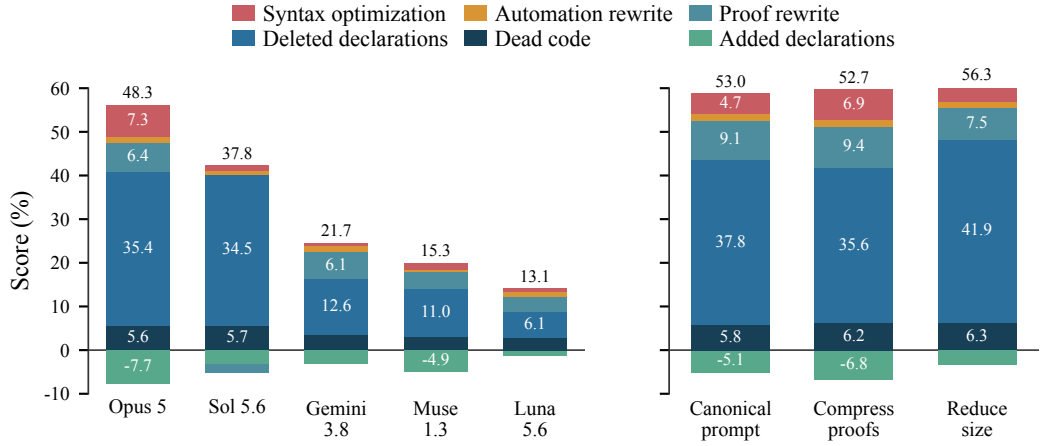



Figure 6: Compression decreases with repository scale.

Table 3: Categories used to analyze the origin of Lean token compression.

| Category | Description |
|---|---|
| Dead code | Removal of declarations that are unreachable from preserved results. |
| Deleted declarations | Deleted declarations on which a preserved declaration depended. |
| Added declarations | New declarations, excluding abbreviations counted as syntax optimization. |
| Proof rewrite | Simplification of existing proofs or changes to their dependencies. |
| Automation rewrite | Replacement of explicit proof steps with calls to automation tactics. |
| Syntax optimization | More concise code using macros, notation, definitions, or other shorthand. |




(a) Compression origin across models.

(b) Origin across prompt variants.

Figure 7: Most compression gains originate from deleting unused declarations (a), often enabled by extensive proof rewrites. Opus 5's strategies do not change depending on the instruction (b).

Figure 5 shows the results of this experiment. For each point on the graph, we use the last snapshot before the budget was exceeded. Below a $3 budget, Opus 5 makes no meaningful progress, likely because it first inspects the repository and develops a plan before making any edits. GPT-5.6 Luna leads at low dollar budgets, while Sol leads at short time budgets. Opus 5 takes the lead around one hour or $10, revealing a trade-off between early gains and the best eventual compression.

## 4.2 Compression Strategies

### 4.2.1 Declaration Removal Is the Dominant Compression Strategy

We analyze how models compress each repository by classifying their edits into six categories: (1) dead code, (2) deleted declarations, (3) added declarations, (4) proof rewrite, (5) automation rewrite, and (6) syntax optimization. Table 3 briefly describes each category, and Appendix C explains how we measure their contribution.

Figure 7(a) shows how the categories contribute to compression. The removal of declarations is by far the largest contributor. Only a small proportion are dead code missed by preprocessing, meaning that most gains come from restructuring the proof to make previously used declarations unnecessary. Such interventions require high-level reasoning to simplify proofs and refactor the codebase. In contrast, syntactic optimizations and increased automation contribute a much smaller share. Only Opus 5 makes a significant number of syntactic optimizations, showing its ability to balance high-level reasoning with low-level code manipulation. Interestingly, GPT-5.6 Sol's proof rewrites add code, likely because it folds abstractions that are used only once into the retained proofs.

### 4.2.2 Performance Is Stable Across Instructions

In the original prompt, we instruct models to make real simplifications, extract reusable common helpers, and refactor the codebase. These instructions may bias the models towards certain compression strategies, so we perform an ablation study that removes these instructions to see if they

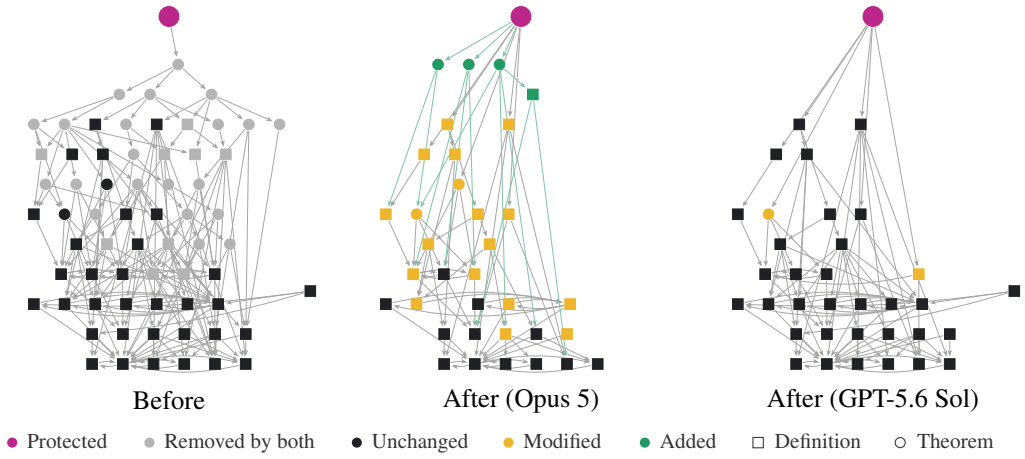



Figure 8: Dependency graphs for a repository before and after compression show that Opus 5 modifies and inserts declarations extensively, while GPT-5.6 Sol focuses on removing declarations.

affect performance. In two variants, we replace the instruction with "Compress the proofs in this codebase" or "Reduce the size of this Lean codebase", respectively with and without a no-cheating instruction. The full prompts can be found in Appendix G. We run this ablation on a subset of four repositories using Opus 5 (Table 11). The similar scores and compression profiles in Figure 7(b) suggest that Opus 5's strategy is robust to these instruction changes on the four tested repositories, and that prompting for real simplification does not significantly bias the model's behavior.

#### 4.2.3 Qualitative Compression Analysis

We now examine selected case studies to understand how models achieve compression beyond the aggregate categories in §4.2.1. Appendix E provides a deeper analysis on more examples.

**Similar score through different strategies.** In the example shown in Figure 8, both Opus 5 and GPT-5.6 Sol obtain high compression scores and remove many live declarations. However, the models use completely different strategies to rewire the proof. Opus 5 adds new helper lemmas and modifies many of the existing proofs, while GPT-5.6 Sol makes minimal edits to existing proofs and incorporates the deleted declarations into the proof of the protected theorem.

**Encoding integer tables as syntax optimization.** In another instance, GPT-5.6 Sol reduces the token count by encoding 102 large integer tables as base-62 strings and reconstructing the entries during expansion using a function that decodes the strings. Since base-62 strings are more compact than the original tables, this optimization significantly reduces the token count. This kind of behavior is rare in our benchmark (as shown in §4.2.1), but it demonstrates the potential for models to find creative solutions to reduce code size.

#### 4.2.4 Merging Two Compressions Gives Small Improvements

The similarity between the quantitative strategy profiles across models suggests that models perform similar, unmergeable edits on the same repository. To confirm this, we ask GPT-5.6 Luna to merge the repositories compressed by Opus 5 and GPT-5.6 Sol. We provide the two versions on separate branches of a Git repository and give the model 4 hours to merge them into a single branch. We run this experiment on the same 12 repositories as Fable 5.1 and GPT-6 Astra (Table 11) and report results in Table 4. Opus 5 achieves higher compression than GPT-5.6 Sol on all 12 repositories, and Luna's merged outputs improve on Opus 5's average score by only 1.1 percentage points. The modest gain suggests limited complementarity between the two outputs, although limitations in Luna's merging strategy could also explain the result.

Table 4: Merge of Opus 5 and Sol 5.6 compressions achieves small gains.

| Method | Score (%) |
|---|---|
| GPT-5.6 Sol | 35.02 |
| Opus 5 | 46.10 |
| Merge | **47.20** |

#### 4.2.5 TOOL USE VARIES ACROSS MODELS

To analyze agent behavior, we assign each tool call to one of ten classes: *Read* (inspect files), *Edit* (modify files), *Search* (run search utilities), *Git* (run Git commands), *Build* (run `lake build`), *Lake* (run other Lake commands), *Measure* (count tokens with `proof_length.py`), *Verify* (run `lean_verify`), *Sleep* (`sleep` bash commands), and *Other*.

As shown in Figure 9, models vary in their tool use. Gemini uses the most tool calls, and notably a considerable chunk of *Sleep* calls. Opus and Sol differ in their verification strategies: Sol relies mostly on *Build* actions, while Opus relies more on *Verify*. Sol makes the most edits but compresses less than Opus, showing that more edits do not necessarily pay off.

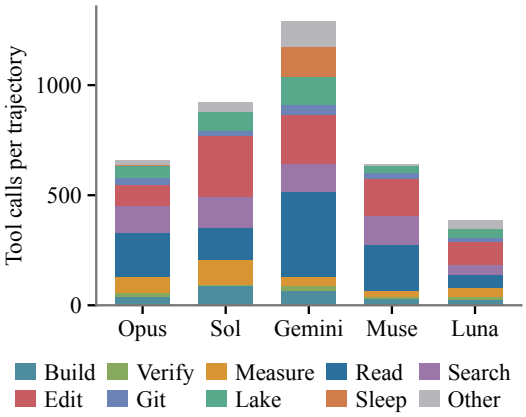



Figure 9: Tool calls per trajectory by model.

## 5 DISCUSSION AND LIMITATIONS

**Alternative metrics.** Lean token compression is useful for evaluating LLMs on repository-scale compression tasks, but other compression metrics can also be optimized. For example, heartbeat counts track allocation-based elaboration work and provide a different optimization objective, as we discuss in Appendix D. We chose Lean token count because it has been used in prior work (Lu et al., 2026), is easy to interpret, and cheap to compute.

**Approximate metric.** Lean token count is only an approximate measure of more important properties such as maintainability. For example, models can make syntactic optimizations that do not improve maintainability. However, as shown in §4, these optimizations only account for a small proportion of compression and they add an additional dimension of difficulty to the compression task: models must trade-off the time available towards making larger structural changes versus making smaller syntactic optimizations. There is no easy way to robustly disallow syntactic optimizations.

**Missing model evaluations.** The high costs of evaluations prevented us from benchmarking all models on all repositories. For instance, GPT-5.6 Sol alone cost more than $8,000 in API fees, and a full evaluation of Fable 5.1 and GPT-6 Astra would have cost thousands more. Table 2 reports preliminary evaluations of Fable 5.1 and GPT-6 Astra, but evaluation cost remains a bottleneck.

**Future work.** Each LEANLEAN task starts from a compiling repository and asks the model to reduce its size while preserving the protected results. Unlike the binary success reward typically used in mathematical environments, compression provides a continuous reward based on the reduction in Lean tokens. This automatically verifiable signal makes these tasks promising environments for reinforcement learning. However, training on this objective would require safeguards against syntactic shortcuts to prevent reward hacking. A key question is whether training on repository compression also improves theorem proving and proof refactoring beyond the benchmark.

## 6 CONCLUSION

We introduce LEANLEAN, a benchmark for compressing 64 real-world Lean repositories. Opus 5 leads the benchmark with a 48.3% compression score. Most gains come from restructuring the proof dependencies by making previously used declarations unnecessary, but Opus 5 also makes use of syntactic optimizations. LEANLEAN provides an ideal testbed for LLM agent evaluation on a range of skills, including long-context reasoning, mathematical reasoning, and code manipulation.

## AI USE STATEMENT

In this work, we used generative AI tools to aid and polish writing, to support qualitative data analysis, and to interpret results. We have not used generative AI tools for any other required disclosure task, and the rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for code generation.

We have extensively reviewed all AI-assisted work. We used a combination of Codex and Claude Code, with extensive human review and editing where necessary. We also used ChatGPT to help with editing the text of this paper, but the content was always first written by an author and then edited by ChatGPT. After any AI-written edits, we extensively reviewed and edited the text to ensure that it was accurate and clear. No AI was used for ideation or to generate any of the original content of this paper. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## ETHICS STATEMENT

We introduce a benchmark for a mathematical task and have not used any sensitive data or human subjects in our work. We abide by the licenses of the repositories we use in our benchmark. We do not foresee any negative societal impacts from this work.

## REPRODUCIBILITY STATEMENT

We describe our methodology in detail and include all code and data necessary to reproduce our results in the supplementary material. This code contains a detailed README with instructions for reproducing our results.

## REFERENCES

Riyaz Ahuja, Tate Rowney, Jeremy Avigad, and Sean Welleck. Improver 2: Iteratively self-improving lms for neurosymbolic proof optimization. *CoRR*, abs/2605.22885, 2026. doi: 10.48550/ARXIV.2605.22885. URL https://doi.org/10.48550/arXiv.2605.22885.

Vitaly Aksenov, Eve Bodnia, Michael H. Freedman, and Michael Mulligan. Compression is all you need: Modeling mathematics, 2026. URL https://arxiv.org/abs/2603.20396.

Anthropic. Formalizing fermat's last theorem. https://www.anthropic.com/research/formalizing-fermats-last-theorem, September 2026a. Published September 4, 2026.

Anthropic. System card: Claude fable 5.1 & claude mythos 5.1. Technical report, Anthropic, September 2026b. URL https://www-cdn.anthropic.com/0339e6a7c5c7b87f5c07798616dc32c215d14235/Claude%20Fable%205.1%20&%20Claude%20Mythos%205.1%20System%20Card.pdf. Accessed: 2026-09-25.

Anthropic. System card: Claude opus 5. Technical report, Anthropic, July 2026c. URL https://www-cdn.anthropic.com/ceaf5c7ff2783855203fde8208ec311252dced5b/Claude%20Opus%205%20System%20Card.pdf. Accessed: 2026-09-25.

Jeremy Avigad. What is mathematics now, and what should it be?, 2026. URL https://arxiv.org/abs/2608.23218.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *CoRR*, abs/2302.12433, 2023. doi: 10.48550/ARXIV.2302.12433. URL https://doi.org/10.48550/arXiv.2302.12433.

Anne Baanen, Matthew Robert Ballard, Johan Commelin, Bryan Gin-ge Chen, Michael Rothgang, and Damiano Testa. Growing mathlib: maintenance of a large scale mathematical library. In Valeria de Paiva and Peter Koepke (eds.), *Intelligent Computer Mathematics - 18th International*

*Conference, CICM 2025, Brasilia, Brazil, October 6-10, 2025, Proceedings*, volume 16136 of *Lecture Notes in Computer Science*, pp. 51–70. Springer, 2025. doi: 10.1007/978-3-032-07021-0_4. URL https://doi.org/10.1007/978-3-032-07021-0_4.

Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge (eds.), *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pp. 32–48. Springer, 2012. doi: 10.1007/978-3-642-31374-5_3. URL https://doi.org/10.1007/978-3-642-31374-5_3.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience. *CoRR*, abs/2512.17260, 2025. doi: 10.48550/ARXIV.2512.17260. URL https://doi.org/10.48550/arXiv.2512.17260.

Shuze Chen, Kunal Marwaha, Xiaoyang Lu, Henry Yuen, and Tianyi Peng. Prove2me: An open collaborative platform for scaling math formalization. *CoRR*, abs/2608.28433, 2026. doi: 10.48550/ARXIV.2608.28433. URL https://doi.org/10.48550/arXiv.2608.28433.

Yiming Fu, Peixuan Liu, Zichen Wang, and Kun Yuan. Proof-refactor: Refactoring generated formal proofs into modular artifacts. *CoRR*, abs/2606.03743, 2026. doi: 10.48550/ARXIV.2606.03743. URL https://doi.org/10.48550/arXiv.2606.03743.

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. Herald: A natural language annotated lean 4 dataset. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=Se6MgCtRhz.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=NiNIthntx7.

Google. Gemini 3.8 flash model card. Technical report, Google, September 2026. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-8-Flash-Model-Card.pdf. Accessed: 2026-09-25.

Alex Gu, Bartosz Piotrowski, Fabian Gloeckle, Kaiyu Yang, and Aram H. Markosyan. Proofoptimizer: Training language models to simplify proofs without human demonstrations. *CoRR*, abs/2510.15700, 2025. doi: 10.48550/ARXIV.2510.15700. URL https://doi.org/10.48550/arXiv.2510.15700.

Jiewen Hu, Thomas Zhu, and Sean Welleck. minictx: Neural theorem proving with (long-)contexts. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=KIgaAqEFHW.

Jiedong Jiang, Wanyi He, Yuefeng Wang, Guoxiong Gao, Yongle Hu, Jingting Wang, Nailing Guan, Peihao Wu, Chunbo Dai, Liang Xiao, and Bin Dong. FATE: A formal benchmark series for frontier algebra of multiple difficulty levels. *CoRR*, abs/2511.02872, 2025. doi: 10.48550/ARXIV.2511.02872. URL https://doi.org/10.48550/arXiv.2511.02872.

Tanya Klowden and Terence Tao. Mathematical methods and human thought in the age of ai, 2026. URL https://arxiv.org/abs/2603.26524.

Joshua Ong Jun Leang, Zheng Zhao, Mihaela Catalina Stoian, Qiyuan Xu, Haonan Li, Wenda Li, Shay B. Cohen, and Eleonora Giunchiglia. Pythagoras-prover: Advancing efficient formal proving via augmented lean formalisation. *CoRR*, abs/2606.12594, 2026. doi: 10.48550/ARXIV.2606.12594. URL https://doi.org/10.48550/arXiv.2606.12594.

leanprover. comparator, 2026. URL https://github.com/leanprover/comparator. GitHub repository, retrieved 2026-09-01.

Austin Letson, Leopoldo Sarra, Auguste Poiroux, Oliver Dressler, Paul Lezeau, Dhyan Aranha, Frederick Pu, Aaron Hill, Miguel Corredera Hidalgo, Julian Berman, George Tsoukalas, and Lenny Taelman. Sorrydb: Can AI provers complete real-world lean theorems? *CoRR*, abs/2603.02668, 2026. doi: 10.48550/ARXIV.2603.02668. URL https://doi.org/10.48550/arXiv.2603.02668.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *CoRR*, abs/2508.03613, 2025. doi: 10.48550/ARXIV.2508.03613. URL https://doi.org/10.48550/arXiv.2508.03613.

Jialin Lu, Soonho Kong, Rodrigo Stehling, Kaiyu Yang, Zhangyang Wang, Weiran Sun, and Wuyang Chen. Lean refactor: Multi-objective controllable proof optimization via agentic strategy search. *CoRR*, abs/2605.20244, 2026. doi: 10.48550/ARXIV.2605.20244. URL https://doi.org/10.48550/arXiv.2605.20244.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. Swe-fficiency: Can language models optimize real-world repositories on real workloads? *CoRR*, abs/2511.06090, 2025. doi: 10.48550/ARXIV.2511.06090. URL https://doi.org/10.48550/arXiv.2511.06090.

Meta. Muse spark 1.3. Technical report, Meta, September 2026. URL https://dev.meta.ai/models/muse-spark. Accessed: 2026-09-25.

Mistral. Leanstral 1.5: Proof abundance for all. Technical report, Mistral, July 2026. URL https://mistral.ai/news/leanstral-1-5/. Accessed: 2026-09-25.

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

OpenAI. gpt-oss-120b & gpt-oss-20b model card. Technical Report arXiv:2508.10925, OpenAI, August 2025. URL https://arxiv.org/abs/2508.10925.

OpenAI. Gpt-5.6 system card. Technical report, OpenAI, July 2026a. URL https://deploymentsafety.openai.com/gpt-5-6/gpt-5-6.pdf. Accessed: 2026-09-25.

OpenAI. Gpt-6 astra system card. Technical report, OpenAI, September 2026b. URL https://deploymentsafety.openai.com/gpt-6-astra/gpt-6-astra.pdf. Accessed: 2026-09-25.

OpenAI. On the Navier–Stokes millennium prize problem. https://openai.com/index/navier-stokes-solution/, September 2026c. Published September 8, 2026.

Palomar. Palomar Registry. https://palomar-registry.org/, 2026. Accessed September 7, 2026.

Auguste Poiroux, Antoine Bosselut, and Viktor Kuncak. Rlmeval: Evaluating research-level neural theorem proving. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pp. 10946–10957. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-EMNLP.581. URL https://doi.org/10.18653/v1/2025.findings-emnlp.581.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *CoRR*, abs/2504.21801, 2025. doi: 10.48550/ARXIV.2504.21801. URL https://doi.org/10.48550/arXiv.2504.21801.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: challenging software optimization tasks for evaluating swe-agents. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/ddd98389004161943e646322a16348c3-Abstract-Datasets_and_Benchmarks_Track.html.

Terence Tao. Mathematics in the age of ai, 2026. URL https://arxiv.org/abs/2608.16753.

The mathlib Community. The lean mathematical library. In Jasmin Blanchette and Catalin Hritcu (eds.), *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pp. 367–381. ACM, 2020. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

Alex Thillen, Niels Mündler, Veselin Raychev, and Martin T. Vechev. Codetaste: Can llms generate human-level code refactorings? *CoRR*, abs/2603.04177, 2026. doi: 10.48550/ARXIV.2603.04177. URL https://doi.org/10.48550/arXiv.2603.04177.

William P. Thurston. On proof and progress in mathematics, 1994. URL https://arxiv.org/abs/math/9404236.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/1582eaf9e0cf349e1e5a6ee453100aa1-Abstract-Datasets_and_Benchmarks_Track.html.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. *CoRR*, abs/2509.22819, 2025. doi: 10.48550/ARXIV.2509.22819. URL https://doi.org/10.48550/arXiv.2509.22819.

Jirí Vyskocil, David Stanovský, and Josef Urban. Automated proof compression by invention of new definitions. In Edmund M. Clarke and Andrei Voronkov (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pp. 447–462. Springer, 2010. doi: 10.1007/978-3-642-17511-4_25. URL https://doi.org/10.1007/978-3-642-17511-4_25.

James Owen Weatherall and Jesse Wolfson. Correctness, artificial intelligence, and the epistemic value of mathematical proof, 2026. URL https://arxiv.org/abs/2602.12463.

Yutong Wu, Di Huang, Ruosi Wan, Yue Peng, Shijie Shang, Chenrui Cao, Lei Qi, Rui Zhang, Xishan Zhang, Zidong Du, Jie Yang, and Xing Hu. Stepfun-formalizer: Unlocking the autoformalization potential of llms through knowledge-reasoning fusion. In Sven Koenig, Chad Jenkins, and Matthew E. Taylor (eds.), *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*, pp. 33980–33988. AAAI Press, 2026. doi: 10.1609/AAAI.V40I40.40691. URL https://doi.org/10.1609/aaai.v40i40.40691.

Huajian Xin, Luming Li, Xiaoran Jin, Jacques D. Fleuriot, and Wenda Li. Ape-bench I: towards file-level automated proof engineering of formal math libraries. *CoRR*, abs/2504.19110, 2025. doi: 10.48550/ARXIV.2504.19110. URL https://doi.org/10.48550/arXiv.2504.19110.

Yutong Xin, Qiaochu Chen, Greg Durrett, and Isil Dillig. Verisoftbench: Repository-scale formal verification benchmarks for lean. *CoRR*, abs/2602.18307, 2026. doi: 10.48550/ARXIV.2602.18307. URL https://doi.org/10.48550/arXiv.2602.18307.

Yisen Xu, Jinqiu Yang, and Tse-Hsun Chen. Swe-refactor: A repository-level benchmark for real-world llm-based code refactoring. *CoRR*, abs/2602.03712, 2026. doi: 10.48550/ARXIV.2602.03712. URL https://doi.org/10.48550/arXiv.2602.03712.

Yuanhe Zhang, Yuekai Sun, Taiji Suzuki, Jason D. Lee, and Fanghui Liu. Leanmarathon: Toward reliable AI co-mathematicians through long-horizon lean autoformalization. *CoRR*, abs/2606.05400, 2026. doi: 10.48550/ARXIV.2606.05400. URL https://doi.org/10.48550/arXiv.2606.05400.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

# A Dependency Mapping

This appendix explains how we construct dependency edges for preprocessing and graph extraction (§3.2). For declarations $X$ and $Y$, an edge $X \to Y$ indicates that $X$ depends on $Y$. Dependency analysis is complicated by Lean's elaboration process, which can introduce dependencies that are not explicit in the source code. We therefore combine three sources of evidence to recover the complete set of edges: source references, compiled kernel terms (`.olean`), and elaborator-resolved IDE references (`.ilean`).

Table 5 reports the number of edges recovered by each source across all repositories, showing that all are necessary. After preprocessing, we verify that the protected results are still proven with Comparator, which shows that the combined sources are sufficient to preserve correctness.

Table 5: Coverage of the dependency-evidence sources across all repositories. Unique edges are only found by the indicated layer and coverage is measured relative to the union of all sources.

| Evidence layer | Edges | Coverage | Unique edges |
|---|---|---|---|
| Kernel (`.olean`) | 1,036,659 | 88.11% | 351,389 |
| Source | 762,302 | 64.79% | 97,845 |
| `.ilean` | 692,064 | 58.82% | 8,363 |
| Found by all three | 595,464 | 50.61% | — |
| Union | 1,176,579 | 100.00% | — |

For each source, we give an example that illustrates a dependency recovered by only one of them. In each example, the dependent declaration $X$ is shown in blue, the dependency $Y$ in red, and shaded text identifies the evidence for the edge.

**Source references.** We scan each declaration's source range and resolve identifiers using its imports, namespaces, and `open` declarations, falling back to suffix matching when a name cannot be resolved exactly. This captures dependencies in source constructs that do not appear as ordinary references in the build artifacts. For instance, in Figure 10, the `collar_boundary` field of `RelativeSubdivisionBoundary` has a default tactic proof that calls `relativeSubdivisionChain_boundary`. Lean stores this proof as unelaborated syntax and runs it when the structure is constructed without an explicit value for the field. The compiled structure therefore stores the lemma's name as syntax data rather than a constant reference, and `.ilean` records no reference because the default proof has not been elaborated. No retained declaration uses this default after stripping, so only the source edge keeps the lemma. Without this edge, stripping would delete the lemma and leave a default proof that fails with `Unknown identifier`.

`RelativeSubdivisionBoundary` ⟶ `relativeSubdivisionChain_boundary` *found only by source text*

```
-- NRR/PrimePolyhedron/FoxNeuwirth/StableCollarRelativeSubdivision.lean
theorem relativeSubdivisionChain_boundary (R : Type) [CommRing R] (X : TopCat)
    (N0 N1 n : Nat) (c : singularChainGroup R X n) (hc : ...) :
    (singularBoundary R X n).hom (relativeSubdivisionChain R X N0 N1 n c) =
      barycentricSubdivisionIterLinearMap R X N0 n c -
        barycentricSubdivisionIterLinearMap R X N1 n c := ...

structure RelativeSubdivisionBoundary (R : Type) [CommRing R] (X : TopCat) (n : Nat) where
  baseCycle : singularChainGroup R X n
  baseCycle_closed : ...
  lowerLevel : Nat
  upperLevel : Nat
  collarChain : singularChainGroup R X (n + 1) :=
    relativeSubdivisionChain R X lowerLevel upperLevel n baseCycle
  collar_boundary : (singularBoundary R X n).hom collarChain = ... := by
    simpa using relativeSubdivisionChain_boundary R X lowerLevel upperLevel n
      baseCycle baseCycle_closed
-- the default proof is kept as unelaborated syntax; it runs only where the
-- structure is built without supplying collar_boundary
```

Figure 10: A structure field whose dependency is recovered only from source references.

**Kernel terms (`.olean`).** Lean's `.olean` files contain compiled declarations, including their elaborated types and values. These terms reveal dependencies introduced during elaboration that are not explicit in the source. Before collecting constants, we erase proof terms, so this layer records dependencies from types, data-carrying values, and structural edges, but not from proofs. Dependencies used only inside proofs are recovered by the source and `.ilean` layers. In Figure 11, `localBranchWitnessBox` uses `Finset.univ` to enumerate all values of `Branch`. This requires a `Fintype Branch` instance, which Lean supplies using `instFintypeBranch`, generated by the `deriving Fintype` clause. The instance is not named in the source, and the elaborator records no reference to it in `.ilean`. The compiled definition, however, passes the instance explicitly to `Finset.univ`, so this edge is recovered only from `.olean`.

`localBranchWitnessBox` ⟶ `instFintypeBranch` *found only by `.olean`*

```
-- ErdosProblems/Erdos730/BranchEvents.lean
inductive Branch where
  | P | Q | R | S
  deriving DecidableEq, Fintype, Repr   -- creates the instance instFintypeBranch

abbrev LocalBranchWitness := Branch × ObstructionWitness

def localBranchWitnessBox (X : N) : Finset LocalBranchWitness :=
  Finset.univ.product (witnessBox X)    -- needs a Fintype Branch instance
-- compiled value: (@Finset.univ Branch instFintypeBranch).product (witnessBox X)
```

Figure 11: An implicit instance dependency recovered only from compiled kernel terms (`.olean`).

**IDE references (`.ilean`).** Lean's `.ilean` files record references resolved by the elaborator and their source locations. These references capture dependencies that require type information to resolve. In Figure 12, `locked_csg_of_noncontracting` constructs a `CS_grammar` with the field assignment `rules := nc_locked_allRules g`. Several visible structures have a field named `rules`, but the expected type identifies this occurrence as `CS_grammar.rules`. The source layer cannot disambiguate the name, and the compiled term uses the structure constructor rather than the field name. The edge is therefore recovered only from `.ilean`.

`locked_csg_of_noncontracting` ⟶ `CS_grammar.rules` *found only by `.ilean`*

```
-- Langlib/Grammars/ContextSensitive/Definition.lean
structure CS_grammar (T : Type) where
  (nt : Type)
  (initial : nt)
  (rules : List (csrule T nt))    -- grammar T also has a field `rules`
  (output_nonempty : forall r in rules, r.output_string != [])

-- Langlib/Grammars/NonContracting/Equivalence/ContextSensitive.lean
noncomputable def locked_csg_of_noncontracting (g : grammar T)
    (_hg : grammar_noncontracting g) : CS_grammar T where
  nt := NC_LockedNT g
  initial := .orig g.initial
  rules := nc_locked_allRules g   -- CS_grammar.rules, fixed by the expected type
  output_nonempty := by ...
-- compiled value: CS_grammar.mk (NC_LockedNT g) ... (nc_locked_allRules g) ...
```

Figure 12: A dependency recovered only from elaborator-resolved IDE references (`.ilean`).

## B Model Configurations

We evaluated each model through its native coding-agent harness rather than imposing a shared, hand-tuned sampling configuration. The only generation parameter we selected was the reasoning-effort level reported in Table 6, all other parameters were inherited from the pinned harness version. The only exception was Muse Spark 1.3, which due to a configuration issue, also used lower output token and compaction limits, as well as a reduced output length limit for each tool. For this reason, we consider Muse Spark 1.3's results to be a lower bound on the model's performance.

For Leanstral 1.5, since it is provided for free by the Mistral organization, we estimate its cost using GPT-OSS-120B (OpenAI, 2025), which has comparable size at 117B compared to 119B.

For every run, subagents were disabled to prevent a single benchmark attempt from expanding into an uncontrolled number of model calls. For improved reproducibility, optional skills and MCP servers were disabled while retaining each harness's bundled system instructions and standard local coding tools. The limits in Table 6 give the maximum context and output sizes in model tokens, as configured by the harness.

Table 6: Model endpoints and effective reasoning configurations.

| Model | Provider | Native harness (version) | Effort | Max. context | Max. output |
|---|---|---|---|---|---|
| GPT-5.6 Sol | OpenAI | Codex (0.154.0) | `xhigh` | 258,400 | 128,000 |
| GPT-6 Astra | OpenAI | Codex (0.153.3) | `medium` | 258,400 | 128,000 |
| Opus 5 | Anthropic | Claude Code (2.1.269) | `high` | 200,000 | 64,000 |
| Fable 5.1 | Anthropic | Claude Code (2.1.269) | `high` | 200,000 | 64,000 |
| Gemini 3.8 Flash | Google | Antigravity CLI (1.1.26) | `high` | 1,048,576 | 65,536 |
| Leanstral 1.5 | Mistral | Mistral Vibe (2.25.0) | `high` | 262,144 | 131,072 |
| GPT-5.6 Luna | OpenAI | Codex (0.154.0) | `xhigh` | 258,400 | 128,000 |
| Muse Spark 1.3[†] | Meta | Muse (1.0.3-R2198.1) | `max` | 512,000 | 32,768 |

[†] Model configuration issue: Muse Spark 1.3 used a lower compaction threshold, output token limit, and a reduced output length limit for each tool than the default.

# C Compression Origin Implementation Details

We explain how we measure the origin of Lean token compression in the LeanLean benchmark. To classify each part of the compression into the six categories, we reuse the dependency graph extraction from the preprocessing pipeline, and analyze the dependency graph of the repository before ($G_1$) and after ($G_2$) compression. Every declaration is then classified based on changes to its source and dependencies. To classify syntactic optimizations and automation rewrites, we apply rule-based methods to the diff of the repository before and after compression.

## C.1 Declaration Categorization

A declaration is *deleted* if it occurs only in $G_1$, *added* if it occurs only in $G_2$, and *preserved* if it occurs in both. We identify declarations by name. A deleted declaration $d$ is counted as dead code if no preserved declaration depends on it in $G_1$. This differs from the preprocessing stage because we conservatively preserve some additional declarations (see §3.2). Otherwise, it belongs to the deleted declarations category. With our edge convention, dependencies are reached by following outgoing edges. Added declarations incur a token cost, except for abbreviations, whose costs are assigned to syntax optimization (see Appendix C.2). For preserved declarations, we split the token change between syntax optimization, automation rewrite, and proof rewrite.

## C.2 Syntax Optimization

A syntax optimization expresses the same argument in fewer tokens. We attribute syntax first. Automation (Appendix C.3) is measured on the old text with syntax already applied, and the remainder counts as proof rewrite.

**Spelling rules.** Within each changed block of a declaration, we apply rewrite rules for common Lean idioms to the old text, for example:

| Claim type | Merged calls | Dot notation |
|---|---|---|
| `-have hx0 : 0 ≤ x := ...` | `-rw [hcycle]` | `-exact le_antisymm h2 h1` |
| `+have hx0 := ...` | `-rw [hi, hj]` | `+exact h2.antisymm h1` |
| | `+rw [hcycle, hi, hj]` | |

A rule is credited only if every token it removes is also deleted in the token-level diff, and each removal is flanked by at least three tokens that appear in the new text. This rejects, e.g., `simp only [a, b, c]` → `simp` when the agent kept `a`. The full list of rules is in our code release.

**Inlining.** A helper deleted and written out at its use changes only spelling. We recognize it when the helper's proof reappears verbatim at its former use site:

```
-  have h := hq_smul 0 x
-  simpa only [zero_smul, norm_zero, pow_two, zero_mul] using h
+  simpa using hq_smul 0 x
```

**New abbreviations.** New definitions, notation and macros count as syntax. Each is credited with what one use saves, the tokens of its expansion minus those of its call, and charged its definition once. The price is computed from the definition, with each parameter counted as one token, and includes nested abbreviations:

```
-theorem integrable_jointUnitExponentialDensity :
+zd integrable_jointUnitExponentialDensity :
     Integrable (jointUnitExponentialDensity :
-      (ℝ × (Fin n → ℝ)) → ℝ) := by
-  rw [Measure.volume_eq_prod]
+      āģ → ℝ) zb
+  zC8
```

### C.3 Automation Rewrite

An automation rewrite replaces hand-written proof steps with a call to automation such as `simp`, `aesop` or `linarith`. Only preserved declarations contribute to automation rewrites. Figure 13 shows an edit that Opus 5 made. In step (a), `simp` is upgraded to `aesop`, and in (b), manual steps are replaced by automation. Both are credited as automation rewrites, saving 7 and 6 tokens. In (c), the agent only dropped a hint, which counts as proof rewrite. Considering line diffs here does not correctly classify the edits into the correct structure, and we therefore propose our step-based edit detection algorithm.

Line diff

```
   · have hHead := ...
-    simp [hTail, hHead, rightEndpoint]
+    aesop                              -- block 1
   · have hTail' : B.core.tail α = 1 := by
       rcases ... with h | h
-      · exact (hTail h).elim
-      · exact h
-    simpa [hTail, hTail', rightEndpoint] using ...
+      simp_all
+      simp_all                         -- block 2
+    simpa [hTail', rightEndpoint] using ...
```

Step edits (ours)

```
   · have hHead := ...
-    simp [hTail, hHead, rightEndpoint]
+    aesop                              -- (a)
   · have hTail' : B.core.tail α = 1 := by
       rcases ... with h | h
-      · exact (hTail h).elim
-      · exact h
+      simp_all
+      simp_all                         -- (b)
-    simpa [hTail, hTail', rightEndpoint] using ...
+    simpa [hTail', rightEndpoint] using ...  -- (c)
```

Figure 13: A proof compressed by Opus, split by a line diff (left) and by our step alignment (right). Edit (b) is an automation rewrite while edit (c) is a proof rewrite. The normal `git diff` does not consider these as two separate changes, but our step-based algorithm does.

**Steps.** We split proofs into tactic steps at `;`, `<;>`, `·`, `by`, `=>` and line breaks. Following Lean's layout rules, lines inside brackets or indented past the start of a step continue that step. Each step is labeled (1) *automation*: `simp`, `grind`, `omega`, `linarith`, macros that expand only to automation, and similar tactics, (2) *routing*: steps that restructure goals without closing them, such as `intro`, `constructor`, `cases` or a `have h :  T` that only states a subgoal, or (3) *manual*: everything else, such as `rw`, `exact`, `apply` or `calc`.

**Edits.** We align the old and new steps, anchoring on identical steps, such as `rcases ... with h | h` in Figure 13, and on nearly identical ones: the same tactic with at least 60% of their tokens shared, such as the two `simpa` calls in (c).[3] Without the second kind of anchor, (b) and (c) would be judged together, and the pruned `simpa` would block the automation rewrite in (b). The old steps between two anchors, together with the new steps that replace them, form an *edit*. We first check the whole proof as a single edit, and only split it at the anchors if that check fails. An edit is an automation rewrite if (i) its new steps are only automation and routing, (ii) its new steps include

[3] Python's `difflib` ratio $2M/(|a| + |b|) \geq 0.6$, where $M$ is the number of matched tokens.

automation the old steps did not have, and (iii) it either replaces a manual step, as in (b), or turns existing automation into fewer calls or a different tactic with fewer tokens, as in (a).

**Absorbed steps.** The agent often deletes hand-written steps and lets nearby automation do their work, either the next tactic (left) or a nested call at the use site (right). We credit both as automation rewrites, unless the new proof cites a new repository lemma or uses inlining (see Appendix C.2).

Absorbed by next step

```
   have htSq : 9/16 < t ^ 2 := by nlinarith
-  have hpoly : t < 2 * t ^ 3 := by
-    rw [pow_succ]
-    nlinarith
   nlinarith
```

Absorbed at use

```
-  have h2 : (0 : ℂ) ≤ (2 : ℂ)⁻¹ := by
-    ...
-  exact (...).smul h2
+  exact (...).smul (by positivity)
```

**Exclusions.** Two changes count as proof rewrite instead (Figure 14). The first is passing the automation a repository lemma that the old proof did not use, which moves the work into a new dependency. Library lemmas and unfolding repository definitions are allowed. The second is removing steps without adding automation, as in (c) or when dropping a trailing `<;> ring`. Deleted steps whose lemmas are all still named in the new proof are not absorbed: they count as syntax if inlined verbatim, and as proof rewrite otherwise.

New dependency

```
-  apply le_trans (hdeg v)
-  exact Nat.le_succ _
+  linarith [deg_le G v]
```

Removal only

```
-  field_simp <;> ring
+  field_simp
```

Figure 14: Edits that count as proof rewrites rather than automation rewrites. Left: the automation uses the repository lemma `deg_le`, which the old proof did not. Right: a step is dropped, but no automation is added.

**Saving.** Each automation rewrite is credited with its old tokens minus its new tokens, plus any helper claims it no longer needs. Savings from abbreviations inside new steps are subtracted, since syntax already counts them. Changes to a declaration's statement are never credited to automation.

## D Further Results

We present additional results from the LeanLean benchmark, including histograms of compression scores, cost, and duration per model, as well as cost by repository size and heartbeat reduction. We also report the number of failed runs per model and how they affect scores.

**Histograms.** Figure 15 shows the distributions of compression scores, cost, and duration across runs from the five evaluated models. We highlight three observations. First, the weaker models' compression scores are concentrated at the lower end, with a long right tail, while Opus 5 and GPT-5.6 Sol achieve scores spread more broadly across the range. Second, the most expensive run was with GPT-5.6 Sol, costing a massive $702. Finally, Opus 5's durations cluster near the 12-hour limit, showing it interprets the time budget instruction as a target rather than a hard limit. In contrast, other models do not consistently use all time available.

**Cost and duration by repository size.** Figure 16 shows the mean cost and wall-clock duration per repository across the four size categories used in Figure 6. Interestingly, both GPT-5.6 Sol and Opus 5 spend less on Massive repositories than on Large ones, which may reflect more time spent on compiling and verifying the massive repositories. Durations grow with repository size for most models, and Opus 5 runs longest in every category.

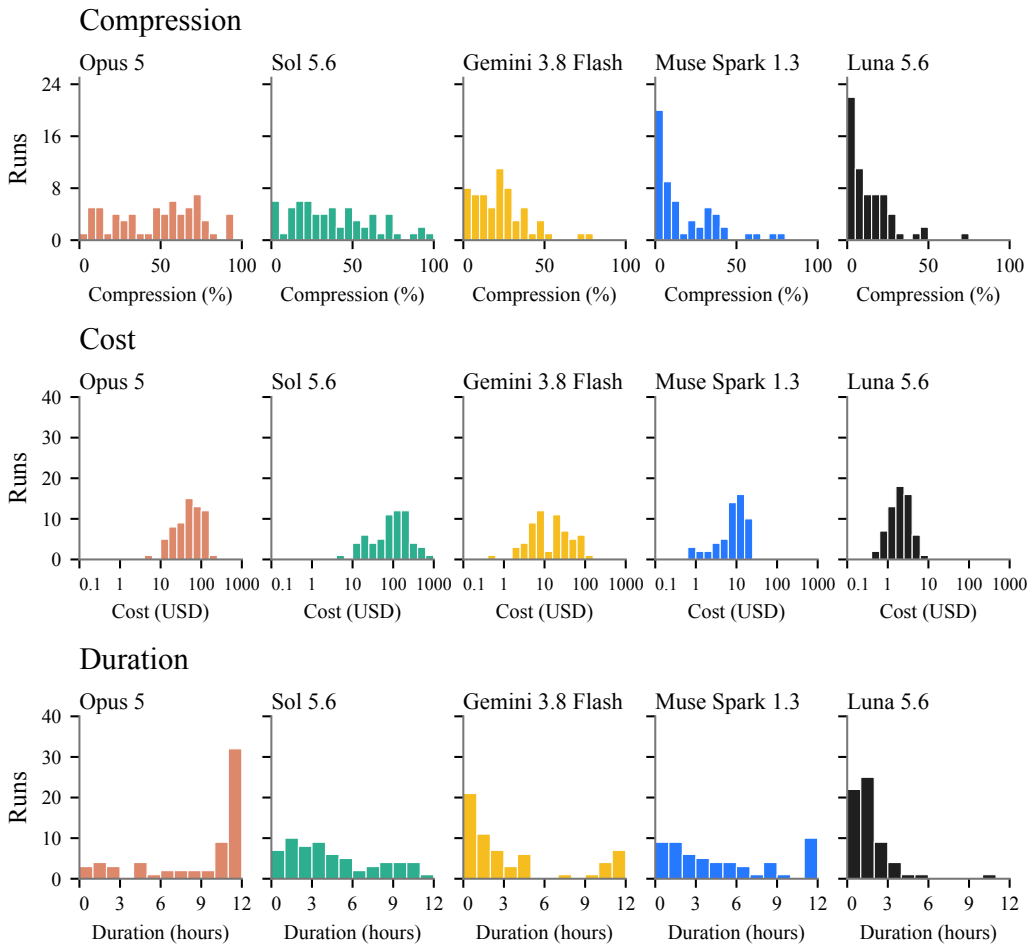



Figure 15: Histograms of compression scores (top), task cost (middle), and task duration (bottom) per model. Cost is shown on a logarithmic scale.

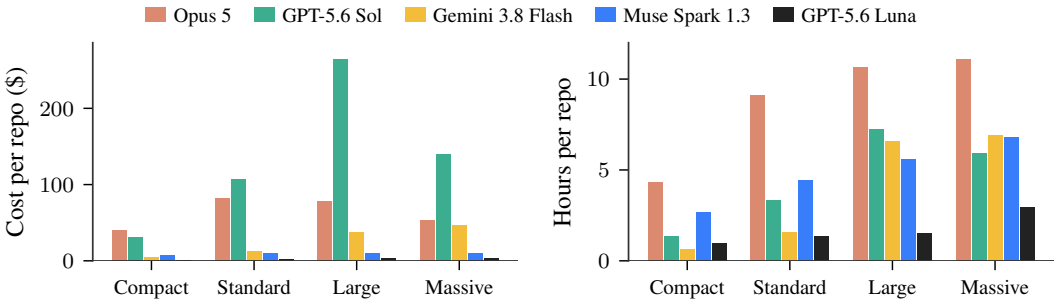



Figure 16: Mean cost per repository in USD (left) and mean duration per repository in hours (right) by repository scale for each evaluated model.

**Heartbeat reduction.** Although the compression task optimizes Lean token count, we also measure changes in Lean heartbeats, an allocation-based measure of elaboration work[4]. Table 7 ranks models by heartbeat reduction, showing that reducing Lean token count does not necessarily reduce elaboration work. For example, Opus 5 increases heartbeat counts by 16.8% despite achieving

[4]We calculate heartbeats by, for each built module, running `lean -j1` with `Elab.async=false` and `maxHeartbeats=0`, using the module's own Lake setup, on a copy of the file ending in `#eval IO.getNumHeartbeats`, and summing the printed counts across modules.

Table 7: Heartbeat leaderboard. Positive reduction indicates less Lean elaboration work. An improved repository has a strictly positive heartbeat reduction. Both columns are over all 64 repositories; failed builds score zero.

| Model | Avg. reduction (%) | Improved repos (%) |
|---|---|---|
| Gemini 3.8 Flash | +7.17 | 71.9 |
| GPT-5.6 Sol | +5.59 | 73.4 |
| Muse Spark 1.3 | +3.11 | 64.1 |
| GPT-5.6 Luna | -2.53 | 78.1 |
| Opus 5 | -16.79 | 46.9 |

Table 8: Outcome of each model's submission on the 64 benchmark repositories. A submission either fails to build, builds but is rejected by the Comparator, or is verified.

| Model | Timeouts | Build failed | Verify failed | Verified |
|---|---|---|---|---|
| Opus 5 | 0 | 0 | 0 | 64 |
| GPT-5.6 Sol | 0 | 1 | 0 | 63 |
| Gemini 3.8 Flash | 8 | 3 | 0 | 61 |
| Muse Spark 1.3 | 6 | 7 | 1 | 56 |
| GPT-5.6 Luna | 0 | 1 | 0 | 63 |

the highest token compression. This increase may reflect greater use of computationally expensive automation tactics such as `grind`.

**Failed runs.** Table 8 reports timeouts, build failures, and verification failures for each model. Failures are rare, with only Muse Spark 1.3 failing in a significant number of repositories ($8/64$), receiving a compression score of zero on each of these eight repositories. Gemini 3.8 Flash times out on eight of the 64 repositories but produces 61 verified submissions, showing that a timeout need not prevent a valid final submission.

# E Case Studies

We present several case studies of compression strategies in LeanLean. Appendix E.1 presents three examples of structural modifications and are the most typical of the changes made by models. Appendix E.2 presents three extreme examples of syntax optimizations, which have little effect on the overall scores but illustrate the models' ability to identify and exploit repeated patterns in proofs.

## E.1 Structural Modifications

**Tucker's Lemma.** Figure 17 shows the formalization of Tucker's lemma (Palomar ID: 2026-09-03-000007) through preprocessing and compression by Opus 5. Preprocessing removes 36 unreachable declarations, saving almost 2,000 Lean tokens. During compression, Opus 5 removes one additional unreachable declaration and 35 declarations previously reachable from the protected results. Opus 5 also adds six theorems and modifies 20 retained declarations, reducing repository size by over $70\%$. These changes combine proof simplifications with the introduction of new helper lemmas. The leftmost protected result, initially isolated, now depends on another protected result.

**Confluence.** The formalization of soundness for finite decreasing-diagram certificates for confluence (Palomar ID: 2026-09-01-000010) illustrates how models reach similar compression through different refactorings (Figure 8). Both Opus 5 and GPT-5.6 Sol remove many declarations and produce repositories of similar size, but take completely different approaches. Opus 5 adds four helper lemmas and modifies many retained declarations to integrate them. GPT-5.6 Sol instead incorporates the logic of the deleted declarations into the proof of the protected result, making few changes to other retained declarations.

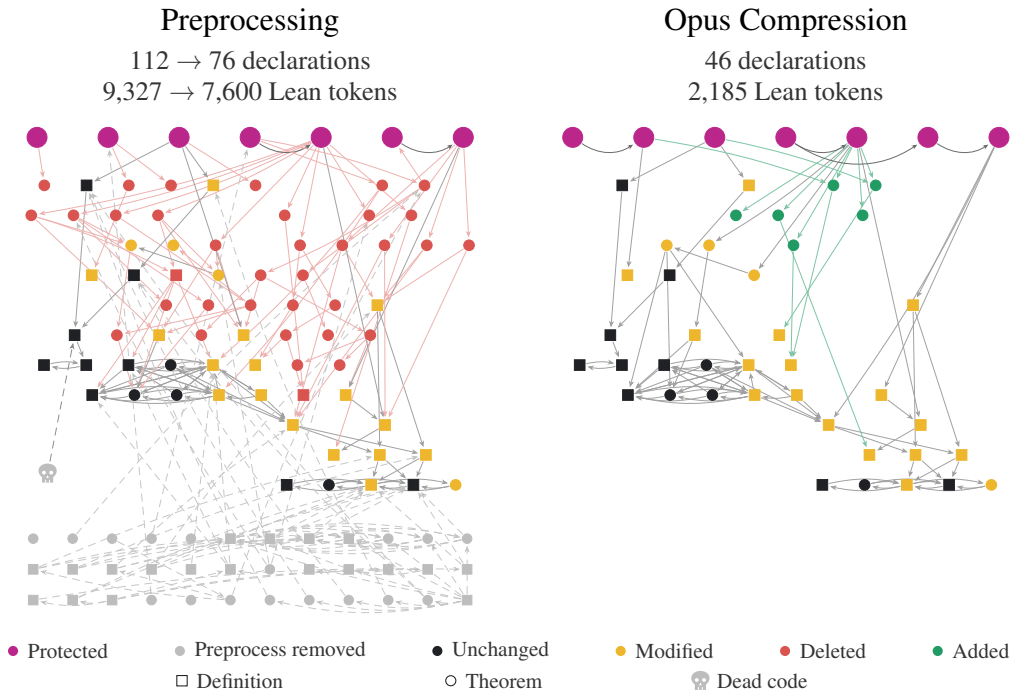



Figure 17: Dependency graphs for Tucker's lemma before preprocessing (left) and after Opus 5 compression (right).

**Morphic Numbers.** The Morphic Numbers formalization classifies morphic numbers as the golden ratio and the plastic number (Palomar ID: 2026-09-01-000008). Figure 18 compares the compressions produced by Opus 5 and Gemini 3.8 Flash. Both models delete all unprotected theorems and rebuild the supporting proof structure with new lemmas. Gemini introduces a larger network of helper lemmas, while Opus 5 introduces four mutually independent helpers. In both outputs, the second protected result from the left depends on the first and third protected results.

### E.2 Syntax Optimization

Models also compress repositories by introducing shorter representations of existing expressions, proofs, and data. The following four examples illustrate these changes and report token savings after accounting for the added definitions and macros.

**A custom DSL.** In the `pengzhang91/Feige` directory (Palomar ID: 2026-08-30-000002), Opus 5 introduces custom syntax for theorem declarations, mathematical expressions, and tactics. For example, the following declaration shrinks from 30 to 17 tokens:

```
-theorem integrable_jointUnitExponentialDensity :
+zd integrable_jointUnitExponentialDensity :
     Integrable (jointUnitExponentialDensity :
-      (ℝ × (Fin n → ℝ)) → ℝ) := by
-  rw [Measure.volume_eq_prod]
+      āǵ → ℝ) zb
+  zC8
   exact Integrable.mul_prod integrable_unitExponentialDensity
     integrable_pi_unitExponentialDensity
```

The proof performs the same rewrite and applies the same lemma. The syntax `zd ... zb ...` expands to `theorem ... := by ...`, while the other abbreviations come from a shared dictionary:

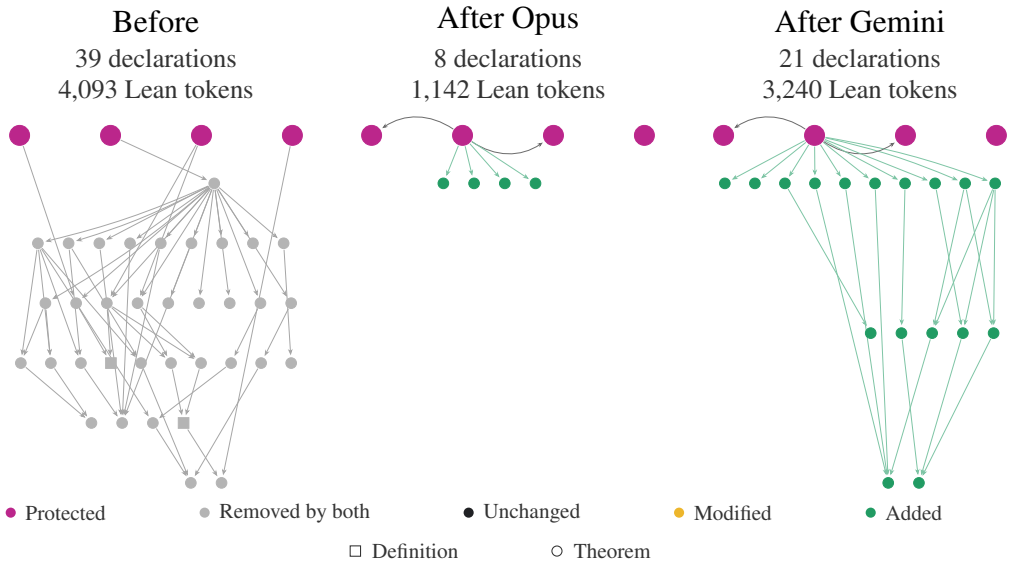



Figure 18: Declaration dependency graphs for the Morphic Numbers example before compression and after compression by Opus 5 and Gemini 3.8 Flash.

```
znℌ (Fin n → ℝ)
znℚ ℝ × ℌ
znāġ (ℚ)
zt zC8 rw [Measure.volume_eq_prod]
```

Here, `zn` introduces notation and `zt` introduces a tactic macro. Across the repository, edits to existing files save 42,458 tokens, while the added dictionary costs 11,807 tokens. The net reduction is 30,651 tokens (28.64%). This repository total includes other edits alongside the illustrated syntax changes.

**Encoding integer tables.** The `SauersML/group-approximation` repository (Palomar ID: 2026-08-24-000006) contains 102 definitions of explicit integer tables. GPT-5.6 Sol encodes each table as a base-62 string and introduces a macro to decode it:

```
-def qNumeratorRow0 : Fin 6 → Fin 22 → ℤ
-  | 0 => integerBlock [29, 43, (-78), 153, ...] (by decide)
-  | 1 => integerBlock [(-44), (-60), (-88), ...] (by decide)
-  ...
-  | 5 => integerBlock [(-84), 9, 59, ...] (by decide)
+def qNumeratorRow0 : Fin 6 → Fin 22 → ℤ :=
+  integerRow% "0000w0001O0002V0004w..."
```

The complete definition shrinks from 517 to 16 tokens. Its 660-character string encodes 132 integers in five-character blocks. The following macro decodes the string:

```
syntax "integerRow% " str : term

macro_rules
  | `(integerRow% $s:str) => do
      let digit (c : Char) :=
        let n := c.toNat
        if n ≤ '9'.toNat then n - '0'.toNat
        else if n ≤ 'Z'.toNat then n - 'A'.toNat + 10
        else n - 'a'.toNat + 36
      let rec decode : List Char → MacroM (Array (TSyntax `term))
        | [] => pure #[]
        | a :: b :: c :: d :: e :: rest => do
            let n :=
              ((((digit a * 62 + digit b) * 62 + digit c) * 62
                + digit d) * 62) + digit e
            let value ← if n % 2 = 0 then
              `($(quote (n / 2)))
            else
              `(-$(quote (n / 2 + 1)))
            return #[value] ++ (← decode rest)
        | _ => Macro.throwError "invalid integer row"
      let values ← decode s.getString.toList
      unless values.size = 132 do
        Macro.throwError "integer row must contain 132 entries"
      let mut blocks := #[]
      for root in List.range 6 do
        let entries := values.extract (root * 22) ((root + 1) * 22)
        blocks := blocks.push
          (← `(integerBlock [$[$entries],*] (by decide)))
      `(finiteTable [$[$blocks],*] (by decide))
```

The macro reads each five-character block as a base-62 natural number $n$, then returns $n/2$ when $n$ is even and $-(n+1)/2$ otherwise. For example:

$$\texttt{0000w} \mapsto 58 \mapsto 29, \qquad \texttt{0001O} \mapsto 86 \mapsto 43, \qquad \texttt{0002V} \mapsto 155 \mapsto -78.$$

The macro reconstructs the six blocks of 22 integers during expansion.

The large saving reflects the Lean token metric: the lexer counts the uninterrupted alphanumeric contents of the string as one token, with the quotes counted separately. Across all 102 definitions, declarations shrink from 53,616 to 1,632 tokens. The syntax declaration and decoder cost 278 tokens, yielding a net saving of 51,706 tokens. The gain comes from packing the same integer data into fewer lexical tokens.

**Syntactic factoring at scale.** In the Heilbronn formalization (Palomar ID: 2026-09-02-000012), Opus 5 compresses the repository from 1,133,524 to 612,720 Lean tokens, saving 45.95%, compared with GPT-5.6 Sol's 3.06%. Most of this reduction is achieved by introducing several layers of abbreviations and macros to share repeated proof syntax. The examine two examples.

First, the lemmas `range_cfg7` and `range_cfg7Family` contain identical 177-token proof bodies. Opus replaces both bodies with the same one-token notation. The following diff abbreviates the repeated branches:

```
-by
-  apply Set.Subset.antisymm
-  · rintro x ⟨i, rfl⟩
-    rcases fin7_cases i with
-      rfl | rfl | rfl | rfl | rfl | rfl | rfl
-    · exact Set.mem_insert_of_mem _ ⟨0, rfl⟩
-    · exact Set.mem_insert_of_mem _ ⟨1, rfl⟩
-    -- Remaining branches omitted.
-  · intro x hx
-    simp only [Set.mem_insert_iff, Set.mem_range] at hx
-    rcases hx with rfl | ⟨j, rfl⟩
-    · exact ⟨6, rfl⟩
-    · rcases fin6_cases j with
-        rfl | rfl | rfl | rfl | rfl | rfl
-      · exact ⟨0, rfl⟩
-      -- Remaining branches omitted.
+Shared_Rd0_0z276
```

The shared notation retains the case analysis, abbreviates some smaller expressions, and costs 162 tokens. Including changes to their statements, the two declarations shrink from 202 to 13 tokens and from 213 to 18 tokens, respectively.

Second, Opus factors proofs that differ in a single lemma application. The 152-token bodies of `prime_central` and `prime_swapped_central` follow the same calculation but use different transport lemmas in a rewrite. For `prime_central`, the change is:

```
-by
-  have hU : 0 < U := by linarith [r.U_lower]
-  have hc : U ^ 2 ≤ -cyclicDet U Sx Sj Tx Tj Rx Rj := by
-    linarith [r.central]
-  have hfac : 0 ≤ V ^ 2 / U ^ 2 :=
-    div_nonneg (sq_nonneg V) (sq_nonneg U)
-  have hm := mul_le_mul_of_nonneg_left hc hfac
-  rw [cyclicDet_prime_transport U V G Sx Sj Tx Tj Rx Rj hU.ne']
-  calc
-    V ^ 2 = (V ^ 2 / U ^ 2) * U ^ 2 := by
-      field_simp [hU.ne']
-    _ ≤ (V ^ 2 / U ^ 2) * (-cyclicDet U Sx Sj Tx Tj Rx Rj) := hm
-    _ = -((V ^ 2 / U ^ 2) * cyclicDet U Sx Sj Tx Tj Rx Rj) := by
-      ring
+Shared_Rd1_0z254 cyclicDet_prime_transport
```

The shared helper retains this calculation while abstracting over the transport lemma:

```
mkn5 by
  Shared_Rd0_1z55
  have hc : Shared_Rd0_2z134 ≤ -Shared_Rd0_2z74 := by
    linarith [r.central]
  have hfac : 0 ≤ Shared_Rd0_3z139 / Shared_Rd0_2z134 :=
    div_nonneg (sq_nonneg V) (sq_nonneg U)
  have hm := mul_le_mul_of_nonneg_left hc hfac
  rw [qp0 U V G Sx Sj Tx Tj Rx Rj hU.ne']
  calc
    Shared_Rd0_3z139 =
        (Shared_Rd0_3z139 / Shared_Rd0_2z134)
          * Shared_Rd0_2z134 := Shared_Rd0_8z164
    _ ≤ (Shared_Rd0_3z139 / Shared_Rd0_2z134)
          * (-Shared_Rd0_2z74) := hm
    _ = -((Shared_Rd0_3z139 / Shared_Rd0_2z134)
          * Shared_Rd0_2z74) := by ring
```

The generator `mkn5` creates the parameterized notation `Shared_Rd1_0z254`, with `qp0` supplying the transport lemma. The helper itself uses shared abbreviations for repeated expressions and proof steps. The second proof therefore becomes:

```
Shared_Rd1_0z254 cyclicDet_prime_swapped_transport
```

Each application costs two tokens, while the shared helper costs 95 tokens. Including statement compression, the two declarations shrink from 221 and 239 tokens to 29 tokens each.

**Heavy use of automation tactics.** In the `WillWhistler/Regts-Sevenster` category-theory formalization (Palomar ID: 2026-08-19-000003), Opus 5 defines a one-character tactic that tries a fixed list of automation tactics in turn:

```
macro "■" : tactic =>
  `(tactic| all_goals first
    | rfl | omega | (simp_all; done) | (aesop; done) | (monoidal; done)
    | (cat_disch; done) | (linarith; done) | grind)
```

Opus then replaces the final steps of proofs with ■ at 1,494 sites, for example:

```
   rw [hin, tensorμ]
-  simp only [Category.assoc]
-  simp only [Iso.inv_hom_id, Category.comp_id]
-  rw [associator_naturality_right_assoc]
-  simp only [← MonoidalCategory.whiskerLeft_comp]
+  ■
```

The repository shrinks from 1,695,971 to 1,587,247 tokens (6.4%). Writing the full tactic chain at every site instead would add 58,216 tokens, more than half of the total reduction. The changes save tokens, but heartbeats rise by 38% (from 26.7M to 36.9M), while the other models' submissions for this repository reduce them by up to 7%. Since the macro is only based on automation tactics, this change is classified as automation rewrite rather than syntax optimization.

## F Muse Spark 1.3 Partial Rerun

Due to the configuration issue of Muse Spark 1.3 (see Appendix B), Muse Spark 1.3 was rerun on 8 of the 12 repositories used to evaluate GPT-6 Astra and Fable 5.1 (Table 11). As Table 9 shows, the corrected configuration places Muse Spark 1.3 above Gemini 3.8 in compression score, by 9.2 percentage points, at a slightly lower cost. It is likely that our full benchmarking results underestimate the performance of Muse Spark 1.3, but its performance remains below that of Opus 5 and GPT-5.6 Sol.

Table 9: Muse Spark 1.3 results might be underestimated on the full benchmark.

| Model | Score (%) | Cost ($) | $t$ (h) |
|---|---|---|---|
| Opus 5 | 54.9 | 73.58 | 7.72 |
| GPT-5.6 Sol | 42.1 | 60.98 | 2.93 |
| Muse Spark 1.3 (corrected) | 30.1 | 17.87 | 2.01 |
| Gemini 3.8 Flash | 20.9 | 18.70 | 2.04 |
| Muse Spark 1.3 (original) | 15.8 | 8.55 | 3.62 |

## G Prompts

We use the model's native system prompt and provide the following user prompt at the beginning of each evaluation. The command names are identical across agentic systems.

**LeanLean prompt**

```
Refactor the proofs in this codebase.

Remove all dead code and do real simplification, i.e. use powerful tactics or factor out shared logic to
simplify the code while preserving the main results.
Success is measured by three factors:
```

```
(1) The repo compiling without errors and relying only on standard axioms.
(2) The benchmark's protected main results compile and remain equivalent. Run the no-argument command `
lean_verify` from the repository root to verify the signatures, exit code 0 means the main results are
preserved according to the lean comparator.
(3) The scoring metric is the reduction in non-whitespace, non-comment "Lean tokens" across all .lean files,
compared to the starting state. Compressing imports does not count towards the compression metric. Run
python3 proof_length.py to measure it.

You are working in a Docker container that is limited to 8 CPUs and 64 GiB of RAM.

You have a wall-clock budget of 12h. It started at {current_time} and expires at {deadline}.

When you believe there is no more refactoring to be done, output <submit> and terminate.
```

### Compress the proofs prompt ablation

```
Compress the proofs in this codebase.

Success is measured by three factors:

(1) The repo compiling without errors and relying only on standard axioms.
(2) The benchmark's protected main results compile and remain equivalent. Run the no-argument command `
lean_verify` from the repository root to verify the signatures, exit code 0 means the main results are
preserved according to the lean comparator.
(3) The scoring metric is the reduction in non-whitespace, non-comment "Lean tokens" across all .lean files,
compared to the starting state. Compressing imports does not count towards the compression metric. Run
python3 proof_length.py to measure it.

You are working in a Docker container that is limited to 8 CPUs and 64 GiB of RAM.

You have a wall-clock budget of 12h. It started at {current_time} and expires at {deadline}.

Don't cheat by hiding Lean code in non-.lean files, in the build artifacts, or anywhere else. Cheating will
be detected.

When you believe there is no more refactoring to be done, output <submit> and terminate.
```

### Reduce the size of this codebase prompt ablation

```
Reduce the size of this Lean codebase.

Success is measured by three factors:

(1) The repo compiling without errors and relying only on standard axioms.
(2) The benchmark's protected main results compile and remain equivalent. Run the no-argument command `
lean_verify` from the repository root to verify the signatures, exit code 0 means the main results are
preserved according to the lean comparator.
(3) The scoring metric is the reduction in non-whitespace, non-comment "Lean tokens" across all .lean files,
compared to the starting state. Compressing imports does not count towards the compression metric. Run
python3 proof_length.py to measure it.

You are working in a Docker container that is limited to 8 CPUs and 64 GiB of RAM.

You have a wall-clock budget of 12h. It started at {current_time} and expires at {deadline}.

When you believe there is no more refactoring to be done, output <submit> and terminate.
```

### Merging two compressions

```
We have two refactors of this Lean codebase, one on the main branch, and one on the other-refactor branch. To
 see the original codebase, refer to the initial commit on the main branch.

Combine ideas from both refactors into a smaller final refactor. Starting from the one on main, incorporate
ideas from the other-refactor branch to compress it even more.

Success is measured by three factors:

(1) The repo compiling from scratch without errors and relying only on standard axioms.
```

```
(2) The benchmark's protected main results compile and remain equivalent. Run the no-argument command `
lean_verify` from the repository root to verify the signatures, exit code 0 means the main results are
preserved according to the lean comparator.

(3) The scoring metric is the reduction in non-whitespace, non-comment "Lean tokens" across all .lean files,
compared to the starting state. Compressing imports does not count towards the compression metric. Run
python3 proof_length.py to measure it.

You have a wall-clock budget of 4h. It started at {current_time} and expires at {deadline}.

You are working in a Docker container with access to 64GiB of RAM and 8 CPUs.

Don't cheat or bypass the rules, any cheating will be detected. Such as putting Lean code in .txt, hiding
code inside the build artifacts. Just make the repo smaller than what is currently on the main branch.

When both refactors are combined and repo is smaller than the original in Lean tokens, output <submit> and
terminate.
```

## H REPOSITORY INDEX

We include a complete index of the 64 benchmark repositories in Table 10. Table 11 lists the 12-repository subset used for the Fable 5.1, GPT-6 Astra, and union experiments, together with the smaller subsets used for the Muse Spark 1.3 rerun and the Opus 5 prompt ablation.

Table 10: LEANLEAN repositories, brief descriptions, protected declaration counts, scale categories (Figure 3(c)), and linked Palomar IDs.

| Repository and description | Protected decls. | Scale | Palomar ID |
|---|---|---|---|
| `SauersML/group-approximation`<br>An explicit finitely presented sofic group that is not MF in the operator-norm sense. | 1 | Massive | `2026-08-24-000006` |
| `Arthur742Ramos/dinitz-garg-goemans-counterexample-lean`<br>A finite counterexample to the cost-preserving single-source unsplittable-flow conjecture. | 17 | Compact | `2026-08-29-000003` |
| `ehrlich-b/euclidean-jordan`<br>Symmetry, associativity, and positive definiteness of the Jordan trace form. | 4 | Standard | `2026-08-29-000001` |
| `teal-sea/zeta-lab`<br>A sharp variational identity for a Fredholm operator built from a zeta pair-correlation form factor. | 3 | Large | `2026-08-21-000004` |
| `teorth/sendov`<br>Sendov's conjecture and the Phelps–Rodriguez conjecture on extremal polynomials. | 2 | Large | `2026-08-13-000001` |
| `gexahedron/sabidussi-lean`<br>Sabidussi's compatibility conjecture. | 1 | Standard | `2026-08-17-000003` |
| `elliotglazer/erdos501`<br>Independence from ZFC and the closed-set case of Erdős problem 501. | 7 | Massive | `2026-08-19-000002` |
| `lennrt/palomar-formalizations`<br>The outer-matrix structure and characteristic polynomial for generalized cubic subdivision. | 5 | Compact | `2026-08-21-000010` |
| `BochaoKong/classical-complex-wpt`<br>The classical complex-analytic Weierstrass preparation theorem at the origin. | 1 | Standard | `2026-08-29-000010` |
| `tejstead/heilbronn-site`<br>Exact convex-hull-normalized Heilbronn triangle bounds for three to eight points. | 29 | Massive | `2026-09-02-000012` |
| `savarin/lean-malliavin`<br>Malliavin calculus on abstract Gaussian Banach spaces. | 1 | Large | `2026-08-31-000018` |
| `catskillsresearch/scott1982`<br>Scott's function-space information system and its characterization by approximable mappings. | 41 | Compact | `2026-08-24-000005` |
| `guanyangwang/ktv-swap-lean-comparator`<br>The spectral-gap theorem for the lazy swap chain on binary contingency tables with fixed margins. | 1 | Compact | `2026-08-20-000004` |
| `nielstron/langlib`<br>Grammar and automaton presentations of the extended Chomsky hierarchy. | 109 | Massive | `2026-08-27-000011` |
| `stalex444/werner-window`<br>Entanglement witnesses, CHSH bounds, and Werner-state correlations. | 10 | Standard | `2026-09-01-000003` |
| `BochaoKong/nullstellensatz`<br>Rückert's Noetherianity theorem and local analytic Nullstellensatz. | 4 | Large | `2026-08-30-000001` |

| Repository and description | Protected decls. | Scale | Palomar ID |
|---|---|---|---|
| `lennrt/palomar-formalizations`<br>A 16-vertex counterexample to prescribed-cycle recovery in Barnette graphs. | 6 | Compact | `2026-08-21-000002` |
| `JD-Jones-ASES/Hadamard-B-Formal`<br>Characterizations and constructions of bordered Goethals–Seidel Hadamard arrays. | 84 | Standard | `2026-09-01-000006` |
| `yuningyang19/parameter_free_gradient_p`<br>Exact value-gradient oracle complexity and parameter-free optimization in $\ell_p$ geometry. | 3 | Massive | `2026-08-28-000001` |
| `npflueger/brill-noether-graphs`<br>Brill–Noether generality for chains of cycles and theta graphs. | 7 | Large | `2026-09-05-000001` |
| `BrandonMYates/schwarzschild-comparator`<br>Einstein-tensor identities for classical spacetime metrics and a static-gauge Birkhoff theorem. | 54 | Compact | `2026-08-27-000020` |
| `shhyang/n4code_lean`<br>The Dong–Yang classification of optimal four-word binary block codes. | 8 | Massive | `2026-09-05-000003` |
| `pengzhang91/Feige`<br>The sharp unit-slack small-deviation bound for sums of independent nonnegative random variables. | 1 | Large | `2026-08-30-000002` |
| `lennrt/palomar-formalizations`<br>A counterexample to Suvagiya's signed-circulant minimum conjecture. | 9 | Standard | `2026-09-02-000008` |
| `novou/square-maximality-palomar`<br>Square maximality for spanning trees of induced square-lattice subgraphs. | 2 | Massive | `2026-08-23-000003` |
| `Arthur742Ramos/Metatheory`<br>Soundness of finite decreasing-diagram certificates for confluence. | 1 | Compact | `2026-09-01-000010` |
| `rwst/Confinement-Certificates`<br>Emptiness certificates for generalized $Z$-number sets. | 24 | Large | `2026-08-24-000002` |
| `catskillsresearch/scott1972`<br>Scott's inverse-limit continuous lattice, homeomorphic to its own function space. | 34 | Standard | `2026-08-24-000003` |
| `mccorvie/classification-of-surfaces`<br>Classification of compact connected topological surfaces, including surfaces with boundary. | 1 | Massive | `2026-08-29-000011` |
| `ukiyois/hs-virial-nbc-identity`<br>The no-broken-circuit volume identity and fork-packing bounds for hard-sphere cluster integrals. | 2 | Standard | `2026-08-29-000014` |
| `stalex444/golden-family-trace-forms`<br>Tensor-product trace forms and the signature product rule for quadratic forms. | 23 | Compact | `2026-09-02-000014` |
| `savarin/lean-spectral-theory`<br>Spectral and Stone theorems for possibly unbounded self-adjoint operators. | 2 | Large | `2026-09-01-000004` |
| `Paul-Lez/hadamard-668-comparator`<br>A Hadamard matrix of order 668. | 1 | Compact | `2026-08-17-000002` |
| `shaikidris/prime-star-spectra-formalization`<br>Uniform asymptotics for the ordered adjacency spectrum of finite prime-cover graphs. | 12 | Large | `2026-08-29-000017` |
| `sebastienpalcoux/sharp-intermediate-subgroup-bounds`<br>Sharp upper bounds on the number of intermediate subgroups of a finite group. | 9 | Standard | `2026-09-03-000005` |

| Repository and description | Protected decls. | Scale | Palomar ID |
|---|---|---|---|
| `WillWhistler/Regts-Sevenster`<br>The Regts–Sevenster characterization of graph parameters by mixed edge-colouring models. | 6 | Massive | `2026-08-19-000003` |
| `sneed-and-feed/lean-theorems-1`<br>Tucker's lemma in dimensions one and two, with a cyclic-boundary condition in dimension two. | 7 | Compact | `2026-09-03-000007` |
| `geoconuk/lean-misc-math`<br>Wyner's asymptotic covering bound for spheres by spherical caps. | 5 | Standard | `2026-08-23-000001` |
| `nimaanari/formalization-beyond-bethe`<br>A deterministic approximation of the permanent improving on the Bethe approximation factor. | 1 | Massive | `2026-08-29-000006` |
| `yoshito-ishiki-math/lean-scottish-book-155`<br>A counterexample to Scottish Book Problem 155 on local and global isometries of Banach spaces. | 1 | Large | `2026-08-31-000008` |
| `Arthur742Ramos/HigherLambdaModel`<br>Higher-dimensional structure in the untyped lambda calculus. | 2 | Compact | `2026-08-27-000015` |
| `ehrlich-b/sequential-products-lean`<br>Classification of sequential products on real and complex Hermitian matrix algebras. | 2 | Large | `2026-08-22-000006` |
| `stalex444/radial-harmonic-classification`<br>Classification of radial harmonic functions in finite-dimensional real inner product spaces. | 23 | Standard | `2026-09-03-000004` |
| `yuma-mizuno/markoff-modp-generalized`<br>Eventual strong approximation for the generic branch of generalized Markoff surfaces. | 1 | Massive | `2026-09-02-000001` |
| `dcposch/jc2-lean`<br>Exclusion of a specified polynomial-cube configuration at partial degrees $(6, 9)$. | 3 | Massive | `2026-08-29-000002` |
| `sloisel/MPE-Formal`<br>Probabilistic convergence theorems for restarted Minimal Polynomial Extrapolation. | 3 | Large | `2026-08-20-000009` |
| `stalex444/Busch-theorem`<br>Busch's finite-dimensional theorem representing effect probabilities by a unique density matrix. | 2 | Compact | `2026-08-31-000014` |
| `xiangyazi24/palomar-hahn-hecke-appell`<br>The Kronecker partial-fraction and boundary-cleared Hickerson–Mortenson Hecke–Appell identities. | 2 | Standard | `2026-08-22-000003` |
| `Arthur742Ramos/NagataFactoriality`<br>Nagata's factoriality theorem for Noetherian integral domains. | 3 | Compact | `2026-08-28-000002` |
| `stalex444/salem-theorem`<br>Salem's theorem: every Pisot number is a two-sided limit of Salem numbers. | 3 | Standard | `2026-09-02-000003` |
| `Arthur742Ramos/HowsonIntersectionTheorem`<br>Howson's theorem on intersections of finitely generated subgroups of finite-rank free groups. | 1 | Large | `2026-09-01-000009` |
| `CBirkbeck/uniform-sheafy-tate-domains-lean`<br>Uniform sheafy Tate rings that are not stably uniform. | 9 | Massive | `2026-08-22-000010` |
| `akopjan/NRR-AAK`<br>Partitions of planar convex figures into convex pieces of equal area and perimeter. | 1 | Massive | `2026-08-31-000005` |
| `Arthur742Ramos/equality-saturation-checker-lean`<br>A certified equality-saturation checker for finite acyclic e-graphs. | 8 | Standard | `2026-09-01-000007` |

| Repository and description | Protected decls. | Scale | Palomar ID |
|---|---|---|---|
| `langlerangle/NashEmbedding`<br>Nash's smooth isometric embedding theorem for closed manifolds. | 1 | Large | `2026-09-02-000004` |
| `ehrlich-b/composition-algebras`<br>Hurwitz's dimension theorem for Euclidean composition algebras. | 1 | Compact | `2026-08-29-000015` |
| `g-pichler/bsc-averaging`<br>Mutual-information bounds for binary channels on a doubly symmetric binary source. | 1 | Standard | `2026-08-26-000006` |
| `gaearon/conway-refinement`<br>Conway's refinement conjecture for omnific integers. | 1 | Massive | `2026-09-03-000002` |
| `CoolRmal/Besicovitchs-1-2`<br>A bound of $0.6934$ for Besicovitch's $1/2$-problem. | 3 | Large | `2026-09-02-000011` |
| `stalex444/only-two-morphic-numbers`<br>Classification of morphic numbers as the golden ratio and the plastic number. | 4 | Compact | `2026-09-01-000008` |
| `SamPetkov/Erdos625-formalization`<br>A quantitative chromatic–cochromatic gap for random graphs, resolving Erdős problem 625. | 3 | Massive | `2026-09-02-000006` |
| `benkeene/erdos266`<br>The Kovač–Tao resolution of Erdős problem 266 on shifted Ahmes series. | 2 | Standard | `2026-08-20-000008` |
| `dsm054/kasami_cyclic_additive`<br>Carlet's Kasami cyclic-additive conjecture. | 1 | Large | `2026-09-04-000005` |
| `roos-j/lean-booleanfun`<br>Arrow's theorem for three-candidate elections via Fourier analysis. | 1 | Compact | `2026-09-01-000011` |

Table 11: Repositories used in the subset experiments. The 12-repository subset contains three repositories from each scale category and is used for Fable 5.1 and GPT-6 Astra (Table 2) and the union ablation (Table 4). The Muse Spark 1.3 rerun (Appendix F) and the Opus 5 prompt ablation (§4.2.2) use subsets of these 12.

| Repository | Scale | Palomar ID | Fable/Astra & union | Muse rerun | Prompt ablation |
|---|---|---|---|---|---|
| `hadamard-668-comparator` | Compact | `2026-08-17-000002` | ✓ | ✓ | |
| `scott1982` | Compact | `2026-08-24-000005` | ✓ | ✓ | |
| `dinitz-garg-goemans-counterexample-lean` | Compact | `2026-08-29-000003` | ✓ | ✓ | ✓ |
| `sabidussi-lean` | Standard | `2026-08-17-000003` | ✓ | ✓ | |
| `palomar-hahn-hecke-appell` | Standard | `2026-08-22-000003` | ✓ | | ✓ |
| `bsc-averaging` | Standard | `2026-08-26-000006` | ✓ | ✓ | |
| `zeta-lab` | Large | `2026-08-21-000004` | ✓ | ✓ | |
| `sequential-products-lean` | Large | `2026-08-22-000006` | ✓ | | |
| `Confinement-Certificates` | Large | `2026-08-24-000002` | ✓ | ✓ | ✓ |
| `Regts-Sevenster` | Massive | `2026-08-19-000003` | ✓ | | |
| `square-maximality-palomar` | Massive | `2026-08-23-000003` | ✓ | | |
| `langlib` | Massive | `2026-08-27-000011` | ✓ | ✓ | ✓ |
| Total | | | 12 | 8 | 4 |