CAM-BENCH: A BENCHMARK FOR COMPUTATIONAL AND APPLIED MATHEMATICS IN LEAN

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal theorem-proving benchmarks enable mechanically verifiable evaluation of mathematical reasoning in large language models. However, existing benchmarks mainly focus on Olympiad-style problems and algebraic domains, leaving computational and applied mathematics underrepresented. We introduce CAM-Bench, a Lean 4 theorem-proving benchmark of 500 designated `sorry` proof obligations. These problems cover optimization, numerical linear algebra, numerical analysis, and closely related topics in applied mathematics, and depend on locally introduced definitions, notation, algorithms, and elementary results. To construct CAM-Bench, we develop a dependency-recovery pipeline that reconstructs the local textbook context needed to state each problem faithfully. It then normalizes each problem into a standalone informal theorem and translates it into a Lean target. We validate the resulting formal problems through Lean compilation and semantic review, checking Lean well-formedness and semantic alignment with the original exercises. For each problem, we release the raw exercise, recovered context, normalized informal theorem, and final Lean target. CAM-Bench complements existing formal mathematics benchmarks by targeting applied mathematics problems that rely on textbook concepts and elementary theorems, many of which are not directly available as standard Mathlib4 lemmas. We evaluate large language models and formalization agents on CAM-Bench and analyze common failure modes in tracking local assumptions, applying elementary results, decomposing proofs, and maintaining long-horizon control in Lean.

1 INTRODUCTION

Mathematical reasoning is a core capability for large language models (LLMs) and a central benchmark for reliable multi-step inference. Recent advances in reasoning models, proof search systems, and reinforcement learning with verifiable feedback have produced strong results on both informal mathematical benchmarks and formal theorem-proving tasks Hendrycks et al. (2021); He et al. (2024); Guo et al. (2025); Comanici et al. (2025); Chen et al. (2025); Ren et al. (2025). Despite this progress, informal evaluation gives only a partial account of mathematical competence. When the final answer is insufficient to validate the derivation, it is difficult to determine whether a model has actually followed a correct line of reasoning. Formal proof assistants such as Lean Moura & Ullrich (2021), Isabelle Paulson (1994), Coq Huet et al. (1997), and HOL Harrison (1996) address this limitation by checking statements and proofs against a trusted kernel. They therefore provide a rigorous way to evaluate whether model-generated mathematics is not only plausible, but mechanically verifiable.

Existing formal reasoning benchmarks have been essential for measuring progress, yet they cover only part of the ability needed for formalization. MiniF2F focuses on Olympiad-style statements across theorem-proving systems Zheng et al. (2021). ProofNet studies autoformalization for undergraduate mathematics Azerbayev et al. (2023). PutnamBench targets competition-level problems Tsoukalas et al. (2024). FormalMATH broadens Lean 4 coverage through a large autoformalized corpus Yu et al. (2025). More recent evaluations further show that success on curated formal benchmarks does not necessarily transfer to research-level formalization projects or program verification tasks Poiroux et al. (2025); Barkallah et al. (2025). These developments motivate benchmarks that test a different capability. Existing benchmarks mainly focus on Olympiad-style problems, alge-

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.

braic domains, and isolated theorem statements, leaving computational and applied mathematics underrepresented.

This gap is important since many problems in computational and applied mathematics are not naturally stated as self-contained theorem statements. Textbook exercises in optimization, numerical linear algebra, and numerical analysis often depend on locally introduced definitions, notation, algorithms, chapter conventions, and elementary results. These local dependencies may determine not only the meaning of the problem, but also the assumptions and proof structure needed for a faithful formal statement. A benchmark in this setting therefore needs to evaluate whether models can track local assumptions, perform proof decomposition, and produce valid formal proofs.

We introduce CAM-Bench, a Lean 4 theorem-proving benchmark of 500 designated `sorry` proof obligations, with coverage of optimization, numerical linear algebra, numerical analysis, and related applied mathematics. The counting unit is the proof placeholder. Each obligation is accompanied by the surrounding declarations and recovered local context needed to state the exercise faithfully. Each task asks a model to construct a proof for a fixed Lean statement under its associated local context. This design complements existing formal mathematics benchmarks by targeting applied mathematics problems that rely on textbook concepts and elementary theorems, many of which are not directly available as standard Mathlib4 lemmas.

To construct CAM-Bench, we develop a dependency recovery pipeline for context-dependent textbook exercises. For each exercise, the pipeline reconstructs the surrounding mathematical context, including relevant definitions, notation, algorithms, assumptions, and elementary results. It then normalizes the exercise into a standalone informal theorem and translates it into a Lean 4 target. Compilation feedback and semantic review are used to check Lean well-formedness and semantic alignment with the original exercise. For each problem, we release the raw exercise, recovered context, normalized informal theorem, and final Lean target, together forming a transparent record of how the textbook exercise is converted into a proof task.

We evaluate widely used large language models and formalization agents on CAM-Bench. The resulting error analysis identifies failures in tracking local assumptions, applying elementary results, decomposing proofs, and maintaining long-horizon control in Lean, suggesting that context-dependent applied mathematics remains challenging even when the target statement is fixed and mechanically checkable. Our main contributions are as follows.

- We introduce CAM-Bench, a Lean 4 theorem-proving benchmark containing 500 designated proof obligations, with traceable artifacts from the raw exercise to the final machine-checked Lean target.
- We design a dependency recovery and formalization pipeline for constructing faithful Lean targets from context-dependent textbook exercises. The pipeline recovers local mathematical context, decomposes heterogeneous exercises into semantic blocks, and uses compilation feedback, semantic review, and constrained repair to validate correctness and alignment.
- We evaluate representative LLM proving strategies on CAM-Bench, including direct proof generation, iterative repair, and agent-based proving. We further analyze their failures in formal representation, missing mathematical infrastructure, proof-artifact fidelity, and long-horizon proof control.

2 Preliminaries and Related Work

Formal benchmarks. Formal benchmarks provide targets for measuring mathematical reasoning under proof assistant verification. MiniF2F established a cross-system testbed for Olympiad-style formal mathematics, and PutnamBench extends this line of evaluation to university-level competition problems Zheng et al. (2021); Tsoukalas et al. (2024). ProofNet focuses on undergraduate mathematics by pairing informal problems with formal statements, making it an important resource for studying autoformalization Azerbayev et al. (2023). Other benchmarks target specialized mathematical domains, including challenging informal to formal translation, combinatorial identities, and algebraic reasoning across multiple difficulty levels Liu et al. (2023; 2025); Jiang et al. (2025). Large Lean corpora such as FormalMATH broaden coverage across mathematical areas and support more

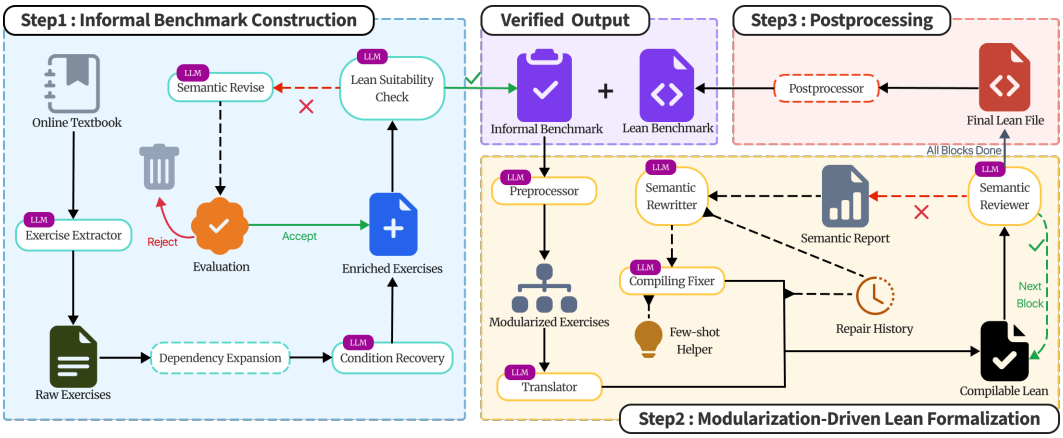


Figure 1: Pipeline Overview

systematic evaluation of formal reasoning models Yu et al. (2025). Recent evaluations also move beyond isolated theorem statements toward more realistic settings, including research-level Lean blueprint projects and Lean 4 code verification tasks Poiroux et al. (2025); Barkallah et al. (2025).

Autoformalization for benchmark construction. Autoformalization translates informal mathematical language into formal statements and proofs, and large language models have made this direction increasingly practical Wu et al. (2022). Recent methods use retrieval, compiler feedback, process supervision, structured prompting, and iterative reflection to improve statement translation and proof generation Weng et al. (2025); Lu et al. (2024); Lin et al. (2024); Zhou et al. (2025). For benchmark construction, however, syntactic validity is only one requirement. A Lean statement may compile while omitting an assumption, weakening the conclusion, changing a quantifier, or replacing the original exercise with a nearby theorem that is easier to prove. Recent robustness studies further show that semantically similar paraphrases of the same informal statement can lead to different formalizations Moore & Shah (2025).

Proof assistants and neural theorem proving. Systems such as Isabelle, Coq, HOL Light, and Lean have long supported large-scale formal mathematics and program verification Paulson (1994); Huet et al. (1997); Harrison (1996); De Moura et al. (2015); Moura & Ullrich (2021). Recent neural theorem provers build on these environments by combining language model generation with search, retrieval, compiler feedback, and iterative repair. Early systems showed that generative models can guide proof search in interactive theorem proving Polu & Sutskever (2020); Lample et al. (2022). Subsequent Lean-based provers make increasing use of retrieved premises, environment interaction, synthetic data, self-correction, and reinforcement learning Yang et al. (2023); Xin et al. (2024); Ren et al. (2025); Wu et al. (2025); Li et al. (2024); Xin et al. (2025); Lin et al. (2025); Wang et al. (2025); Chen et al. (2025); Li et al. (2026). Agent- and project-scale systems further emphasize long-horizon coordination, tool use, and end-to-end formal development Achim et al. (2025); Wang et al. (2026). These systems have substantially improved formal proof generation.

3 Main Pipeline for Benchmark Construction

The overall pipeline for constructing a paired informal–formal Lean benchmark is illustrated in Figure 1. The system is organized into three major steps: **(1) informal benchmark construction**, **(2) modularization-driven Lean formalization**, and **(3) postprocessing**. The first step converts raw textbook exercises into dependency-aware, self-contained informal statements. The second step decomposes each reviewed statement into semantically typed blocks and incrementally translates them into Lean through compiler-guided and semantic repair. The final step normalizes and packages the accepted Lean outputs into the final benchmark.

Algorithm 1: Recursive Dependency Expansion and Enrichment

Input: Exercise x, corpus $\mathcal{C}$, depth budget D
Output: Enriched exercise $\widetilde{x}$

$\mathcal{T} \leftarrow \textsc{Expand}(\textsc{NewNode}(x), \mathcal{C}, D)$;
$\widehat{x} \leftarrow \textsc{Naturalize}(\mathcal{T}.root)$;
return $\textsc{RecoverConditions}(\widehat{x}, \mathcal{T})$;
Function `Expand` $(v, \mathcal{C}, d)$ **:**
  if $d = 0$ **then**
    return v;
  foreach $\rho \in \textsc{ExtractRefs}(\textsc{Text}(v))$ **do**
    $u \leftarrow \textsc{RetrieveNode}(\rho, \mathcal{C})$;
    if $u \neq \varnothing$ *and* $\rho \notin \textsc{Path}(v)$ **then**
      $\textsc{AddChild}(v, \textsc{Expand}(u, \mathcal{C}, d-1))$;
  return v;
Function `Naturalize` (v) **:**
  $\mathcal{M} \leftarrow \{\textsc{Naturalize}(u) : u \in \textsc{Children}(v)\}$;
  $z \leftarrow \textsc{Insert}(\textsc{Text}(v), \mathcal{M})$;
  return LLM-$\textsc{Naturalize}(z, \textsc{IsRoot}(v))$;

3.1 Informal Benchmark Construction

Exercise extraction. The pipeline begins by extracting raw exercise statements from online textbooks. These raw exercises preserve the surface form of the original problems, but they are often not directly suitable for formalization. They may refer to previous theorems, rely on chapter-level notation, or omit inherited assumptions. Thus, the goal of this stage is not merely to extract exercises, but to transform them into self-contained formalization targets.

Recursive dependency expansion and enrichment. A central challenge in textbook-based benchmark construction is that exercise statements are rarely self-contained: a cited theorem may itself depend on earlier definitions, lemmas, equations, algorithms, or notation conventions, so resolving only explicitly mentioned references is insufficient for a Lean-suitable target.

To address this issue, we introduce a recursive dependency expansion and enrichment procedure, shown in Algorithm 1. Starting from a raw exercise x, the procedure builds a bounded dependency tree $\mathcal{T}$ by repeatedly extracting references, retrieving the corresponding corpus items, and expanding their internal references up to depth budget D. Each dependency node records its source text, reference label, hop depth, and provenance information, so each inserted item can be traced back to the dependency that contributed it. Cyclic expansion is avoided by ignoring references already on the expansion path.

The tree is then processed bottom-up: each node merges the enriched representations of its children, and an LLM naturalizer rewrites the result to resolve local references, standardize notation, smooth transitions, and remove duplicated material, ensuring lower-level definitions are incorporated before they are used by higher-level dependencies.

At the root, a broader cleanup removes textbook-specific phrasing and pedagogical hints while preserving the mathematical objective. An implicit-condition recovery step then makes required assumptions explicit—such as domain restrictions, positivity, dimensional compatibility, and convexity or regularity conditions—but only when supported by the original problem, recovered context, or well-formedness requirements of objects already present in the statement. This enrichment is source-grounded rather than free-form: inserted definitions, equations, notation, and assumptions must be traceable to the source exercise or recovered dependencies; unresolved dependencies are held out rather than completed by conjecture. The output $\widetilde{x}$ is a declarative, self-contained informal target that preserves the original assumptions, quantifiers, domains, and conclusion. Additional details on statement cleaning, recursive dependency recovery, source-grounded enrichment, and implicit-condition completion are provided in Appendix A.1–A.4.

Multi-round review and constrained repair. Since dependency enrichment is LLM-assisted, the resulting informal statements are passed through a validation loop before formalization. Each en-

riched exercise is reviewed against the original source problem for missing assumptions, task drift, missing dependencies, unresolved definitions, and Lean suitability. The instance is then classified as **accept**, **revise**, or **hold**.

Repair is used only for local, review-identified defects: it may normalize notation, inline already recovered dependencies, restore source-supported assumptions, or convert question-style prompts into theorem-style statements. It may not introduce unsupported assumptions, invent missing dependencies, weaken or strengthen the claim, or change the proof objective. Every repaired item is re-reviewed before proceeding to Lean formalization; cases with unrecovered dependencies are held out rather than completed by conjecture. This loop prevents dependency enrichment from becoming unconstrained rewriting and keeps the informal benchmark faithful to the original textbook intent. See Appendix B.1 for the review criteria and decision labels, and Appendix B.2 for the constrained repair and re-review protocol.

3.2 Modularization-Driven Lean Formalization

Semantic block decomposition. Textbook exercises are often heterogeneous: a single problem may combine definitions, optimization objectives, algorithms, intermediate claims, hints, and theorem targets. Directly translating such mixed content into one Lean fragment can cause role confusion and makes errors difficult to localize.

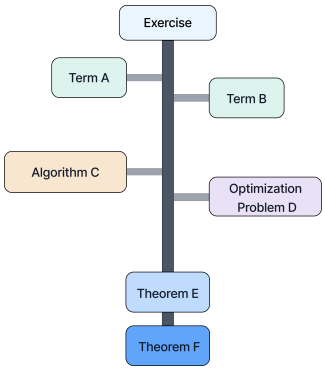


Figure 2: Semantic block tree.

We therefore first convert each enriched exercise into a structured collection of semantic blocks before Lean generation. Each block is assigned a functional role according to how it should be used in the formalization process. For example, algorithmic descriptions and update rules are separated from optimization formulations; formalizable concepts and assumptions are separated from the final theorem target; and informal hints are retained as guidance but excluded from the generated Lean declarations. This decomposition turns a heterogeneous exercise into a theorem-centered dependency structure. The target theorem serves as the root, while the remaining blocks serve as branches that provide supporting definitions, assumptions, constructions, or contextual information, as illustrated in Figure 2. By making the semantic role of each component explicit before Lean generation, later stages can focus on the relevant block rather than repeatedly rewriting the entire formalization.

Frozen-context block-wise construction. Lean formalization proceeds incrementally over the modularized blocks rather than by repeatedly regenerating the full Lean file.

At each step, the system focuses on a single target block, while previously accepted declarations, assumptions, notation, and auxiliary definitions are kept fixed as read-only context.

This frozen-context protocol yields a stable, monotonically growing prefix: local edits cannot corrupt earlier blocks, cascading errors are reduced, and failures can usually be localized to the current block or its interface with the frozen prefix.

Translation, compilation repair, and semantic rewriting. For each target block, the translator generates Lean code according to the block type, using all previously accepted blocks as read-only context. The generated Lean code is then checked by the Lean compiler.

If the current block fails to compile, a compiler-guided fixer produces a local patch using the compiler diagnostics and the frozen context of previously accepted blocks. Typical repairs include correcting syntax, adding missing type annotations, resolving namespace issues, or adjusting declarations to satisfy Lean's type system. This compilation repair loop continues until the block becomes compilable or the repair budget is exhausted.

Every compilable candidate is then passed to a semantic reviewer. The reviewer compares the Lean candidate with the corresponding block and checks whether the intended meaning is faithfully preserved. If a mismatch is detected, the candidate is sent to a semantic rewriter, which produces a local semantic patch for the current block. The patched code is then returned to the compilation repair

loop to ensure that the semantic correction remains Lean-valid. Each block is accepted only after it compiles and passes semantic review against the informal statement.

History-aware repair and rejection control. Because iterative repair can oscillate, the system maintains a bounded history for each block, including recent attempts, compiler error signatures, failed patches, semantic review outcomes, and rejection reasons. At each iteration, this history is provided together with the current feedback, making repair path-aware and discouraging previously failed edit directions. In this way, failed attempts are not discarded, but are converted into negative constraints that guide subsequent repairs.

The pipeline also rejects rewrites that introduce semantic degradation, such as strengthening assumptions, weakening conclusions, dropping quantifiers, altering variable dependencies, changing notation in a meaning-changing way, or modifying optimization objectives. This check is intended to prevent Lean-valid patches from making the target easier by changing its mathematical content. Together, history-aware repair and rejection control encourage monotonic progress toward blocks that are both compilable and faithful to the original textbook semantics.

Postprocessing. The final stage performs lightweight normalization and packaging. Accepted Lean outputs are cleaned, organized, and assembled into final benchmark files for downstream evaluation. Additional details on semantic block decomposition, frozen-context construction, compiler-guided repair, semantic rewriting, history-aware rejection control, and postprocessing are provided in Appendix C.1 –C.4.

4 Benchmark Analysis

4.1 Overall Dataset Statistics

CAM-Bench contains 500 designated `sorry` proof obligations. The counting unit is the proof placeholder. Each obligation retains the surrounding declarations and recovered local context required for verification.

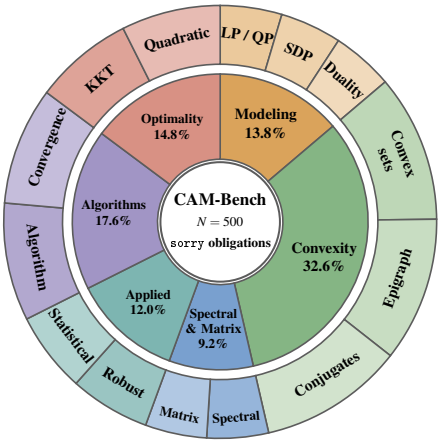


Figure 3: Primary reasoning-demand distribution among the 500 designated `sorry` obligations.

We organize the suite by the primary mathematical demand of a target rather than by textbook chapter alone. The taxonomy in Figure 3 uses six primary labels whose mathematical content may overlap: optimization modeling and duality (including LP/QP, conic and semidefinite reformulations); convexity and geometric inequalities (convex sets and functions, epigraphs, perspectives, conjugates, and support functions); spectral and matrix reasoning (positive-semidefinite structure, rank, pseudoinverses, trace/log-determinant identities, and eigenvalue inequalities); applied modeling (robust, statistical, inverse, network, and resource allocation formulations); algorithms and convergence (iterative schemes, line-search arguments, and limiting behavior); and optimality conditions and constrained quadratic reasoning (KKT conditions, stationarity, and quadratic certificates). Each target is assigned exactly one primary label for counting, while the prose records the mathematical connections that frequently cross those labels.

The resulting counts are 69 modeling, 163 convexity, 46 spectral or matrix, 60 applied modeling, 88 algorithms and convergence, and 74 optimality or constrained-quadratic obligations. These counts sum to the complete 500-target release; the outer ring expands the same labels into representative subtopics.

This organization shows why the suite tests more than isolated algebraic rewriting. A typical target combines a structural statement about an objective, constraint, matrix, or algorithm with local definitions and inherited assumptions from its source exercise. The resulting proof may require moving

between scalar inequalities, matrix identities, optimization semantics, and elementary auxiliary facts that are not directly available as standard Mathlib4 lemmas.

4.2 Preservation and Semantic Validation

We assess construction quality at two levels. Source-level curation asks whether the reviewed informal target remains faithful to the textbook exercise and self-contained after dependency recovery and repair. Lean-level semantic validation then asks whether the final Lean declaration preserves that reviewed target after formal translation. The quantitative statistics below are derived from recorded semantic-review logs produced by an independent LLM-as-a-judge reviewer and serve as diagnostic signals for filtering and repair, rather than as the sole correctness criterion. At each semantic-review stage, every problem that enters the stage is assigned a structured review log recording the resulting status together with the supporting reasons; representative source-level and Lean-level logs are shown in Appendix F.8 and Appendix F.7. We retain these logs as diagnostic construction records, and include representative examples in the appendix to make the source-to-Lean decision process more auditable.

Source-level review. At the source level, we use the funnel-style process in Figure 4 to filter and recover candidate exercises. In the first semantic review round, each candidate is assigned to one of three outcomes: *accept*, *revise*, or *hold*. As shown in the figure, 26.5% of candidates are accepted immediately, while 13.6% are marked as *revise* and 59.9% as *hold*. Repair-eligible candidates from the revise branch and non-dependency hold branch then undergo targeted repair and re-evaluation. This step recovers additional Lean-ready informal targets, increasing the source-level usable yield from 26.5% to 54.3% before Lean formalization. These percentages describe the broader candidate pool used during construction; they are not estimates of the 500-obligation release.

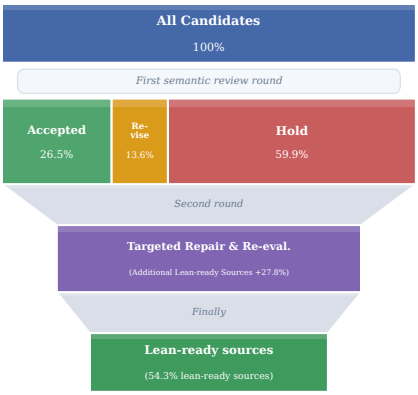


Figure 4: Source Review Funnel.

Candidates that remain unsatisfactory after this source-level review are excluded from the Lean-ready source pool. This conservative funnel is designed to preserve faithfulness to the original source exercise, self-containment, dependency completeness, and suitability as Lean theorem targets.

Lean-level validation. After formal translation, we perform a separate semantic validation stage for the generated Lean declarations. Compilation with `sorry` checks syntactic and type-theoretic well-formedness, but does not by itself ensure semantic fidelity. We therefore compare each candidate Lean declaration against its corresponding reviewed informal target for variables, assumptions, quantifiers, objectives, constraints, and conclusions. Under the final per-declaration review result, 92.5% of candidate declarations are classified as usable, while 7.5% are classified as not usable. These percentages likewise describe the candidate declaration pool at this stage, not the composition of the final released benchmark.

The final 500-obligation release consists of targets that pass source-level review for faithfulness, self-containment, dependency completeness, and Lean suitability, compile as fixed Lean declarations with the designated `sorry`, and pass the final Lean-level semantic review. Candidates that fail any of these criteria are held out. The six primary labels are then assigned to the retained targets, with exactly one label used for counting.

In this sense, the released benchmark is a conservatively retained subset of the raw pipeline output rather than an automatic dump of all generated formalizations. We report source-level validation details in Appendix B and Lean-level declaration validation details in Appendix C.3.

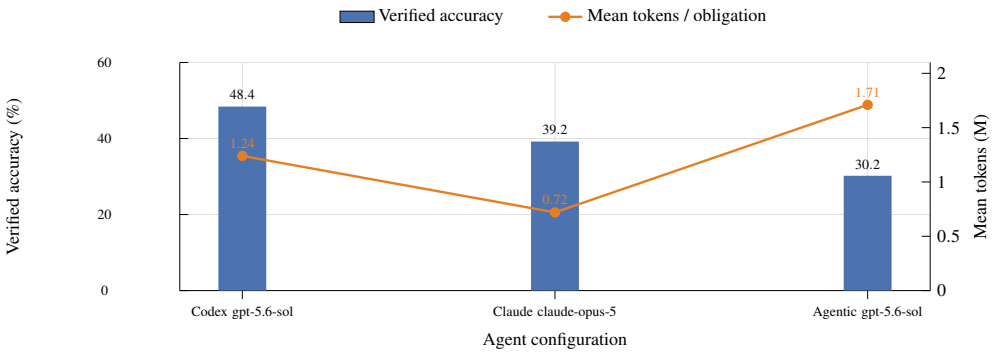


Figure 5: Model performance and token efficiency on CAM-Bench. Verified Lean accuracy is shown as a percentage on the left y-axis, while mean token usage per evaluated benchmark obligation is shown in millions of tokens (M) on the right y-axis. All three configurations use a cumulative 2M-token limit, and an obligation counts as verified only when its reconstructed Lean target passes the final kernel and integrity checks. The figure compares the trade-off between verified proof success and token expenditure across Codex `gpt-5.6-sol`, Claude `claude-opus-5`, and agentic `gpt-5.6-sol`. The agentic setting is described in Appendix D.5.

5 Experiments

We evaluate three primary configurations on all 500 proof obligations under a cumulative budget of 2M tokens per obligation. We additionally evaluate two external theorem-proving systems on a uniformly sampled subset of 200 obligations under an 8-hour wall-clock budget per obligation. The Codex and Claude configurations use independent pass@32 candidate generation with local checking, whereas the agentic configuration performs multiturn proof search with compiler-guided tool interaction.

5.1 Benchmark Performance

Full-suite performance. Figure 5 reports the full 500-obligation evaluation. Codex `gpt-5.6-sol` obtains the highest verified rate at 48.4%, followed by `claude-opus-5` at 39.2% and the agentic `gpt-5.6-sol` configuration at 30.2%. Codex exceeds the agentic configuration by 18.2 percentage points and Claude by 9.2 percentage points.

Codex combines the highest verified rate with a mean of 1.24M tokens per obligation. Claude uses 0.72M tokens on average, while the agentic configuration uses 1.71M tokens per obligation because it spends additional turns on compiler diagnostics, intermediate proof states, and repair decisions. The results show that token expenditure alone does not determine success: library grounding, local assumption tracking, and proof decomposition remain decisive.

5.2 Comparison on Agentic Proof Systems

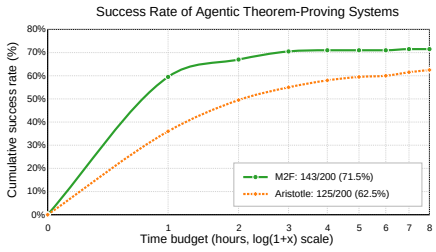


Figure 6: Cumulative success rate over time on the 200-instance sample.

We evaluate two external agentic theorem-proving systems, Aristotle Achim et al. (2025) and M2F Wang et al. (2026), on 200 benchmark instances sampled uniformly at random from the released benchmark. The same instance set, 8-hour wall-clock budget, and local Lean validation script are used for both systems to ensure a fair comparison. Figure 6 shows the cumulative proportion of solved instances under our strict validation protocol within the 8-hour budget. The horizontal axis uses a pseudo-$\log(1+x)$ transform to show both the early-time behavior and the long tail. Unlike pass@k decoding, these systems perform long-horizon proof search with Lean compiler feedback and may construct auxiliary lemmas during the solving process. We therefore

adopt a strict single-problem external validation protocol. An instance is counted as solved only if it is completed within 8 hours, the returned Lean project recompiles locally, preserves the original theorem statements and existing definitions, and contains no pseudo-proof artifacts such as `sorry`, `sorryAx`, `admit`, `axiom`, or unresolved `exact?`.

Agent performance. Under this protocol, M2F solves 143/200 sampled CAM-Bench instances (71.5%), while Aristotle solves 125/200 (62.5%). These results show that a substantial fraction of the sampled obligations can be solved by existing theorem-proving systems, while 57 and 75 obligations remain unsolved by M2F and Aristotle, respectively. The remaining failures often involve missing mathematical infrastructure, long auxiliary-lemma chains, or optimization-specific formal reasoning.

5.3 Error Analysis

We analyze failed outcomes from the Claude Code and Codex evaluation and the external agentic systems reported in Table 1. For the Claude Code and Codex column, one observation is a failed task file at the terminal runner state; repeated attempts within that task are not counted separately. For M2F and Aristotle, one observation is an unsolved sampled obligation. We inspect the final artifact associated with each failed observation. The runner emits operational labels, which we map to the three coarse categories below: infrastructure errors map to library/API grounding, boundary or integrity violations map to formal representation, and unverified proofs map to proof decomposition. These labels are diagnostic proxies for the Claude Code and Codex run, while the M2F and Aristotle entries retain the semantic multi-label review used for their failed obligations. Detailed definitions and fine-grained error statistics are provided in Appendix E.1, with representative examples in Appendix F.10. At a coarse level, the categories cover library/API grounding and infrastructure, formal representation and type discipline, and proof decomposition with long-horizon control; definitions and examples are given in Appendix E.1.

Table 1: Coarse failure attribution for the Claude Code and Codex evaluation and for unsolved M2F and Aristotle obligations. The Claude Code and Codex percentages are terminal runner-status proxies; the external-system labels are multi-label and therefore do not sum to 100%.

Failure attribution	Claude Code & Codex	M2F (N=57)	Aristotle (N=75)
Library/API grounding and infrastructure	9.1%	31/57 (54.4%)	38/75 (50.7%)
Formal representation and type discipline	58.4%	26/57 (45.6%)	29/75 (38.7%)
Proof decomposition and long-horizon control	32.5%	39/57 (68.4%)	52/75 (69.3%)

The first column reports percentages over failed terminal task files, with boundary and integrity statuses combined under formal representation. The M2F and Aristotle columns use multi-label review over their unsolved obligations. Overall, the failures are rarely isolated syntax errors. The results indicate that CAM-Bench evaluates not only informal mathematical reasoning, but also the ability to ground applied-mathematics arguments in the actual Lean environment and its available formal infrastructure.

6 Conclusion

We introduce CAM-Bench, a Lean 4 theorem-proving benchmark for computational and applied mathematics, constructed from textbook problems in optimization, numerical linear algebra, and numerical analysis. We develop a dependency-recovery and formalization pipeline that turns context-dependent exercises into self-contained Lean targets while preserving source traceability. CAM-Bench remains challenging for current LLM-based theorem provers: direct and agentic configurations expose complementary limitations, while external theorem-proving systems solve only part of the suite under strict validation. These results highlight persistent bottlenecks in Mathlib grounding, Lean type discipline, and long-horizon proof decomposition, and suggest that applied mathematics is a valuable testbed for evaluating formal proof systems beyond isolated theorem statements.

AI USE STATEMENT

We used large language models to assist with benchmark construction, including cleaning and reformulating textbook exercises, integrating source-grounded context, decomposing statements into semantic blocks, translating them into Lean 4, and performing compiler-guided repair and semantic review (Section 3 and Appendices A–C). We also used language models and agentic theorem-proving systems to generate and refine Lean proof attempts in the reported experiments (Appendix D). Generated benchmark artifacts were checked using the source-alignment reviews and Lean compilation procedures described in the paper; Lean proof successes were determined by compilation and integrity checks rather than by an LLM judge. We additionally used generative AI for manuscript editing and LaTeX formatting. The authors take responsibility for the final text, mathematical claims, benchmark artifacts, and reported results, including all AI-assisted content.

REFERENCES

Tudor Achim, Mingda Sun, Qingxiang Guo, Krunal Guliani, Kush Bhatia, Vineet Kumar, Mehran Shakerinava, Haowei Hu, Rohan Daniel, Stella Mai, et al. Aristotle: A recipe for math lms. *arXiv preprint arXiv:2510.01346*, 2025.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Slim Barkallah, Srivatsava Daruru, Brando Miranda, Leni Aniva, Allen Nie, and Sanmi Koyejo. Veribench-ftp: A formal theorem proving benchmark in lean 4 for code verification. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*, 2025.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025.

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

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pp. 378–388. Springer, 2015.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

John Harrison. Hol light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pp. 265–269. Springer, 1996.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 178:113, 1997.

Jiedong Jiang, Wanyi He, Yuefeng Wang, Guoxiong Gao, Yongle Hu, Jingting Wang, Nailing Guan, Peihao Wu, Chunbo Dai, Liang Xiao, et al. Fate: A formal benchmark series for frontier algebra of multiple difficulty levels. *arXiv preprint arXiv:2511.02872*, 2025.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

Chenyi Li, Yanchen Nie, Zhenyu Ming, Gong Zhang, Kun Yuan, and Zaiwen Wen. Optprover: Bridging olympiad and optimization through continual training in formal theorem proving. *arXiv preprint arXiv:2604.23712*, 2026.

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

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

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

Chengwu Liu, Jianhao Shen, Huajian Xin, Zhengying Liu, Ye Yuan, Haiming Wang, Wei Ju, Chuanyang Zheng, Yichun Yin, Lin Li, et al. Fimo: A challenge formal dataset for automated theorem proving. *arXiv preprint arXiv:2309.04295*, 2023.

Junqi Liu, Xiaohan Lin, Jonas Bayer, Yael Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, et al. Combibench: Benchmarking llm capability for combinatorial mathematics. *arXiv preprint arXiv:2505.03171*, 2025.

Jianqiao Lu, Yingjia Wan, Zhengying Liu, Yinya Huang, Jing Xiong, Chengwu Liu, Jianhao Shen, Hui Jin, Jipeng Zhang, Haiming Wang, Zhicheng Yang, Jing Tang, and Zhijiang Guo. Process-driven autoformalization in lean 4. *arXiv preprint arXiv:2406.01940*, 2024.

Hayden Moore and Asfahan Shah. Evaluating autoformalization robustness via semantically similar paraphrasing. *arXiv preprint arXiv:2511.12784*, 2025.

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

Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

Auguste Poiroux, Antoine Bosselut, and Viktor Kunčak. RLMEval: Evaluating research-level neural theorem proving. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 10946–10957. Association for Computational Linguistics, 2025.

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

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

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

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *Advances in Neural Information Processing Systems*, 37: 11545–11569, 2024.

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

Yuefeng Wang, Jiedong Jiang, Wanyi He, Guoxiong Gao, Chenyi Li, Yongle Hu, Mengxiao Sun, Jin Tan, Jingting Wang, Liang Xiao, et al. M2f: Proof automation with lean from autoformalization to verified complete proofs. *arXiv preprint arXiv:2602.17016*, 2026.

Ke Weng, Lun Du, Sirui Li, Wangyue Lu, Haozhe Sun, Hengyu Liu, and Tiancheng Zhang. Autoformalization in the era of large language models: A survey. *arXiv preprint arXiv:2505.23486*, 2025.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. Internlm2. 5-stepprover: Advancing automated theorem proving via critic-guided search. In *2nd AI for Math Workshop@ ICML 2025*, 2025.

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

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32588–32599, 2025.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, et al. Formalmath: Benchmarking formal mathematical reasoning of large language models. *arXiv preprint arXiv:2505.02735*, 2025.

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

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025.

A Details of Informal Benchmark Construction

This appendix supplements Section 3.1 with implementation details for the natural-language construction stage. While the main text describes the overall extraction, recursive dependency expansion, and review pipeline, this appendix specifies the intermediate records used to make the process traceable and auditable.

The goal of this stage is to convert a context-dependent textbook exercise into a self-contained informal mathematical target before Lean code is generated. The input is a raw exercise statement extracted from an online textbook. The output is `problem_finally`, a Lean-facing informal target that preserves the original mathematical objective while making assumptions, definitions, domains, notation, and conclusions explicit.

A.1 Intermediate Records

Each exercise is stored as a sequence of intermediate JSON fields rather than as a single rewritten statement. These fields record the main transformations from the raw textbook exercise to the final informal target.

Field	Role
`problem`	Raw textbook exercise and semantic anchor.
`problem_clean`	Cleaned mathematical task after removing non-essential hints, remarks, background narrative, plotting instructions, or implementation-specific guidance.
`problem_with_context`	Cleaned task enriched with recovered textbook dependencies, such as referenced definitions, equations, theorem statements, algorithms, notation conventions, and surrounding assumptions.
`problem_standardized_math`	Dependency-aware statement with normalized mathematical wording, consistent notation, and clarified locally inherited objects.
`problem_finally`	Self-contained Lean-facing informal target used as input to later Lean formalization.

Table 2: Intermediate natural-language fields used before Lean generation.

These records make different sources of error easier to localize. Cleaning errors can be identified by comparing `problem` and `problem_clean`. Dependency-recovery errors can be inspected through `problem_with_context`. Normalization errors can be checked against `problem_standardized_math`. Missing assumptions or unsupported additions can then be reviewed in `problem_finally`.

A.2 Cleaning the Raw Exercise

The first transformation produces `problem_clean` from the raw extracted exercise. This step is intentionally conservative. It removes non-essential pedagogical material while preserving formulas, variables, constraints, assumptions, conclusions, numbering, and explicit references.

The purpose of this step is not to solve, simplify, or reinterpret the problem. Rather, it isolates the mathematical task from surrounding textbook exposition. For example, hints, explanatory remarks, plotting instructions, and informal implementation advice may be removed, but the mathematical objective and all relevant hypotheses must remain unchanged.

Illustrative examples are provided in Appendix F.2, and the corresponding cleaning prompt is shown in Appendix G.1.

A.3 Dependency recovery and Source-grounded enrichment

Starting from `problem_clean`, the pipeline recovers local textbook dependencies through the recursive dependency expansion procedure described in Section 3.1. The resulting field

`problem_with_context` records both the enriched exercise and the dependency information used to construct it. This includes retrieved source fragments, original reference labels, provenance information, and, when applicable, multi-hop dependency expansions.

This enrichment step is source-grounded rather than free-form. The model may incorporate retrieved definitions, equations, notation, or assumptions, but the inserted information must be traceable to the source exercise or recovered dependencies.

The enriched statement is then converted into `problem_standardized_math`. This step folds the recovered context into a more uniform mathematical statement. In particular, it may expand unresolved references, standardize notation, clarify variables inherited from the textbook environment, and remove duplicated inserted material. Its role is normalization rather than mathematical rewriting: it must not introduce unsupported assumptions, change the proof objective, or alter the original mathematical content.

An illustrative example is provided in Appendix F.3.

A.4 Constructing the Self-Contained Target

The field `problem_finally` is the candidate self-contained informal target passed to the next stage of the benchmark construction pipeline. It is constructed primarily from `problem_standardized_math`, with `problem_with_context` used only for explicitly recovered dependencies.

This step makes implicit conditions explicit when they are needed for a well-formed Lean-facing statement. Examples include nonemptiness, nonzero denominators, positivity, invertibility, dimensional compatibility, convexity or regularity assumptions, and domain restrictions. Such conditions may be added only when supported by the original problem, recovered context, or the well-formedness requirements of objects already present in the exercise.

The output is a declarative informal target rather than a question-style textbook prompt. However, this transformation is constrained: the final statement must preserve the original assumptions, task objective, quantifiers, domains, and conclusions.

An illustrative example is given in Appendix F.4, and the prompt used for this step is shown in Appendix G.2.

A.5 Source-Level Review and Repair Interface

Before Lean formalization, each `problem_finally` is reviewed against the original textbook problem. The review checks whether the candidate target preserves the source assumptions, variables, domains, quantifiers, objectives, constraints, and conclusions; whether all necessary definitions and dependencies are present; and whether the statement is suitable as a Lean-facing theorem target.

The reviewer assigns one of three labels: `accept`, `revise`, or `hold`. Accepted items proceed to Lean formalization. Items marked `revise` contain local repairable issues, such as a missing assumption, a limited task drift, or a missing definition. These items enter a constrained repair branch and are then submitted to re-review.

Items marked `hold` are handled separately because the label covers different semantic risks. If the item depends on unresolved external context, such as an unrecovered theorem, tagged equation, algorithm, or previous exercise, we do not repair it by conjectural completion; the item is excluded unless the missing context can be recovered from verified source material. If the item is held only because of Lean-unsuitable wording or failed review parsing, it may enter a separate hold-repair branch, provided that the original source supports a conservative reformulation into a precise theorem-style statement.

Repair is not an acceptance step. In both repair branches, only `problem_finally` may be modified, while earlier fields such as `problem`, `problem_clean`, and `problem_standardized_math` remain fixed as audit anchors. The repair may restore missing assumptions, make definitions explicit, inline already recovered dependencies, or convert an open-ended exercise into an explicit theorem-style claim. It may not introduce unsupported

assumptions, weaken or strengthen the mathematical claim, change the proof objective, or invent missing dependencies. Every repaired item must pass re-review before it can be retained for subsequent Lean formalization.

The full classification labels and review protocol are reported in Appendix B. The representative examples are given in Appendix F.8 and Appendix F.9, and the prompts used for these steps are shown in Appendix G.3 and Appendix G.4. The statistics are shown in Appendix B.3.

B SOURCE-LEVEL SEMANTIC REVIEW

This appendix describes the source-level semantic review procedure used during benchmark construction.

B.1 REVIEW CRITERIA AND DECISION LABELS

The source-level semantic reviewer compares `problem_finally` against the original `problem`. The goal is not to improve style, but to determine whether the candidate target preserves the original mathematical meaning, contains the assumptions and definitions needed for self-containedness, and is suitable for Lean formalization.

The review first records diagnostic issue flags. These flags identify the type of semantic defect, if any, before the reviewer assigns a final decision label. Table 3 summarizes the issue categories used in this stage.

Table 3: Source-level semantic review criteria and diagnostic issue flags.

Review aspect	**Issue flag**	**Criterion**
Assumption preservation	`missing_assumption`	Flags cases where a necessary assumption from the source problem is absent or changed in `problem_finally`. This includes missing domain restrictions, nonzero, nonnegative, positivity, positive-definiteness, index range, continuity, differentiability, boundedness, invertibility, existence, uniqueness, or other well-posedness assumptions.
Task preservation	`task_drift`	Flags non-equivalent changes to the original mathematical task. This includes changed quantifiers, changed logical direction, changed domains, weakened or strengthened conclusions, dropped conclusions, added conclusions, or replacing the source task by a nearby but different statement. Equivalent Lean-facing reformulations are allowed only when they preserve the same domain, quantifiers, and conclusion.
Self-containedness	`missing_dependency`	Flags unresolved dependencies that prevent the statement from standing on its own, such as missing equation tags, theorem labels, algorithms, previous exercise references, local conventions, or cited facts that are mentioned but not restated clearly enough.
Definition completeness	`missing_def`	Flags technical terms, functions, operators, mappings, objects, or notation that are used without definitions precise enough for formalization. This issue is used only when the missing definition creates genuine ambiguity or blocks formalization, not for harmless presentation-level refinements.
Formalization suitability	`hold` condition	Checks whether the task is suitable as a Lean-facing target. Proof-oriented mathematical claims and precise computational statements are suitable. Open-ended or presentation-oriented tasks such as "describe", "formulate", "compute", "plot", "sketch", or informal discussion are held unless they have already been converted into a precise theorem-style statement.

Table 4: Decision labels for source-level semantic review.

Label	Assignment rule
`accept`	Used only when no issue is present and the candidate target is suitable for Lean formalization. The statement must preserve the original mathematical task, be self-contained, and have a precise formalization-ready conclusion.
`revise`	Used when the issue is local and repairable. In this workflow, `revise` is triggered by `missing_assumption`, `task_drift`, or `missing_def`, provided that the needed dependency context is available and the task is otherwise formalizable.
`hold`	Used when the candidate has unresolved dependencies or is not yet suitable for Lean formalization. This includes missing external context, unresolved tags or theorem references, open-ended exercise wording, non-formalizable tasks, or failed/uncertain review outputs.

After these issue flags are recorded, the reviewer assigns one of three final decision labels: `accept`, `revise`, or `hold`. Table 4 gives the assignment rule for each label.

The prompt for this stage is provided in Appendix G.3. After review, each problem is accompanied by a review log that records its decision label and the reasons for that decision. An illustrative example is provided in Appendix F.8.

B.2 Constrained Repair and Re-review

Items marked `revise` are not rewritten freely. Instead, they enter a constrained repair stage limited to the specific defect identified by the review report. The repair step is not allowed to introduce unsupported new assumptions, replace the original task by a nearby easier statement, or alter the intended mathematical objective.

After repair, the updated `problem_finally` is sent back to the same source-level reviewer for re-evaluation. This second pass checks whether the reported issue has been resolved without introducing a new one, and whether the repaired target remains faithful, self-contained, and suitable for Lean formalization.

Not every `hold` item enters the repair path. If the review report shows that the candidate still depends on missing external context, such as an unexpanded theorem reference, algorithm, or tagged formula, we do not attempt to repair it by conjectural completion. Such cases are filtered out unless the missing dependency can be recovered from verified source material. For Lean-unsuitable cases, repair is considered only when the source exercise already supports a conservative reformulation into a precise theorem-style target; otherwise the item remains on hold and is excluded from the released set.

The prompt for constrained repair and re-review is shown in Appendix G.4. An illustrative example is shown in Appendix F.9.

B.3 Global Source-Level Statistics

We now summarize the aggregate outcomes of the source-level review, repair, and re-review pipeline after data correction. The statistics are reported in three steps, corresponding to three different input pools.

First, the original review is applied to the full source-level input pool. We call this pool `OI`. Second, the first-round `revise` cases are repaired and sent to re-review; this gives the `RI` pool. Third, the first-round `hold` cases are filtered before re-review: cases held for `missing_dependency` are removed unless the missing context can be recovered from verified source material. The remaining hold-derived pool is called `HI`. Thus,

$$\texttt{RI} = \texttt{R1 revise}, \qquad \texttt{HI} = \texttt{R1 hold} - \texttt{missing_dependency}.$$

Step 1 reports the final decision distribution for each input pool. As shown in Table 5, the three pools have different accept, revise, and hold rates, so we report them separately rather than merging them into a single later-round result.

Table 5: Outcome distributions for the three source-level review input pools after data correction.

Input pool	Input size	Accepted	Revised	Held
OI	1550	410 (26.5%)	211 (13.6%)	929 (59.9%)
RI	211	93 (44.1%)	67 (31.8%)	51 (24.2%)
HI	578	338 (58.5%)	94 (16.3%)	146 (25.3%)

Step 2 examines the cases marked `revise`. Table 6 shows the distribution of revise reasons within each input pool. Since a single problem may carry more than one issue label, the total number of issue labels can exceed the number of revised cases.

Table 6: Reason distributions within the `revise` subset of each source-level review input pool.

Input pool	Revised	Missing assumption	Task drift	Missing definition
OI	211	45 (21.3%)	118 (55.9%)	112 (53.1%)
RI	67	16 (23.9%)	47 (70.1%)	29 (43.3%)
HI	94	16 (17.0%)	63 (67.0%)	18 (19.1%)

Step 3 examines the cases marked `hold`. These cases are not treated as ordinary local repair targets. Instead, they indicate unresolved dependencies, Lean-unsuitable task forms, or parse failures. Table 7 breaks down the hold reasons for each input pool.

Table 7: Reason distributions within the `hold` subset of each source-level review input pool.

Input pool	Held	Missing dependency	Lean-unsuitable	Parse error
OI	929	351 (37.8%)	512 (55.1%)	66 (7.1%)
RI	51	20 (39.2%)	12 (23.5%)	19 (37.3%)
HI	146	2 (1.4%)	83 (56.8%)	61 (41.8%)

C Details of Lean Formalization Pipeline

This section supplements the Lean formalization pipeline in the main text. We describe the record format, block-wise construction, semantic review, and final packaging used to produce the benchmark artifacts.

C.1 Lean-Facing Semantic Block Records

After preprocessing, each exercise is represented as an ordered list of Lean-facing semantic records:

`index, source, source_idx, kind, content.`

The optional `term` field is used when a block introduces a named definition, optimization problem, or algorithm. The field `kind` determines how the block is translated, as summarized in Table 8.

A flattened theorem record has the following form:

```
{
  "index": 3,
  "source": "book/convex_optimization_Chapter3",
  "source_idx": "Exercise 3.24-(c)",
  "kind": "thm",
  "content": "Let a_1,...,a_n be real numbers with a_1 < ... < a_n.
              Prove that f is convex, concave, quasiconvex, and
              quasiconcave."
}
```

Kind	Semantic role	Lean-level treatment
`hints`	Informal guidance.	Kept for provenance, usually excluded from Lean generation.
`defn`	Definitions, predicates, notation, or named terms.	Translated into Lean definitions or auxiliary declarations.
`opt_prob`	Objectives, feasible sets, constraints, or optimality conditions.	Translated into structures, feasibility predicates, or objective functions.
`algo` / `alg`	Algorithms, updates, or iterative constructions.	Translated into step functions, states, invariants, or validity predicates.
`thm`	Theorem, lemma, proposition, or final target.	Translated into a Lean theorem or lemma with proof body initialized as `by sorry`.

Table 8: Lean-facing semantic block records.

Records are processed in `index` order. The orchestrator inserts immutable block-boundary comments into the Lean file, so each generated declaration can be traced back to its semantic record.

C.2 Block-Wise Construction and Compilation Repair

Lean generation proceeds block by block. For each target block, the prefix of already accepted Lean code is treated as a frozen context; the model may only generate or repair the current block. The combined file is checked by:

```
lake env lean --threads=<LEAN_THREADS> <file>
```

For theorem-like blocks, `by sorry` is allowed during construction, so compilation validates the statement and declaration interface rather than a completed proof.

Algorithm 2 summarizes the translation-repair loop. The few-shot helper retrieves similar translation or repair examples according to the current block and, during repair, the Lean diagnostics.

A compact translation prompt is shown in Appendix G.5.

C.3 Semantic Review and Alternating Repair

Compilation does not ensure semantic faithfulness. A compilable block may omit assumptions, alter quantifiers, change domains, or weaken the intended conclusion. Thus, each compilable block is reviewed against its semantic record.

The reviewer returns a structured report:

```
{
  "math_equivalent": true,
  "truth_judgement": "true|false|unknown|not_applicable",
  "overall_status": "usable|usable_with_revision|not_usable_yet",
  "dominant_issue_types": ["missing_assumption", "task_drift"],
  "top_priority_fix": "...",
  "issues": [
    {
      "severity": "P0|P1|P2",
      "issue_type": "missing_assumption|wrong_boundary_case|task_drift",
      "location": "current declaration",
      "reason": "...",
      "suggested_fix": "..."
    }
  ],
  "confidence": 0.0
}
```

Algorithm 2: Block-wise Lean translation with local repair

Input: Semantic blocks $\mathcal{B} = (b_1, \ldots, b_m)$; Lean scaffold L_0; repair budget R; few-shot helper $\mathcal{E}$
Output: Combined Lean file L

$L \leftarrow L_0$
for $b \in \mathcal{B}$ **do**
  if b *is non-formalizable* **then**
    continue
  insert immutable source comment for b into L
  $F \leftarrow \textsc{FrozenContext}(L, b)$
  $E \leftarrow \textsc{RetrieveFewShot}(\mathcal{E}, b)$
  $c \leftarrow \textsc{TranslateBlock}(b, F, E)$
  $L' \leftarrow \textsc{InsertBlock}(L, b, c)$
  if $\textsc{Compile}(L')$ *succeeds* **then**
    $L \leftarrow L'$; freeze b; **continue**
  for $r = 1$ **to** R **do**
    $D \leftarrow \textsc{Diagnostics}(L')$
    $E \leftarrow \textsc{RetrieveFewShot}(\mathcal{E}, b, D)$
    $\widehat{c} \leftarrow \textsc{RepairBlock}(b, F, c, D, E)$
    $L' \leftarrow \textsc{ReplaceBlock}(L, b, \widehat{c})$
    if $\textsc{Compile}(L')$ *succeeds* **then**
      $L \leftarrow L'$; freeze b; **break**
    $c \leftarrow \widehat{c}$
  if b *is not frozen* **then**
    mark b as compilation failure and continue when possible
$\textsc{Compile}(L)$ `// final validation`
return L

Blocks marked `usable` are accepted. Otherwise, the current block is semantically rewritten, recompiled, and reviewed again. If semantic rewriting causes Lean errors, compilation repair is invoked under the semantic constraint that the intended correction must be preserved. Algorithm 3 summarizes this alternating loop.

C.4 Postprocessing, Packaging, and Validation

After semantic review, accepted Lean files are conservatively postprocessed and exported. This stage standardizes comments, namespaces, formatting, and metadata while leaving accepted declarations unchanged.

Each benchmark instance contains the Lean file and a metadata record with the source exercise, informal statement, block identifiers, validation status, and basic construction statistics. The pipeline also saves compiler diagnostics, semantic-review reports, repair histories, and failure records. The final artifacts separate statement construction from proof-generation evaluation: downstream provers replace only the `sorry` placeholders while the formalized statements remain fixed.

D Experimental Settings

D.1 Code and Data Availability

We provide the code and dataset to support reproducibility. The anonymized code repository is available at: `https://anonymous.4open.science/r/CAM-Bench-2D32`.

The dataset is hosted separately and can be accessed through the following DOI: `https://doi.org/10.7910/DVN/B5JTZI`.

Algorithm 3: Block-wise semantic review and repair

Input: Lean file L; blocks $\mathcal{B} = (b_1, \ldots, b_m)$; budgets S, R; few-shot helpers $\mathcal{H}$
Output: Reviewed Lean file L

$\mathcal{A} \leftarrow [\,]$ `// accepted previous blocks`
for $i = 1$ **to** m **do**
  $b \leftarrow b_i$
  if b *is not formalizable* **then**
    continue
  $F \leftarrow \textsc{FrozenContext}(L, b, \mathcal{A})$
  $\mathsf{Hist}_b \leftarrow [\,]$
  for $s = 1$ **to** S **do**
    $c \leftarrow \textsc{ExtractBlock}(L, b)$
    $\rho \leftarrow \textsc{Review}(b, c, F, \mathcal{A})$
    if ρ *accepts* c **then**
      append b to $\mathcal{A}$; mark b accepted; **break**
    $c' \leftarrow \textsc{Rewrite}(b, c, \rho, F, \mathcal{A}, \mathsf{Hist}_b)$
    $L' \leftarrow \textsc{PatchCurrentBlock}(L, b, c')$
    if $\textsc{Compile}(L')$ *fails* **then**
      $\Delta \leftarrow \textsc{Diagnostics}(L')$
      $\mathsf{Hist}_b \leftarrow \mathsf{Hist}_b \cup \{(s, c', \rho, \Delta)\}$
      $\mathcal{E} \leftarrow \textsc{SelectFewShots}(\mathcal{H}, \Delta, \rho)$
      $L' \leftarrow \textsc{CompilationRepair}(L', b, F, \mathcal{A}, R, \Delta, \mathsf{Hist}_b, \mathcal{E})$
    if $\textsc{Compile}(L')$ *succeeds* **then**
      $L \leftarrow L'$ `// re-review repaired block next round`
    else
      mark b failed; **break**
return L

D.2 Baseline Setup

We evaluate Lean proof generation on the released suite of 500 designated `sorry` proof obligations. Each obligation is paired with its recovered local context.

All Lean verification is performed with Lean 4 and Mathlib through Lake:

`leanprover/lean4:v4.28.0`, `mathlib:v4.28.0`.

Each candidate proof is checked by running `lake env lean`. A proof attempt is considered successful only if the resulting Lean file compiles and passes integrity checks.

We compare Codex `gpt-5.6-sol`, Claude `claude-opus-5`, and an agentic `gpt-5.6-sol` configuration. All three configurations use the same cumulative 2M-token budget per obligation. The Codex and Claude baselines use the pass@32 protocol described below, with local Lean validation of each candidate. Baseline candidates are generated against the same fixed target and do not receive compiler feedback from earlier baseline attempts. The agentic configuration instead performs a multiturn proof-search trajectory with compiler diagnostics, theorem search, and execution-guided repair before its final submission.

The proof-search solver presents the fixed obligation and its recovered local context, asks the model for the proof body, and checks the resulting declaration with Lean. An obligation is counted as solved only when the submitted proof passes the final integrity and kernel checks.

D.3 Evaluation Protocol

All experiments use a fixed proof-generation prompt template. The prompt provides the model with the informal problem, the Lean statement, and the surrounding Lean context, and asks it to return the proof body for the designated `sorry`. Each obligation receives a cumulative 2M-token budget. Here, `pass@32` denotes only the Codex and Claude baseline settings, with up to 32 independently generated candidate proof attempts for each designated obligation. Every candidate is inserted into

Table 9: Full-suite verified accuracy and mean token use on 500 designated proof obligations.

Agent configuration	Verified	Accuracy	Mean tokens/obligation
Codex `gpt-5.6-sol`	242/500	48.4%	1.24M
Claude `claude-opus-5`	196/500	39.2%	0.72M
Agentic `gpt-5.6-sol`	151/500	30.2%	1.71M

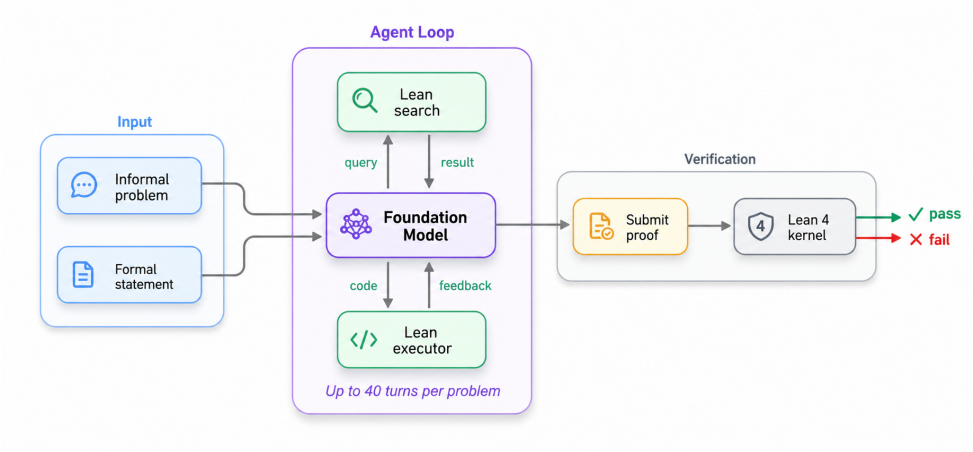


Figure 7: Agentic proof-search setting used in CAM-Bench. The model receives both the informal problem and the formal Lean target, iteratively interacts with Lean search and a Lean executor, and submits a proof for final Lean-kernel verification. Compared with the FormalProofBench (Ravi et al., 2026) setup, our setting follows the same multiturn proof-search protocol but replaces Lean Loogle with Lean search.

the fixed Lean target and checked with the same integrity and kernel validation; compiler diagnostics from one baseline attempt are not fed into later baseline attempts. The procedure stops when a candidate is accepted, when 32 attempts have been checked, or when the cumulative 2M-token budget is exhausted, whichever comes first. The final score is the fraction of the 500 obligations whose reconstructed Lean declaration verifies.

D.4 Run Summaries

Table 9 reports the full-suite results for all three configurations.

All three measurements use the common denominator of 500 obligations, and the reported percentages are computed directly from the verified counts. Mean token use counts model input and output together with any recorded tool or compiler interaction, averaged over all 500 obligations.

D.5 Agentic Proof-Search Setting

Figure 7 illustrates our agentic proof-search setting. Our setting follows the multiturn, tool-augmented evaluation style of FormalProofBench (Ravi et al., 2026), where the model is given the natural-language problem, the Lean 4 formalization, and the required headers and context, and may interact with proof-assistance tools before submitting a final proof. While FormalProofBench uses Lean Loogle as its theorem-search tool, our implementation uses **Lean search** instead.

For each obligation, the agent receives the normalized informal theorem, the corresponding Lean statements, and all accepted local declarations recovered during benchmark construction. The agent may query Lean search for relevant Lean/Mathlib facts, test candidate proof fragments with the Lean executor, and use compiler diagnostics, unsolved goals, type mismatches, missing-lemma errors, and previous failed attempts to guide iterative repair. The agent is allowed up to 40 interaction turns per obligation within the same cumulative 2M-token budget used by the direct baselines. The

trajectory stops when an accepted final proof is produced, when 40 turns have been used, or when the cumulative token budget is exhausted, whichever comes first. The final answer must be a proof body starting with `by`, which is inserted into the original theorem without modifying the theorem statement or any preceding definitions.

We use a strict verifier for final acceptance. A proof obligation is counted as solved only if the completed Lean file recompiles successfully and the submitted proof contains no prohibited proof shortcuts or unresolved holes, including `sorry`, `admit`, `axiom`, `sorryAx`, `exact?`, or `apply?`. We also require that the original theorem name, statement, assumptions, and previously accepted declarations are preserved. Thus, the agentic setting evaluates interactive proof construction rather than statement rewriting or benchmark-specific weakening of the target.

Comparison of interaction modes. The agentic configuration receives the same 500 designated obligations and 2M-token limit as the other two configurations, while allowing explicit compiler-guided repair. It reaches 30.2% verified accuracy with mean token use of 1.71M per obligation.

E Details of External Agent Evaluation and Failure Analysis

E.1 Details of Failure Analysis

This appendix provides additional details for the formalization-error analysis in Section 5.3. We analyze the failed outcomes reported in Table 1. For the Claude Code and Codex column, the unit is a failed task file at the terminal runner state; repeated attempts within a task are not counted separately. The runner reports infrastructure errors, boundary violations, integrity violations, and unverified proofs. We map infrastructure errors to library/API grounding, boundary or integrity violations to formal representation, and unverified proofs to proof decomposition. For M2F and Aristotle, the unit is an unsolved sampled obligation. The Claude Code and Codex mapping is an operational diagnostic proxy, whereas the external-system labels are semantic and multi-label. In each case, the terminal artifact fails at least one local acceptance check, such as compilation, integrity, or target-boundary validation. For the external systems, labels are diagnostic rather than mutually exclusive: a single failed outcome may involve multiple symptoms.

Library/API Grounding and Mathematical Infrastructure. This category covers failures where the model cannot ground a plausible mathematical proof in the available Lean/Mathlib environment.

1. **Hallucinated or invalid APIs.** The model invokes nonexistent theorem names, invalid fields, unsupported API patterns, or real declarations with incompatible signatures.
2. **Missing bridge results.** The proof requires intermediate lemmas that connect the current goal to available Mathlib facts, but these bridge results are absent or not found.
3. **Domain-specific infrastructure gaps.** The failure occurs in higher-level applied-mathematics infrastructure, such as matrices, spectra, positive semidefinite cones, convexity, optimality conditions, duality, or convergence.

Formal Representation and Type Discipline. This category covers failures caused by Lean's precise representation requirements.

1. **Type, dimension, and index errors.** The generated proof uses incompatible types, dimensions, or indexing conventions, especially for vectors, matrices, finite index sets, and parameterized constraints.
2. **Coercion and typeclass failures.** The proof misses required coercions, scalar casts, subtype conversions, order or normed-space instances, or other typeclass obligations.
3. **Notation and side-condition failures.** The attempt uses unsuitable notation or leaves side conditions generated by rewriting, simplification, inequalities, or algebraic transformations unresolved.

Proof Decomposition and Long-Horizon Control. This category covers failures where the proof requires a longer chain of intermediate claims than the model can reliably organize.

1. **Missing auxiliary lemmas.** The model does not introduce the intermediate claims needed to connect the assumptions to the final theorem.
2. **Unfinished proof construction.** The final artifact still contains residual placeholders, incomplete helper lemmas, abandoned branches, or proof plans that exceed the search or repair budget.
3. **Loss of long-horizon control.** The proof loses consistency across a multi-step argument involving optimization models, matrix identities, convexity reasoning, certificates, invariants, algorithms, or convergence claims.

Overall, the observed failures are rarely isolated syntax errors. They usually arise from the interaction of missing or misused Mathlib infrastructure, Lean type-system constraints, and incomplete long-horizon proof planning.

We give three short case studies that illustrate the coarse error categories defined above and discussed in Appendix F.10.

F Illustrative Examples

This appendix provides representative examples for the main construction stages used in our benchmark pipeline. The examples are organized in the same order as the pipeline: semantic block decomposition, raw-to-clean preprocessing, dependency recovery and standardization, self-contained target construction, source-level review, and constrained repair.

F.1 Example: Semantic Block Decomposition

This example illustrates how a heterogeneous textbook exercise is decomposed into typed semantic blocks before Lean generation. The raw exercise combines assumptions, notation conventions, an optimization formulation, a request to derive KKT conditions, and a separate optimality claim. Directly translating such mixed content into Lean is difficult because these components play different formal roles and should not be treated as a single undifferentiated problem statement.

Our preprocessing step separates the exercise into reusable mathematical background, problem-specific data, and theorem targets. General concepts such as KKT conditions, stationarity, primal feasibility, and complementary slackness are extracted as definition blocks. The logarithmic simplex program is isolated as an optimization-problem block, and the two requested tasks are separated into theorem-target blocks.

Raw exercise. Let $n \in \mathbf{N}$ with $n \geq 2$. Let $a = (a_1, \ldots, a_n) \in \mathbf{R}^n$ satisfy

$$a_1 \geq a_2 \geq \cdots \geq a_n > 0,$$

and define $b = (b_1, \ldots, b_n) \in \mathbf{R}^n$ by

$$b_k = \frac{1}{a_k}, \qquad k = 1, \ldots, n.$$

Let $\mathbf{1} \in \mathbf{R}^n$ denote the all-ones vector, and let $x \succeq 0$ mean $x_i \geq 0$ for all $i = 1, \ldots, n$.

The exercise considers the logarithmic simplex program

$$\begin{aligned} &\text{minimize} && -\log(a^T x) - \log(b^T x) \\ &\text{subject to} && x \succeq 0, \quad \mathbf{1}^T x = 1, \end{aligned}$$

with variable $x \in \mathbf{R}^n$. It then asks the solver to write the Karush–Kuhn–Tucker conditions explicitly and to show that

$$x = \left(\frac{1}{2}, 0, \ldots, 0, \frac{1}{2}\right)$$

is optimal.

The KKT part of the exercise asks to show that the conditions are the existence of a scalar multiplier $\nu \in \mathbf{R}$ and inequality multipliers $\lambda = (\lambda_1, \ldots, \lambda_n) \in \mathbf{R}^n$ such that

$$\mathbf{1}^T x = 1, \qquad x_i \geq 0, \qquad \lambda_i \geq 0, \qquad \lambda_i x_i = 0 \quad (i = 1, \ldots, n),$$

and, for each $i = 1, \ldots, n$,

$$-\frac{a_i}{a^T x} - \frac{b_i}{b^T x} + \nu - \lambda_i = 0.$$

Preprocessed semantic blocks. After preprocessing, the raw exercise is converted into the following typed semantic blocks.

1. **Definition block: Karush–Kuhn–Tucker conditions.** For a problem of minimizing $f(x)$ subject to inequality constraints $g_i(x) \leq 0$ and equality constraints $h_j(x) = 0$, the KKT conditions are the existence of multipliers $\lambda_i \geq 0$ and ν_j such that primal feasibility holds, stationarity holds,
$$\nabla f(x) + \sum_i \lambda_i \nabla g_i(x) + \sum_j \nu_j \nabla h_j(x) = 0,$$
and complementary slackness holds:
$$\lambda_i g_i(x) = 0 \quad \text{for all } i.$$

2. **Definition block: complementary slackness.** For inequality constraints $g_i(x) \leq 0$ with multipliers $\lambda_i \geq 0$, complementary slackness means
$$\lambda_i g_i(x) = 0 \quad \text{for every } i.$$

3. **Definition block: stationarity.** A KKT point satisfies stationarity if the gradient of the Lagrangian with respect to the primal variable vanishes:
$$\nabla_x L(x, \lambda, \nu) = 0.$$

4. **Definition block: primal feasibility.** A point x is primal feasible if it satisfies all original constraints of the optimization problem.

5. **Optimization-problem block: logarithmic simplex program.** The optimization problem is isolated as
$$\begin{aligned} &\text{minimize} && -\log(a^T x) - \log(b^T x) \\ &\text{subject to} && x \succeq 0, \quad \mathbf{1}^T x = 1, \end{aligned}$$
with variable $x \in \mathbf{R}^n$.

6. **Theorem-target block: explicit KKT conditions.** Under the assumptions
$$n \geq 2, \qquad a_1 \geq a_2 \geq \cdots \geq a_n > 0, \qquad b_k = \frac{1}{a_k},$$
the first theorem target asks for an explicit characterization of the KKT conditions for the logarithmic simplex program. The target statement asserts the existence of $\nu \in \mathbf{R}$ and $\lambda = (\lambda_1, \ldots, \lambda_n) \in \mathbf{R}^n$ satisfying
$$\mathbf{1}^T x = 1, \qquad x_i \geq 0, \qquad \lambda_i \geq 0, \qquad \lambda_i x_i = 0 \quad (i = 1, \ldots, n),$$
together with the stationarity equations
$$-\frac{a_i}{a^T x} - \frac{b_i}{b^T x} + \nu - \lambda_i = 0, \qquad i = 1, \ldots, n.$$

7. **Theorem-target block: optimality of the endpoint mixture.** Under the same assumptions and for the same logarithmic simplex program, the second theorem target asks to prove that
$$x = \left(\frac{1}{2}, 0, \ldots, 0, \frac{1}{2}\right)$$
is optimal.

Discussion. This decomposition makes the role of each component explicit before Lean generation. General mathematical background is separated from problem-specific data, the optimization formulation is isolated from the requested claims, and the two theorem targets can be translated, repaired, and validated independently.

F.2 Example: Raw-to-Clean Preprocessing

This example illustrates the cleaning step that converts a raw textbook exercise into the `problem_clean` field. The cleaning step removes non-essential pedagogical material while preserving the mathematical statement, notation, assumptions, and objective.

F.2.1 Removing Proof Hints

Raw exercise.

> Exercise 11.5. When $r : \mathbf{R}^n \to \mathbf{R}^n$, show that the function
>
> $$\phi(\lambda) = \left\|(J^T J + \lambda I)^{-1} J^T r\right\|$$
>
> is monotonically decreasing in λ, unless $J^T r = 0$. Hint: Use the singular-value decomposition of J.

Cleaned exercise.

> Exercise 11.5. When $r : \mathbf{R}^n \to \mathbf{R}^n$, show that the function
>
> $$\phi(\lambda) = \left\|(J^T J + \lambda I)^{-1} J^T r\right\|$$
>
> is monotonically decreasing in λ, unless $J^T r = 0$.

Discussion. This example illustrates a minimal cleaning operation. The mathematical statement, notation, and exception condition are preserved exactly, while the proof hint is removed.

F.2.2 Removing Application Narrative

Raw exercise.

> Exercise 17.3. Flux balance analysis in systems biology. Flux balance analysis is based on a simple model of reactions in a cell, keeping track only of the gross rate of consumption and production of chemical species. Based on known stoichiometry and upper bounds on some reaction rates, we can compute bounds on other reaction rates or on cell growth.
>
> We focus on m metabolites $M_1, \ldots, M_m$ and n reactions $R_1, \ldots, R_n$, with nonnegative reaction rates $v_1, \ldots, v_n$. The stoichiometry matrix $S \in \mathbf{R}^{m \times n}$ is defined so that S_{ij} is the rate of production of metabolite M_i due to unit reaction rate $v_j = 1$, with consumption represented by negative production.
>
> The last reaction R_n corresponds to biomass creation, so v_n is the cell growth rate. Conservation of metabolites is expressed as
>
> $$Sv = 0.$$
>
> We are also given upper limits on some reaction rates, written as
>
> $$v \succeq 0, \qquad v \preceq v^{\max},$$
>
> where $v_j^{\max} = \infty$ if no upper bound on reaction rate j is known.
>
> The questions below pertain to the data found in `fba_data.m`.
>
> (a) Find the maximum possible cell growth rate $G^\star$, as well as optimal Lagrange multipliers for the reaction rate limits. How sensitive is the maximum growth rate to the various reaction rate limits?

Cleaned exercise.

> Exercise 17.3. Let $S \in \mathbf{R}^{m\times n}$ be the stoichiometry matrix, let $v_1, \ldots, v_n$ be nonnegative reaction rates, and let $v^{\max}$ satisfy $v_j^{\max} = \infty$ if no upper bound on reaction rate j is known. The last reaction rate v_n is the cell growth rate. Consider the constraints
>
> $$Sv = 0, \qquad v \succeq 0, \qquad v \preceq v^{\max}.$$
>
> The data are given in `fba_data.m`.
>
> (a) Find the maximum possible cell growth rate $G^\star$, and optimal Lagrange multipliers for the reaction rate limits. Determine the sensitivity of the maximum growth rate to the reaction rate limits.

Discussion. This example shows a larger cleaning operation. The biological motivation and illustrative reaction examples are removed, while the formal variables, constraints, data dependency, and optimization task are retained.

F.3 Example: Dependency Recovery and Standardization

This example illustrates a case in which an exercise depends on both a target problem reference and an algorithm reference. The algorithm further introduces a second-level formula dependency. The recovered record therefore contains four main components: the cleaned exercise, dependency metadata, the context-enriched statement, and the standardized mathematical statement used downstream.

Source. `book/Numerical Optimization_Chp16`, Exercise 16.24.

Recovered dependency metadata. Table 10 summarizes the dependency chain identified during recursive dependency recovery. The original exercise directly refers to problem (16.76) and Algorithm 16.4, while Algorithm 16.4 introduces an additional formula dependency on (16.58).

Table 10: Recovered dependencies for Exercise 16.24.

Level	Field	Label	Evidence
1	`body_refs`	`eq:16.76`	`problem (16.76)`
1	`theorem_refs`	`alg:16.4`	`refs_attr`
2	`inner_body_refs`	`eq:16.58`	`(16.58)`
2	`expanded_body_refs`	`eq:16.58`	`via:Algorithm 16.4`

Recovered record. The following display shows the intermediate records produced by dependency recovery and standardization.

Input. Cleaned exercise.

> Exercise 16.24. Program Algorithm 16.4 and use it to solve problem (16.76). Set all initial variables to be the vector $e = (1, 1, \ldots, 1)^T$.

Stage 1. Context-enriched problem. The context-enriched version keeps the inserted references explicit and traceable.

Reference formula (16.76).

$$\begin{array}{ll} \max & 6x_1 + 4x_2 - 13 - x_1^2 - x_2^2, \\ \text{subject to} & x_1 + x_2 \le 3, \qquad x_1 \ge 0, \qquad x_2 \ge 0. \end{array} \tag{16.76}$$

Reference result Algorithm 16.4. Algorithm 16.4, the predictor-corrector algorithm for quadratic programming, computes (x_0, y_0, λ_0) with $(y_0, \lambda_0) > 0$. For $k = 0, 1, 2, \ldots$, set

$$(x, y, \lambda) = (x_k, y_k, \lambda_k)$$

and solve (16.58) with $\sigma = 0$ for

$$(\Delta x^{\text{aff}}, \Delta y^{\text{aff}}, \Delta \lambda^{\text{aff}}).$$

Then calculate

$$\mu = \frac{y^T \lambda}{m},$$

$$\hat{\alpha}_{\text{aff}} = \max\left\{\alpha \in (0,1] \mid (y,\lambda) + \alpha(\Delta y^{\text{aff}}, \Delta \lambda^{\text{aff}}) \geq 0\right\},$$

and

$$\mu_{\text{aff}} = \frac{(y + \hat{\alpha}^{\text{aff}} \Delta y^{\text{aff}})^T (\lambda + \hat{\alpha}^{\text{aff}} \Delta \lambda^{\text{aff}})}{m}.$$

The centering parameter is then set to

$$\sigma = \left(\frac{\mu_{\text{aff}}}{\mu}\right)^3.$$

Second-level reference formula (16.58).

$$\begin{bmatrix} G & 0 & -A^T \\ A & -I & 0 \\ 0 & \Lambda & \mathcal{Y} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -r_d \\ -r_p \\ -\Lambda \mathcal{Y} e + \sigma \mu e \end{bmatrix}. \tag{16.58}$$

STAGE 2. STANDARDIZED MATHEMATICAL STATEMENT. The standardized version removes explicit provenance markers and rewrites the recovered references into a self-contained mathematical task.

> Use the predictor-corrector algorithm for quadratic programming to solve
>
> $$\begin{array}{ll} \max & 6x_1 + 4x_2 - 13 - x_1^2 - x_2^2, \\ \text{subject to} & x_1 + x_2 \leq 3, \qquad x_1 \geq 0, \qquad x_2 \geq 0. \end{array}$$
>
> Take all initial variables equal to $e = (1, 1, \ldots, 1)^T$. Equivalently, compute an initial point (x_0, y_0, λ_0) with all components equal to 1 and with $(y_0, \lambda_0) > 0$.
>
> For $k = 0, 1, 2, \ldots$, set
>
> $$(x, y, \lambda) = (x_k, y_k, \lambda_k)$$
>
> and solve, with $\sigma = 0$,
>
> $$\begin{bmatrix} G & 0 & -A^T \\ A & -I & 0 \\ 0 & \Lambda & \mathcal{Y} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -r_d \\ -r_p \\ -\Lambda \mathcal{Y} e + \sigma \mu e \end{bmatrix},$$
>
> where
>
> $$r_d = Gx - A^T \lambda + c, \qquad r_p = Ax - y - b.$$
>
> This gives
>
> $$(\Delta x^{\text{aff}}, \Delta y^{\text{aff}}, \Delta \lambda^{\text{aff}}).$$
>
> Next compute
>
> $$\mu = \frac{y^T \lambda}{m},$$
>
> $$\hat{\alpha}_{\text{aff}} = \max\left\{\alpha \in (0,1] \mid (y,\lambda) + \alpha(\Delta y^{\text{aff}}, \Delta \lambda^{\text{aff}}) \geq 0\right\},$$
>
> and
>
> $$\mu_{\text{aff}} = \frac{(y + \hat{\alpha}^{\text{aff}} \Delta y^{\text{aff}})^T (\lambda + \hat{\alpha}^{\text{aff}} \Delta \lambda^{\text{aff}})}{m}.$$
>
> Set the centering parameter to
>
> $$\sigma = \left(\frac{\mu_{\text{aff}}}{\mu}\right)^3.$$
>
> Solve the given quadratic programming problem with this initialization.

Discussion. This example highlights the distinction between enrichment and normalization. The context-enriched record preserves the provenance of inserted references in a traceable form, while the standardized mathematical statement converts those recovered dependencies into a cleaner self-contained problem for downstream completion and review.

F.4 Example: Self-Contained Target Construction

This example illustrates how a standardized mathematical statement is converted into the final self-contained informal target. The final target makes domains, definitions, and implicit well-formedness conditions explicit before Lean formalization.

Input. Cleaned exercise.

> Exercise 8.14. Let C be a polyhedron in $\mathbf{R}^n$ described as $C = \{x \mid Ax \preceq b\}$, and suppose that $\{x \mid Ax \prec b\}$ is nonempty. (a) Show that the maximum volume ellipsoid enclosed in C, expanded by a factor n about its center, contains C.

Stage 1. Standardized mathematical statement.

> Let
> $$C = \{x \in \mathbf{R}^n \mid Ax \preceq b\},$$
> and assume that the strict inequality system is feasible, i.e.,
> $$\{x \in \mathbf{R}^n \mid Ax \prec b\} \neq \emptyset.$$
> Show that if an ellipsoid of maximum volume is contained in C, then the ellipsoid obtained by expanding it by a factor n about its center contains C.

Stage 2. Self-contained target.

> Let $A \in \mathbf{R}^{m\times n}$ and $b \in \mathbf{R}^m$, and define
> $$C = \{x \in \mathbf{R}^n \mid Ax \preceq b\},$$
> where $(Ax)_i \leq b_i$ for all $i = 1, \ldots, m$. Assume
> $$\{x \in \mathbf{R}^n \mid (Ax)_i < b_i \text{ for all } i = 1, \ldots, m\} \neq \emptyset.$$
> An ellipsoid in $\mathbf{R}^n$ is a set of the form
> $$E = \{x \in \mathbf{R}^n \mid (x-c)^T P^{-1}(x-c) \leq 1\},$$
> where $c \in \mathbf{R}^n$ and $P \in \mathbf{R}^{n\times n}$ is symmetric positive definite. For $\alpha > 0$, define the expansion of E about its center c by
> $$c + \alpha(E-c) = \{c + \alpha(x-c) \mid x \in E\}.$$
> Show that if E is a maximum-volume ellipsoid contained in C, then $c + n(E-c)$ contains C.

Discussion. The final target preserves the original mathematical task while making the objects needed for a formal theorem statement explicit. In particular, it specifies the dimensions of A and b, expands the componentwise inequality notation, defines the ellipsoid model, and formalizes the expansion operation about the center.

F.5 Example: Multi-Block Decomposition

We illustrate the block decomposition (Section C.1) with Exercise 3.12 from the Boyd–Vandenberghe extra exercises, which concerns a trace-minimization semidefinite program (SDP). The preprocessor decomposes the exercise into four ordered records:

index	kind	term	content (abbreviated)
4	`defn`	semidefinite program	A semidefinite program has a symmetric-matrix variable…
5	`opt_prob`	trace minimization SDP	minimize $\mathrm{tr}\, X$ s.t. $\begin{bmatrix} A & B \\ B^\top & X \end{bmatrix} \succeq 0$
6	`thm`	—	Show $X = B^\top A^{-1} B$ solves the SDP.
7	`thm`	—	Conclude $(A, B) \mapsto \mathrm{tr}(B^\top A^{-1} B)$ is convex.

The four records are translated according to their semantic roles. Block 4 (`defn`) is retained as provenance and contextual information, but it does not necessarily produce a standalone Lean declaration. Block 5 (`opt_prob`) is translated into Lean declarations that encode the problem data, the feasibility predicate, and the objective function. In the generated Lean file, these declarations appear under explicit block-boundary comments:

```
/- [BLOCK Exercise 3.12 | 5 | opt_prob] -/
structure TraceMinimizationSDP (m n : ) where
 A    : Matrix (Fin m) (Fin m)
 A_symm  : A = A
 A_posDef : Matrix.PosDef A
 B    : Matrix (Fin m) (Fin n)

def TraceMinimizationSDP.isFeasible {m n : }
  (P : TraceMinimizationSDP m n)
  (X : Matrix (Fin n) (Fin n) ) : Prop :=
 X = X  Matrix.PosSemidef (P.blockMatrix X)

def TraceMinimizationSDP.traceObjective {m n : }
  (P : TraceMinimizationSDP m n)
  (X : Matrix (Fin n) (Fin n) ) : :=
 Matrix.trace X
```

Blocks 6 and 7 become separate Lean theorem targets with `by sorry` proof bodies. These proof placeholders are intentional: they mark the targets that are later passed to downstream proof generation and verification.

```
/- [BLOCK Exercise 3.12 | 6 | thm] -/
theorem traceMinimizationSDP_solution
  {m n : } (P : TraceMinimizationSDP m n) :
  P.isFeasible (P.B * P.źA * P.B)
    X : Matrix (Fin n) (Fin n) , P.isFeasible X
     Matrix.trace (P.B * P.źA * P.B)  Matrix.trace X := by
 sorry

/- [BLOCK Exercise 3.12 | 7 | thm] -/
theorem trace_btranspose_inv_mul_b_convex_on {m n : } :
  ConvexOn
   {p : Matrix (Fin m) (Fin m)  Œ Matrix (Fin m) (Fin n) |
     p.1 = p.1  Matrix.PosDef p.1}
   (fun p => Matrix.trace (p.2 * pź.1 * p.2)) := by
 sorry
```

The block-boundary comments, such as `/- [BLOCK Exercise 3.12 | 6 | thm] -/`, are inserted by the orchestrator before each generated declaration. They make the Lean artifact traceable back to the corresponding semantic record.

F.6 Example: One Translation–Repair Cycle

We illustrate the compilation-repair loop (Section C.2) with the `thm` block for Exercise 3.7-(b), which asks to show that a power-constraint set is convex. The translator's first attempt used an invalid summation form for the target Lean 4 environment:

```
-- First attempt: compile fails.
theorem power_constraint_set_convex
  (m n : ) (hm : 10  m) (a : Fin n  Fin m )
  (I_des : ) (hI : 0 < I_des) :
  Convex  {p : Fin m  |
   ( j, 0  p j)
    S : Finset (Fin m), S.card = 10
     j in S, p j  (1 / 2 : ) *  j : Fin m, p j} := by
 sorry
```

Lean emitted the following diagnostic:

```
unexpected token 'in'; expected ','
  j in S, p j (1 / 2 :) * j : Fin m, p j
```

The repair step receives this diagnostic together with the frozen Lean prefix. It then rewrites only the current block, replacing the unsupported notation with a Lean 4-compatible expression:

```
-- Repair r = 1: compile succeeds.
theorem power_constraint_set_convex
  (m n :) (hm : 10 m) (a : Fin n Fin m )
  (I_des :) (hI : 0 < I_des) :
  Convex {p : Fin m |
   ( j, 0 p j)
    S : Finset (Fin m), S.card = 10
     Finset.sum S p (1 / 2 :) * j : Fin m, p j} := by
 sorry
```

The combined Lean file compiles after one repair attempt, so the corrected block is accepted and frozen. Later blocks can use it as read-only context, but the repair loop is not allowed to modify it again.

F.7 Example: One Semantic Review–Rewrite Cycle

We illustrate the semantic review loop (Section C.3) with the `thm` block for Exercise 9.4-(a) in *Numerical Optimization*. The exercise asks to show that six distinct collinear points in $\mathbf{R}^2$ do not uniquely determine a quadratic. The initial translation produced a Lean statement that compiled, but its hypothesis did not faithfully encode collinearity:

```
-- Compilable but semantically incorrect.
theorem six_collinear_points_do_not_determine_quadratic_uniquely
  (P : Fin 6  Œ )
  (hcol : A B C :,
   A  0  B  0
    i : Fin 6, A * (P i).1 + B * (P i).2 = C)
  (hdistinct : Function.Injective P) :
   q q : Œ  , q  q
    q  q
    i : Fin 6, q (P i) = q (P i) := by
 sorry
```

Because $\wedge$ binds more tightly than $\vee$, the hypothesis `hcol` is parsed as

$$\exists A\, B\, C,\ A \neq 0 \ \vee\ \big(B \neq 0 \wedge \forall i,\ A(P_i)_1 + B(P_i)_2 = C\big).$$

Thus, in the branch `A ≠ 0`, the theorem imposes no line constraint on the six points. The semantic reviewer therefore flags a `missing_assumption` issue:

```
{
 "math_equivalent": false,
 "overall_status": "not_usable_yet",
 "dominant_issue_types": ["missing_assumption"],
 "top_priority_fix":
  "Fix logical grouping in hcol: require (A  0  B  0)
     and the pointwise line condition together.",
 "issues": [{
  "severity": "P0",
  "issue_type": "missing_assumption",
  "reason":
   "In the A  0 branch, no condition is imposed on the points,
        so collinearity is not actually assumed.",
  "suggested_fix":
   " A B C : , (A  0  B  0)
         i : Fin 6, A * (P i).1 + B * (P i).2 = C"
 }]
}
```

The semantic rewrite step corrects the logical grouping while preserving the surrounding theorem structure:

```
-- After semantic rewrite: compile succeeds and reviewer accepts.
theorem six_collinear_points_do_not_determine_quadratic_uniquely
  (P : Fin 6 Œ)
  (hcol : A B C :,
   (A 0 B 0)
    i : Fin 6, A * (P i).1 + B * (P i).2 = C)
  (hdistinct : Function.Injective P):
   q q : Œ ,q q
    q q
    i : Fin 6, q (P i) = q (P i) := by
 sorry
```

The rewritten statement compiles immediately, so no additional compilation-repair sub-loop is triggered. This example shows why semantic review is needed even after Lean accepts the file: a precedence error can silently weaken a mathematical assumption without causing a compiler diagnostic.

F.8 Example: Source-Level Review

This example shows representative source-level review records. Each record compares the final informal target against the original problem and assigns a review label such as **accept**, **revise**, or **hold**. The records are shown in their original JSON-style format.


```
[
  {
    "case": "accept",
    "source_idx": "Exercise 7.2",
    "source": "book/Numerical Optimization_Chp7",
    "problem": "Show that the matrix $\hat{H}_{k+1}$ in (7.21) is singular.",
    "problem_finally": "Let $f : \mathbb{R}^n \to \mathbb{R}$ be differentiable, and let
    $s_k, y_k, p_k \in \mathbb{R}^n$ satisfy $y_k^T s_k \neq 0$. Define $\hat{H}_{k+1} = I - s_k y_k^T/(y_k^T s_k)$, where $I$
    is the $n \times n$ identity matrix. Suppose $p_{k+1} = \ldots = -\hat{H}_{k+1}\nabla f_{k+1}$. Show
    that $\hat{H}_{k+1}$ is singular.",
    "semantic_report": {
      "overall_status": "accept",
      "reason": "The final version preserves the original goal, makes
    the needed nonzero denominator assumption explicit, is
    self-contained, and remains theorem-style. Boundary case check:
    even for $n = 1$, $\hat{H}_{k+1} = 1 - (s_k y_k)/(y_k s_k) = 0$, so singularity still
    holds.",
      "issues": []
    }
  },
  {
    "case": "revise / missing_assumption",
    "source_idx": "Exercise 14.8",
    "source": "book/Numerical Optimization_Chp14",
    "problem": "Prove that the coefficient matrix in (14.16) is
    nonsingular if and only if $A$ has full row rank.",
    "problem_finally": "Let $m, n \in \mathbb{N}$, let $A \in \mathbb{R}^{m \times n}$, and let $X, S \in \mathbb{R}^{n \times n}$
    be diagonal matrices. Prove that the block matrix
    $[[0, A^T, I], [A, 0, 0], [S, 0, X]]$ is nonsingular if and only if $A$ has full
    row rank.",
    "semantic_report": {
      "overall_status": "revise",
      "reason": "The final statement drops the interior-point context
    assumption that $X$ and $S$ have positive/nonzero diagonal entries.
    As written it is false: if $A$ has full row rank but $X = 0$ and $S = 0$
    , the block matrix is singular.",
      "true_issue": "missing_assumption",
```

```
      "suggestion_fix": "Add that X and S are diagonal matrices with
    strictly positive diagonal entries, equivalently invertible
    diagonal matrices."
     }
  },
  {
    "case": "revise / task_drift",
    "source_idx": "Exercise 7.7",
    "source": "book/Numerical Optimization_Chp7",
    "problem": "Minimum surface problem. Divide each edge of the unit
    square into q intervals, label the (q+1)^2 grid points row by row,
    approximate derivatives by finite differences, and show that f_j
    has the displayed form involving (x_j - x_{j+q+1})^2 and (x_{j+1} - x_{j+q})^2.",
    "problem_finally": "Let q ≥ 1. Divide [0,1]^2 into q × q subsquares.
    Label the (q+1)^2 grid points row by row. For each subsquare A_j,
    indexed by its lower-left vertex, let its four vertices
    correspond to x_j, x_{j+1}, x_{j+q}, x_{j+q+1}. Show the approximate area
    contribution formula and determine the interior variables
    minimizing ∑_j f_j(x).",
    "semantic_report": {
      "overall_status": "revise",
      "reason": "The final version changes the subsquare vertex
    indexing. With row-by-row indexing on a grid of width q+1, the
    vertical offset should be q+1, not q. This is a substantive
    mathematical drift and conflicts with the formula.",
      "true_issue": "task_drift",
      "suggestion_fix": "Use vertices x_j, x_{j+1}, x_{j+q+1}, x_{j+q+2}
    consistently, or otherwise repair the indexing."
     }
  },
  {
    "case": "hold / missing_dependency",
    "source_idx": "Exercise 7.3",
    "source": "book/Numerical Optimization_Chp7",
    "problem": "Derive the formula (7.23) under the assumption that
    line searches are exact.",
    "problem_finally": "Let f : R^n → R be differentiable. For each
    iteration k, let x_{k+1}, p_k, y_k, and H_{k+1} be such that the
    expressions are defined. Assume exact line search, so
    ∇f(x_{k+1})^T p_k = 0, and assume y_k^T p_k ≠ 0. Derive
    p_{k+1} = -H_{k+1}∇f(x_{k+1}) = -∇f(x_{k+1}) + ...p_k.",
    "semantic_report": {
      "overall_status": "hold",
      "hold_reason_type": "missing_dependency",
      "reason": "The statement asks to derive formula (7.23) but does
    not give the defining relation for H_{k+1} or the algorithmic update
    context. Thus the conclusion does not follow from the stated
    assumptions.",
      "suggestion_fix": "Restate the missing update relation or prior
    formula defining H_{k+1} and p_{k+1}."
     }
  },
  {
    "case": "hold / unsuitable_lean",
    "source_idx": "Exercise 18.5",
    "source": "book/Numerical Optimization_Chp18",
    "problem": "Consider the constraint x_1^2 + x_2^2 = 1. Write the
    linearized constraints (18.7b) at the points (0,0)^T, (0,1)^T,
    (0.1, 0.02)^T, and -(0.1, 0.02)^T.",
    "problem_finally": "Let c : R^2 → R be c(x) = x_1^2 + x_2^2 - 1. For a point x_k
    , define c_k = c(x_k) and A_k = ∇c(x_k)^T. The linearized equality
    constraint is A_k p + c_k = 0. Write this constraint explicitly at the
    four listed points.",
    "semantic_report": {
      "overall_status": "hold",
      "hold_reason_type": "unsuitable_lean",
      "reason": "The statement is self-contained and faithful, but
    the task is phrased as a compute/write-out exercise rather than a
    theorem-style proposition suitable for Lean formalization.",
```

```
      "suggestion_fix": "Rewrite as explicit theorem statements
    giving the four resulting linearized constraints."
     }
  }
]
```

Discussion. These examples show how the semantic reviewer separates faithful, self-contained targets from cases requiring repair or exclusion. The **revise** cases are locally repairable, while the **hold** cases require missing dependencies or task-form issues to be resolved before formalization.

F.9 Example: Constrained Repair and Re-review

This example shows two representative repair cases. The first is a local assumption-restoration case that remains in the `revise` path. The second begins as a `hold` case because the task form is not yet suitable for Lean and a missing definition issue must also be resolved before a precise theorem-style target can be stated.

F.9.1 Case 1: Revise / Missing Assumption

{
"source_idx": "Exercise 14.8",
"repair_type": "revise / missing_assumption",
"original_problem": "Exercise 14.8. Prove that the coefficient matrix in (14.16) is nonsingular if and only if A has full row rank.",
"before_repair_problem_finally": "Let $m, n \in \mathbf{N}$, let $A \in \mathbf{R}^{m\times n}$, and let $X, S \in \mathbf{R}^{n\times n}$ be diagonal matrices. Prove that the block matrix

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix}$$

is nonsingular if and only if A has full row rank.",
"repair_reason": "The original context needs X and S to be positive/invertible diagonal matrices. Without this, e.g. $X = 0$ and $S = 0$ gives a singular block matrix even when A has full row rank.",
"after_repair_problem_finally": "Let $m, n \in \mathbf{N}$, let $A \in \mathbf{R}^{m\times n}$, and let $X, S \in \mathbf{R}^{n\times n}$ be diagonal matrices with strictly positive diagonal entries. Prove that the block matrix

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix}$$

is nonsingular if and only if A has full row rank."
}

F.9.2 Case 2: Hold / Unsuitable Lean and Missing Definition

{
"source_idx": "Exercise 7.4",
"repair_type": "hold / unsuitable_lean + missing_def",
"original_problem": "Exercise 7.4. Consider limited-memory SR1 updating based on (7.30). Explain how the storage can be cut in half if the basic matrix B_0^k is kept fixed for all k. (Hint: Consider the matrix $Q_k = [q_0, \ldots, q_{k-1}] = Y_k - B_0 S_k$.)",
"before_repair_problem_finally": "Consider the limited-memory SR1 update with memory parameter $m \in \mathbf{N}$. For iteration k, let $p = \min\{k, m\}$, and let the stored quasi-Newton pairs be $s_\ell, y_\ell \in \mathbf{R}^n$. Define $S_k = [\, s_{k-m}, \ldots, s_{k-1}\,] \in \mathbf{R}^{n\times p}$ and $Y_k = [\, y_{k-m}, \ldots, y_{k-1}\,] \in \mathbf{R}^{n\times p}$, and define the strictly lower-triangular matrix L_k by

$$(L_k)_{i,j} = \begin{cases} (s_{k-m-1+i})^T y_{k-m-1+j}, & i > j, \\ 0, & i \le j. \end{cases}$$

Assume that the basic matrix is fixed for all k, so $B_0^k = B_0 \in \mathbf{R}^{n\times n}$ for every k. Define $Q_k = [q_0, \ldots, q_{p-1}] = Y_k - B_0 S_k$, with $q_i = y_i - B_0 s_i$ for each stored pair. Explain how the storage can be reduced by one-half under the assumption that B_0^k is fixed for all k.",

"repair_reason": "Before repair, the statement remains open-ended because it asks the reader to explain how the storage reduction works, which is not suitable as a Lean theorem target. In addition, the column indices q_i of Q_k are inconsistent with the storage-window indexing used for S_k and Y_k.",

"after_repair_problem_finally": "Let $m \in \mathbf{N}$, let $k \in \mathbf{N}$, and let $p = \min\{k, m\}$. For the stored quasi-Newton pairs $(s_j, y_j) \in \mathbf{R}^n \times \mathbf{R}^n$, define $S_k = [s_{k-p}, \ldots, s_{k-1}] \in \mathbf{R}^{n\times p}$ and $Y_k = [y_{k-p}, \ldots, y_{k-1}] \in \mathbf{R}^{n\times p}$. Assume the basic matrix is fixed, i.e. $B_0^k = B_0 \in \mathbf{R}^{n\times n}$ for all k. For $i = 0, \ldots, p-1$, define

$$q_i := y_{k-p+i} - B_0 s_{k-p+i},$$

and let

$$Q_k = [q_0, \ldots, q_{p-1}] = Y_k - B_0 S_k.$$

Define the strictly lower-triangular matrix $L_k \in \mathbf{R}^{p\times p}$ by

$$(L_k)_{i,j} = \begin{cases} s_{k-p+i}^T y_{k-p+j}, & i > j, \\ 0, & i \le j, \end{cases} \qquad (i, j = 0, \ldots, p-1).$$

Prove that for all $i > j$,

$$(L_k)_{i,j} = s_{k-p+i}^T q_j + s_{k-p+i}^T B_0 s_{k-p+j}.$$

Hence, when B_0 is fixed, the information needed to form L_k can be recovered from S_k, Q_k, and B_0, so it is unnecessary to store both Y_k and Q_k; equivalently, one may replace storage of the pair (S_k, Y_k) by storage of (S_k, Q_k), which cuts the storage for the update pairs in half."

}

Discussion. These two cases illustrate the intended boundary of constrained repair. In the first case, the reviewer identifies a single missing assumption, and the repair restores the positivity condition needed for the original equivalence to be correct. In the second case, the original target is not merely incomplete but also unsuitable for direct formalization: the open-ended instruction must be converted into a precise theorem-style statement, and the indexing of the stored quantities must be aligned before re-review can succeed.

F.10 Examples: Representative formalization failures

Library/API grounding. For example, in a convexity target about monotonicity of gradients, the final snapshot contains:

```
theorem gradient_monotone_of_convex_differentiable
    {n : }
    {f : EuclideanSpace  (Fin n)  }
    (hconv : ConvexOn  Set.univ f)
    (hfdiff : Differentiable  f) :
    MonotoneMap (fun x => gradient f x) := by
  intro x y
  simpa [gradient, sub_eq_add_neg, inner_add_left, inner_add_right,
    inner_neg_left, inner_neg_right]
    using hconv.monotoneOn_gradient hfdiff x (by simp) y (by simp)
```

The intended mathematical route is reasonable: convex differentiable functions have monotone gradients. However, the proof is not grounded in actual Lean/Mathlib declarations. The compiler reports an invalid-field error for `monotoneOn_gradient`.

Formal representation and type discipline. Matrix-analysis targets often expose the gap between informal linear algebra notation and Lean's exact representation of dimensions, indices, rank, and positive-definiteness predicates.

```
theorem full_column_rank_iff_transpose_mul_self_posDef
    {m n : Type*} [Fintype m] [Fintype n] [DecidableEq m] [DecidableEq n]
    (h : Fintype.card n  Fintype.card m) (J : Matrix m n ) :
    J.rank = Fintype.card n
```

```
      Matrix.PosDef (J.transpose * J) := by
  simpa [Matrix.posDef_iff]
    using Matrix.rank_transpose_mul_self_eq_card_iff_posDef
      (m := m) (n := n) J
```

This proof mirrors the standard informal theorem that full column rank is equivalent to positive definiteness of $J^\top J$. The Lean attempt fails because the required representation bridge is not available under these names: the compiler reports unknown constants such as `Matrix.posDef_iff` and `Matrix.rank_transpose_mul_self_eq_card_iff_posDef`. The failure is therefore not just a missing algebraic idea; it reflects the need to align the informal matrix theorem with Mathlib's actual rank API, matrix index types, and positive-definiteness definitions.

Proof decomposition and long-horizon control. Agentic systems also fail when a proof requires a long chain of auxiliary analytic lemmas. In an Aristotle run on `problem-77.lean`, the target is Q-linear convergence of Newton's method at a multiple root:

```
theorem classicalNewtonIteration_multipleRoot_qLinear
    (N : ClassicalNewtonIteration)
    (hroot : IsMultipleRoot N.f 0)
    (hf'_correct :  x : , N.f' x = deriv N.f x)
    (hC1 :  s : Set , s   0  ContDiffOn  1 N.f s) :
    IsQLinearlyConvergentTo N.xSeq 0 := by
  sorry
```

The run did make structured progress: its summary reports helper lemmas such as `f_zero_of_multipleRoot`, `hasDerivAt_zero_of_multipleRoot`, `seq_ne_zero`, `newton_step_eq`, and `newton_contraction_ratio`. The remaining gap was the analytic lemma `newton_ratio_tendsto`, which must show that

$$x_k^{m-1} g(x_k)/f'(x_k) \to 1/m$$

along the Newton sequence. This is not a local syntax error: closing the proof requires maintaining a convergence argument through multiple helper lemmas, derivative facts, and asymptotic estimates. It is representative of long-horizon proof-control failures on optimization and numerical-analysis targets.

G Prompt Templates

This appendix lists the prompt templates used in the benchmark construction and evaluation pipeline. The prompts are organized in the same order as the pipeline: raw exercise cleaning, self-contained target construction, source-level review, constrained repair, JSON-to-Lean translation, and pass@k proof generation.

G.1 Raw-to-Clean Prompt

This prompt is used to construct the `problem_clean` field from the raw extracted exercise. Its purpose is to remove non-essential hints, remarks, background narrative, and implementation-oriented text while preserving the mathematical statement and all meaningful references.

Prompt template.

```
You are a preprocessing cleaner for optimization and mathematics
    exercise statements.

Role.
Your task is to remove hint/remark prompt-like information and
    background narrative from the original problem statement, while
    keeping all valid mathematical information unchanged, keeping all
    tags unchanged, and cleaning the original problem text only.

Input.
A single JSON object with a field problem containing raw exercise
    text.

Rules.
```

```
1. Preserve the mathematical meaning exactly.
2. Preserve symbols, formulas, numbering, and mathematically
    meaningful references.
3. Do not delete, alter, renumber, or rewrite any referenced
    identifier or index, including equation numbers, theorem numbers,
    algorithm numbers, exercise numbers, part labels, and
    cross-references such as "as in (4.13)", "part (b)", "Theorem
    12.1", or "Algorithm 3.2".
4. Preserve referential links exactly: if a clause refers to an
    earlier object by number or label, keep that number or label
    unchanged and attached to the same object.
5. Do not change the task intent, target, or scope. For example, do
    not convert "prove" to "compute", do not replace "minimum and
    maximum" with only "minimum", and do not weaken or strengthen
    quantifiers or assumptions.
6. Cleaning only: remove noise, wrappers, and non-mathematical
    exposition, but do not rewrite core mathematical content into a
    different statement.
7. Do not introduce new claims, examples, explanations, or solution
    steps.
8. Mandatory boundary-case safeguard: before treating a cleaned
    statement as acceptable, explicitly check edge cases such as
    nonemptiness and zero-valued conditions. If an edge case fails,
    construct a concrete counterexample and treat the item as needing
    revise with a clear reason in downstream review.

You must always produce a cleaned version that is shorter or at least
    meaningfully rewritten compared to the input. Never return the
    input text unchanged.

Primary objective.
Keep only mathematically essential content. Delete all irrelevant
    exposition. The cleaned result should read like a compact
    mathematical problem statement, not like textbook narration,
    software instructions, or pedagogical commentary.

Lean-oriented principle.
Treat the input as material to be normalized before formalization.
Prefer content that can later become:
1. definitions;
2. hypotheses;
3. variables and domains;
4. assumptions and constraints;
5. equations and inequalities;
6. precise goals and conclusions.

Delete anything that does not help identify or formalize these items.
    Default to deletion.

Keep only the following content:
1. exercise numbers, part labels, and mathematically meaningful
    cross-references such as 4.13, (a), Theorem X.Y, Algorithm X.Y,
    equation (15), or part (b);
2. definitions, notation, symbol declarations, and domain
    declarations;
3. assumptions, hypotheses, regularity conditions, feasibility
    assumptions, and rank or positivity assumptions;
4. optimization variables, objectives, constraints, equalities,
    inequalities, quantifiers, and formulas;
5. dual problems, KKT conditions, conjugates, SDP/QP/SOCP/LP
    formulations, feasibility claims, and exact statements of derived
    reformulations when they are part of the problem;
6. explicit problem data needed for the statement, including
    matrices, vectors, dimensions, constants, tolerances, and named
    data files when required to define the task;
7. the actual task directives, such as show, prove, derive, find,
    compute, determine, formulate, verify, construct, characterize,
    or solve;
8. minimal clarifying text only if removing it would destroy
    mathematical meaning.

Delete aggressively:
```

```
1. motivational background, applications, interpretation, intuition,
    historical remarks, and storytelling, including introductory
    sentences that describe why a topic is useful or interesting;
2. audience-directed wording such as "you may", "you will", "you
    should", "you need", "your job is", "we seek", "we consider", "we
    are interested in", "as a courtesy", "answer the following", "you
    should feel free to", and similar phrases;
3. all Hint, Hints, Remark, Remarks, Note, and Notes sections as
    labeled discourse, including multi-language hint blocks such as
    Matlab/Python/Julia subsections inside a Hints section;
4. solution strategies, method suggestions, and pedagogical advice,
    including phrases such as "you can do this by", "one approach is
    to", "a standard method", or "a typical method";
5. explanatory prose that merely paraphrases formulas already present;
6. software-tutorial language such as "Hello World", "how to use
    CVX", "tricks for using CVX", or similar presentation wording;
7. MATLAB, Python, Julia, CVX, CVXPY, Convex.jl, solver, and
    implementation instructions, except for mathematical data or
    tolerances that are part of the problem statement;
8. code snippets, random-seed commands, API usage directions, and
    multi-language code blocks, unless the exact numeric instance
    defined there is essential problem data;
9. comments such as "it is easy to see", "roughly speaking", "you can
    think of", "in a few very simple cases", "evidently", "of
    course", and similar non-mathematical narration;
10. all plotting, graphing, figure-generation, and visualization
    instructions, such as "Plot X vs Y", "Generate a figure", or
    "Graph the trajectory". These are presentation tasks, not
    mathematical content for formalization;
11. comparison-to-baseline instructions such as "compare it to the
    non-hybrid version" or "compare with the naive method", unless
    the comparison itself is a mathematical claim to prove;
12. "Briefly explain why" and "explain your reasoning" wrappers. Keep
    only the underlying mathematical task, such as "Determine
    whether..." or "Show that...".

Mandatory treatment of Hint and Remark.
Delete the labels Hint, Hints, Remark, Remarks, Note, and Notes from
    the output.
Do not preserve hint or remark paragraphs as paragraphs.
If a Hint, Remark, or Note contains indispensable mathematical
    content, such as a formula, inequality, or fact needed to state
    the problem, extract only that mathematical content and merge it
    into the cleaned problem statement without the discourse wrapper.
If a Hint or Remark section consists entirely of software
    instructions, such as MATLAB code, Python snippets, Julia
    variable declarations, or CVX API directions, delete the entire
    section with no extraction.
```

G.2 Self-Contained Target Construction Prompt

This prompt is used to construct `problem_finally` from `problem_standardized_math`, with `problem_with_context` used only for explicitly recovered dependencies. Its purpose is to produce a condition-complete, self-contained informal target before Lean formalization.

Prompt template.

```
You are given one optimization exercise statement.
Rewrite it into a condition-complete, self-contained, textbook-style
    problem.

Input use rule.
- Primary source: problem_standardized_math. Preserve its task
    intent, core assumptions, and structure first.
- Secondary source: problem_with_context. Use it only to supplement
    missing dependency assumptions or formulas, including Reference,
    Double, Third, or More reference lines, when clearly relevant.
- Before all other processing, ignore any hint/hints content
    entirely; hint text is disposable and must not be used in
    rewriting.
```

```
- If problem_standardized_math and problem_with_context differ,
    prefer problem_standardized_math unless context provides explicit
    supporting mathematics needed for completeness.
- If problem_standardized_math lacks necessary mathematical support,
    such as assumptions, formulas, or definitions, explicitly
    retrieve and integrate that support from problem_with_context.
- Keep strict dependency folding: third-hop to second-hop, second-hop
    to first-hop, and first-hop to the original problem. If
    more-level dependencies exist, unfold and fold them
    layer-by-layer before final rewrite.
- Carefully check that every concept, symbol, function, and operator
    used in the problem is defined. If any definitions are missing,
    add precise definitions so the problem is fully self-contained
    and suitable for formalization.

Hard constraints.
- Do not solve the problem.
- Do not use, preserve, paraphrase, or output any hint/hints text;
    discard hints directly.
- Do not rewrite, split, reorder, or restate the original task
    objective in any form.
- The task objective must remain strictly identical to
    problem_standardized_math.
- Only identify and explicitly add implicit assumptions or conditions
    that are directly supported by input context.
- Add only information that is genuinely useful for mathematical
    completeness, dependency resolution, or precise definition; do
    not add irrelevant background, commentary, examples, or
    decorative exposition.
- Keep the original mathematical intent and task type.
- Do not add unnecessary stronger assumptions.
```

G.3 Source-Level Review Prompt

This prompt is used to review each `problem_finally` against the original `problem`. The reviewer checks mathematical faithfulness, self-containedness, missing assumptions, unresolved dependencies, and suitability for Lean formalization.

Prompt template.

```
Role.
You are reviewing a mathematical exercise for Lean formalization
    readiness.

Your task is to determine whether the candidate final version
    preserves the original mathematical meaning, includes all
    necessary assumptions, is fully self-contained, and is suitable
    for Lean formalization.

You are not asked to improve wording for style alone. Focus on
    mathematical faithfulness, completeness, self-containedness, and
    formalizability.

Before deciding accept, you must explicitly check boundary cases such
    as nonemptiness and zero-valued edge conditions. If a boundary
    case fails, provide a concrete counterexample.

Core objective.
Compare problem_finally against problem.

Your review must answer four questions:
1. Does problem_finally preserve all assumptions from the original
    problem?
2. Does problem_finally preserve the original task and conclusion
    without drift?
3. Is problem_finally fully self-contained?
4. Is the resulting task suitable for Lean formalization?

Review scope.
You must check:
```

```
- whether any assumption has been lost;
- whether any assumption has been changed;
- whether any quantifier or logical direction has changed;
- whether any domain restriction has changed;
- whether any conclusion has been dropped;
- whether any conclusion has been added;
- whether any required dependency is missing;
- whether any technical term or function is used without definition;
- whether the task is actually formalizable in Lean.

Detailed review criteria.

1. Assumption preservation.
Check whether problem_finally is missing any assumption that is
    explicit or implicit in problem. Examples include nonzero
    assumptions, nonnegative assumptions, positivity assumptions,
    positive-definite assumptions, domain restrictions, index range
    restrictions, quantifier scope restrictions, existence
    conditions, uniqueness conditions, continuity, differentiability,
    integrability, measurability, boundedness, invertibility, and
    other regularity or well-posedness conditions. If a necessary
    assumption from problem is absent in problem_finally, flag
    missing_assumption.

2. Task preservation.
Check whether the mathematical task in problem_finally is still the
    same as in problem. Pay special attention to non-equivalent
    drift, including changing exists to forall, changing equivalence
    to one-way implication, dropping one direction of a two-sided
    result, changing the domain or scope of variables, weakening or
    strengthening the conclusion, removing a conclusion, or adding a
    new conclusion. Equivalent reformulations are acceptable only
    when they are genuinely mathematically equivalent. If the final
    version changes the task in a non-equivalent way, flag task_drift.

3. Self-containedness.
Check whether problem_finally can be understood and formalized on its
    own. Flag missing_dependency if the statement relies on something
    not actually included or specified, such as a missing tag or
    unresolved reference, a missing theorem, lemma, proposition,
    corollary, or algorithm, a missing dependency on another exercise
    or previous result, or a required cited fact that is not stated
    clearly enough to use. When judging missing_dependency, read the
    full text carefully and confirm that the dependency is genuinely
    absent, rather than merely named while its relevant mathematical
    content has already been restated in the problem itself. Also
    flag missing_dependency if problem_finally refers to a tag,
    Theorem X.Y, Alg. X.Y, Algorithm X.Y, part (b), or another
    exercise-internal reference, but does not explain or restate the
    referenced content clearly enough for the statement to stand on
    its own.

4. Missing definitions.
Check whether problem_finally uses a specialized, uncommon, or
    technical term without defining it. Also flag this issue if a
    function, operator, mapping, object, or construction is used
    without a definition precise enough for formalization. Examples
    include uncommon terms or symbols such as ill-conditioned,
    L(x,y), or similarly specialized notation introduced without any
    definition. If this happens, flag missing_def.

5. Formalizability.
Judge whether the task is suitable for Lean formalization. Usually
    suitable: proof-oriented mathematical statements, precisely
    stated claims with explicit mathematical objects and conclusions,
    and mathematically precise computational statements. Usually not
    suitable: give a description, formulate, compute, plot, sketch,
    discuss informally, open-ended explanatory tasks, or tasks whose
    result is mainly visual, rhetorical, or pedagogical rather than
    formally mathematical. If problem_finally contains verbs such as
    formulate, compute, or similar wording that is not suitable for
    Lean formalization as a precise theorem-style target, judge the
    task as not suitable for Lean formalization and choose hold. If
```

```
    the task itself is not suitable for Lean formalization, the
    overall status should be hold.
```

G.4 Constrained Repair and Re-review Prompt

This prompt is used when the source-level reviewer classifies an instance as repairable. The repair step is constrained to modify only `problem_finally`; earlier fields remain fixed as audit anchors.

Prompt template.

```
Your task is to rewrite only problem_finally, using the review result
    and the provided fix suggestions, so that the final statement is
    mathematically faithful to the original task, self-contained, and
    easier to formalize in Lean.

You must read the input very carefully. You will be audited after
    your rewrite.
Be precise, conservative, and faithful to the given instructions.

Rewrite problem_finally according to the repair suggestions.

Your goal is to produce a corrected problem_finally that:
1. preserves the original mathematical meaning of problem;
2. follows the correction instructions in value_true_issues;
3. remains consistent with problem_clean and
    problem_standardized_math;
4. is self-contained, mathematically precise, and suitable for Lean
    formalization.

You must rewrite problem_finally.
You must not change any other field.

Do not output overall_status, reason, or value_true_issues.
Do not introduce any new correction that is not required by the
    provided review reasons and suggestions.
Do not add extra assumptions, conclusions, definitions, or
    reformulations beyond what is needed to carry out the stated
    repair.
Do not improve style for its own sake.

You must read reason and value_true_issues carefully.
Use them as the direct authority for what needs to be fixed.

Add back the missing assumption exactly as needed, without changing
    the mathematical task.
Rewrite problem_finally so that it matches the original task in
    problem again.
Restore the correct variables, domain, quantifiers, logical
    direction, and conclusion.
Add the missing definition or make the relevant notion explicit, but
    only to the extent required by the repair suggestion.
Apply all repair suggestions together, but keep the final statement
    minimal and faithful.

Priority of authority:
1. problem is the source of truth for the original mathematical task.
2. value_true_issues and reason tell you exactly what must be
    repaired.
3. problem_standardized_math helps preserve the already standardized
    mathematical structure.
4. problem_clean helps remove irrelevant textbook narrative.
5. problem_with_context may be consulted only when needed to resolve
    ambiguity, not to invent new content.

If these sources conflict, preserve the original mathematical intent
    of problem and follow the explicit repair instructions.

Do not:
- modify problem, problem_clean, or problem_standardized_math;
```

```
- add new assumptions not justified by the repair feedback;
- weaken the conclusion;
- strengthen the conclusion;
- introduce a new proof strategy;
- insert hints or remarks;
- output analysis, commentary, or explanation outside the JSON.

All fields except problem_finally must remain unchanged from the
    input.

Before you output, verify all of the following:
1. Only problem_finally has been changed.
2. The rewrite follows the provided repair suggestions.
3. The rewritten problem_finally does not introduce any extra content
    beyond the required repair.
4. The rewritten problem_finally is self-contained and suitable for
    Lean formalization.
5. The output is valid JSON.
6. Mandatory boundary-case safeguard: explicitly check edge
    conditions such as nonemptiness and zero-valued cases for the
    repaired statement. If an edge case fails, produce a concrete
    counterexample in your verification and treat the item as
    requiring revise with a clear reason in downstream review.

Your job is not to generate a better problem in general.
Your job is to repair problem_finally, and only problem_finally, so
    that it correctly reflects the original task and the given review
    suggestions.
```

G.5 Compact Translator Prompt

This prompt is used for the initial JSON-to-Lean translation stage. It translates one semantic block at a time into a Lean declaration while reusing previously accepted declarations from the frozen context.

Prompt template.

```
You are a JSON-to-Lean block translator.

Input.
- current JSON semantic block with fields such as index, source,
    source_idx, kind, term, content, and problem;
- section_context: previously accepted Lean declarations in the same
    namespace;
- optional mcp_context or external source context.

Task.
Translate exactly one JSON semantic block into one Lean 4
    declaration, plus helper declarations only when necessary. Reuse
    existing declarations from section_context whenever possible.
    Output only the Lean code fragment for the current block.

Use semantic content in this priority order:
1. content;
2. problem;
3. term, only as a naming or contextual hint.

Declaration rules by kind:
- defn: produce one source-faithful Lean def. Do not use sorry or
    vacuous bodies.
- thm: produce exactly one theorem as the primary declaration. Keep
    all source assumptions explicit. Use a concise, semantic,
    Lean-safe name. The proof body must be exactly "by sorry".
- algo: produce one structure encoding source-grounded algorithmic
    state, parameters, and invariants. Do not encode theorem
    conclusions as fields.
- opt_prob: produce one structure encoding decision variables,
    objectives, constraints, and source-grounded optimization
    components.
```

```
Context reuse rules.
Before generating code, scan section_context. Reuse matching
    definitions, structures, theorem names, notation, helper
    predicates, and directives. Do not redefine equivalent symbols
    unless the meaning differs.

Lean and safety constraints:
- Output only Lean code for insertion inside the shared namespace.
- Do not output imports, namespace wrappers, closing "end"s, block
    comments, Markdown fences, JSON/YAML, or explanations.
- Do not introduce vacuous placeholders such as "def X := True",
    empty Prop wrappers, or meaningless predicates.
- Do not use axiom, admit, unsafe, opaque, or fake mathematical
    content.
- For theorem blocks, the only allowed unfinished proof is "by sorry".
- Preserve source assumptions, equalities, relations, constructions,
    and boundary conditions.
- Do not weaken or strengthen the source statement except minimally
    to obtain a well-typed Lean formalization.
- Add open, open scoped, variable, or set_option directives only when
    needed and not already provided by section_context.
- Prefer Mathlib-native notation and fully qualified names when they
    avoid unnecessary directives.
- Use "fun" instead of lambda symbols.
- In structures, put each field on its own line.
- For Euclidean inner products, use the notation supported by the
    current Lean environment; do not add unsupported scalar-field
    subscripts.

Final check.
The output must preserve the source intent, reuse available context,
    match the required declaration shape for the block kind, avoid
    placeholders, and contain only the Lean fragment for the current
    block.
```

G.6 PASS@k PROOF-GENERATION PROMPTS

These prompts are used in the non-agentic pass@k experiments. Each model receives a Lean block with one designated proof placeholder, `FILL_PROOF_HERE`, and returns only the proof body that replaces it. Other `sorry` tokens in the template are left untouched.

G.6.1 SYSTEM PROMPT

Prompt template.

```
You are a Lean 4 proof assistant. Your task is to fill in exactly one
    proof placeholder.

Rules.
- Output only the Lean proof term or tactic body that should replace
    the /- FILL_PROOF_HERE -/ marker.
- Do not output the surrounding theorem, lemma, or def block.
- Do not output thinking, reasoning, chain-of-thought, analysis, or
    any hidden deliberation; output the code directly.
- Do not output imports, namespaces, sections, declarations,
    comments, explanations, or Markdown outside one Lean code block.
- The code you output will be inserted verbatim at the marker. It
    must fit exactly at that position.
- Do not modify, repeat, comment out, delete, rename, weaken, or
    otherwise alter the original theorem, lemma, or def statement,
    hypotheses, target, imports, namespace, or surrounding code.
- Do not touch any other sorry tokens already present in the
    template; they belong to later steps.
- Do not add or use axiom, admit, unsafe, opaque, or
    proof-search/suggestion commands such as apply?, exact?, rw?,
    simp?, aesop?, solve_by_elim?, or library_search.
- Never use sorry.
- Before using any lemma, theorem, or definition name, verify it is
    resolvable in the current context.
- Never guess or invent lemma names based on similarity.
```

```
- If a needed lemma is not resolvable from the search results, either
    switch to available lemmas or define a helper first without
    sorry, axiom, admit, unsafe, or opaque.
- Wrap your code in a Lean code block.
```

G.6.2 First-Turn User Prompt

Prompt template.

```
Here is a Lean 4 block containing a proof placeholder /-
    FILL_PROOF_HERE -/ and possibly other sorry tokens, which you
    must leave untouched.

Lean block.

{block_code}

Available tool/search context.

{mcp_sections}

Output only the Lean proof term or tactic body that should replace /-
    FILL_PROOF_HERE -/.

Do not output thinking, reasoning, chain-of-thought, analysis, or any
    hidden deliberation; output the code directly.
Do not output the theorem, lemma, or def block itself.
The proof body will be inserted verbatim at the marker.

If the required lemma or definition is not found in the search
    results, define it yourself first before using it.
Use exact identifier names from searchable context; do not use
    similar-looking names that are not resolvable.

Do not use sorry, axiom, admit, unsafe, opaque, apply?, exact?, rw?,
    simp?, aesop?, solve_by_elim?, or library_search.
Return only the proof body in a Lean code block.
```

G.6.3 Retry-Turn User Prompt

Prompt template.

```
Your previous proof body produced compile errors.
Here is the original template, your last proof body, and the compiler
    diagnostics.

Original template with placeholder.

{block_template}

Your previous proof body.

{prev_code}

Compile errors.

{compile_errors}

Available tool/search context.

{mcp_sections}

Output only the corrected Lean proof term or tactic body that should
    replace /- FILL_PROOF_HERE -/ in the original template.

Do not output thinking, reasoning, chain-of-thought, analysis, or any
    hidden deliberation; output the code directly.
Do not output the theorem, lemma, or def block itself.
The proof body will be inserted verbatim at the marker.
```

```
Keep any other sorry tokens unchanged by not mentioning them.

If the required lemma or definition is not found in the search
    results, define it yourself first before using it.
Before writing the final proof, ensure every new identifier you use
    is resolvable; otherwise revise the approach.

Do not use sorry, axiom, admit, unsafe, opaque, apply?, exact?, rw?,
    simp?, aesop?, solve_by_elim?, or library_search.
Do not add comments, explanations, or text outside the Lean code
    block.
Output only the corrected proof body in a Lean code block.
```

H LIMITATIONS

CAM-Bench has several limitations. First, although CAM-Bench targets computational and applied mathematics, its source material is still optimization-centric, with coverage mainly in optimization, numerical linear algebra, and numerical analysis. Other areas of applied mathematics, such as PDEs, control, and scientific computing, are not yet systematically covered.

Second, benchmark construction uses an LLM-assisted pipeline for dependency recovery, normalization, Lean formalization, and semantic review. We reduce this risk through source-grounded dependency expansion, constrained repair, Lean compilation, and semantic validation. However, subtle mismatches may still remain, including at boundary cases.

Finally, the reported comparison uses a fixed 2M-token limit per obligation. Future work can examine how success changes under larger budgets and broader applied mathematics coverage.

I BROADER IMPACTS

CAM-Bench aims to support reliable evaluation of mathematical reasoning systems by providing Lean-verified proof targets in computational and applied mathematics. By focusing on context-dependent textbook problems, it helps diagnose failures in assumption tracking, library grounding, type discipline, and long-horizon proof decomposition. The benchmark may encourage progress in formal applied mathematics, including stronger Mathlib infrastructure, more capable theorem-proving agents, and more faithful autoformalization pipelines. It also complements existing formal benchmarks that primarily emphasize Olympiad-style problems, algebraic domains, or isolated theorem statements. CAM-Bench should therefore be used as one component of a broader evaluation suite. Strong performance on this benchmark indicates progress in Lean-based proof generation for context-dependent applied mathematics.