LeanCat: A Lean Dataset for Evaluating Library-Grounded Category-Theoretic Reasoning

Rongge Xu [1] **Hui Dai** [2] **Yiming Fu** [3] **Jiedong Jiang** [4] **Tianjiao Nie** [5] **Junkai Wang** [1] **Holiverse Yang** [6] **Zhi-Hao Zhang** [7]

Abstract

Modern formal theorem proving increasingly depends on large formal libraries, motivating focused evaluation datasets for library-grounded reasoning in addition to established olympiad-style, undergraduate, and other formal reasoning tasks. Category theory is a natural domain for this evaluation because proofs often rely on abstract interfaces, universal properties, and reusable library constructions. We introduce **LeanCat**, a compact evaluation dataset of 100 statement-level category-theory tasks in Lean 4 and Mathlib. Each task includes a natural-language statement, a Lean-checkable formal theorem file, topic and difficulty labels, and source metadata. LeanCat evaluates how models navigate Mathlib interfaces and compose existing abstractions, including tasks that require local definitions or bridge lemmas beyond direct theorem lookup. We evaluate strong generalist models, specialized provers, and retrieval-augmented proving protocols across separate settings. The best static generalist baseline solves 12/100 tasks under controlled pass@4 evaluation, while retrieve–generate–verify protocols reach 24/100 overall and 1/40 High tasks in the best controlled setting, highlighting how formal grounding, retrieval, and interaction change measured performance on this focused benchmark.

[1]Yau Mathematical Sciences Center, Tsinghua University, Beijing, China [2]Iluvatar CoreX [3]Department of Mathematics, Southern University of Science and Technology [4]Westlake Institute for Advanced Study, Westlake University [5]Qiuzhen College, Tsinghua University [6]Department of Physics, The Chinese University of Hong Kong [7]Yanqi Lake Beijing Institute of Mathematical Sciences and Applications (BIMSA). Correspondence to: Rongge Xu <rongge@tsinghua.edu.cn>.

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

1. Introduction

Evaluation in formal theorem proving extends beyond filling in short proof scripts. Modern proof assistants are organized around large libraries, reusable interfaces, typeclass inference, and domain-specific encodings. In such environments, a prover must identify the right abstractions, retrieve relevant definitions and lemmas, and maintain a coherent proof plan across several layers of formal infrastructure.

Early Lean-based neural provers (Polu et al., 2022), large-scale proof-search systems such as AlphaProof (Hubert et al., 2025), and specialized provers such as DeepSeek-Prover-V2 (Ren et al., 2025) and Seed-Prover 1.5 (Chen et al., 2025) show that large-scale agentic RL, formal feedback, and test-time scaling can solve increasingly nontrivial Lean problems. These advances sharpen the evaluation question: which benchmarks test routine proof completion, and which test whether systems can operate reliably inside a mature mathematical library?

Existing formal-theorem-proving benchmarks such as miniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2022), PutnamBench (Tsoukalas et al., 2024), and FIMO (Liu et al., 2023) have been valuable for tracking progress on olympiad, undergraduate, and competition-style mathematics. Algebra-focused suites such as FATE (Jiang et al., 2025) have also moved evaluation toward richer formal domains. These benchmarks do not, however, directly target the kind of library-grounded abstraction that appears in much of modern formalized mathematics. Much of this mathematics is written at high generality, organized around reusable interfaces, and intertwined with large libraries of definitions and lemmas, so success depends less on a single tactic than on navigating abstraction and coherently composing library knowledge over longer horizons. Category theory is one natural domain for studying this regime: as an interface language for modern mathematics, its proofs often proceed by composing universal properties, naturality conditions, functoriality, adjunctions, limits, colimits, monads, and exactness structures, rather than by isolated symbolic manipulations.

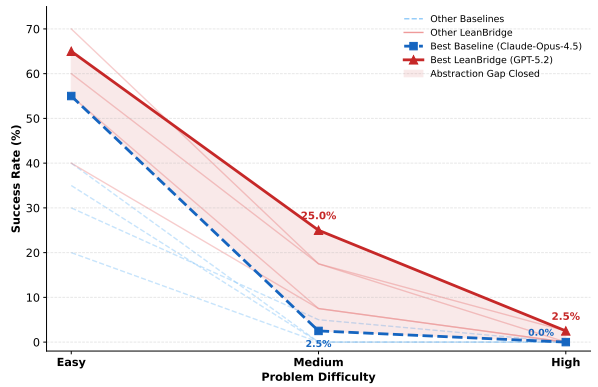

Figure 1. **LeanCat shows protocol-dependent performance on abstraction-heavy tasks.** Static pass@k baselines (dashed lines) degrade sharply across the Easy/Medium/High split, while the controlled LeanBridge protocol (solid lines) partly recovers measured success rates on harder tasks. Bold lines indicate the best-performing settings (Claude-Opus-4.5 for static pass@4, GPT-5.2 for LeanBridge with *NL+Statement*), while faint lines show the remaining evaluated models. Because the High split contains 40 tasks, its percentages are coarse-grained; for example, 2.5% corresponds to 1/40 solved tasks.

We introduce **LeanCat**[1], a Lean dataset of 100 category-theory tasks formalized in Lean 4 with Mathlib. Each item is a statement-level proving task: the theorem statement and imports are provided, but no scaffolded sequence of intermediate lemmas is given. This design makes LeanCat suitable for evaluating whether automated systems can work inside a mature formal library, rather than only solve decomposed proof exercises. Several tasks are standard facts whose difficulty lies in the formalization burden; others require local infrastructure not available as direct Mathlib statements. This mix is intentional: it probes whether a system can move between mathematical intent, library encodings, and proof engineering. LeanCat complements algebra-focused evaluation suites such as FATE by shifting the focus from algebraic manipulation toward category-theoretic interfaces and universal-property reasoning. Table 2 provides a compact comparison with prior evaluation datasets.

The contribution has two parts: a new Lean dataset, and baseline experiments that use this dataset to compare proof-generation protocols. Figure 1 summarizes the main empirical pattern: static sampling degrades sharply on Medium and High tasks, while retrieval-augmented refinement partly changes measured behavior. Under a controlled static pass@4 protocol, the best evaluated generalist baseline solves 12/100 tasks overall and 0/40 High tasks; under the controlled LeanBridge setting, retrieval-augmented refinement reaches 24/100 overall, including 1/40 High tasks. The results characterize protocol-level behavior rather than a single unified model ranking.

Our contributions are:

- **A Lean dataset for abstract formalization.** LeanCat contains 100 Lean 4 category-theory tasks across eight topic clusters, selected to probe interface-level reasoning and library navigation. A strict Mathlib-overlap audit finds that only 11/100 tasks have direct Mathlib counterparts, with none in the High split.
- **Curation and documentation.** The dataset combines expert problem selection, LLM-assisted statement drafting, human review, Mathlib overlap auditing, natural-language companion statements, and reproducibility metadata.
- **Baseline protocol experiments.** Using LeanCat, we evaluate static pass@k sampling, specialized provers, a controlled retrieve–generate–verify baseline, and a small exploratory autonomous-agent setting, making tool-access assumptions explicit.
- **Failure analysis.** We identify recurring failure modes, including hallucinated APIs, missing bridge lemmas, lazy fallback, mathematically invalid strategies, and semantic "hack" failures that preserve compilation while changing the intended meaning.

2. Dataset Design and Scope

2.1. What LeanCat Measures

LeanCat evaluates *library-grounded abstraction* in a concrete formal environment. Each item is a statement-level task: the system receives a target theorem statement and its imports, but not a scaffolded chain of intermediate hints. Operationally, this means that tasks require a system to:

- **Navigate** Mathlib's categorical interfaces and compose reusable structures such as functors, natural transformations, adjunctions, limits, colimits, monads, and exactness notions;
- **Bridge** textbook mathematical statements and Lean encodings, including by constructing local intermediate lemmas or definitions when direct lookup is insufficient;
- **Stay** within the provided formal setup rather than relying on redefinition or shadowing shortcuts that trivialize the goal.

This scope is narrower than general mathematical reasoning and broader than tactic-level syntax repair. The current

[1]Release artifacts: https://huggingface.co/datasets/scicraft/LeanCat and https://github.com/sciencraft/LeanCat.

Table 1. **LeanCat topic clusters.** The benchmark covers 1-category theory and emphasizes interface-level formalization.

Cluster	Main competencies tested
Basic category properties	morphisms, initial/terminal objects, idempotents
Adjunctions	unit/counit reasoning, comma categories, universal properties
Reflective/coreflective subcategories	subcategory structure and examples
Concrete categories	faithful functors, set-level encodings, examples
Limits and colimits	cones, preservation, lifting, universal properties
Cocompletions	new definitions and frontier formalization
Abelian categories	kernels, cokernels, exactness, homological structure
Monads	Kleisli and Eilenberg–Moore constructions

release is a compact dataset for one domain; the experiments in this paper use it to compare broad protocol regimes rather than to support fine-grained rankings from small split-level differences.

2.2. Content and Sources

LeanCat consists of 100 category-theory tasks, each provided as a Lean declaration with the required imports and local setup, paired with a companion natural-language statement and metadata. Problems are organized into the eight topic clusters shown in Table 1.

The sources are organized into three groups. The *abstract* part draws primarily on standard references in category theory, especially Riehl (2017) and Mac Lane (1998), supplemented by lecture notes (Kong, n.d.; Zheng, n.d.). The *concrete* part draws heavily on Adámek et al. (1990), which provides a systematic source of exercises on concretizability, injectivity, and related examples. A smaller *other* group contains problems inspired by research papers and advanced community-driven references (Chen, 2021; Adámek et al., 2021). The dataset intentionally mixes standard textbook facts with tasks that require nontrivial adaptation to Mathlib; in several High-difficulty cases, human formalizers needed to introduce intermediate definitions or bridge lemmas rather than simply locate a pre-existing theorem. Appendix A provides the topic ranges and intended competencies for the eight clusters.

The source design separates two forms of overlap. Many mathematical statements are adapted from public references, while the released tasks fix concrete Lean 4 and Mathlib encodings. The Mathlib overlap audit below focuses on direct formal counterparts in Mathlib; source metadata records the informal provenance of each task.

2.3. Curation Workflow

LeanCat was curated through a three-stage process. First, category-theory experts selected candidate problems to cover core categorical interfaces and representative proof patterns: diagram chasing, universal properties, naturality, preservation/lifting statements, and interactions between abstract and concrete categories. Second, several LLMs were used to draft candidate Lean statements; experts retained only semantically correct formal statements, and problems for which no model produced an acceptable statement were completed by Lean and category-theory specialists. Third, independent reviewers checked compilation, mathematical meaning, universe and typeclass assumptions, and consistency with Lean conventions. LLM drafts served as candidate encodings; final acceptance was based on expert semantic review. Each released Lean file is self-contained at the statement level: it imports Mathlib and explicitly includes any local definitions or lemmas required to state the theorem, and is paired with a companion Markdown statement supporting LaTeX notation so that evaluation protocols can distinguish formal-only from NL+Statement proving against the same verified target.

2.4. Difficulty Labels

The main split contains 20 Easy, 40 Medium, and 40 High tasks, derived from a hybrid score that combines human ratings with machine-facing evidence from baseline attempts. The pipeline has three parts: model-based scoring, independent human scoring, and final aggregation. For model-based scoring, each problem is attempted with the LeanBridge sequential-refinement protocol across the five generalist models, and per-task counts are aggregated over models: N_{proof} is the number of model runs that produce a verified proof, and N_{stmt} the number of remaining runs that produce a correct formal statement. We then compute

$$\text{Diff} = \max(0, 10 - 2N_{\text{proof}} - 0.5N_{\text{stmt}}).$$

In parallel, two raters with Lean and category-theory expertise independently scored each problem on the same 1–10 scale. The final hybrid score weights human and model-based ratings equally, then bins tasks into the 20/40/40 split. The split is intentionally Medium/High-heavy: LeanCat is intended to emphasize formalization pressure on current systems rather than to track pure mathematical difficulty.

The hybrid split measures current-system formalization hardness rather than only mathematical depth. Table 7 reports a human-only robustness check, with related audit and uncertainty tables collected in Appendix B. Under the human-only split, the static High-split result remains 0 solved tasks, while the best LeanBridge setting solves 2/38 High tasks. The Medium/High separation is weaker, reflecting the difference between human mathematical depth and current-system formalization hardness.

2.5. Mathlib Overlap Audit

To distinguish LeanCat from a filtered subset of existing Mathlib theorems, we audited the tasks against Mathlib v4.19.0. This audit supports the construct validity of the dataset: LeanCat is intended to measure library-grounded formalization, not merely recall of already named category-

Table 2. **Qualitative positioning against prior evaluation datasets.** Coarse low/medium/high labels summarize how centrally each dataset's evaluation depends on existing library infrastructure and on high-level abstract interfaces.

Dataset	Domain	Library reliance	Category theory	Abstraction-heavy
miniF2F	olympiad	low	no	no
ProofNet	undergraduate	medium	no	no
PutnamBench	competition	medium	no	no
FATE	algebra	high	no	moderate
LeanCat	category theory	high	yes	central

theory declarations. This distinction matters because direct-overlap items and adaptation-heavy items place different formalization demands on evaluated systems. A task was counted as a direct counterpart only when Mathlib already contained a theorem matching the LeanCat statement up to routine renaming or immediate specialization. Under this strict criterion, only 11/100 items have direct Mathlib counterparts: 8 Easy, 3 Medium, and 0 High. The remaining tasks require adaptation, local bridge lemmas, or new setup. The audit separates direct formal overlap in Mathlib from the source metadata used to document each task's mathematical provenance. Table 6 reports the audit in more detail.

2.6. Comparison with Prior Datasets

Table 2 gives a compact qualitative comparison between LeanCat and several prior Lean-oriented evaluation datasets, positioning LeanCat as a library-grounded benchmark for abstraction-heavy category-theoretic reasoning. The *Library reliance* column summarizes how centrally success depends on working inside an existing formal library, and the *Abstraction-heavy* column reflects how central high-level abstract interfaces and universal-property style reasoning are to the evaluation.

3. Dataset Documentation

LeanCat is submitted as a dataset release rather than only as a PDF result table. Complete release artifacts are available on Hugging Face at `https://huggingface.co/datasets/scicraft/LeanCat` and on GitHub at `https://github.com/sciencraft/LeanCat`. The release artifact contains:

- **Tasks:** Lean 4 files for all 100 formal theorem statements (with imports and local definitions) and companion Markdown files with natural-language problem descriptions;
- **Metadata:** per-task difficulty labels, topic labels, source annotations, and declaration fields;
- **Pinned environment:** lockfiles for Lean 4.19.0 and Mathlib v4.19.0;
- **Evaluation harness:** valid-proof criteria, prompt templates, evaluation configuration files, and lightweight drivers for the reported baselines;
- **Documentation:** dataset-card style documentation and license information.

LeanCat itself is the released artifact; the proof-generation protocols in this paper are accompanying baseline experiments that use it to study how tool access, retrieval, and interaction affect measured performance.

4. Evaluation Protocols

We evaluate LeanCat under several proof-generation protocols. The protocols are deliberately separated because they make different assumptions about available inputs, retrieval, compiler feedback, and test-time interaction. They should therefore be read as distinct evaluation settings rather than collapsed into a single overall score.

Static pass@k. Generalist models receive the Lean statement, necessary imports, and a prompt to generate a complete Lean proof. We evaluate GPT-5.2 (OpenAI, 2025), Gemini-3-Pro (Gemini Team, Google, 2025), Claude-Opus-4.5 (Anthropic, 2025), DeepSeek-V3.2-Thinking (Liu et al., 2025), and Kimi-K2 (Team et al., 2025). No natural-language hints, decomposed lemmas, retrieval, or compiler-feedback loop are provided. We report pass@1 and pass@4 using the standard pass@k criterion (Chen et al., 2021): a task is solved if at least one sampled attempt compiles and verifies. This protocol is intentionally constrained; it measures static proof generation from the formal statement, not the full capability of a tool-using agent.

Specialized prover pass@32. Specialized neural provers are evaluated under the same formal-statement-only, no-retrieval assumption but with a larger sampling budget, pass@32. We include DeepSeek-Prover-V2-671B (Ren et al., 2025), Goedel-Prover-v2-32B (Lin et al., 2025), StepFun-Prover-32B (Shang et al., 2025), and Kimina-72B (Wang et al., 2025). This tests whether domain-specific formal-proving models can solve LeanCat through independent sampling alone.

LeanBridge. LeanBridge is a controlled diagnostic baseline (Algorithm 1). It uses a retrieve–generate–verify loop with Mathlib retrieval through LeanExplore (Asher, 2025), whose retriever combines semantic embeddings, BM25+, and PageRank-style signals. The retrieval corpus is Mathlib v4.19.0: initial queries are formed from the natural-language task statement, later queries may be requested by the model after seeing compiler feedback, and retrieved context consists of relevant Mathlib declarations, theorem statements, definitions, and source/documentation snippets exposed by

Algorithm 1 LeanBridge diagnostic inference loop. Starting from the task statement, the agent alternates between proof generation, Lean verification, and optional Mathlib retrieval until it produces a valid proof or exhausts the iteration budget.

Input: natural-language statement S_{NL}, optional Lean statement S_{Lean}
Output: verified proof script π or failure
$C \leftarrow \{S_{NL}, S_{Lean}\}$
$C \leftarrow C \cup \mathrm{Retrieve}(S_{NL})$
for $t = 1$ **to** $T_{\max}$ **do**
  $a_t \leftarrow \mathrm{LLM}(C)$
  if a_t is a search request **then**
    $C \leftarrow C \cup \mathrm{Retrieve}(a_t)$
  else if a_t is a proof candidate π **then**
    $(ok, msg) \leftarrow \mathrm{VerifyLean}(\pi)$
    if ok **then**
      return π
    else
      $C \leftarrow C \cup \{\pi, msg\}$
    end if
  end if
end for
return failure

LeanExplore. Retrieval does not provide LeanCat reference proofs, task-specific hidden lemmas, or future Mathlib content. We evaluate two input settings: NL-only and NL+Statement. The loop runs for at most four refinement iterations and is intended to isolate the effects of iterative refinement, formal grounding, and retrieval within the same dataset.

Verification constraints. All reported proof attempts are checked against Lean 4 and Mathlib v4.19.0. A proof is valid only if the Lean kernel accepts it, the intended theorem statement and local setup are not weakened or changed, and the submitted file contains no `sorry`, `admit`, `unsafe`, or additional `axiom` declarations. All compiling candidates counted as successes additionally undergo expert semantic review against the released task file. The review checks that the target statement is unchanged, assumptions are not weakened, local notions are not shadowed or redefined to trivialize the goal, and the proof does not solve a semantically different task. Candidates that solve a modified, shadowed, or semantically weakened task are rejected. Each attempt has a 50,000-token generation budget and a 300-second verification timeout unless otherwise stated. Appendix D summarizes the experimental configuration.

Exploratory autonomous-agent check. We also report a small Cursor-agent evaluation in Appendix E. This setting uses newer models, a different native-agent harness, a selected subset, and much larger budgets, and is reported separately as an exploratory reference point about test-time interaction.

5. Results

Table 3 summarizes the controlled static and LeanBridge experiments. Static entries report pass@1/pass@4 solve rates, while LeanBridge entries report NL-only/NL+Statement solve rates. Specialized prover pass@32 results are reported in the text because they use the same formal-statement-only, no-retrieval assumption as the static protocol but with a larger independent sampling budget.

Static baselines. The strongest static result is Claude-Opus-4.5, with 8.25% at pass@1 and 12/100 at pass@4. Even this setting solves 0/40 High tasks, and the Easy split is far from saturated: the strongest pass@4 result is 11/20. Across all static generalist experiments, only 14 distinct tasks are solved by at least one model, indicating that independent sampling under this protocol does not exhaust the dataset.

Specialized provers. Specialized neural provers also struggle under no-retrieval sampling, despite the larger pass@32 budget. Table 4 shows that DeepSeek-Prover-V2-671B solves 9/100 tasks, Goedel-Prover-v2-32B and StepFun-Prover-32B solve 5/100 each, and Kimina-72B solves 2/100. No specialized prover solves a High task. This does not imply that specialized provers would fail with retrieval or interactive repair; rather, it shows that independent sampling alone is insufficient for many LeanCat items.

LeanBridge. The controlled retrieve–generate–verify protocol changes measured behavior. For GPT-5.2, the NL-only LeanBridge setting reaches 14/100, compared with 7/100 under static pass@4. With NL+Statement input, GPT-5.2 reaches 24/100 overall, with a split-level breakdown of 13/20 Easy, 10/40 Medium, and 1/40 High. Gemini-3-Pro with NL+Statement also solves 1/40 High tasks. These results indicate that retrieval and iterative verification can reach tasks that static pass@k does not solve, while the absolute High-split counts remain too small for stable rankings.

Retrieval ablation. To separate iterative verification from retrieval, we evaluate a no-search LeanBridge variant using Gemini-3-Pro. This variant removes both the initial retrieval step and later search requests while keeping the same verify–refine loop after compiler feedback. Table 5 shows a monotonic pattern: static pass@4 reaches 8/100, no-search LeanBridge reaches 12/100, and full LeanBridge reaches 20/100. The search gain is concentrated on Medium and High tasks, making retrieval a material evaluation variable for LeanCat.

Table 3. **Main controlled results.** Static entries show pass@1/pass@4 solve rates (in %). LeanBridge entries show NL-only/NL+Statement solve rates (in %) for the retrieve–generate–verify protocol. Raw counts for representative cells are discussed in the text because small split-level percentages are coarse.

Protocol	Model	Easy	Medium	High	Overall
Static pass@k	Claude-Opus-4.5	**40.00/55.00**	0.63/2.50	0.00/0.00	**8.25/12.00**
Static pass@k	GPT-5.2	27.50/35.00	0.00/0.00	0.00/0.00	5.50/7.00
Static pass@k	Gemini-3-Pro	13.75/30.00	**1.25/5.00**	0.00/0.00	3.25/8.00
Static pass@k	DeepSeek-V3.2	20.00/40.00	0.00/0.00	0.00/0.00	4.00/8.00
Static pass@k	Kimi-K2	10.00/20.00	0.00/0.00	0.00/0.00	2.00/4.00
LeanBridge	Claude-Opus-4.5	**70.0/70.0**	5.0/17.5	0.0/0.0	**16.0**/21.0
LeanBridge	GPT-5.2	60.0/65.0	5.0/**25.0**	0.0/**2.5**	14.0/**24.0**
LeanBridge	Gemini-3-Pro	45.0/60.0	**12.5**/17.5	0.0/**2.5**	14.0/20.0
LeanBridge	DeepSeek-V3.2	55.0/40.0	0.0/7.5	0.0/0.0	11.0/11.0
LeanBridge	Kimi-K2	30.0/55.0	2.5/7.5	0.0/0.0	7.0/14.0

Table 4. **Specialized prover pass@32 results.** Entries are solve rates (in %) under the formal-statement-only, no-retrieval protocol. Despite the larger sampling budget, no specialized prover solves a High task.

Model	Easy	Medium	High	Overall
DeepSeek-Prover-V2-671B	**45.0**	0.0	0.0	**9.0**
Goedel-Prover-v2-32B	20.0	**2.5**	0.0	5.0
StepFun-Prover-32B	20.0	**2.5**	0.0	5.0
Kimina-72B	10.0	0.0	0.0	2.0

Table 5. **Retrieval ablation with Gemini-3-Pro.** Search improves the controlled agent baseline beyond compiler-feedback refinement alone.

Split	Pass@1	Pass@4	No Search	Search
Easy	13.75	30.00	45.00	**60.00**
Medium	1.25	5.00	7.50	**17.50**
High	0.00	0.00	0.00	**2.50**
Overall	3.25	8.00	12.00	**20.00**

Raw-count interpretation. LeanCat's 100-task scale makes overall rates useful, but split-level rankings remain coarse. On the High split, 2.5% is exactly 1/40 solved task. High-split improvements therefore indicate that a protocol can reach some tasks unsolved by static sampling, rather than stable split-level estimates of system quality. Table 8 reports representative raw counts and confidence intervals.

Human-only split robustness. Appendix B also reports representative results under a human-only split. The qualitative pattern is unchanged: Claude-Opus-4.5 pass@4 solves 0/38 High tasks, and GPT-5.2 LeanBridge solves 2/38 High tasks. This robustness check separates human ratings from the machine-facing hybrid labels used in the main split.

6. Analysis

Difficulty split and abstraction pressure. The main result tables show that performance drops sharply from Easy to Medium to High under static sampling, while the controlled LeanBridge setting partly smooths this decline without eliminating it. This pattern suggests that the released split tracks a meaningful increase in machine-facing library-grounded formalization difficulty. At the same time, even Easy tasks are far from saturated: the best static pass@4 setting solves 11/20 Easy tasks. LeanCat therefore tests both high-end category-theory formalization and routine-looking abstract interfaces that remain challenging for current systems.

Independent sampling is a weak scaling axis. Moving from pass@1 to pass@4 yields modest absolute improvements for the strongest static models. Many failures involve missing library knowledge, wrong abstraction choices, or absent bridge lemmas, so independent resampling often repeats the same structural error. This indicates that pass@k alone is an incomplete view of progress on LeanCat: increasing independent attempts is not equivalent to improving interaction with the library or maintaining semantic alignment over longer proof trajectories.

Formal grounding and retrieval matter. For LeanBridge, providing both the natural-language statement and the Lean formal statement is most useful beyond Easy tasks. The formal statement anchors universe levels, typeclass assumptions, and the exact Mathlib encoding, reducing drift from the intended theorem. The retrieval ablation shows a complementary effect: the no-search variant reaches 12/100 overall, while full search reaches 20/100, with the gap concentrated on Medium and High tasks. On LeanCat, retrieval access and formal grounding therefore materially change measured behavior.

Failure modes. Manual inspection of failed Medium and High attempts identifies five recurring patterns. **Math failures** use invalid strategies, such as proving an existential statement by choosing an empty category, or replacing a genuine filtered-colimit construction with a constant diagram. **Lean grammar failures** have plausible high-level ideas but fail at syntax, binder order, notation, private constructors, or elaboration before the proof content is reached. **Hallucination failures** invoke nonexistent Mathlib names or projections, for example by guessing API names for full functors, images, or presentability lemmas from nearby mathematical terminology. **Lazy failures** leave required bridge lemmas undeveloped and fall back to automation, comments, `infer_instance`, or placeholders when a local construction is needed. **Hack failures** preserve compilation by changing the formal meaning, for example by shadowing a local notion so the theorem reduces to a tautology.

These failures suggest that the main bottlenecks extend beyond surface-level Lean syntax. They reflect four broader pressures: library awareness, abstraction control, long-horizon coherence across local definitions and bridge lemmas, and semantic preservation of the target statement. The case studies in Appendix C illustrate both sides of this spectrum: Problem 59 succeeds after the construction is aligned with Lean's universe and category instances, while Problem 63 fails because the missing bridge between categorical finite presentability and the concrete group-presentation predicate is not supplied.

Limitations. LeanCat is intentionally compact. Its current 100-task size does not support fine-grained rankings on small cells, especially the High split, and the difficulty labels combine human ratings with model outcomes. The human-only robustness check shows that mathematical difficulty and current-system formalization difficulty are related but not identical. Source statements are adapted from public mathematical references, while the Mathlib overlap audit only addresses direct formal counterparts. LLM-assisted statement drafting can also affect encoding style, although final acceptance used expert review. The main experiments focus on controlled protocols; results may vary with different retrieval tools or Mathlib corpora, and the autonomous-agent check is reported as a separate exploratory setting.

Responsible use and roadmap. LeanBridge is evaluated on LeanCat as a controlled diagnostic baseline. More generally, the reported experiments compare evaluation regimes—static sampling, specialized prover sampling, retrieval-augmented refinement, and autonomous-agent scaffolding—rather than a single fine-grained ranking of systems. Meaningful reuse of LeanCat depends on pinned Lean/Mathlib versions, clear checking instructions, and an update policy that distinguishes dataset evolution from leaderboard continuity. Future dataset extensions can expand beyond 1-categories to monoidal and higher-categorical structures, and can test constructive ability by masking definitions or object constructions in addition to proving fully stated propositions.

7. Conclusion

LeanCat provides a compact Lean 4 dataset for evaluating library-grounded category-theoretic reasoning. The dataset focuses on statement-level proving in a mature abstract library, where progress depends on retrieval, formal grounding, semantic preservation, and the construction of intermediate bridge lemmas. Across static sampling, specialized prover sampling, and retrieval-augmented refinement, the experiments show that evaluation protocol materially affects observed performance. These results support using LeanCat as a focused benchmark for studying how formal-mathematics systems interact with abstraction-heavy Mathlib category-theory interfaces, and for developing more reliable tool-using workflows in Lean.

Acknowledgements

We thank Hongwei Wang and Jiatong Yang for assistance with parts of the Lean statement formalization process.

References

Adámek, J., Herrlich, H., and Strecker, G. *Abstract and concrete categories*. Wiley-Interscience, 1990.

Adámek, J., Chen, L.-T., Milius, S., and Urbat, H. Reiterman's theorem on finite algebras for a monad. *ACM Transactions on Computational Logic (TOCL)*, 22(4):1–48, 2021.

Anthropic. System card: Claude opus 4.5. Anthropic, November 2025. URL https://www.anthropic.com/claude-opus-4-5-system-card. Released November 24, 2025. Accessed on June 22, 2026.

Asher, J. LeanExplore: A search engine for Lean 4 declarations, 2025. URL https://arxiv.org/abs/2506.11085.

Azerbayev, Z., Piotrowski, B., and Avigad, J. ProofNet: A benchmark for autoformalizing and formally proving undergraduate-level mathematics problems. In *Second MATH-AI Workshop*, 2022.

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

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, R. On sifted colimits in the presence of pullbacks. *arXiv preprint arXiv:2109.12708*, 2021.

Gemini Team, Google. Gemini 3 pro model card. Google, November 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf.

Hubert, T., Mehta, R., Sartran, L., Horváth, M. Z., Žužić, G., Wieser, E., Huang, A., Schrittwieser, J., Schroecker, Y., Masoom, H., et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, pp. 1–3, 2025.

Jiang, J., He, W., Wang, Y., Gao, G., Hu, Y., Wang, J., Guan, N., Wu, P., Dai, C., Xiao, L., et al. Fate: A formal benchmark series for frontier algebra of multiple difficulty levels. *arXiv preprint arXiv:2511.02872*, 2025.

Kong, L. Category theory. Unpublished lecture notes, n.d.

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

Liu, A., Mei, A., Lin, B., Xue, B., Wang, B., Xu, B., Wu, B., Zhang, B., Lin, C., Dong, C., et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Liu, C., Shen, J., Xin, H., Liu, Z., Yuan, Y., Wang, H., Ju, W., Zheng, C., Yin, Y., Li, L., et al. FIMO: A challenge formal dataset for automated theorem proving. *arXiv preprint arXiv:2309.04295*, 2023.

Mac Lane, S. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1998.

OpenAI. Update to gpt-5 system card: Gpt-5.2. OpenAI, December 2025. URL https://openai.com/index/gpt-5-system-card-update-gpt-5-2/. Released December 11, 2025. Accessed on June 22, 2026.

Polu, S., Han, J. M., Zheng, K., Baksys, M., Babuschkin, I., and Sutskever, I. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

Ren, Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Riehl, E. *Category theory in context*. Courier Dover Publications, 2017.

Shang, S., Wan, R., Peng, Y., Wu, Y., hui Chen, X., Yan, J., and Zhang, X. Stepfun-prover preview: Let's think and verify step by step, 2025. URL https://arxiv.org/abs/2507.20199.

Team, K., Bai, Y., Bao, Y., Chen, G., Chen, J., Chen, N., Chen, R., Chen, Y., Chen, Y., Chen, Y., et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. PutnamBench: Evaluating neural theorem-provers on the putnam mathematical competition. In Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 11545–11569. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/1582eaf9e0cf349e1e5a6ee453100aa1-Paper-Datasets_and_Benchmarks_Track.pdf.

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

Zheng, H. Category note. Unpublished lecture notes, n.d.

Zheng, K., Han, J. M., and Polu, S. MiniF2F: A cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations*, 2022.

This appendix provides supplementary dataset breakdowns, robustness checks, raw-count interpretations, experimental configuration details, and the exploratory autonomous-agent check referenced in the main text.

A. Detailed Dataset Breakdown

The eight topic clusters cover the following ranges of problem IDs and intended competencies:

- **Basic category properties (1–18):** monomorphisms, epimorphisms, initial and terminal objects, factorization of idempotents, and basic constructions.
- **Adjunctions (19–29):** adjoint functors, comma-category criteria, and concrete examples involving universal properties.
- **Reflective and coreflective subcategories (30–33):** structural properties and concrete classifications, including reflective-subcategory examples that require careful use of subcategory encodings.
- **Concrete categories (34–41):** faithful forgetful functors, concretizability, injectivity, and examples from topology, order theory, and set theory.
- **Limits and colimits (42–73):** cones, cocones, preservation, lifting, and constructions that often require local bridge lemmas.
- **Cocompletions (74–78):** tasks based on recent cocompletion work, including definitions not already available as direct Mathlib facts.
- **Abelian categories (79–90):** kernels, cokernels, exactness, and homological-algebra interfaces.
- **Monads (91–100):** Kleisli and Eilenberg–Moore categories, algebraic laws, and preservation conditions.

B. Additional Tables

Table 6. **Strict Mathlib overlap audit.** Direct counterparts are rare and absent from the High split.

Audit category	Easy	Medium	High	Overall
Direct Mathlib counterpart	8	3	0	11
Requires adaptation, bridge lemmas, or new setup	12	37	40	89

Table 7. **Human-only difficulty robustness check.** The Medium separation weakens under human-only labels, but the High split remains essentially unsolved.

Setting	Split source	Easy	Medium	High
Claude-Opus-4.5 pass@4	Hybrid	11/20 (55.00%)	1/40 (2.50%)	0/40 (0.00%)
Claude-Opus-4.5 pass@4	Human-only	8/24 (33.33%)	4/38 (10.53%)	0/38 (0.00%)
GPT-5.2 LeanBridge	Hybrid	13/20 (65.00%)	10/40 (25.00%)	1/40 (2.50%)
GPT-5.2 LeanBridge	Human-only	10/24 (41.67%)	12/38 (31.58%)	2/38 (5.26%)

C. Representative Case Studies

Figures 2 and 3 show representative LeanBridge trajectories for one High-split success and one High-split failure.

Problem 59: a High-split success. Problem 59 asks for a functor that lifts limits but is not faithful. A successful LeanBridge trajectory constructs a constant functor from $\mathrm{Type}\, u_1$ to a terminal category on `PUnit`. This construction makes limit lifting straightforward in the terminal codomain, while non-faithfulness is witnessed by two distinct endomorphisms in the domain that both map to the unique morphism in the codomain.

Table 8. **Raw-count interpretation of representative results.** Percentages on small splits are paired with solved counts and approximate 95% Wilson confidence intervals.

Setting	Split	Solved / Total	Rate	95% CI	Interpretation
Claude-Opus-4.5 pass@4	Easy	11/20	55.0%	[34.2, 74.2]%	coarse signal
Claude-Opus-4.5 pass@4	Medium	1/40	2.5%	[0.4, 12.9]%	one solved item
Claude-Opus-4.5 pass@4	High	0/40	0.0%	[0.0, 8.8]%	no observed solve
Claude-Opus-4.5 pass@4	Overall	12/100	12.0%	[7.0, 19.8]%	whole-set estimate
GPT-5.2 LeanBridge	Easy	13/20	65.0%	[43.3, 81.9]%	coarse signal
GPT-5.2 LeanBridge	Medium	10/40	25.0%	[14.2, 40.2]%	clear improvement
GPT-5.2 LeanBridge	High	1/40	2.5%	[0.4, 12.9]%	one solved item
GPT-5.2 LeanBridge	Overall	24/100	24.0%	[16.7, 33.2]%	whole-set estimate

The trajectory illustrates why the task is not a routine theorem lookup. Early attempts try vacuous or ill-typed constructions, including an empty codomain, and later attempts encounter universe mismatches when using objects at the wrong universe level. The successful repair uses `ULift Bool : Type u_1`, obtains canonical limit data in $\mathrm{Type}\, u_1$, and separates the two endomorphisms by evaluating at `ULift.up true`. The proof succeeds only after the mathematical example is aligned with Lean's universe and category instances.

Problem 63: a High-split failure. Problem 63 relates compact objects in the category of groups to finitely presented groups and filtered colimits. The target combines categorical finite presentability in `Grp` with a concrete finite-presentation predicate based on `PresentedGroup`, then asks for every group to be represented as a filtered colimit of finitely presented groups.

The attempted strategy is mathematically plausible: reduce categorical finite presentability to a group-theoretic characterization, bridge that characterization to the `PresentedGroup`-based predicate, and invoke a filtered-colimit theorem for groups. The run fails because the required bridge lemmas are not available under the guessed names and are not constructed locally. Subsequent repairs try to replace the missing colimit construction with a constant diagram, which would incorrectly force every group to be finitely presented, or fall back to `simp`, contradiction-style tactics, and placeholders. This case illustrates a different LeanCat difficulty: some tasks require building missing formal infrastructure, not merely locating a named theorem or repairing syntax.

D. Experimental Configuration

Table 9 summarizes the main experimental assumptions used across the reported protocols. The prompt templates below follow the NeurIPS version and specify the inputs used in static generation and LeanBridge refinement. The purpose is to make protocol comparisons reproducible and to separate input assumptions from model family differences.

Table 9. **Experimental configuration.** Shared generation, verification, and LeanBridge settings used in the reported experiments unless otherwise stated.

Component	Setting
Lean environment	Lean 4.19.0 with Mathlib v4.19.0
Static generalist protocol	Formal statement only; pass@1/pass@4
Specialized prover protocol	Formal statement only; pass@32
LeanBridge inputs	NL-only and NL+Statement
LeanBridge retrieval	LeanExplore over Mathlib v4.19.0 declarations and source context
LeanBridge refinement	At most four retrieve–generate–verify iterations
Verification timeout	300 seconds per attempt unless otherwise stated
Generation budget	50,000 tokens per attempt unless otherwise stated
Valid proof criteria	Lean kernel acceptance; no `sorry`, `admit`, `unsafe`, or added `axiom`
Semantic review	Rejects modified, shadowed, weakened, or redefined target statements

```
You are an expert in Lean 4 and Mathematics. Please finish the following proof in Lean4
    code.
Do not change the original statement. Copy the final statement to prove exactly.
Please include the complete header, including imports and namespaces, so that your code
    can pass the Lean4 compiler.
You may introduce auxiliary definitions, instances, and lemmas before the target
    statement if needed. The target statement and all auxiliary code must contain no
    sorry, admit, axiom, or unsafe declarations.

The statement for you to complete is:

```lean4
{formal_statement}
```
```

Listing 1. Prompt template for static pass@k proof generation

```
You are an expert in Lean 4 and Mathematics. Please finish the following proof in Lean4
    code.
If provided a formal statement, do not change it. Include the complete header so that
    your code can pass the Lean4 compiler.
You may introduce auxiliary definitions, instances, and lemmas before the target
    statement if needed. The target statement and all auxiliary code must contain no
    sorry, admit, axiom, or unsafe declarations.

Natural language statement:
{nl_statement}

Lean 4 formal statement:
{formal_statement}

Reference knowledge from Mathlib search:
{search_contents}
```

Listing 2. Prompt template for LeanBridge proof generation

```
The previous Lean 4 proof attempt contains errors. Your goal is to fix them according to
    the compiler's error message.

Input:
Lean 4 code:
{original_proof}

Error messages:
{error_messages}

Instructions:
1. First, analyze the error message carefully in a "## Error Analysis" section.
2. Decide whether you need to search Mathlib for external knowledge or can fix the proof
    immediately.
3. You may preserve or add auxiliary definitions, instances, and lemmas needed by the
    final proof, but keep the target statement unchanged. The target statement and all
    auxiliary code must contain no sorry, admit, axiom, or unsafe declarations.

Condition A: If you need external information
Output the search query wrapped in the tag below:
[SEARCH: your_search_query]

Condition B: If you can fix it immediately
Output the analysis and the fully refined code.
```

Listing 3. Prompt template for LeanBridge refinement

E. Exploratory Autonomous-Agent Check

Modern coding agents provide a different comparison class from both static pass@k and LeanBridge because they run in native orchestration harnesses with richer tool use and much larger test-time budgets. We ran a small exploratory check in Cursor agent mode on 14 harder LeanCat items: Medium problems 9, 22, 51, 73, 87 and High problems 15, 31, 41, 59, 74, 75, 76, 77, 78. We used newer models than the main experiments, Claude Opus-4.6 and GPT-5.4, with approximately 80–100M tokens of exploratory budget per model. Table 10 reports the corresponding solved counts for this separate native-agent setting.

Table 10. **Exploratory native-agent check.** Entries report solved counts on a 14-problem subset consisting of 5 Medium and 9 High tasks. This separate high-budget setting illustrates the effect of richer agentic scaffolding on the harder subset.

Agent setting	Medium solved	High solved	Overall
Cursor agent mode + Claude Opus-4.6	5/5	2/9	7/14
Cursor agent mode + GPT-5.4	4/5	2/9	6/14

These results are reported as an exploratory reference point. The subset is small, the harness and token budgets differ from the main experiments, and the evaluated models are newer than those in the controlled tables. Within the nine High tasks, both models solve Problems 31 and 59 and no others; among these, Problem 59 is the only High task also solved under LeanBridge. The results indicate that Medium tasks become substantially more accessible with richer agentic scaffolding, while High tasks remain selective even with larger budgets and broader tool access.

Problem 59 Proving Trajectory

```
-- Attempt 1: Failed (Universe & Logic Error)
theorem exists_functor_lifts_limit_and_not_faithful : ... := by
  classical
  -- [Error] Trying to use PEmpty causes instance synthesis failures
  -- and logical contradictions when C is inhabited.
  refine PUnit, (by infer_instance : Category.{u} PUnit), PEmpty, (by infer_instance :
   ↪  Category.{u} PEmpty), ?_, ?_, ?_
    exact (Functor.empty _)
    intro J _ K
    refine ?_   -- [Error] Proof stalls here; cannot inhabit the goal.
    intro c hc
    exfalso
    cases c.pt.elim  -- [Error] Ill-typed destruction.
    ...
```

```
-- Attempt 2: Failed (Type Mismatch & Universe Levels)
  ...
  let C' : Type (u + 1) := Type u
  let D' : Type u := OneObjTriv
  -- ... (Functor definition omitted, correctness verified) ...

  -- [Error Section: Non-faithfulness proof]
    intro hfaith
    let X : C' := ULift Bool
    let f : X → X := fun x ⇒ x
    -- [Error] Type mismatch: `!x.down` returns Bool, but expected ULift Bool.
    -- The model forgot the explicit `ULift.up` wrapper.
    let g : X → X := fun x ⇒ !x.down

    have hmap : F'.map f = F'.map g := rfl
    have hinj : f = g := (Functor.map_injective F') hmap
    -- [Error] The simplification proof fails due to the malformed `g`.
    have : (true : ULift Bool) = !true := by ...
```

```
-- Final Version: Success
theorem exists_functor_lifts_limit_and_not_faithful : ... := by
  classical
  -- 1. Setup Canonical Counterexample
  let C' : Type (u₁ + 1) := Type u₁
  let D' : Type u₂ := OneObjTriv
  let F' : C' ⥤ D' :=
    { obj := fun _ ⇒ PUnit.unit
      map := fun {_ _} _ ⇒ PUnit.unit }
  refine ⟨C', (by infer_instance : Category.{u₁} C'),
    D', (by infer_instance : Category.{u₂} D'), F', ?_, ?_⟩

  -- 2. Prove LiftsLimit (Trivial in Terminal Category)
  · intro J _ K
    refine ⟨{ lifts := ?_ }⟩
    intro c hc
    refine
      { liftedCone := limit.cone K
        validLift := by
          -- Key fix: Explicitly constructing the Iso
          refine Cones.ext (Iso.refl _) ?_
          intro j; rfl
        isLimit := limit.isLimit K }

  -- 3. Prove Not Faithful (Witnessed by Bool)
  · intro hfaith
    let X : Type u₁ := ULift Bool
    let f : X → X := fun x ⇒ x
    -- Key fix: Correctly wrapping the boolean negation with ULift.up
    let g : X → X := fun x ⇒ ULift.up (!x.down)

    have hinj : f = g := (Functor.map_injective F') rfl
    -- Explicit contradiction proof
    have hpoint : f (ULift.up true) = g (ULift.up true) := by
      simpa using congrArg (fun h : X → X ⇒ h (ULift.up true)) hinj

    have : (true : Bool) = false := by
      have : (ULift.down (f (ULift.up true))) = ULift.down (g (ULift.up true)) := by
        exact congrArg ULift.down hpoint
      simpa [f, g] using this
    simpa using this
```

Figure 2. **Case study success: iterative refinement on Problem 59.** A representative LeanBridge trajectory in which iterative verification and local repair converge to a valid proof on a High-split task.

Problem 63 Proving Trajectory

```
-- Attempt 1: Trivial Diagram (Failed)
theorem group_realized_as_direct_limit_of_finitely_presented_groups (X : Type u) [Group X] :
  ... := by
  classical
  -- [Strategy Error] Trying to use PUnit (a single point) as the index category.
  -- This implies X itself is finitely presented, which is false.
  refine ⟨PUnit, inferInstance, inferInstance, (Functor.const _ (Grp.of X)), ?_⟩
  intro j
  refine ⟨?_, ?_⟩
    -- [Hopeful Thinking] Hoping `aesop` can prove X is finitely presented.
    simpa using (by classical; aesop)
    -- [Hopeful Thinking] Hoping `aesop` can prove colimit isomorphism trivially.
    simpa using (by classical; aesop)
```

```
-- Attempt 2 & 3: Giving Up (Explicit Acknowledgment)
theorem group_realized_as_direct_limit_of_finitely_presented_groups ... := by
  -- [Agent Comment] "Requires the result that every group is a filtered
  -- colimit of finitely presented groups, and must align with the custom predicate.
  -- Current search results do not provide relevant lemmas."
  -- "Cannot construct existence proof under current constraints."
  fail
```

```
-- Attempt 4: Lazy/Hallucinated Contradiction (Failed)
theorem group_realized_as_direct_limit_of_finitely_presented_groups ... := by
  classical
  -- [Failure] No progress possible without additional lemmas.
  -- Reverts to the invalid PUnit construction.
  refine PUnit, inferInstance, inferInstance, (Functor.const (C := PUnit) (Grp.of X)), ?_
  intro j
  refine ?_, ?_
    -- [Lazy Mode] Tries to claim a contradiction (False.elim)
    -- where none exists. This is a common LLM failure mode when stuck.
    exact False.elim (by trivial)
    exact False.elim (by trivial)
```

Figure 3. **Case study failure: iterative refinement on Problem 63.** A representative failure trajectory in which iterative repair does not overcome missing infrastructure and abstraction-level mistakes on a High-split task.