An Auditable Harness for AI-Assisted Lean Formalization: Pilot Logs from Finite Differential Privacy

Anonymous Authors[1]

Abstract

Workflow claims about AI-assisted formal mathematics are hard to evaluate when prompts, raw outputs, compile traces, and theorem statements are not released together. We report a small reproducibility pilot that pairs a kernel-checked Lean 4 formalization of finite pure differential privacy with a 24-trial audit harness. The harness logs, for each attempt, the prompt SHA-256, the verbatim or status-tagged transcript, the extracted Lean file, the `lake env lean` compile log, an escape-hatch scan, and a statement/context audit that guards against the main compile-only failure mode of proving a weakened theorem under modified definitions. Across six finite-DP lemmas, two `gpt-5`-family models, and two trials per cell, 21 of 24 attempts compiled and 24 of 24 preserved their assigned theorem statement and local Lean context. We classify the three failures as one hallucinated lemma and two tactic-script errors. We frame this as harness design and audit methodology, not a model evaluation: the trial count is small, the tasks are likely contaminated by public Mathlib and DP textbooks, and only one model family is covered. The contribution is the reproducible logging schema and pipeline, and a worked instantiation that other AI-assisted formalization studies can adapt.

1. Introduction

The claim that a language model "can prove" a Lean theorem is tempting to take at face value when the proof compiles, but it is hard to evaluate without seeing the exact prompt, the verbatim model output, the compile log against a pinned toolchain, and a check that the compiled declaration is the one that was asked for. Anything less leaves common failure modes invisible: a model can subtly modify the goal, a tool-using agent can rewrite local context, or a compile success can hide `sorry` or `axiom` escape hatches.

Lean 4 (de Moura & Ullrich, 2021) with Mathlib (The mathlib Community, 2020) provides a strong correctness oracle: each declaration is elaborated by a small kernel admitting a fixed set of rules. Recent AI-for-math work pairs this oracle with language models for tactic suggestion or proof search (Yang et al., 2023; Song et al., 2024; Welleck & Saha, 2023; Xin et al., 2024; Polu & Sutskever, 2020; Jiang et al., 2023; First et al., 2023; Thakur et al., 2024), on benchmarks such as miniF2F (Zheng et al., 2022) and ProofNet (Azerbayev et al., 2023) and through autoformalization (Wu et al., 2022). A recurring methodological concern is reproducibility: prompts, raw transcripts, sampling parameters, and contamination caveats are unevenly reported, making it difficult to compare workflows or to retire spurious capability claims (Golchin & Surdeanu, 2024; Sainz et al., 2023; Zhang et al., 2024).

This paper describes a small audit harness for making a single AI-assisted formalization attempt reproducible end-to-end, and reports a 24-trial pilot on six finite differential-privacy lemmas drawn from a kernel-checked Lean development. The contribution is the harness design and audit methodology, not a benchmark or model evaluation. We do not claim the observed pass rate generalizes: the sample is small, the tasks likely overlap pretraining data, and only one model family is covered. We do claim that the harness produces enough artifacts, per attempt, for an external reviewer to reproduce the compile, inspect the failure mode, and check that the proved declaration matches the intended statement.

2. Harness design

The harness wraps a single AI-generated proof attempt in a fixed pipeline. Each task supplies a Lean file containing the target theorem statement together with the local definitions it depends on; the model is asked to return a complete Lean source file that closes the goal under a pinned Mathlib version. The output is then independently checked by Lean and audited against the assigned statement.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

Table 1. Tasks in the pilot. Reference lines count the human Lean proof.

Task	Diff.	Ref. lines
`eventMass_le_one`	easy	9
`pointwise_to_event`	med.	8
`event_to_pointwise`	med.	4
`postprocess`	m-h.	11
`compose_mass_one`	med.	14
`adaptive_composition`	hard	12

Tasks. Six theorems were extracted from a finite pure DP formalization (Table 1). Each task lists its difficulty band, a reference proof line count, and the Mathlib lemmas its reference proof relies on. Tasks are intentionally short and self-contained: the harness is meant for fine-grained audit, not for measuring long-horizon proof search.

Prompt. A single prompt template is used for all tasks. It fixes the toolchain, forbids escape hatches, and embeds the task body verbatim:

> *You are a Lean 4 expert with deep knowledge of Mathlib4. Prove the theorem stated at the bottom of this prompt. Output ONLY a complete Lean 4 source file (with all imports) that closes the goal. Do NOT use `sorry`, `admit`, `unsafe`, or `axiom`. The file MUST compile against current Mathlib4 (toolchain `leanprover/lean4:v4.27.0`, Mathlib `v4.27.0`). Wrap your entire response in a single fenced code block tagged `lean`. Keep the namespace and definitions exactly as given. ...*

The prompt SHA-256 is logged so a reader can verify which prompt produced a given trial. All six task prompts and their full bodies are included in the supplement.

Models and trials. We invoked two models from the `gpt-5`-family via the Codex CLI (OpenAI, 2024): `gpt-5.4` and `gpt-5.5`. Each (task, model) cell was run for two trials, giving $6 \times 2 \times 2 = 24$ attempts. Sampling parameters beyond the model identifier were left at the CLI defaults; we did not control temperature or seed, so the trials are not bitwise reproducible. The harness records the CLI command line and exit code per attempt.

Compile verification. The model output is parsed for a single fenced `lean` block. The extracted file is written to a fixed path and checked with `lake env lean` against the project's pinned Lean and Mathlib versions. Compile stdout and stderr are saved verbatim. A run is `correct` only if the compile exits with code zero and a separate escape-hatch scan finds no occurrences of `sorry`, `admit`, `axiom`, or `unsafe` in the extracted file.

Statement and context audit. A compile-zero exit code is not sufficient: a model could rename the theorem, weaken its statement, or rewrite local definitions so that the new declaration trivially type-checks. The harness compares the extracted file against the original task file using a syntactic audit (`audit_statement_matches.py`) that checks (i) the theorem name and signature match the assigned task statement and (ii) the local Lean context (definitions of `Kernel`, `PureDP`, `Stochastic`, etc.) is unchanged. In the pilot, this audit reports 24/24 statement matches and 24/24 context matches, with zero issues. We treat this as the principal correctness check beyond compilation: a failed audit would invalidate a passing compile.

Failure-mode taxonomy. Each non-correct attempt receives one or more tags from a fixed taxonomy: `compile-fail-tactic` (a tactic step does not close its goal, e.g. `simp` makes no progress, or a script reaches "no goals to be solved"), `compile-fail-unknown-lemma` (the proof references a Mathlib name not in scope), `compile-fail-typeclass` (missing or ambiguous instance), `compile-fail-syntax` (parse error), `compile-fail-other`, `cheated` (any escape hatch), `extraction-fail`, `cli-error`, and `timeout`.

Logging schema. Each attempt produces one JSON record (`log_schema.json`, supplement). Required fields include `task_id`, `model`, `trial`, `prompt_sha256`, `prompt_chars`, `outcome`, `compile_log_path`, `first_error_line`, `first_error_message`, `proof_lines`, `used_sorry/admit/axiom/unsafe`, `elapsed_seconds`, `cli_command`, `cli_exit_code`, and `timestamp_utc`. A `raw_output_release_status` field distinguishes `available` transcripts from those withheld pending sanitization (`withheld-local-unsanitized`, when transcripts contain local paths or agent tool traces). Of the 24 trials, 16 raw transcripts are released and 8 are withheld; for the withheld ones, the extracted Lean file, compile log, prompt hash, and metadata are still committed and auditable.

Scope. The harness is single-shot: the model receives the task and produces one file. Iterative repair, retrieval, tactic-level interaction, and proof-search with rollback are out of scope; adding them would require additional logging fields beyond the current schema.

3. Pilot results

Aggregate. The 24-trial matrix produced 21 correct compiles, 2 `compile-fail-tactic` runs, and 1

Table 2. Outcome distribution across all 24 trials.

Outcome	Count
`correct`	21
`compile-fail-tactic`	2
`compile-fail-unknown-lemma`	1
`compile-fail-typeclass`	0
`compile-fail-syntax`	0
`cheated`	0
`extraction-fail`	0
`timeout`	0

Table 3. Pass rate by task ($n = 4$ trials per task: 2 models $\times$ 2 trials). Wilson 95% CIs are wide because $n = 4$.

Task	Correct	Pass	Wilson 95% CI
`adaptive_composition`	4/4	100%	51.0%–100%
`compose_mass_one`	3/4	75%	30.1%–95.4%
`eventMass_le_one`	3/4	75%	30.1%–95.4%
`event_to_pointwise`	4/4	100%	51.0%–100%
`pointwise_to_event`	3/4	75%	30.1%–95.4%
`postprocess`	4/4	100%	51.0%–100%

`compile-fail-unknown-lemma` run, for a pass rate of 87.5% (Wilson 95% CI 69.0% to 95.7%). All 24 attempts passed the statement and context audit. None used `sorry`, `admit`, `axiom`, or `unsafe`. Median elapsed wall time was 48 seconds; mean was 75 seconds; the slowest correct trial took 265 seconds (`postprocess`, trial 1) within the 300-second timeout. Exact counts per outcome are in Table 2. We report Wilson 95% confidence intervals throughout to make the small-sample uncertainty explicit; with $N = 24$ the intervals are wide and our claims are intentionally weak.

Per task. Pass rates by task are shown in Table 3. Four of six tasks compiled in 4/4 trials; the remaining two compiled in 3/4 trials. Every task had at least one correct attempt from each model.

Per model. Of 12 `gpt-5.4` trials, 12 compiled; of 12 `gpt-5.5` trials, 9 compiled. We deliberately do not present this as a head-to-head comparison: with only six tasks, two trials per cell, and likely contamination of both models on the same source material, the difference is well within sampling noise and tells us nothing reliable about relative model capability. We report it only because the harness output contains it.

Failure detail. The three failures all came from `gpt-5.5` on its first trial; in each case the second trial succeeded. We summarize them here using only fields recorded by the harness. (i) On `eventMass_le_one`, the proof referenced the constant `Finset.sum_le_univ_sum`, which does not exist in the pinned Mathlib snapshot; the compile log records "Unknown constant". We classify this as a hallucinated lemma. (ii) On `pointwise_to_event`, the proof script ran one tactic past closure, producing "no goals to be solved"; this is a routine over-eager tactic mistake. (iii) On `compose_mass_one`, a `simp` step did not rewrite anything ("`simp` made no progress"); we classify this as a stale or incorrectly fired automation step. None of the failures introduced an escape hatch and none modified the assigned theorem statement; they were attempts at honest proofs that did not type-check.

Table 4. Per-trial ledger (24 attempts). Outcomes are abbreviated: C = correct, T = compile-fail-tactic, U = compile-fail-unknown-lemma.

Task	Model	Trial	Out.	sec.
adaptive_composition	gpt-5.4	1	C	121
adaptive_composition	gpt-5.4	2	C	258
adaptive_composition	gpt-5.5	1	C	41
adaptive_composition	gpt-5.5	2	C	24
compose_mass_one	gpt-5.4	1	C	117
compose_mass_one	gpt-5.4	2	C	143
compose_mass_one	gpt-5.5	1	T	19
compose_mass_one	gpt-5.5	2	C	20
eventMass_le_one	gpt-5.4	1	C	139
eventMass_le_one	gpt-5.4	2	C	73
eventMass_le_one	gpt-5.5	1	U	18
eventMass_le_one	gpt-5.5	2	C	19
event_to_pointwise	gpt-5.4	1	C	38
event_to_pointwise	gpt-5.4	2	C	28
event_to_pointwise	gpt-5.5	1	C	65
event_to_pointwise	gpt-5.5	2	C	19
pointwise_to_event	gpt-5.4	1	C	44
pointwise_to_event	gpt-5.4	2	C	52
pointwise_to_event	gpt-5.5	1	T	22
pointwise_to_event	gpt-5.5	2	C	121
postprocess	gpt-5.4	1	C	265
postprocess	gpt-5.4	2	C	59
postprocess	gpt-5.5	1	C	70
postprocess	gpt-5.5	2	C	22

Audit observations. The compile-only failure mode the audit was designed to catch (proving a renamed or weakened theorem under modified local definitions) did not appear in the pilot. We still recommend running the audit because it can be the difference between a defensible and an indefensible compile-positive result; in our experience it is also the cheapest single check to add to an existing AI-for-Lean pipeline.

4. Caveats

Sample size. $N = 24$ over six tasks is too small for benchmark-style conclusions; per-task Wilson 95% CIs span more than half the $[0, 1]$ interval. We report intervals so that no fragile point estimate is left to anchor a reader.

Training-data contamination. Both Mathlib and the standard finite-DP textbook material (Dwork & Roth, 2014; Warner, 1965) are publicly available and likely represented in pretraining (Golchin & Surdeanu, 2024; Sainz et al., 2023; Zhang et al., 2024). The six tasks are short, named lemmas with idiomatic Mathlib proof shapes, exactly the regime where memorization or near-memorization is most plausible. The pass rate here should be read as an upper bound on AI-assisted formalization performance on novel material, not as evidence about it.

Model and task coverage. Only two models from one provider family (`gpt-5.4`, `gpt-5.5`) were run. We have no coverage of Lean-specialized provers (Xin et al., 2024), retrieval systems (Yang et al., 2023), or copilot-style integrations (Song et al., 2024; Welleck & Saha, 2023); meaningful comparisons would require redesigning the prompt interface, since several of those systems consume tactic states rather than file-level prompts. The six tasks are closely related lemmas from a single Lean file with shared idioms (finite sums, nonnegativity bookkeeping, `Real.exp` algebra). Performance on diverse or novel projects is not implied.

Agent vs. sealed call; stochasticity. The Codex CLI (OpenAI, 2024) is an agent: it can read files, edit, and re-run. Our prompt asks for a single file and the harness records only the final extracted output, but agent tool traces appear in raw transcripts. We did not control sampling temperature or seeds, so re-running would not produce bitwise-identical outputs.

Withheld transcripts; framing. For 8 of 24 attempts, the raw transcript is withheld pending sanitization; the extracted Lean, compile log, prompt SHA-256, and metadata are committed for those, so the pipeline remains end-to-end auditable. The harness is the contribution: reading the pilot as a benchmark or a model evaluation would be a category error.

5. Related work

AI-for-Lean tooling. LeanDojo (Yang et al., 2023) provides a tactic-level Lean interaction environment together with the ReProver retrieval-augmented prover; Lean Copilot (Song et al., 2024) and LLMSTEP (Welleck & Saha, 2023) integrate LLM tactic suggestion into the editor; DeepSeek-Prover (Xin et al., 2024) is a dedicated Lean-tuned model trained on synthetic data; CoPrA (Thakur et al., 2024) runs an in-context proof-search agent. Our harness is not a competing prover. It targets a different quantity: the artifacts and checks needed to reproduce and audit one workflow attempt. It is compatible with any of these tools as the proof-producing component, provided the tool can be wrapped to expose a per-attempt prompt and a single Lean file output.

Benchmarks and autoformalization. miniF2F (Zheng et al., 2022) and ProofNet (Azerbayev et al., 2023) are Lean/Isabelle benchmarks for olympiad-level and undergraduate-level theorem proving. Whole-proof and proof-repair work (First et al., 2023; Polu & Sutskever, 2020; Jiang et al., 2023), together with autoformalization (Wu et al., 2022), increasingly report compile-positive rates on these benchmarks. Our scope is much smaller: we do not introduce a benchmark, and we explicitly argue against benchmark framing for the pilot data.

Formal DP and machine-checked privacy. The Lean development the tasks are drawn from is a finite, real-valued kernel formalization of pure DP, in the same broad area as SampCert (de Medeiros et al., 2025), the Isabelle/HOL formalization of Sato & Minamide (2025), CertiPriv (Barthe et al., 2013), and the finite-computer Coq line (Rindisbacher et al., 2023; Balcer & Vadhan, 2019); relational program logics and sensitivity types such as apRHL (Barthe et al., 2016) and Fuzz (Reed & Pierce, 2010) are complementary. We treat that development as an externally-checked correctness oracle for the harness; the harness itself is agnostic to the mathematical domain.

Reproducibility and contamination. Our caveats follow recent work documenting how easy it is to overstate LLM capability on benchmarks that overlap pretraining data (Golchin & Surdeanu, 2024; Sainz et al., 2023; Zhang et al., 2024). The harness pushes the same discipline to the artifact level: prompts, raw outputs, compile logs, and audit results are all committed alongside the aggregate numbers, so a reader can re-run any specific attempt and form their own judgment.

6. Conclusion

We described a fully-logged audit harness for AI-assisted Lean formalization and a 24-trial pilot on six finite-DP lemmas. The harness logs prompt hashes, raw or status-tagged transcripts, extracted Lean files, compile traces, escape-hatch scans, and a statement/context audit; the pilot yields 21 of 24 compiling proofs with all 24 declarations preserving their assigned statements. We resist any benchmark or capability interpretation. The audit catches the principal compile-only failure mode that pure compile-success reporting misses, and the failure-mode taxonomy makes the three errors specific and inspectable. Useful next steps include sanitizing the withheld transcripts, extending coverage to Lean-specialized provers and to deliberately uncontaminated tasks, and formalizing the contamination-control protocol so that future pilots can report a contamination-adjusted pass rate alongside the raw one.

References

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. In *Workshop on Mathematical Reasoning and AI at NeurIPS 2023*, 2023.

Balcer, V. and Vadhan, S. Differential privacy on finite computers. *Journal of Privacy and Confidentiality*, 9(2), 2019. doi: 10.29012/jpc.679.

Barthe, G., Köpf, B., Olmedo, F., and Zanella-Béguelin, S. Probabilistic relational reasoning for differential privacy. *ACM Transactions on Programming Languages and Systems*, 35(3):9:1–9:49, 2013. doi: 10.1145/2492061.

Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., and Strub, P.-Y. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016)*, pp. 749–758, 2016. doi: 10.1145/2933575.2934554.

de Medeiros, M., Naveed, M., Lepoint, T., Kahsai, T., Ravitch, T., Zetzsche, S., Joshi, A., Tassarotti, J., Albarghouthi, A., and Tristan, J.-B. Verified foundations for differential privacy. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1094–1118, 2025. doi: 10.1145/3729294.

de Moura, L. and Ullrich, S. The Lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G. (eds.), *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Dwork, C. and Roth, A. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–487, 2014. doi: 10.1561/0400000042.

First, E., Rabe, M. N., Ringer, T., and Brun, Y. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*, pp. 1229–1241, 2023. doi: 10.1145/3611643.3616243.

Golchin, S. and Surdeanu, M. Time travel in LLMs: Tracing data contamination in large language models. *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *The Eleventh International Conference on Learning Representations (ICLR 2023)*, 2023.

OpenAI. Codex CLI: A lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2024. Software release.

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

Reed, J. and Pierce, B. C. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pp. 157–168, 2010. doi: 10.1145/1863543.1863568.

Rindisbacher, V., de Amorim, A. A., and Gaboardi, M. Verified differential privacy for finite computers. In *The 9th International Workshop on Coq for Programming Languages (CoqPL 2023)*, 2023. Extended abstract.

Sainz, O., Campos, J. A., García-Ferrero, I., Etxaniz, J., de Lacalle, O. L., and Agirre, E. NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 10776–10787, 2023. doi: 10.18653/v1/2023.findings-emnlp.722.

Sato, T. and Minamide, Y. Formalization of differential privacy in Isabelle/HOL. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2025)*, pp. 67–82, 2025. doi: 10.1145/3703595.3705875.

Song, P., Yang, K., and Anandkumar, A. Towards large language models as copilots for theorem proving in Lean. *arXiv preprint arXiv:2404.12534*, 2024.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving. In *First Conference on Language Modeling (COLM 2024)*, 2024.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pp. 367–381, 2020. doi: 10.1145/3372885.3373824.

Warner, S. L. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965. doi: 10.1080/01621459.1965.10480775.

Welleck, S. and Saha, R. LLMSTEP: LLM proofstep suggestions in Lean. In *Workshop on Mathematical Reasoning and AI at NeurIPS 2023*, 2023.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022.

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

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023.

Zhang, H., Da, J., Lee, D., Robinson, V., Wu, C., Song, W., Zhao, T., Raja, P., Zhuang, C., Slack, D., Lyu, Q., Hendryx, S., Kaplan, R., Lunati, M., and Yue, S. A careful examination of large language model performance on grade school arithmetic. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024.

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 (ICLR 2022)*, 2022.