LeanFlow: A Case Study in Workflow-Driven Lean Autoformalization

Lazar Milikić [1] **Simon Guilloud** [1] **Khanh Nguyen** [1] **Viktor Kunčak** [1]

Abstract

We present and evaluate LeanFlow, an LLM agent system specialized for translating mathematical papers into buildable Lean projects. Recent verifier-in-the-loop systems show that large formal artifacts can be produced, but it remains unclear which runtime mechanisms affect completion, auditability, or efficiency in document-to-project formalization. We study this question through case studies on two previously unformalized mathematical papers in number theory and measure theory, using model, proof-workflow, and toolset ablations with Kimi-K2.6 and GPT-5.5; we report task outcome, API calls, input tokens, and output tokens. With Kimi-K2.6, the full workflow completes both document-level projects within the 2000-call budget, while no-queue variants reach the budget limit; with GPT-5.5, all document-level variants complete, and the full workflow has the lowest or tied-lowest input-token cost on both sources. As complementary calibration, LeanFlow reaches 75.7% BEq+ on the PFR slice of RLM25 and solves all five ICML 2026 AI for Math TCS challenge projects in our GPT-5.5 runs.

1. Introduction

Autoformalization is moving from isolated theorem translation toward turning whole mathematical sources into Lean projects. Early language-model formalization work focused on translating individual natural-language statements into proof-assistant syntax (Wu et al., 2022), and many theorem-proving benchmarks still ask the model to prove one supplied formal statement at a time (Zheng et al., 2022; Azerbayev et al., 2023). A document-level agent faces a different problem: it must convert mathematical prose, TEX structure, references, definitions, and proofs into a buildable Lean project while preserving the intended statements and managing a long sequence of dependent proof obligations. M2F (Wang et al., 2026) and recent textbook- or project-scale systems (Gloeckle et al., 2026; Math, Inc., 2025; Hariharan et al., 2026; Tsoukalas et al., 2026) show that verifier-in-the-loop formalization can produce large Lean artifacts; we examine which parts of the runtime help agents keep the source claims fixed and finish the resulting proof work.

Motivated by this document-scale setting, we introduce LeanFlow as a Lean-specific runtime that separates mathematical editing from workflow control. It first runs a deterministic source preflight: the runtime resolves the input document, extracts or records source blocks, labels, references, PDFs, figures, and support files, and refuses ambiguous TEX roots before the model starts. The formalizer then builds a blueprint, a project-local source map linking source spans to planned Lean declarations, dependencies, proof notes, and recorded scope changes. Before proof search, a statement/source gate checks that the generated declaration types still match the original source claims. During proving, a programmatic workflow manager owns the theorem queue, failed-attempt memory, retry budgets, verification records, logs, and checkpoints. The model proposes edits and helper lemmas, while the manager assigns one target at a time and advances only after external Lean verification accepts the edit.

This separation has different benefits across the two model settings in our experiments. For Kimi-K2.6 (Moonshot AI, 2026), workflow control is decisive in our two document-level runs: the full workflow finishes both sources, while no-queue variants exhaust the call budget. For GPT-5.5, all document-level variants finish, so the observed benefit is not a binary success difference; the same structure instead improves token efficiency and preserves an explicit audit trail for the proof work.

LeanFlow also introduces LeanProbe[1], a cached same-file verifier built on LeanInteract (Poiroux et al., 2025b), for repeated proof-repair checks. LeanProbe gives the proving agent low-latency diagnostics and proof-state feedback while leaving final acceptance to standard Lean/Lake checks. In the sequential same-file benchmark summarized in Ta-

[1]EPFL, Lausanne, Switzerland. Correspondence to: Lazar Milikić <lazar.milikic@epfl.ch>.

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]https://github.com/epfl-lara/LeanProbe

ble 6, cached checking is roughly 9–14× faster than rerunning growing-prefix Lake checks.

Our main evaluation ablates LeanFlow on real document-level formalizations, where the system must recover statements from source documents and then complete the resulting proof work. We compare queue management with free-running proof loops, full Lean tool access with terminal-only interaction, and source-blueprint checkpoints with direct source-to-code generation. Proof-only benchmarks remain useful calibration, but they provide the formal statement in advance and only test whether the system can replace the proof placeholder. We therefore report RLM25-PFR agent runs and the ICML 2026 AI for Math TCS challenge (AI for Math Workshop, 2026) as complementary measurements.

The two document-level case studies are Frisch and Vaserstein's *Parametrization of Pythagorean Triples by a Single Triple of Polynomials* (Frisch & Vaserstein, 2007) and Lyons and Zumbrun's *A Calculus Proof of the Cramer–Wold Theorem* (Lyons & Zumbrun, 2017). The Pythagorean source combines integer-valued polynomial definitions, an obstruction over $\mathbb{Z}[x_1, \ldots, x_n]$, explicit rational-polynomial witnesses, and custom plain-TEX theorem macros. The Cramer–Wold source combines probability measures on Euclidean space, closed half-space values, Crofton reconstruction of average-distance functions, odd-dimensional Laplacian inversion, and an even-to-odd embedding step. These sources are short enough for controlled ablations, but they are still full mathematical papers with nontrivial definitions, representation choices, and proof dependencies. Both require source-aware definitions and reusable Lean infrastructure that the prose leaves implicit.

This paper makes three **main contributions**:

- A document-to-project workflow for Lean formalization. LeanFlow starts from a paper or TEX project, records which source text supports each generated declaration, checks statements before proof search, and moves to the next theorem only after Lean accepts the current proof edit.
- A fast local verifier for proof repair. LeanProbe caches the Lean environment before the current declaration and returns proof states and diagnostics through a CLI and MCP server; final acceptance remains a standard Lean/Lake build.
- An evaluation on full mathematical sources and proof-only benchmarks. We ablate queue control, tool access, and source checkpoints on two paper formalizations with Kimi-K2.6 and GPT-5.5, and report RLM25-PFR and ICML 2026 AI for Math TCS challenge runs as calibration.

We release on GitHub the LeanFlow implementation, LeanProbe verifier, and generated case-study projects.

2. Related Work

Formal theorem proving with language models. Lean 4 (de Moura & Ullrich, 2021) and Mathlib (The mathlib Community, 2020) provide the verification substrate for many recent neural theorem-proving systems. Benchmarks such as miniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), PutnamBench (Tsoukalas et al., 2024), FormalMATH (Yu et al., 2025), SorryDB (Letson et al., 2026), and VeriSoftBench (Xin et al., 2026) measure different proof-only or repository-scale capabilities. They provide useful calibration, but most start from an existing formal statement or repository context. Our work instead studies the runtime needed when the source begins as mathematical prose, and the output must be a buildable project.

Autoformalization across scales. Early LLM autoformalization work studied natural-language-to-formal translation at theorem scale (Wu et al., 2022); later work improved process supervision and evaluation for Lean statements (Lu et al., 2024; Poiroux et al., 2025c;a). M2F (Wang et al., 2026) and Automatic Textbook Formalization (Gloeckle et al., 2026) formalize long mathematical sources into Lean projects, audit generated statements, and repair proofs under a pinned environment. Math, Inc.'s Gauss system is an autoformalization agent for large Lean projects (Math, Inc., 2025). In the sphere-packing project (Hariharan et al., 2026), it helps complete a sorry-free formalization of the dimension-8 result from an existing blueprint and Lean development. AlphaProof Nexus (Tsoukalas et al., 2026) uses an agentic evolutionary framework for formal proof search over a large set of formalized conjectures. Direct comparison with M2F is difficult from the currently available public artifacts: reproducing our document-level runs under M2F would require the same source documents, model, Mathlib version, budget, prompts, workflow logs, tool surface, and final acceptance criteria. Our study is smaller in document volume but different in emphasis: we isolate queue control, tool routing, statement handoff, and model-side budget under the two model settings in our experiments.

Lean tooling for agents. Retrieval and environment access are central to practical Lean proving. LeanDojo (Yang et al., 2023) studies retrieval-augmented theorem proving, and LeanExplore (Asher, 2025) provides search over Lean declarations. LeanInteract (Poiroux et al., 2025b) exposes a Python interface to Lean execution. LeanFlow builds on this tool ecosystem but introduces LeanProbe as its own cached same-file verifier for agent loops; final acceptance still comes from standard Lean/Lake verification.

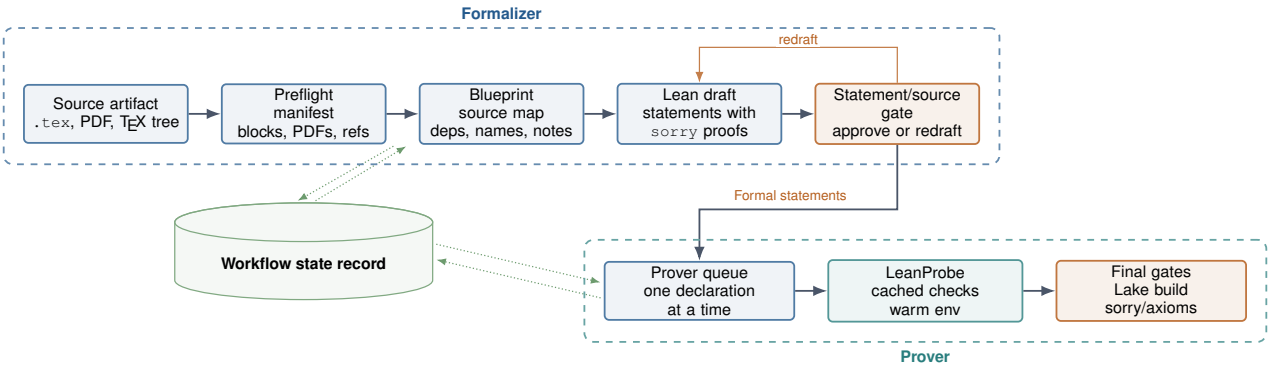

Figure 1. LeanFlow document-to-project pipeline. Dashed regions separate the formalizer and prover phases; the green silo is the project-local workflow state record shared by both. The formalizer emits a well-typed statement skeleton and a durable blueprint, and formal theorem statements pass through a statement/source gate before the prover queue attempts proof closure under verifier and hygiene gates. The workflow state records activity, proof checkpoints, failed attempts, route decisions, prover plans, and outcomes, so redrafts and build failures persist outside the transient conversation state.

3. Problem Setting

Let D be a mathematical source artifact: a LaTeX file, a PDF, or a directory containing a TeX project. Let E be a pinned Lean environment: a Lean toolchain, a fixed mathematical-library revision such as a Mathlib commit, and the Lake project configuration used for checking. The mathematical result should not depend on this particular snapshot, but the evaluation of builds, diagnostics, imports, and available lemmas is always relative to E. The goal is to produce a Lean project P together with declaration-level provenance. For each generated declaration $d \in \mathrm{Decl}(P)$, the workflow should record

$$\pi(d) \subseteq \mathrm{Span}(D),$$

where $\pi(d)$ is the finite set of source spans, labels, sections, pages, equations, or bibliography references justifying d.

We separate three checks. Statement type-checking asks whether the generated Lean declaration is well typed under E when theorem-like proof bodies are replaced by explicit `sorry` placeholders. Statement faithfulness asks whether that well-typed declaration states the source claim, including statement type, quantifiers, domains, hypotheses, and conclusion. Proof completion asks whether a faithful declaration has a complete proof with no unacceptable placeholders or custom axioms. The final project gate is stricter than a theorem-local check: the requested file or full Lake project must build under E, and a hygiene scan must report no unapproved `sorry`, `admit`, `unsafe`, or hidden axiom.

4. Method

4.1. Workflow Overview

Figure 1 and Algorithm 1 summarize the document-to-project contract. LeanFlow uses two primary workflows: `formalize`, which turns a source document into source-backed Lean declarations and proof plans, and `prove`, which repairs or completes Lean proofs until the requested file or project compiles successfully. The workflows are deliberately separated. Formalization may introduce definitions, names, imports, and theorem statements, but it stops at a proof skeleton with source-backed `sorry` bodies. Proof repair then treats those declarations as fixed targets: it may edit proof bodies and add local helper lemmas, but it may not modify theorem statements, add axioms, or leave the assigned proof partially solved.

Algorithm 1 LeanFlow agent workflow contract

Resolve source document D and pinned Lean environment E.
Extract a preflight manifest: source blocks, labels, references, PDFs, figures, and support files.
Create a "blueprint" mapping source spans to planned declarations, dependencies, scope changes, and proof notes.
Draft Lean declarations with placeholders only for theorem-like proofs; verify that the statement layer type-checks.
Run a statement/source gate; if faithfulness fails, redraft before proof search.
while the requested file or project still contains assigned proof obligations **do**
  Assign one theorem-like declaration from the queue.
  Use search, proof context, and LeanProbe-backed checks to repair the assigned proof.
  Persist failures, diagnostics, route decisions, and proof checkpoints after each attempt.
  Accept the assignment only after verification, no relevant `sorry`, and an acceptable axiom profile.
end while

4.2. Source-Backed Skeleton Construction

LeanFlow begins by resolving the source deterministically using document processing tools. For a TEX project directory, deterministic preflight uses regular expression search to identify and select a main entry point, follow local includes, record bibliography files, figures, PDFs, and support files, and fail on ambiguous roots before the agent starts. The same pass extracts standard theorem-like environments, custom theorem declarations, plain-TEX theorem/proof blocks, labels, references, citations, section structure, and nearby proof environments when available. For PDFs, it records extracted text, metadata, and image inventories through local Poppler tools such as `pdftotext`, `pdfinfo`, and `pdfimages`; the model reads this bounded source view through `read_pdf`.

The *formalize* workflow next creates a project-local blueprint before the proof skeleton is considered ready. The blueprint is the durable source map for the run: it lists source statements, planned Lean names, dependencies, source locators, statement-fidelity checks, proof notes, and known scope changes. For each source theorem or lemma, it spells out the parts of the formal type that are most prone to drift: statement type, quantifier order, parameter domains, codomains, hypotheses, side conditions, and follow-on equivalences. The generated Lean module must type-check with theorem bodies left as `by sorry`; proof filling is deferred until the statement/source gate has approved the skeleton.

4.3. Statement/Source Gate

The statement/source gate checks the generated skeleton before proof search begins. Compilation only shows that a declaration is well typed; it does not show that the declaration is the theorem stated in the paper. LeanFlow therefore launches a fresh LLM reviewer context—a separate model context with no human in the loop—over the blueprint, generated Lean declarations, and original source spans. The reviewer is a single invocation of a thinking model; it can approve, reject, or request redrafting, but it does not fill proof bodies. If any check in Table 1 fails, the formalizer returns to the skeleton before the *prove* workflow starts; the prover should not receive a weakened, strengthened, or incomparable target. We call the version that passes this gate the *reviewed skeleton*.

4.4. Project-wide Prover Queue

The project-wide prover queue chooses which file the prover should work on next. When a requested project still contains `sorry`, the manager scans the files, ranks candidate files using declaration dependencies, local hints, examples, and size estimates, records the route decision, and hands one file to the file-scoped declaration queue. This outer queue is needed because document formalizations produce many dependent proof obligations. Without route control, the agent can spend calls on the wrong file, revisit failed paths, or move to later work before earlier dependencies are stable.

Table 1. Statement/source review checks applied before proof repair.

Check	Rejection signal
Statement type	The type of the generated Lean declaration does not fully match the source statement, so a proof may certify a different claim
Quantifiers	Order, dependency, or implicit parameters changed
Variable types	Variables range over the wrong type
Hypotheses	Added assumption, missing side condition, changed nonemptiness
Conclusion	Weakened equality, image statement, equivalence, or existence
Encoding bridge	A local encoding is used without the definitions or equivalence/evaluation lemmas needed to recover the source statement
Proof notes	Source proof idea missing for a nontrivial theorem
Hygiene	Hidden `axiom`, `unsafe`, `admit`, or unreported `sorry`

4.5. File-Scoped Declaration Queue

Figure 2 shows the declaration queue and repair loop inside an assigned file. Once the project queue selects a file, the manager builds a file-scoped declaration queue from pending `sorry` declarations and diagnostics. The front queue item becomes the active declaration; the prover loop repairs only that declaration before the manager advances to the next queue item. For the active queue item, the manager stores the target, theorem-local failed attempts and diagnostics, retry counters, and the latest verification record; it also tracks the remaining declarations in the file and each accepted or failed outcome. It builds the bounded context shown to the model, runs or routes verification, and lets the model advance only after the assigned declaration is clean. The model sees the current target, a small view of the remaining file work, relevant source/proof context, and theorem-local failed attempts.

The file-scoped declaration queue keeps proof search focused by exposing only one active assignment at a time. It prevents common long-run failures: losing the active theorem after compaction, retrying a previously failed proof shape, changing a future statement, or advancing because a text search no longer finds `sorry` while Lean still reports open goals. After each edit, the proving LLM agent refreshes the structured Lean state and asks the manager whether to continue the same declaration, accept the edit, restore a baseline `sorry`, or assign the next declaration.

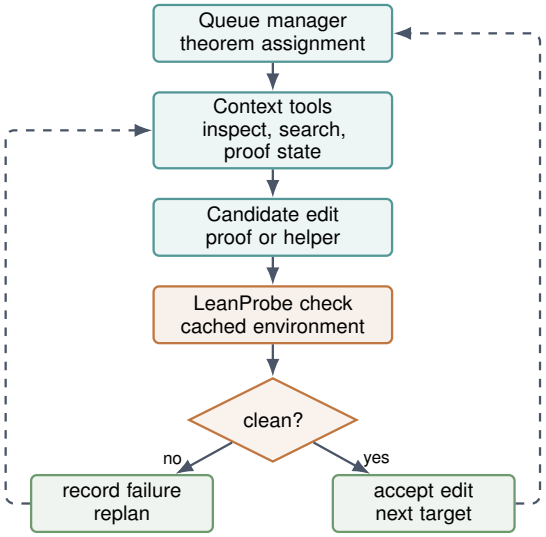

Figure 2. File-scoped proving loop. Failed checks update theorem-local memory; accepted edits advance to file/Lake verification sweeps.

LeanProbe feedback on a failed candidate

```
theorem add_comm_candidate (x y : Nat) :
  x + y = y + x := by
/- <feedback>
-- proof state:
x y : Nat
⊢ x + y = y + x
</feedback> -/
  rfl
/- <feedback>
-- error: tactic 'rfl' failed
</feedback> -/
```

Figure 3. Example LeanProbe feedback embedded into a failed Lean candidate. The proof state is injected before the failing tactic (green marker) and the diagnostic after it (red marker); the failing tactic is underlined in red.

4.6. LeanProbe and Tool Surface

LeanProbe is a declaration-level verifier for repeated proof attempts in one Lean file. It is a standalone CLI, Python library, and MCP server built on LeanInteract (Poiroux et al., 2025b), and is openly available at `epfl-lara/LeanProbe`. At assignment time, LeanProbe runs a *prepare* step for the active declaration: it checks imports, the file header, and all prior declarations, then stores the resulting environment immediately before the target. Each *check* step replaces the target declaration with one candidate proof or helper block and checks it against that cached environment. The cache advances only after a complete declaration replacement is accepted, so failed candidates give diagnostics without changing the environment seen by later obligations.

LeanProbe returns proof-agent feedback: diagnostics, warnings, `sorry` detection, tactic metadata, goal states, and inline Lean comments at the failing location. A failed candidate can be returned as model-readable Lean with compact comments containing the local proof state and the diagnostic beside the failing tactic, as illustrated in Figure 3.

Table 2. Default tool surface exposed to the agent currently comprises 27 tools. The following table groups them by purpose. The manager decides which tool class is relevant for the current proof obligation or failure mode.

Tool class	Agent use
Source readers	Inspect TEX/PDF inputs, extracted text, image inventory, and source spans
File editing	Read, search, patch, and write project files under manager checks
Lean verification	Run canonical Lean/Lake checks and LeanProbe cached target checks
Search/context	Retrieve local/Mathlib facts, nearby declarations, and theorem context
Proof exploration	Try bounded proof candidates, request reasoning help, and decompose helpers
Code hygiene	Scan for remaining `sorry`, custom axioms, unsafe constructs, and axiom profiles
Workflow memory	Reattach skills, blueprints, session notes, and theorem-local failed attempts
Terminal fallback	Run explicit shell commands when the bounded tools are insufficient

This information is used for fast local repair inside the queue loop, not for final acceptance. The final gates remain module or project Lake builds and verification that the project is free from unapproved `axiom` declarations and remaining `sorry` placeholders. Table 2 summarizes the tool classes exposed to the model; ablations in Section 5 remove this surface to isolate its effect.

5. Evaluation

The experiments test whether the LeanFlow workflow components in Figure 1 improve real formalization and proving runs. Each completed run reports task outcome, API calls, input tokens, and output tokens from the workflow logs; the released LeanFlow artifacts include the run configurations, prompts, and logs used for these tables. All model-backed runs use fixed decoding settings across conditions: temperature 0.3, top-$p = 0.95$, and a maximum output budget of 65,536 tokens per model turn. We count only model API calls. For GPT-5.5, one Codex CLI model turn is one call; for Kimi-K2.6, one request to the EPFL-hosted inference endpoint is one call. Terminal commands, file reads and edits, Lean/Lake invocations, LeanProbe checks, and other structured tool invocations are not counted as API calls. For cost accounting, the GPT-5.5 runs used the Codex CLI under a US$200 GPT Pro subscription; one month of this subscription covered all GPT-5.5 experiments reported here. The Kimi-K2.6 runs used a self-hosted EPFL deployment on 8×H200 GPUs, and token-cost estimates use CHF 0.4802 per million input tokens and CHF 1.4406 per million output tokens. Because these two models use different accounting regimes, the tables report calls and tokens rather than

converting all rows to a single monetary scale.

5.1. Document-Level Case Studies

The two document-level case studies are selected for mathematical content and area diversity. Frisch and Vaserstein's Pythagorean-polynomial paper (Frisch & Vaserstein, 2007) requires formalizing the classical $T(a, b, c)$ parametrization, parity and divisibility side conditions, the obstruction to a single integer-coefficient polynomial triple, and the explicit four-variable triple of integer-valued rational polynomials. Lyons and Zumbrun's Cramer–Wold paper (Lyons & Zumbrun, 2017) requires formalizing closed half-spaces as measurable sets, equality of measures from matching half-space values, the Crofton step from half-space values to average-distance functions, the odd-dimensional Laplacian recovery argument, and the even-to-odd embedding $\mathbb{R}^{2m} \hookrightarrow \mathbb{R}^{2m+1}$. Section A gives the detailed source-to-Lean decomposition for both case studies. To the best of our knowledge, neither source theorem had a prior proof-assistant formalization. The completed Pythagorean project comprises 83 Lean declarations (34 of them `theorem`/`lemma` statements) and the Cramer–Wold project 114 declarations (101 `theorem`/`lemma` statements); both build sorry-free and free of unapproved axioms.

For the workflow ablations reported in Table 3, each source first runs through the pre-proof stages in Figure 1 and Algorithm 1: source resolution, blueprint construction, statement generation with proof placeholders, and statement/source review. The result is a reviewed skeleton, i.e., a Lean statement layer that type-checks with theorem bodies left as placeholders and has passed the source-faithfulness checks in Table 1. The Pythagorean reviewed skeleton is produced with Kimi-K2.6; the Cramer–Wold reviewed skeleton is produced with GPT-5.5 because the initial representation setup for probability measures, half-spaces, and the analytic inversion target requires more representation choices than the Pythagorean source. These pre-proof stages are held fixed across the proof-repair ablations, and the 2000-call budget in Table 3 begins after the reviewed skeleton has been accepted. Statement translation, skeleton construction, the statement/source gate, and proof repair all run autonomously, with no human edits to the reviewed skeletons or to the proof runs in Table 3; expert involvement is limited to verifying faithfulness after the fact (Section 6), because no automated faithfulness guarantee is currently available.

5.2. Workflow Ablations on Real Sources

The primary evaluation is an ablation study: it compares the full LeanFlow against variants that remove the queue system and/or the specialized Lean tools. For each model, we run the full 2×2 design: queue versus no queue, and full tool surface versus terminal-only interaction. Table 3 is organized by source so that completion behavior and model-side budget can be compared within the same mathematical project. Here and in later tables, *queue* means the two-layer queue system: the project-wide file queue plus the file-scoped declaration queue. All four conditions for a given source start from the same reviewed skeleton and use the same proof goal, prompt template, model settings, and call budget; only the queue system and tool surface are changed. All document-level proving runs have a maximum budget of 2000 prover-agent API calls, where one call is one model turn in the proof-repair loop. In the terminal-only condition, the agent may still read and edit files, search with shell tools such as `rg`, and run commands such as `lake build`, but it has no access to LeanProbe, MCP-backed Lean tools, structured declaration search, or proof-state APIs. In the no-queue condition, an outer runner still keeps the prover running until the project succeeds or the call budget expires, but both queue layers are disabled: the manager does not route files through a project queue, does not restrict the model to one top-of-queue declaration, and does not use theorem-local failed-attempt history to gate the next target. Source preflight, the statement/source gate, final Lean/Lake verification, and hygiene checks are held fixed across conditions.

Table 3 shows that the combined queue system is decisive for Kimi-K2.6 in these two document-level runs: both no-queue variants exhaust the 2000-call budget on both case studies, while full LeanFlow succeeds in 1043 calls on Pythagorean triples and 1278 calls on Cramer–Wold. The effect of the tool surface is most visible on the analytic case study, where Kimi-K2.6 with a queue but terminal-only interaction still fails on Cramer–Wold. For GPT-5.5, all document-level variants succeed, so these rows do not establish LeanFlow as necessary for completion under this model. Instead, they support an efficiency and auditability claim: full LeanFlow has the lowest or tied-lowest input-token cost on both sources, although the Cramer–Wold no-queue/full-tools run uses slightly fewer calls at a higher input-token cost.

5.3. RLM25-PFR Autoformalization

We evaluate on RLM25-PFR, the PFR slice of RLM25 (Poiroux et al., 2025a;c), rather than on the full RLM25 benchmark. This slice contains 145 theorem-statement examples from the Polynomial Freiman–Ruzsa project. We report it separately from the document-to-project case studies because each example is an independent theorem task rather than a complete source document.

Each RLM25-PFR example is run as a two-stage Lean workflow with one shared 200-call model budget per example. First, the agent receives the informal statement, project con-

Table 3. Long-form workflow ablation on the two document-level case studies. Workflow abbreviations encode queue-system and tool configuration: Full = both queue layers with full tools; NoQ+Tools = no queue system with full tools; Q+CLI = both queue layers with terminal-only interaction; NoQ+CLI = no queue system with terminal-only interaction. Outcome is *success* when the requested project builds under E from the reviewed skeleton with no remaining `sorry` and no unapproved axioms within the 2000-call budget, and *failure* otherwise. Calls are prover-agent API calls with a 2000-call cap.

Source	Model	Workflow	Outcome	Calls	In tok.	Out tok.
Pythagorean	Kimi-K2.6	Full	**success**	1043	46.9M	514.1K
Pythagorean	Kimi-K2.6	NoQ+Tools	failure	2000	160.2M	682.0K
Pythagorean	Kimi-K2.6	Q+CLI	**success**	1561	91.3M	537.8K
Pythagorean	Kimi-K2.6	NoQ+CLI	failure	2000	157.5M	615.5K
Cramer–Wold	Kimi-K2.6	Full	**success**	1278	66.0M	684.1K
Cramer–Wold	Kimi-K2.6	NoQ+Tools	failure	2000	127.8M	583.2K
Cramer–Wold	Kimi-K2.6	Q+CLI	failure	2000	113.1M	579.0K
Cramer–Wold	Kimi-K2.6	NoQ+CLI	failure	2000	167.0M	496.4K
Pythagorean	GPT-5.5	Full	**success**	258	14.3M	190.5K
Pythagorean	GPT-5.5	NoQ+Tools	**success**	568	45.0M	240.3K
Pythagorean	GPT-5.5	Q+CLI	**success**	277	14.3M	177.0K
Pythagorean	GPT-5.5	NoQ+CLI	**success**	984	89.9M	340.7K
Cramer–Wold	GPT-5.5	Full	**success**	495	25.9M	305.5K
Cramer–Wold	GPT-5.5	NoQ+Tools	**success**	448	32.8M	259.7K
Cramer–Wold	GPT-5.5	Q+CLI	**success**	959	59.5M	518.0K
Cramer–Wold	GPT-5.5	NoQ+CLI	**success**	476	49.7M	281.6K

text, and any available informal proof or blueprint material, and synthesizes a Lean declaration that must type-check, typically with a placeholder proof. The generated statement is evaluated against the reference formal statement using the benchmark's BEq and BEq+ statement-equivalence metrics. Second, the generated statement is treated as a fixed proof target: the prover may fill proof bodies and add local helper lemmas, but it may not weaken statements, introduce custom axioms, or leave `sorry`/`admit` placeholders. The call budget is shared by statement synthesis and proof completion, and the call totals in Table 4 include both stages. Runs are autonomous after launch, with no human edits to the generated statements or proofs. If the 200-call budget is exhausted before final verification, the example is counted as a proof failure. Proof success requires final Lean/Lake verification and the same hygiene scan used in our document-level runs. Table 4 summarizes the completed GPT-5.5 runs. In this single GPT-5.5 run, LeanFlow reduces calls (3,541 vs. 4,053) and input tokens (203.2M vs. 236.6M) while increasing BEq+ from 72.9% to 75.7%; because the runs lack variance estimates, we treat the BEq+ change as a small effect rather than a significance claim. The gap between proof success and statement-equivalence metrics reinforces why verified proof completion cannot replace statement/source checks: a proved declaration can still formalize the wrong mathematical claim.

To our knowledge, no published baseline reports provide proof-of-completion results for the RLM25-PFR two-stage setting; the original source (Poiroux et al., 2025a) is not concerned with producing proofs of statements.

Table 4. RLM25-PFR agent-run results on the PFR slice. Proof success is the share of examples whose final Lean code verifies with no remaining placeholders or unapproved axioms under the 200-call per-example budget; BEq and BEq+ are statement-equivalence metrics against reference formal statements. Calls and tokens are totals over the slice and include both statement synthesis and proof completion.

Model	Workflow	Proof succ.	BEq	BEq+	Calls	In tok.	Out tok.
GPT-5.5	LeanFlow	81.2%	70.8%	75.7%	3541	203.2M	2.1M
GPT-5.5	terminal-only agent	80.6%	68.1%	72.9%	4053	236.6M	2.3M

5.4. Challenge Tasks

Project-level proving tasks provide the formal statements and isolate proof construction. The ICML 2026 AI for Math Track 2 TCS proving challenge tests project-level algorithmic proof tasks in graph theory, combinatorics, computational complexity, and algorithm correctness using CSLib and a comparator-backed pass-rate metric (AI for Math Workshop, 2026). The five challenge projects cover binary heap operations, a heap-backed Dijkstra proof, Kruskal minimum spanning trees, segment-tree construction and range queries, and treap invariant/runtime obligations. Each run uses the same 2000-call proof budget as the document-level proof-repair runs. Success means that the final project passes the official SafeVerify/comparator check and also passes our local final Lean/Lake verification and hygiene scan with no remaining `sorry`/`admit` placeholders or unapproved axioms. We run all five projects with GPT-5.5 under three conditions: full LeanFlow, no queue system with LeanProbe still available, and no queue system with only terminal Lean/Lake interaction. For Kimi-K2.6, we

additionally run the Kruskal project with full LeanFlow and without either the queue system or the specialized tool surface. All challenge runs are autonomous after launch, with no human edits to the proof files during the run. The public Track 2 leaderboard lists our competition submission under the `lmilikic` identifier at rank 2 in the combined Phase 1+2 ranking (AI for Math Workshop, 2026).

Table 5 shows that all GPT-5.5 variants solve all five TCS projects, so these rows do not isolate a single decisive workflow component. The results are also consistent with LeanProbe being useful as a feedback surface: workflow logs show that GPT-5.5 used LeanProbe in 37% of recorded agent steps in TCS runs where the tool was available, and NoQ+Probe uses fewer calls and lower input tokens than NoQ+CLI on four of five projects. This aligns with the tool design in Section 4.6, where failed candidates can return local compiler diagnostics and proof-state feedback before the next proof attempt. For Kimi-K2.6, the tested Kruskal run succeeds with the full workflow, while the no-queue terminal-only variant reaches the 2000-call budget.

5.5. LeanProbe Utility

LeanProbe is evaluated separately because it changes verifier latency rather than model capability. We compare cached LeanProbe checks against terminal `lake env lean` checks on two workloads that match the proof loop in Figure 2. The repeated-target workload measures many candidate replacements for one active declaration after a single prepare step. The sequential workload measures a queue-style file run: partial candidates are rejected against the current cached environment, while accepted declarations advance the environment for later targets.

Table 6 supports the design choice of checking candidate edits frequently instead of waiting for large repair batches. The prepare step is paid once before a target, and then each candidate proof replacement is checked in milliseconds to tens of milliseconds. Sequential checks also match the queue design: the environment advances only after a complete declaration succeeds, so failed candidates are diagnosed without changing the context for later obligations. The detailed LeanProbe experimental setup and per-file results are reported in Section B.

5.6. Statement-Skeleton Case Study

We use a qualitative Pythagorean-polynomial comparison to check whether the source-to-statement stages in Algorithm 1 prevent statement drift before proof search. The no-review condition skips the blueprint and statement/source review stages in Steps 3 and 5: after seeing the source, the model is asked to produce Lean translations directly, without a recorded source-to-declaration map or an independent faithfulness check before proving. The no-review draft and the reviewed skeleton both build, but they formalize different mathematical objects. The no-review draft would support a less source-faithful function-level formalization of the result: it defines the main witnesses as rational-valued functions on integer inputs. The reviewed skeleton is the source-faithful version under the checks in Table 1: it defines the witnesses as rational multivariate polynomials and separately proves integer-valuedness, matching the paper's theorem. The de-

Table 5. AI4Math TCS challenge outcomes by project and workflow condition. Workflow abbreviations encode queue-system and tool configuration: Full = both queue layers with full tools; NoQ+Probe = no queue system with LeanProbe available; NoQ+CLI = no queue system with terminal-only Lean/Lake interaction. Calls are prover-agent API calls with a 2000-call per-project cap.

Project	Model	Workflow	Outcome	Calls	In tok.	Out tok.
Binary heap	GPT-5.5	Full	**success**	54	2.86M	49.8K
Binary heap	GPT-5.5	NoQ+Probe	**success**	27	741.3K	7.1K
Binary heap	GPT-5.5	NoQ+CLI	**success**	72	5.03M	71.1K
Dijkstra	GPT-5.5	Full	**success**	61	4.04M	75.1K
Dijkstra	GPT-5.5	NoQ+Probe	**success**	20	770.2K	26.0K
Dijkstra	GPT-5.5	NoQ+CLI	**success**	29	1.15M	23.9K
Segment tree	GPT-5.5	Full	**success**	62	4.01M	35.0K
Segment tree	GPT-5.5	NoQ+Probe	**success**	85	7.01M	42.2K
Segment tree	GPT-5.5	NoQ+CLI	**success**	73	4.67M	45.4K
Treap	GPT-5.5	Full	**success**	142	10.88M	85.1K
Treap	GPT-5.5	NoQ+Probe	**success**	144	10.80M	95.6K
Treap	GPT-5.5	NoQ+CLI	**success**	188	14.10M	116.8K
Kruskal	GPT-5.5	Full	**success**	47	1.75M	16.2K
Kruskal	GPT-5.5	NoQ+Probe	**success**	37	1.56M	17.7K
Kruskal	GPT-5.5	NoQ+CLI	**success**	53	2.94M	26.8K
Kruskal	Kimi-K2.6	Full	**success**	942	107.9M	3.1M
Kruskal	Kimi-K2.6	NoQ+CLI	failure	2000	189.1M	930.4K

Table 6. LeanProbe latency benchmarks. Repeated-target times are average ranges in seconds after one prepare step; sequential rows report total seconds per compact file with five declarations and ten partial/full candidate scenarios.

Workload	OS	Full-file Lake	Cached check	Workload	OS	Growing-prefix Lake	Cached total	Speedup
TCS	macOS	2.082–2.617s	0.031–0.049s	Compact files	macOS	36.474–67.992s	3.789–4.775s	9.63–14.24×
TCS	Linux	1.495–1.886s	0.032–0.054s	Compact files	Linux	22.593–23.186s	2.301–2.547s	9.05–9.82×

tailed declaration-level contrast is reported in Appendix A (Table 7).

6. Threats to Validity

The evaluation has several limitations. Buildability is necessary but not sufficient: a project can build while encoding a different claim. The statement/source gate runs autonomously, but we still confirm faithfulness by expert inspection, because there is currently no automated way to guarantee that a well-typed declaration matches the source; reducing this manual confirmation is important future work. The workflow also assumes that the source definitions and theorem statements are the targets to preserve. After the statement/source gate accepts a skeleton, the prover is not allowed to change definitions or theorem statements; this improves auditability, but it can make a proof obligation impossible if the source contains a typo, an omitted side condition, or an undetected mistranslation. Such errors must be caught during statement/source review or expert audit rather than silently repaired during proof search. The document-level evidence covers only two case studies; the Cramer–Wold skeleton was initialized with GPT-5.5 rather than Kimi-K2.6 because of its representation choices. The ablations are single runs, so they do not estimate variance; repeated document-level runs would be more informative but costly, since each can require hundreds to thousands of calls, tens to hundreds of millions of input tokens, and substantial wall-clock time. Several mechanisms also remain bundled: queue/tool ablations do not isolate blueprinting, statement review, failed-attempt memory, hygiene scanning, and LeanProbe feedback. The complementary benchmarks do not fully exercise document-to-project formalization, the RLM25-PFR rows cover only the PFR slice rather than the full RLM25 benchmark, the GPT-5.5 rows support efficiency claims more than necessity claims, and we do not include a matched document-level M2F comparison. Calls and token counts are useful proxies for model-side effort, but not full measures of wall-clock, monetary, or human engineering cost.

7. Conclusion

LeanFlow treats autoformalization as an evolving Lean project with durable workflow state, not as independent theorem prompts. The central design choice is to keep project control outside the model: the runtime records source-to-Lean decisions, reviews statements before proof search, assigns proofs one at a time, checks candidate edits quickly, and accepts only builds that pass the verifier and contain no leftover placeholders or unapproved axioms. The document-level evidence is model-dependent: in our two case studies, workflow control is decisive for Kimi-K2.6 completion under budget, while for GPT-5.5 it mainly improves token efficiency and audit discipline because every document-level variant completes.

Impact Statement

This work aims to make Lean autoformalization more inspectable, reproducible, and verifier-grounded. Potential positive impacts include reducing the engineering burden of proof repair, improving the auditability of machine-assisted formalization, and making document-level formal verification workflows easier to study. Potential risks include over-reliance on generated formal artifacts, mistaking buildability for source faithfulness, and unequal access to the compute and model resources needed for long formalization runs. LeanFlow is designed to mitigate these risks by recording source-to-Lean decisions, separating statement review from proof search, preserving workflow logs and checkpoints, and accepting final artifacts only after external Lean verification with no leftover placeholders or unapproved axioms. The system is intended to assist expert formalizers rather than replace human mathematical judgment.

Author Contributions

Lazar Milikić led the system design, implementation, experiments, and paper writing. Simon Guilloud helped with benchmarking code, research direction, and paper writing. Khanh Nguyen contributed with research direction and paper writing. Viktor Kunčak supervised the project, contributed to the research direction, and helped revise the manuscript.

Acknowledgments

This research is supported by the AI For Math Fund, administered by Renaissance Philanthropy and funded by XTX Markets. We also gratefully acknowledge the support of the EPFL School of Computer and Communication Sciences.

References

AI for Math Workshop. ICML 2026 AI for math workshop track 2 leaderboard. https://ai4math2026.github.io/track2/leaderboard/, 2026. Track 2 theoretical computer science proving in Lean leaderboard and challenge scoring summary.

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

Azerbayev, Z., Piotrowski, B., Schoelkopf, H. W., Ayers, E. W., Radev, D., and Avigad, J. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics, 2023. URL https://arxiv.org/abs/2302.12433.

de Moura, L. and Ullrich, S. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Frisch, S. and Vaserstein, L. Parametrization of Pythagorean triples by a single triple of polynomials, 2007. URL https://arxiv.org/abs/0706.0290.

Gloeckle, F., Rammal, A., Arnal, C., Munos, R., Cabannes, V., Synnaeve, G., and Hayat, A. Automatic textbook formalization, 2026. URL https://arxiv.org/abs/2604.03071.

Hariharan, S., Birkbeck, C., Lee, S., Ma, H. K. G., Mehta, B., Poiroux, A., and Viazovska, M. A milestone in formalization: The sphere packing problem in dimension 8, 2026. URL https://arxiv.org/abs/2604.23468.

Letson, A., Sarra, L., Poiroux, A., Dressler, O., Lezeau, P., Aranha, D., Pu, F., Hill, A., Hidalgo, M. C., Berman, J., Tsoukalas, G., and Taelman, L. SorryDB: Can AI provers complete real-world Lean theorems?, 2026. URL https://arxiv.org/abs/2603.02668.

Lu, J., Wan, Y., Liu, Z., Huang, Y., Xiong, J., Liu, C., Shen, J., Jin, H., Zhang, J., Wang, H., Yang, Z., Tang, J., and Guo, Z. Process-driven autoformalization in Lean 4, 2024. URL https://arxiv.org/abs/2406.01940.

Lyons, R. and Zumbrun, K. A calculus proof of the Cramér–Wold theorem, 2017. URL https://arxiv.org/abs/1607.03206.

Math, Inc. Introducing Gauss, an agent for autoformalization. https://www.math.inc/gauss, 2025. Company announcement.

Moonshot AI. Kimi-K2.6: Open-weights native multimodal agentic model. https://huggingface.co/moonshotai/Kimi-K2.6, 2026. Hugging Face model card for moonshotai/Kimi-K2.6.

Poiroux, A., Bosselut, A., and Kuncak, V. RLMEval: Evaluating research-level neural theorem proving, 2025a. URL https://arxiv.org/abs/2510.25427.

Poiroux, A., Kuncak, V., and Bosselut, A. LeanInteract: A python interface for Lean 4. https://github.com/augustepoiroux/LeanInteract, 2025b.

Poiroux, A., Weiss, G., Kunčak, V., and Bosselut, A. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17947–17969, Suzhou, China, 2025c. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.907. URL https://aclanthology.org/2025.emnlp-main.907/.

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

The PhysLib community. PhysLib: The Lean physics library. https://github.com/leanprover-community/physlib, 2024. Formerly PhysLean/HepLean.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. PutnamBench: Evaluating neural theorem-provers on the putnam mathematical competition, 2024. URL https://arxiv.org/abs/2407.11214.

Tsoukalas, G., Kovsharov, A., Shirobokov, S., Surina, A., Firsching, M., Bérczi, G., Ruiz, F. J. R., Suggala, A., Wagner, A. Z., Wieser, E., Yu, L., Huang, A., Horváth, M. Z., Ferrauiolo, A., Michalewski, H., Grosu, C., Hubert, T., Balog, M., Kohli, P., and Chaudhuri, S. Advancing mathematics research with ai-driven formal proof search, 2026. URL https://arxiv.org/abs/2605.22763.

Wang, Z., Ma, W., Ming, Z., Zhang, G., Yuan, K., and Wen, Z. M2F: Automated formalization of mathematical literature at scale, 2026. URL https://arxiv.org/abs/2602.17016.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models, 2022. URL https://arxiv.org/abs/2205.12615.

Xin, Y., Chen, Q., Durrett, G., and Dillig, I. VeriSoftBench: Repository-scale formal verification benchmarks for Lean, 2026. URL https://arxiv.org/abs/2602.18307.

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, 2023. URL https://arxiv.org/abs/2306.15626.

Yu, Z., Peng, R., Ding, K., Li, Y., Peng, Z., Liu, M., Zhang, Y., Zheng, Y., Xin, H., Huang, W., Wen, Y., and Liu, W. FormalMATH: Benchmarking formal mathematical reasoning of large language models, 2025. URL https://arxiv.org/abs/2505.02735.

Zheng, K., Han, J. M., and Polu, S. MiniF2F: A cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

A. Document Case Study Details

The LeanFlow implementation is available on GitHub at `epfl-lara/LeanFlow`, the LeanProbe verifier at `epfl-lara/LeanProbe`, and the generated formalization projects at `epfl-lara/AutoformalizedProjects`. Each released project is self-contained and pins its own Lean toolchain and Mathlib revision (for example, Lean `v4.30.0-rc2` for the Pythagorean project and `v4.29.1` for Cramer–Wold), so builds are reproducible without a single global environment. The two case studies have also been contributed to the community `lean-pool` repository of AI-produced Lean developments (pull requests #185 and #186).

Pythagorean-polynomial source. The Frisch–Vaserstein source first defines Pythagorean triples, polynomial parametrization over $\mathbb{Z}[x_1, \ldots, x_n]$, and integer-valued parametrization over $\mathrm{Int}(\mathbb{Z}^n) \subseteq \mathbb{Q}[x_1, \ldots, x_n]$. The formalization therefore introduces both integer-coefficient and rational-polynomial representations before stating the main claims. The source-level proof obligations are: the standard two-family parametrization of triples, the obstruction to a single integer-coefficient triple, the explicit four-variable integer-valued witness, the positive-triple variant, and the four-square unrestricted-parameter variant. The obstruction proof is algebraic: it uses the UFD structure of $\mathbb{Z}[x_1, \ldots, x_n]$, a gcd decomposition, parity facts, and the examples $(3, 4, 5)$ and $(4, 3, 5)$ to force a contradiction. The construction proof uses the source map

$$T(a, b, c) = \big(c(a^2 - b^2)/2,\ cab,\ c(a^2 + b^2)/2\big)$$

and the parity condition that makes all three coordinates integral. The reviewed Lean layout separates these concerns into basic definitions, source handoff lemmas, obstruction lemmas, integer-valued construction lemmas, positive-triple lemmas, and explanatory material for cited background. This layout also records source claims that are likely to be omitted in a proof-only draft: the finite-cover theorem for integer-valued parametrizations, the displayed binomial-polynomial identity, and the non-UFD motivation for $\mathrm{Int}(\mathbb{Z})$.

Table 7. Declaration-level contrast in the Pythagorean statement skeleton. The no-review draft is generated directly from the source; the reviewed skeleton is the version accepted by the statement/source review conditions described in Table 1.

Source requirement	No-review draft	Reviewed skeleton
Integer-valued polynomial objects	`IntegerValued4` is a predicate definition on functions $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Q}$; integer-valuedness is checked only after evaluation at integer inputs.	`RatPoly n` is a type abbreviation for rational multivariate polynomials; `IsIntValued` is a predicate definition, and `IntValuedSubring n` is a subring definition.
Displayed witnesses are polynomials	`f_param`, `g_param`, and `h_param` are function definitions of four integer arguments, so the displayed source witnesses are encoded only by their evaluations.	`f_param`, `g_param`, and `h_param` are polynomial definitions of type `RatPoly 4`; evaluation at integer tuples is handled separately by `ratPolyEval`.
Main theorem type	`integer_valued_parametrization` is a theorem proving integer-valuedness and image equality for the function model.	`exists_int_valued_parametrization` is a theorem with conclusion `IntValuedParametrizes ...`, so it states existence of integer-valued rational polynomials that parametrize the triples.
Background claims retained	The direct translation omits theorem or definition declarations for the finite-cover result and the integer-valued-polynomial ring background.	The reviewed skeleton keeps theorem declarations for the finite-cover result and non-UFD claim, plus polynomial definitions and theorems for the binomial-polynomial identity.

Cramer–Wold source. The Lyons–Zumbrun source states that Borel probability measures on $\mathbb{R}^n$ are determined by their values on closed half-spaces. The formalization represents $\mathbb{R}^n$ as Lean Euclidean space, defines closed half-spaces by a normal vector and threshold, and records half-space value functions and the average-distance function

$$f_\mu(y) = \int (\|y - x\| - \|x\|)\, d\mu(x).$$

The first analytic proof block packages the Crofton/Fubini argument showing that half-space values determine f_μ. The second proof block packages the odd-dimensional inversion step: after reindexing dimensions as $2m + 1$, the needed kernel

identity is that an $(m+1)$-fold Laplacian of the norm kernel recovers a nonzero multiple of the point mass at the origin. In Lean-facing terms, this requires a tempered-distribution norm kernel, Fourier/Laplacian algebra for iterated Laplacians, transport between the Euclidean model used by PhysLib (The PhysLib community, 2024) and the project's odd-dimensional space, and a bridge from PhysLib's real distribution API to Mathlib's complex tempered distributions. The analytic formalization also produced reusable PhysLib infrastructure for distributional Laplacians of norm-kernel fundamental solutions, contributed upstream in `leanprover-community/physlib#1175`. The even-dimensional Cramer–Wold theorem is then reduced to the odd-dimensional one by embedding $\mathbb{R}^{2m}$ into $\mathbb{R}^{2m+1}$ and transporting half-space values along that embedding. The reviewed skeleton keeps the Crofton and inversion blocks as named proof targets rather than folding them into the final theorem; this makes clear which missing analytic facts are mathematical infrastructure, not model-generated assumptions.

B. LeanProbe Benchmark Results

This appendix records the detailed LeanProbe benchmark rows behind Table 6. The snapshot was refreshed on May 13, 2026, with Lean 4.30.0-rc2. These rows are verifier-latency measurements, not model-performance measurements: they isolate the cost of checking candidate Lean edits under different reuse policies. The benchmark suite has two parts: grouped repeated-target checks, which stress many alternatives for one active declaration, and sequential same-file checks, which stress queue-style advancement through a file. All runs use one measured run per target/file, no benchmark warmups, warm Lake caches from prior validation, and a fresh-server baseline. LeanProbe remains a feedback surface; final file or project acceptance still uses standard Lean/Lake verification.

Grouped repeated-target benchmark. The repeated-target workload measures the agent pattern of trying several complete replacements for one active declaration. For each target, the benchmark writes a temporary full file and times terminal `lake env lean`; then it starts LeanProbe, prepares the environment immediately before the target, checks the replacement against that cached environment, optionally requests proof-state feedback, and finally repeats the check with a fresh LeanProbe/LeanInteract server to measure the no-cache baseline. The compact groups are Mathlib-facing examples: `analysis_real` covers elementary real-analysis facts, `algebra_order` covers ordered algebra and inequalities, `sets_functions` covers set/function image and preimage reasoning, and `number_theory_nat` covers elementary natural-number arithmetic. The TCS groups are longer challenge-style extracts: `tcs_binary_heap` contains heap operations such as `heapify`, `extract_min`, `insert`, `merge`, and `remove`; `tcs_treap_analysis` contains the two probability-sum obligations used in the treap analysis; and `tcs_weighted_graph` contains weighted-graph helper declarations through the `Sym2order` prefix. In Table 8, *Targets* is the number of active declarations in the group, *Lake full* is full-file terminal verification, *Prepare* is the one-time cached-environment setup, *Cached* is a target replacement check after prepare, *Feedback* includes diagnostic/proof-state metadata, *Fresh* is the same check with a fresh server, and *Fresh/cached* is the fresh-check time divided by the cached-check time.

Sequential same-file benchmark. The sequential workload models a queue-managed file run rather than repeated attempts on one target. Each compact file has five declarations and ten scenarios: for every targetable declaration, the benchmark checks a partial declaration containing `sorry` and then the complete declaration. LeanProbe reports the partial scenario and detects the `sorry`, but it advances the cached environment only after the complete declaration succeeds. The Lake baselines rerun terminal checks for every scenario, either on a growing prefix that includes accepted prior declarations or on a full temporary file in which only the current declaration is replaced. In Table 9, *Lake prefix* and *Lake full* are these two terminal baselines, *Probe cached* is one warm LeanProbe/LeanInteract server walking the whole file, *Probe fresh* restarts LeanProbe for every scenario, and the three speedup columns divide the corresponding baseline total by the cached LeanProbe total.

Table 8. LeanProbe repeated-target grouped results. Times are averages in seconds; fresh/cached is the ratio between a fresh LeanProbe server and a cached check.

Platform	Group	Targets	Lake full	Prepare	Cached	Feedback	Fresh	Fresh/cached
macOS	analysis_real	5	3.893	6.024	0.039	0.022	4.014	139.7×
macOS	algebra_order	5	3.900	3.683	0.048	0.039	3.987	106.7×
macOS	sets_functions	5	3.708	3.502	0.008	0.007	3.766	454.7×
macOS	number_theory_nat	5	3.731	3.478	0.011	0.006	3.776	420.0×
Linux	analysis_real	5	2.276	2.315	0.025	0.024	2.412	103.2×
Linux	algebra_order	5	2.301	2.317	0.046	0.043	2.516	78.2×
Linux	sets_functions	5	2.233	2.257	0.011	0.009	2.383	245.8×
Linux	number_theory_nat	5	2.199	2.217	0.009	0.008	2.390	322.0×
macOS	tcs_binary_heap	9	2.576	2.931	0.049	0.042	2.589	155.9×
macOS	tcs_treap_analysis	2	2.082	2.219	0.034	0.034	2.181	77.8×
macOS	tcs_weighted_graph	9	2.617	2.461	0.031	0.028	2.603	194.9×
Linux	tcs_binary_heap	9	1.886	1.807	0.054	0.051	1.877	103.2×
Linux	tcs_treap_analysis	2	1.495	1.441	0.036	0.040	1.560	53.1×
Linux	tcs_weighted_graph	9	1.771	1.683	0.032	0.034	1.761	127.5×

Table 9. LeanProbe sequential same-file results. Each file has five declarations and ten partial/full scenarios. Times are total seconds for the file.

Platform	File	Decls.	Scen.	Lake prefix	Lake full	Probe cached	Probe fresh	vs prefix	vs full	vs fresh
macOS	analysis_real	5	10	67.992	40.216	4.775	44.699	14.24×	8.42×	9.36×
macOS	algebra_order	5	10	46.870	48.163	4.339	40.953	10.80×	11.10×	9.44×
macOS	sets_functions	5	10	45.421	42.237	3.916	37.797	11.60×	10.79×	9.65×
macOS	number_theory_nat	5	10	36.474	36.648	3.789	36.736	9.63×	9.67×	9.70×
Linux	analysis_real	5	10	22.765	22.958	2.515	24.890	9.05×	9.13×	9.90×
Linux	algebra_order	5	10	23.186	23.489	2.547	25.147	9.10×	9.22×	9.87×
Linux	sets_functions	5	10	22.679	22.567	2.384	24.195	9.51×	9.47×	10.15×
Linux	number_theory_nat	5	10	22.593	22.829	2.301	24.086	9.82×	9.92×	10.47×

Repeated-target rows isolate the cost of checking many replacements for one active declaration after a single prepare step. For n complete replacement attempts on the same target, the comparison is

$$\text{Probe total}(n) = \text{Prepare} + n \cdot \text{Cached}, \qquad \text{Lake total}(n) = n \cdot \text{Lake full}.$$

The preparation cost is therefore amortized as soon as a target receives several candidate repairs: after setup, each additional candidate costs the cached-check time instead of another full-file `lake env lean` run. The grouped rows report this steady-state behavior directly through *Cached*, *Fresh*, and *Fresh/cached*; the sequential rows report the end-to-end cost when successful declarations advance the environment. Sequential rows model queue-style proving in one file: partial `sorry` scenarios are diagnosed but not cached, while complete accepted declarations advance the environment for later targets. This matches the proof-loop policy used in our runs: keep the LeanProbe server warm, prepare once before a target when several repairs are likely, and advance the cached environment only after a complete declaration has been accepted.