BEYOND THE LIBRARY: AN AGENTIC FRAMEWORK FOR AUTOFORMALIZING RESEARCH MATHEMATICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Research mathematics often introduces concepts and dependencies beyond the coverage of existing formal libraries. Autoformalizing such work requires constructing suitable definitions, checking their correspondence with the source, and developing proofs over them. We introduce *Theo*, an agentic framework powered by general coding LLMs that integrates these tasks in a definition-first workflow. To select among candidate formalizations of new definitions, Theo generates auxiliary lemmas and attempts to prove them under each candidate. It favors candidates that make these proofs easier, using proof difficulty as a proxy for formalization quality. An orchestrator then builds on the selected definitions through recursive proof development and semantic review, revising them when downstream issues arise. We evaluate proof generation on 32 randomly sampled PutnamBench problems and obtain checked proofs for all 32. We also formalize selected main results from nine research papers, including five STOC papers, Babai's graph isomorphism result and the Strong Perfect Graph Theorem of Chudnovsky et al. The case studies expose a gap in a STOC proof, three incorrect intermediate statements in the Strong Perfect Graph Theorem proof, and three gaps in Babai's graph isomorphism manuscript. For the latter two developments, we prove repaired statements and steps that preserve the main results. These findings demonstrate how formalization can complement mathematical proof review.

1 INTRODUCTION

As AI systems become increasingly capable of generating proofs for real-world mathematical theorems (Woodruff et al., 2026), the burden of proof verification has emerged as a primary bottleneck. Large Language Models (LLMs) tend to make subtle logical errors that differ substantially from typical human mistakes (Li et al., 2024), making manual verification by mathematicians both time-consuming and exceptionally difficult. This challenge underscores a critical need for automated verification, which would allow researchers to rigorously test and validate generated proofs at scale. Formal mathematical languages, such as Lean 4 (de Moura & Ullrich, 2021), currently offer the most robust environment for this type of mechanical checking. Consequently, automating the translation of informal reasoning into these strict, verifiable languages serves as a critical stepping stone for the future of AI-assisted mathematics.

Autoformalization entails translating informal mathematical concepts into machine-verifiable code. This pipeline naturally divides into two distinct tasks: formalizing the theorem's statement and formalizing its proof. Once a statement is established in a formal language, the correctness of any accompanying proof can be mechanically verified. However, unlike proof formalization, which benefits from direct compiler feedback, statement formalization lacks a mechanical ground truth. Because it is impossible to automatically verify that a translated statement perfectly aligns with its natural language source, statement autoformalization poses unique challenges. [1]

We propose *Theo*, an autoformalization setup where an orchestrator manages execution of two pipelines: one that formalizes a paper's statements and one that formalizes their proofs (Figure 1). Our setup aims to address the limitations of both single agents (Pozo et al., 2026; Xu & Odersky,

[1] This is because the theorem statements in their initial form are in natural language and hence they cannot be mechanically processed.

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

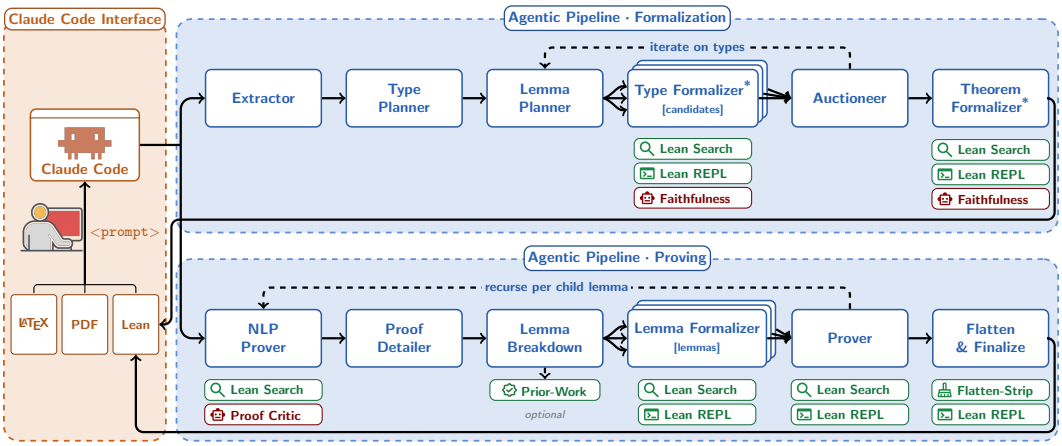


Figure 1: Overview of Theo. The user interacts through the Claude Code interface (left), providing the paper in LaTeX and PDF form together with a prompt, and the system returns Lean code. The orchestrator drives two pipelines. The *Formalization* pipeline (top, Section 2.1) extracts the main theorem, then iterates over the required types, planning and formalizing each type together with its auxiliary lemmas, and finally formalizes the theorem statement. The *Proving* pipeline (bottom, Section 2.2) drafts a natural-language proof, details and decomposes it into lemmas, formalizes and proves them, and finalizes a self-contained Lean proof. Stages marked by "*" expand into sub-pipelines of their own. Green badges denote MCP tools available to a stage. The dashed arrows show the two control loops: the formalizer iterates over types, and the prover recurses on each child lemma. The pipeline is not static: the orchestrator can reorder stages and open feedback bridges between them.

2026) and existing pipelines (Ren et al., 2026; Wang et al., 2026d; Chen et al., 2025a; Varambally et al., 2025; Achim et al., 2025). In single-agent setups, failed proof attempts and compilation errors rapidly saturate the context window, degrading long-horizon reasoning; our system circumvents this by delegating sub-tasks to subagents, preserving the orchestrator's context for high-level strategy. Our system also aims to improve on pipeline models (Ren et al., 2026; Wang et al., 2026d; Chen et al., 2025a; Varambally et al., 2025; Achim et al., 2025) through flexible stage ordering and pipeline-level backtracking. Specifically, our orchestrator can dynamically backtrack to a previous pipeline state on the fly, potentially avoiding the hard failures common to rigid pipelines. Finally, this flexible design seamlessly supports human-in-the-loop interaction, allowing researchers to inject domain expertise or correct logic mid-process without forcing a complete pipeline restart.

In research-level mathematics autoformalization, one major bottleneck is the limited scope of formal libraries. Because repositories like Mathlib (The mathlib Community, 2020) lack coverage across many established fields, attempting to directly translate complex theorems in unsupported domains routinely fails. To bridge this gap, our architecture introduces a "type-first" formalization paradigm. Before addressing the primary theorem, the orchestrator explicitly defines any missing domain-specific concepts to build the necessary mathematical vocabulary. By systematically overcoming the constraints of existing libraries, this approach makes it possible to successfully formalize a significantly broader range of research papers.

Simply defining new mathematical vocabulary is insufficient if the formalized definitions are wrong or difficult to use. To assess the faithfulness and usability of these newly formalized types, we introduce an auxiliary lemma technique that first generates several theorems related to a new type and then attempts to prove them. If the prover fails, it may indicate that the formalized type is either incorrect or poorly formalized, making proofs unnecessarily difficult in future, or simply reflect prover limitations. Conceptually, this approach functions much like unit testing in software engineering (Sommerville, 2011; Yang et al., 2024). Just as unit tests check the fundamental components of a program, auxiliary lemmas provide evidence for the correctness of formalized types, which serve as the building blocks of the formalization. Ultimately, missing assumptions or poor formalizations can make the auxiliary lemmas harder to prove, which allows the orchestrator to retry formalization before going forward.

We integrate three established techniques to further enhance pipeline robustness. First, to verify the main theorem's statement, we employ back-translation (Guo et al., 2025): agents translate the generated Lean code into natural language to detect structural discrepancies against the original text. Second, to streamline the proof generation process, a specialized agent decomposes complex proofs into smaller, manageable lemmas (Zhou et al., 2025). Finally, inspired by MerLean (Ren et al., 2026), we do not necessarily formalize the proofs of prior work. Instead, we may declare these prior results as axioms, isolating the formalization effort strictly to the paper's core contributions.

Beyond formalizing statements, our system includes a full proof-formalization pipeline that produces machine-checked proofs. It recursively decomposes a theorem into lemmas and, at each node, proves the node from its sub-lemma statements before recursing into them, admitting only clearly-labeled results from prior work as axioms. Run end-to-end on research papers, this pipeline produced machine-checked proofs across nine papers, including five from the ACM Symposium on Theory of Computing (STOC) (Pham, 2025; Mackenzie & Saffidine, 2025; Gravin & Jia, 2025; Rivkin et al., 2025; Kalai et al., 2010); two recent OpenAI manuscripts, OpenAI (2026b), and OpenAI (2026a), and the Strong Perfect Graph Theorem (Chudnovsky et al., 2006) in Annals of Mathematics; and Babai's graph isomorphism result (Babai, 2018). In addition, Theo uncovered a gap in a published STOC proof (Pham, 2025) and three gaps in the Strong Perfect Graph Theorem paper (Chudnovsky et al., 2006); see Appendix C.9.1. We also found three main gaps in Babai's graph isomorphism manuscript; see Appendix C.10 for the counterexamples and repairs.

To evaluate our proof-formalization pipeline, we also ran it end-to-end on PutnamBench (Tsoukalas et al., 2024), whose formal Lean statements are given. For each problem the pipeline's own Natural Language Prover generates an informal proof from the statement alone, which the pipeline then details, decomposes, and formalizes into a machine-checked Lean proof. We tested this on 32 randomly sampled problems, and our pipeline proved all of them. This resulted in a lower-bound accuracy of 91.3% across the entire dataset, with a confidence level greater than 95%. Notably, this approach outperforms state-of-the-art specialized models using only a standard $200 software subscription, averaging an operational cost of approximately $5 per problem.

In summary, this work makes the following key contributions:

- A definition-first workflow that generates auxiliary properties to diagnose newly introduced mathematical vocabulary and combines them with semantic review.
- An agentic implementation that coordinates statement construction and recursive proof development, including returns to definitions and dependency planning when difficulties arise.
- Research-level formalizations of main results from nine papers, uncovering proof gaps in three of these works.

2 Autoformalization Design

The *Orchestrator*, implemented as a Claude Code session, coordinates agents that extract the target theorem, construct its definitions and auxiliary lemmas, and develop a Lean proof. Figure 1 shows the two pipelines, whose components are described below.

2.1 Statement Formalization

The statement formalization pipeline is shown in the top half of Figure 1. The system is invoked by the user with a natural-language prompt asking the orchestrator to formalize the provided paper or problem; the exact prompts we used are given in Appendix E. First, the *Extractor* finds the main theorem from the paper's PDF and LaTeX source. It captures all definitions and assumptions required to understand the theorem, not just the theorem statement itself. Then, our *Type Planner* determines what types beyond Lean's standard libraries are needed for the formalization, generating a *type plan* representing the essential concepts absent from Mathlib.

The orchestrator then iterates over the types in the plan, following their dependency order in the graph. In each iteration, a *Lemma Planner* agent first identifies well-known properties and lemmas associated with the current type, which test intended properties and generality of its formalization. These lemma

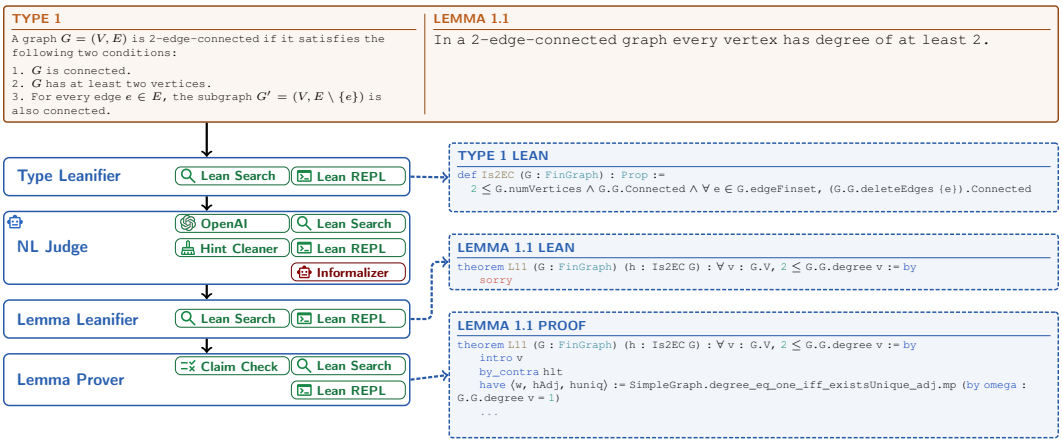


Figure 2: Type Formalizer Expanded; This is part of the main pipeline where we formalize and investigate each type candidate.

statements are kept as general as possible and are not tailored to the current problem. The orchestrator then spawns k *Type Formalizer* agents for the current type, each tasked with translating the type and its lemmas from natural language to Lean, attempting to prove the associated auxiliary lemmas, and verifying them with a *Faithfulness Judge* agent (Section 2.1.3).

Once these parallel attempts conclude for the type, the *Auctioneer* evaluates the candidates using a *best-of-k* selection strategy. The auctioneer ranks the generated formalizations based on three criteria: the proportion of successfully proved lemmas, semantic alignment verified by a Faithfulness Judge agent, and the length of the resulting Lean code. Because a flawed foundational type would fatally propagate errors through the rest of the pipeline, this selection process is strictly gated; if no candidate meets the required quality, the orchestrator dynamically spawns additional Type Formalizers until a candidate satisfying the acceptance checks is generated. Once a type's definition is accepted, the orchestrator advances to the next type in the graph, building on the definitions established so far. With the necessary mathematical vocabulary established, the orchestrator finally spawns a *Theorem Formalizer* agent to translate the primary theorem statement. The agent operates in a back-and- forth collaboration with the Faithfulness Judge agent. Having outlined the stages of the pipeline, we next describe the key building blocks in more detail.

2.1.1 Type Formalizer

The *Type Formalizer* is responsible for translating given types and their associated auxiliary lemmas into Lean and for proving those lemmas. There are four agents performing these tasks (Figure 2). Initially, the *Type Leanifier* translates the given informal type candidate, which includes definitions, functions, and properties, into Lean. Through iterative back-and-forth with the *Faithfulness Judge* (Section 2.1.3), we achieve a likely semantically correct formalization of the types.

Once the type formalization is semantically verified, the *Lemma Formalizer* translates the associated lemmas into Lean. Next, the *Lemma Prover*, which can be any off-the-shelf prover or the model itself, attempts to prove the lemmas. Since the model might alter the statement formalization during proving rather than just providing a proof, the *Claim Check* tool replaces the original statement formalization with the original statement and then compiles it to ensure the prover cannot cheat.

These checks supply the semantic assessments, proof success rates, and code lengths used by the Auctioneer.

2.1.2 Theorem Formalizer

The *Theorem Formalizer* builds the main statement from the accepted definitions and submits it to the Faithfulness Judge. A failed review can trigger new definitions or revisions to existing ones. The resulting statement is then shown to the user for review.

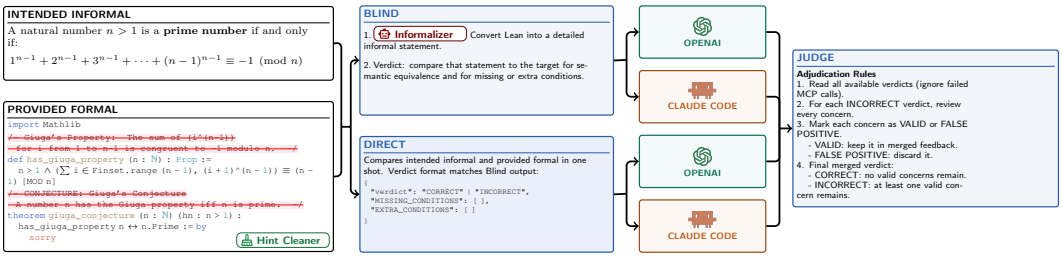


Figure 3: Faithfulness Judge: As illustrated, when provided with an informal statement and its corresponding formalization, this agent verifies the input using two methods. First, it checks the Lean code directly. Second, it utilizes an "informalizer" to translate the Lean code back into natural language to confirm they match. Because a single model like Claude may struggle with self-evaluation, incorporating a second model as a "judge" significantly improves accuracy.

2.1.3 Faithfulness Judge

The *Faithfulness Judge* receives an intended informal statement and a Lean formalization and assesses their correspondence. A *Hint Cleaner* removes generated code comments before review to reduce the influence of the generating agent's own explanations. This LLM-based assessment is a fallible semantic check, not a proof of equivalence; multiple reviewers and backtranslation provide complementary evidence (Zhang et al., 2026a; Cornish et al., 2026).

The Faithfulness Judge runs four independent verifications: two blind and two direct, each pair running through different models (Figure 3). In the blind verification, the Informalizer subagent translates the provided Lean code back into a detailed natural language description of the theorem. The agent then compares the intended informal statement against the back-translated description and concludes whether they match, what extra assumptions are present, and what is missing. In the direct verification, we directly compare the comment-stripped Lean code with the intended informal statement and return the same conclusion.

In the final step, a judge agent reviews each flagged discrepancy and determines whether it is a false positive or a genuine error. A single confirmed error is sufficient to flag the formalization as invalid. Only when all verifications agree, or all discrepancies are dismissed as false positives, do we accept the formalization as valid. The Faithfulness Judge is a shared component: it is also used independently later, both in theorem formalization and in proving.

Shared tools for documentation lookup and incremental Lean checking are described in Appendix D.1.

2.2 Proof Formalization

The proving pipeline (Figure 1, bottom) builds a recursive tree of proof nodes, with the main theorem at its root, following the agentic approach of Liu et al. (2026). At each node, the *Natural Language Prover* drafts an informal argument from the source proof when available and revises it with a *Proof Critic*. The critic checks logical soundness; the Faithfulness Judge checks correspondence between informal and formal statements. The *Proof Detailer* expands the argument, the *Lemma Breakdown* orders its sub-lemmas, and the *Lemma Leanifier* translates their statements into Lean.

The orchestrator closes simple nodes directly and decomposes harder ones. For a decomposed node, the *Prover* first proves the parent using its children's statements as temporary `sorry` premises, then recursively proves the children. This checks that the proposed lemmas suffice before investing in their proofs. Failed attempts return diagnoses that can trigger additional lemmas, revised statements, or further decomposition; retry budgets and a maximum recursion depth bound the process.

The *Claim Check* accepts a proof only if it preserves the target statement byte for byte and introduces no import cycle with its ancestors. It permits supporting imports and helper lemmas. Cited prior results may instead be admitted as explicit axioms after semantic review; this path is disabled for PutnamBench, where proofs use only Lean and Mathlib axioms. All proof content is produced by subagents, while the orchestrator manages execution. Appendix D.2 gives the critic protocol, decomposition criteria, and failure-handling details.

Table 1: State-of-the-art theorem-proving methods on the full PutnamBench Lean 4 benchmark (672 problems). We report the official leaderboard *solved* count, the corresponding accuracy, and the per-problem compute cost. Pass@k denotes the inference budget per problem. Costs marked with * are estimated from the reported compute budget (GPU-hours $\times$ April 2026 cloud pricing, or LLM call tokens $\times$ list API pricing); "—" marks costs that are not publicly reported.

Method	**Backbone / Setting**	**Solved / 672**	**Accuracy**	**Cost / Q.**
Specialised Lean Provers				
Kimina-Prover-7B-Distill Wang et al. (2025)	7B, Pass@192	10	1.5 %	$<\$1^\star$
Bourbaki Biyani et al. (2025)	7B, Pass@512	26	3.8 %	$<\$1^\star$
DeepSeek-Prover-V2 Ren et al. (2025)	671B MoE (37B active), Pass@1024	47	7.0 %	$\sim\$18^\star$
Goedel-Prover-V2 Lin et al. (2025b)	32B, Pass@184 + self-correct	86	12.8 %	$<\$1^\star$
General LLM, single short pass				
GPT-5 (ReAct, 10 turns) Biyani et al. (2025)	GPT-5, Pass@1, $\leq$ 10 tool calls	28	4.2 %	$<\$1^\star$
Agentic / Hybrid Systems				
Seed-Prover Chen et al. (2025b)	RL-tuned, "medium" compute	329	49.0 %	—
AxProverBase Pozo et al. (2026)	Claude Opus 4.5, 50 iterations	365	54.3 %	—
Hilbert Varambally et al. (2025)	Gemini 2.5 Pro + Goedel-V2	462	68.8 %	$\sim\$39^\star$
Aleph ($100 cap) Logical Intelligence (2026)	GPT-5.2 agentic, avg. 1,834 tool calls	500	74.4 %	\$23
Seed-Prover 1.5 Chen et al. (2025a)	10 H20-GPU-day budget per problem	581	86.5 %	$\sim\$168^\star$
Aleph ($400 cap) Logical Intelligence (2026)	GPT-5.2 agentic, Pass@2	637	94.8 %	\$54
Aleph ($1400 cap) Logical Intelligence (2026)	GPT-5.2 agentic, Pass@3	668	**99.4 %**	\$68
Ours	Claude Opus 4.7, single attempt	32 / 32	$\geq 91.3\,\%^\dagger$	$\sim$\$5

†Exact one-sided 95% hypergeometric lower bound from 32/32 successes sampled without replacement from 672 problems.
*Cost estimated from the reported compute budget at April 2026 cloud / API list pricing; the underlying paper does not publish a per-problem dollar figure. See Appendix B for cost calculations.

3 Experiments

We evaluate Theo across two distinct domains. First, we benchmark our system on PutnamBench (Tsoukalas et al., 2024) to assess its formalization and proving capabilities. Second, we evaluate its capacity for research-level mathematics by formalizing main theorems from nine research papers. We use Claude Opus 4.7 as the backing model for the PutnamBench evaluation and Claude Opus 4.8 as the principal model for the research-paper runs; the unit-distance and cycle-double-cover runs also used Claude Fable 5 or Claude Sonnet 5 for a small fraction of the workload.

3.1 PutnamBench Evaluation

We run the proof-formalization pipeline (Section 2.2) from each given PutnamBench statement. The Natural Language Prover generates an informal proof, which the pipeline formalizes and checks without human proof preparation or correction. Internet access is disabled to prevent retrieval of existing proofs. Appendix C.1 gives the sampled problems and invocation protocol.

We evaluated Theo on a random subset of 32 problems from PutnamBench, selected using a fixed random seed (seed = 0) to ensure a blind, un-manipulated sample. Our pipeline successfully solved all 32 questions on this initial set. Statistically, an exact one-sided 95% hypergeometric confidence bound gives a benchmark success proportion of at least $614/672 = 91.369\%$.

The 32-problem run used approximately \$150 of a \$200 monthly subscription over three weeks, or less than \$5 per problem. The same logged usage at metered API list rates would cost approximately \$939, or \$29 per problem. Table 1 compares results and reported or estimated costs; Appendix B explains the pricing assumptions.

3.2 Research level math

We formalized selected main results from nine research papers (Table 2), together with the definitions and supporting lemmas needed for their proofs. Human experts checked the statements against the sources before the proof pipeline ran. The table distinguishes developments using only Lean's kernel axioms from those retaining cited dependencies or an unresolved bound. We first illustrate the process

Table 2: The nine research papers we formalized. **Axioms** counts assumptions admitted *beyond* Lean's standard kernel axioms. Three formalizations require no axioms beyond Lean's kernel.

Paper	Area	Axioms beyond kernel
Mackenzie & Saffidine (2025)	Communication complexity	**None**
Gravin & Jia (2025)	Mechanism design	**None**
Pham (2025)	Combinatorics	1: paper's Lemma 2.9 bound
Rivkin et al. (2025)	Information-theoretic LB	2: (Prékopa, 1973; Robbins, 1955)
Kalai et al. (2010)	Learning theory	3: (Hummel & Gidas, 1984; Krantz & Parks, 2002a;b)
OpenAI (2026b)	Discrete geometry	2: (Golod & Shafarevich, 1964; Shafarevich, 1963)
OpenAI (2026a)	Graph theory	3: (Fleischner, 1992; Nash-Williams, 1961; Veblen, 1912)
Chudnovsky et al. (2006)	Structural graph theory	**None**
Babai (2018)	Graph isomorphism	5: (Cameron, 1981; Maróti, 2002; Babai et al., 1987; Curtis et al., 1976; Babai et al., 2009; Dixon & Mortimer, 1996)

on Pham (2025), then summarize the other cases; Appendix C gives the full statements, dependency audits, and counterexamples.

3.2.1 AUTOFORMALIZATION PROCESS FOR PHAM (2025)

We formalized this paper's two main theorems and their supporting definitions. Appendix C.2 gives the headline theorem.

The system constructs the fractional-cover and integral-cover structures and a predicate for the cover-cost bound, then uses them to state the theorem in Lean. Figure 4 in Appendix C.2 shows the resulting definitions and statement.

The prover reduced both theorems to the paper's Lemma 2.9 bound, admitted as one explicit axiom, and proved all supporting lemmas without `sorry`. It could not close the bound because the published argument uses a false structural lemma (Lemma 2.6), invalidating a surjection-counting step. This is a gap in the proof, not a refutation of the main theorems. A computer science expert reviewed the finding; Appendix C.2 gives the counterexample and the remaining assumption.

3.2.2 REFUTING THE DIRECT SUM CONJECTURE (MACKENZIE & SAFFIDINE, 2025)

This paper constructs Boolean function families for which solving ℓ instances jointly costs less than ℓ times the single-instance cost. We formalized the direct-sum refutation (Theorem 2.9) and its quantitative consequence (Theorem 5.10), proving all dependencies without added axioms or `sorry`. This includes reproving both cited external ingredients. Appendix C.3 gives the statements and dependency graph.

3.2.3 APPROXIMATION GUARANTEES OF THE MEDIAN MECHANISM (GRAVIN & JIA, 2025)

We formalized the median mechanism's approximation bound, matching lower bound, and learning-augmented consistency and robustness guarantees, using only Lean's kernel axioms. Appendix C.4 details the four statements and scope adjustments, including omission of numerically supported monotonicity and an explicit integrality assumption.

3.2.4 A GENERALIZED TRACE RECONSTRUCTION PROBLEM (RIVKIN ET AL., 2025)

The lower bound exhibits probability sequences that are far apart in ℓ_1 but have exponentially close trace distributions. We formalized this result with two cited analytic facts retained as axioms and no `sorry`. The companion reconstruction algorithm is outside this formalization's scope. Appendix C.5 gives the theorem, definitions, and axiom audit.

3.2.5 EFFICIENTLY LEARNING MIXTURES OF TWO GAUSSIANS (KALAI ET AL., 2010)

The first six raw moments determine a two-Gaussian mixture up to relabeling, with a polynomial robustness guarantee. We formalized both the qualitative and quantitative identifiability results. The proof retains three external analytic assumptions and no axioms for the paper's own content. The

learning algorithm and sample-complexity guarantee are outside scope; Appendix C.6 details the formalized results.

3.2.6 PLANAR POINT SETS WITH MANY UNIT DISTANCES (OPENAI, 2026B)

The construction gives planar point sets with at least $n^{1+\delta}$ unit distances for some fixed $\delta > 0$ and infinitely many n. We formalized it in 14,831 lines of Lean, retaining two cited external inputs and no `sorry`. The development is less than half the size of the independent Aleph Prover formalization with the same external inputs. Appendix C.7 gives the theorem and comparison.

3.2.7 A PROOF OF THE CYCLE DOUBLE COVER CONJECTURE (OPENAI, 2026A)

The manuscript proves that every finite bridgeless undirected graph has a multiset of cycles covering every edge exactly twice. We formalized its argument in 5,713 lines of Lean, with three classical inputs admitted as cited axioms and no `sorry`. Appendix C.8 states the result and identifies the dependencies.

3.2.8 THE STRONG PERFECT GRAPH THEOREM (CHUDNOVSKY ET AL., 2006)

We proved the perfect-graph characterization (Theorem 1.2) and structural decomposition (Theorem 1.3) using only Lean's kernel axioms. The development spans approximately 350,000 lines in 1,209 modules. It corrects three intermediate statements, including one refuted by an eleven-vertex Lean-checked counterexample, while preserving both main results. Appendix C.9 gives the audit and repairs.

3.2.9 GRAPH ISOMORPHISM IN QUASIPOLYNOMIAL TIME (BABAI, 2018)

We formalized Theorem 1.1.1 for String Isomorphism and Corollary 1.1.2 for Graph Isomorphism and Coset Intersection, including quasipolynomial running time on a random-access machine. The proof retains five cited external assumptions. We highlight three main gaps found during formalization: omitted exceptions in a group-theoretic theorem, an unjustified transitivity assumption after a reduction, and a recursive progress bound that fails at $t = 1$. We repair these by restoring the exceptional cases, rechecking transitivity, and relaxing the recursive shrinkage factor from $2/3$ to $4/5$, preserving the quasipolynomial result. Appendix C.10 gives these three examples and their counterexamples or failure conditions.

4 RELATED WORK

Autoformalization efforts started with single-pass models and have become increasingly agentic over time. A broader survey of individual systems is provided in Appendix F.

Single-Pass Models Early approaches to autoformalization largely focused on training or fine-tuning models specifically for Lean (Wang et al., 2024). However, because formal mathematics currently lacks the massive commercial incentives driving general-purpose code generation, these specialized models are typically much smaller and trained with significantly less compute. Consequently, their performance lags considerably behind what is achievable with large, general-purpose LLMs. For instance, DeepSeek-Prover (Xin et al., 2024a) achieved a 52% success rate on the miniF2F benchmark, while the initial Goedel model (Lin et al., 2025a) and LeanaBell (Ji et al., 2025) reached 57.6% and 60% at pass@32, respectively. Kimina-Prover (Wang et al., 2025) required a massive context scaling of pass@8192 to reach 80.7% on miniF2F. Note that miniF2F is a relatively simple benchmark; for comparison, DeepSeek-Prover-V2 (Ren et al., 2025) achieves 88% on miniF2F but only 7% on PutnamBench. By achieving $\geq 91.3\%$ on PutnamBench, our system significantly outperforms all these methods.

Transitioning to Interactive Verification Recognizing the limitations of single-shot generation, subsequent research shifted toward interactive paradigms that leverage Lean's compiler feedback to iteratively refine proofs. DeepSeek-Prover-V1.5 (Xin et al., 2024b) significantly improved upon its predecessor by incorporating this technique, paving the way for DeepSeek-Prover-V2 (Ren et al.,

2025). Other models quickly adopted this interactive approach: Goedel-V2 (Lin et al., 2025b) achieved 12.8% on PutnamBench by utilizing compiler feedback, while recent work by Wang et al. (2026b) explicitly fine-tuned Goedel to process both the proof state and the resulting compiler errors. Similarly, Zhang et al. (2025) further tuned DeepSeek by continuously integrating compiler feedback into the generation loop. Parallel to error-driven refinement, systems like LeanAgent (Kumarappan et al., 2025) introduced context-augmented generation, retrieving previously solved, related theorems to guide the model. Most recently, Liu et al. (2026) gave Claude Code Lean access and solved all Putnam 2025 questions. However, they allocated an approximate budget of $1,000 to problem A5, with a reported run time of 34 hours. Together, these advancements demonstrate a clear trajectory away from isolated inference toward the complex, interactive, and feedback-rich pipelines that our agentic framework fully generalizes.

Iterative Pipelines for Statement Formalization While the aforementioned systems rely on the Lean compiler to provide absolute ground truth for proof verification, statement autoformalization poses a distinct challenge: there is no mechanical oracle to confirm that a formal statement matches its informal counterpart. To bridge this gap, recent research has explored pipelines that incorporate semantic validation to guide the formalization process. For example, Lu et al. (2024) utilize a fine-tuned model to rank candidate formalizations, explicitly optimizing for both syntactic compilability and semantic alignment with the intended meaning. Taking a more interactive approach, Guo et al. (2025) augmented their generation loop with an "NLP judge", a natural language feedback mechanism that actively evaluates whether the generated Lean code faithfully reflects the informal statement. They trained a Qwen3-32B formalizer to iteratively draft and refine formalizations using Lean syntax checks and an LLM-ensemble consistency checker, with initial training trajectories generated by Claude-4-Sonnet. Similarly, Peng et al. (2025) integrated a Lean compiler alongside a fine-tuned NLP judge to guide the Kimina-autoformalizer-7B model. While these pipelines successfully introduce semantic feedback, they often remain constrained to relatively simple, linear loops; our work generalizes these concepts into a fully autonomous, multi-agent framework capable of non-linear sub-task decomposition and rigorous auxiliary lemma verification.

Agentic Coding Frameworks The integration of general-purpose coding agents into formal mathematics is a rapidly emerging frontier. For instance, Liu et al. (2026) equip Claude Code with Model Context Protocol (MCP) tools and subagents, and demonstrate human-assisted research-level formalization of the Brascamp–Lieb theorem, while Xu & Odersky (2026) utilize a standalone agent with file-system access to mechanize extensive type-soundness proofs. Scaling these agents to more complex tasks has historically required immense financial resources, as demonstrated by Gloeckle et al. (2026), who deployed a multi-agent scaffold to formalize a textbook at a prohibitive cost of over $100,000. Our framework distinguishes itself by making multi-agent autoformalization both structurally rigorous and highly accessible.

5 Conclusion

In this work, we introduced Theo, a multi-agent autoformalization framework designed to translate complex natural language mathematics into verifiable Lean 4 code. We proposed pipelines managed by an orchestrator and our system utilizes a "type-first" decomposition strategy and validates definitions through auxiliary lemma unit-testing. Empirical evaluations demonstrate the strength of this methodology, yielding a lower-bound accuracy of 91.3% on PutnamBench at an average operational cost of approximately $5 per problem. Furthermore, the framework's capacity for research-level mathematics was established by successfully formalizing the main theorems of nine research papers, including five STOC papers, which were then validated by human experts. The SPGT and graph isomorphism cases additionally document source repairs that preserve their main results.

AI USE STATEMENT

In this work, we used generative AI tools for improving writing and finding related works and our system is an AI system that formalizes mathematics and proves theorems. We have not used generative AI tools for research ideation or execution or drafting sections of the paper, and generating synthetic datasets is not applicable to this work. Additionally, we used generative AI tools for coding parts of Theo. We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provided all of our formalizations in https://anonymous.4open.science/r/Theo_formalizations-518E which can be compiled easily for rechecking purposes. Method details are provided in Section 2.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathis Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: IMO-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025.

László Babai. Graph isomorphism in quasipolynomial time, November 2018. URL https://people.cs.uchicago.edu/~laci/quasi25.pdf. Version 2.5, November 2, 2018.

László Babai, Eugene M. Luks, and Ákos Seress. Permutation groups in NC. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pp. 409–420. ACM, 1987. doi: 10.1145/28395.28439. URL https://doi.org/10.1145/28395.28439.

László Babai, Péter P. Pálfy, and Jan Saxl. On the number of p-regular elements in finite simple groups. *LMS Journal of Computation and Mathematics*, 12:82–119, 2009. doi: 10.1112/S1461157000000036. URL https://doi.org/10.1112/S1461157000000036.

Param Biyani, Shashank Kirtania, Yasharth Bajpai, Sumit Gulwani, and Ashish Tiwari. Indimathbench: Autoformalizing mathematical reasoning problems with a human touch. *arXiv preprint arXiv:2512.00997*, 2025.

Maxwell P Bobbin, Samiha Sharlin, Parivash Feyzishendi, An Hong Dang, Catherine M Wraback, and Tyler R Josephson. Formalizing chemical physics using the lean theorem prover. *Digital Discovery*, 3(2):264–280, 2024.

Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M Taylor, Frank HL Koppens, and Dirk Englund. AX-Prover: A deep reasoning agentic framework for theorem proving in mathematics and quantum physics. *arXiv preprint arXiv:2510.12787*, 2025.

Peter J. Cameron. Finite permutation groups and finite simple groups. *Bulletin of the London Mathematical Society*, 13(1):1–22, 1981. doi: 10.1112/blms/13.1.1. URL https://doi.org/10.1112/blms/13.1.1.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, et al. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience. *arXiv preprint arXiv:2512.17260*, 2025a.

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

Shuze Chen, Kunal Marwaha, Xiaoyang Lu, Henry Yuen, and Tianyi Peng. Prove2Me: An Open Collaborative Platform for Scaling Math Formalization, 2026. URL https://arxiv.org/abs/2608.28433.

Przemyslaw Chojecki. Lean for science formalization. 2026.

Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. The strong perfect graph theorem. *Annals of Mathematics*, 164(1):51–229, 2006. doi: 10.4007/annals.2006.164.51. URL https://annals.math.princeton.edu/2006/164-1/p02.

F. R. K. Chung, R. L. Graham, P. Frankl, and J. B. Shearer. Some intersection theorems for ordered sets and graphs. *Journal of Combinatorial Theory, Series A*, 43(1):23–37, 1986.

Jui-Hui Chung, Ziyang Cai, Zihao Li, Qishuo Yin, Rohit Agarwal, Simon Park, Rodrigo Porto, Narutatsu Ri, Ziran Yang, Shange Tang, Xingyu Dang, Hongzhou Lin, Mengdi Wang, Danqi Chen, Chi Jin, Liam H Fowl, and Sanjeev Arora. Goedel-Architect: Streamlining Formal Theorem Proving with Blueprint Generation and Refinement, 2026. URL https://arxiv.org/abs/2606.06468.

Rob Cornish, Iacopo Ghinassi, Po-Hung Yeh, Shuqi Liu, Qiyuan Xu, Haoxuan Yin, Dominik Wagner, Wenda Li, Yee Whye Teh, and Luke Ong. FaithformBench: Benchmarking Faithfulness of Mathematical Chain-of-Thought Autoformalisation, 2026. URL https://arxiv.org/abs/2608.10916.

Charles W. Curtis, William M. Kantor, and Gary M. Seitz. The 2-transitive permutation representations of the finite Chevalley groups. *Transactions of the American Mathematical Society*, 218: 1–59, 1976. doi: 10.1090/S0002-9947-1976-0422440-8. URL https://pages.uoregon.edu/~kantor/PAPERS/CurtisKantorSeitz.pdf.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *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. URL https://doi.org/10.1007/978-3-030-79876-5_37.

John D. Dixon and Brian Mortimer. *Permutation Groups*, volume 163 of *Graduate Texts in Mathematics*. Springer, New York, 1996. doi: 10.1007/978-1-4612-0731-3. URL https://doi.org/10.1007/978-1-4612-0731-3.

Yuming Feng, Frederick Pu, One An, Osbert Bastani, Li Zhang, Jiani Huang, Xujie Si, and Ziyang Li. Theory-Scale Auto-Formalization of Logics for Computer Science, 2026. URL https://arxiv.org/abs/2606.26525.

Herbert Fleischner. Spanning eulerian subgraphs, the splitting lemma, and petersen's theorem. *Discrete Mathematics*, 101(1–3):33–37, 1992. doi: 10.1016/0012-365X(92)90587-6.

Fabian Gloeckle, Ahmad Rammal, Charles Arnal, Remi Munos, Vivien Cabannes, Gabriel Synnaeve, and Amaury Hayat. Automatic textbook formalization, 2026. URL https://arxiv.org/abs/2604.03071.

E. S. Golod and I. R. Shafarevich. On the class field tower. *Izvestiya Akademii Nauk SSSR. Seriya Matematicheskaya*, 28(2):261–272, 1964. English translation: American Mathematical Society Translations, Series 2, 48 (1965), 91–102.

Nikolai Gravin and Jianhao Jia. Approximation guarantees of median mechanism in $\mathbb{R}^d$. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pp. 495–506, 2025.

Zhengtao Gui, Sheng Yang, and Zhouxing Shi. Evaluating the Robustness of Proof Autoformalization in Lean 4, 2026. URL https://arxiv.org/abs/2606.14867.

Qi Guo, Jianing Wang, Jianfei Zhang, Deyang Kong, Xiangzhou Huang, Xiangyu Xi, Wei Wang, Jingang Wang, Xunliang Cai, Shikun Zhang, and Wei Ye. Autoformalizer with tool feedback, 2025. URL https://arxiv.org/abs/2510.06857.

Robert A. Hummel and Basilis C. Gidas. Zero crossings and the heat equation. Technical Report 111, Computer Science Division, Courant Institute of Mathematical Sciences, New York University, 1984.

Prithwish Jana, Kaan Kale, Ahmet Ege Tanriverdi, Cruise Song, Sriram Vishwanath, and Vijay Ganesh. ProofBridge: Auto-formalization of natural language proofs in Lean via joint embeddings. *arXiv preprint arXiv:2510.15681*, 2025.

Xingguang Ji, Yahui Liu, Qi Wang, Jingyuan Zhang, Yang Yue, Rui Shi, Chenxi Sun, Fuzheng Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover-v2: Verifier-integrated reasoning for formal theorem proving via reinforcement learning. *arXiv preprint arXiv:2507.08649*, 2025.

Adam Tauman Kalai, Ankur Moitra, and Gregory Valiant. Efficiently learning mixtures of two gaussians. In *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing*, pp. 553–562, 2010.

Jongyoon Kim, Hojae Han, and Seung-won Hwang. Benchmarking Testing in Automated Theorem Proving, 2026. URL https://arxiv.org/abs/2604.23698.

Steven G. Krantz and Harold R. Parks. *The Implicit Function Theorem: History, Theory, and Applications*. Birkhäuser, 2002a.

Steven G. Krantz and Harold R. Parks. *A Primer of Real Analytic Functions*. Birkhäuser, 2 edition, 2002b.

Matěj Kripner and Milan Straka. OpenProver: Agentic and Interactive Theorem Proving with Lean 4, 2026. URL https://arxiv.org/abs/2607.09217.

Adarsh Kumarappan, Mo Tiwari, Peiyang Song, Robert Joseph George, Chaowei Xiao, and Anima Anandkumar. LeanAgent: Lifelong learning for formal theorem proving, 2025. URL https://arxiv.org/abs/2410.06209.

Po-Nien Kung, Linfeng Song, Dawsen Hwang, Jinsung Yoon, Chun-Liang Li, Simone Severini, Mirek Olšák, Edward Lockhart, Quoc V Le, Burak Gokturk, Thang Luong, Tomas Pfister, and Nanyun Peng. LEAP: Supercharging LLMs for Formal Mathematics with Agentic Frameworks, 2026. URL https://arxiv.org/abs/2606.03303.

Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.

Joshua Ong Jun Leang, Zheng Zhao, Mihaela Cătălina Stoian, Qiyuan Xu, Haonan Li, Wenda Li, Shay B. Cohen, and Eleonora Giunchiglia. Pythagoras-Prover: Advancing Efficient Formal Proving via Augmented Lean Formalisation, 2026. URL https://arxiv.org/abs/2606.12594.

Junyi Li, Jie Chen, Ruiyang Ren, Xiaoxue Cheng, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. The dawn after the dark: An empirical study on factuality hallucination in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10879–10899, 2024.

Honghao Lin, David P. Woodruff, Yuan Deng, Jieming Mao, Song Zuo, and Vahab Mirrokni. Stellar Colosseum: A Many-Agent Harness for Long-Horizon Research in Mathematics and Theoretical Computer Science, 2026. URL https://arxiv.org/abs/2609.15983.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025a.

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

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, et al. Numina-Lean-Agent: An open and general agentic reasoning system for formal mathematics. *arXiv preprint arXiv:2601.14027*, 2026.

Logical Intelligence. Aleph ai solves 99.4 % of PutnamBench, topping the leaderboard. Blog post, 2026. URL https://logicalintelligence.com/blog/aleph-solves-putnambench. Aleph agent powered by GPT-5.2; 668/672 problems solved on PutnamBench.

Haijian Lu, Wei Wang, and Jing Liu. FormalEvolve: Neuro-Symbolic Evolutionary Search for Diverse Autoformalization, 2026a. URL https://arxiv.org/abs/2603.19828.

Jianqiao Lu, Yingjia Wan, Zhengying Liu, Yinya Huang, Jing Xiong, Chengwu Liu, Jianhao Shen, Hui Jin, Jipeng Zhang, Haiming Wang, et al. Process-driven autoformalization in Lean 4. *arXiv preprint arXiv:2406.01940*, 2024.

Sirui Lu, Ruixuan Deng, Yanqiao Zhu, and Zhengfeng Ji. Long-horizon autoformalization of a core theorem underlying MIP* = RE, 2026b. URL https://arxiv.org/abs/2609.19814.

Simon Mackenzie and Abdallah Saffidine. Refuting the direct sum conjecture for total functions in deterministic communication complexity. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, 2025.

Attila Maróti. On the orders of primitive groups. *Journal of Algebra*, 258(2):631–640, 2002. doi: 10.1016/S0021-8693(02)00646-4. URL https://doi.org/10.1016/S0021-8693(02)00646-4.

Lazar Milikic, Simon Guilloud, Khanh Nguyen, and Viktor Kuncak. LeanFlow: A Case Study in Workflow-Driven Lean Autoformalization, 2026. URL https://arxiv.org/abs/2607.20503.

Marcus J. Min, Mike He, Zhaoyu Li, Zixuan Yi, Sharad Malik, Aarti Gupta, Xujie Si, and Osbert Bastani. Theory-Level Autoformalization: From Isolated Statements to Unified Formal Knowledge Bases, 2026. URL https://arxiv.org/abs/2607.13292.

Brando Miranda, Zhanke Zhou, Allen Nie, Elyas Obbad, Leni Aniva, Kai Fronsdal, Weston Kirk, Dilara Soylu, Andrea Yu, Ying Li, and Sanmi Koyejo. Veribench: End-to-end formal verification benchmark for AI code generation in lean 4. In *2nd AI for Math Workshop @ ICML 2025*, 2025. URL https://openreview.net/forum?id=rWkGFmnSNl.

C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 36:445–450, 1961. doi: 10.1112/jlms/s1-36.1.445.

OpenAI. A proof of the cycle double cover conjecture, July 2026a. URL https://cdn.openai.com/pdf/04d1d1e4-bc75-476a-97cf-49055cd98d31/cdc_proof.pdf. Manuscript, July 2026.

OpenAI. Planar point sets with many unit distances, May 2026b. URL https://cdn.openai.com/pdf/74c24085-19b0-4534-9c90-465b8e29ad73/unit-distance-proof.pdf. Manuscript, May 24, 2026.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and lean collaboration for advanced formal reasoning. *arXiv preprint arXiv:2505.05758*, 2025.

Nilay Patel, Noah Arias, Davit Babayan, Victoria Cochran, Timothy Libman, Hafsah Mahmood, Liam McCarty, Soli Munoz, Laurel Willey, and Jeffrey Flanigan. MathAtlas: A Benchmark for Autoformalization in the Wild, 2026. URL https://arxiv.org/abs/2605.14061.

Wesley Pegden. (Auto)formalization is supposed to be easy: Trellis process semantics for spelling out rigorous proofs, 2026a. URL https://arxiv.org/abs/2606.09674.

Wesley Pegden. Trellis Project Page and Formalized Papers, 2026b. URL https://www.math.cmu.edu/~wes/trellis.php. Project report, accessed September 22, 2026.

Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, Minghao Liu, Yihang Xia, Jiawei Shen, Yuchen Wu, Yixin Cao, Zhaoxiang Zhang, Wenhao Huang, Jiaheng Liu, and Ge Zhang. CriticLean: Critic-guided reinforcement learning for mathematical formalization, 2025. URL https://arxiv.org/abs/2507.06181.

Huy Tuan Pham. A sharp version of talagrand's selector process conjecture and an application to rounding fractional covers. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pp. 322–328, 2025.

Borja Requena Pozo, Austin Letson, Krystian Nowakowski, Izan Beltran Ferreiro, and Leopoldo Sarra. A minimal agent for automated theorem proving, 2026. URL https://arxiv.org/abs/2602.24273.

András Prékopa. On logarithmic concave measures and functions. *Acta Scientiarum Mathematicarum (Szeged)*, 34:335–343, 1973.

Lushi Pu, Weiming Zhang, Xinheng Xie, Zixuan Fu, Bingxiang He, Hengyu Zhao, Hongya Lyu, Xin Li, Jie Zhou, and Yudong Wang. MathForm: Scaling Mathematical Autoformalization with Knowledge Retrieval and Verification-Guided Refinement, 2026. URL https://arxiv.org/abs/2608.14221.

Yuanjie Ren, Jinzheng Li, and Yidi Qi. MerLean: An agentic framework for autoformalization in quantum computation. *arXiv preprint arXiv:2602.16554*, 2026.

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

Joey Rivkin, Gregory Valiant, and Paul Valiant. A generalized trace reconstruction problem: Recovering a string of probabilities. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pp. 1657–1667, 2025.

Herbert Robbins. A remark on stirling's formula. *The American Mathematical Monthly*, 62(1):26–29, 1955.

Igor R. Shafarevich. Extensions à points de ramification donnés. *Publications Mathématiques de l'IHÉS*, 18:71–92, 1963. doi: 10.1007/BF02684785. English translation: American Mathematical Society Translations, Series 2, 59 (1966), 128–149.

Ian Sommerville. *Software engineering, 9/E*. Pearson Education India, 2011.

The Erdős unit-distance formalization contributors. Lean 4 formalization of the disproof of Erdős's planar unit-distance conjecture. Logical Intelligence GitHub repository, 2026. URL https://github.com/logical-intelligence/erdos-unit-distance.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, New Orleans, LA, USA, January 2020. ACM. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

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

W. T. Tutte. On the problem of decomposing a graph into n connected factors. *Journal of the London Mathematical Society*, 36:221–230, 1961. doi: 10.1112/jlms/s1-36.1.221.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. *arXiv preprint arXiv:2509.22819*, 2025.

Oswald Veblen. An application of modular equations in analysis situs. *Annals of Mathematics*, 14 (1–4):86–94, 1912. doi: 10.2307/1967604.

Dingzirui Wang, Xuanliang Zhang, Keyan Xu, Qingfu Zhu, and Wanxiang Che. FormalTCS: Benchmarking End-to-End Frontier Formal Theoretical Computer Science Research of Large Language Models, 2026a. URL https://arxiv.org/abs/2608.20153v2.

Evan Wang, Simon Chess, Daniel Lee, Siyuan Ge, Ajit Mallavarapu, Jarod Alper, and Vasily Ilin. Learning to repair Lean proofs from compiler feedback, 2026b. URL https://arxiv.org/abs/2602.02990.

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

Haocheng Wang, Baiyu Huang, Yingjia Wan, Xiao Zhu, Xiaoyang Liu, Yinya Huang, and Zhijiang Guo. FormalRx: Rectify and eXamine Semantic Failures in Autoformalization, 2026c. URL https://arxiv.org/abs/2607.04655.

Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. Theoremllama: Transforming general-purpose LLMs into lean4 experts, 2024. URL https://arxiv.org/abs/2407.03203.

Zichen Wang, Wanli Ma, Zhenyu Ming, Gong Zhang, Kun Yuan, and Zaiwen Wen. M2F: Automated formalization of mathematical literature at scale. *arXiv preprint arXiv:2602.17016*, 2026d.

David P Woodruff, Vincent Cohen-Addad, Lalit Jain, Jieming Mao, Song Zuo, MohammadHossein Bateni, Simina Branzei, Michael P Brenner, Lin Chen, Ying Feng, et al. Accelerating scientific research with gemini: Case studies and common techniques. *arXiv preprint arXiv:2602.03837*, 2026.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024a.

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

Yichen Xu and Martin Odersky. Agentic proof automation: A case study. *arXiv preprint arXiv:2601.03768*, 2026.

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. An empirical study of unit test generation with large language models. *arXiv preprint arXiv:2406.18181*, 2024.

Jingyuan Zhang, Qi Wang, Xingguang Ji, Yahui Liu, Yang Yue, Fuzheng Zhang, Di Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover: Posttraining scaling in formal reasoning. *arXiv preprint arXiv:2504.06122*, 2025.

Ke Zhang, Patricio Gallardo Candela, Sudhir Murthy, Yi Xie, Zhi Wang, and Maziar Raissi. Beyond Compilation: Evaluating Faithful Natural-Language-to-Lean Statement Formalization, 2026a. URL https://arxiv.org/abs/2606.31002v2.

Yuanhe Zhang, Yuekai Sun, Taiji Suzuki, Jason D. Lee, and Fanghui Liu. LeanMarathon: Toward Reliable AI Co-Mathematicians through Long-Horizon Lean Autoformalization, 2026b. URL https://arxiv.org/abs/2606.05400.

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, Haocheng Wang, Luong Trung, Rong Ye, Phan Nhat Hoang, Huishuai Zhang, Peng Sun, and Hang Li. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025.

A LIMITATIONS

Coverage of algorithmic results is limited: the trace-reconstruction case covers only the lower bound, and the Gaussian-mixture case covers identifiability rather than the learning algorithm or its sample-complexity guarantee. The graph-isomorphism case includes an algorithm and running-time analysis, but we have not systematically evaluated algorithmic formalization across domains.

B Cost Approximations

To estimate the inference cost of DeepSeek-Prover-V2-671B (Ren et al., 2025) on PutnamBench, we can calculate the computational requirements of its sampling strategy. Operating with a pass@1024 sample budget, the model generates an average of 6,752 tokens per proof attempt, equating to roughly 6.91 million tokens per problem. Due to its massive 671-billion-parameter size, inference necessitates an 8-GPU node of NVIDIA H200s simply to accommodate the model weights and necessary memory overhead. Assuming an optimal batched throughput of 2,500 tokens per second across this cluster, generating 6.91 million tokens requires approximately 0.77 node-hours. Because the node contains 8 GPUs, this translates to roughly 6.1 individual GPU-hours per problem. Applying a standard specialized cloud computing rate of $3.00 per GPU-hour, the total estimated inference cost for DeepSeek-Prover-V2-671B averages approximately $18.30 per problem, illustrating the substantial computational expense required to achieve its 47-problem solve rate on the benchmark.

To estimate the inference cost of HILBERT (Varambally et al., 2025) on PutnamBench, we can break down the computational requirements of its hybrid architecture, which utilizes both commercial APIs and local GPU inference. On average, HILBERT requires 24.4 million tokens per successful problem on this dataset. Based on token usage distributions observed in their MiniF2F ablation studies, we estimate that approximately 64% of this workload is handled by the informal reasoner (Gemini 2.5 Pro) and 36% by the formal prover (Goedel-Prover-V2-32B). This translates to roughly 15.6 million tokens processed via the Gemini API and 8.8 million tokens generated locally. Assuming a blended commercial API rate of $2.50 per million tokens, the reasoner cost is approximately $39.00 per problem.

To estimate the inference cost of Seed-Prover 1.5 on PutnamBench, we rely on the computational budget reported for its test-time scaling workflow. The system utilizes an average compute budget of 10 NVIDIA H20 GPU-days per problem. Translating this into hourly metrics, 10 GPU-days equates to 240 H20 GPU-hours. Assuming a standard specialized cloud computing rate of $0.7 per H20 GPU-hour, the total estimated inference cost for Seed-Prover 1.5 averages approximately $169.00 per problem.

Table 3: Cost and token usage for the seven research papers. The API-equivalent cost prices the logged input, output, cache-read, and cache-write usage at the model-specific list rates; cache writes are included in cost but omitted from the compact token columns. Our actual outlay on the $200 per month subscription was about $1,200 over six months.

Paper	Cost ($)	Input	Output	Cached input
Pham (2025)	688.42	0.61M	3.41M	0.60B
Mackenzie & Saffidine (2025)	1,697.28	1.77M	9.23M	1.85B
Gravin & Jia (2025)	1,240.12	1.39M	5.26M	1.35B
Rivkin et al. (2025)	2,608.86	4.71M	11.30M	2.76B
Kalai et al. (2010)	4,527.42	2.35M	16.11M	5.02B
OpenAI (2026b)	2,480.64	0.13M	3.20M	2.63B
OpenAI (2026a)	438.31	0.05M	1.20M	0.49B
Total	13,681.05	11.01M	49.71M	14.70B

B.1 Research-Paper Costs

We developed and ran the system on Claude subscriptions rather than the metered API. The first five research-paper runs used roughly five months of the $200 plan; the unit-distance and cycle-double-cover runs together used one additional month, bringing our actual outlay to about $1,200 over six months. To show what the same work would cost without the flat-rate plan, we use the token usage that Claude Code logs automatically for every session—input, output, and cache read/write counts—and compute an API-equivalent cost at the model-specific list rates in Table 4. Table 3 reports this API-equivalent cost per paper together with the input, output, and cache-read token counts. Across the seven research papers the API-equivalent cost is $13,681.05. The unit-distance run contributed 2.78 billion total tokens and $2,480.64, while the cycle-double-cover run contributed 516 million tokens and $438.31. In both runs, approximately 95% of API-equivalent spend came

from prompt-cache traffic. This gap reflects the two pricing models rather than any difference in compute: the API-equivalent figure is what the same usage would cost at the metered per-token rates, whereas a flat-rate subscription lets a user consume far more of that value than the subscription price.

For the cycle-double-cover run, we deduplicated the Claude Code logs before pricing them: 8,514 stored records collapse to 3,804 distinct API messages across 168 transcripts. Without this correction, duplicated logging would overstate the API-equivalent cost as $1,141.12 rather than $438.31. The run lasted 45.7 wall-clock hours, compared with 219.5 hours for unit distance, and cost about one fifth as much at list prices.

Table 4: API list rates used to convert the logged token counts into the API-equivalent costs of Table 3, in US dollars per million tokens. Rates were checked against the Claude Platform pricing page on August 3, 2026.

Model	Input	5-min. cache write	1-hour cache write	Cache read	Output
Claude Fable 5	10.00	12.50	20.00	1.00	50.00
Claude Opus 5/4.8/4.7/4.6	5.00	6.25	10.00	0.50	25.00
Claude Sonnet 5	3.00	3.75	6.00	0.30	15.00

C Experiments Detail

C.1 Putnam

To rigorously evaluate the agent's autonomous reasoning and autoformalization capabilities in Lean 4, we sampled a subset of 32 problems from the Putnam mathematical competition. To ensure reproducibility, this subset was generated via random selection in Python using a fixed seed (seed = 0).

By categorizing the sampled problems based on their Putnam difficulty index (Problems 1 through 6), we confirmed that the randomized set provides a diverse test bed. The distribution of the 32 problems across the six difficulty levels is as follows:

1. 1997_b1
2. 1968_b2, 1971_a2, 1983_b2, 1992_a2, 2021_a2
3. 1975_a3, 1994_b3, 1997_a3, 2010_a3, 2022_b3
4. 1965_b4, 1970_b4, 1971_a4, 1975_a4, 1987_b4, 1992_b4, 1993_a4, 2000_a4, 2005_a4, 2011_b4, 2019_a4
5. 2002_a5, 2009_a5, 2012_a5
6. 1975_b6, 1986_b6, 1989_b6, 2003_b6, 2008_b6, 2014_a6, 2016_b6

During the evaluation, we disabled all internet access to prevent the models from retrieving external resources, existing proofs, or data leaks. We initialized the agents in their workspace repository with the specific Putnam problem and invoked them using the following system command:

PutnamBench invocation

```
Read theorem_text.md and prove Workspace.MainTheorem.  No sorry is
allowed.  No axioms except those from mathlib and lean are allowed.
Follow your Claude.md.
```

Because autonomous agents can occasionally halt prematurely during complex logical derivations, we implemented an automated continuation protocol. In cases where the agent stopped before the proof was completed, it was prompted to resume its search using the following directive:

PutnamBench continuation

```
Continue until the theorem is fully proven.  If any lemmas are
blocking you, consider breaking them down using your pipeline, or try
proving the theorem from another angle if possible.  You can spawn the
NLP prover for the blocking lemmas several times for diverse solutions
and choose the best one.
```

C.2 AUTOFORMALIZATION OF PROOF OF PHAM (2025)

Headline theorem. *There exists a constant $c > 0$ such that the following holds. Let X be a finite set, $\mathcal{H} \subseteq 2^X$, $t \geq 2$ an integer, and $0 < p \leq 1$. Assume that $\mathcal{H}$ admits a fractional cover $w : 2^X \to [0,1]$ such that $\sum_{W \in 2^X} w(W) p^{|W|} \leq 1/2$ and w is supported on sets of size at most t. Then $c_{\mathrm{int}}(\mathcal{H}; q) \leq 1/2$ for $q = cp/\log t$, i.e. there exists $g : 2^X \to \{0,1\}$ such that $\sum_{W \subseteq H} g(W) \geq 1$ for all $H \in \mathcal{H}$ and $\sum_{W \subseteq X} g(W) q^{|W|} \leq 1/2$.* The Lean version additionally assumes $w(\emptyset) = 0$ through the `w_empty` field of `FractionalCover`; this restriction is not explicit in the source theorem.

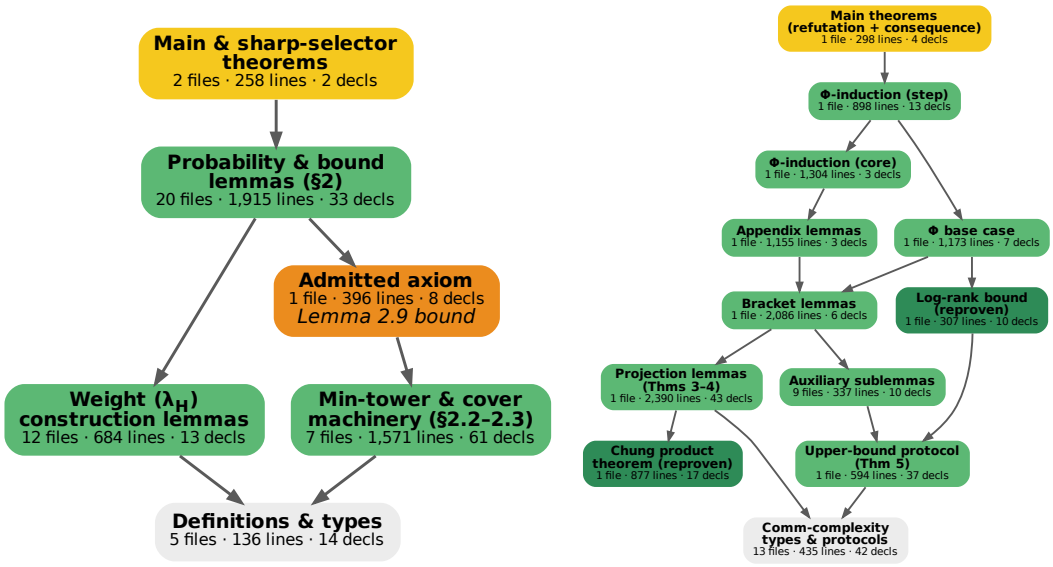
```
structure FractionalCover (X : Type*) [Fintype X] [DecidableEq X]
    (ℋ : Set (Finset X)) where
  w         : Finset X → ℝ
  nonneg    : ∀ W : Finset X, 0 ≤ w W
  le_one    : ∀ W : Finset X, w W ≤ 1
  is_cover  : ∀ H ∈ ℋ, (1 : ℝ) ≤ ∑ W ∈ H.powerset, w W
  w_empty   : w ∅ = 0

structure IntegralCover (X : Type*) [Fintype X] [DecidableEq X]
    (ℋ : Set (Finset X)) where
  g         : Finset X → Bool
  is_cover  : ∀ H ∈ ℋ, ∃ W : Finset X, W ⊆ H ∧ g W = true

def IsPSmall {X : Type*} [Fintype X] [DecidableEq X]
    (ℋ : Set (Finset X)) (p : ℝ) : Prop :=
  ∃ g : IntegralCover X ℋ,
    (∑ W : Finset X, (if g.g W then (1:ℝ) else 0) * p ^ W.card) ≤ 1/2

theorem main_theorem :
    ∃ c : ℝ, 0 < c ∧
      ∀ {X : Type} [Fintype X] [DecidableEq X]
        (t : ℕ) (p : ℝ) (ℋ : Set (Finset X))
        (cov : FractionalCover X ℋ),
        2 ≤ t →
        0 < p → p ≤ 1 →
        (∀ W : Finset X, t < W.card → cov.w W = 0) →
        (∑ W : Finset X, cov.w W * p ^ W.card) ≤ (1 : ℝ) / 2 →
        IsPSmall ℋ (c * p / Real.log t) := by
```

Figure 4: Lean formalization of the headline theorem of Pham (2025): the `FractionalCover` and `IntegralCover` structures, the `IsPSmall` predicate (the cover cost c_{int}), and the statement of `main_theorem`.

We attempted to formalize the proof of this paper, and the prover proved every supporting lemma, reducing both primary theorems to a single admitted axiom: the paper's averaged Lemma 2.9 bound (See Figure 5a). The resulting development is otherwise `sorry`-free.

Notably, in the course of formalizing the surrounding argument our system surfaced a genuine gap in the paper's proof. The published proof establishes the Lemma 2.9 bound via a surjection-counting argument (its Lemma 2.6 and the surjection Ψ_u); our formalization, checked against the paper's literal definitions, found that this argument does not go through for general $\mathcal{H}$ (we verified explicit counterexamples to Lemma 2.6 as stated). We emphasize that this concerns the *proof*, not the theorem: we do *not* claim the paper's results are false, only that this particular proof of the Lemma 2.9 bound is incomplete as written, and the theorem may well still be correct. Lacking a correct proof, our pipeline admits the bound as one scoped axiom.

In more detail, the defect lies in the proof of Lemma 2.6, the "key property of towers of minimum fragments". The proof shrinks a minimum tower and asserts that the shrunken tower keeps the same feasibility certificate, a step that silently requires the unstated condition $\hat{W}_i \cap T_i \subseteq C_i$, which is false in general. We confirmed this with an explicit counterexample, found and re-derived by independent



(a) Pham (2025) (combinatorics)

(b) Mackenzie & Saffidine (2025) (communication complexity)

Figure 5: Group-level proof-dependency DAGs (box = proof stage, with Lean files · lines · declarations; an edge $A \to B$ means stage A uses stage B, transitively reduced so a direct edge is dropped whenever a longer path already implies it, with mutually-recursive stages shown by a bidirectional edge). **(a)** Pham (2025): `sorry`-free, with every stage proved except a single admitted axiom (orange)—the paper's averaged Lemma 2.9 bound, whose published surjection-counting proof does not go through for general $\mathcal{H}$. **(b)** Mackenzie & Saffidine (2025): `sorry`-free with no axioms beyond Lean's kernel; the two cited external results (Chung's product theorem and the Kushilevitz–Nisan log-rank bound, dark green) are reproved from scratch.

computer enumeration faithful to the paper's literal definitions, already at a single procedure step ($s = 1$): the ground set $X = \{0, \ldots, 12\}$ with $\mathcal{H} = \{\{0, 1, 2\}\} \cup \{\{3\}, \ldots, \{12\}\}$ (one triple and ten disjoint singletons). This family is a legitimate instance of the selector theorem because it is not p-small for $p = 1/16$ (its cheapest integral cover costs $\approx 0.625 > 1/2$), yet its minimum tower violates the inclusion that Lemma 2.6 claims. Because Lemma 2.6 fails, the surjection Ψ_u of §2.3 is not surjective for not-p-small $\mathcal{H}$, and the counting inequality that the proof of the Lemma 2.9 bound relies on is unjustified, so the issue is confined to the proof; a correct proof appears to require an additional argument that the paper does not provide.

This illustrates a general pattern: when a paper implicitly makes assumptions or omits a step, the model cannot close the corresponding lemma and instead either isolates it as a clearly-labeled axiom (if it is a cited, self-contained fact) or, as here, surfaces the gap explicitly. When a complex literature result is properly cited, the model formalizes it as an axiom without proof and proceeds; otherwise it keeps trying to reconstruct the missing content, which makes the formalization substantially harder.

C.3 Autoformalization of Mackenzie & Saffidine (2025) Statement and Proof

The paper refutes the strongest form of the direct-sum conjecture in deterministic communication complexity: it exhibits a family of total Boolean functions for which solving ℓ instances jointly costs strictly less than ℓ times the single-instance cost. We formalized two main theorems together with the supporting structural propositions:

- **Refutation (Theorem 2.9).** There is a function family f such that for all N, L, C there exist $n \geq N$ and $\ell \geq L$ with $D(f_n) > D(f_n^{\oplus \ell})/\ell + C$, where $D(\cdot)$ is deterministic communication complexity and $f_n^{\oplus \ell}$ is the ℓ-fold direct sum. This negates the conjecture by an unbounded additive margin.

- **Multiplicative consequence (Theorem 5.10).** A quantitative bound $D(f^{\oplus 178}) \leq 178\, D_{\mathrm{mat}}(f) + 177$ relating the direct-sum cost to the matrix-form complexity.

Beyond these, we formalized Proposition 2.7 (invariance of complexity under transposition) and Proposition 3.11 (subgames are easier).

A fully axiom-free formalization. This is one of our three developments without added axioms. Our prover pipeline closed *every* theorem and lemma with no `sorry` and no axioms beyond Lean's three standard kernel axioms (`propext`, `Classical.choice`, `Quot.sound`). Crucially, the two external results the paper cites—Chung's product theorem (Chung et al., 1986) and the log-rank lower bound of Kushilevitz–Nisan (Kushilevitz & Nisan, 1997)—were not admitted as axioms but *reproved from scratch* as part of the development, so the formal proof is self-contained down to Mathlib. This demonstrates that, for a self-contained combinatorial argument that is written out in full, our pipeline can deliver a completely verified proof of a research-level theorem. The dependency DAG—entirely green supporting lemmas under the yellow root theorems, with no red `sorry` or axiom nodes anywhere—is shown in Figure 5b.

C.4 Autoformalization of Gravin & Jia (2025) Statement and Proof

This mechanism-design paper studies the coordinate-wise median mechanism for facility location in $\mathbb{R}^d$: agents sit at points $\mathbf{p}_1, \ldots, \mathbf{p}_n \in \mathbb{R}^d$, a single facility $\mathbf{f}$ is placed to serve them, and the social cost is $\mathrm{SC}(\mathbf{P}, \mathbf{f}) = \sum_i \|\mathbf{p}_i - \mathbf{f}\|_q$. Its main result (Theorem 1) bounds how far the coordinate-wise median is from the optimum: under every ℓ_q cost it is within a single dimension-independent factor $\mathrm{UB}(q)$ of optimal, with $\mathrm{UB}(1) = 1$, $\mathrm{UB}(2) = \sqrt{6\sqrt{3} - 8} \approx 1.547$, and $\mathrm{UB}(q) \to 3$ as $q \to \infty$. The paper also proves a matching lower bound (Theorem 2) and learning-augmented consistency and robustness guarantees for a median-with-prediction mechanism (Theorems 3 and 4). We formalized and machine-checked all four.

What we formalized. The statement rests on just three ingredients, and our system introduced one definition for each: the ℓ_q norm $\|x\|_q = (\sum_j |x_j|^q)^{1/q}$, defined uniformly for every real $q \geq 1$ so that the $q = 1$ case and the $q \to \infty$ limit are ordinary real-analysis facts; the social cost $\mathrm{SC}(P, f) = \sum_i \|p_i - f\|_q$ of a placement, together with its optimum $\mathrm{OPT}(P) = \inf_f \mathrm{SC}(P, f)$; and the notion of a coordinate-wise median. We defined the last as a *property* a point may have — that m is *a* coordinate-wise median of P — rather than as one specific chosen median, so the bound is required to hold for every median at once, faithfully matching the paper's arbitrary-but-consistent tie-breaking when n is even.

What the prover built. Our prover pipeline closed all four theorems with no remaining gaps and no axioms beyond Lean's standard kernel, over more than a hundred supporting lemmas. The bulk of the work reconstructs, from first principles, the paper's analytic derivation of the optimal constant $\mathrm{UB}(q)$: that the underlying optimization attains its minimum, the first-order conditions that pin down a local optimum and the sign analysis that selects the relevant case, the reduction of the problem to a single scalar parameter, and the solution of the resulting equations for the closed-form values $\mathrm{UB}(1) = 1$, $\mathrm{UB}(2) = \sqrt{6\sqrt{3} - 8}$, and $\mathrm{UB}(q) \to 3$. The lower bound (Theorem 2) is proved on the paper's explicit worst-case instance, and the consistency and robustness theorems (Theorems 3 and 4) reuse this machinery after augmenting the instance with copies of the prediction. The full dependency graph of the proof is shown in Figure 6.

Faithfulness fixes the system surfaced. Holding the formalization to the paper's literal claims forced two honest adjustments that are themselves findings. First, *monotonicity*: the theorem asserts that UB is non-decreasing, but the paper establishes this only *numerically* — it reads the shape off a plot of $\mathrm{UB}(q)$ and gives no analytic proof — so rather than admit it as an axiom or substitute a derivative argument the paper never makes, we omitted the monotonicity clause from the formal statement. Second, *integrality*: the consistency and robustness bounds are stated for $\lfloor cn \rfloor$ prediction copies but are genuinely *false* for finite n when $\lfloor cn \rfloor < cn$ (the effective weight $\lfloor cn \rfloor / n < c$ yields a strictly worse constant); the paper silently ignores non-integrality by working in the $n \to \infty$ regime, so we recorded the standing assumption $\lfloor cn \rfloor = cn$ as an explicit hypothesis instead of proving a false statement.

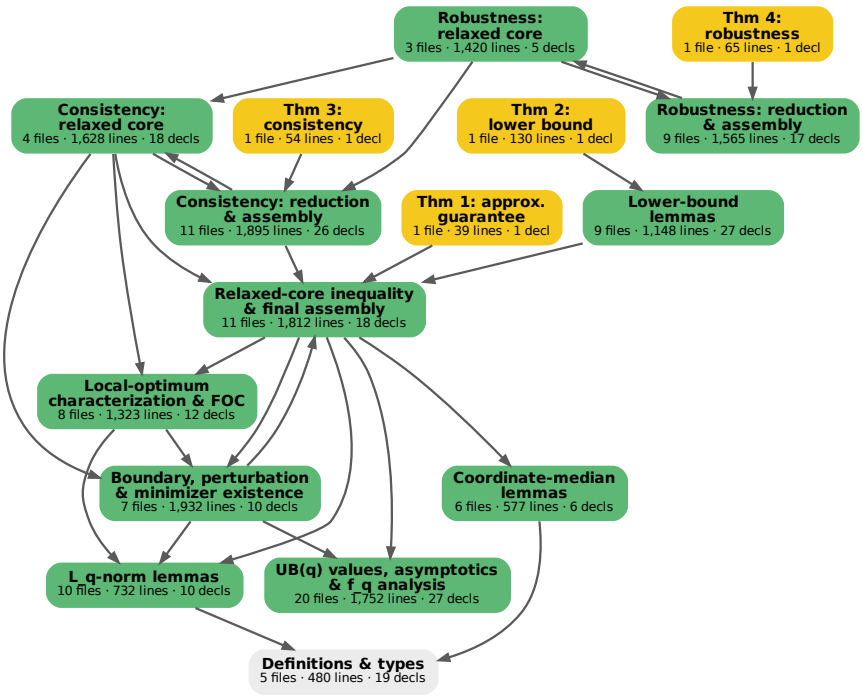


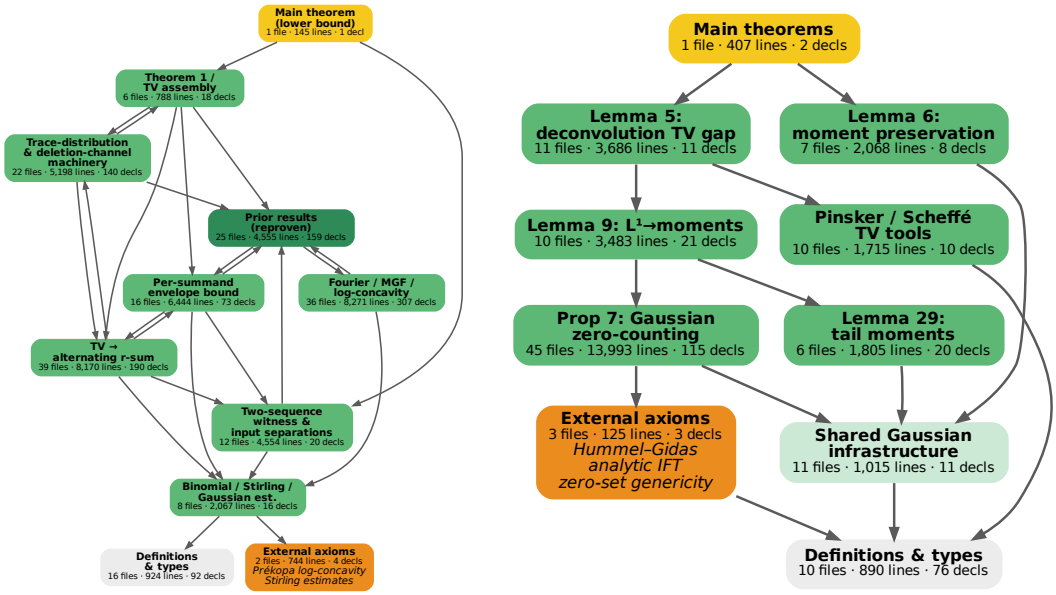
Figure 6: Group-level proof-dependency DAG for the four theorems of Gravin & Jia (2025) (box = proof stage, with Lean files · lines · declarations); an edge $A \to B$ means stage A uses stage B, with edges transitively reduced (a direct edge $A \to B$ is omitted whenever a longer path $A \to \cdots \to B$ already implies the dependency). Stages that split a single argument across two mutually-recursive boxes are shown with a bidirectional edge. The four theorems share the main upper-bound lemmas and are all axiom- and `sorry`-free.

C.5 Autoformalization of Rivkin et al. (2025) Statement and Proof

The paper's main lower-bound theorem is as follows: *There exists a pair of length n sequences $S = p_1, \ldots, p_n$ and $S' = p'_1, \ldots, p'_n$ with constant ℓ_∞ distance and ℓ_1 distance $\Theta(\sqrt{n})$, and an absolute constant c such that for any deletion probability $\delta \geq \frac{c}{\sqrt{n}}$ — and in particular, for all constant deletion probabilities — the distribution of traces drawn from S versus S' have total variation distance $e^{-\Omega(\sqrt{n})}$.*

Our model defined 11 types, 9 of which are directly related to the statement above. First, the model defined *ProbVec* n, a sequence of length n of probabilities. It then defined two types for the ℓ_∞ and ℓ_1 distances of such objects. It then formalized *TraceDist* n S δ, representing the distribution of traces drawn from S with deletion probability δ. It also formalized *TVDistance*, representing total variation distance, and finally formalized *LowerBoundConstants* to formally represent $e^{-\Omega(\sqrt{n})}$.

We ran our prover pipeline on the lower bound and assembled a complete, `sorry`-free proof of it. We formalized only this lower bound; the paper's companion algorithmic upper bound (a reconstruction procedure) is an algorithmic result of the kind covered by our limitations (Appendix A), and we did not formalize it. The proof is not axiom-free, however. The pipeline proves the paper's Fourier-analytic and total-variation lemmas internally (including the alternating-sum total-variation reduction and the Fourier estimates the paper invokes); what it admits, as exactly two clearly-labeled axioms, are two standard analytic facts: Prékopa–Leindler convolution log-concavity (Prékopa, 1973) and a Stirling-type binomial $L^{1/2}$-norm estimate that the paper states as its Fact 9 without proof (Robbins, 1955). This paper thus sits at the more heavily axiomatized end of our spectrum, opposite Mackenzie & Saffidine (2025): the pipeline assembles and proves the high-level argument while deferring these two classical analytic estimates to cited axioms. In the dependency DAG (Figure 7a), green nodes are proven lemmas and red nodes mark the admitted axioms at the analytic core.



(a) Rivkin et al. (2025) (information-theoretic lower bound)

(b) Kalai et al. (2010) (learning theory)

Figure 7: Group-level proof-dependency DAGs (box = proof stage, with Lean files · lines · declarations; an edge $A \to B$ means stage A uses stage B, transitively reduced so a direct edge is dropped whenever a longer path already implies it, with mutually-recursive stages shown by a bidirectional edge). **(a)** Rivkin et al. (2025): `sorry`-free, reducing only to two external-citation axioms (orange), Prékopa log-concavity and Stirling estimates. **(b)** Kalai et al. (2010): acyclic and `sorry`-free, reducing through the Proposition 7 Gaussian zero-counting engine to three external-citation axioms (Hummel–Gidas and two real-analytic implicit-function-theorem facts).

C.6 Autoformalization of Kalai et al. (2010) Statement and Proof

This classic learning-theory paper gives a polynomial-time algorithm for learning a mixture of two Gaussians; its analytic heart is a *polynomial-robustness identifiability* statement, which is the object we formalized. We did *not* formalize the learning algorithm itself or its sample-complexity guarantee—those are algorithmic results of the kind covered by our limitations (Appendix A). Informally the identifiability statement reads: the first six raw moments of a one-dimensional mixture of two Gaussians pin down its parameters, and the dependence is polynomially robust. We formalized two statements:

- **Six moments suffice.** If two non-degenerate two-Gaussian mixtures agree on their first six raw moments, they are identical up to relabeling the two components.
- **Quantitative robustness (Theorem 4).** There is a universal constant $c > 0$ such that, for $0 < \varepsilon < c$, any two one-dimensional two-Gaussian mixtures forming an ε-standard *pair* differ by at least ε^{67} on one of the first six raw moments. As in Definition 1 of Kalai et al. (2010), an ε-standard pair has all four mixing weights at least ε, all component variances at most 1, and all component means bounded in absolute value by $1/\varepsilon$. Within each mixture, the two components satisfy $|\mu_1 - \mu_2| + |\sigma_1^2 - \sigma_2^2| \geq \varepsilon$. Between the two mixtures, the sum of absolute differences in matched weights, means, and variances is at least ε under each of the two possible component matchings. This last condition excludes identical mixtures, including relabelings.

Our system decomposed the argument across more than one hundred Lean files, reconstructing the paper's three-step deconvolution-and-moment argument (Lemmas 5, 6, and 9 and the Proposition 7 zero-counting core). Through a dedicated de-axiomatization effort, the final formalization is `sorry`-free and admits *none* of the paper's own content as an axiom—in particular the hard zero-counting

argument of §6.1, which an earlier draft had assumed, was proved in full. The completed proof depends only on three genuine external citations admitted as axioms: the Hummel–Gidas variation-diminishing theorem for Gaussian convolution (Hummel & Gidas, 1984), the real-analytic implicit function theorem (Krantz & Parks, 2002a), and a genericity property of zeros of real-analytic families (Krantz & Parks, 2002b). This formalization preserves the line between what the paper proves and what it cites. The dependency DAG is shown in Figure 7b.

C.7 Autoformalization of Planar Point Sets with Many Unit Distances

This 2026 manuscript disproves Erdős's unit-distance conjecture by proving that there is an absolute constant $\delta > 0$ and infinitely many n for which an n-point planar set determines at least $n^{1+\delta}$ unit distances. Our `sorry`-free development formalizes the main theorem and its supporting argument in 14,831 lines of Lean code, while admitting only two results from prior work cited by the paper, namely the Golod–Shafarevich filtration input (Golod & Shafarevich, 1964) and Shafarevich's relation-rank estimate (Shafarevich, 1963). We manually checked both statements against their cited sources. The independent Aleph Prover formalization contains roughly 33,000 lines of Lean and relies on the same two external mathematical inputs (The Erdős unit-distance formalization contributors, 2026), making this a matched comparison in which our development is less than half the size.

C.8 Autoformalization of the Cycle Double Cover Conjecture

This recent graph-theory manuscript proves that every finite bridgeless undirected graph has a cycle double cover, meaning a multiset of cycles in which every edge occurs exactly twice. Our pipeline produced a `sorry`-free formalization in 5,713 lines of Lean code, formalizing the manuscript's full argument while tracing its two immediate prior-work dependencies to deeper sources. The final axiom audit contains exactly three citation-backed classical inputs, specifically Fleischner's splitting lemma (Fleischner, 1992), the Nash–Williams–Tutte spanning-tree packing theorem (Nash-Williams, 1961; Tutte, 1961), and Veblen's even-graph cycle decomposition (Veblen, 1912). Apart from these inputs, the development uses only Lean's standard kernel axioms.

C.9 Autoformalization of the Strong Perfect Graph Theorem

Statement and proof structure. For a finite simple graph G, the main theorem of Chudnovsky et al. (2006) states

$$G \text{ is perfect} \quad \Longleftrightarrow \quad G \text{ is Berge.}$$

Perfection requires $\chi(H) = \omega(H)$ for every induced subgraph H; the Berge condition excludes odd holes and odd antiholes of length at least five. The formalization builds the graph-theoretic vocabulary needed for the paper's structural argument, including basic graph classes, proper 2-joins, proper homogeneous pairs, and balanced skew partitions. It proves Theorem 1.3: every Berge graph is basic, admits a proper 2-join in itself or its complement, admits a proper homogeneous pair, or admits a balanced skew partition. Following the published proof, a minimum imperfect counterexample is incompatible with every alternative, yielding Theorem 1.2.

Scope and axiom audit. Our development comprises approximately 350,000 lines in 1,209 modules, using Lean 4.30.0-rc2 and Mathlib at commit `2f9133a`. The root theorem is `Workspace.MainTheorem.SPGT.thm_1_2`. Its axiom audit contains only `propext`, `Classical.choice`, and `Quot.sound`, with no `sorry`. The development also proves the eight external results imported by the paper's argument, including Lovász's perfect graph theorem and replication lemma, the Roussel–Rubio lemma, and the exclusions of certain decompositions in minimum imperfect graphs. It additionally proves the uncited Dirac subdivision theorem used in the proof of Theorem 5.3. Thus this case extends the pipeline's ability to discharge prior-work dependencies to a large structural graph-theory proof. The counterexample and the corrections are given below.

C.9.1 Three intermediate statements requiring correction

Formalization exposed three false intermediate statements in the SPGT proof. We give the failures, an explicit counterexample, the corrected claims proved in Lean, and their downstream uses here.

The theorem numbers below refer to Chudnovsky et al. (2006). All three corrections preserve Theorems 1.2 and 1.3, and none introduces an additional axiom. We write $N_G(v)$ for a vertex's neighbourhood. A vertex is X-complete if it is adjacent to every vertex of X, and an edge is X-complete if both endpoints are. A set is anticonnected if it induces a connected subgraph of the complement; an antipath is an induced path in the complement. Path lengths count edges.

Theorem 9.3: an overly strong conclusion. The theorem concerns a knot (P_1, P_2, Q_1, Q_2) inducing K in a Berge graph G, where the P_i are paths and the Q_i are antipaths. Its hypotheses exclude K_4-enlargement appearances and overshadowed K_4 appearances in both G and its complement, and require a connected set F outside K whose attachments in K are not local. Under these hypotheses, it asserts one of four outcomes. In outcome 4, after a permitted relabelling, a vertex $f \in F$ must satisfy

$$N_G(f) \cap S = N_G(x_1) \cap S \quad \text{and} \quad f y_1 \notin E(G), \qquad S = V(P_1) \cup V(P_2) \cup V(Q_2),$$

where x_1, y_1 are the ends of Q_1. For completeness, the other three outcomes are:

1. Some vertex of F has a neighbour set in K that resolves the knot.
2. Up to symmetry, a path $R \subseteq F$ with ends r_1, r_2 has r_1 matching the neighbours of a_1 on $T = V(P_2) \cup V(Q_1) \cup V(Q_2)$, $R \setminus \{r_1\}$ anticomplete to T, r_2 adjacent to a vertex of $V(P_1) \setminus \{a_1\}$, and $R \setminus \{r_2\}$ anticomplete to that set.
3. Up to symmetry, an odd path $R \subseteq F$ has its ends r_1, r_2 matching the neighbours of a_1, b_1, respectively, on T, its interior anticomplete to T, and no edges to P_1 except possibly $r_1 a_1$ and $r_2 b_1$.

Here two sets are anticomplete if no edge joins them; the knot terminology and permitted relabellings are those of the original SPGT paper. The specified non-adjacency to y_1 need not hold. When Q_1 has length greater than one, the proof only establishes that f misses some vertex of Q_1 other than x_1.

Explicit counterexample. Let H be the bipartite subdivision of K_4 with branch vertices $0, 1, 2, 3$, square 0–1–2–3–0, and diagonal branches 0–4–5–6–2 and 1–7–3. Label its ten edges as follows:

Edge of H	01	12	23	30	04	45	56	62	17	73
Vertex label	b_1	a_2	a_1	b_2	x_1	z_1	z_2	y_1	x_2	y_2

Take $G_0 = \overline{L(H)}$, so two labelled vertices are adjacent exactly when the corresponding edges of H are disjoint. The lists

$$P_1 = [a_1, b_1], \qquad P_2 = [a_2, b_2], \qquad Q_1 = [x_1, z_1, z_2, y_1], \qquad Q_2 = [x_2, y_2]$$

form a knot on $V(K) = V(G_0)$. Add an eleventh vertex f with

$$N_G(f) = \{a_1, a_2, x_1, z_2, y_1, x_2, y_2\} = V(K) \setminus \{b_1, b_2, z_1\}, \qquad F = \{f\}.$$

Figure 8 draws this graph, including every edge.

Checking the hypotheses. The Lean declaration `printed_nine_three_false` checks every hypothesis. The graph G is Berge: neither G nor its complement has an odd induced 2-regular subgraph on at least five vertices. The four lists form a knot, $F = \{f\}$ is connected and outside K, and its attachments are nonlocal. To exclude the forbidden appearances in G, the proof checks that every vertex subset of size at least eight has a vertex of induced degree outside the range 2–4, whereas a K_4 appearance requires all degrees in that range. In the complement, every K_4 appearance is degenerate and has no odd branch of length at least three; the formal checks exclude both overshadowed appearances and K_4-enlargement appearances. The declaration then verifies that all four printed outcomes fail:

- Outcome 1 fails because the attachment set does not resolve the knot: neither endpoint of the edge $b_1 z_1$ belongs to $N_G(f)$.
- Outcomes 2 and 3 require a path in F with an endpoint matching the neighbours of one of a_1, b_1, a_2, b_2 on the opposite side of the knot. The only possible path is $[f]$, and z_1 distinguishes f from all four endpoints.

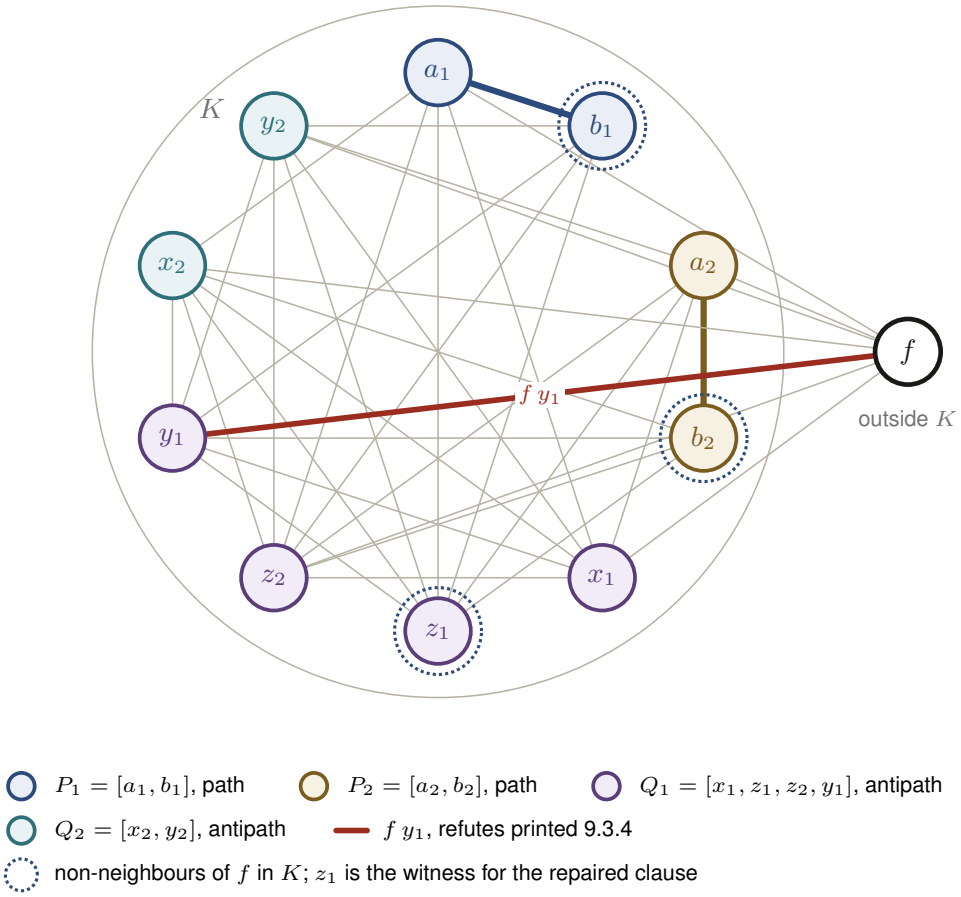


Figure 8: The eleven-vertex counterexample to Theorem 9.3. The ten vertices of the knot K are grouped by their paths P_1, P_2 and antipaths Q_1, Q_2; f is outside K. All 36 edges are shown. The red edge fy_1 violates the printed non-adjacency clause, while the absent edge fz_1 supplies the witness for the repaired clause. Dotted rings mark the three non-neighbours of f in K.

- For outcome 4 with $(x, y) = (x_1, y_1)$, the neighbourhood equality holds, but $fy_1 \in E(G)$. The other three choices (y_1, x_1), (x_2, y_2), and (y_2, x_2) fail the neighbourhood equality at a_1, b_2, and b_1, respectively.

Correction and downstream use. Keep the neighbourhood equality and replace only the final clause of outcome 4 by

$$\exists w \in V(Q_1) \setminus \{x_1\} \quad fw \notin E(G).$$

The printed clause implies this correction by taking $w = y_1$; when Q_1 has length one, the clauses coincide. The counterexample satisfies the corrected clause with $w = z_1$, as checked by `repaired_conclusion11` in the same module. The two consumers, Theorems 9.4 and 9.5, do not use this final conjunct, so the weaker conclusion suffices. The counterexample checks and the proof of the corrected theorem use only `propext`, `Classical.choice`, and `Quot.sound`.

Theorem 18.3: a missing length restriction. Under the theorem's hypotheses, $G \in \mathcal{F}_7$, and X, Y are disjoint nonempty anticonnected sets complete to each other. The path $P = p_1 \cdots p_n$ avoids $X \cup Y$, has $n \geq 5$, and has exactly its two endpoints X-complete. Assuming at least two vertices of P are Y-complete, the second conclusion concerns a maximal subpath P' with no Y-complete internal vertex. As printed, it asserts

$$\ell(P') \equiv \left|\{u \in \mathrm{ends}(P') \cap \{p_1, p_n\} : u \text{ is not } Y\text{-complete}\}\right| \pmod 2.$$

This fails when P' is a single Y-complete edge. Such a subpath has no internal vertex and is maximal: extending it makes a Y-complete endpoint internal. Its length is one, whereas the counted set is empty, giving the false parity equality $1 \equiv 0 \pmod 2$.

Correction and downstream use. Require $\ell(P') \geq 2$ for this conclusion. The proof already treats subpaths under this restriction and separately allows length-one subpaths, so the correction restores the scope of its argument. The first conclusion, that P has even length, and the final parity statement about the number of Y-complete edges retain their original hypotheses. The only consumer of the corrected subpath clause, Theorem 18.5, applies it to subpaths of length at least three.

Theorem 2.5: a missing anticonnectivity hypothesis. The printed statement takes a Berge graph G, a set $X \subseteq V(G)$, and an odd induced path $P = p_1 \cdots p_n$ outside X. Both ends of P are X-complete, no edge of P is X-complete, and a vertex v is X-complete. It then claims

$$vp_1 \in E(G) \quad \text{or} \quad vp_2 \in E(G) \quad \text{or} \quad N_G(v) \cap V(P^*) = \{p_{n-1}\},$$

where P^* is the interior of P. The statement omits anticonnectivity of X, although its proof invokes Theorems 2.2 and 2.1, both of which require it. For an explicit counterexample without this assumption, take vertices $\{x_1, x_2, p_1, p_2, p_3, p_4, v\}$ and exactly the edges

$$E(G) = \{x_1x_2, x_1p_1, x_1p_2, x_1p_4, x_1v,\ x_2p_1, x_2p_3, x_2p_4, x_2v,\\ p_1p_2, p_2p_3, p_3p_4\}.$$

Its only holes are the four-cycles on $\{x_1, x_2, p_2, p_3\}$, $\{x_1, p_2, p_3, p_4\}$, and $\{x_2, p_1, p_2, p_3\}$; its complement has no hole, so G is Berge. With $X = \{x_1, x_2\}$ and $P = p_1p_2p_3p_4$, all the printed hypotheses hold. However, X is not anticonnected, and v has no neighbour on P, refuting every alternative of the conclusion.

Correction and downstream use. Add the hypothesis that X is anticonnected, leaving the conclusion unchanged. This supplies the missing premises of the two invoked results. The sole application, in Theorem 2.8, already has X anticonnected, so its proof continues to apply the corrected statement.

C.10 Autoformalization of Graph Isomorphism in Quasipolynomial Time

Scope and assumptions. We follow version 2.5 of Babai (2018), dated November 2, 2018. The formalization proves String Isomorphism in quasipolynomial time (Theorem 1.1.1), including the search form that returns automorphism generators and an isomorphism representative, and derives Graph Isomorphism and Coset Intersection (Corollary 1.1.2). The machine theorem measures running time in the encoded input length. The declarations `Algo.theorem_1_1_1` and `Algo.corollary_1_1_2` appear in `Workspace/Algo/TopMasterClose.lean`. The accompanying `audits/faithfulness_C.md` records five dependencies beyond Lean's standard axioms: the Cameron–Maróti theorem on large primitive groups (Cameron, 1981; Maróti, 2002), polynomial-time computation of Cameron-group structure (Babai et al., 1987), the degree-of-transitivity bound for non-giant groups (Curtis et al., 1976), a bound on alternating groups involved in general linear groups (Babai et al., 2009, Prop. 1.22), and Schreier's hypothesis (Dixon & Mortimer, 1996, App. A). The development proves the intervening source repairs; they introduce no additional assumed gap.

Three main gaps. We highlight three gaps affecting the mathematical or algorithmic argument and the repairs used in the formalization. The source is version 2.5, which explicitly describes itself as an incomplete revision. Entry numbers below refer to the accompanying `GraphIsomorphism/DEVIATIONS.md`.

Omitted exceptional groups (Theorem 10.4.2; entry 14). The printed Jordan–Liebeck statement asserts that if $A(\Omega) \leq K \leq S(\Omega)$, $H \leq K$, $n = |\Omega| \geq 9$, $1 \leq r < n/2$, and $[K : H] < \binom{n}{r}$, then some $T \subset \Omega$ with $|T| < r$ satisfies

$$A(\Omega)_{(T)} \leq H \leq S(\Omega)_T,$$

where the subscripts denote pointwise and setwise stabilizers, respectively. Take $n = 10$, $r = 4$, $K = S_{10}$, and $H = S_5 \wr S_2$, the stabilizer of a partition into two sets of size five. Then

$$[K : H] = \frac{10!}{2(5!)^2} = 126 < 210 = \binom{10}{4}.$$

Yet H is transitive, so an H-invariant proper subset T must be empty. The conclusion would then require $A_{10} \leq H$, impossible because $|H| = 28{,}800 < 1{,}814{,}400 = |A_{10}|$. The formalization restores the omitted exceptional alternatives from Dixon–Mortimer, proves the corrected theorem, and handles the affected Johnson-scheme argument separately. The arithmetic and the transitivity obstruction are checked by `jordanLiebeck_counterexample_arith` and the stabilizer lemma in `Workspace/AltQuotients/SmallIndex.lean`.

Transitivity lost during reduction (Section 11.3; entry 72). Procedure Reduce-to-Johnson promises a transitive output group, but the subgroup obtained by Luks descent need not remain transitive on the original domain. For a concrete construction, let $\bar{G} = S_k \wr S_2$ act on $[k]^2$, let $\varepsilon : \bar{G} \to \mathbb{Z}_2$ record whether coordinates are exchanged, and let $G = \{(g, \varepsilon(g)) : g \in \bar{G}\}$ act on $\Omega = [k]^2 \times \mathbb{Z}_2$ by

$$(p, b) \longmapsto (gp, b + \varepsilon(g)).$$

This action is transitive and induces $\bar{G}$ on the two-element blocks. At $k = 16$ it meets the procedure's large-group threshold. The admissible normal subgroup returned by descent lies in the base group $S_k \times S_k$, so its preimage preserves the two layers $[k]^2 \times \{b\}$ and is intransitive. The repair tests transitivity again and, when it fails, applies the existing Chain Rule to the orbits. Normality makes the orbits equally sized, each at most half the original domain, so recursive progress is preserved. This branch is implemented in `Workspace/Master/Line1.lean` and justified in `Workspace/Master/Descent.lean`.

The progress bound at $t = 1$ (Sections 14–15; entry 96). The argument for Eq. (73) requires a recursive window to shrink to at most $2n/3$ or the Johnson parameter to halve. An allowed colored $4/5$-partition can instead have a dominant color class that is a single block, carrying neither of the structures used to justify halving. For $t \geq 2$, its lifted relative size is at most $(4/5)^t \leq 16/25 < 2/3$. At $t = 1$, it can be $4/5$, so neither claimed form of progress follows. The imprimitive action $\mathbb{Z}_2 \wr S_m$ on $[m] \times \{0, 1\}$ shows that the algorithm can reach the relevant branch with $t = 1$; the gap concerns what its allowed partition outcome guarantees. The formalization changes the window threshold to $4n/5$ and reproves the progress and recurrence bounds. Since the shrinkage factor remains a fixed constant below one, the final running time remains quasipolynomial.

D Pipeline Implementation Details

D.1 Shared Tools

Throughout the pipeline, agents have access to a tool that looks up concepts within the Lean documentation (Lean Search), a tool for retrieving the full documentation for a given Lean or Mathlib syntax (Lean Lookup), and a Lean REPL (Read-Eval-Print Loop) that compiles Lean code snippets and allows agents to build incrementally on previously established definitions. Lean REPL reports proof goals, open assumptions, error messages, and compilation status.

D.2 Proof Formalization Details

The proving pipeline is shown in the bottom half of Figure 1. Following the same agentic paradigm as our statement formalization and building on Liu et al. (2026), our prover treats a proof as a recursive tree of *proof nodes*, where each node is a single statement to prove and the main theorem is the root. Every node is processed by the same chain. The *Natural Language Prover* first proposes an informal proof, drawing on the paper's own argument when one exists. It then enters an internal revision loop with the *Proof Critic*, revising until the proof reads cleanly or a round cap is reached. Keeping this loop inside the Natural Language Prover hides the intermediate critic rounds from the orchestrator, sparing its context for high-level strategy. The accepted informal proof is then expanded by the *Proof Detailer* into tactic-sized steps and named sub-lemmas. The *Lemma Breakdown* agent decomposes these into a topologically ordered set of lemmas with stable identifiers, and the *Lemma Leanifier* translates each into a Lean statement whose body is left as `sorry`.

The *Proof Critic* audits a single informal proof for soundness, and is the proving-stage counterpart of the Faithfulness Judge. The two are not the same agent: the Faithfulness Judge (Section 2.1.3) checks whether Lean code matches an informal *statement*, whereas the Proof Critic checks whether

an informal *proof* is itself correct, so it works purely in natural language and never inspects Lean. Rather than returning a pass-or-fail verdict, the critic reads the argument top to bottom and emits a list of concrete questions: inferences that do not follow without an unstated hypothesis, terms used before they are defined, hidden case splits, swapped quantifier orders, and appeals to a "standard argument" or to a step that silently proves something weaker or stronger than the stated goal. The Natural Language Prover answers each question against the source paper and rewrites the affected passages, and the loop repeats until the critic raises no further questions. This keeps the eventual decomposition and Lean proof from inheriting gaps that were already latent in the informal argument.

Before proving a node, the orchestrator decides whether to recurse on it at all. It scores the node on signals such as its depth in the tree, the length of its Lean statement, the nesting of its quantifiers, and cues in the source text (for example, a step the paper calls immediate, versus one to which it devotes a full section), and combines them into a single leaf-or-recurse judgement. A low-complexity node is treated as a leaf and its goal is closed directly; a higher-complexity node is broken down into the sub-lemmas above. In the latter case we always prove the node *before* its children: the orchestrator first proves the node using each child lemma's statement as an unproven `sorry` premise, and only afterwards recurses to prove those children. The reason is that the parent proof is what reveals whether a child lemma's statement is actually usable. A lemma whose statement looks reasonable may turn out to be too weak, too strong, or wrong for the goal it is meant to support, and proving it first would sink effort that the parent-level attempt would have shown to be misdirected. Every node, the root theorem and each lemma alike, is closed by the same *Prover*, using its children's statements as premises.

A proof attempt is accepted only if it leaves the node's statement untouched. As in statement formalization, the *Claim Check* tool admits an attempt exactly when the node's statement is preserved byte for byte, while still letting the attempt add whatever imports, `opens`, or helper lemmas the proof needs. It also rejects any import that would create a cycle against the node's ancestors, and on success it promotes the attempt into the workspace. This ensures a proof cannot silently weaken the statement it closes, since that statement is fixed in advance.

When an attempt fails, the prover returns a structured diagnosis rather than a bare failure: that a needed hypothesis is missing, that an existing child lemma is too weak or contradictory, or that the goal is stuck despite valid tactics. The orchestrator routes each case differently, re-running the breakdown to introduce a missing lemma, reformalizing a conflicting child, or, when a node first thought to be a leaf keeps failing, promoting it to a non-leaf and re-entering it into the full pipeline. This structured-failure feedback is what gives the framework dynamic backtracking rather than a fixed pipeline, and retry and escalation budgets together with a maximum recursion depth keep the process terminating.

Finally, the breakdown may tag a lemma as a result cited from prior published work rather than something to prove. Such a lemma is admitted as a Lean `axiom`, and the Faithfulness Judge then verifies that the axiom's statement faithfully matches the cited result. This is the only way an axiom beyond those of Lean and Mathlib enters a development. For benchmark evaluation this path is disabled, so proofs of PutnamBench problems stay self-contained and depend only on the axioms of Lean and Mathlib. Throughout the pipeline the orchestrator authors no proof content of its own: every line of Lean and natural-language proof comes from a subagent, which keeps the derivation auditable.

E User Prompts

For completeness we record the natural-language prompts we used to drive the orchestrator on the research papers; these are distinct from the PutnamBench invocation given in Appendix C. We reproduce the prompt that starts statement formalization, the prompt that starts the proof, and the prompt that advances the prover into its second, de-axiomatization phase.

Start of statement formalization

```
I want you to formalize this paper's main theorems (all of them), and
also formalize any concept, lemma, fact, or prior-work lemma/theorem
that is needed in the proofs of the main theorems.  So you have to
read the theorems and also the proofs carefully.  This is very
important.  Do not start before you know the theorems and the proofs.
I want the formalizations to be 100% faithful to the paper, so no
deviation.  I want you to work for 23 hours on this task, so record
the time now in your memory and work until then.  Do not stop until
then.  Do not ask questions until then.
```

Start of proof

```
I want you to prove the main theorem fully.  It is important that we
have all the paper's proofs in Lean.  We can keep prior work as a
sorryAx or axiom (things whose proof does not exist in the paper), but
theorems/lemmas whose proofs exist should be formalized.  Also, I
don't want a different approach to the proofs compared to the paper; I
want exactly the proof that exists in the paper.  A correct proof that
differs from the paper's approach is useless, and I will delete it
later if I catch it.  So whenever you find a proof that takes a
different approach, or differs in any way, correct it the moment you
find it.  Tell the NLP provers that they are only allowed to read the
proof from the paper and fill in the gaps / unwritten parts; they are
not allowed to invent their own proofs.  Also, I'm going to leave, and
I don't want you to stop working on the formalization.  If things are
hard, just break them down and do a small part until I come back.  I
will come back in 20 hours.  So read the proof and force the agents to
follow the paper.  You have the .tex files, which are easier for
agents to read; try to use them.
```

Start of proving phase 2 (de-axiomatization)

```
After you have fully translated all the paper proofs related to the
main theorem (only after you have done this fully), and you are left
with several axioms based on prior work (not the things the paper
itself proves), you should go to phase 2.  Before going to phase 2,
you should finish phase 1 fully; do not rush it.  After phase 1 is
done, everything should be faithful to the paper, and only if some
paper proof is wrong are you allowed to write a document explaining
the bug and try to prove it in another way.  In phase 2, you should
try to prove the prior-work axioms, pushing them further back to the
prior works of those papers, and further back iteratively, until you
are left with only Mathlib axioms.  In this phase you should not
change the proofs from phase 1, unless you find that the formalization
of one of the axioms is wrong and you need to correct it.  I'm going
to leave for another 10 hours, so do not stop until you have fully
proven the main theorem with only Mathlib axioms.
```

F Extended Related Works, Lean 4 Autoformalization Systems

The past two years have seen a rapid proliferation of systems targeting automated Lean 4 formalization. We organize them by their primary input/output contract and approach. Table 5 gives a structured overview; the subsections below discuss each cluster in turn. Here we focus on Lean 4 specifically and on systems that interact with the Lean compiler during inference. We include both automated theorem proving (ATP) systems, which take a formal statement as input, and autoformalization systems, which begin from natural language.

Table 5: Overview of recent Lean 4 formalization systems. **Input**: NL = natural language statement or document; FS = formal Lean 4 statement; NLP = natural language proof. **Output**: FP = formal Lean 4 proof; FS = formal Lean 4 statement; FS+FP = both.

System	Input	Output	Key idea
TheoremLlama (Wang et al., 2024)	FS+NLP	FP	NL-FL aligned data; curriculum training
LeanAgent (Kumarappan et al., 2025)	FS	FP	Lifelong retriever training across repos
Seed-Prover (Chen et al., 2025b)	FS	FP	Experience-based training at scale
DeepSeek-Prover-V1.5 (Xin et al., 2024b)	FS	FP	MCTS with intrinsic rewards
Goedel (Lin et al., 2025a)	FS	FP	SFT on large translated synthetic dataset
LeanaBell (Ji et al., 2025)	FS	FP	Multi-turn verifier interactions
Zhang et al. (Zhang et al., 2025)	FS	FP	Posttraining scaling; GRPO RL
Kimina-Prover (Wang et al., 2025)	FS	FP	RL-driven reasoning exploration
DeepSeek-Prover-V2 (Ren et al., 2025)	FS	FP	Recursive subgoal decomposition
Goedel-V2 (Lin et al., 2025b)	FS	FP	Verifier-guided self-correction
HILBERT (Varambally et al., 2025)	FS	FP	Subgoal decomposition; informal + formal LLMs
Aristotle (Achim et al., 2025)	FS	FP	Monte Carlo search; informal lemma generator
APOLLO (Ospanov et al., 2025)	FS	FP	Sorrifier isolates sub-goals; recursive LLM repair
Ax-Prover (Breen et al., 2025)	FS	FP	Multi-agent; incremental have-step proving
Delta Prover (Zhou et al., 2025)	FS+NL	FP	Decomposition and iterative reflection
Minimal Agent (Pozo et al., 2026)	FS	FP	Memory manager and Lean compiler feedback
APRIL (Wang et al., 2026b)	FP (broken)	FP	Supervised proof repair from compiler feedback
Process-Driven (PDA) (Lu et al., 2024)	NL	FS	Compiler feedback as process-level training signal
DeepSeek-Prover (Xin et al., 2024a)	NL	FS+FP	Large-scale data via NL to formal translation
ATF (Guo et al., 2025)	NL	FS	LLM-ensemble semantic consistency check
CriticLean (Peng et al., 2025)	NL	FS	RL-trained critic evaluates semantic fidelity
ProofBridge (Jana et al., 2025)	NL+NLP	FS+FP	Joint Embeddings for Retrieval: NL $\rightleftharpoons$ FS
MerLean (Ren et al., 2026)	NL doc	FS+FP	LaTeX→Lean→NL; quantum computing domain
M2F (Wang et al., 2026d)	NL doc	FS+FP	Statement compilation + proof Repair and splitting
Textbook Formalizer (Gloeckle et al., 2026)	NL doc	FS+FP	Multi-agent Software engineer scaffold
Numina-Lean-Agent (Liu et al., 2026)	FS/NL Doc	FP	Claude Code + MCP toolset
Agentic Proof Automation (Xu & Odersky, 2026)	FS	FP	Claude Code + lean4check; 14k-line case study
Seed-Prover-V1.5 (Chen et al., 2025a)	NL	FS+FP	Natural language sketch to formal decomposition
Ours	FS/NL Doc	FS+FP	Type-first; aux-lemma verification.

F.1 Automated Theorem Proving: Fine-Tuned Models

Early successes in Lean 4 ATP came from fine-tuning language models on aligned datasets. **DeepSeek-Prover** (Xin et al., 2024a) catalyzed this by generating a massive synthetic dataset of 8 million formal statements and proofs by translating high-school and undergraduate math problems into Lean 4.**Goedel** (Lin et al., 2025a) expanded on this paradigm with a self-improving bootstrap mechanism, creating the 1.64 million statement Goedel-Pset-v1 dataset and fine-tuning on recursively solved proofs. **TheoremLlama** (Wang et al., 2024) constructs the Open Bootstrapped Theorems (OBT) dataset by deformalizing $\sim$100k Mathlib theorems into natural language using Gemini, then fine-tunes Llama 3-8B with block training (attending over both NL and Lean 4 tokens in a single sequence) and curriculum sorting by proof complexity. The key insight is that providing the natural-language proof alongside the formal statement substantially improves performance, reaching 36.5% on MiniF2F-Valid. **LeanAgent** (Kumarappan et al., 2025) extends this with a lifelong learning loop: a retrieval model is progressively trained on theorems sorted by a curriculum, with one epoch per repository to mitigate catastrophic forgetting. At each proof attempt the retrieved premises are appended to the current Lean goal state, and tactic candidates are generated via beam search. **Seed-Prover** (Chen et al., 2025b) scales experience-based training, collecting verified proofs discovered during search and distilling them back into the model, reaching strong results on undergraduate-level benchmarks including PutnamBench (50.4%). **APRIL** (Wang et al., 2026b) takes a different angle: rather than proving from scratch, it trains a model to *repair* broken proofs using compiler diagnostic messages paired with human-written natural-language explanations, building a dataset of 22k repair pairs. Post-training strategies have further refined these capabilities; **Leanabell-Prover** (Zhang et al., 2025) continually trains provers to emulate cognitive behaviors like hypothesis refinement,

subsequently applying GRPO-based reinforcement learning using Lean compiler feedback. While these methods are impressive, they can only solve significantly simpler questions compared to ours.

F.2 Automated Theorem Proving: Search and Agentic Methods

As models improved, inference-time search and agentic systems became the dominant paradigm. Explicit search algorithms gained traction with **DeepSeek-Prover-V1.5** (Xin et al., 2024b), which introduced RMaxTS (a variant of Monte-Carlo Tree Search) guided by intrinsic rewards and intermediate tactic states. Moving away from explicit search trees, **Kimina-Prover** (Wang et al., 2025) utilizes a formal reasoning pattern driven by large-scale reinforcement learning to implicitly flatten search into a single context window. Decomposition has become equally critical. **DeepSeek-Prover-V2** (Ren et al., 2025) employs a 671B Mixture-of-Experts architecture to decompose complex theorems into explicit `have` subgoals, unifying informal reasoning and formal verification through consistency rewards.

Self-correction is a core mechanism in recent models. **Goedel-V2** (Lin et al., 2025b) leverages verifier-guided self-correction, feeding exact compiler error traces back into its context to perform deductive debugging, achieving state-of-the-art efficiency. Similarly, **LeanaBell** (Ji et al., 2025) focuses on multi-turn verifier interactions and feedback token masking to stabilize RL optimization based purely on outcome rewards. Multi-agent orchestration pushes these capabilities to graduate-level mathematics. **Seed-Prover-V1.5** (Chen et al., 2025a) dissects formalization into a triad of specialized agents (Natural Language Prover, Sketch Model, and Agentic Lean Prover) that collaboratively decompose informal sketches into verifiable lemmas.

HILBERT (Varambally et al., 2025) is a publicly available system that pairs a strong general-purpose reasoning model (e.g., Gemini 2.5 Pro) with a specialized prover LLM (e.g., Goedel-Prover-V2-32B) in a recursive decomposition loop. When the prover fails, the reasoner generates an informal proof, extracts `have` subgoals, and recursively delegates each to the prover. HILBERT achieves 99.2% on MiniF2F and 70% on PutnamBench. **Aristotle** (Achim et al., 2025) (a proprietary system) combines Monte Carlo graph search over Lean proof states with an informal reasoning pipeline that generates and formalizes auxiliary lemmas, plus a dedicated geometry engine. It achieved gold-medal-equivalent performance on IMO 2025. **APOLLO** (Ospanov et al., 2025) introduces a *sorrifier* that parses a failing proof into a syntax tree, replaces each failing block with `sorry`, and recursively re-prompts the LLM on the isolated sub-goal; an automated solver (nlinarith, ring, simp, norm_num) handles trivial residuals. **Ax-Prover** (Breen et al., 2025) uses the Model Context Protocol to expose Lean's language server to an orchestrating LLM, which advances proofs one `have` step at a time using diagnostic messages. **Delta Prover** (Zhou et al., 2025) alternates between decomposing the goal into sub-lemmas and using iterative reflection to repair failed sub-proofs. **Minimal Agent** (Pozo et al., 2026) is notable as a rigorous ablation study: it isolates the contributions of memory (history vs. self-managed notes), search (library search, web search), and context management to performance on PutnamBench, FATE, and LeanCat. **Numina-Lean-Agent** (Liu et al., 2026) wraps Claude Code in a modular MCP toolset (Lean-LSP-MCP for compilation and semantic awareness, LeanDex for natural-language library search, an informal prover module, a discussion-partner network, and a blueprint DAG planner). The blueprint planner decomposes long-horizon tasks into verifiable subgoals and revises them based on compiler feedback.

F.3 Autoformalization: Statement and End-to-End Systems

Autoformalization is harder to evaluate because compiler success is only a necessary, not sufficient, condition for correctness. **Process-Driven Autoformalization (PDA)** (Lu et al., 2024) introduces the FormL4 benchmark and a Process-Supervised Verifier (PSV) model that uses Lean 4 compiler feedback at the level of individual formalization steps as a training signal, improving accuracy with less data than standard SFT. **ATF** (Guo et al., 2025) takes a two-tool approach: a syntax checker batches candidate statements for efficient compilation, and a semantic consistency checker uses an LLM ensemble (requiring unanimous agreement to confirm correctness) to detect meaning-altering discrepancies. **CriticLean** (Peng et al., 2025) trains *CriticLeanGPT* via RL on a 48k-sample dataset, CriticLeanInstruct, to classify whether a generated Lean 4 statement is semantically faithful to its natural-language source; a loop in which the critic rejects and the formalizer regenerates boosts single-pass accuracy from 38% to 84% on Omni-MATH. **ProofBridge** (Jana et al., 2025) maps

informal proofs and formal Lean proofs into a shared embedding space trained with contrastive objectives, then uses the retrieved formal examples as context for iterative proof repair.

At document scale, two recent systems target the end-to-end formalization of long texts. **M2F** (Wang et al., 2026d) normalizes a LaTeX document into an ordered sequence of atomic blocks, generates Lean declaration skeletons, and uses a file-level verifier oracle to accept only patches that strictly reduce compilation errors or proof holes. Statement correctness is evaluated by human domain experts via provenance-linked side-by-side review. **Textbook Formalizer** (Gloeckle et al., 2026) treats formalization as software engineering: a git-based multi-agent system with trunk development, code review, and a file-system issue tracker. Specialized agents (sketcher, prover, semantic reviewer, engineering reviewer, maintainer, triage, scanner, progress tracker) are deployed concurrently on isolated worktrees, requiring dual-reviewer approval before merging. Applied to a 500-page algebraic combinatorics textbook with $\sim$340 target theorems, the system deploys over 20k agent instances and cost $100,000 to execute. **MerLean** (Ren et al., 2026) targets quantum computing, adding a reverse-translation step that converts verified Lean 4 code back into natural language to enable human auditing of the formalized content.

F.4 Agentic Coding as the Shared Foundation

The most important conceptual thread unifying recent progress is the shift from model-centric to agent-centric design. All systems listed above involve iterative compiler feedback, multi-step orchestration, or tool use beyond simple next-token prediction. Several—including Numina-Lean-Agent and Agentic Proof Automation—use off-the-shelf coding frameworks (Claude Code) without any fine-tuning, relying entirely on the model's programming ability. **Agentic Proof Automation** (Xu & Odersky, 2026) makes this case most sharply: they show that Claude Code with a single `lean4check` tool achieves 87% success on 189 proof engineering tasks. Their analysis distinguishes human task types (proof, repair, refactor, state-and-prove) and shows agents excel at mechanical proof development while still requiring human creativity for non-trivial strategy choices.

Our system builds on the same coding-agent foundation but addresses the earlier, harder problem of statement formalization from raw mathematical documents. The main differences are: (i) we begin from a PDF rather than a pre-formalized statement; (ii) we introduce a type-first decomposition strategy and auxiliary-lemma correctness proxies; and (iii) we evaluate on research-level theoretical computer science and discrete geometry, where formal statements are not directly available.

F.5 Lean for Other Fields

Researchers in fields beyond mathematics are beginning to use Lean as a formalization engine. (Chojecki, 2026) suggests that Lean can be applied to physics, chemistry, biology, and economics, while (Bobbin et al., 2024) uses Lean to formalize aspects of chemical physics. Additionally, (Miranda et al., 2025) applies Lean to code verification. This growing adoption further increases the need for automated approaches to formalizing foundational concepts, enabling subsequent research to be formalized more seamlessly.

F.6 Concurrent Work on Research Autoformalization

Concurrent work investigates several complementary approaches to research-level autoformalization: maintaining formalization projects over long runs, revising proof plans, checking semantic faithfulness, and evaluating systems on mathematics with substantial dependencies. We discuss these approaches by their technical relationship to Theo. The comparisons distinguish formalizing a supplied statement, constructing the definitions needed to state a result, and preserving a source argument.

F.6.1 Long-Horizon Formalization of Research Papers

Trellis. Pegden (2026a) introduces a workflow in which general-purpose agents iteratively elaborate informal proofs under a deterministic supervisor. Its process semantics constrain the allowed transitions and checks during formalization. Like Theo, Trellis treats a long proof as work that must be decomposed and checked incrementally. The architectural distinction concerns how progress

and acceptance are controlled: Trellis emphasizes supervisor-enforced process semantics, whereas Theo organizes definition construction, auxiliary-lemma testing, semantic review, and recursive proving through specialized agents. The Trellis project also reports a formalization of both the Strong Perfect Graph Theorem and its structural decomposition theorem using only standard axioms (Pegden, 2026b). This overlapping target makes theorem coverage, the interpretation of definitions, and the treatment of source repairs meaningful comparison points.

LeanMarathon. Zhang et al. (2026b) uses an evolving Lean blueprint as both a proof skeleton and a shared record of the informal argument. Construction, auditing, proving, and repair agents operate under separate contracts, with target-fidelity review followed by parallel proof rounds over the dependency graph. The authors report formalizations of seven target theorems across two research papers, with 258 lemmas and theorems and no remaining `sorry`. Its revisable blueprint shares Theo's concern with recovering from failed intermediate steps. Theo's emphasis is on evaluating candidate definitions through generated auxiliary properties before relying on them in the main development.

FormalFlow. Lu et al. (2026b) coordinates proving agents under human supervision through a shared blueprint and nested planning, proving, and review loops. Its case study formalizes the quantum soundness of the classical low individual-degree test, a core theorem underlying $\mathrm{MIP}^* = \mathrm{RE}$. The reported development corrects intermediate errors and side conditions while preserving the final error bound under corrected assumptions. This is closely related to Theo's use of formalization to expose and repair gaps in research proofs.

LeanFlow. Milikic et al. (2026) studies a workflow for translating mathematical papers into buildable Lean projects, separating source-backed statement preparation from queued proof work and repair. Its case studies vary the model, workflow, and toolset, showing that the effect of queue-based organization depends on the backing model. LeanFlow and Theo both address document-level formalization and statement fidelity. Their comparison concerns the mechanisms used to prepare and validate statements, organize unresolved obligations, and recover from failure, rather than whether either system uses compiler feedback or multiple stages.

F.6.2 Blueprints, Proof Search, and Collaboration

LEAP and Goedel-Architect. LEAP (Kung et al., 2026) connects informal reasoning and formal proof construction through decomposition and repeated interaction with Lean. Goedel-Architect (Chung et al., 2026) generates a graph of definitions and lemma statements, proves open nodes in parallel, and refines the blueprint when obligations fail. These systems share important elements with Theo's proof pipeline, including intermediate statements and feedback from unsuccessful proof attempts. Theo additionally foregrounds the construction and testing of mathematical vocabulary needed by a source document.

OpenProver and Prove2Me. OpenProver (Kripner & Straka, 2026) combines a planner, workers, and a verifier with persistent notes and a repository of intermediate findings; its interactive interface also permits human steering. Prove2Me (Chen et al., 2026) provides a collaborative platform in which humans and agents contribute to shared formalization missions and reuse one another's results. These approaches address coordination and continuity of mathematical work. Theo instead specifies a particular sequence of definition checks, statement review, and proof decomposition within a formalization run. Persistent project state and collaborative infrastructure are complementary to those semantic checks.

Broader research orchestration. Stellar Colosseum (Lin et al., 2026) explores alternative research strategies, decomposes sufficiently developed plans, and routes verification feedback to the relevant parts of an argument. Its focus on long mathematical and theoretical-computer-science arguments overlaps with Theo's orchestration problem, although its research outputs and evaluation protocols are not uniformly Lean proof artifacts.

F.6.3 Testing Definitions and Choosing Formal Representations

Testing through mathematical consequences. The T^2 framework of Kim et al. (2026) evaluates a generated declaration by checking whether its dependent successor theorems still compile. It thereby tests compatibility with existing consumers of the declaration. Theo's auxiliary lemmas address a related question when introducing new definitions: the system generates source-motivated properties and attempts to prove them before the main theorem's downstream proof is available. The distinction is between testing against an existing dependency context and constructing diagnostic properties during formalization.

Representation search and library grounding. FormalEvolve (Lu et al., 2026a) searches over alternative formal representations to improve subsequent proving. MathForm (Pu et al., 2026) retrieves relevant Mathlib knowledge and refines generated statements using compiler and semantic-consistency feedback. These approaches separate finding a usable representation from merely producing compiling syntax. Theo shares that concern, but uses auxiliary proof attempts as an additional signal when selecting definitions absent from the available library. Theo evaluates mathematical correspondence through its separate semantic review.

F.6.4 Semantic Faithfulness and Source Repairs

Evaluation beyond compilation. Zhang et al. (2026a) combines compilation with semantic consensus between two models and calibrates the resulting criterion against human review. Its experiments find a gap between compilation and faithful statement translation across all evaluated systems. FormalRx (Wang et al., 2026c) adds a diagnostic taxonomy and models for classifying, locating, and correcting semantic errors. These studies support evaluating statement correspondence separately from Lean acceptance and recording the kinds of discrepancies detected. Theo's Faithfulness Judge serves this separate semantic role, while auxiliary lemmas probe consequences of a candidate formalization.

Preserving incorrect inputs. FaithformBench (Cornish et al., 2026) tests whether autoformalizers preserve both valid reasoning steps and deliberately invalid perturbations, finding that systems can silently turn incorrect inputs into provable statements. The robustness study of Gui et al. (2026) similarly examines sensitivity to paraphrases and local alterations of informal proofs. These evaluations distinguish faithful translation from successful theorem proving. Theo's research-paper case studies document source repairs through explicit counterexamples, corrected hypotheses or conclusions, and the downstream uses of the corrected statements.

F.6.5 Evaluation Beyond Isolated Contest Problems

FormalTCS (Wang et al., 2026a) evaluates research-level theoretical computer science using paper-specific definitions, assumptions, and proof dependencies, including separate statement-formalization and proof tasks. LCS-Bench (Feng et al., 2026) addresses collections of interdependent definitions and results from logic and provides several evaluation tracks, including checks of definitional equivalence. MathAtlas (Patel et al., 2026) includes graduate-level textbook definitions and theorems together with their dependency structure. These benchmarks are closely aligned with the difficulties posed by Theo's research-paper cases: establishing formal vocabulary, maintaining consistency across a development, and reasoning beyond shallow library reuse. The position paper on theory-level autoformalization (Min et al., 2026) articulates the corresponding shift from isolated statements to coherent formal libraries.

Contest-oriented proof results remain useful but measure a different part of the pipeline. For example, Pythagoras-Prover (Leang et al., 2026) studies efficient specialized provers and augmented formal training data. This work complements Theo's emphasis on constructing and validating the definitions and dependencies needed to formalize research papers.

Relationship to Theo. These concurrent works share general coding agents, revisable proof plans, compiler feedback, and semantic review with Theo. Theo integrates generated auxiliary-lemma checks into definition-first formalization, recursive proof development, and explicit documentation of assumptions and source repairs.