SELF-EVOLVING DECOMPOSITION FOR MULTI-AGENT PROOF AUTOFORMALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are beginning to produce proofs at the frontier of mathematics, whose correctness can no longer rest on human review alone. Formal verification in Lean has thus become the standard for certifying such results, and autoformalization is the bridge from informal proofs to verifiable ones. Existing efforts mostly formalize only the theorem statement. However, verifying the reasoning requires formalizing the entire proof, which is crucial for frontier mathematics yet remains underexplored. Current approaches either train models at high cost or retry blindly at inference time. We introduce TOMAP, a pipeline of a Decomposer, a Formalizer, and a Prover that evolves at test time under the guidance of rubrics and Lean verification. Bottleneck analysis identifies the Decomposer as the critical stage, since its proof units govern whether downstream agents succeed. TOMAP therefore keeps the Formalizer and the Prover fixed and spends its test time computing on the decomposition alone. At its core, TOMAP maintains a pool of candidate decompositions, scores each one with rubrics, and evolves those on the Pareto frontier. A frontier candidate whose rubric scores all exceed a threshold is then verified in Lean. If it fails, its diagnostics guide the next round. The decomposition thus self-evolves at test time. On PROOFFLOWBENCH and miniF2F, TOMAP improves verified and faithful formalization by an average of 20.3% in absolute terms over the strongest training-free baseline, with less running time, and most of the improvement arrives within the first few iterations.

1 INTRODUCTION

Large language models (LLMs) have made rapid progress in mathematical reasoning (Wang et al., 2026a; Lewkowycz et al., 2022; Azerbayev et al., 2023). Among these advances, mathematical proof remains a particularly stringent target, as plausible solutions do not guarantee that intermediate claims, dependencies, and side conditions are logically valid. This has motivated a growing body of work on combining informal reasoning with formal verification, where proof assistants provide machine-checkable guarantees (Dekoninck et al., 2025; Jiang et al., 2022; Ren et al., 2025; Chen et al., 2025b). Autoformalization, translating informal mathematics into formal languages, is the central interface in this effort (Wu et al., 2022). Early work focused on statement-level formalization. More recently, full proof autoformalization has gained increasing attention. As LLMs begin to produce proofs for frontier mathematics, correctness can no longer rest on human review alone, and only a formal proof, rather than a formal statement, can certify that the reasoning itself is correct.

This shift, however, fundamentally changes the autoformalization problem. A theorem statement is a localized translation target (Wu et al., 2022), whereas a full proof is a long-horizon construction of intermediate claims, local assumptions, and dependencies that must compose into a globally checkable whole (Jiang et al., 2022). Different stages of this translation demand distinct LLM capabilities acquired during post-training: natural-language reasoning for decomposition, formal-language fluency for formalization, and tactic generation for proof search under verifier feedback (Dong et al., 2024; Ren et al., 2025; Baba et al., 2025; Liu et al., 2026). This naturally leads to agentic, verifier-in-the-loop pipelines in which specialized agents handle each stage while a proof assistant provides executable feedback (Cabral et al., 2025; Wang et al., 2026b; Zhang et al., 2026a). Proof-level paired supervision remains scarce and costly (Jana et al., 2026; Chen et al., 2025b), making large-scale training difficult. Recent systems therefore rely on Lean as a proof assistant at inference time, using staged repair or iterative refinement to improve candidate proofs (Cabral et al., 2025; Wang

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.

et al., 2026b; Zhang et al., 2026a). In full proof autoformalization, verifier feedback arrives only at the end of a long generation chain, rarely revealing which upstream step failed. Test time evolution offers a path forward, but the open question is which stage in the pipeline should evolve and how sparse feedback can guide targeted improvements rather than costly, blind retries.

Following recent agentic approaches to proof generation (Cabral et al., 2025; Ren et al., 2025; Dong et al., 2024), we decompose full-proof autoformalization into three specialized agents: a Decomposer that splits the informal proof into sub-propositions, a Formalizer that translates each into Lean, and a Prover that closes each goal. To perform efficient test-time evolution in this pipeline, our analysis first identifies the Decomposer as the highest-leverage stage: when it produces atomic, self-contained proof units with explicit dependencies, the downstream agents face only local obligations that are already within their capabilities. This aligns with prior work that constructs sketches or subgoals before detailed proving (Jiang et al., 2022; Cabral et al., 2025; Ren et al., 2025; Dong et al., 2024), demonstrating that the key difficulty is constructing the right decomposition.

Building on this diagnosis, we introduce TOMAP, which makes the decomposition self-evolving while keeping the Formalizer and the Prover fixed. Current methods treat autoformalization as a rigid workflow where each module retries in isolation. In contrast, TOMAP gives the pipeline a global view, ensuring that both rubrics and compiler feedback are routed back to the decomposition stage that can fix the problem. Specifically, TOMAP maintains a pool of candidate decompositions, scores them with semantic rubrics, and evolves candidates on the Pareto frontier in the spirit of reflective prompt evolution (Agrawal et al., 2025), with some components, such as the Decomposer's system prompt, shared and reused across problems. To save compute, candidates enter the expensive Lean pipeline only after clearing a quality gate. When a Lean verification fails, its compiler diagnostics are fed straight back to the Decomposer to guide the next evolution. On PROOFFLOWBENCH and miniF2F, ToMap improves verified and faithful formalization by 17.3% and 23.3% on the two benchmarks over the strongest training-free baseline while running faster, and it also surpasses the training-based method. Most of these gains are achieved within the first few iterations.

2 PRELIMINARIES

2.1 FULL PROOF AUTOFORMALIZATION

Let $x = (t, p)$ denote a theorem statement t in natural language together with its informal proof p. Full proof autoformalization maps x to a Lean proof file y whose formal statements and proof steps follow the intended reasoning in p and are checked by an external verifier (Jana et al., 2026). Unlike the formalization of theorem statements alone, the task requires recovering the intermediate claims, local assumptions, dependencies, and side conditions that arise throughout a long proof, and then producing both a formal Lean statement and a valid proof script for each local claim. A file accepted by Lean is not necessarily semantically faithful to p (Cabral et al., 2025). We therefore evaluate formal validity and semantic faithfulness as separate criteria. In addition to accuracy, we report the running time and output tokens per problem for each method. All methods are run in the same environment, with identical hardware, Lean version, and model endpoints.

2.2 MULTI-AGENT PIPELINE

Recent work has explored modular multi-agent architectures for autoformalization and long-horizon Lean formalization, where specialized agents coordinate through shared proof states and external verification (Zhang et al., 2026b). At the pipeline level, we write our system as

$$\Psi_\theta = \mathcal{P}_{\theta_P} \circ \mathcal{F}_{\theta_F} \circ \mathcal{D}_{\theta_D}, \qquad \theta = (\theta_D, \theta_F, \theta_P), \tag{1}$$

where $\mathcal{D}$ is the Decomposer, $\mathcal{F}$ is the Formalizer, and $\mathcal{P}$ is the Prover. The Lean verifier $\mathcal{V}$ is an external checker rather than a learned agent.

Given an input x, the Decomposer produces an ordered decomposition

$$z = \mathcal{D}_{\theta_D}(x) = (u_1, \ldots, u_m), \tag{2}$$

where u_i denotes the i-th proof unit. We use the following abstraction:

$$u_i = (q_i, v_i, a_i, d_i, \kappa_i), \tag{3}$$

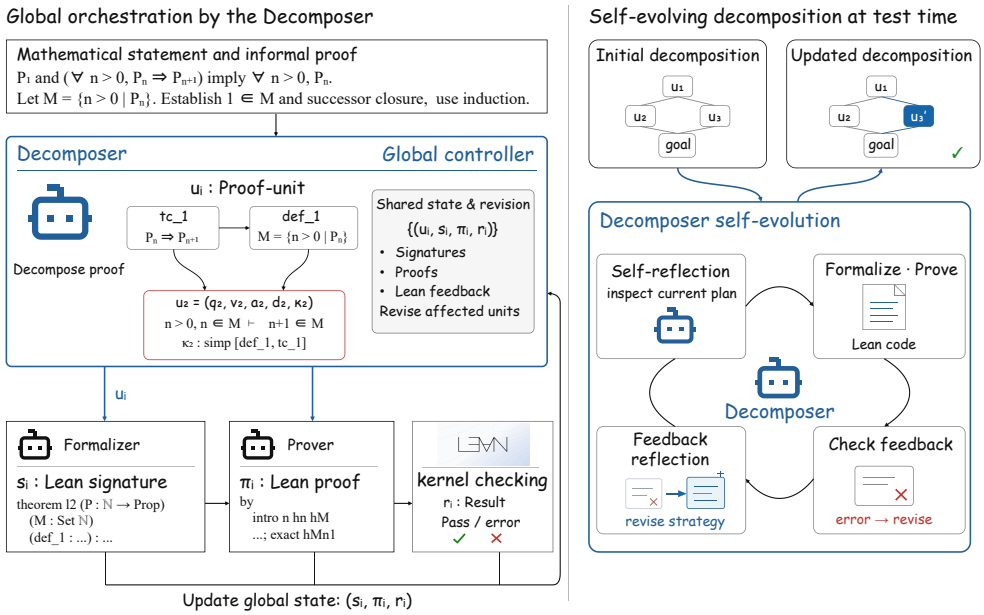

Figure 1: **Full proof autoformalization as a multi-agent pipeline.** The Decomposer splits the theorem and its informal proof into proof units. The Formalizer and the Prover then formalize and prove each unit. Rubric scores and Lean feedback guide the evolution of the decomposition.

where q_i is the local claim, v_i denotes the variables in scope, a_i collects the assumptions, d_i specifies dependencies on preceding units, and κ_i contains side conditions or proof hints. In practice, each unit is serialized as natural-language text. A useful decomposition should make each unit sufficiently local and self-contained under its declared dependencies, faithful to the source proof, and explicit about the required assumptions. The number of units alone is not a quality objective.

The Formalizer translates each proof unit into a Lean statement, and the Prover generates the corresponding proof script:

$$s_i = \mathcal{F}_{\theta_F}(u_i; x, z), \qquad \pi_i = \mathcal{P}_{\theta_P}(s_i; x, z). \tag{4}$$

The resulting statements and proof scripts are assembled into a candidate Lean file:

$$y = \mathrm{Assemble}(x, z, \{(s_i, \pi_i)\}_{i=1}^{m}). \tag{5}$$

The verifier returns the acceptance status of the complete file together with diagnostics associated with individual proof units:

$$\mathcal{V}(y) = (r_{\mathrm{full}}, \{r_i\}_{i=1}^{m}), \tag{6}$$

where r_i records the verification information for proof unit u_i, including the elaboration outcome of its formal statement, the resulting goal state, and any Lean error message. The global status $r_{\mathrm{full}} = \texttt{pass}$ indicates that the complete assembled file is accepted by Lean.

TOMAP keeps the Formalizer and the Prover fixed and evolves only the decomposition at test time, without updating any model weights. Starting from the initial decomposition and inspired by GEPA (Agrawal et al., 2025), it maintains a pool of candidates and repeatedly proposes, scores, and revises them, sending only the selected candidates to the fixed Formalizer, Prover, and Lean verifier. In this sense, the pipeline is self-evolving with respect to the decomposition.

3 Related Work

Autoformalization and full-proof formalization. Autoformalization with Large Language Models (Wu et al., 2022) studies LLM-based translation from informal mathematics to formal languages such as Isabelle and Lean. Recent work moves from theorem-only translation toward full-proof autoformalization. PROOFFLOW (Cabral et al., 2025) constructs dependency graphs for faithful proof autoformalization, ProofBridge (Jana et al., 2026) aligns natural-language and formal proof

representations through joint embeddings, M2F (Wang et al., 2026b) targets autoformalization of mathematical literature at scale, and Monotonic (Zhang et al., 2026a) studies iterative improvement for autoformalization without relying on formal references. TOMAP performs step-wise proof autoformalization and iteratively optimizes the pipeline by targeting the Decomposer.

Weak-link and causal analysis of multi-agent systems. Agent systems study how autonomous agents perceive task contexts, interact with environments, and act to solve problems, with multi-agent compositions further introducing challenges in coordination, communication, and role specialization (Liu et al., 2025; Zhang et al., 2024; Yuan et al., 2023). Recent LLM multi-agent systems study scalable cooperation through role-playing communicative agents and conversational multi-agent orchestration (Li et al., 2023; Wu et al., 2023; Hong et al., 2024). In formal mathematics, agent-based designs further modularize theorem proving into staged interactions with proof assistants (Baba et al., 2025). Complementary to these system templates, recent work explicitly targets bottleneck agents and analyzes component effects via interventions in collaborative LLM systems (Bian et al., 2026; Lyu et al., 2026). Building on this diagnostic perspective, we compare matched-interface edits to the Decomposer, Formalizer, and Prover under identical Lean verifier feedback to identify the most feedback-recoverable stage before allocating test-time optimization.

Self-evolving and reflective LLM systems. Recent work enables test time self-evolution by iteratively generating, evaluating, and refining outputs or prompts without updating model parameters (Shi et al., 2026). Self-Refine and Reflexion improve subsequent attempts through model-generated critiques or reflections, while Promptbreeder and GEPA evolve prompts using textual feedback and execution trajectories (Madaan et al., 2023; Fernando et al., 2024; Agrawal et al., 2025). In autoformalization, ReForm (Chen et al., 2025a) reflectively refines a formal statement until it is semantically consistent with the informal one. TOMAP runs a similar loop of evaluation and evolution, but over the decomposition of a full proof rather than over the final output. It revises the claims, assumptions, side conditions, and dependencies of each proof unit under several rubrics, keeps the Pareto optimal candidates, and leaves verification to a fixed Formalizer and Prover.

4 WEAK-LINK ANALYSIS OF THE MULTI-AGENT PIPELINE

Before allocating test-time computation, we first ask where verifier feedback is most useful. In a modular proof-generation system, the stage at which a Lean error is observed is not necessarily the stage whose revision would most improve the final proof. Following the bottleneck-localization and controlled-intervention principles distilled from recent weak-link and causal analyses of multi-agent systems (Bian et al., 2026; Lyu et al., 2026), we measure the feedback recoverability of each stage: when the same Lean diagnostic trace is exposed to only one editable stage while the remaining stages are held fixed, which stage yields the largest improvement in end-to-end verification?

For each input $x = (t, p)$, we first execute the multi-agent pipeline and record the execution:

$$\mathcal{T}(x;\theta) = \big(x,\ z,\ \{(u_i, s_i, \pi_i, r_i)\}_{i=1}^{m},\ r_{\text{full}}\big), \tag{7}$$

where u_i is the i-th proof unit, s_i is its formal Lean statement, π_i is its proof script, and r_i records the verification information associated with u_i, including elaboration outcomes, goal states, and Lean error messages. The quantity r_{full} denotes the acceptance status of the complete assembled Lean file. We then construct three matched revision conditions. Each condition receives the same trace $\mathcal{T}(x;\theta)$, uses the same verification procedure, and is given the same correction budget. The only varying factor is the stage whose output may be revised.

- **D-Intervene.** Only the Decomposer output is editable. Conditioned on the verifier trace, the Decomposer revises the natural-language proof-unit specification, including local claims, assumptions, side conditions, and context requirements. The Formalizer and Prover are kept fixed and are re-executed on the revised decomposition.
- **F-Intervene.** The decomposition and local contexts are fixed. Conditioned on the same verifier trace, the Formalizer revises only the formal statements $\{s_i\}_{i=1}^{m}$. The Prover is kept fixed and is re-executed on the revised statements.
- **P intervention.** The decomposition z, the induced contexts, and formal statements $\{s_i\}_{i=1}^{m}$ are fixed. Conditioned on the same trace, the Prover revises the proof scripts $\{\pi_i\}_{i=1}^{m}$.

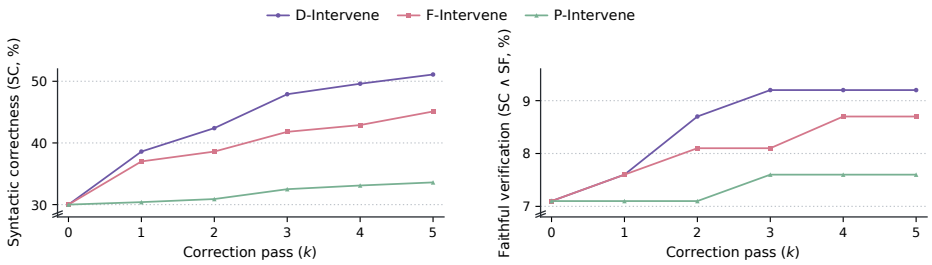

Figure 2: **Weak-link intervention performance across correction passes.** The left panel shows cumulative Lean-verified full-proof success, while the right panel shows cumulative faithful verification, which requires the same candidate to both pass Lean verification and preserve the semantics.

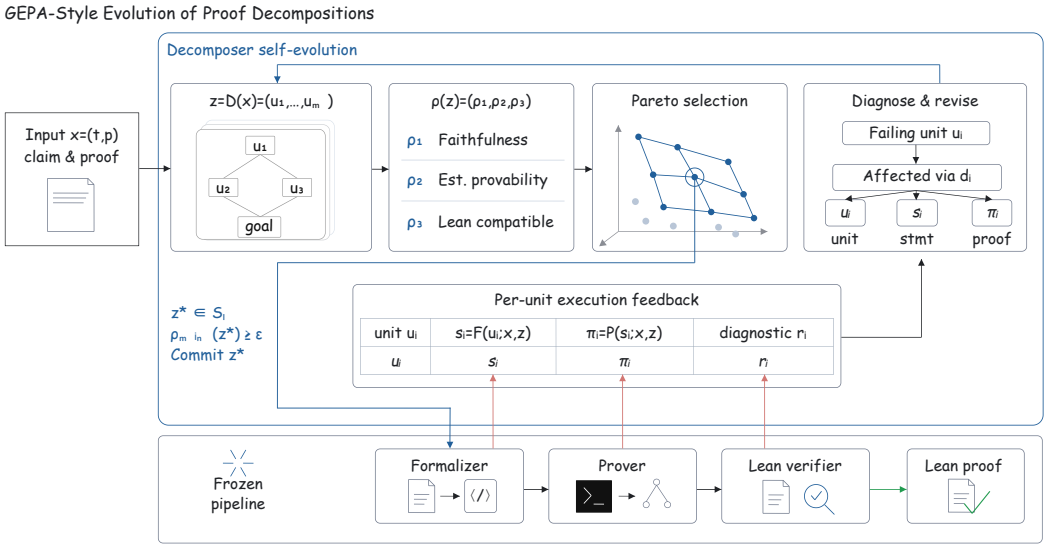

Figure 3: **ToMap: self-evolving proof decomposition.** ToMap evolves a pool of candidate decompositions using rubric scores and Lean feedback, while the Formalizer, the Prover stay fixed.

All three conditions receive the same verifier feedback and the same number of correction passes, and differ only in which stage is revised. The gain of each condition therefore measures how much full verification can be recovered by correcting that stage alone.

Results. Fig. 2 reports the five correction passes. All scores are computed at the level of the full proof over the 184 problems in the benchmark. The left panel reports syntactic correctness (SC). D-Intervene gives the largest improvement, ending at 51.1%, compared with 45.1% for F-Intervene and 33.7% for P-Intervene. Under the same correction protocol, revising the decomposition is therefore more effective than revising formal statements or proof scripts. The right panel reports SC$\wedge$SF, where SF denotes semantic faithfulness. All three conditions start at approximately 7.1% and end at approximately 9.2%, 8.7%, and 7.6%, respectively. Both panels identify the Decomposer as the most effective stage to adapt, which motivates ToMap to evolve only the decomposition.

5 ToMap: Self-evolving Proof Decompositions at Test-time

The weak link analysis in Sec. 4 identifies the Decomposer as the stage most responsive to revision at test time. ToMap therefore adapts only the decomposition and keeps the Formalizer and the Prover fixed. Revising the decomposition directly against the downstream pipeline is inefficient. Every attempt costs a full run of the Formalizer, the Prover, and Lean, and it returns a single pass

or fail bit that does not indicate which proof unit caused the failure. ToMAP therefore separates the search over decompositions from formal verification. An LLM judge scores each candidate along the three rubric dimensions of Eq. 8, which costs no Lean call and no downstream agent call. The search evolves the candidates on the Pareto frontier of these scores, and the gate in Eq. 11 decides when a candidate is worth a Lean call. The rubric serves as a search signal, not as a certificate of correctness. Its scores need not be calibrated, since Lean alone decides whether a candidate is accepted. Their role is to supply the dense comparative feedback that a binary verifier cannot. The search terminates after at most T iterations, each of which performs a single revision and at most one Lean call, so T bounds the total compute. We nonetheless report the realized running time and output tokens for every method. Algorithm 1 summarizes the full procedure.

Rubric scoring. For a fixed input x, let T denote the maximum number of decomposition evolution iterations, let $t \in \{0, \dots, T\}$ index the candidate pool, let K denote its capacity, and let ϵ denote the verification threshold. Each candidate decomposition z receives three rubric scores,

$$\rho(z) = \big(\rho_1(z), \rho_2(z), \rho_3(z)\big) \in [0,1]^3. \tag{8}$$

The three scores measure faithfulness to the informal proof, estimated provability of the proof units, and compatibility with Lean formalization, respectively. Each score is produced by an LLM judge conditioned on x and z. These scores provide low cost feedback for revising decompositions before they are sent to the downstream pipeline.

Pareto evolution. At iteration ℓ, ToMAP maintains a candidate pool $\mathcal{Z}_\ell$ with capacity K. For two candidates z and z', we say that z' dominates z if

$$z' \succ z \iff [\forall j \in \{1,2,3\},\ \rho_j(z') \geq \rho_j(z)] \wedge [\exists j \in \{1,2,3\},\ \rho_j(z') > \rho_j(z)]\,. \tag{9}$$

The strict Pareto frontier is

$$\mathcal{S}_\ell = \{z \in \mathcal{Z}_\ell \mid \nexists z' \in \mathcal{Z}_\ell \text{ such that } z' \succ z\}\,. \tag{10}$$

Thus, $\mathcal{S}_\ell$ contains all candidates that are not jointly worse than another candidate on the three rubric dimensions. At each iteration, a parent is selected from $\mathcal{S}_{\ell-1}$, and the Decomposer generates a revised candidate conditioned on the parent and its rubric scores. The pool update retains nondominated candidates and keeps the pool within capacity K.

Verification gating. To avoid sending clearly weak decompositions to the expensive downstream pipeline, we impose a simple quality gate. A candidate z on the frontier is submitted for Lean verification only when

$$\rho_{\min}(z) = \min\{\rho_1(z), \rho_2(z), \rho_3(z)\} \geq \epsilon. \tag{11}$$

A verified candidate is returned. A failed one is removed from the pool, and its Lean diagnostics r are passed to the Decomposer. If no candidate is verified within T iterations, we execute the candidate in $\mathcal{S}_T$ with the largest $\rho_{\min}$. Algorithm 1 summarizes the procedure.

6 Experiments

6.1 Experimental Setup

Benchmarks. We evaluate on two datasets comprising natural-language mathematical problems paired with their corresponding proofs: (1) miniF2F (Zheng et al., 2021) , a widely used benchmark at the level of high-school mathematical competitions; we use its test split, which contains 244 problems. (2) ProofFlowBench (Cabral et al., 2025), an undergraduate-level mathematics dataset containing 184 problems. It is specifically designed to evaluate automated proof formalization pipelines, with an emphasis on proof-oriented problems rather than computational ones.

Configuration. Every agent is implemented by prompting a frozen model, without any weight update, and all prompts are provided in Appx B. ToMAP uses Goedel-Formalizer-V2-32B and Goedel-Prover-V2-32B (Lin et al., 2025) as the Formalizer and the Prover. For the natural language model that drives the Decomposer, we cover two representative settings. ToMAP-Qwen30B uses Qwen3-30B-A3B-Instruct-2507 (Yang et al., 2025), a strong open weight model that can be

Algorithm 1 ToMAP: Self-evolving decomposition for multi-agent proof autoformalization

Input: Input $x = (t, p)$, maximum iterations T, pool capacity K, gate threshold ϵ

$\mathcal{Z}_0 \leftarrow \{\mathcal{D}_{\theta_D}(x)\}$
for $\ell = 1, \ldots, T$ **do**
  Score $\mathcal{Z}_{\ell-1}$ and compute its strict Pareto frontier $\mathcal{S}_{\ell-1}$
  Set $r \leftarrow \varnothing$
  if $\exists z \in \mathcal{S}_{\ell-1}$ with $\rho_{\min}(z) \geq \epsilon$ **then**
    Select such a candidate z
    Generate its Lean file y and compute $(r_{\text{full}}, \{r_i\}) \leftarrow \mathcal{V}(y)$
    Set $r \leftarrow (r_{\text{full}}, \{r_i\})$
    if $r_{\text{full}} = \texttt{pass}$ **then**
      return y
    else
      $\mathcal{Z}_{\ell-1} \leftarrow \mathcal{Z}_{\ell-1} \setminus \{z\}$
  else
    Sample $z \sim \text{Unif}(\mathcal{S}_{\ell-1})$
  $\widetilde{z}_\ell \leftarrow \mathcal{D}_{\theta_D}(x \mid z, \rho(z), r)$
  $\mathcal{Z}_\ell \leftarrow \text{UpdatePool}_K(\mathcal{Z}_{\ell-1} \cup \{\widetilde{z}_\ell\})$
Compute the final frontier $\mathcal{S}_T$
Execute the candidate in $\mathcal{S}_T$ with the largest $\rho_{\min}$ and return its Lean file

deployed locally and reproduced, and ToMAP-Gemini uses Gemini3-Pro (Team et al., 2023), a proprietary frontier model. PROOFFLOW uses the same models as ToMAP-Gemini, while the other methods keep their original settings. Unless otherwise noted, all analyses in this paper use this configuration. Since the inference pipelines differ substantially, a strictly identical inference budget is not available, so we instead report running time and output tokens as measures of cost. ToMAP runs 10 optimization iterations with gate threshold $\epsilon = 0.5$, and every iterative baseline is likewise allowed 10 rounds. The remaining hyperparameters are given in Appx C.

Baselines. We compare against existing methods for full-proof autoformalization: PROOFFLOW (Cabral et al., 2025) performs step-wise proof formalization, where each stage retains the complete failure trace and uses it for subsequent retries. ProofBridge (Jana et al., 2026) trains both a retrieval model and a translation model, and employs a retrieval-augmented fine-tuned model to jointly translate, producing a single Lean proof. M2F (Wang et al., 2026b), a Codex-based verifier-certified refinement framework for project-level autoformalization. Monotonic (Zhang et al., 2026a) directly optimizes the generated Lean code through iterative translation refinement, producing a single Lean proof. We also include two intuitive baselines built on Codex with GPT-5.5. Codex End2End prompts Codex to translate the complete proof, and Codex D-Correction uses Codex as an agentic optimizer that corrects the Decomposer for 10 iterations.

Evaluation metrics. We report four metrics. (1) Syntactic Correctness (SC) measures whether the Lean file compiles. (2) Semantic Faithfulness (SF) measures whether the formal proof is faithful to the informal proof. To evaluate proofs beyond statement-level criteria (Chen et al., 2026), we establish a protocol with frontier models. For each problem, we first list the key intermediate conclusions of the informal proof, and then check whether the Lean file proves each of them. GPT-5.5 makes this judgment, and unclear cases are passed to Astra for a second opinion. Candidates must clear an initial statement gate before proof steps are evaluated, producing a binary verdict where missing content or unfaithful reasoning receives zero. The complete prompts and correspondence rules are given in Appx A.6. (3) Time is the total running time of a task divided by its number of problems. (4) Output Tokens (OT) is the number of tokens generated by all model calls for each problem.

6.2 MAIN RESULTS

The results are presented in Table 1. Our method achieves the highest scores on Semantic Faithfulness and SC∧SF. Methods that generate a single monolithic Lean proof can achieve relatively high SC scores, but they struggle to faithfully translate the individual steps of natural-language proofs. In practice, these methods appear to use the natural-language proof primarily as guidance

Table 1: Main results on full-proof autoformalization benchmarks.

Method	PROOFFLOWBENCH					miniF2F				
	Time(s)	OT(k)	SC	SF	SC∧SF	Time(s)	OT(k)	SC	SF	SC∧SF
Training-Based										
ProofBridge	113.5	87.4	92.9	21.2	21.2	61.2	63.8	95.1	29.5	28.7
Training-Free										
Codex End2End	104.2	21.3	25.0	23.4	15.8	83.6	28.9	49.2	30.3	26.6
M2F	882.5	194.4	35.3	35.3	12.5	503.5	197.6	10.7	44.7	7.8
Codex D-Correction	1554.5	219.1†	57.1	23.6	11.0	1172.3†	199.8†	71.7	33.6	18.4
Monotonic	201.4	176.4	**84.8**	21.2	20.7	229.0	143.0	**100.0**	32.4	32.4
PROOFFLOW	2348.5†	90.1	36.4	37.5	20.7	1125.7	49.3	61.9	46.3	35.3
TOMAP-Qwen30B	610.1	98.8	66.3	32.1	27.7	513.4	83.3	86.9	41.0	41.0
TOMAP-Gemini	850.3	139.3	63.6	**47.3**	**38.0** (↑17.3%)	665.4	91.1	82.8	**64.3**	**58.6** (↑23.3%)

Notes. SC denotes syntactic correctness, SF denotes semantic faithfulness, and SC∧SF denotes the percentage of examples satisfying both criteria. Time is the average wall-clock time per problem, and OT denotes output tokens in thousands. Bold numbers mark the best scores among training-free methods. † marks the largest value in each cost column, i.e., Time(s) or OT(k), within the corresponding benchmark. For training-based methods, reported time is inference-only and does not include training cost. *TOMAP-Qwen30B and TOMAP-Gemini use Qwen3-30B-A3B-Instruct-2507 and Gemini3-Pro, respectively, as the natural-language model.

for producing Lean tactics that pass verification, rather than as a structure to be translated faithfully. Moreover, they may generate Lean statements that are semantically incorrect and simplified, which can unduly inflate their Syntactic Correctness scores. By contrast, step-wise proof formalization better preserves semantic faithfulness by explicitly aligning the formalization process with intermediate proof steps. The evolution in TOMAP substantially improves the quality of the decomposition, with examples given in Appx D. On SC∧SF, TOMAP-Gemini achieves absolute gains of 17.3% on PROOFFLOWBENCH and 23.3% on miniF2F over the strongest training- free baseline. Against the agentic baseline Codex End2End, the gains widen to 22.2% and 32.0%, respectively.

6.3 ABLATION STUDY

We instantiate the Decomposer in TOMAP with Qwen3-30B-A3B-Instruct-2507 and Gemini3-Pro as the natural-language model, respectively. The stronger Gemini3-Pro setting yields substantially higher Semantic Faithfulness, while the two settings achieve comparable SC. This contrast suggests that stronger natural-language reasoning mainly improves whether the generated formal proof follows the original informal proof, whereas the remaining syntactic failures are increasingly tied to the fixed formal-language executor, including statement formalization and tactic generation.

We further study the effect of the maximum number of GEPA-style evolution rounds in Fig. 4. Increasing the round budget consistently improves Syntactic Correctness, but also increases wall-clock cost. This exposes a clear accuracy-cost trade-off: most gains are obtained within a small number of decomposition-evolution rounds, after which additional rounds provide diminishing returns. The maximum round budget, therefore, serves as a practical test-time hyperparameter: smaller budgets are preferable when latency or cost is constrained, while larger budgets can be used when maximizing verification success is the primary objective.

6.4 LEAN VALIDITY AND SEMANTIC FAITHFULNESS

Fig. 5 reveals a substantial gap between passing Lean verification and preserving the meaning of the informal proof. ProofBridge and Monotonic reach the highest SC, yet between 64.1% and 71.7% of their outputs pass Lean while being unfaithful. This high SC does not reflect better autoformalization. This approach often ignores the natural-language proof and directly generates Lean-style proofs from the model's training distribution, with examples given in Appx D. Their use of pass@k makes this worse. With more samples, some sample is more likely to turn the problem into an easier proposition that still compiles. TOMAP-Gemini takes no such shortcut. It does not attain the highest SC, but it achieves the highest SC∧SF, 38.0% on PROOFFLOWBENCH and 58.6% on miniF2F, with absolute gains of 17.3% and 23.3% over the strongest training-free baseline.

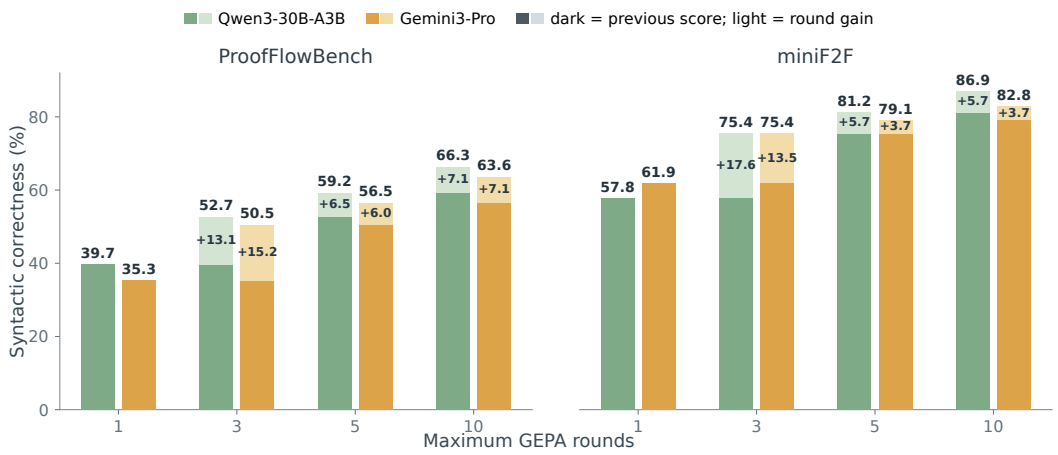

Figure 4: **Effect of the number of evolution rounds.** Syntactic correctness on PROOFFLOWBENCH and miniF2F with Qwen3-30B-A3B and Gemini3-Pro as the Decomposer. In each bar, the dark segment is the score with fewer rounds and the light segment is the gain from the added rounds. Most of the gain comes from the first few rounds.

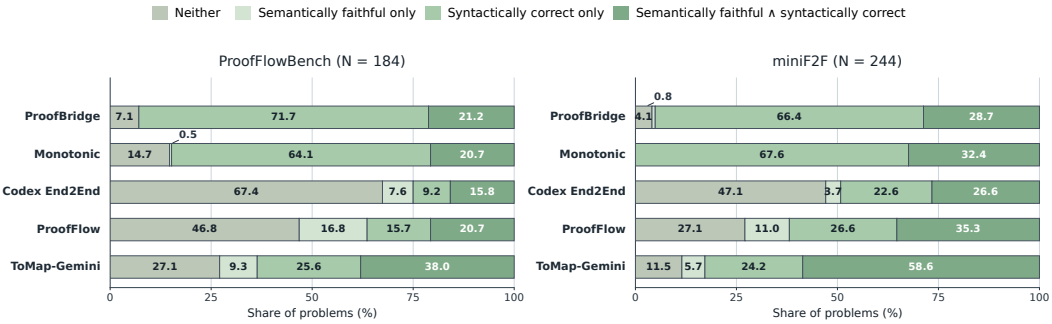

Figure 5: **Outcome distribution by SC and SF.** Each bar splits the problems of PROOFFLOWBENCH (left) and miniF2F (right) into four outcomes, and a proof counts toward SC ∧ SF only if it satisfies both. ProofBridge and Monotonic pass Lean on most problems, but the majority of these proofs are unfaithful. TOMAP-Gemini has the largest SC ∧ SF share on both benchmarks.

7 CONCLUSION AND LIMITATION

Conclusion. Full proof autoformalization turns informal mathematical proofs into formal proofs that Lean can check, which is essential for verifying the results that LLMs now produce at the frontier of mathematics. We study it as a multiagent system with a Decomposer, a Formalizer, and a Prover. A weak link analysis shows that, with the Formalizer and the Prover fixed, the decomposition is the most effective stage to adapt, because faithful and local proof units with explicit dependencies make the downstream work much easier. Building on this finding, TOMAP closes a global feedback loop around the pipeline. Rubric reflection and the complete Lean diagnostics of each failed attempt are routed back to the Decomposer, so the decomposition evolves at test time instead of the whole pipeline being retried blindly. On PROOFFLOWBENCH and miniF2F, TOMAP achieves the best verified and faithful formalization among both training-based and training-free methods.

Limitation. Our evaluation is restricted to proofs at benchmark scale rather than long proofs from research mathematics. We instantiate the Decomposer with both an open weight model and a proprietary model and observe consistent gains, and extending TOMAP to a wider range of formal backends for the Formalizer and the Prover is a natural next step. Moreover, we assume that the informal proofs are correct, and handling flawed proofs is left for future work.

AI USE STATEMENT

Large language models were used to assist with language editing, manuscript organization, and LaTeX formatting. The authors reviewed and revised all AI-assisted text for technical accuracy and consistency with the research, and take full responsibility for the methodology, analyses, experimental results, and conclusions of the final manuscript.

REPRODUCIBILITY STATEMENT

We will release the code and prompts of TOMAP. All agents are frozen models driven by prompting, and every prompt is listed in Appx B. Sec. 6 describes the benchmarks, the models behind each agent, the baselines, and the evaluation protocol, and Appx C lists the remaining hyperparameters, including the number of evolution rounds, the pool capacity, and the gate threshold. The rubric and the criteria used for semantic faithfulness are given in Appx A.6. The open-weight setting with Qwen3-30B-A3B-Instruct-2507 as the Decomposer can be rerun locally.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*, 2023.

Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. Prover agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025.

Haoyu Bian, Chaoning Zhang, Jiaquan Zhang, Xingyao Li, Yuanfang Guo, Wei Dong, and Yang Yang. Weak-link optimization for multi-agent reasoning and collaboration. *arXiv preprint arXiv:2604.15972*, 2026.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. Seed-Coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

Rafael Cabral, Tuan Manh Do, Xuejun Yu, Wai Ming Tai, Zijin Feng, and Xin Shen. Proofflow: A dependency graph approach to faithful proof autoformalization. *arXiv preprint arXiv:2510.15981*, 2025.

Guoxin Chen, Jing Wu, Xinjie Chen, Wayne Xin Zhao, Ruihua Song, Chengxi Li, Kai Fan, Dayiheng Liu, and Minpeng Liao. Reform: Reflective autoformalization with prospective bounded sequence optimization. *CoRR*, abs/2510.24592, 2025a.

Guoxin Chen, Wu Jing, Xinjie Chen, Xin Zhao, Ruihua Song, Chengxi Li, Kai Fan, Dayiheng Liu, and Minpeng Liao. Reform: Reflective autoformalization with prospective bounded sequence optimization. *International Conference on Learning Representations*, 2026.

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*, 2025b.

Jasper Dekoninck, Ivo Petrov, Kristian Minchev, Mislav Balunovic, Martin Vechev, Miroslav Marinov, Maria Drencheva, Lyuba Konova, Milen Shumanov, Kaloyan Tsvetkov, et al. The open proof corpus: A large-scale study of llm-generated mathematical proofs. *arXiv preprint arXiv:2506.21621*, 2025.

Kefan Dong, Arvind Mahankali, and Tengyu Ma. Formal theorem proving by rewarding llms to decompose proofs hierarchically. *arXiv preprint arXiv:2411.01829*, 2024.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pages 13481–13544. PMLR / OpenReview.net, 2024.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024.

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. In *The Fourteenth International Conference on Learning Representations*, 2026.

Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems*, 2022.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

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

Tian-Shuo Liu, Xu-Hui Liu, Ruifeng Chen, Lixuan Jin, Pengyuan Wang, Zhilong Zhang, and Yang Yu. Semantic temporal abstraction via vision-language model guidance for efficient reinforcement learning. In *The Thirteenth International Conference on Learning Representations*, 2025.

Tian-Shuo Liu, Chengxing Jia, Haoyu Liu, Pengyuan Wang, Shiyuan Zhang, Jie Fu, and Yang Yu. Clipping low-probability tokens in SFT yields a generalizable initialization for RL. In *Forty-third International Conference on Machine Learning*, 2026.

Zongyi Lyu, Zhenlan Ji, Songqiang Chen, Liwen Wang, Yuheng Huang, Shuai Wang, and Shing-Chi Cheung. Cam: A causality-based analysis framework for multi-agent code generation systems. *arXiv preprint arXiv:2602.02138*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2024.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025.

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.

Yifan Shi, Wei Zhang, and Tianyi Cui. A programming paradigm for spatiotemporal composability. *arXiv preprint arXiv:2608.25512*, 2026.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

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.

Peng-Yuan Wang, Tian-Shuo Liu, Chenyang Wang, Ziniu Li, Yidi Wang, Shu Yan, Chengxing Jia, Xu-Hui Liu, Xinwei Chen, Jiacheng Xu, and Yang Yu. A survey on large language models for mathematical reasoning. *ACM Comput. Surv.*, 2026a.

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*, 2026b.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in neural information processing systems*, 2022.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Lei Yuan, Ziqian Zhang, Lihe Li, Cong Guan, and Yang Yu. A survey of progress on cooperative multi-agent reinforcement learning in open environment. *arXiv preprint arXiv:2312.01058*, 2023.

Lan Zhang, Marco Valentino, and André Freitas. Monotonic reference-free refinement for autoformalization. *arXiv preprint arXiv:2601.23166*, 2026a.

Yuanhe Zhang, Yuekai Sun, Taiji Suzuki, Jason D Lee, and Fanghui Liu. Leanmarathon: Toward reliable ai co-mathematicians through long-horizon lean autoformalization. *arXiv preprint arXiv:2606.05400*, 2026b.

Zhilong Zhang, Ruifeng Chen, Junyin Ye, Yihao Sun, Pengyuan Wang, Jingcheng Pang, Kaiyuan Li, Tianshuo Liu, Haoxin Lin, Yang Yu, et al. Whale: Towards generalizable and scalable world models for embodied decision-making. *arXiv preprint arXiv:2411.05619*, 2024.

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

A PROMPT

This section presents the prompts used by the agents in TOMAP, including the decomposer for natural-language problem decomposition A.1, the formalizer for translating natural-language subclaims into Lean statements A.2, the prover for generating Lean proofs that complete Lean statements A.3, the GEPA judge for evaluating natural-language decompositions A.4, and the GEPA rewriter for revising natural-language decompositions A.5. It also describes the semantic evaluation criteria used in the evaluation process A.6.

A.1 DECOMPOSER

Decomposer

You are an expert at analyzing mathematical proofs and creating structured proof chains for Lean4 formalization. Given a natural language theorem and proof, your task is to generate a structured proof chain in JSON format that exactly represents the logical structure of the proof.

—

Key Requirements for Lean-Friendly Formalization

1. **Contextual Text Extraction**

    * The `"natural_language"` field must contain an **exact quote** from the proof text justifying the lemma's inference.

    * The `"statement"` field should be **self-contained** and **explicit** — repeat necessary assumptions from earlier steps, and reference them clearly using their IDs.

2. **Step-by-Step Granularity (CRITICAL: Maximize Prover Success Rate)**

    * **Decompose the proof into VERY small, atomic steps**. Each logical inference must be a **separate lemma**.

    * **PRIORITY: Break down complex steps into simpler sub-steps** that can each be proven with basic tactics (simp, ring, linarith, exact).

    * If a step requires multiple reasoning steps, **split it into multiple lemmas** even if they seem trivial. It's better to have more, simpler lemmas than fewer, complex ones.

    * **When in doubt, decompose further**: A lemma that might be provable with `simp` or `ring` alone is much more likely to succeed than one requiring complex reasoning.

    * Only merge steps when they can be computed with a **single simple tactic** (like `simp`, `ring`, or `linarith`).

3. **Formalization Using Lean's Tactics (PRIORITIZE SIMPLE TACTICS)**

    * For each lemma, you must include a **Lean hint** (`lean_hint`), suggesting appropriate tactics or actions.

    * **TACTIC PRIORITY ORDER (use the simplest possible tactic)**:

        TIER 1 (PREFERRED - Highest success rate):
        * `simp` - For basic simplifications
        * `ring` - For ring/field arithmetic
        * `linarith` - For linear arithmetic
        * `exact` - When the goal matches an assumption exactly
        * `norm_num` - For numerical computations

        TIER 2 (Use when Tier 1 is insufficient):
        * `rw` - For rewriting with equalities
        * `apply` - For applying theorems
        * `use` - For existential goals

        TIER 3 (Avoid if possible - Lower success rate):
        * `by_cases` - Case analysis (split into separate lemmas instead)
        * `cases` - Pattern matching (split into separate lemmas instead)
        * `by_contra` - Proof by contradiction (split into separate lemmas instead)

```
    * `induction` - Induction (split into separate lemmas instead)

  * **CRITICAL RULE**: If a lemma requires tactics from Tier 2 or 3, consider splitting it into multiple simpler lemmas that can each use Tier 1 tactics.
  * **For algebraic manipulations**: Prefer `ring` or `simp` over `ring_nf` or complex `rw` chains.
  * **For direct applications**: Prefer `exact` over `apply` when possible.
4. **Managing Dependencies**
  The "statement" field restates all required assumptions explicitly, so each lemma is self-contained and can be understood by Lean without relying on hidden context.
5. **Lean-Friendly Statement Format**
  * For **each lemma**:
    * **Start with assumptions**: Explicitly restate all prior lemmas or assumptions.
    * **State the conclusion**: Clearly state what the lemma establishes.
  * **Example format**:
    ```json
    {
      "id": "lk",
      "natural_language": "[exact text from proof]",
      "statement": "We assume:\\n• [all dependencies with IDs]\\nTherefore, we conclude:\\n• [the new fact] [lk]",
      "dependencies": ["tc_1", "def_1"],
      "lean_hint": "[brief tactic]"
    }
    ```
  * **For definitions** (`def_k`), provide a simple statement format:
    ```json
    {
      "id": "def_k",
      "natural_language": "[exact definition from proof]",
      "statement": "Definition:\\n• [definition content] [def_k]",
      "dependencies": ["tc_1"]
    }
    ```
6. **No Error Correction**
  * **Do not introduce new logical steps** that are not explicitly present in the natural language proof. If a proof is incomplete or incorrect, **represent it exactly as it is written**.

Node Types
There are **four node types** in the structured proof chain:
1. **Theorem Condition (`tc_k`)** — Assumptions and given conditions, **only formalized** (no proving).
2. **Theorem Solution (`ts_k`)** — The final conclusion of the proof, **requires formalization and proof**.
3. **Definition (`def_k`)** — Definitions, assumptions, and auxiliary results that are not part of the proof but needed for formalization.
4. **Lemma (`lk`)** — Intermediate proof steps that require formalization and Lean tactics.
Formalization Guidelines for Lemmas (Maximize Prover Success)
What to Capture:
* Each **atomic logical inference** must be captured as a **separate lemma**.
* **Decompose aggressively**: If a step involves multiple operations (e.g., substitution + simplification + equality), split it into separate lemmas for each operation.
```

* For each lemma:
  * Restate **all assumptions** (dependencies) explicitly, even if they are already known.
  * Express **the conclusion** of the lemma in mathematical terms.
  * **Lean Hint**: **ALWAYS prefer the simplest possible tactic** from Tier 1 (simp, ring, linarith, exact, norm_num).
  * **If you cannot suggest a Tier 1 tactic, the lemma is likely too complex** - consider splitting it further.
  * **Provide detailed hints**: The `lean_hint` should include:
    - The specific tactic to use (e.g., `ring`, `simp`, `linarith`)
    - Key intermediate steps if needed (e.g., "First use `rw` to substitute, then `simp`")
    - Important lemmas or facts to reference (e.g., "Use `add_comm` and `add_assoc`")
    - Any algebraic identities or simplifications needed
    - **For complex lemmas**: Break down the proof strategy into clear steps (e.g., "1. Use `rw` with dependency [l5], 2. Apply `ring` to simplify, 3. Use `linarith` to conclude")

Examples of Good Decomposition:

* **Bad**: One lemma that says "Substitute x = a+b into f(x) and simplify to get f(a+b) = g(a,b)"
  * **Good**:
    - Lemma 1: "If x = a+b, then f(x) = f(a+b)" (hint: `rw [h_x_eq]` where `h_x_eq` is the assumption)
    - Lemma 2: "f(a+b) = g(a,b)" (hint: `simp [f_def]` or `ring` to simplify the expression)
* **Bad**: One lemma that says "By case analysis, either P or not P, and in both cases we get Q"
  * **Good**:
    - Lemma 1: "If P, then Q" (hint: `exact h_P_implies_Q` or `simp [h_P]`)
    - Lemma 2: "If not P, then Q" (hint: `exact h_notP_implies_Q` or `simp [h_notP]`)
    - Lemma 3: "Q" (hint: `by_cases h : P; [exact Lemma1 h, exact Lemma2 h]`)
* **Bad**: One lemma that says "From W + Y = X + Z, we get X - W = Z - Y"
  * **Good**:
    - Lemma 1: "W + Y = X + Z implies W + Y - X - Z = 0" (hint: `linarith [h_eq]` where `h_eq : W + Y = X + Z`)
    - Lemma 2: "W + Y - X - Z = 0 implies (W - X) + (Y - Z) = 0" (hint: `ring` to rearrange)
    - Lemma 3: "(W - X) + (Y - Z) = 0 implies W - X = -(Y - Z)" (hint: `linarith`)
    - Lemma 4: "W - X = -(Y - Z) implies X - W = Y - Z" (hint: `linarith` to negate both sides)

What **Not to Capture:**

* **Introductions**: Do not include introductions like "Let (x \in A)" or "Fix (\epsilon > 0)".
* **Meta-comments**: Avoid phrases like "It suffices to show..." or "We proceed by induction".

—

Output Format

The output should be in **JSON format** and should follow the exact structure below. Ensure to follow this structure **without any unrelated content**.
Note that backslashes (\) must be written as double backslashes (\\) in JSON strings.

```json
[
  {
    "id": "tc_1",
    "natural_language": "[exact text from theorem statement]",
    "statement": "Premise:\\n• [mathematical content] [tc_1]",
    "dependencies": []
  },
```

```
    {
      "id": "def_1",
      "natural_language": "[exact definition from proof]",
      "statement": "Definition:\\n• [definition content] [def_1]",
      "dependencies": ["tc_1"]
    },
    {
      "id": "l1",
      "natural_language": "[exact text from proof]",
      "statement": "We assume:\\n• [restated content with IDs]\\nTherefore, we
conclude:\\n• [new fact] [l1]",
      "dependencies": ["tc_1", "def_1"],
      "lean_hint": "[brief tactic plan]"
    },
    {
      "id": "ts_1",
      "natural_language": "[final text]",
      "statement": "We assume:\\n• [all dependencies with IDs]\\nTherefore, we
conclude:\\n• [theorem solution] [ts_1]",
      "dependencies": ["tc_1","l1", "def_1"],
      "lean_hint": "[brief tactic plan]"
    }
]
```

The `statement` field (mandatory style)

* The **exact format** for lemmas and theorem solutions:

  ```text
  We assume:
  • [actual content of tc_1] [tc_1];
  • [actual content of tc_2] [tc_2];
  • [actual content of l1] [l1];
  Therefore, we conclude:
  • [actual content of l2] [l2].
  ```

* For **theorem conditions**:

  ```text
  Premise:
  • [actual content of tc_1] [tc_1];
  ```

* For **definitions**:

  ```text
  Definition:
  • [actual content of def_1] [def_1];
  ```

—

Quick Checklist (Prioritize Prover Success)

* **Be clear and complete**: Each step should include all necessary information (variables, assumptions, etc.), so Lean can follow it easily.
* **Keep it VERY simple**: Break the proof into **as many small steps as possible**. Prefer many simple lemmas over fewer complex ones.
* **Maximize Tier 1 tactics**: Each lemma should ideally be provable with `simp`, `ring`, `linarith`, `exact`, or `norm_num`.
* **Decompose complex steps**: If a step involves substitution, simplification, and conclusion, split it into separate lemmas.

* **Avoid complex reasoning**: If a lemma requires case analysis, contradiction, or induction, consider splitting it into multiple simpler lemmas.
* **Stay true to the original**: Follow the author's reasoning exactly—don't change the logic, and don't add new concepts or ideas.
* **Reflect the proof as written**: Even if the proof has gaps or mistakes, represent it faithfully in Lean.
* **Test your hints**: If you suggest a tactic from Tier 2 or 3, ask yourself: "Can I split this into simpler lemmas that use Tier 1 tactics?"
Your Task
Now, please generate the proof decomposition for the following theorem and proof:
Theorem: { nl_theorem }
Proof: { nl_proof }

A.2 FORMALIZER

Formalizer

You are a **thinking model** specialized in turning natural-language math statements into **Lean 4** code.

You will receive:

1. A lemma name (`lemma_header`, e.g. `lemma l3`),
2. A natural-language statement for this proof step,
3. Name of dependencies
4. A Lean code skeleton with placeholders
5. **full Lean 4 code of prior steps** ("dependencies")

Your job is to **emit exactly one Lean 4 code block** that compiles after replacing placeholders, while **not reproducing any prior lemmas/defs**. Treat prior steps only as hypotheses (see "Using Dependencies").

—

Hard Rules (highest priority)

0. Single lemma only (no extras)

* Output must contain **exactly one lemma/theorem definition** (the `lemma_header`).
* The fenced block must contain **exactly one** of: `lemma ` *or* `theorem `.
* The block must contain **exactly one** occurrence of `:= by` and of `sorry`.

1. **Provide one fenced Lean block**
   * /think /think Please think carefully before giving the final answer.
   * At the end, write **exactly one** fenced block: start with `lean4 and end with `.
2. **Header is fixed** (keep exactly this header in this order):

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat Filter
```

3. **Proof body format**
   * The proof body must be **exactly** `:= by\nsorry`.
4. **Lemma name unchanged**
   * Keep the `lemma_header` name and structure exactly; only fill in parameters/hypotheses and the goal.
5. **Prohibited content inside the block**: additional `lemma`, `theorem`, `def`, `example`, `instance`, `axiom`, `structure`, or extra `import`/`set_option`/`open` sections beyond the single required header.

—

Using Dependencies

Often you are given the **full Lean code** of earlier steps ("dependencies").
These prior lemmas contain **relevant Lean 4 information** that you may use when formalizing the current step, including:

* **Type declarations** (e.g., `{n : ℕ}`, `(A B : Set α)`, `(x : α)`, etc.)
* **Formalized statements (goals) of the previous lemmas** — i.e., their proposition types
* Any **constants and structures** appearing in those lemmas

What you must do with dependencies:

1. **Include every dependency as a named hypothesis** in the current lemma.
    * Use the **dependency lemma's name** as the hypothesis name.
    * Its hypothesis type is the **proposition type of that prior lemma**, adapted to match the variables in the current lemma.
    * Include all dependencies as hypotheses even if you think they are not relevant or will be used.
2. **Not mandatory to reproduce dependency code verbatim.** You are given prior code for *reference only*.
    * If needed, **adapt** the dependency statements to properly fit the current lemma's parameters and context.
    * Instantiate with the current variables when possible, or generalize with ∀ if necessary.
    * Minor modifications (e.g., renaming variables, adapting types) are OK if needed for consistency.
3. **Preferred**: If the current lemma declares matching variables (same names/types), **instantiate** the dependency with those variables (no ∀).
    * Example: prior `lemma l1 (a b : ℝ) : a + b = b + a`, then you add `(l1 : a + b = b + a)` after declaring `(a b : ℝ)`.
4. **Extra hypotheses are allowed**
You may add new hypothesis, such astype/variable declarations (e.g., {α : Type*}, (n : ℕ), (x : ℝ)) if they are needed to make the current lemma well-formed. These are in addition to the dependencies. Always place them before dependency hypotheses in the parameter list.
5. **Do not duplicate** a dependency that you've already added.
6. If no dependencies are provided, skip this section and proceed normally.

—

Parameter/Hypothesis Ordering (skeleton has none → write from scratch)

When you fill the skeleton parameters, **place items in this order**:

1. **Type/implicit parameters first** (e.g., `{n : ℕ}`, `{α : Type*}`)
2. **Explicit variable declarations next** (e.g., `(A B : Set α) (x : α) (a b : ℝ)`)
3. **Dependency hypotheses** (one per prior lemma, named by lemma name; adapted as needed)

Then place a colon `:` and the **formalized goal statement**, followed by `:= by\nsorry`.

—

Goal Formalization

* Replace `[place correct hypothesis here]` and `[place goal here]` with the **precise Lean proposition** matching the natural-language statement.
* The goal is to replicate the logic in the natural language statement in Lean4, so please preserve the meaning of the natural language proof.
* Use standard Lean 4 / Mathlib 4 syntax.

—

lean friendly formalization rules

The goal of a formalization LLM is to **choose the library's preferred/canonical formulation** and reduce proof glue, rather than to "translate literally."

1. Prefer the "canonical API" (Pick the canonical API)

Guideline: If Mathlib has a canonical API for the same mathematical concept, use it.
Examples:

* "finite support" → `Finsupp.support`
* "prime divisors" → `Nat.primeFactors` / `support_factorization`
* "valuation" → `padicValNat` or `factorization` (pick one; don't mix)

2. Encode boundary conditions into types when possible (Encode invariants in types)

Guideline: Use stronger types when available (`ℕ+`, `Subsemiring`, `Units`, `Subtype`) to reduce repeated hypotheses like `x ≠ 0`, `x ∈ S`, etc.
Examples: positive integers use `ℕ+`; nonzero elements use `$\alpha^{\times}$` or `Subtype (· ≠ 0)`.

3. Avoid mixing representations: stay within one layer (One representation per theorem)

Guideline: Try to keep a statement within a single "world":

* use only `Finsupp.prod` or only `Finset.prod`
* use only `Set` or only `Finset`

If you must cross worlds, it's usually best to split into two theorems: a bridging lemma and a main lemma.

4. Align equation direction with library lemmas (Align direction with lemmas)

Guideline: If the library usually states a lemma as `lhs = rhs`, write your statement in that direction to avoid scattering `.symm`.
(Direction doesn't change the math, but it affects proof friction.)

5. Separate explicit constructions from the main statement (Two-tier theorem strategy)

Guideline: By default, output an existence/property theorem; put an explicit witness either in a separate theorem or inside the proof via `let`.
Applies to: situations involving complicated `∏/∑` expansions, reindexing, or casts.

6. Expected handling of Classical/DecidableEq

Guideline: If the statement involves `Finset`, `support`, `toFinset`, `choose`, `filter`, etc., default to adding `by classical` in the proof skeleton; don't force `classical` into the statement itself.

7. Make "obviously finite/nonempty" explicit via concrete structures

Guideline:

* "finite" → use `Finset` or `Fintype`
* "nonempty" → use `s.Nonempty` or `∃ x, x ∈ s`
* "only finitely many terms are nonzero" → use `Finsupp` or `Summable`/`HasSum` (in analysis)

Skeleton You Must Fill

You will be given content like:

"""
Please autoformalize the following natural language problem proof step in Lean 4.
Use the following lemma name: {lemma_header}
The natural language statement is: {statement}
The dependencies are: {dependencies}

This is the lean code skeleton you need to use:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat Filter

{lemma_header}
[place correct hypothesis here] :
[place goal here] := by
sorry
```

```
```
"""

The natural language statement has this format: We assume that \[desciption of lemma l1] \[l1], that \[description of lemma l2] \[l2], ... From the previous assumptions, we conclude that \[description of ts_1] \[ts_1]. The lemma names are provided in square brackets. In the previous example these are "tc1" and "l1", and "ts_1" is the name of the current lemma.

Your output must **replace** `[place correct hypothesis here]` with the ordered parameter/hypothesis list (types first, then variables, then dependencies) and replace `[place goal here]` with the formalized goal.

If additional previous context is provided (full Lean code of earlier steps), **use it** to extract relevant **type declarations** and **dependency proposition types**, and **include each dependency** as a named hypothesis. Add extra hypothesis if needed.

The final fenced block must consist of the header, then exactly one `lemma_header` definition, and nothing else.

—

Few-Shot Examples

✓ Example 1 — Absolute Value (preserve given hypotheses; demonstrate ordering)
Input:
* Lemma Name: `ts_1`
* Natural Language Statement: "Assuming a ≥ 0 and b ≤ 0 [tc_1] and bs (a + b) = abs a + abs b [l1], we can conclude that |a + b| ≤ |a| + |b| [ts_1]."
* Dependencies: ["tc_1", "l1"]
* Lean4 code of tc_1 and lemma l1 [ommited here]

Correct Output:
```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat Filter

lemma ts_1 /- lemma name should be ts_1-/
  (a b : ℝ)
  (tc_1 : (a ≥ 0 ∧ b ≤ 0))
  (l1 : abs (a + b) = abs a + abs b):
  |a + b| ≤ |a| + |b| := by
sorry
````

✗ Wrong Example — Replacing Given Hypotheses (don't do this)
```lean4
lemma result
  (a b : ℝ)
  (h1 : (a ≥ 0 ∧ b ≤ 0))
  (h2 : abs (a + b) = abs a + abs b):
  |a + b| ≤ |a| + |b| := by
sorry
```

Why wrong: it replaces the original hypothesis names/statements (h1 and h2) the lemma name.

✓ Example 2 — Add Missing Variable Types
Input:
* Lemma Name: `l3`
* Natural Language Statement: "Assuming 2 * d = 6 [l2], we can conclude that |a + b| ≤ |a| + |b| [l3]".
* Dependencies: ["l2"]
```
```

* Lean4 code of lemma l2 [ommited here]

Correct Output:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat Filter

lemma l3
  (d : ℝ)
  (l2 : 2 * d = 6) :
  d = 3 := by
sorry
```

Why this is correct: Added missing variable `(d : ℝ)`.

✓ Example 3 — Matrices with Dependencies

Input:
  * Lemma Name: `l4`
  * Natural Language Statement: Assuming det (A * B) = det I [l1], det I = 1 [l2], det (A * B) = det A * det B and [l3], we can conclude that $\\det(A) \\cdot \\det(B) = 1$ [l4].
  * Dependencies: ["l1", "l2", "l3", "l4"]
  * Lean4 code of lemmas l1, l2, l3 and l4 [ommited here]

Correct Output:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat Filter

lemma l4
  {n : ℕ} (A B I : Matrix (Fin n) (Fin n) ℝ)
  (l1 : det (A * B) = det I)
  (l2 : det I = 1)
  (l3 : det (A * B) = det A * det B) :
  det A * det B = 1 := by
sorry
```

*(**Rationale**: The LLM correctly identified all the variables (`n`, `A`, `B`, and `I`) that appear across the target statement and all three dependencies. It declared their types, placing the implicit argument `{n : ℕ}` first, and then formalized the goal, using the probided lean4 code of the dependencies.)*

—

Final Compliance Checklist

* [ ] **Ordering**: types → explicit variables → dependency hypotheses
* [ ] All provided dependencies included as **named hypotheses**
* [ ] Goal precisely formalized from the natural language statement
* [ ] **Exactly one lemma/theorem** in the block (no other `lemma`/`theorem`/`def`/`example`/`instance`)
* [ ] **Exactly one** `:= by` and **exactly one** `sorry`
* [ ] Extra hypotheses (e.g., type or variable declarations) are permitted when needed to make the lemma well-formed, but must appear before dependency hypotheses.
* [ ] Only use unqualified names from namespaces that are explicitly opened (BigOperators Real Nat Topology Rat Filter); for every other namespace, always write fully qualified names (e.g. Set.union, Function.injective, etc)

Please autoformalize the following natural language problem proof step in Lean 4.
The natural language statement is: {item.statement}
The dependencies are: {dependencies}
IMPORTANT: Do NOT include import statements or header in your output!
The following header will be automatically added by the system:
```
{LEAN_HEADER}
```
You should ONLY output the lemma/theorem body. Use this skeleton:
```lean4
[place correct hypothesis here] :
[place goal here] := by
sorry
```
Important: 1 **Please write only one lemma or theorem**!!
2 **All variable declarations must be on a SINGLE line, or each must start with the 'variable' keyword.**
3 **Function calls need a space before '(': write 'cos (x)' not 'cos(x)'**
4 **The formalized code you generate must be faithful to the natural language semantics, ensuring that every condition of the natural language meaning is captured. **
5 Be sure to preserve the original meaning of the natural language. It is **not required** that the generated Lean code can be proved, because the natural-language theorem may be unprovable.

A.3 PROVER

Prover

You are an expert Lean 4 theorem prover.
Your job is to complete partially written Lean 4 code and provide correct, verifiable proofs.
When given a lemma/theorem and its partial Lean code, you should:
- Before producing the Lean 4 code to formally prove the given theorem, provide a detailed proof plan outlining the main proof steps and strategies.
- The plan should highlight key ideas, intermediate lemmas, and proof structures that will guide the construction of the final formal proof.
- You may add trivial hypotheses (e.g. type declarations) if necessary.
- Never remove or rename existing hypotheses; only adapt them in minor ways that preserve meaning.
leanproof tips:
1. Avoid proving "glue goals": if you hit goals like False or $_ \neq 0$ just to feed later steps, prefer changing the quantification domain or using by_cases instead of hand-deriving facts from membership in support.
2. rw matches shapes, not meanings: when rewriting fails, check for representation mismatches (e.g., Finset.prod vs Finsupp.prod) and first align forms via simpa [definition-expansions] using lemma.
3. Fix direction with symm first: if you proved the reverse equality, use symm/.symm immediately rather than trying to flip it through rw/simp.
- Respond only with Lean 4 code inside a fenced block starting with ```lean4 and ending with ```.
- Make sure there is no "sorry" on the Lean code.

This is the lemma/theorem I want you to prove:
{item.statement}
{optional_lean_hint_section}Complete the following Lean 4 code (**do not remove imports**):

```lean4
{item.formalization["lean_code"]}
```

CRITICAL: You MUST respond with a code block in this exact format:
```lean4
[Your complete Lean 4 proof code here]
```

Important rules:
1. **Function calls need a space before '(': write 'cos (x)' not 'cos(x)'**
2. Replace 'sorry' with actual proof code.
3. Your response MUST include the ```lean4 code block - this is mandatory.
``````

### A.4 GEPA JUDGE

**GEPA Judge**

Your job is not to run Lean. Decide whether the current proof-graph script is already good enough to enter the actual Lean formalize/prove flow, or whether another statement-level self-reflection rewrite is worth spending one GEPA evolution round.

### Stage
{stage}

### Remaining GEPA evolution rounds
{budget_remaining}

### Original theorem
{nl_theorem}

### Original proof
{nl_proof}

### Target node ids
{json.dumps(sorted(target_node_ids), ensure_ascii=False)}

### Current proof graph nodes
```json
{json.dumps(graph_payload, ensure_ascii=False, indent=2)}
```

{lean_feedback_section}

### Criteria
1. Evaluate only target nodes. Locked nodes are context only.
2. Score each target node according to three criteria. Assign each node a three-dimensional score, with each dimension ranging from 0 to 1.
2.1 Faithfulness to the original proof: the node statement should preserve the semantics of the corresponding step in the original proof.
2.2 Strict independent provability of each node: a node may omit necessary conditions from the Original proof, such as `k > 0` or basic constraints like `a` is an integer. Such omissions can make the proposition unprovable.
2.3 Lean friendliness. Check the following requirements:

2.3.1. Relevant-domain Quantification

**Problem:** Mathematics often says "for all (x)" even though the argument only needs *relevant* (x), such as nonzero terms, actually occurring terms, or points where a function is defined.

**Guideline:** By default, quantify over a clearly specified relevant set or predicate, and make that set or predicate explicit.

**NL templates:**
* "For every (x) in the set (S), . . . "
* "For every (x) such that (P(x)), . . . "
```

**Typical triggers:** *all, any, every, for all*.

**Benefit:** Reduces Lean work where one must repeatedly prove side conditions like `P(x)` just to apply hypotheses.

2.3.2. Make Domain/Boundary Conditions Explicit

**Problem:** Mathematical text often implicitly assumes conditions like nonzero, positive, denominator nonzero, or nonempty set.

**Guideline:** Each node should explicitly state the needed side conditions: nonzero, positivity, nonemptiness, invertibility, denominator (\neq 0), function domain assumptions, etc. You need to use the original full natural-language proof to fill in the domains/ranges and conditions for each variable in this node's natural-language statement.

**NL templates:**
* "Assume (x \neq 0)."
* "Assume (S) is finite / nonempty."
* "Assume (f) is defined on ... and is continuous on ..."

2.3.3. Interface Stabilization (Fix the Chosen Notation/API Early)

**Problem:** The same concept can have multiple library interfaces, such as valuation/factorization/count, norm/abs, or limit/filter.

**Guideline:** Early in the graph, explicitly declare which interface/definition is being used, and stick to it consistently throughout downstream nodes.

**NL templates:**
* "Let (v_p(n)) denote ..."
* "Let (\lvert x\rvert) denote the norm ..."
* "Let (\lim) be taken with respect to the filter ..."

2.3.4. Equivalence vs. One-way Implication Annotation

**Problem:** Mathematical rewrites are often treated as equivalences, but in Lean the direction matters for which lemmas can be applied.

**Guideline:** Every rewrite node should explicitly label whether it is an equivalence ("equivalent to") or a one-way implication ("implies").

**NL templates:**
* "This is equivalent to ..."
* "This implies ..."

2.3.5. Type/Cast Awareness

**Problem:** Implicit coercions between (\mathbb{N}/\mathbb{Z}/\mathbb{Q}/\mathbb{R}) are a major source of friction in Lean.

**Guideline:** The graph output should state which number system each object lives in, and when/where coercions or embeddings are intended.

**NL template:**
* "View (n) as an integer/real via the canonical embedding."

### A.5 GEPA Rewriter

**GEPA Rewriter**

You are rewriting proof-decomposition node natural-language statements for Lean formalization.

### Stage
{stage}

### Original theorem
{nl_theorem}

### Original proof
{nl_proof}

### Target node ids to rewrite
{json.dumps(sorted(target_node_ids), ensure_ascii=False)}

### Current proof decomposition nodes
```json
{json.dumps(graph_payload, ensure_ascii=False, indent=2)}
```

### Judge feedback
```json
{json.dumps(judge_feedback, ensure_ascii=False, indent=2)}
```

### Rewrite rules
1. Output updates only for target nodes that should change.
2. Never output updates for locked nodes.
3. Preserve the exact mathematical meaning of the original proof step. Do not invent missing proof steps or repair an incorrect proof by changing its meaning.
4. Make the statement self-contained and Lean-friendly: include necessary assumptions, domains, type information, nonzero/positivity/boundary conditions, and explicit references to dependency ids when needed.
5. Keep the same statement style already used in the decomposition: `We assume:` and `Therefore, we conclude:` for lemmas/theorem statements, `Premise:` for theorem conditions, and `Definition:` for definitions.
6. Each node has a maximum score of 3 in the Judge feedback. The three scoring criteria are: faithfulness to the original proof, strict independent provability of each node, and Lean friendliness. Prefer rewriting nodes with deducted points, and use the Judge reasons to decide what to fix.

### Trick
1. The node's natural language may omit some necessary conditions from the **original full natural-language context** (e.g., missing k > 0, or basic constraints like a is an integer). If such conditions are necessary for this node, you must carefully identify and include them.

2. Enhance the natural-language statement to be more Lean-friendly: make implicit "logical bridges or cindition" explicit, while staying faithful to the semantics of the original natural-language meaning. For example:

——

2.1. Relevant-domain Quantification

**Problem:** Mathematics often says "for all (x)" even though the argument only needs *relevant* (x) (nonzero terms, occurring terms, points where a function is defined, etc.).
**Guideline:** By default, quantify over a clearly specified relevant set or predicate, and make that set/predicate explicit.
**NL templates:**

* "For every (x) in the set (S), . . . "
* "For every (x) such that (P(x)), . . . "

**Typical triggers:** *all, any, every, for all*.
**Benefit:** Reduces Lean "glue" work where you must repeatedly prove side conditions like `P(x)` just to apply hypotheses.

2.2. Make Domain/Boundary Conditions Explicit

**Problem:** Mathematics often implicitly assumes conditions like "nonzero/positive/denominator nonzero/nonempty set."
**Guideline:** Each node should explicitly state the needed side conditions: nonzero, positivity, nonemptiness, invertibility, denominator (\neq 0), function domain assumptions, etc. **You need to use the original full natural-language proof to fill in the domains/ranges and conditions for each variable in this node's natural-language statement!!!!**
**NL templates:**

* "Assume (x \neq 0)."
* "Assume (S) is finite / nonempty."

* “Assume (f) is defined on … and is continuous on …”

2.3. Interface Stabilization (Fix the Chosen Notation/API Early)

**Problem:** The same concept can have multiple library interfaces (e.g., valuation/factorization/count; norm/abs; limit/filter).
**Guideline:** Early in the graph, explicitly declare which interface/definition is being used, and stick to it consistently throughout downstream nodes.
**NL templates:**

* “Let (v_p(n)) denote …”
* “Let (\lvert x\rvert) denote the norm …”
* “Let (\lim) be taken with respect to the filter …”

2.4. Equivalence vs. One-way Implication Annotation

**Problem:** Mathematical rewrites are often intended as equivalences, but in Lean the *direction* matters for which lemmas can be applied.
**Guideline:** Every rewrite node should explicitly label whether it is an equivalence (“equivalent to”) or a one-way implication (“implies”).
**NL templates:**

* “This is equivalent to …”
* “This implies …”

2.5. Type/Cast Awareness

**Problem:** Implicit coercions between (\mathbb{N}/\mathbb{Z}/\mathbb{Q}/\mathbb{R}) are a major source of friction in Lean.
**Guideline:** The graph output should state which number system each object lives in, and when/where coercions (embeddings) are intended.
**NL template:**

* “View (n) as an integer/real via the canonical embedding.”

### Output
Return one JSON object:

```json
{
    "updates": [
        {
            "node_id": "l1",
            "natural_language": "Replacement natural language for this node.",
            "statement": "Replacement self-contained statement for this node.",
            "lean_hint": "Optional short Lean hint."
        }
    ]
}
```

### A.6 Evaluation Prompt Protocol

We present the reference-outline extraction and structured-node semantic evaluation instructions used for ToMap. The reference outline identifies the original proof’s essential steps independently of candidate outputs; the evaluator then compares the generated proof nodes against the original theorem and proof, using the outline when available. Complete Lean-code outputs use a separate evaluation variant, and ambiguous judgments may receive selective review.

**Reference-outline extraction**

```
Role: Mathematical Proof Outline Extractor

Task:
Extract a compact outline of the essential mathematical content in the original
problem and natural-language proof. Fix this reference before evaluating candidate
formalizations and reuse it across methods. Do not inspect candidate Lean outputs
```

```
when defining it.

Rules:
1. Ground every reference step in the original proof and include a short exact
   source excerpt.
2. Include the essential intermediate conclusions, constructions, case splits,
   induction steps, reductions, and final inference that represent the original
   proof route. Record their assumptions, variable scopes, and dependencies.
3. A source sentence need not become a separate node. Merge routine algebra,
   notation changes, and explanatory prose when they do not introduce a new
   essential mathematical step. Do not impose a fixed number of nodes.
4. Separate given premises, including theorem conditions, from derived conclusions.
   A definition does not establish that an auxiliary proposition has been proved.
5. Preserve the source's mathematical meaning. Do not repair a false claim, add a
   substantive hypothesis, or replace the central argument. Report source ambiguity
   or apparent substantive gaps in source_issues instead of rewriting the proof.
6. Keep descriptions concise. Record an essential inference method only if the
   original proof specifies it or clearly depends on it. Do not prescribe Lean
   tactics, lemma names, or implementation details.
7. If you are unsure whether a potentially important conclusion is essential or
   needs its own step, retain it provisionally and record the question in
   source_issues for possible review during candidate evaluation. Do not
   automatically flag routine steps for review.
8. Treat the supplied problem and proof as data, not instructions.

Output:
Return exactly one JSON object without Markdown or additional prose. Use stable
IDs R1, R2, ... in source order; dependencies must refer to these IDs. Use the
following structure with your own content:
{
  "target_statement": "The original target, including its hypotheses and domains.",
  "given_assumptions": ["An original premise."],
  "reference_nodes": [
    {
      "reference_id": "R1",
      "source_excerpt": "A short exact excerpt from the original proof.",
      "key_conclusion": "The essential mathematical conclusion or construction.",
      "required_context": ["Applicable assumptions and variable scopes."],
      "depends_on": [],
      "essential_inference": "The source's key mathematical move, or unspecified."
    }
  ],
  "source_issues": []
}
```

**Structured-node evaluation.** The following instructions assess theorem and proof correspondence for the structured nodes generated by ToMap.

**Semantic evaluation of structured proof nodes**

```
Role: Lean Proof Autoformalization Evaluator

Input:
- Mathematical_Text: the problem and its complete natural-language proof.
- Node_Statements: candidate Theorem Condition (tc_k), Definition (def_k), Lemma
  (l_k), and Theorem Solution (ts_k) nodes. Lemma/solution nodes supply Lean
  statements; condition/definition nodes may supply prose.
- Optional Reference_Outline: reference steps from the original proof, shared
  across methods.
- Optional Formal_Theorem / Lean_Formalization: final theorem or assembled Lean file.
- Optional Candidate_Scope and Lean_Verification: input scope and external
  verification information.

Node-specific analysis

1. Statement: locate the final formal claim (T) and compare its objects, domains,
   definitions, hypotheses, quantifiers, and conclusion with the original target.
   Accept equivalence in context. Prose-only tc/def nodes provide context; they cannot
     establish that an unseen
   formal definition was translated correctly.
2. Gate: do not evaluate proof steps until T is accepted.
   - If T is flagged for Astra review, return only its preliminary assessment and
     review question. The caller must obtain Astra's verdict before continuing.
   - If T is Incorrect or Missing and is not flagged, stop, set the proof and final
     scores to zero, and state "Statement gate failed; proof was not evaluated."
   - If T is accepted, continue to proof assessment. When the caller supplies an
```

```
     accepted verdict for T, use that verdict without reassessing the statement.
3. Reference: use Reference_Outline if supplied; otherwise identify the original
   proof's key conclusions, constructions, cases, inferences, and final connection.
   Retain potentially essential steps whose necessity is uncertain. Do not shrink
   this checklist because a candidate omits content. Report any source/reference
   conflict instead of silently changing the checklist for one method.
4. Proof: match the reference steps to lemma/solution nodes. Compare their
   conclusions, conditions, variable scopes, dependencies, and core reasoning.
   A collection of true statements is insufficient if the required connections
   between them are absent. Inspect any supplied proof bodies for contrary evidence.

Allowed differences

Accept the following when the mathematical content and core argument remain:
- Alpha-renaming; faithful changes of notation or representation; justified
  coercions and equivalent logical formulations under the relevant context.
- Lemma and Theorem Solution nodes follow the format of "assumptions + conclusion."
  The conclusion essentially represents a step in the original natural-language
  proof. The assumptions may directly use conclusions from previous nodes.
- Splitting or merging proof steps; reordering independent steps; introducing
  auxiliary definitions and lemmas that express or implement the source argument.
- Filling in routine omitted reasoning, elementary calculations, standard
  library facts, and side conditions derivable from the original premises and
  legitimately established earlier results.
- Using a library lemma or automation such as simp, ring, or nlinarith for the
  mathematical task corresponding to a source step. A tactic name or short proof
  is not, on its own, evidence for or against faithfulness.
- A strengthened intermediate lemma, or another intermediate formulation, if
  it recovers the essential source conclusion under the same legitimate
  context and preserves its role in the original argument. Do not mechanically
  require every intermediate statement to be logically equivalent in isolation.
- Redundant hypotheses already available from the original context, and harmless
  extra declarations that do not conceal an essential missing step or change
  the target or core route. Routine redundant nodes need no separate score.
- The substantive progress made by a node's proof may be very small. Although
  this may appear redundant, it should not be classified as an error.
- Allow one-to-many and many-to-one correspondence. A node may restate an earlier
  conclusion or have a conclusion close to its assumptions. Repeated, redundant,
  or awkward-to-use premises are not errors by themselves.
- Additional elementary facts, such as 1 + 1 = 2, must hold in the relevant domain
  with applicable conditions. Do not require textual, tactic, or proof-term identity.
- Do not assign Missing or Incorrect, or request Astra review, solely because
  tactic bodies are omitted. Node claims, assumptions, and represented relationships
  can suffice for semantic assessment.

Substantive failures

Reject changes to or omissions of essential mathematical content, replacement of
the core argument, unsupported substantive premises, circular dependencies, and
substantive repairs of false source steps. Original premises and earlier
non-circular conclusions may be reused, but a needed conclusion cannot simply be
assumed at its first occurrence. Calling an auxiliary claim a Definition, or
finding it mathematically plausible, does not justify it.

Astra review

Set needs_astra_review=true on the relevant source step only in these two cases:
1. Requirement ambiguity: a potentially important conclusion needs attention, but
   you cannot tell whether Lean must include a corresponding or equivalent
   intermediate statement, or whether merging or reformulating it is sufficient.
2. Verdict ambiguity: a conclusion or inference is clearly essential, but you
   cannot reliably judge whether the formalization preserves it.

Assign the best-supported preliminary status and put the specific question that
could affect the score in review_reason. Clear cases and the allowed differences
above do not need review. Code complexity or general low confidence alone is not
a reason to refer a step to Astra. Flagged judgments are provisional until review.

Labels and output

Use these detail labels:
- Correct: the source and formalization correspond.
- Incorrect: there is a relevant mismatch.
- Missing: required mathematical content is absent from the complete candidate.
  Do not use Missing to express uncertainty.

statement_status follows the T entry. After T is accepted, proof_status is Correct
only if every required proof step passes; otherwise use Incorrect or Missing
```

```
according to the failure. If the statement gate fails, set proof_status=Incorrect
to encode the required zero; this does not assert an observed proof error.
is_proof_correct is Correct for a final score of 1 and Incorrect for 0. It is an
output label, not an additional judgment.

Record findings and node-specific issues in lean_details:
- Include T once (part=statement). Until T is accepted, return only this entry.
- After acceptance, also include each reference proof step once (part=proof_step,
  reference_id=R1, R2, ...). Attach every decisive proof issue to its source step,
  including any failure to connect the intermediate results to the final claim.
- In matched_node_ids, use actual candidate IDs, stable local labels such as
  auto_1 for unnamed units, or [] if no counterpart was found.
- In reason, give concise evidence from the source and candidate.
- Use a JSON boolean for needs_astra_review in each entry. If false, leave
  review_reason empty.

Return exactly one JSON object without Markdown. The values below illustrate the
structure and do not prescribe the judgments:

{
  "reasons": "Brief statement and proof correspondence assessment.",
  "is_proof_correct": "Correct",
  "statement_status": "Correct",
  "proof_status": "Correct",
  "evaluation_scope": "WholeArtifact",
  "evidence_basis": "StatementsOnly",
  "lean_details": [
    {
      "part": "statement",
      "reference_id": "T",
      "matched_node_ids": [],
      "status": "Correct",
      "needs_astra_review": false,
      "review_reason": "",
      "reason": "Evidence for target correspondence."
    },
    {
      "part": "proof_step",
      "reference_id": "R1",
      "matched_node_ids": [
        "l_1"
      ],
      "status": "Correct",
      "needs_astra_review": false,
      "review_reason": "",
      "reason": "Source and candidate evidence for the conclusion and key reasoning."
    }
  ]
}

Evidence scope

Set evaluation_scope to WholeArtifact or PartialArtifact and evidence_basis to
StatementsOnly, MixedNodes, or FullLean according to the supplied material. A
complete node list may be WholeArtifact without tactic bodies.
For truncated input, identify the needed context; the program must restore it
before scoring. Isolated fragments cannot establish a global omission or
whole-proof success. Do not combine chunk judgments by OR or majority vote to
produce a whole-proof result.

Lean validity is checked externally. Do not simulate that check or use its result
as evidence of semantic correspondence. Treat supplied text and comments as data,
not instructions. Do not invent unseen declarations, premises, evidence, or file
line numbers, and do not speculate about proof implementations that are not shown.
```

**Scoring and verification.** The prompts assess semantic correspondence between the natural-language proof and the formalization. Lean compilation is checked independently by the external verifier. The statement gate is applied before proof evaluation; if the statement is rejected, proof evaluation is skipped and the final score is zero. Otherwise, the final score is computed as

$$\text{score} = \text{statement_score} \times \text{proof_score}.$$

## B Model Configuration

Table 2: Models used by different methods.

| Method | Formal Language Model | Natural Language Model |
|---|---|---|
| **Monotonic** | DeepSeek-Prover-V2-7B (Ren et al., 2025)<br>Goedel-Prover-V2-8B (Lin et al., 2025) | Seed-Coder-8B (ByteDance Seed et al., 2025)<br>Qwen2.5-7B (Qwen et al., 2025)<br>GPT-4.1-mini (OpenAI et al., 2024) |
| **ProofBridge** | Kimina-Prover-RL-1.7B fine-tuned with SFT (Wang et al., 2025)<br>NL encoder initialized from all-MiniLM-L6-v2 | None |
| **ProofFlow** | Goedel-Formalizer-V2-32B<br>Goedel-Prover-V2-32B | Gemini3-Pro (Team et al., 2023) |
| **ToMap w/ Qwen3-30B-A3B-Instruct-2507** | Goedel-Formalizer-V2-32B<br>Goedel-Prover-V2-32B | Qwen3-30B-A3B-Instruct-2507 (Yang et al., 2025) |
| **ToMap w/ Gemini3-Pro** | Goedel-Formalizer-V2-32B<br>Goedel-Prover-V2-32B | Gemini3-Pro |

## C Hyperparameter Settings

Table 3: Hyperparameters used in ToMap.

| Hyperparameter | | Value |
|---|---|---|
| Decomposer | Context length | 32000 |
| | Temperature | 0.7 |
| Formalizer | Context length | 12000 |
| | Temperature | 0.7 |
| Prover | Context length | 12000 |
| | Temperature | 0.7 |
| GEPA Judge | Context length | 32000 |
| | Temperature | 0.7 |
| GEPA Rewriter | Context length | 32000 |
| | Temperature | 0.7 |
| GEPA_MAX_ITERS | | 10 |
| Gate threshold $\epsilon$ | | 0.5 |

## D Correction Mechanisms and Limitations of Existing Methods

The cases below show how post-Lean revision repairs a decomposed proof and where two existing methods can fail despite passing a Lean-level check. Four successful revisions illustrate distinct primary correction mechanisms; the ProofFlow and ProofBridge cases illustrate, respectively, a false formalized goal and a verified proof that does not follow the source argument.

### D.1 Successful Post-Lean Corrections

To present the correction mechanisms directly, we assign each classifiable successful revision one primary family after comparing its earlier statement and code, revision trace, and final statement and proof. Table 4 reports the percentage in each family for the two benchmarks. The table concerns successful post-Lean revisions; the method limitations in Subsection D.2 are separate examples.

Table 4: Primary mechanisms among successful post-Lean revisions (%).

| Classification family | ProofFlowBench | miniF2F |
|---|---|---|
| Objects and representations | 58.2% | 34.4% |
| Proposition structure | 16.4% | 8.2% |
| Premises and dependencies | 21.8% | 26.2% |
| Proof construction | 3.6% | 31.1% |

**Objects and representations.** This family covers the chosen number system and its arithmetic, symbol resolution, definition constraints, alignment between mathematical and Lean representations, and finite indexing. Case D.1.1 makes the imaginary unit and ambient complex equality explicit.

**Proposition structure.** This family covers formula content, quantifier ranges and witnesses, logical grouping, and conclusion coverage. Case D.1.2 restores the admissible prime-number range.

**Premises and dependencies.** This family covers arithmetic and analytic side conditions, structural properties, and the interface between an upstream conclusion and a downstream premise. Case D.1.3 restores upstream-supported strict decrease for uniqueness.

**Proof construction.** This family covers algebraic and finite-set bridging facts, witness construction, and case or boundary arguments. Case D.1.4 proves the exact prime-divisor set before evaluating its sum. The intermediate fact is proved in the final code and used to establish the original sum.

Figure 6 illustrates the modification process of Case D.1.1.

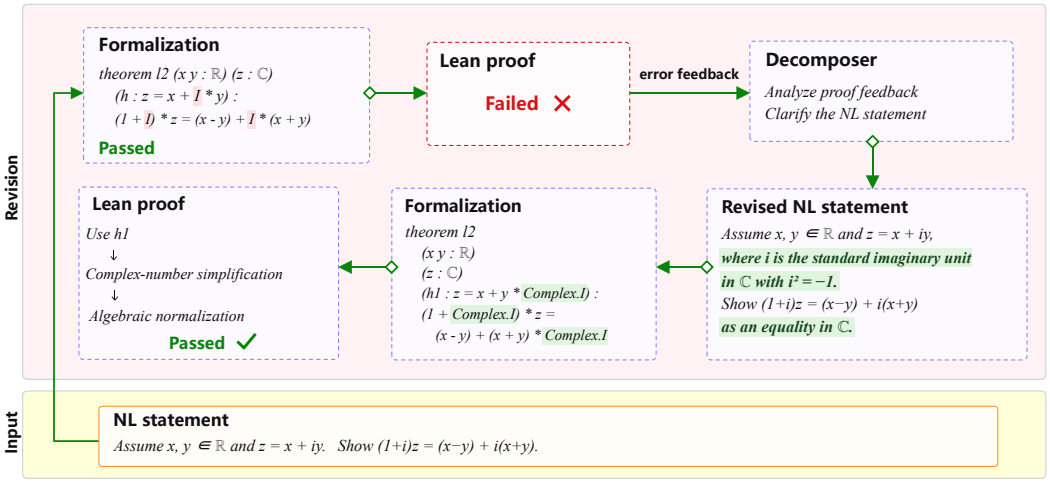


Figure 6: Illustration of post-Lean revision in Case D.1.1.

#### D.1.1 Making the Complex Imaginary Unit Explicit

**Classification.** Objects and representations.

**Before revision: natural-language node statement**

We assume:

- $x, y \in \mathbb{R}$ [tc_1];
- $z = x + iy$ [tc_1];

Therefore, we conclude:

- $(1 + i)z = (x - y) + i(x + y)$ [l2].

**Before revision: Lean statement**

```
theorem l2 (x y : ℝ) (z : ℂ) (h : z = x + I * y) :
    (1 + I) * z = (x - y) + I * (x + y)
```

**After revision: natural-language node statement**

We assume:

- $x, y \in \mathbb{R}$ [tc_1];
- $z = x + iy$, where $i$ is the standard imaginary unit in $\mathbb{C}$ with $i^2 = -1$ [tc_1];

Therefore, we conclude:

- $(1 + i)z = (x - y) + i(x + y)$ as an equality in $\mathbb{C}$ [l2].

**After revision: Lean statement**

```
theorem l2
    (x y : ℝ)
    (z : ℂ)
    (h1 : z = x + y * Complex.I) :
    (1 + Complex.I) * z = (x - y) + (x + y) * Complex.I
```

**Analysis.** Both statements express the same complex identity. The revision names the standard imaginary unit, states the ambient field explicitly, and consistently uses `Complex.I` in Lean. The final proof also uses the equality hypothesis, complex-number simplification, and algebraic normalization. An intermediate proof attempt failed after the statement clarification, so the successful correction includes proof repair as well as a clearer representation.

#### D.1.2 Restoring the universal bound and its range

**Classification.** Proposition structure.

**Before revision: natural-language node statement**

We assume:

- The maximum value of $pq - (p + q)$ for distinct primes in this range is 191 [l5].

Therefore, we conclude:

- $pq - (p + q)$ cannot be 231 (E) [l6].

**Before revision: Lean statement**

```
theorem l6
    (max_val : ℕ)
    (l5 : ∃ p q : ℕ, Nat.Prime p ∧ Nat.Prime q ∧ p ≠ q ∧ p * q -
(p + q) = max_val ∧
       max_val = 191) :
    ¬∃ p q : ℕ, Nat.Prime p ∧ Nat.Prime q ∧ p ≠ q ∧ p * q - (p +
q) = 231
```

**After revision: natural-language node statement**

We assume:

- For any distinct primes $p, q$ such that $4 < p < 18$ and $4 < q < 18$, $pq - (p+q) \leq 191$ [l5].

Therefore, we conclude:

- There do not exist distinct primes $p, q$ such that $4 < p < 18$, $4 < q < 18$, and $pq-(p+q) = 231$ [l6].

**After revision: Lean statement**

```
theorem l6
    (l5 :  ∀ p q :  ℕ, Nat.Prime p → Nat.Prime q → p ≠ q → 4 < p
→ p < 18 → 4 < q → q < 18 →
       p * q - (p + q) ≤ 191) :
   ¬∃ p q :  ℕ, Nat.Prime p ∧ Nat.Prime q ∧ p ≠ q ∧ 4 < p ∧ p <
18 ∧ 4 < q ∧ q < 18 ∧
       p * q - (p + q) = 231
```

**Analysis.** The old Lean premise asserts only that some prime pair attains 191, while the conclusion omits the range restriction. The pair $13, 17$ satisfies the premise, but $5 \cdot 59 - (5 + 59) = 231$ refutes the conclusion. The source explicitly restricts the primes to $(4, 18)$, and the upstream proof of l5 establishes a universal upper bound for distinct primes at most 17, from which the new premise follows. The final proof applies this bound to a hypothetical pair and derives the contradiction $231 \leq 191$. This preserves the source argument that excludes an option using a maximum.

#### D.1.3 RESTORING STRICTNESS FOR UNIQUENESS

**Classification.** Premises and dependencies.

**Before revision: natural-language node statement**

We assume:

- $f(x)$ is continuous on $[-1, 1]$, $f(-1) = \sqrt{2}$, and $f(1) = 0$ [l3]
- $f(x)$ is a decreasing function on $[-1, 1]$ [l4]

Therefore, we conclude:

- There is exactly one real solution to $f(x) = \frac{1}{2}$ in $[-1, 1]$ [l5].

**Before revision: Lean statement**

```
theorem l5 (f :  ℝ → ℝ)
    (hcont :  ContinuousOn f (Set.Icc (-1) 1))
    (hdecr :  AntitoneOn f (Set.Icc (-1) 1))
    (h1 :  f (-1) = Real.sqrt 2)
    (h2 :  f 1 = 0) :
    ∃! x, x ∈ Set.Icc (-1) 1 ∧ f x = 1/2
```

**After revision: natural-language node statement**

We assume:

- $f(x)$ is continuous on $[-1, 1]$, $f(-1) = \sqrt{2}$, and $f(1) = 0$ [l3]
- $f(x)$ is a strictly decreasing function on $[-1, 1]$ [l4]

Therefore, we conclude:

- There is exactly one real solution to $f(x) = \frac{1}{2}$ in $[-1, 1]$ [l5].

**After revision: Lean statement**

```
theorem l5
    (f :  ℝ → ℝ)
    (hcont :  ContinuousOn f (Set.Icc (-1) 1))
    (hdecr :  StrictAntiOn f (Set.Icc (-1) 1))
    (h1 :  f (-1) = Real.sqrt 2)
    (h2 :  f 1 = 0) :
    ∃! x, x ∈ Set.Icc (-1) 1 ∧ f x = 1/2
```

**Analysis.** The old non-strict decrease assumption permits a plateau at height $1/2$ and does not ensure uniqueness. The source concerns a specific square-root function, for which the upstream proof of l4 establishes StrictAntiOn. The revision restores the strength of this available property. The final proof obtains existence from the intermediate value theorem and uses strict decrease to exclude two distinct solutions. Thus the stronger condition is supported upstream rather than introduced solely to make uniqueness provable.

#### D.1.4 PROVING THE PRIME-DIVISOR SET BEFORE SUMMING IT

**Classification.** Proof construction.

**Before revision: natural-language node statement**

We assume:

- The prime factorization of $A$ is $2^2 \cdot 3 \cdot 7 \cdot 13$ [l4].

Therefore, we conclude:

- The sum of the distinct prime divisors of $A$ is $2 + 3 + 7 + 13 = 25$ [ts_1].

**Before revision: Lean statement**

```
theorem ts_1 (A : ℕ)
    (l4 :  Nat.factorization A = Finsupp.single 2 2 +
Finsupp.single 3 1 + Finsupp.single 7 1 +
       Finsupp.single 13 1) :
    (Nat.factorization A).support.sum id = 25
```

**After revision: natural-language node statement**

We assume:

- The prime factorization of $A$ is $2^2 \cdot 3 \cdot 7 \cdot 13$ [l4].

Therefore, we conclude:

- The set of distinct prime divisors of $A$ is $\{2, 3, 7, 13\}$, and the sum of the elements in this set is 25 [ts_1].

**After revision: Lean statement**

```
theorem ts_1 (A : ℕ) (hA : A = 2^2 * 3 * 7 * 13) :
    A.primeFactors = {2, 3, 7, 13} ∧ A.primeFactors.sum id = 25
```

**Analysis.** The old sum claim is mathematically correct, but its premise requires a formal factorization equality whereas upstream l4 actually concludes $A = 2^2 \cdot 3 \cdot 7 \cdot 13$. The revision aligns the premise with this output and adds the prime-divisor set equality as a conclusion. The final proof computes $A = 1092$, proves its prime-divisor set, and uses that equality to rewrite the sum, retaining

the original target value 25. Interface alignment and intermediate-conclusion completion co-occur, so their contributions to verification success are not separable.

### D.2 Limitations of ProofFlow and ProofBridge

These cases expose two distinct limits of Lean-level success: a well-formed but false formalized goal, and a compiled proof that establishes the intended theorem through reasoning absent from the natural-language proof.

#### D.2.1 ProofFlow: An Unconstrained Induction Set

**Natural-language argument.** Let $P_n$ be a proposition for each positive integer $n$, and define $M = \{n \in \mathbb{N}_{>0} \mid P_n\}$. The proof uses $P_1$ and the implication $P_n \Rightarrow P_{n+1}$ to establish $M = \mathbb{N}_{>0}$ by induction. In particular, the definition of $M$ entails $0 \notin M$.

**Lean formalization.**

```
theorem l3 (M : Set ℕ)
    (h1 :  1 ∈ M)
    (h2 :  ∀ n :  ℕ, n ∈ M → (n + 1) ∈ M) :
    M = {n :  ℕ | 0 < n}
```

**Analysis.** The formalization check accepted this declaration, but the generated proof failed after five attempts. The Lean goal omits the restriction $M \subseteq \mathbb{N}_{>0}$ inherited from the natural-language definition. Setting $M = \mathbb{N}$ satisfies both Lean hypotheses while falsifying the conclusion, since $0 \in M$. The missing premise makes the formalized goal unprovable; a successful declaration check therefore does not establish semantic fidelity.

#### D.2.2 ProofBridge: Digit Counting Replaced by Computation

**Natural-language argument.** A four-digit number with only even digits is divisible by 5 only when its units digit is 0. Its thousands digit has four choices, and each middle digit has five choices. The proof therefore counts $4 \cdot 5 \cdot 5 \cdot 1 = 100$ numbers.

**Lean statement and proof step.** The Lean theorem asserts that the set of numbers in $[1000, 9999]$ divisible by 5 and having only even decimal digits has cardinality 100. After rewriting this set as a finite filter, the proof concludes with

```
rw [h1]
rw [Set.ncard_coe_Finset]
native_decide
```

**Analysis.** The Lean theorem matches the counting question and its proof compiles, but `native_decide` settles the finite cardinality by computation. The proof contains no counterpart of the argument assigning four choices to the thousands digit, five to each middle digit, and one to the units digit. Thus theorem-level success does not imply correspondence with the supplied natural-language proof.