RAF: A REPAIR-AUGMENTED FRAMEWORK FOR LLM-GENERATED LEAN PROOFS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly used to generate complete formal proofs that can be verified by proof assistants such as Lean. However, successful verification requires knowledge of the target formal environment, and errors in generated proofs may exceed the capabilities of any single repair strategy. We present RAF, a training-free pipeline that combines external formal knowledge with complementary repair strategies for whole-proof generation in Lean 4. By retrieving relevant library declarations before generation, RAF provides explicit knowledge of available theorems and their signatures. After generation, RAF combines repair strategies with different repair boundaries to address complementary failure cases. We introduce lightweight sorrification, a diagnostic-guided strategy that localizes failing proof regions while preserving surrounding proof content, providing an alternative to broader repair strategies. The resulting proof obligations are addressed through local tactic repair, and only proofs verified by Lean without unresolved placeholders are counted as complete. Experiments across five LLMs on miniF2F show that RAF improves proof completion by at least 5% over the baseline repair pipeline, supporting the value of complementary repair within a knowledge-augmented proof generation pipeline.

1 INTRODUCTION

Mathematical proofs provide the foundation for establishing the correctness of mathematical claims. As mathematical arguments become longer and more complex, constructing and checking them manually can require substantial expert effort. This difficulty is particularly apparent in large formalization projects and computer-assisted proofs, where thousands of definitions, intermediate lemmas, and logical dependencies must be managed consistently (Hales, 2006). Machine-checked proofs can provide stronger assurance by expressing every inference in a formal language and verifying it with a small trusted kernel. The same principle has also enabled the verification of complex software systems, including the CompCert compiler and the seL4 microkernel (Leroy, 2024; Klein et al., 2009).

Automated theorem proving (ATP) aims to reduce the human effort required to construct such formal proofs. Early ATP systems treated proof construction primarily as a symbolic search problem (Gelernter, 1995; Gilmore, 1960). Modern systems combine automated reasoning with interactive theorem provers, in which a proof assistant checks every generated proof term against a precise logical foundation. More recently, large language models (LLMs) have been used to predict individual tactics, guide proof search, or generate complete proof scripts (Polu & Sutskever, 2020; First et al., 2023). These methods benefit from the ability of LLMs to produce plausible mathematical arguments, while the proof assistant provides a definitive test of whether the generated argument is formally correct.

Lean 4 is both a programming language and an interactive theorem prover based on dependent type theory (Moura & Ullrich, 2021). Its trusted kernel verifies proof terms, while its tactic language provides automation for constructing them. Lean is supported by Mathlib (mathlib Community, 2020), a large community-maintained library containing reusable definitions, theorems, and tactics from many areas of mathematics. Existing theorem-proving systems exploit this environment in different ways. Some retrieve potentially useful premises from the library, some explore intermediate proof states, and others generate an entire proof before submitting it to Lean for verification (Yang et al.,

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.

2023; Lample et al., 2022; Kripner et al., 2025). The size of the Lean ecosystem creates an important opportunity for automated reasoning, but it also introduces a practical difficulty: a successful proof must identify the appropriate declarations and use their exact names, namespaces, types, and arguments.

Although general-purpose LLMs can generate convincing mathematical explanations, they do not reliably satisfy the strict syntactic, type-theoretic, and library-specific requirements of Lean. A model may refer to a nonexistent theorem, use a valid theorem with incompatible arguments, omit a required namespace, or produce proof code that expresses the intended mathematical idea but fails to elaborate. These problems can be reduced by training models on large collections of formal proofs. However, constructing suitable datasets and training specialized theorem-proving models require substantial computation and engineering effort. The resulting models may also remain dependent on the particular proof-assistant version and library environment used during training. Training-free methods are therefore attractive when adapting an existing language model to a formal environment.

Existing approaches to recovering failed proofs generally follow two directions. One approach asks the LLM to regenerate or refine its proof using feedback from the verifier. This approach is flexible, but it still relies on the model to discover which library theorem is relevant and determine how that theorem should be applied. When the model has incomplete knowledge of Lean or Mathlib, repeated generation may reproduce the same mistake or introduce new ones. A second approach applies offline repair after proof generation, using compiler diagnostics, source transformations, placeholders, or tactic automation to correct the candidate (First et al., 2023; Ospanov et al., 2025). Offline repair does not require additional model training and can be executed quickly, but an individual repair algorithm is usually designed for a limited class of failures. A syntax transformation, localized source repair, or tactic portfolio may solve one type of error while overlooking failures that require missing library knowledge or a different proof strategy.

The key observation behind this work is that these methods have complementary strengths. A proof that cannot be recovered by one strategy may still be completed by another strategy that targets a different failure mode. RAF is a general, training-free framework for composing whole-proof generation, optional formal-context retrieval, Lean verification, and repair strategies. It does not prescribe a single fixed sequence of components. Instead, a RAF instantiation selects and orders strategies according to the failure modes and computational budget of a particular setting. The instantiation evaluated in this paper combines one-shot direct generation, Apollo repair, and a diagnostic-guided Lightweight Sorrifier. The latter replaces only Lean-reported failing regions with explicit obligations and attempts to discharge them while preserving surrounding proof content.

We evaluate this RAF instantiation on the miniF2F benchmark (Zheng et al., 2021) using five LLMs. Within each model, the evaluation compares one-shot direct generation, the deterministic Apollo repair pipeline, and the full evaluated RAF instantiation on the same theorem statements and generated candidates. The Lightweight Sorrifier contributes between 0 and 10% additional complete proofs beyond Apollo, depending on the model. Additional experiments on external formal knowledge and DSP-based initial generation are reported separately in Appendix A so that they are not conflated with the repair-composition comparison.

This work makes the following contributions:

- We identify incomplete formal knowledge and limited repair coverage as two complementary sources of failure in LLM-generated Lean proofs, and motivate the composition of strategies that address different failure modes.
- We introduce RAF, a general training-free framework in which whole- proof generation, optional retrieval, verification, and repair strategies can be composed without retraining the underlying language model.
- We develop a modular architecture in which multiple generation and repair strategies can operate under a shared Lean verifier, allowing the failure cases of one method to be recovered by another.
- We provide a controlled evaluation on miniF2F across five LLMs. The evaluated RAF instantiation matches or improves upon Apollo for every model, with the Lightweight Sorrifier adding up to 10% complete proofs beyond the deterministic Apollo result.

2 Background and Related Work

2.1 Formal Proof Synthesis in Lean 4

Lean 4 is both a programming language and an interactive theorem prover based on dependent type theory (Moura & Ullrich, 2021). Under the Curry–Howard view, a theorem is a type and a proof is a term inhabiting that type. Tactics construct such terms by transforming a goal into simpler subgoals until no obligations remain. Lean's elaborator and kernel therefore provide a deterministic boundary for LLM-based systems: a generated script is useful as a complete proof only when it is accepted and contains no unresolved placeholders.

Lean permits the token `sorry` as an explicit placeholder. A source file containing `sorry` can elaborate successfully, which is valuable during interactive development, but it does not constitute a completed proof for our evaluation. We consequently distinguish *accepted* candidates, which pass Lean's checks even if placeholders remain, from *complete* candidates, which pass Lean and contain no reported sorries. This distinction is central to evaluating repair: a method can increase compilability merely by deleting proof content without solving the theorem.

2.2 Search-Based Theorem Proving

Many neural theorem provers cast proof construction as guided search. Each node is a proof state, each tactic is an action, and a proof is a path that closes all subgoals. GPT-f learns tactic generation for this setting (Polu & Sutskever, 2020); LeanDojo and ReProver combine retrieval with best-first exploration in Lean (Yang et al., 2023); HyperTree Proof Search represents tactics with multiple subgoals as hyperedges (Lample et al., 2022); and LeanTree factorizes proof states so that independent components can be processed and verified separately (Kripner et al., 2025). Search provides frequent formal feedback, but repeated model calls and state expansion can be computationally expensive.

2.3 Whole-Proof Generation and Repair

Whole-proof methods instead ask a model to synthesize an entire proof before verification. This provides a global plan and amortizes model interaction, but a locally incorrect step can invalidate the entire output. Baldur First et al. (2023) demonstrates whole-proof generation and a separate error-conditioned repair model for Isabelle/HOL (Nipkow et al., 2002). Draft, Sketch, and Prove (DSP) decomposes a proof into an informal proof draft, a formal sketch with proof obligations, and automated discharge of those obligations (Jiang et al., 2022). Apollo brings compiler-guided generation and repair to Lean 4: it normalizes generated source, sorrifies failing regions, invokes automated tactics, and can recursively attempt remaining obligations (Ospanov et al., 2025). RAF belongs to this family but provides a more general composition layer: a RAF instantiation may combine optional formal-context retrieval with one or more verifier-guided repair strategies that use different repair boundaries.

2.4 Positioning of This Work

RAF is a general training-free framework for assembling whole-proof generation and repair workflows under a shared Lean verifier. It neither trains a tactic policy nor introduces a new proof-search algorithm. Its closest repair comparison is Apollo. The evaluated instantiation uses Apollo followed by a diagnostic-guided Lightweight Sorrifier, while RAF itself can accommodate other strategy portfolios and execution orders.

3 Problem Setting

We study training-free whole-proof generation for Lean 4. Given a theorem, the system asks a fixed language model to generate a complete proof body, checks the resulting source with Lean, and may apply one or more repair procedures when the initial candidate is not complete. RAF operates on complete Lean programs: each strategy generates or modifies a proof script, and Lean returns its

verification result and diagnostics. Unlike tactic-search systems, RAF does not require the language model to interact with Lean one proof state at a time.

3.1 Proof Task and Whole-Proof Generation

A proof task is represented as

$$T = (\Gamma_0, G_0), \tag{1}$$

where Γ_0 contains the imports, declarations, variables, and hypotheses available at the target theorem, and G_0 is its proposition. When external knowledge is enabled, a retriever may additionally provide a finite set of Lean declaration headers

$$K = R(T, \mathcal{L}), \tag{2}$$

drawn from the available library $\mathcal{L}$; otherwise $K = \emptyset$.

Given T and K, a fixed language model G_θ produces one complete candidate proof body

$$P_0 = G_\theta(T, K). \tag{3}$$

The candidate is combined with the original imports, preceding declarations, and theorem header before verification. This is *whole-proof generation*: the model produces the candidate before receiving Lean feedback about any of its intermediate commands.

3.2 Lean Verification and Outcomes

For a reconstructed candidate P, the verifier returns

$$V(P) = \big(A(P), C(P), E(P), S(P)\big), \tag{4}$$

where $A(P)$ records whether Lean accepts and elaborates the source, $C(P)$ records whether it is a complete proof, $E(P)$ contains Lean diagnostics and their source locations, and $S(P)$ is the set of unresolved `sorry` placeholders locations.

Acceptance and completeness are intentionally distinct. Lean can elaborate a source file containing `sorry`, but such a file does not constitute a finished proof. We therefore require

$$C(P) = 1 \quad \Longleftrightarrow \quad A(P) = 1 \ \wedge \ S(P) = \emptyset. \tag{5}$$

This gives the four outcome categories used in our experiments: `COMPLETE` denotes $C(P) = 1$; `ACCEPTED` denotes $A(P) = 1$ but $C(P) = 0$; `REJECTED` denotes source that Lean does not accept; and `ERROR` denotes a pipeline failure for which no valid Lean judgment is obtained. Only `COMPLETE` counts as solving the theorem.

Verifying P_0 immediately, without post-processing or repair, gives the *Direct* result. This result isolates the proof-generating ability of the model, with or without the optional retrieved context K.

3.3 Post-Generation Repair Objective

When $C(P_0) = 0$, a repair strategy may use the theorem, optional retrieved context, candidate source, and verifier feedback to propose another candidate:

$$P_i = r_i\big(T, K, P_0, V(P_0)\big). \tag{6}$$

The function r_i may implement coarse or localized source replacement, tactic-based gap filling, regeneration, or a composition of these operations. The formulation does not assume that every strategy edits the output of the preceding strategy: a later strategy may instead restart from P_0 and use a different interpretation of the same diagnostics.

Given an ordered portfolio $\mathcal{R} = (r_1, \ldots, r_m)$, the system succeeds on T if the direct candidate or at least one repair candidate is complete:

$$C(P_0) = 1 \quad \vee \quad \exists r_i \in \mathcal{R}\ C(P_i) = 1. \tag{7}$$

For a benchmark $\mathcal{D}$, our primary objective is therefore the end-to-end completion rate

$$\text{CompletionRate} = \frac{1}{|\mathcal{D}|} \sum_{T \in \mathcal{D}} \mathbf{1}[C(P_T^\star) = 1]\,, \tag{8}$$

where $P_T^\star$ is the candidate returned by the evaluated configuration. Chapter 4 specifies the retrieval and repair strategies used by RAF; the present section defines only their shared inputs and success criterion.

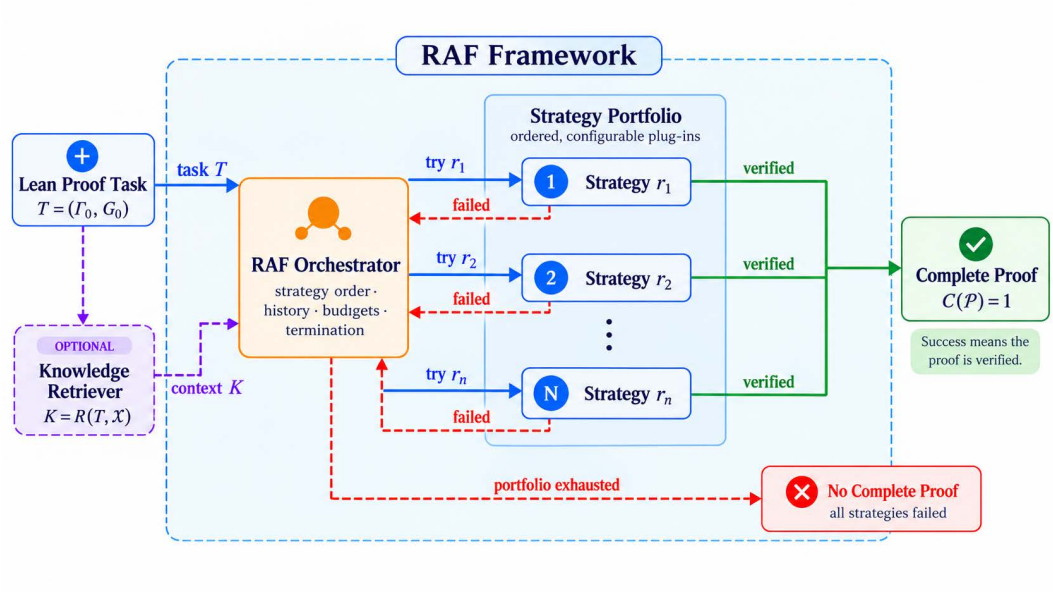

Figure 1: Overview of the RAF framework. An optional retriever supplies formal context to the orchestrator, which executes an ordered portfolio of proof strategies. Each candidate is checked by Lean; RAF returns the first complete verified proof, or reports failure after the portfolio is exhausted.

4 The RAF Framework

RAF is a training-free framework for composing heterogeneous Lean proof strategies behind a shared verification interface. It does not prescribe a particular generator, repair algorithm, portfolio size, or execution order. Instead, a configuration specifies which strategies are available and how an unsuccessful result falls back to another strategy. Direct generation, DSP, heavyweight Sorrifier in Apollo, and the Lightweight Sorrifier are therefore concrete strategy implementations used in this paper, rather than fixed components in the definition of RAF.

Figure 1 summarizes the framework's verification-guided control flow. The orchestrator receives a proof task and optional retrieved context, dispatches an ordered portfolio of strategies, and returns the first candidate that Lean verifies as complete.

4.1 Framework Components

RAF separates proof construction from orchestration and formal verification. Its execution model contains four components.

Proof task and optional context. The input is a Lean proof task T as defined in Section 3. An optional retriever may construct formal context K from the available library; when retrieval is disabled, $K = \emptyset$.

Strategy portfolio. A portfolio $\mathcal{R} = (r_1, \ldots, r_m)$ contains independently implemented proof strategies. A strategy may generate a complete proof from the theorem, construct and fill a proof sketch, transform an existing candidate, or repair localized failures. RAF requires neither a common internal algorithm nor a common repair granularity.

Shared Lean verifier. Every candidate is judged by the same verifier V. Strategy code does not declare its own success: RAF admits a result only when Lean classifies it as complete according to Section 3. Diagnostics, source locations, and unresolved placeholders remain available to later strategies.

Orchestrator. The orchestrator executes the configured strategy order, records intermediate candidates and verifier results, enforces attempt budgets, and stops at the first complete proof. If a candidate is not complete, control returns to the orchestrator, which selects the next configured strategy. In the current implementation this control logic is realized by composite strategies such as the Direct–Heavyweight–Lightweight and DSP–Heavyweight–Lightweight configurations; the policy is deliberately simple and configuration driven.

4.2 Common Strategy Interface

Each RAF strategy implements a common source-to-source interface. For strategy r_i, let T_i denote the Lean task source selected by the orchestrator as its input. The strategy produces a candidate

$$P_i = r_i(T_i, K), \tag{9}$$

after which RAF obtains $V(P_i)$. The optional context K is exposed through the shared prompt-building environment; a strategy that does not require retrieval may ignore it.

The input selector is part of the configuration rather than the strategy interface. Consequently, T_i may be the original theorem source, the initial generated candidate, or the output of an earlier strategy. This distinction is important for non-destructive fallback. For example, the Lightweight Sorrifier in our main configuration restarts from the initial generated candidate instead of modifying the already coarsened Apollo output.

The implementation captures this contract in a common `ProofStrategy` abstraction. A strategy receives Lean source and an optional proof environment, returns transformed source, and is followed by the same verifier call. Composite strategies use this interface to combine simpler strategies while recording each intermediate result separately.

4.3 Verification-Guided Orchestration

Algorithm 1 gives the generic control loop. The history $\mathcal{H}$ stores strategy identities, candidates, and verification results. The configuration-specific function SelectInput determines whether the next strategy receives the original source or an earlier candidate.

Algorithm 1: RAF strategy orchestration

Input: Proof task T, optional context K, ordered portfolio $\mathcal{R} = (r_1, \ldots, r_m)$, verifier V
Output: A complete Lean proof or failure

$\mathcal{H} \leftarrow \emptyset$;
for $i \leftarrow 1$ **to** m **do**
  $T_i \leftarrow \textsc{SelectInput}(i, T, \mathcal{H})$;
  $P_i \leftarrow r_i(T_i, K)$;
  $Z_i \leftarrow V(P_i)$;
  $\mathcal{H} \leftarrow \mathcal{H} \cup \{(r_i, P_i, Z_i)\}$;
  if $C(P_i) = 1$ **then**
    return P_i
return NoCompleteProof;

This control flow gives strategies a uniform fallback semantics without making them depend on one another. Strategy r_i need not know whether it is the first strategy, which strategy precedes it, or which strategy will run next. The orchestrator alone owns order and termination. The same interface also supports a larger search budget: a configuration may invoke a strategy more than once or allow it to return several candidates, with every candidate still subject to the shared completeness test.

4.4 Optional Formal Context

Retrieval is a context service shared by the framework, not a mandatory first strategy. Given task T and a Lean library $\mathcal{L}$, it constructs

$$K = R(T, \mathcal{L}) = \{d_1, \ldots, d_k\}, \tag{10}$$

where each d_i contains a declaration name and type signature but not a proof body. Strategies may consume K when building a prompt; the orchestration and verification interfaces are unchanged when $K = \emptyset$.

Our implementation asks Lean for the raw `Syntax` tree of the target theorem, extracts application heads and recognized operator forms while excluding local binders, matches these signals against an index of Mathlib theorem headers, and uses BM25 to order structurally matched candidates. The top $k = 10$ headers form K. These steps define the extraction and ranking procedure used by the optional context service.

4.5 Strategies Used in This Work

Our main RAF instantiation uses Direct generation followed by two complementary repair strategies. DSP-based initial generation is described in Appendix A.2.

4.5.1 Direct Generation Strategy

Direct prompts a fixed language model once for a complete proof body, reconstructs it with the original imports, preceding declarations, and theorem header, and submits it to the shared Lean verifier. Retrieved context K is included when enabled. Direct initializes all three main configurations below.

4.5.2 Apollo Repair Strategy

Apollo is integrated as a repair strategy operating on an existing candidate. It first normalizes generated syntax, constructs a proof tree, and applies Apollo's structure-aware sorrification and repair procedure (Ospanov et al., 2025). Its handlers may replace a containing line, branch, or subtree in response to a diagnostic. This broader repair boundary can be robust when the reported location does not isolate the semantic cause, although it may also discard surrounding proof content that another strategy could reuse. RAF treats the complete Apollo pipeline as one strategy and judges only its returned candidate with the shared verifier.

4.5.3 Lightweight Sorrification Strategy

The Lightweight Sorrifier provides a second, independent repair strategy. It uses verifier spans directly rather than selecting a larger region from the proof tree. For each diagnostic $e \in E(P)$, let $\mathrm{span}(e)$ be its reported line interval. One lightweight step is

$$\mathrm{LS}(P, E(P)) = P\big[\mathrm{span}(e) \leftarrow \texttt{sorry} \mid e \in E(P)\big]. \tag{11}$$

Overlapping intervals are merged, and replacements are applied in reverse source order so that later edits do not invalidate earlier locations. Lean then checks the modified candidate again. Because an early failure can conceal later errors, the process repeats until Lean accepts the sorrified candidate or the cycle budget is exhausted.

Algorithm 2: Verifier-guided lightweight sorrification

Input: Candidate Lean source P, verifier V, cycle budget M
Output: Accepted candidate or failure
for $t \leftarrow 1$ **to** M **do**
  $(A, _, E, _) \leftarrow V(P)$;
  if A **then**
    return P
  $R \leftarrow$ MergeSpans(E);
  if $R = \emptyset$ **then**
    break
  foreach $r \in$ Reverse(R) **do**
    $P \leftarrow$ ReplaceWithSorry(P, r);
return Failure;

After sorrification, the strategy attempts to close every remaining `sorry`. It first queries Lean's `hint` facility and accepts a suggestion when a single usable tactic is exposed. Remaining holes receive a bounded tactic portfolio for simplification, normalization, ring reasoning, and linear or nonlinear arithmetic. A tactic is retained only when subsequent Lean verification shows that it closes the associated goal. The strategy returns the reconstructed source to RAF, where it is successful only if the shared verifier reports $C(P) = 1$.

The lightweight and Apollo strategies intentionally remain separate. A diagnostic span may preserve useful local structure when it is precise, whereas Apollo's broader structural boundary can be more robust when localization is poor. RAF does not encode either preference globally; their order is a property of the evaluated configuration.

4.6 Evaluated RAF Configurations

The repair-composition experiment uses three cumulative configurations:

$$\mathcal{R}_{\mathrm{D}} = [\textsc{Direct}], \tag{12}$$

$$\mathcal{R}_{\mathrm{DA}} = [\textsc{Direct}, \textsc{Apollo}], \tag{13}$$

$$\mathcal{R}_{\mathrm{DAL}} = [\textsc{Direct}, \textsc{Apollo}, \textsc{Lightweight}]. \tag{14}$$

The final strategy in $\mathcal{R}_{\mathrm{DAL}}$ receives the original Direct candidate when Apollo remains incomplete. Thus, its result measures complementary coverage rather than repair of Apollo's already modified output. The experimental tables use these portfolio symbols directly; prose may refer to the full $\mathcal{R}_{\mathrm{DAL}}$ instantiation as *RAF*. This label denotes the evaluated portfolio, not a restriction on the framework.

5 Experiments

We evaluate whether composing Direct generation with complementary repair strategies improves end-to-end proof completion across different LLM backends. The main study isolates repair composition, while Appendix A separately evaluates external formal knowledge and transfer to DSP-based initial generation. Because those runs use independently generated candidates, we treat them as supplementary evidence rather than part of the controlled repair-composition comparison.

5.1 Experimental Setup

Benchmark and models. We evaluate on the miniF2F test set, which contains 244 competition-style mathematical problems formalized in Lean. The repair-composition study uses five LLM backends: Goedel-V2-8B, GPT-5.6 Luna, gemini-3.8-flash, o4-mini, and o3-mini (Lin et al., 2026; OpenAI, 2026; Google, 2026; OpenAI, 2025b;a). Within each comparison, all configurations use the same theorem statements, prompt format, Lean environment, and model backend.

Evaluated configurations. The main experiment evaluates three cumulative configurations. *Direct* generates one whole-proof candidate and verifies it immediately. *Direct + Apollo* applies Apollo to Direct candidates that remain incomplete. *RAF* retains proofs completed by either preceding stage and applies the Lightweight Sorrifier after Apollo remains incomplete. Apollo uses a broad, tree-based repair boundary, whereas the Lightweight Sorrifier replaces only the source spans identified by Lean diagnostics. The Lightweight Sorrifier restarts from the original Direct candidate rather than modifying Apollo's coarsened output.

Protocol and metrics. The repair-composition experiment uses the full 244-problem test set as the denominator and one generated candidate per theorem. Missing or unrecorded executions are counted as `ERROR`. A `COMPLETE` result is a Lean-verified proof with no unresolved `sorry`; this count and its rate are our primary metrics. An `ACCEPTED` result elaborates but still contains at least one `sorry`, `REJECTED` denotes source that Lean does not accept, and `ERROR` denotes an execution-pipeline failure. Pipeline errors are treated as non-complete outcomes in all completion-rate calculations.

5.2 Experimental Results

5.2.1 Repair Composition across Models

Table 1 reports the cumulative outcome distribution for $\mathcal{R}_{\mathrm{D}}$, $\mathcal{R}_{\mathrm{DA}}$, and $\mathcal{R}_{\mathrm{DAL}}$, as defined in Section 4. Each configuration retains proofs completed by its preceding strategies.

Table 1: End-to-end miniF2F results across five LLM backends. Each row contains 244 problems, and the configurations are cumulative. The completion rate uses `COMPLETE` only.

Model	Configuration	Complete	Accepted	Rejected	Error	Rate
Goedel-V2-8B	$\mathcal{R}_{\mathrm{D}}$	135	11	98	0	55.33%
	$\mathcal{R}_{\mathrm{DA}}$	152	71	21	0	62.30%
	$\mathcal{R}_{\mathrm{DAL}}$	**154**	68	21	1	**63.11%**
GPT-5.6 Luna	$\mathcal{R}_{\mathrm{D}}$	135	1	107	1	55.33%
	$\mathcal{R}_{\mathrm{DA}}$	**150**	83	10	1	**61.48%**
	$\mathcal{R}_{\mathrm{DAL}}$	**150**	84	9	1	**61.48%**
gemini-3.8-flash	$\mathcal{R}_{\mathrm{D}}$	129	15	100	0	52.87%
	$\mathcal{R}_{\mathrm{DA}}$	144	85	15	0	59.02%
	$\mathcal{R}_{\mathrm{DAL}}$	**149**	84	11	0	**61.07%**
o4-mini	$\mathcal{R}_{\mathrm{D}}$	15	0	229	0	6.15%
	$\mathcal{R}_{\mathrm{DA}}$	67	120	56	1	27.46%
	$\mathcal{R}_{\mathrm{DAL}}$	**78**	138	27	1	**31.97%**
o3-mini	$\mathcal{R}_{\mathrm{D}}$	4	0	238	2	1.64%
	$\mathcal{R}_{\mathrm{DA}}$	50	120	71	3	20.49%
	$\mathcal{R}_{\mathrm{DAL}}$	**75**	127	39	3	**30.74%**

Table 1 shows that Apollo substantially improves over Direct generation across all five backends, while the Lightweight Sorrifier provides a further model-dependent gain, adding 2, 0, 5, 11, and 25 complete proofs, respectively. The Direct completion rates suggest that the first three models already have relatively strong Lean proof-generation capability on miniF2F. Consequently, fewer of their remaining failures fall within the limited scope of a simple strategy such as Lightweight, and its marginal improvement is modest. In contrast, o4-mini and o3-mini have much lower Direct completion rates, and Lightweight recovers 11 and 25 additional proofs that Apollo leaves incomplete. These results illustrate the main benefit of RAF: strategies with different repair boundaries can address complementary failure modes, so composing multiple strategies provides broader coverage than relying on a single repair procedure.

6 Conclusion

In this work, we introduced RAF, a training-free framework that addresses a central bottleneck in automated theorem proving (ATP): bridging the gap between high-level generative reasoning in LLMs and the rigid, unforgiving elaboration constraints of interactive proof assistants like Lean 4. Rather than relying on computationally prohibitive fine-tuning or rigid, single-boundary repair heuristics, RAF establishes a modular architecture that combines optional pre-generation formal context retrieval with complementary post-generation repair strategies. Central to this approach is our diagnostic-guided Lightweight Sorrifier, which replaces failing source spans with explicit proof obligations while strictly preserving valid surrounding derivations. Evaluations across five diverse LLMs on the miniF2F benchmark demonstrate the practical power of this paradigm: composing repair boundaries consistently matches or outperforms baseline repair pipelines, yielding up to a 10.25-percentage-point improvement and completing more theorems than monolithic repair methods. These results highlight that failure modes in neural formal proofs are inherently heterogeneous and thus require multi-granularity recovery mechanisms. Beyond its benchmark performance, RAF points to a scalable, cost-effective paradigm for modern ATP. By decoupling proof generation, verifier diagnostics, and specialized repair portfolios behind a shared, verified interface, the framework provides a general blueprint for integrating arbitrary language models into formal mathematics environments without retraining.

AI USE STATEMENT

In this work, we used generative AI tools for language polishing of author-written text and for assistance with LaTeX formatting. We have not used generative AI tools for research ideation, experimental design, data generation, or analysis of results. All AI-assisted text was reviewed by the authors, and all technical claims, proofs, and experimental results were produced and verified by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1229–1241, 2023.

Herbert Gelernter. Realization of a geometry-theorem proving machine. In *Computers & thought*, pp. 134–152. 1995.

Paul C Gilmore. A proof method for quantification theory: its justification and realization. *IBM Journal of research and development*, 4(1):28–35, 1960.

Google. Gemini 3.8 flash. `https://ai.google.dev/gemini-api/docs/models/gemini-3.8-flash/`, 2026.

Thomas C Hales. Introduction to the flyspeck project. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2006.

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.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, 2009.

Matěj Kripner, Michal Šustr, and Milan Straka. Leantree: Accelerating white-box proof search with factorized states in lean 4. *arXiv preprint arXiv:2507.14722*, 2025.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

Xavier Leroy. *The CompCert C verified compiler: Documentation and user's manual*. PhD thesis, Inria, 2024.

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

The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20, pp. 367–381. ACM, January 2020. doi: 10.1145/3372885.3373824. URL `http://dx.doi.org/10.1145/3372885.3373824`.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

OpenAI. Openai o3-mini system card. https://openai.com/index/o3-mini-system-card/, 2025a.

OpenAI. Openai o3 and o4-mini system card. https://openai.com/index/o3-o4-mini-system-card/, 2025b.

OpenAI. Gpt-5.6 luna. https://developers.openai.com/api/docs/models/gpt-5.6-luna, 2026.

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

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

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 Additional Experiments

This appendix studies two optional variations that are deliberately separated from the main repair-composition experiment: retrieved formal context for Direct generation and DSP as an alternative initial proof strategy.

A.1 External Formal Knowledge for Direct Generation

Table 2 compares initial whole-proof generation with and without retrieved formal knowledge. Conditions are paired by theorem within each model. Because these are separate generation runs from the main repair-composition experiment, the appropriate baseline for each model is the No-K column in this table, not the Direct row in Table 1.

Table 2: Effect of external formal knowledge on Direct generation over the miniF2F test set. Each entry reports the number of complete proofs followed by the completion rate in parentheses.

Model	**No** K	**With** K
Goedel-V2-8B	133 (54.51%)	136 (55.74%)
GPT-5.6 Luna	131 (53.69%)	153 (62.70%)
gemini-3.8-flash	143 (58.61%)	140 (57.38%)
o4-mini	18 (7.38%)	24 (9.84%)
o3-mini	4 (1.64%)	10 (4.10%)

Retrieval increases Direct completion for Goedel-V2-8B, GPT-5.6 Luna, o4-mini, and o3-mini, but decreases it for gemini-3.8-flash. The largest absolute gain is obtained by GPT-5.6 Luna, which completes 22 additional proofs with retrieved context (from 131 to 153). The relative gains for o4-mini and o3-mini are also notable because their no-retrieval completion rates are substantially lower than those of Goedel-V2-8B, GPT-5.6 Luna, and gemini-3.8-flash. One plausible explanation is that general-purpose models with less Lean-specific knowledge make more errors that retrieved declarations and examples can help prevent. A model already exposed to more Lean code has less missing formal knowledge for retrieval to supply, so the same context provides a smaller marginal benefit.

The decrease for gemini-3.8-flash also shows why retrieval is not uniformly helpful. Retrieved material can be irrelevant to the required proof, introduce incompatible lemmas or notation, or distract generation from a proof the model could already produce without assistance. We therefore interpret retrieval as most useful when it compensates for missing Lean knowledge and when the retrieved context is sufficiently precise. This is an explanatory hypothesis rather than a causal error analysis. The Direct-generation ablation isolates the effect on initial generation.

A.2 DSP-Based Initial Generation

A.2.1 DSP Initial Proof Strategy

The Draft, Sketch, and Prove (DSP) strategy introduced by Jiang et al. (2022) first asks the model for a concise informal derivation, uses that derivation to construct a structured Lean sketch, and then attempts to fill the resulting proof obligations. Our experiments use a Lean 4 adaptation of DSP as an alternative initial strategy, demonstrating that RAF does not require the initial candidate to originate from one-shot Direct generation.

A.2.2 Portfolio Configurations and Results

We apply the same repair composition used in the main experiment to structured candidates produced by this DSP adaptation. The supplementary study uses three cumulative portfolios:

$$\mathcal{R}_{\mathrm{DSP}} = [\mathrm{DSP}], \tag{15}$$

$$\mathcal{R}_{\mathrm{DSPA}} = [\mathrm{DSP}, \textsc{Apollo}], \tag{16}$$

$$\mathcal{R}_{\mathrm{DSPAL}} = [\mathrm{DSP}, \textsc{Apollo}, \textsc{Lightweight}]. \tag{17}$$

Table 3 reports the full repair composition without external knowledge. Because external knowledge is supplied only to the initial DSP strategy, Table 4 separately isolates its effect on DSP generation. Each condition contains all 244 problems; missing or unrecorded final candidates count as `ERROR`.

Table 3: Full-set DSP portfolio results without external knowledge ($K = \emptyset$). Configurations are cumulative and each row contains 244 problems.

Model	Configuration	Complete	Accepted	Rejected	Error	Rate
Goedel-V2-8B	$\mathcal{R}_{\mathrm{DSP}}$	44	99	99	2	18.03%
	$\mathcal{R}_{\mathrm{DSPA}}$	76	155	11	2	31.15%
	$\mathcal{R}_{\mathrm{DSPAL}}$	**77**	151	14	2	**31.56%**
GPT-5.6 Luna	$\mathcal{R}_{\mathrm{DSP}}$	0	0	244	0	0.00%
	$\mathcal{R}_{\mathrm{DSPA}}$	**71**	164	8	1	**29.10%**
	$\mathcal{R}_{\mathrm{DSPAL}}$	**71**	150	22	1	**29.10%**
gemini-3.8-flash	$\mathcal{R}_{\mathrm{DSP}}$	1	2	241	0	0.41%
	$\mathcal{R}_{\mathrm{DSPA}}$	73	155	15	1	29.92%
	$\mathcal{R}_{\mathrm{DSPAL}}$	**74**	152	17	1	**30.33%**
o4-mini	$\mathcal{R}_{\mathrm{DSP}}$	0	0	244	0	0.00%
	$\mathcal{R}_{\mathrm{DSPA}}$	70	161	12	1	28.69%
	$\mathcal{R}_{\mathrm{DSPAL}}$	**72**	144	27	1	**29.51%**
o3-mini	$\mathcal{R}_{\mathrm{DSP}}$	0	0	244	0	0.00%
	$\mathcal{R}_{\mathrm{DSPA}}$	60	117	66	1	24.59%
	$\mathcal{R}_{\mathrm{DSPAL}}$	**75**	142	26	1	**30.74%**

Table 4: Effect of external formal knowledge on DSP initial generation over the miniF2F test set. Each entry reports the number of complete proofs followed by the completion rate in parentheses.

Model	No K	With K
Goedel-V2-8B	44 (18.03%)	49 (20.08%)
GPT-5.6 Luna	0 (0.00%)	0 (0.00%)
gemini-3.8-flash	1 (0.41%)	0 (0.00%)
o4-mini	0 (0.00%)	0 (0.00%)
o3-mini	0 (0.00%)	0 (0.00%)

Table 3 shows that Apollo supplies most of the recovery from DSP candidates. The Lightweight Sorrifier contributes a further 1, 0, 1, 2, and 15 complete proofs for Goedel-V2-8B, GPT-5.6 Luna, gemini-3.8-flash, o4-mini, and o3-mini, respectively, with its largest marginal gain occurring for o3-mini.

Table 4 isolates retrieval at the initial DSP stage. Goedel-V2-8B completes 44 proofs without retrieval and 49 with it, whereas the other backends produce at most one without retrieval and none with it. This suggests that external knowledge can guide a model that already produces correct proofs but sometimes fails because it lacks a relevant declaration or example. When a model produces no complete proofs, however, the bottleneck is not a local knowledge gap that retrieval can fill; extra context alone does not make the strategy viable in this experiment.

B Prompt Template

B.1 Structured Whole-Proof Generation Prompt

For whole-proof generation, we use a structured prompt that presents the target Lean source, optional retrieved knowledge, and an explicit response schema. We refer to it as the *structured whole-proof generation prompt*; its format does not assume a particular model family. The target block contains the imports and declarations preceding the theorem, any available informal description, and the theorem header with its proof replaced by `sorry`. When external knowledge retrieval is enabled, the prompt additionally includes the retrieved Lean declaration headers as optional hints. When no hints are available, the entire hint block is omitted.

The complete prompt template is shown below. Braced text denotes content filled in for each theorem.

```
You are a Lean 4 proof assistant.
Given a Lean statement, you must generate complete Lean 4 code
that compiles.

#PROVE_BEGIN
{imports and preceding declarations}
{optional informal description}
{theorem header}  sorry
#PROVE_END

The following are hints that you may use or not:
#HINT_BEGIN
{retrieved Lean declaration headers and optional context}
#HINT_END

Complete `sorry` in the proof and give step-by-step reasoning.
Each `have` clause must have a clear type.
Explain any theorem used from the supplied hints.
Return exactly this structure:

#REASONING_BEGIN 1
...
#REASONING_END 1

#CODE_BEGIN 1
...
#CODE_END 1
```

The tagged response separates explanatory reasoning from executable Lean code. The post-processor extracts the content between `#CODE_BEGIN` and `#CODE_END`. If the model returns a complete theorem declaration, the post-processor retains the proof body following `:= by`. The extracted proof body is then reconstructed with the original imports, declarations, and theorem header before Lean verification. This separation allows reasoning text to guide generation without passing non-code content to the verifier.

C External Knowledge Extraction

This appendix gives a concrete example of how the optional context K is constructed. The procedure has three stages: obtain Lean's raw syntax tree, extract library-relevant terms while excluding local names, and rank matching Mathlib declarations.

C.1 Obtaining the Raw Syntax Tree

We define a small Lean command that receives the following theorem declaration as syntax and prints its raw `Syntax` representation. The command is placed immediately before the target theorem in the source sent to the Lean server.

```
open Lean Elab Command
set_option pp.raw true

elab "#print_theorem_syntax " stx:command : command => do
  logInfo m!"{stx.raw}"

#print_theorem_syntax
theorem mathd_numbertheory_495
    (a b : Nat)
    (h0 : 0 < a /\ 0 < b)
    (h1 : a % 10 = 2)
    (h2 : b % 10 = 4)
    (h3 : Nat.gcd a b = 6) :
    108 <= Nat.lcm a b := by sorry
```

Lean returns a tree whose outer nodes describe the declaration, theorem name, binders, result type, and proof body. A shortened fragment is shown below; the actual output retains the complete tree.

```
(Command.declaration
  (Command.theorem "theorem"
    (Command.declId `mathd_numbertheory_495 [])
    (Command.declSig
      [(Term.explicitBinder ... [`a `b] ...)
       ...
       (Term.app `Nat.gcd [`a `b]) ...]
      (Term.typeSpec ":"
        (... (Term.app `Nat.lcm [`a `b]))))
    (Command.declValSimple ":="
      (Term.byTactic "by" ...))))
```

C.2 Extracting Retrieval Terms

We traverse this tree and record application heads and recognized operator forms from the theorem statement. At the same time, names introduced by explicit, implicit, or instance binders are collected as local names and excluded from the query. In the example, `a`, `b`, and the hypothesis names are local binders, whereas `Nat.gcd`, `Nat.lcm`, remainder, equality, order, and conjunction contribute library-relevant signals. This syntax-based extraction preserves resolved qualified names such as `Nat.lcm` instead of relying only on words in the surface string.

C.3 Matching and Ranking Mathlib Declarations

We scan an index containing Mathlib declaration names and type signatures, but not proof bodies. Declarations that share extracted heads or normalized operator aliases form the candidate set. We then rank these candidates with BM25 and return the highest-ranked k headers. BM25 is used as a standard ranking component rather than introduced as a contribution of this work; the retrieval contribution here is the Lean-aware construction of the query and candidate set. The selected names and signatures form K and may be included in a generation strategy's prompt.