Proof-Refactor: Refactoring Generated Formal Proofs into Modular Artifacts

Yiming Fu [1] **Peixuan Liu** [1] **Zichen Wang** [2] **Kun Yuan** [3]

Abstract

Generated formal proofs often compile but remain hard to read, maintain, and reuse. We study the post-generation refactoring problem: given a verified proof, produce a better-structured verified proof artifact for the same theorem. We introduce **Proof-Refactor**, an agentic framework that decomposes this task into extracting candidate proof fragments, designing helper declarations, proving the extracted and designed components, and repairing the original proof using the verified helpers. The system combines Claude Code, `lean-lsp-mcp`, a custom `lean_extract` tool, and independent-context external assistance for fragment selection and helper design. On generated Lean proofs from PutnamBench and Putnam2025, Proof-Refactor improves LLM-as-a-judge rubric-based refactoring scores over a baseline that uses Claude Code with `lean-lsp-mcp` and `lean4-skills`, with the largest gains in signature quality and human readability. These results suggest that process-guided refactoring can improve the structure of verified proof artifacts without optimizing directly for proof length.

1. Introduction

Formal theorem proving with proof assistants such as Lean (de Moura et al., 2015) and Isabelle (Paulson, 1989) produces machine-verifiable mathematical proofs. This is useful for verified mathematics and for training LLMs, since proof checking provides a precise reward signal for reinforcement learning (RL). Recent systems have made progress on university-level mathematics, including PutnamBench (Tsoukalas et al., 2024) and Putnam2025 problems (Axiom Math Team, 2025).

However, proof-assistant verification also induces a narrow objective: a generated proof is rewarded primarily for compiling. This binary feedback ensures correctness, but it does not by itself encourage readability, modularity, maintainability, or library integration. An agent may therefore produce a passing proof that leaves reusable arguments inlined, introduces problem-specific definitions that reinvent existing APIs, or relies on heavy automation and enlarged resource limits rather than factoring the argument into intermediate declarations. This reflects a mismatch between what is easy to check and the broader qualities expected of library-quality formal mathematics.

This gap motivates us to study post-generation refactoring as a way to improve proof quality beyond correctness. In this setting, the input is an already Lean-verified proof of a fixed theorem statement; the goal is not to prove the theorem from scratch, but to produce a new Lean-verified proof artifact for the same statement with cleaner helper declarations and a more readable owner proof.

Existing approaches to improving proof quality (Ahuja et al., 2024; Gu et al., 2025) often rely on computable objectives. In practice, the most reliable automatic objectives are largely length-based, while qualities such as readability, modularity, maintainability, and reusability are difficult to measure. Length-based objectives can also favor direct compression (golfing) over semantically meaningful helper declarations.

This viewpoint also mirrors human formalization practice. A formalizer may first obtain a compiling proof and then refactor it. The central step is not only to reduce length, but to identify reusable proof fragments, generalize them into declarations that fit the surrounding library, prove these declarations from the existing fragments, and rewrite the original proof to call them. Proof golfing is a legitimate part of refactoring in this broad sense, but it is not the whole process: focusing only on golfing misses the helper-oriented workflow that turns local arguments into reusable library components.

We therefore view proof refactoring as a structured process rather than end-to-end optimization against a proxy metric. We propose **Proof-Refactor**, a Claude Code-

Code: `github.com/pelicanhere/proof-refactor`.

[1]Department of Mathematics, Southern University of Science and Technology [2]School of Mathematical Sciences, Peking University [3]Center for Machine Learning Research, Peking University. Correspondence to: Kun Yuan <kunyuan@pku.edu.cn>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

based agentic framework that decomposes Lean proof refactoring into four phases: extracting candidate proof fragments, designing helper declarations, proving the extracted and designed components, and repairing the original proof with the verified components. Proof-Refactor does not include a dedicated length-minimization stage; instead, it focuses on extracting, generalizing, and reusing intermediate proof components. The framework integrates `lean-lsp-mcp` (Dressler, 2025), a custom Lean 4 `extract` tactic for turning proof fragments into standalone theorems, and external assistance for selecting fragments and designing helper declarations. Experiments on generated PutnamBench and Putnam2025 proofs provide evidence that Proof-Refactor improves refactoring quality over the baseline, with the clearest gains in helper-signature quality, proof structure, reuse of mathematical infrastructure, and human readability.

2. Refactoring: Task and Evaluation

Task Definition. Let E denote the ambient formal environment available to a declaration, including imported library declarations. Given a verified declaration (S, p), where S is its statement and p is its original proof, the initial data are (E, p), satisfying $E \vdash p : S$.

We formulate an individual refactoring as a constrained transformation $(E, p) \rightsquigarrow_{\text{refactor}} (E', p')$:

$$(E, p) \rightsquigarrow_{\text{refactor}} (E', p') \quad \text{s.t.} \quad \begin{cases} E' \vdash p' : S \\ (E', p') \succ_{\mathcal{Q}} (E, p) \end{cases} \tag{1}$$

Here, $E \vdash p : S$ means: under the environment E, the proof p of the statement S complies. The relation $(E', p') \succ_{\mathcal{Q}} (E, p)$ states that the refactored proof is preferred to the original proof according to the quality preference $\mathcal{Q}$. The same preference can also be applied to a finite set of candidate refactorings to compare or rank them against the original proof. More generally, E' denotes a refactored proof environment: it may reorganize the surrounding artifact, but it must preserve the mathematical meaning of the fixed statement S and introduce no new axioms. In this paper, we only consider transformations where $E' = E \cup H$, where H is a set of newly introduced declarations; the original environment E is left unchanged. The proof itself may differ from the original proof, as long as the final artifact verifies the same statement under these constraints.

Evaluation. We instantiate $\mathcal{Q}$ with a rubric-based quality score that compares candidate refactorings against the original proof. The judge protocol and length-based diagnostics are described in Section 4.

3. Proof-Refactor: Process-Driven Refactoring

As shown in Figure 2, Proof-Refactor is a four-phase agentic framework for Lean proof refactoring, built on Claude Code and `lean-lsp-mcp`. The pipeline can call external models through an independent-context interface. We extend `lean-lsp-mcp` with `lean_extract`, a custom extraction tool based on the Lean tactic `extract`, which can extract proof fragments into standalone theorems.

3.1. Lean-LSP-MCP

We use `lean-lsp-mcp`, which is the Model Context Protocol (MCP) server for Lean. We implement a prompt about the `lean-lsp-mcp` API modified from Freer (2025); the prompt files are included in the accompanying code repository. Concretely, we use these tools in three main ways:

- **Context inspection.** We use `lean_file_outline`, `lean_hover_info`, `lean_goal`, and `lean_diagnostic_messages` to inspect file structure, declaration information, proof states, and Lean error messages.
- **Declaration search.** The system uses `lean_local_search` to retrieve local and Mathlib declarations by name, `lean_leansearch` for natural-language search over Mathlib, and `lean_loogle` for type-pattern search.
- **Proof interaction.** We use `lean_multi_attempt` to test multiple candidate tactics in parallel, `lean_code_actions` to resolve Lean suggestions, and `lean_run_code` to execute standalone Lean snippets. Each attempted change is checked using `lean_goal` and `lean_diagnostic_messages`.

3.2. The Extract Tactic

The `extract` tactic is a custom Lean tactic for proof extraction. It lifts a selected tactic block inside a proof into a standalone theorem. We design it for two cases: (1) the block closes the goal or leaves a goal definitionally equal to the original goal, so the extracted theorem keeps that goal; and (2) the block leaves one remaining non-equivalent goal, so the extracted theorem states the implication from the post-block goal to the pre-block goal. The extracted theorem keeps only variables actually used by the tactic block and removes unused variables.

This design supports both `have`-style subproofs and contiguous tactic blocks that transform the goal to another

goal. It differs from Mathlib's `extract_goal`, which extracts a theorem from the current goal but does not support case (2) and cannot clean up some variables properly. We integrate `extract` into `lean-lsp-mcp` through `lean_extract`, a script that inserts `extract` wrappers and the extracted theorem into the file. We call this inserted declaration a *scaffold*: an intermediate top-level theorem whose signature is produced by `extract`. A scaffold is not yet a final helper declaration; it is a structured artifact passed to the later phases. Figure 1 shows the resulting before/after structure on a small proof fragment.

Scaffolds play three roles. First, they expose local assumptions, goals, and proof fragments for helper design. Second, they localize proving to smaller lemmas rather than the whole owner proof. Third, during repair, the inserted `extract` wrapper records the original tactic block, and the scaffold proof body gives a verified guide for reconstruction.

3.3. Four-phase Pipeline

Proof-Refactor decomposes refactoring into four sequential phases. Each phase is a separate Claude Code session with `lean-lsp-mcp` that does not inherit the previous session's context; the Lean file produced by the previous phase is the only artifact passed forward.

External Assistance. We implement a CLI that allows the pipeline to call external models with an independent context. The purpose is to decouple high-level refactoring reasoning from low-level interaction with Lean: the external model can reason about which fragments are mathematically meaningful and how they should be generalized, while the Claude Code agent remains responsible for editing, checking, and repairing Lean code. In practice, external assistance is used primarily in the extraction and helper design phases, where it helps select proof fragments and propose reusable helper declarations. The pipeline can also consult it as a secondary fallback when revising helper signatures or proving unusually difficult obligations.

Phase 1: Extraction. The extraction phase begins by asking external assistance to read the original Lean proof and return extraction suggestions, each corresponding to a tactic block. The agent then calls `lean_extract` to turn the suggestions into scaffolds. It checks each scaffold with `lean-lsp-mcp`; if any errors, the agent repairs them before moving to the next phase. The inserted `extract` wrapper is left in the file for later phases.

Phase 2: Helper Design. The design phase reads each scaffold together with its owner declaration, and then groups scaffolds by similarity. The agent writes the scaffolds, owner declarations, relevant Mathlib declarations, and search evidence into a Markdown file, then asks the external model to design helpers for each group. The agent reviews the proposed helper signatures against design principles. If needed, the agent may make a secondary call to external assistance for revision. The resulting helper declarations are inserted into a Lean file, and each declaration is annotated with dependency information to organize the proof. In this dependency structure, scaffolds are temporary: no other declaration is allowed to depend on them. The agent then proceeds to the next phase.

Phase 3: Proving. The proving phase fills in each `sorry` in the Lean file. The agent reads the scaffolds, helper declarations, owner declarations, and extraction sites, and then proves each theorem guided by the dependency annotations. It may consult the external model when it encounters a difficult proof obligation, but this is rarely used in practice: most goals are small lemmas, and the orig-

Before extraction

```
theorem Set.inter_comm (s t : Set Nat) : s ∩ t = t ∩ s := by
  ext x; constructor
  · intro h
    rw [Set.mem_inter_iff] at h ⊢
    exact h.symm
  · intro h
    rw [Set.mem_inter_iff] at h ⊢
    exact h.symm
```

After extraction

```
theorem Set.inter_comm.mem_inter_swap_1 (s t : Set ℕ) (x : ℕ) :
  x ∈ s ∩ t → x ∈ t ∩ s := sorry

theorem Set.inter_comm.mem_inter_swap_2 (s t : Set ℕ) (x : ℕ) :
  x ∈ t ∩ s → x ∈ s ∩ t := sorry

theorem Set.inter_comm (s t : Set Nat) : s ∩ t = t ∩ s := by
  ext x; constructor
  · extract "mem_inter_swap_1" {
    intro h
    rw [Set.mem_inter_iff] at h ⊢
    exact h.symm
    }
  · extract "mem_inter_swap_2" {
    intro h
    rw [Set.mem_inter_iff] at h ⊢
    exact h.symm
    }
```

Figure 1. Example proof before and after extraction. The two inserted declarations are scaffolds, and `Set.inter_comm` is the owner declaration.

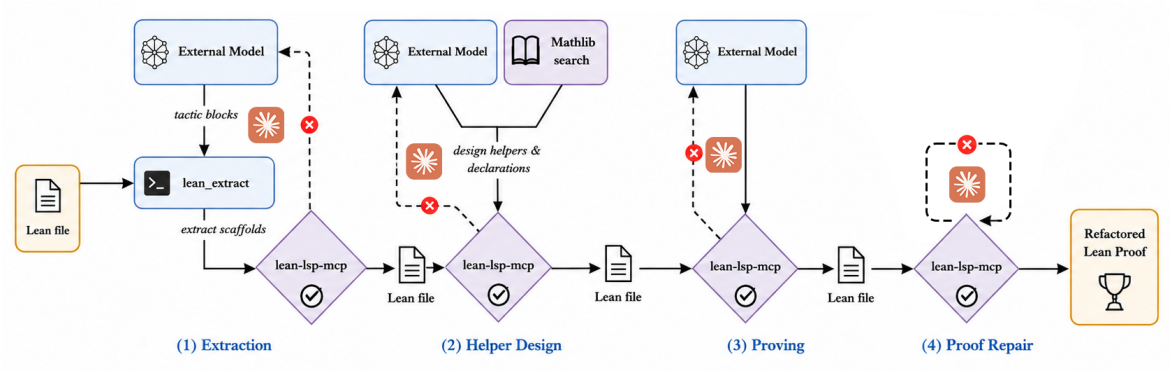

Figure 2. Overview of the Proof-Refactor pipeline.

inal proof fragment usually provides the proof idea, even when its code cannot be reused verbatim. Once all scaffolds and helper declarations have been proved, the agent proceeds to the next phase.

Phase 4: Proof Repair. Finally, the agent repairs the original proof by following the dependency annotations and invoking the helper declarations. At each extraction site, it uses the corresponding scaffold body as guidance for rewriting, while ensuring that the repaired proof does not call the scaffold itself. The agent then removes the `extract` wrapper and the scaffolds, and adds mathematical annotations.

4. Experiments

We evaluate Proof-Refactor on generated Lean proofs from PutnamBench and Putnam2025. Each evaluation instance is a self-contained Lean file whose only external dependency is mathlib. Our evaluation focuses on refactoring quality rather than direct length-based optimization; accordingly, we do not emphasize proof golfing as an explicit objective.

4.1. Experimental setup

Datasets. We evaluate on generated Lean proofs from PutnamBench (Tsoukalas et al., 2024) and Putnam2025. For PutnamBench, we use Claude Code with DeepSeek-V4-Pro (DeepSeek-AI, 2026) to generate candidate proofs and select 96 proofs whose lengths mostly range from 50 to 500 lines, with a median of approximately 160 lines. We additionally evaluate on 12 Putnam2025 proofs generated by Axiom Math Team (2025).

Models and tools. Except for the evaluation judge described below, all refactoring agents are built on Claude Code and `lean-lsp-mcp`. Both our method and the baseline use Claude Opus 4.6 (Anthropic, 2026) as the underlying model. The baseline is Claude Code with the `lean4:refactor` command from lean4-skills (Freer, 2025). When our method invokes an external model, we use Gemini 3.1 Pro (Google DeepMind, 2026). We consider this a strong refactoring baseline: the lean4-skills agent can inspect Lean proof states through `lean_goal` and use them to guide proof-state-guided lemma extraction, and its prompts require the extracted lemma signatures to be clean and general. Moreover, its authors intentionally separate refactoring from proof golfing, which matches our experimental setup.

Evaluation. All evaluated files are required to pass Lean verification, so our evaluation does not score correctness. Instead, we evaluate the quality of the refactored proof artifact. This is difficult to capture with a single automatic metric: proof length may reward golfing rather than readability, and the number of introduced lemmas may reward fake modularity. We therefore use a rubric-based evaluation protocol that scores refactoring quality along five dimensions: structure, signature quality, tactic quality, reuse, and human readability. The rubric explicitly penalizes wrappers, dead helpers, oversized helpers, and broad tactics that hide the main proof idea, while rewarding meaningful helper lemmas with natural, general, and locally checkable statements.

Our primary evaluator is an LLM-as-a-judge agent. To reduce dependence on the system prompt or tool behavior of the refactoring agents, the judge is implemented separately from Claude Code. The judge is given the original proof and the refactored outputs, together with the fixed rubric. It can query LeanSearch (Gao et al., 2025) to retrieve relevant Mathlib declarations, allowing it to assess whether a refactoring follows library style by reusing existing abstractions rather than reinventing them. For each method, the judge assigns scores to the rubric submetrics, and the overall score is the average of the non-overall rubric scores.

As a sanity check for the automatic evaluation, we additionally conduct a human review on 28 valid problems using

the same rubric. We compare the judge with the human review at the theorem level by asking whether they select the same better refactoring between the two methods. The judge agrees with the human review on this pairwise ranking for 75.0% of the sampled theorems.

Finally, following prior work (Gu et al., 2025), we also report length-based diagnostics. In Table 1, we instantiate length as the word count of the whole Lean file: for an original file p and a refactored file p', the word reduction rate is

$$\mathrm{WRR}(p, p') = \frac{\mathrm{words}(p) - \mathrm{words}(p')}{\mathrm{words}(p)},$$

where $\mathrm{words}(\cdot)$ denotes whole-file word count. We also report the average number of words reduced, $\mathrm{words}(p) - \mathrm{words}(p')$, averaged over instances. These length-based diagnostics are used only as secondary measures, not as the primary measure of refactoring quality. We freeze all prompts, scripts, model versions, and evaluation settings before running the experiments.

4.2. Main results

Table 1 reports the main results. Proof-Refactor improves the average quality score over the baseline by +0.31 on Putnam2025 and +0.45 on PutnamBench. The largest gains appear in Signature Quality and Human Readability, suggesting that explicitly designing reusable helper declarations produces proofs that are more natural and better aligned with library conventions. Structure and Reuse also improve consistently, indicating that the refactored proofs are better organized and make more effective use of existing mathematical infrastructure.

The length-based diagnostics in Table 1, measured by whole-file word counts, show a more nuanced pattern. Proof-Refactor achieves a higher word reduction rate on PutnamBench (+2.5pp over the baseline), but a lower reduction rate on Putnam2025 (-3.8pp). This is consistent with our objective: the pipeline is not optimized directly for proof golfing, and its quality gains primarily come from reorganizing proofs around reusable helper declarations. The gain in Tactic Quality is therefore smaller but remains positive, reflecting our focus on high-level proof architecture rather than tactic-level compression.

Table 1. Comparison between baseline and ours.

Dataset	Metric	Baseline	Proof-Refactor	Improvement
Putnam2025 (12 problems)	**Average score**	3.96	**4.27**	**+0.31**
	Structure	4.01	4.31	+0.29
	Signature Quality	3.76	**4.28**	**+0.51**
	Tactic Quality	4.04	4.22	+0.18
	Reuse	4.00	4.29	+0.29
	Human Readability	3.99	4.26	+0.28
	Word reduction rate	8.3%	4.5%	-3.8pp
	Avg. words reduced	625.9	337.2	-288.7
PutnamBench (96 problems)	**Average score**	3.75	**4.19**	**+0.45**
	Structure	3.80	4.21	+0.41
	Signature Quality	3.46	**4.14**	**+0.68**
	Tactic Quality	3.97	4.19	+0.22
	Reuse	3.84	4.24	+0.41
	Human Readability	3.67	**4.18**	**+0.50**
	Word reduction rate	4.0%	6.5%	+2.5pp
	Avg. words reduced	105.5	118.1	+12.6

4.2.1. Case study: Putnam 1968 B6.

```
theoremputnam_1968_b6
  : ¬∃ K : ℕ → Set ℚ, (∀ n : ℕ, IsCompact (K n)) ∧ (∀ S : Set ℚ, IsCompact S → ∃ n : ℕ, S ⊆ K n) := by
    intro h; rcases h with⟨K, ⟨hK, hcover⟩⟩
    have h_ball_nonempty (n : ℕ) : (Metric.ball (0 : ℚ) (1 / ((n : ℝ) + 1))).Nonempty := by...
    have h_exists_q (n : ℕ) : ∃ q : ℚ, q ∈ Metric.ball 0 (1/((n:ℝ)+1)) ∧ q ∉ K n := by...
    choose q hq_ball hq_notin using h_exists_q
    let S : Set ℚ := {0} ∪ Set.range q
    have hS_compact : IsCompact S := by
      rw [IsCompact_iff_finite_subcover]
      intro ι U hU_open hU_cover; classical
      rcases Set.mem_iUnion.mp (hU_cover (by simp [S])) with⟨i0, hi0⟩
      obtain⟨ε, hε_pos, hball_sub⟩ : ∃ ε > 0, Metric.ball 0 ε ⊆ U i0 := ...
      obtain⟨N, hN⟩ : ∃ N : ℕ, 1 / (((N:ℝ):ℝ) + 1) < ε := exists_nat_one_div_lt hε_pos
      have h_tail_in_U0 (n : ℕ) (hn : N ≤ n) : q n ∈ U i0 := by ...
      have h_fin_covers_head (n : ℕ) (hn : n < N) : ∃ i : ι, q n ∈ U i := by ...
      let idx_fn : ℕ → ι := fun n => if hn : n < N then Classical.choose ... else i0
      let head_indices : Finset ι := Finset.image idx_fn (Finset.range N)
      let t : Finset ι := {i0} ∪ head_indices
      refine⟨t, ?_⟩
      intro x hxS; rcases hxS with (rfl | ⟨n, rfl⟩)
      · apply Set.mem_iUnion.mpr; refine⟨i0, Finset.mem_union.mpr (Or.inl (by simp)), hi0⟩
      · by_cases hnN : n < N
        · ... (deep nesting proving x in subcover through idx_fn) ...
        · ... (deep nesting proving x in subcover tail cases) ...
    have hS_not_covered (n : ℕ) : ¬ S ⊆ K n := by ...
    rcases hcover S hS_compact with⟨n, hn⟩; exact hS_not_covered n hn
```

Original generated proof (86 lines)

```
private lemmaisCompact_zero_union_range (q : ℕ → ℚ)
  (hq_ball : ∀ n : ℕ, q n ∈ Metric.ball (0 : ℚ) (1 / ((n : ℝ) + 1))) :
  IsCompact (({0 : ℚ}) ∪ Set.range q) := by
    rw [isCompact_iff_finite_subcover]
    intro ι U hU_open hU_cover; classical
    rcases Set.mem_iUnion.mp (hU_cover (by simp)) with⟨i0, hi0⟩
    obtain⟨ε, hε_pos, hball_sub⟩ : ∃ ε > (0 : ℝ), Metric.ball 0 ε ⊆ U i0 := ... (open set logic)
    obtain⟨N, hN⟩ : ∃ N : ℕ, 1 / (((N:ℝ):ℝ) + 1) < ε := exists_nat_one_div_lt hε_pos
    have h_tail_in_U0 (n : ℕ) (hn : N ≤ n) : q n ∈ U i0 := by ... (ε-bound logic)
    have h_fin_covers_head (n : ℕ) (hn : n < N) : ∃ i : ι, q n ∈ U i := by ...
    -- Manual explicit construction of finite indexing set
    let idx_fn : ℕ → ι := fun n => if hn : n < N then Classical.choose (h_fin_covers_head n hn) else i0
    let head_indices : Finset ι := Finset.image idx_fn (Finset.range N)
    let t : Finset ι := {i0} ∪ head_indices
    refine⟨t, ?_⟩
    intro x hxS; rcases hxS with (rfl | ⟨n, rfl⟩)
    · exact Set.mem_iUnion.mpr ⟨i0, Finset.mem_union.mpr (Or.inl (by simp)), hi0⟩
    · by_cases hnN : n < N
      · ... (prove idx_fn mapping resolves into head_indices)
      · ... (prove N ≤ n lands correctly into h_tail_in_U0)
theoremputnam_1968_b6
  : ¬∃ K : ℕ → Set ℚ, (∀ n : ℕ, IsCompact (K n)) ∧ (∀ S : Set ℚ, IsCompact S → ∃ n : ℕ, S ⊆ K n) := by
    intro h; rcases h with⟨K, ⟨hK, hcover⟩⟩
    choose q hq_ball hq_notin using h_exists_q ... (density logic hidden)
    let S : Set ℚ := {0} ∪ Set.range q
    have hS_compact : IsCompact S := isCompact_zero_union_range q hq_ball
    ... (derive contradiction same as pipeline) ...
```

Baseline refactoring (84 lines)

```
lemmaRat.exists_mem_open_not_mem_compact
  {U K : Set ℚ} (hU_open : IsOpen U) (hU_nonempty : U.Nonempty) (hK_compact : IsCompact K) :
  ∃ q ∈ U, q ∉ K := by
    have hdense : Dense (Kᶜ) := Rat.dense_compl_compact hK_compact
    have h_inter := (dense_iff_inter_open.mp hdense) _ hU_open hU_nonempty
    rcases h_inter with⟨q, hq_U, hq_compl⟩
    exact⟨q, hq_U, hq_compl⟩
lemmatendsto_of_mem_ball_one_div_succ
  {α : Type*} [PseudoMetricSpace α] (x : α) (q : ℕ → α)
  (hq : ∀ (n : ℕ), q n ∈ Metric.ball x (1 / ((n : ℝ) + 1))) :
  Filter.Tendsto q Filter.atTop (nhds x) := by
    rw [Metric.tendsto_atTop]
    intro ε hε
    obtain⟨N, hN⟩ := exists_nat_one_div_lt hε
    refine⟨N, fun n hn => ?_⟩
    ... (linarith bounds checking on 1/(n+1) ≤ 1/(N+1) < ε) ...
theoremputnam_1968_b6
  : ¬∃ K : ℕ → Set ℚ, (∀ n : ℕ, IsCompact (K n)) ∧ (∀ S : Set ℚ, IsCompact S → ∃ n : ℕ, S ⊆ K n) := by
    intro h; rcases h with⟨K, ⟨hK, hcover⟩⟩
    have h_exists_q (n : ℕ) : ∃ q : ℚ, q ∈ Metric.ball 0 (1/((n:ℝ)+1)) ∧ q ∉ K n := by
      exact Rat.exists_mem_open_not_mem_compact Metric.isOpen_ball (h_ball_nonempty n) (hK n)
    choose q hq_ball hq_notin using h_exists_q
    -- Key API usage makes the main logic extremely concise
    have hq_tendsto := tendsto_of_mem_ball_one_div_succ 0 q hq_ball
    let S : Set ℚ := {0} ∪ Set.range q
    have hS_compact : IsCompact S := hq_tendsto.isCompact_insert_range
    have hS_not_covered (n : ℕ) : ¬ S ⊆ K n := by
      intro h_sub; exact hq_notin n (h_sub (by simp [S]))
    have h_contra := hcover S hS_compact
    rcases h_contra with⟨n, hn⟩
    exact hS_not_covered n hn
```

Proof-Refactor (52 lines)

Figure 3. Case study on Putnam 1968 B6. Snippets are lightly abridged and reformatted to fit the main text.

To better understand the aggregate gains in Table 1, we examine Putnam 1968 B6 as a representative case where the main challenge is not shortening tactics, but exposing the proof's mathematical structure; the complete Lean artifacts for this case are listed in Appendix A.1. The proof naturally decomposes into four steps: assuming, for contradic-

tion, that there exists a sequence K_n covering all compact sets; choosing a point q_n close to 0 but not contained in K_n for each K_n; using $q_n \to 0$ to construct the compact set $S = \{0\} \cup \mathrm{range}(q)$; and finally deriving a diagonal contradiction from $q_n \in S$ and $q_n \notin K_n$.

While the original proof interleaves these steps with substantial low-level topological details—such as expanding the compactness proof into an explicit open-cover argument—the baseline merely extracts this into a highly specialized helper lemma, `isCompact_zero_union_range`. Because this baseline lemma fixes the space to $\mathbb{Q}$, the limit point to 0, the radius to $1/(n+1)$, and the conclusion directly to the compactness of $\{0\} \cup \mathrm{range}(q)$, it acts as a direct translation of the original proof structure; moreover, its proof body is itself largely copied from the corresponding part of the original proof. This highlights a primary failure mode of the baseline: because it attempts an end-to-end refactor within a single coding session, it is overburdened. Consequently, it struggles to propose clean, library-style lemma signatures or discover simpler proofs, even when explicitly prompted with a checklist.

By contrast, Proof-Refactor mitigates these limitations by using a four-phase pipeline with external assistance. The two lemmas extracted by our approach correspond to core mathematical steps that substantially generalize the original local fragments. Specifically, `Rat.exists_mem_open_not_mem_compact` generalizes "choosing $q_n \notin K_n$ from a specific ball" into the statement that every nonempty open subset of the rationals contains a point outside any given compact set; meanwhile, `tendsto_of_mem_ball_one_div_succ` abstracts the open-cover estimate into a general convergence criterion stating that in any pseudo-metric space, if the n-th term of a sequence lies in the ball centered at x with radius $1/(n+1)$, then the sequence converges to x.

This allows the main proof to obtain the compactness of S directly via `hq_tendsto.isCompact_insert_range`. The case study thus explains the pattern observed in Table 1: Proof-Refactor's improvements come primarily from changing the interface between the main theorem and its helper declarations, rather than from local tactic optimization. By turning local proof fragments into reusable mathematical facts and then reusing appropriate library declarations, the pipeline directly improves Signature Quality, Structure, and Reuse, while also making the resulting proof easier to inspect for human readability. This also explains why the gain in Tactic Quality is smaller: Proof-Refactor targets high-level proof architecture, not tactic-level proof golfing.

Figure 3 illustrates the same point at the code level. The important difference is not simply that the final proof has fewer lines, but that the long local compactness argument is hidden behind two helper statements whose interfaces can be read without reconstructing the entire Putnam proof. The main theorem therefore records the diagonal construction, while the reusable topology facts are isolated as separate declarations. By contrast, the baseline still asks the reader to trust a problem-specific compactness lemma with a large proof body. This is the distinction captured by the rubric: reusable interfaces and library-level facts are rewarded even when they do not produce the shortest local tactic script. The visual contrast also clarifies the line-count numbers: the 52-line version is shorter because general facts absorb repeated topology details, not because the proof is compressed into a denser sequence of tactics. This makes the improvement more robust than a surface-level reduction in tokens.

4.3. Ablations and analysis

We run an ablation that removes external assistance while keeping all other settings fixed. The ablation is evaluated on Putnam2025 and a 48-problem subset of PutnamBench. All metrics are measured as improvements over the original generated proofs, and we report the ablated score minus the full-pipeline score as the ablation gap. This ablation removes the primary external calls used for fragment selection and helper design, as well as the rare fallback calls used for helper revision or difficult proof obligations; the corresponding prompt files are included in the accompanying code repository.

Table 2 shows that removing external assistance changes the pipeline from a positive average improvement (+0.31) to a negative one (-0.55), increases the loss rate from 52.2% to 69.6%, and reduces the win rate by 8.7 percentage points. The largest drops are in Structure (-1.34), Human Readability (-1.17), and Signature Quality (-0.98). This supports the importance of decoupling high-level reasoning from Lean interaction. Without external assistance, the same agent context must simultaneously manage tool calls, Lean diagnostics, local proof repair, search results, and the higher-level task of designing reusable helper declarations. This mixed context can become overloaded and poorly focused, which directly hurts the design phase: helper signatures become less clean, less general, and less aligned with library conventions. Reuse also drops substantially (-0.91), suggesting that the ablated pipeline is less able to turn local proof fragments into reusable components. Tactic Quality drops less sharply (-0.59), consistent with the fact that local proof repair can still succeed even when the higher-level refactoring architecture is weaker. The rare fallback calls used during proving are included in the ablation, but the main effect is the loss of the reasoning–interaction separation used for fragment selection and helper design.

Table 2. Ablation study on removing external assistance.

Group	Metric	Full	Ablated	Gap
Overall shared probs.	Pipeline Δ	+0.31	-0.55	-0.86
	Win rate	39.1%	30.4%	-8.7pp
	Loss rate	52.2%	69.6%	+17.4pp
Dimension improv.	Structure	+0.41	-0.93	-1.34
	Sig. quality	+0.68	-0.30	-0.98
	Human read.	+0.50	-0.67	-1.17
	Reuse	+0.41	-0.50	-0.91
	Tactic qual.	+0.22	-0.37	-0.59

5. Related work

Proof Optimization. A related line of work improves formal proofs through computable objectives, especially proof length (Ahuja et al., 2024; Gu et al., 2025). Ahuja et al. (2024) combine LLM search with retrieval-augmented generation (RAG) and Chain-of-States, while ProofOptimizer (Gu et al., 2025) trains models with length-filtered data and length-based RL rewards. These systems show that post-generation proof improvement is feasible and can produce synthetic training data for models. Our work targets a complementary form of improvement: helper declarations are the central object of a structured refactoring pipeline, rather than a byproduct of optimizing a metric.

Proof Extraction. Another related direction is extracting artifacts from existing formal proofs. REFACTOR (Zhou et al., 2024) trains a graph neural network (GNN) to identify proof-tree nodes that form extractable theorem components in Metamath (Megill & Wheeler, 2019), then standardizes the predicted components into valid theorems. Xin et al. (2025) study a more symbolic approach, discovering reusable tactics by searching for recurring, collapsible subgraphs in Tactic Dependence Graphs. These works are closely related to proof refactoring, but extraction alone tends to preserve the form and specificity of the original fragments rather than lifting them into more general, library-style artifacts. This motivates the dedicated helper-design phase in Proof-Refactor, which explicitly revises extracted scaffolds into cleaner and more reusable declarations.

Lemmas and Library Growth. Lemmas have long supported proof search, from lemma-driven proving and induction lemma discovery to learning-assisted selection from large libraries (Boyer & Moore, 1977; Kapur & Subramaniam, 1996; Kaliszyk & Urban, 2015). LEGO-Prover (Wang et al., 2023) also emphasizes verified lemmas, but its mechanism is different. It assumes an informal statement/proof and a formal statement; a prompt-driven decomposer emits lemma requests, and a library-level solver/evolver retrieves, proves, or transforms skills through predefined, heuristic directions. This is forward, informal-sketch-driven proof search for not-yet-verified theorems, rather than extraction from an existing verified artifact. Proof-Refactor starts from the opposite end: an already Lean-verified proof, where `extract` exposes formal proof fragments as scaffolds that record local assumptions, goals, used variables, and tactic-block effects. Helper signatures are designed from these proof-state-backed scaffolds and repaired back into the same owner proof. Thus, the shared focus on lemmas hides different machinery: prompt-driven library growth for theorem proving versus proof-state-backed refactoring of the current verified artifact.

Agentic Systems for Theorem Proving. Hilbert (Varambally et al., 2026) and Numina-Lean-Agent (Liu et al., 2026) are open-source agentic systems for formal theorem proving. Hilbert combines informal reasoning, theorem retrieval, formal proving, and verifier feedback to recursively decompose hard goals into subgoals. Numina-Lean-Agent builds on Claude Code and Numina-Lean-MCP, using tools for Lean interaction, retrieval, informal proving, and external-model consultation through a discussion partner. These systems are related to the draft-sketch-prove paradigm (Jiang et al., 2023), where informal reasoning supplies high-level proof sketches that guide formal provers toward a complete proof.

We adopt two system-level ideas from this line: using Claude Code with `lean-lsp-mcp` as the Lean interaction substrate, and allowing the agent to consult an external model. However, our use of external assistance is not an informal-to-formal sketching pipeline. Proof-Refactor starts from an existing Lean proof, and external assistance serves as an auxiliary reasoning channel for refactoring decisions over the current formal context, rather than only producing informal proof sketches. Thus, while it is related in spirit to Numina-Lean-Agent's discussion partner, the goal is to support proof refactoring rather than to produce a proof plan for theorem proving.

6. Conclusion

We introduced Proof-Refactor, a process-guided framework for refactoring generated Lean proofs into more modular artifacts. Rather than treating proof improvement as direct optimization of a single proxy such as length, Proof-Refactor decomposes the process into extraction, helper design, proving, and proof repair. This makes helper declarations the central object of refactoring: local proof fragments are exposed, generalized into reusable declarations, proved, and then used to reorganize the original proof. This staged design avoids overloading a single end-to-end agent

with extraction, design, proving, and repair at once, and allows each phase to focus on a more specific refactoring subproblem.

A key design principle is to decouple high-level refactoring reasoning from low-level Lean interaction. External assistance is used mainly to reason about meaningful proof fragments and helper interfaces, while the Claude Code agent remains responsible for Lean editing, checking, proving, and repair. The ablation study supports this separation: removing external assistance substantially degrades structure, signature quality, readability, and reuse, suggesting that mixing helper design with tool-heavy Lean interaction in a single context weakens the refactoring process.

Experiments on generated PutnamBench and Putnam2025 proofs provide evidence that this process improves refactoring quality over a strong baseline, with the clearest gains in Signature Quality, Structure, Reuse, and Human Readability. The case study further shows that Proof-Refactor can recover the conceptual structure of a proof rather than merely copying local proof-engineering details.

7. Limitations

Scope. This work studies a restricted refactoring setting: each evaluation instance is a self-contained Lean file, and the refactored environment only adds new declarations. The pipeline does not edit existing library declarations, reorganize projects, change imports, or perform multi-file refactoring, so our experiments do not establish project- or library-level scalability. Future work should extend Proof-Refactor to reason about existing declarations, naming conventions, import structure, and changes to the surrounding environment, with the long-term goal of making generated formal proofs suitable for integration into mature libraries such as mathlib (mathlib Community, 2019).

Evaluation. Our main evaluation uses a rubric-based LLM-as-a-judge protocol, augmented with retrieval from mathlib and supplemented by human review on a subset. This provides a practical way to evaluate high-level proof quality, but it is not a substitute for large-scale expert review. The benchmark is also limited to generated Putnam-style Lean proofs, so the results may not directly generalize to other mathematical domains, larger developments, or proofs written by human formalizers.

Cost. Proof-Refactor uses four sequential phases and external assistance, making it more expensive than a single end-to-end refactoring pass.

Failure modes. Because the pipeline is sequential, errors in early phases can propagate. In particular, failures or bugs in `lean_extract` can produce poor scaffolds, making helper design, proving, and repair harder. The design phase may also reject useful problem-specific lemmas when it applies the generality criterion too strictly.

Impact Statement

Beyond Proof-Refactor, `lean_extract` may help theorem-proving systems isolate local contexts and extracted subgoals, and derive formalization data from existing proofs. The staged decomposition may also inform code refactoring; generated helpers should remain drafts before integration.

References

Ahuja, R., Avigad, J., Tetali, P., and Welleck, S. Improver: Agent-based automated proof optimization. *arXiv preprint arXiv:2410.04753*, 2024.

Anthropic. Claude opus 4.6 system card, February 2026. URL https://www.anthropic.com/claude-opus-4-6-system-card.

Axiom Math Team. From seeing why to checking everything. https://axiommath.ai/territory/from-seeing-why-to-checking-everything, 2025. Axiom Math blog post.

Boyer, R. S. and Moore, J. S. A lemma driven automatic theorem prover for recursive function theory. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 511–519, 1977. URL https://www.ijcai.org/Proceedings/77-1/Papers/089.pdf.

de Moura, L. M., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In Felty, A. P. and Middeldorp, A. (eds.), *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pp. 378–388. Springer, 2015. doi: 10.1007/978-3-319-21401-6_26. URL https://doi.org/10.1007/978-3-319-21401-6_26.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Dressler, O. Lean LSP MCP: Tools for agentic interaction with the Lean theorem prover, 3 2025. URL https://github.com/oOo0oOo/lean-lsp-mcp.

Freer, C. Lean 4 Skills: Theorem proving skill and workflow pack for AI coding agents, October 2025. URL https://github.com/cameronfreer/lean4-skills.

Gao, G., Ju, H., Jiang, J., Qin, Z., and Dong, B. A semantic search engine for mathlib4, 2025. URL https://arxiv.org/abs/2403.13310.

Google DeepMind. Gemini 3.1 pro model card. Technical report, Google DeepMind, February 2026. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf.

Gu, A., Piotrowski, B., Gloeckle, F., Yang, K., and Markosyan, A. H. Proofoptimizer: Training language models to simplify proofs without human demonstrations, 2025. URL https://arxiv.org/abs/2510.15700.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs, 2023. URL https://arxiv.org/abs/2210.12283.

Kaliszyk, C. and Urban, J. Learning-assisted theorem proving with millions of lemmas. *Journal of symbolic computation*, 69: 109–128, 2015.

Kapur, D. and Subramaniam, M. Lemma discovery in automating induction. In *Automated Deduction – CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pp. 538–552. Springer, 1996. doi: 10.1007/3-540-61511-3_112.

Liu, J., Zhou, Z., Zhu, Z., Santos, M. D., He, W., Liu, J., Wang, R., Xie, Y., Zhao, J., Wang, Q., Zhi, L., Li, J., and Li, W. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics, 2026. URL https://arxiv.org/abs/2601.14027.

mathlib Community, T. The lean mathematical library. *CoRR*, abs/1910.09336, 2019. URL http://arxiv.org/abs/1910.09336.

Megill, N. and Wheeler, D. A. *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019.

Paulson, L. C. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989. doi: 10.1007/BF00248324. URL https://doi.org/10.1007/BF00248324.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. PutnamBench: Evaluating neural theorem-provers on the putnam mathematical competition. *Advances in Neural Information Processing Systems*, 37: 11545–11569, 2024.

Varambally, S., Voice, T., Sun, Y., Chen, Z., Yu, R., and Ye, K. Hilbert: Recursively building formal proofs with informal reasoning, 2026. URL https://arxiv.org/abs/2509.22819.

Wang, H., Xin, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

Xin, Y., Xin, J., Poesia, G., Goodman, N., Chen, Q., and Dillig, I. Automated discovery of tactic libraries for interactive theorem proving, 2025. URL https://arxiv.org/abs/2503.24036.

Zhou, J. P., Wu, Y., Li, Q., and Grosse, R. Refactor: Learning to extract theorems from proofs. *arXiv preprint arXiv:2402.17032*, 2024.

A. Additional Artifacts

This appendix records selected Lean examples and the evaluation rubric. We omit the full prompt suites from this appendix to keep the source concise; full prompt files are available in the accompanying code repository. Individual listings are captioned by role rather than by repository path.

Table 3. Appendix artifact index.

ID	Artifact group	Appendix location
A1	Lean showcase artifacts	Appendix A.1, Listings 1–6
A2	Evaluation rubric	Appendix A.2, Listing 7

A.1. Lean Showcase Files

The listings below provide representative Lean artifacts, including the Putnam 1968 B6 case study and another representative example. Each example includes the original generated proof, the `lean4:refactor` baseline, and Proof-Refactor.

```
import Mathlib

open Finset Polynomial Topology Filter Metric

/--
Prove that no sequence $\{K_n\}_{n=0}^{\infty}$ of compact (closed and bounded) sets of rational numbers has the
     property that every compact set of rational numbers is contained by at least one $K_n$.
-/
theorem putnam_1968_b6
  : ¬ K :   Set , ( n : , IsCompact (K n))  ( S : Set , IsCompact S   n : , S  K n) := by
  intro h
  rcases h with K, hK, hcover
  have h_ball_nonempty (n : ) : (Metric.ball (0 : ) (1 / ((n : ) + 1))).Nonempty := by
    rw [Metric.nonempty_ball]
    positivity
  have h_exists_q (n : ) :  q : , q  Metric.ball (0 : ) (1 / ((n : ) + 1))  q  K n := by
    have hdense : Dense ((K n)) := Rat.dense_compl_compact (hK n)
    have hball_open : IsOpen (Metric.ball (0 : ) (1 / ((n : ) + 1))) := Metric.isOpen_ball
    have h_inter := (dense_iff_inter_open.mp hdense) _ hball_open (h_ball_nonempty n)
    rcases h_inter with q, hq_mem, hq_not
    exact q, hq_mem, hq_not
  choose q hq_ball hq_notin using h_exists_q
  let S : Set  := {0}  Set.range q
  have hS_compact : IsCompact S := by
    rw [isCompact_iff_finite_subcover]
    intro  U hU_open hU_cover
    classical
    have h0_mem_S : (0 : )  S := by
      simp [S]
    have h0_in_cover : (0 : )   i, U i := hU_cover h0_mem_S
    rcases Set.mem_iUnion.mp h0_in_cover with i0, hi0
    have hi0_open : IsOpen (U i0) := hU_open i0
    obtain , h_pos, hball_sub :   > (0 : ), Metric.ball (0 : )   U i0 := by
      rcases (Metric.isOpen_iff.mp hi0_open) (0 : ) hi0 with , h_pos, hball_sub
      exact , h_pos, hball_sub
    obtain N, hN :  N : , 1 / (((N : ) : ) + 1) <  :=
      exists_nat_one_div_lt h_pos
    have h_tail_in_U0 (n : ) (hn : N  n) : q n  U i0 := by
      have h_bound : dist (q n) (0 : ) <  := by
        have hqb := Metric.mem_ball.mp (hq_ball n)
        have h_denom : 1 / (((n : ) : ) + 1)  1 / (((N : ) : ) + 1) := by
          refine (one_div_le_one_div (by positivity) (by positivity)).mpr ?_
          have h_n_ge_N : (N : )  (n : ) := by exact mod_cast hn
          linarith
        linarith
      have hmem : q n  Metric.ball (0 : )  := by
        rw [Metric.mem_ball]
        exact h_bound
      exact hball_sub hmem
    have h_fin_covers_head (n : ) (hn : n < N) :  i : , q n  U i := by
      have hqn_mem_S : q n  S := by
        simp [S]
      have hqn_in_cover : q n   i, U i := hU_cover hqn_mem_S
      rcases Set.mem_iUnion.mp hqn_in_cover with i, hi
      exact i, hi
    let idx_fn :    := fun n =>
      if hn : n < N then Classical.choose (h_fin_covers_head n hn) else i0
```

```
    let head_indices : Finset  :=
      Finset.image idx_fn (Finset.range N)
    let t : Finset  := {i0}  head_indices
    refine t, ?_
    intro x hxS
    rcases hxS with (rfl | n, rfl)
    · apply Set.mem_iUnion.mpr
      refine i0, Finset.mem_union.mpr (Or.inl (by simp)), hi0
    · by_cases hnN : n < N
      · have hi_mem : idx_fn n  head_indices := by
          dsimp [head_indices]
          apply Finset.mem_image.mpr
          exact n, Finset.mem_range.mpr hnN, rfl
        have hqn_in_U : q n  U (idx_fn n) := by
          dsimp [idx_fn]
          rw [dif_pos hnN]
          exact Classical.choose_spec (h_fin_covers_head n hnN)
        apply Set.mem_iUnion.mpr
        exact idx_fn n, Finset.mem_union.mpr (Or.inr hi_mem), hqn_in_U
      · have hNle : N  n := by omega
        have hx_in_U0 : q n  U i0 := h_tail_in_U0 n hNle
        apply Set.mem_iUnion.mpr
        refine i0, Finset.mem_union.mpr (Or.inl (by simp)), hx_in_U0
  have hS_not_covered (n : ) : ¬ S  K n := by
    intro h_sub
    have hqn_in_Kn : q n  K n := h_sub (by simp [S])
    exact hq_notin n hqn_in_Kn
  have h_contra := hcover S hS_compact
  rcases h_contra with n, hn
  exact hS_not_covered n hn
```

Listing 1. Original generated proof for Putnam 1968 B6.

```
import Mathlib

open Finset Polynomial Topology Filter Metric

-- {0}  range(q) is compact when q(n)  0 at rate 1/(n+1)
private lemma isCompact_zero_union_range (q :  )
    (hq_ball :  n : , q n  Metric.ball (0 : ) (1 / ((n : ) + 1))) :
    IsCompact ({(0 : )}  Set.range q) := by
  rw [isCompact_iff_finite_subcover]
  intro  U hU_open hU_cover
  classical
  have h0_mem : (0 : )  ({(0 : )}  Set.range q) := by simp
  have h0_in_cover : (0 : )  i, U i := hU_cover h0_mem
  rcases Set.mem_iUnion.mp h0_in_cover with i0, hi0
  have hi0_open : IsOpen (U i0) := hU_open i0
  obtain , h_pos, hball_sub :  > (0 : ), Metric.ball (0 : )  U i0 := by
    rcases (Metric.isOpen_iff.mp hi0_open) (0 : ) hi0 with , h_pos, hball_sub
    exact , h_pos, hball_sub
  obtain N, hN :  N : , 1 / (((N : ) : ) + 1) <  :=
    exists_nat_one_div_lt h_pos
  -- All q(n) for n  N land in U(i0) via the -ball
  have h_tail_in_U0 (n : ) (hn : N  n) : q n  U i0 := by
    have h_bound : dist (q n) (0 : ) <  := by
      have hqb := Metric.mem_ball.mp (hq_ball n)
      have h_denom : 1 / (((n : ) : ) + 1)  1 / (((N : ) : ) + 1) := by
        refine (one_div_le_one_div (by positivity) (by positivity)).mpr ?_
        have h_n_ge_N : (N : )  (n : ) := by exact mod_cast hn
        linarith
      linarith
    exact hball_sub (Metric.mem_ball.mpr h_bound)
  -- The finitely many q(n) for n < N each have a cover element
  have h_fin_covers_head (n : ) (hn : n < N) :  i : , q n  U i := by
    have hqn_mem : q n  ({(0 : )}  Set.range q) := by simp
    rcases Set.mem_iUnion.mp (hU_cover hqn_mem) with i, hi
    exact i, hi
  let idx_fn :  := fun n =>
    if hn : n < N then Classical.choose (h_fin_covers_head n hn) else i0
  let head_indices : Finset  :=
    Finset.image idx_fn (Finset.range N)
  let t : Finset  := {i0}  head_indices
  refine t, ?_
  intro x hxS
  rcases hxS with (rfl | n, rfl)
  · exact Set.mem_iUnion.mpr i0, Finset.mem_union.mpr (Or.inl (by simp)), hi0
  · by_cases hnN : n < N
    · have hi_mem : idx_fn n  head_indices := by
        dsimp [head_indices]
```

```
        exact Finset.mem_image.mpr n, Finset.mem_range.mpr hnN, rfl
      have hqn_in_U : q n  U (idx_fn n) := by
        dsimp [idx_fn]
        rw [dif_pos hnN]
        exact Classical.choose_spec (h_fin_covers_head n hnN)
      exact Set.mem_iUnion.mpr
        idx_fn n, Finset.mem_union.mpr (Or.inr hi_mem), hqn_in_U
    · have hNle : N  n := by omega
      exact Set.mem_iUnion.mpr
        i0, Finset.mem_union.mpr (Or.inl (by simp)), h_tail_in_U0 n hNle

/--
Prove that no sequence $\{K_n\}_{n=0}^{\infty}$ of compact (closed and bounded) sets of rational numbers has the
    property that every compact set of rational numbers is contained by at least one $K_n$.
-/
theorem putnam_1968_b6
  : ¬ K :   Set , ( n : , IsCompact (K n))  ( S : Set , IsCompact S   n : , S  K n) := by
  intro h
  rcases h with K, hK, hcover
  -- For each n, choose q(n) near 0 but outside K(n), using density of K
  have h_ball_nonempty (n : ) : (Metric.ball (0 : ) (1 / ((n : ) + 1))).Nonempty := by
    rw [Metric.nonempty_ball]
    positivity
  have h_exists_q (n : ) :  q : , q  Metric.ball (0 : ) (1 / ((n : ) + 1))  q  K n := by
    have hdense : Dense ((K n)) := Rat.dense_compl_compact (hK n)
    have hball_open : IsOpen (Metric.ball (0 : ) (1 / ((n : ) + 1))) := Metric.isOpen_ball
    have h_inter := (dense_iff_inter_open.mp hdense) _ hball_open (h_ball_nonempty n)
    rcases h_inter with q, hq_mem, hq_not
    exact q, hq_mem, hq_not
  choose q hq_ball hq_notin using h_exists_q
  -- S = {0}  range(q) is compact (q converges to 0) but avoids every K(n)
  let S : Set  := {0}  Set.range q
  have hS_compact : IsCompact S := isCompact_zero_union_range q hq_ball
  have hS_not_covered (n : ) : ¬ S  K n := by
    intro h_sub
    exact hq_notin n (h_sub (by simp [S]))
  rcases hcover S hS_compact with n, hn
  exact hS_not_covered n hn
```

Listing 2. Baseline refactoring for Putnam 1968 B6.

```
import Mathlib

open Finset Polynomial Topology Filter Metric

/-- In the rationals, any nonempty open set contains a point outside any given compact set. -/
lemma Rat.exists_mem_open_not_mem_compact
    {U K : Set } (hU_open : IsOpen U) (hU_nonempty : U.Nonempty) (hK_compact : IsCompact K) :
     q  U, q  K := by
  have hdense : Dense (K) := Rat.dense_compl_compact hK_compact
  have h_inter := (dense_iff_inter_open.mp hdense) _ hU_open hU_nonempty
  rcases h_inter with q, hq_U, hq_compl
  exact q, hq_U, hq_compl

/-- A sequence whose n-th term lies in the ball of radius `1/(n+1)` around a point converges to that point. -/
lemma tendsto_of_mem_ball_one_div_succ
    { : Type*} [PseudoMetricSpace ] (x : ) (q :   )
    (hq :  (n : ), q n  Metric.ball x (1 / ((n : ) + 1))) :
    Filter.Tendsto q Filter.atTop (nhds x) := by
  rw [Metric.tendsto_atTop]
  intro  h
  obtain N, hN := exists_nat_one_div_lt h
  refine N, fun n hn => ?_
  have h1 := Metric.mem_ball.mp (hq n)
  have h2 : (1 : ) / ((n : ) + 1)  1 / ((N : ) + 1) := by
    apply one_div_le_one_div_of_le (by positivity)
    have : (N : )  (n : ) := by exact_mod_cast hn
    linarith
  linarith

/--
Prove that no sequence $\{K_n\}_{n=0}^{\infty}$ of compact (closed and bounded) sets of rational numbers has the
    property that every compact set of rational numbers is contained by at least one $K_n$.
-/
theorem putnam_1968_b6
  : ¬ K :   Set , ( n : , IsCompact (K n))  ( S : Set , IsCompact S   n : , S  K n) := by
  intro h
  rcases h with K, hK, hcover
  have h_ball_nonempty (n : ) : (Metric.ball (0 : ) (1 / ((n : ) + 1))).Nonempty := by
    rw [Metric.nonempty_ball]
```

```
    positivity
  have h_exists_q (n : ) :  q : , q  Metric.ball (0 : ) (1 / ((n : ) + 1))  q  K n := by
    exact Rat.exists_mem_open_not_mem_compact Metric.isOpen_ball (h_ball_nonempty n) (hK n)
  choose q hq_ball hq_notin using h_exists_q
  have hq_tendsto := tendsto_of_mem_ball_one_div_succ 0 q hq_ball
  let S : Set  := {0}  Set.range q
  have hS_compact : IsCompact S := hq_tendsto.isCompact_insert_range
  have hS_not_covered (n : ) : ¬ S  K n := by
    intro h_sub
    have hqn_in_Kn : q n  K n := h_sub (by simp [S])
    exact hq_notin n hqn_in_Kn
  have h_contra := hcover S hS_compact
  rcases h_contra with n, hn
  exact hS_not_covered n hn
```

Listing 3. Proof-Refactor output for Putnam 1968 B6.

```
import Mathlib

open Set

abbrev putnam_1971_a2_solution : Set (Polynomial ) := {Polynomial.X}
-- {Polynomial.X}
/--
Determine all polynomials $P(x)$ such that $P(x^2 + 1) = (P(x))^2 + 1$ and $P(0) = 0$.
-/
theorem putnam_1971_a2
    (P : Polynomial ) :
    (P.eval 0 = 0  ( x : , P.eval (x^2 + 1) = (P.eval x)^2 + 1))  P  putnam_1971_a2_solution := by
  rw [putnam_1971_a2_solution, Set.mem_singleton_iff]
  constructor
  · intro h0, heq
    by_contra h_ne
    let a :    :=  n => Nat.recOn n 0 ( _ ak => ak ^ 2 + 1)
    have ha0 : a 0 = 0 := rfl
    have ha_succ :  n, a (n + 1) = (a n) ^ 2 + 1 :=  _ => rfl
    have ha_nonneg :  n, 0  a n := by
      intro n
      induction' n with n ih
      · rfl
      · rw [ha_succ n]
        nlinarith
    have ha_pos :  n, 0 < a (n + 1) := by
      intro n
      rw [ha_succ n]
      nlinarith [ha_nonneg n]
    have ha_strictMono :  n, a n < a (n + 1) := by
      intro n
      rw [ha_succ n]
      rcases Classical.em (a n = 0) with (hzero | hpos_case)
      · rw [hzero]; norm_num
      · have hpos : 0 < a n := lt_of_le_of_ne (ha_nonneg n) (Ne.symm hpos_case)
        nlinarith
    have ha_inj : Function.Injective a := by
      intro x y h
      rcases Nat.lt_trichotomy x y with (hlt | heq | hlt)
      · have hle : x.succ  y := hlt
        have : a x < a y :=
          Nat.le_induction
            (by simpa [Nat.succ_eq_add_one] using ha_strictMono x)
            (fun k hk ih =>
              lt_trans ih (by simpa [Nat.succ_eq_add_one] using ha_strictMono k))
            y hle
        linarith
      · exact heq
      · have hle : y.succ  x := hlt
        have : a y < a x :=
          Nat.le_induction
            (by simpa [Nat.succ_eq_add_one] using ha_strictMono y)
            (fun k hk ih =>
              lt_trans ih (by simpa [Nat.succ_eq_add_one] using ha_strictMono k))
            x hle
        linarith
    have hP_a :  n, P.eval (a n) = a n := by
      intro n
      induction' n with n ih
      · exact h0
      · rw [ha_succ n, heq (a n), ih]
    have hroot :  n, (P - Polynomial.X).eval (a n) = 0 := by
      intro n
```

```
      rw [Polynomial.eval_sub, hP_a n, Polynomial.eval_X, sub_self]
    set Q := P - Polynomial.X with hQ_def
    have hQ_ne : Q  0 := by
      intro hQzero
      apply h_ne
      rw [hQ_def] at hQzero
      exact sub_eq_zero.mp hQzero
    set d := Q.natDegree with hd_def
    let Z : Finset  := (Finset.range (d + 1)).image a
    have hZ_card : Z.card = d + 1 := by
      rw [Finset.card_image_of_injective _ ha_inj, Finset.card_range]
    have hZ_val_subset : Z.val  Q.roots := by
      rw [Multiset.subset_iff]
      intro x hx
      rw [Finset.mem_val] at hx
      have hxZ : x  Z := hx
      rw [Polynomial.mem_roots hQ_ne]
      rcases Finset.mem_image.mp hxZ with n, _, rfl
      dsimp [Q, Polynomial.IsRoot]
      rw [Polynomial.eval_sub, hP_a n, Polynomial.eval_X, sub_self]
    have h_le : Z.card  Q.natDegree :=
      Polynomial.card_le_degree_of_subset_roots hZ_val_subset
    rw [hZ_card, hd_def] at h_le
    omega
  · intro hP
    subst hP
    refine ?_, ?_
    · simp
    · intro x; simp
```

Listing 4. Original generated proof for Putnam 1971 A2.

```
import Mathlib

open Set

abbrev putnam_1971_a2_solution : Set (Polynomial ) := {Polynomial.X}
-- {Polynomial.X}
/--
Determine all polynomials $P(x)$ such that $P(x^2 + 1) = (P(x))^2 + 1$ and $P(0) = 0$.
-/
theorem putnam_1971_a2
    (P : Polynomial ) :
    (P.eval 0 = 0  ( x : , P.eval (x^2 + 1) = (P.eval x)^2 + 1))  P  putnam_1971_a2_solution := by
  rw [putnam_1971_a2_solution, Set.mem_singleton_iff]
  constructor
  · intro h0, heq
    by_contra h_ne
    -- Define the orbit sequence a(0) = 0, a(n+1) = a(n)^2 + 1
    let a :    :=  n => Nat.recOn n 0 ( _ ak => ak ^ 2 + 1)
    have ha_succ :  n, a (n + 1) = (a n) ^ 2 + 1 :=  _ => rfl
    have ha_nonneg :  n, 0  a n := by
      intro n
      induction' n with n ih
      · rfl
      · rw [ha_succ n]; nlinarith
    -- The sequence is strictly increasing, hence injective
    have ha_strictMono :  n, a n < a (n + 1) := by
      intro n
      rw [ha_succ n]
      rcases Classical.em (a n = 0) with (hzero | hpos_case)
      · rw [hzero]; norm_num
      · have hpos : 0 < a n := lt_of_le_of_ne (ha_nonneg n) (Ne.symm hpos_case)
        nlinarith
    have ha_inj : Function.Injective a := (strictMono_nat_of_lt_succ ha_strictMono).injective
    -- P fixes every term of the sequence: P(a(n)) = a(n)
    have hP_a :  n, P.eval (a n) = a n := by
      intro n
      induction' n with n ih
      · exact h0
      · rw [ha_succ n, heq (a n), ih]
    -- So Q = P - X vanishes on every a(n)
    have hroot :  n, (P - Polynomial.X).eval (a n) = 0 := by
      intro n
      rw [Polynomial.eval_sub, hP_a n, Polynomial.eval_X, sub_self]
    -- Contradiction: Q has degree d but at least d + 1 distinct roots
    set Q := P - Polynomial.X with hQ_def
    have hQ_ne : Q  0 := by
      intro hQzero
      apply h_ne
```

```
      rw [hQ_def] at hQzero
      exact sub_eq_zero.mp hQzero
    set d := Q.natDegree with hd_def
    let Z : Finset  := (Finset.range (d + 1)).image a
    have hZ_card : Z.card = d + 1 := by
      rw [Finset.card_image_of_injective _ ha_inj, Finset.card_range]
    have hZ_val_subset : Z.val  Q.roots := by
      rw [Multiset.subset_iff]
      intro x hx
      rw [Finset.mem_val] at hx
      rw [Polynomial.mem_roots hQ_ne]
      rcases Finset.mem_image.mp hx with n, _, rfl
      exact hroot n
    have h_le : Z.card  Q.natDegree :=
      Polynomial.card_le_degree_of_subset_roots hZ_val_subset
    rw [hZ_card, hd_def] at h_le
    omega
  · intro hP
    subst hP
    refine ?_, ?_
    · simp
    · intro x; simp
```

Listing 5. Baseline refactoring for Putnam 1971 A2.

```
import Mathlib

open Set

abbrev putnam_1971_a2_solution : Set (Polynomial ) := {Polynomial.X}
-- {Polynomial.X}

/-- For a sequence with `a 0 = 0` and `a (n+1) = a n ^ 2 + 1`, every term is nonnegative. -/
private lemma seq_nonneg
    (a :   ) (ha0 : a 0 = 0)
    (ha_succ :  n, a (n + 1) = a n ^ 2 + 1) (n : ) :
    0  a n := by
  induction n with
  | zero => simp [ha0]
  | succ n ih => rw [ha_succ]; nlinarith [sq_nonneg (a n)]

/-- A nonnegative sequence satisfying `a (n+1) = a n ^ 2 + 1` is strictly increasing. -/
private lemma seq_strictMono
    (a :   ) (ha_succ :  n, a (n + 1) = a n ^ 2 + 1)
    (ha_nonneg :  n, 0  a n) (n : ) :
    a n < a (n + 1) := by
  rw [ha_succ]
  nlinarith [sq_nonneg (a n), ha_nonneg n]

/-- If `P(0) = 0` and `P(x^2 + 1) = P(x)^2 + 1`, then `P(a) = a` for the sequence
`a 0 = 0`, `a (n+1) = a n ^ 2 + 1`. -/
private lemma eval_seq_eq_seq { : Type*} [CommRing ]
    (P : Polynomial ) (a :   )
    (h0 : P.eval 0 = 0)
    (heq :  x : , P.eval (x ^ 2 + 1) = P.eval x ^ 2 + 1)
    (ha0 : a 0 = 0)
    (ha_succ :  n, a (n + 1) = a n ^ 2 + 1) (n : ) :
    P.eval (a n) = a n := by
  induction n with
  | zero => simp [ha0, h0]
  | succ n ih => rw [ha_succ, heq, ih]

/-- The multiset underlying the image of a finite set under `f` is contained in the roots of `p`
whenever `p.eval (f i) = 0` for each `i` in the set. -/
lemma image_val_subset_roots {  : Type*} [DecidableEq ] [CommRing ] [IsDomain ]
    {p : Polynomial } (hp : p  0) {f :   } {s : Finset }
    (h_root :  i  s, p.eval (f i) = 0) :
    (s.image f).val  p.roots := by
  intro x hx
  rw [Polynomial.mem_roots hp]
  obtain i, hi, rfl := Finset.mem_image.mp (Finset.mem_val.mp hx)
  exact h_root i hi

/--
Determine all polynomials $P(x)$ such that $P(x^2 + 1) = (P(x))^2 + 1$ and $P(0) = 0$.
-/
theorem putnam_1971_a2
    (P : Polynomial ) :
    (P.eval 0 = 0  ( x : , P.eval (x^2 + 1) = (P.eval x)^2 + 1))  P  putnam_1971_a2_solution := by
  rw [putnam_1971_a2_solution, Set.mem_singleton_iff]
```

```
constructor
· intro h0, heq
  by_contra h_ne
  let a :    :=  n => Nat.recOn n 0 ( _ ak => ak ^ 2 + 1)
  have ha_succ :  n, a (n + 1) = (a n) ^ 2 + 1 :=  _ => rfl
  have ha_nonneg :  n, 0  a n := seq_nonneg a rfl ha_succ
  have ha_strictMono :  n, a n < a (n + 1) := seq_strictMono a ha_succ ha_nonneg
  have ha_inj : Function.Injective a :=
    (strictMono_nat_of_lt_succ ha_strictMono).injective
  have hP_a :  n, P.eval (a n) = a n := eval_seq_eq_seq P a h0 heq rfl ha_succ
  set Q := P - Polynomial.X with hQ_def
  have hQ_ne : Q  0 := by
    intro hQzero
    apply h_ne
    rw [hQ_def] at hQzero
    exact sub_eq_zero.mp hQzero
  set d := Q.natDegree with hd_def
  let Z : Finset  := (Finset.range (d + 1)).image a
  have hZ_card : Z.card = d + 1 := by
    rw [Finset.card_image_of_injective _ ha_inj, Finset.card_range]
  have hZ_val_subset : Z.val  Q.roots := by
    apply image_val_subset_roots hQ_ne
    intro i _
    show Polynomial.eval (a i) (P - Polynomial.X) = 0
    rw [Polynomial.eval_sub, Polynomial.eval_X, hP_a i, sub_self]
  have h_le : Z.card  Q.natDegree :=
    Polynomial.card_le_degree_of_subset_roots hZ_val_subset
  rw [hZ_card, hd_def] at h_le
  omega
· intro hP
  subst hP
  refine ?_, ?_
  · simp
  · intro x; simp
```

Listing 6. Proof-Refactor output for Putnam 1971 A2.

A.2. Evaluation Rubric

The listing below records the rubric used by the LLM-as-a-judge evaluator.

```
===== DETAILED EVALUATION RUBRIC =====

The proof files are assumed to have already passed Lean verification. Do not score correctness. Score only
     refactor quality: structure, helper quality, tactic transparency, reuse, and human readability.

All scores are from 1.0 to 5.0, where higher is better. Half-points such as 3.5 or 4.5 are allowed. Be consistent
     across methods. When comparing benchmark_refactor and proof_refactor_pipeline, calibrate their scores
     relative to each other.

Do not reward theorem count or lemma count by itself. Do not reward length reduction by itself. Penalize fake
     modularity: wrappers, dead helpers, huge helpers, or tactic-golf helpers. Prefer meaningful, named, locally
     checkable mathematical lemmas. A single-use helper is acceptable if it exposes real structure. Broad tactics
     are acceptable only when they close local routine goals, not when they hide the main proof idea.

## structure

Required metrics:

- main_theorem_slimness
- complexity_distribution
- dependency_clarity

Meaning:

- main_theorem_slimness: whether the main theorem has become a readable proof outline rather than a monolithic
     proof body.
- complexity_distribution: whether proof complexity is distributed into meaningful helper lemmas rather than
     hidden in one huge helper.
- dependency_clarity: whether the dependency relation among helpers and the main theorem is clear and shallow.

Scoring guide:

- 5: main proof is slim, helper structure is balanced, dependencies are clear.
- 4: good structure with minor imbalance.
- 3: some structural improvement, but still proof-script-heavy or uneven.
- 2: main proof is shorter mostly by hiding complexity.
- 1: structure is worse or obviously metric-gamed.

Inspection checklist:

- Inspect top-level helper lemmas.
- Inspect private helper lemmas.
- Inspect the main theorem body.
- Inspect local `have` blocks.
- Judge whether the main theorem reads like a proof outline.
- Judge whether helper lemmas carry meaningful proof structure.

## signature_quality

Required metrics:

- statement_naturalness
- binder_economy
- generality

Meaning:

- statement_naturalness: whether helper statements look like meaningful mathematical lemmas.
- binder_economy: whether helper signatures avoid excessive local hypotheses and unnecessary parameters.
- generality: whether helper lemmas are generalized beyond a local proof fragment when appropriate.

Note that: If two helpers' signatures have the same conclusion, but one has more general assumptions, the more
     general one is better and add extra assumptions should be penalized.




Scoring guide:

- 5: helper statements are natural, concise, and reusable.
- 4: mostly natural, with some theorem-specific statements.
- 3: usable but local or proof-script-like.
- 2: many over-specialized helpers.
- 1: helpers are mostly meaningless wrappers or fragments.
```

```
Inspection checklist:

- Judge whether each helper name expresses mathematical content.
- Judge whether each helper statement is natural.
- Judge whether the statement is too local to the original proof.
- Judge whether it has too many binders or hypotheses.
- Judge whether it is more general than the immediate scaffold or merely copied from it.

## tactic_quality

Required metrics:

- tactic_transparency
- explicit_lemma_use
- broad_tactic_control

Meaning:

- tactic_transparency: whether proof bodies use understandable local reasoning rather than opaque closures.
- explicit_lemma_use: whether the refactored proof actually calls named helpers in meaningful places.
- broad_tactic_control: whether broad tactics such as `simp_all`, `aesop`, `grind`, `omega`, or large `nlinarith`
     calls are controlled and not hiding major structure.

Do not penalize all automation. Local uses of `ring`, `linarith`, `norm_num`, `field_simp`, `positivity`, or
     explicit `simp [lemmas]` may be good style. A typical good use case is
to use this tactic to prove some straightforward algebraic manipulation or inequality
or logical tautology but tedious.


Benign or usually acceptable tactics:

```lean
ring
norm_num
field_simp
positivity
linarith [explicit facts]
simp [specific_lemma]
rw [specific_lemma]
```

Potentially broad or opaque tactics:

```lean
simp_all
aesop
grind
omega
large nlinarith without explicit structure
```

Scoring guide:

- 5: explicit lemma use dominates; broad tactics are rare and local.
- 4: some broad tactics, but they do not hide major proof structure.
- 3: moderate opaque closure.
- 2: many helpers are closed by broad tactics.
- 1: proof is mostly tactic golf.

Inspection checklist:

- Judge whether broad tactics are used only for local routine goals.
- Judge whether named helpers are actually called in meaningful places.
- Judge whether the proof hides the main idea behind automation.

reuse

Required metrics:

- helper_usefulness
- reuse_potential
- no_dead_helpers

Meaning:

- helper_usefulness: whether helpers play real structural roles in the proof.
- reuse_potential: whether some helpers could plausibly be reused outside the exact local proof position.
- no_dead_helpers: whether there are no unused, orphan, or purely decorative helpers.

Scoring guide:
```

```
- 5: helpers are clearly useful; several have reuse potential; no dead helpers.
- 4: all helpers are justified; some reuse potential.
- 3: mostly single-use helpers, but structurally useful.
- 2: many single-use helpers with weak semantic value.
- 1: unused or fake helpers are common.

Inspection checklist:

- Check whether helpers are actually used.
- Check whether single-use helpers are structurally meaningful.
- Check whether helpers are reused by multiple parts of the proof.
- Check whether any helpers are unused or merely decorative.
- Check whether any helpers are wrappers around another lemma without real abstraction.

## human_review

Required metrics:

- proof_readability
- mathlib_style
- maintainability

Meaning:

- proof_readability: whether a human can read the proof as a clear mathematical argument.
- mathlib_style: whether names, locality, statement shape, binder order, and tactic usage are close to Mathlib
     style.
- maintainability: whether future changes would likely be localized and easy to repair.

Mathlib style is general as much as possible, if you generalize type or typeclass assumptions, e.g form `f :   `
     to `f :   ` with some typeclass assumptions, it is better. And if you find the proof use more mathlib lemmas
     for those new proof, it is better.

Scoring guide:

- 5: very readable, Mathlib-like, easy to maintain.
- 4: good proof-engineering style.
- 3: acceptable but still somewhat ad hoc.
- 2: hard to maintain.
- 1: barely readable despite compiling.

Inspection checklist:

- Judge whether a human reader can recover the mathematical proof outline.
- Judge whether names, binder order, locality, and statement shape are close to Mathlib style.
- Judge whether future proof repairs would likely be localized.

## overall

Compute overall.score as the simple average of all 15 non-overall metric scores for the method, rounded to 2
     decimals.

## Output requirement

Return only the required JSON object. Each method must include a `reason` string. The `reason` should be concise
     but substantive, usually 3-6 sentences, explaining why the method received its scores. Do not put Markdown
     outside the JSON.
```

Listing 7. Evaluation rubric.