LeanMarathon: Toward Reliable AI Co-Mathematicians through Long-Horizon Lean Autoformalization

Yuanhe Zhang* Yuekai Sun† Taiji Suzuki‡ Jason D. Lee§ Fanghui Liu¶

Abstract

Long-horizon autoformalization of research mathematics fails not only at hard lemmas, but at scale: statements drift, dependencies tangle, context decays, and local repairs corrupt distant work. We present LeanMarathon, a multi-agent harness for reliable research-level Lean autoformalization. Its core abstraction is an evolving blueprint: a Lean file that serves simultaneously as formal proof skeleton, natural-language proof graph, and shared system of record. Four contract-scoped agents construct, audit, prove, and repair this blueprint. These agents are coordinated by a two-stage orchestrator that first stabilizes target fidelity through adversarial review and then discharges the proof directed acyclic graph (DAG) from its dynamic leaves upward in parallel CI-gated rounds. LeanMarathon turns one brittle multi-hour run into many local, recoverable, parallel transactions. We evaluate LeanMarathon on two recent research papers spanning four Erdős problems (#1051, #1196, #164, #1217). Across three autonomous runs, it formalizes all seven target theorems with no `sorry`, proving 258 lemmas and theorems. These results show that reliable AI co-mathematics requires not only stronger provers, but durable harnesses that preserve target fidelity across long mathematical developments. The code can be found at `https://github.com/YuanheZ/LeanMarathon`.

1 Introduction

AI-assisted mathematics can be organized into three interlocking stages, as emphasized by Tao (2026): proof *generation* large language models (LLMs), *verification* via Lean 4, and *digestion* by human. Generation by LLMs has advanced fast, such as Aletheia (Feng et al., 2026; Zheng et al., 2026) and GPT-5 (Bubeck et al., 2025). These agents can produce long natural-language proofs and solve open problems, subject to human verification. Verification, the stage that turns such a natural language proof into a machine-checked artifact, has kept pace only on isolated goals: a prover discharges one or multiple Lean 4 lemmas at a time. Verifying an *entire research paper* remains largely open: every definition, lemma, and theorem must be formalized so the whole argument type-checks with no `sorry`.

The difficulty is therefore not merely the formalization of one hard lemma. A research paper may require hundreds of mutually dependent declarations, and these declarations must remain coherent while an autonomous system repeatedly edits, checks, and repairs a growing Lean development. A local change to a definition can invalidate distant proofs; a misformalized intermediate lemma can make downstream work formally correct but mathematically irrelevant; and a seemingly successful repair can silently move the formal

*Department of Statistics, University of Warwick, UK; also Center for Advanced Intelligence Project, RIKEN, Japan. Email: `yuanhe.zhang@warwick.ac.uk`

†Department of Statistics, University of Michigan, USA. Email: `yuekai@umich.edu`

‡Department of Mathematical Informatics, The University of Tokyo; also Center for Advanced Intelligence Project, RIKEN, Japan. Email: `taiji@mist.i.u-tokyo.ac.jp`

§Department of Electrical Engineering and Computer Sciences, also Department of Statistics, University of California, Berkeley, USA. Email: `jasondlee@berkeley.edu`

¶School of Mathematical Sciences, Institute of Natural Sciences and MOE-LSC, Shanghai Jiao Tong University, China. Email: `fanghui.liu@sjtu.edu.cn` (Corresponding author)

proof away from the intended theorem. Long-running autoformalization thus fails in ways that resemble software-engineering failure as much as theorem-proving failure: stale context, dependency tangles, statement drift, and repairs whose effects are hard to localize.

Automating this verification stage is important for two reasons. First, machine-checked formalization, e.g., Lean 4, is the strongest available guarantee that an AI-discovered proof is correct rather than merely plausible. It is therefore a prerequisite for making AI-assisted mathematics reliable enough to ***accelerate research at scale***. Second, formalization can also support proof digestion. Since a Lean proof records a proof at a very fine level of detail, it can be translated back into natural language at multiple resolutions: a high-level overview for orientation, a structured proof outline for learning, and a fully detailed derivation for verification. This suggests a future form of mathematical textbook in which the same formal artifact supports both machine checking and human-facing explanations at different levels of granularity.

For research-level formalization, the core bottleneck is formalization shifts. The limiting factor is not (just) the model's capability on a single goal but its *agent durability*: whether an autonomous system stays coherent across a multi-hour run, preserves the intended target, calibrates the failure state, and keeps one wrong decision from corrupting the rest of the proof. This differs significantly from textbook formalization (Wang et al., 2026; Gloeckle et al., 2026) that provides the agent with a fine-grained blueprint of the full reasoning pipeline then the agent's core task is to translate this existing blueprint into formal Lean code, rather than to discover the logical structure of the reasoning itself. In research-level formalization, no such blueprint is available in advance. The only fixed anchors are the terminal target theorems specified by human researchers. The source proof, whether written by humans or generated with AI assistance, may contain noise, implicit steps, gaps, or even errors.

This absence of a trusted fine-grained proof plan creates two pervasive failure modes. The first is *goal drift*, where the agent's intermediate reasoning gradually deviates from the logical path required to prove the target theorem, resulting in a formally correct but irrelevant reasoning graph; The second is *lost-in-the-middle*, where the agent becomes trapped in an exponentially growing space of unproductive subproblems, unable to navigate back to the core target or prioritize high-impact intermediate steps. Both failure modes are relatively muted in textbook settings where the proof graph is supplied, but they become central in research settings where the system must discover, maintain, and repair the graph itself.

A monolithic agent asked to read the paper, design the formal skeleton, prove the lemmas, diagnose failures, and repair its own mistakes is fragile. A single defect such as missed hypothesis, drifted statement, or confident but wrong repair can invalidate hours of downstream work with high risk of context rot. The question addressed in this paper is therefore not simply how to make an LLM prove a Lean goal, but how to design a multi-agent harness that makes long-horizon Lean autoformalization of research mathematics legible, recoverable, and resistant to drift. More generally, our work explores the possibility of autoformalization of research mathematics via harness design if the related Lean library (e.g., Mathlib) is sufficient and complete.

1.1 Contributions

We propose LeanMarathon, a multi-agent harness for long-horizon Lean autoformalization of research mathematics, exemplified by Erdős problems (#1051, #1196, #164, #1217) from two recent papers (Barreto et al., 2026; Alexeev et al., 2026). At its center is the *blueprint*: a single Lean file that is at once a formal proof skeleton and a natural-language proof graph. Each `lemma` or `theorem` is a *node* pairing a Lean type with the LaTeX statement and proof it formalizes; when one node's proof invokes another, it induces a directed dependency edge. These nodes and edges form the *proof directed acyclic graph (DAG)* that every agent reads, extends, and repairs (definitions are shared global context, not nodes). The formalization is complete when every node carries a Lean proof and the file type-checks with no `sorry`. We expect the proof DAG as well as harness design for long-horizon autoformalization follows four principles:

- **Decompose with dynamic proof DAG.** The initial decomposition is uncertain, so the system never freezes it. It generates the DAG and then lets it evolve, splitting an over-large node or repairing a misformalized one, until each piece is small enough for one agent to discharge.
- **External or deterministic verification.** No agent judges its own output. The progress is decided by the the deterministic verifier and external agents, not by an agent's self-assessment, which avoids potential formalization drifts.

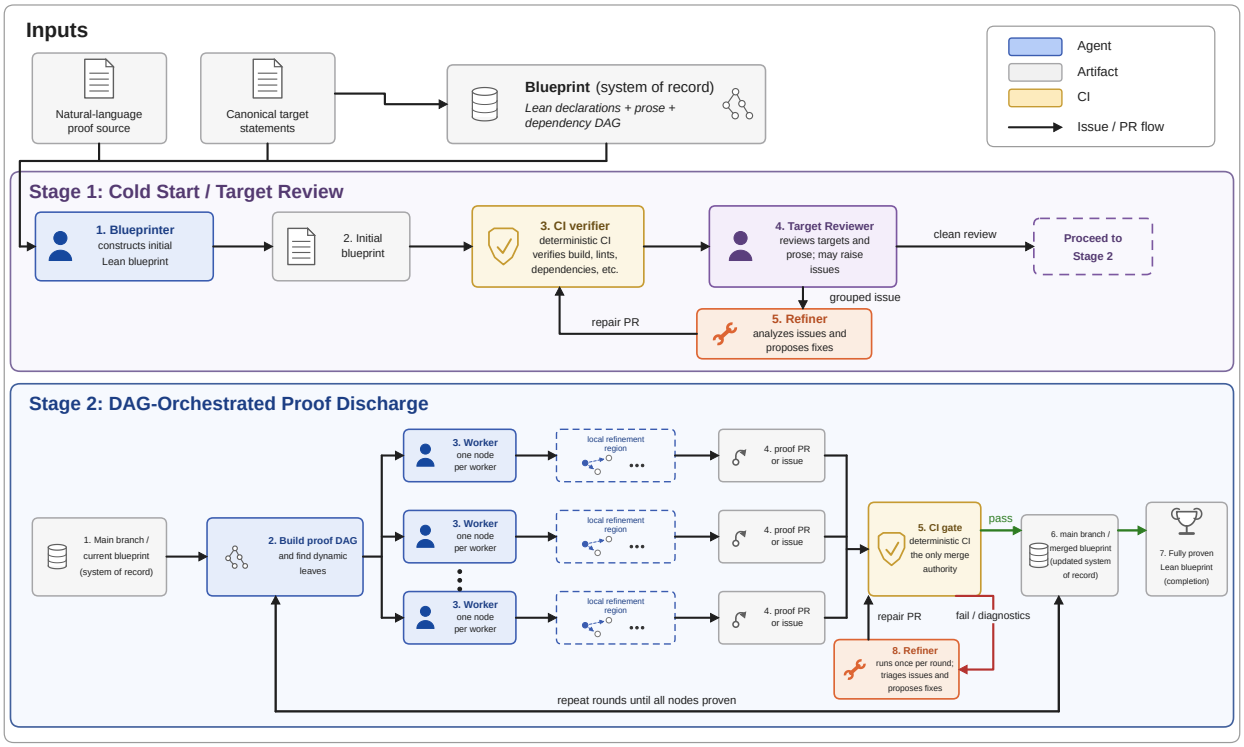

Figure 1: Overview of LeanMarathon.

- **Restrict tools scope.** Each agent is expected to edit only a bounded region. Constraining the action space calibrates the agent's trace, so the worst outcome of a mistake is a rejected patch rather than corruption of another agent's state.
- **Informalize while formalizing.** Each node keeps its LaTeX prose beside its Lean type. This keeps the artifact human-readable and also guards against drift: the verifier enforces parity between the prose dependency graph and the elaborator's, so the natural-language and formal graphs cannot silently diverge.

Fig. 1 shows how LeanMarathon instantiates these principles. Four contract-scoped agents share the blueprint, each with a narrow input, a single output, and a bounded edit scope. The Blueprinter reads the source proof and the canonical target statements and writes the initial skeleton, with every statement type-checking and every proof body a `sorry` placeholder. The Target-Reviewer audits that skeleton for fidelity, checking that each Lean type states the theorem the paper intends, and files an issue on any mismatch. The Worker discharges one node behind a quality gate, proving its body inside a frozen local region. The Refiner collects open issues and repairs multi-node defects in one pull request (PR) per round.

These agents are coordinated by a two-stage orchestrator. Stage 1 runs an adversarial review loop between the Target-Reviewer and Refiner until the target statements are certified as faithful. Stage 2 repeatedly extracts the current proof DAG, identifies dynamic leaves whose dependencies have already been proved, and assigns them to Workers in parallel. All pull requests pass through a deterministic CI gate before reaching `main`; passing non-conflicting PRs are squash-merged independently as their checks complete. The contributions of this paper can be summarized as below.

- We identify **agent durability** as the central bottleneck in research-level autoformalization, and characterize three failure modes any long-running agent exhibits. *Coherence loss*: the task is globally coupled, like Sudoku, where every step must respect the whole proof, so a monolithic agent turns myopic and robs one part of the proof to patch another. *Self-evaluation bias*: asked to evaluate its own

output, the agent appears to be overly confident. *Irreversibility*: once the work drifts from the target, the agent cannot recover it, and the damage compounds across the rest of the proof. A single agent will always make such errors, so the design goal is not an infallible agent but a system of fallible agents in which no one error spreads: **fault containment**. We realize it through four contract-scoped agents and a two-stage orchestrator that separates adversarial target review from parallel proof discharge, turning one brittle multi-day run into many short, recoverable, parallel ones.

- Our core formulation is the **dynamic proof DAG**: one blueprint that is at once a formal Lean skeleton, a natural-language proof graph, and the shared system of record. The harness never freezes the initial decomposition; instead it grows and repairs the DAG and discharges it from its dynamic leaves upward in parallel. Holding this moving graph coherent is the job of the whole harness, enforced concretely by **deterministic CI gate** and encoded into the contracts of agents.

- We evaluate LeanMarathon on **research-level Erdős problems** for autoformalization. Across two 2026 papers spanning four Erdős problems (Barreto et al., 2026; Alexeev et al., 2026), LeanMarathon formalizes all seven target theorems with no `sorry`, proving 258 lemmas and theorems in total. A commercial single-agent baseline (Achim et al., 2025) fails on both papers after tens of hours. LeanMarathon localizes failures to individual nodes and keeps incorrect early formalizations from silently consuming downstream prover compute.

1.2 Related Work

Autoformalization and neural theorem proving. Early systems prompt an LLM to translate statements (Wu et al., 2022) or to draft informal proofs into formal sketches (Jiang et al., 2023), and later work scales paired data and Lean-specific translators (Ying et al., 2024; Gao et al., 2025). In parallel, neural provers search for proofs: tactic models trained on proof data (Polu & Sutskever, 2020; Han et al., 2022), tree search (Lample et al., 2022), premise retrieval (Yang et al., 2023), interleaved informal reasoning (Lin et al., 2025a), reinforcement learning from proof-assistant feedback and subgoal decomposition (Xin et al., 2024; Ren et al., 2025), self-correction from compiler errors (Lin et al., 2025b), and large reasoning provers (Wang et al., 2025; Chen et al., 2025). The targets which can be formalized by these systems are at most IMO-level.

AI for research-level mathematics. Recent systems push AI from competition problems toward open research questions. AlphaProof reached olympiad-medal level by proving in Lean with reinforcement learning (Hubert et al., 2026), while AlphaGeometry (Trinh et al., 2024) and AlphaEvolve (Novikov et al., 2025) attack open problems through specialized or evolutionary search, though their outputs are constructions and programs, not machine-checked proofs. Benchmarks confirm the distance to research difficulty (Glazer et al., 2024). General models now contribute steps to live mathematics: a Gemini system surveyed hundreds of Erdős problems (Feng et al., 2026) and GPT-5 produced new results with mathematicians (Bubeck et al., 2025). Both episodes drew scrutiny over whether claims were proved or merely retrieved, which is exactly what a formal proof settles (Tao, 2025; Bloom, 2026). AlphaProof Nexus (Tsoukalas et al., 2026) is an evolutionary Lean proof-search system: prover subagents edit marked regions of Lean sketches, invoke AlphaProof on subgoals, validate candidates, and use rater agents with a population database to select promising sketches, resolving 9 of 353 Erdős problems and 44 of 492 OEIS conjectures. In contrast, LeanMarathon targets paper-level autoformalization: it builds an audited blueprint DAG from source and proves the resulting multi-result Lean development bottom-up.

Autonomous formalization agents and large-scale formalization. A wave of 2025 and 2026 systems formalize at the scale of whole results or papers, the regime LeanMarathon targets. AxiomProver (Axiom Math, 2025), Gauss (Hariharan et al., 2026), and Aristotle (Achim et al., 2025) are capable to formalize papers but they are fully close-source company products. However, we have no clues about such harness design. Besides, large-scale community projects are growing: the Polynomial Freiman-Ruzsa conjecture (Dillies et al., 2023), the Liquid Tensor Experiment (Scholze, 2022), Fermat's Last Theorem (Buzzard et al., 2024), Carleson's theorem (Becker et al., 2024), the Equational Theories Project (Bolan et al., 2025), statistical learning theory (Sonoda et al., 2025; Zhang et al., 2026b), and reinforcement learning theory (Zhang, 2025).

We present LeanMarathon in two parts: its infrastructure design in Section 2 which introduces the blueprint and the agents that act on it, and then the orchestration that drives them to be a multi-hour, parallel autoformalization recoverable and resistant to drift in Section 3. Section 4 provides the experimental evaluations. The conclusion is drawn in Section 5.

2 Harness Infrastructure

This section covers our harness's fundamental ingredients. Section 2.1 describes the blueprint format we use. Section 2.2 then introduces the four contract-scoped agents that maintain the blueprint, each owning a well-separated task, skilled workflow, and bounded edit scope.

2.1 The Blueprint as the System of Record

The central artifact in our harness is the blueprint, a Lean file, that serves simultaneously as a formal proof skeleton, a natural-language proof graph, and the task interface exposed to agents. All durable mathematical state is stored in this file: theorem statements, intermediate lemmas, definitions, prose explanations, and declared proof dependencies. Agents may read, prove, review, or repair parts of the blueprint, but they do not maintain any hidden shared memory outside it.

Each mathematical node in the blueprint is represented by a Lean declaration annotated with structured metadata, in the format of LeanArchitect (Zhu et al., 2026): the blueprint metadata lives in an in-source `@[blueprint ...]` attribute, and the dependency graph and `sorry`-status are inferred from Lean's elaborator. This complements the original blueprint methodology (Massot, 2020). A typical proof node has the form:

```
@[blueprint "lem:weighted-tail-bound"
  (statement := /-- LaTeX statement text -/)
  (proof     := /-- LaTeX proof prose with \cref{...} citations    -/)
  (title     := /-- one-line LaTeX title                           -/)
  (latexEnv  := "lemma")]
lemma weighted_tail_bound ... : ... := by
  sorry_using [aux_lemma_one, aux_lemma_two]
```

The Lean declaration gives the formal statement that must type-check. The `statement` field records the corresponding mathematical statement in LaTeX. The `proof` field records the natural-language proof sketch, including explicit references to earlier nodes. The `title` and `latexEnv` fields provide presentation metadata and allow the verifier to check that the Lean declaration and the prose environment agree.

Proof nodes may have one of three proof-body states: unproved, unproved but with a dependency list, and proved. A node may be unproved, written as `by sorry`; it may be unproved but equipped with an intended dependency list, written as `by sorry_using [...]`; or it may contain a complete Lean proof. Definitions are treated as global context and are not proof nodes in the dependency DAG. The proof DAG is formed only from lemma and theorem declarations. Note that, `\cref{...}` citations in the LaTeX prose are not decorative: the verifier extracts the actual proof dependencies via Lean's elaborator metadata and enforces *two-way* parity between the `\cref` edges and the elaborator's edges. The blueprint obtains the property that the natural-language graph and the typed graph cannot drift apart: a PR that lets either side disagree with the other is rejected before it merges.

2.2 Contract-Scoped Agents

Our harness decomposes research-level autoformalization into four contract-scoped agents: the Blueprinter, the Target-Reviewer, the Worker, and the Refiner. Each agent owns a narrow interface, a bounded edit scope, and a specific failure mode that it is designed to expose or contain. Table 1 summarizes the four agents along these axes.

Table 1: The four contract-scoped agents. Only the BLUEPRINTER and REFINER are given the source proof; the TARGET-REVIEWER and WORKER see only the canonical statements and the blueprint. Every PR reaches `main` only through the CI verifier.

Agent	Input	Output	Allowed edits	Failure mode
BLUEPRINTER	source proof, canonical statements, blueprint	PR	writes the whole skeleton, bodies as placeholders	poor decomposition, i.e. a large repair radius
TARGET-REVIEWER	canonical statements, blueprint	issue/None	none (read-only)	a misformalized target: a valid but wrong theorem
WORKER	canonical statements, blueprint	PR/issue	the node's prose, its proof body, its local refinement region	a local proof failure, or silently proving a misformalized node
REFINER	source proof, canonical statements, open issues, blueprint	PR	one connected illness sub-DAG	blueprint drift and source gaps

This separation is essential: a single agent asked to design the proof graph, judge its own fidelity, discharge proofs, and repair global errors has no reliable mechanism for detecting its own drift. In our harness, every agent produces an externally checkable artifact, and every artifact is accepted only through the verifier.

2.2.1 BLUEPRINTER

The BLUEPRINTER converts the source proof and the canonical target statements into the initial blueprint. Its job is not to prove the paper but to choose a decomposition of the argument that is faithful enough for later review and local enough for later repair. A good blueprint should isolate mathematical commitments so that, if one statement is later found to be misformalized, the number of downstream declarations that must be changed is small.

We frame the BLUEPRINTER's job as a *repair-radius optimization problem*: draft an initial blueprint which minimizes the expected number of declarations that must change if one declaration turns out to be wrong. The decomposition follows a published rubric via `decomposition.md`. The BLUEPRINTER writes every proof body as `sorry` or `sorry_using`, and delivers a single PR to merge the blueprint into the `main` branch. The agent is supposed to exit after PR creation, then the **stop hook** is triggered to monitor the merge status. If merge fails, the stop hook will block the exit, serve a resume instruction with CI run's job logs. Mathematical repair of the source is explicitly out of scope, the agent's job is to *isolate* it, not to *fix* it. This boundary design provides a clear separation with later REFINER and allocate more agent's workloads for decomposition. The in-workspace knowledge-store layout of BLUEPRINTER is given in Fig. 6 from Appendix A.

2.2.2 TARGET-REVIEWER

The TARGET-REVIEWER audits the blueprint before large-scale proving begins. Its role is to prevent the most expensive failure mode in research-level autoformalization: proving a formally valid Lean theorem that is not the theorem intended by the source paper. For instance, if a root `theorem` statement is misformalized, every `lemma` the WORKER proves afterwards is wasted compute, so an early certification at the roots is what makes later orchestration loop more productive.

For every target theorem, the REVIEWER compares three objects: the canonical target statement, the LaTeX statement stored in the blueprint, and the Lean type. It checks whether they express the same mathematical claim, with the same hypotheses, quantifiers, definitions, and conclusion.

The REVIEWER is not allowed to edit the blueprint. A clean review allows our harness to enter the proof-discharge stage. A failed review produces a grouped issue describing the suspected mismatch and the affected nodes; the issue is then handled by the REFINER.

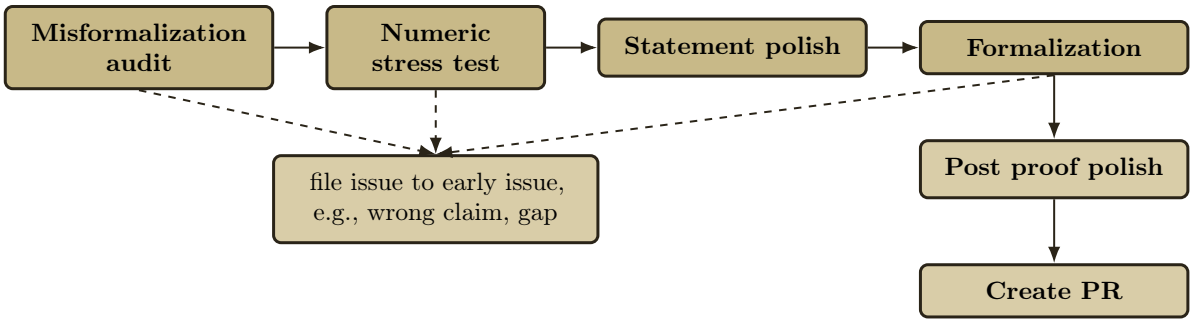

Figure 3: The executable workflow of per-node Worker.

2.2.3 Per-node Worker

A Worker is assigned one proof node whose declared dependencies have already been proved. Its job is local: either prove the assigned Lean statement or report why the node should not yet be proved. The Worker proceeds through four ordered phases, see Fig. 3.

Phase 1: Misformalization audit. The Worker first treats the Lean type as a suspect specification. It compares the Lean statement with the blueprint prose, checks why the node exists, identifies which downstream nodes use it, and verifies that the statement provides the fact those downstream nodes need. If the type is missing a hypothesis, states the wrong conclusion, uses an unsuitable abstraction, or does not match its intended role in the proof DAG, the Worker stops and files an issue.

Phase 2: Cheap falsification. When the claim admits finite, numerical, or boundary-case testing, the Worker attempts to refute it before spending prover compute. Failure to find a counterexample is not treated as a proof; it is only a low-cost sanity check. A discovered counterexample or suspicious boundary case is reported as an issue.

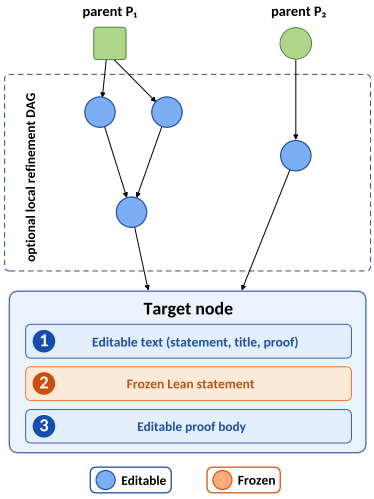

Figure 2: Expected editing behaviour.

Phase 3: Statement polish. If the Lean type passes the first two phases, the Worker may edit only the node's prose fields: the LaTeX statement, title, and proof text. The goal is to make the natural-language text describe the Lean statement exactly, neither stronger nor weaker.

Phase 4: Formalization. The Worker then attempts to replace the placeholder body with a complete Lean proof. The Lean type of the assigned node is frozen throughout this phase. The Worker may introduce fresh helper lemmas only inside the local refinement region immediately before the target node. These helpers must precede the target, depend only on already visible declarations or earlier local helpers, and terminate at the assigned target. If Worker cannot complete the formalization within all boundaries, then it needs to file an issue with clear evidence to exit.

During formalization, The Worker's edit scope is mechanically enforced via an editing MCP server built by patching Codex's `apply-patch` tool, illustrated in Fig. 2. It may edit the assigned node's prose fields, its proof body, and its local refinement region, but not the Lean type of the target or unrelated blueprint nodes. This restriction makes parallel formalization safe: different Workers operate on disjoint editable regions, so successful patches commute by construction, while failed attempts become rejected local patches or diagnostic issues rather than global corruption. To be specific, the Lean file is partitioned around the assigned target node T into frozen and editable spans:

```
-- previous node ends here
-- BEGIN editable local refinement area for T
@[blueprint "lem:X" ...]
lemma X ... := by
...

@[blueprint "lem:Y" ...]
lemma Y ... := by
...
-- END editable local refinement area for T
@[blueprint "lem:T" -- frozen
  (statement := /-- editable -/)
  (proof := /-- editable -/)
  (title := /-- editable -/)
  (latexEnv := "lemma")] -- frozen
lemma T ... : ... := by -- frozen
-- editable proof body
```

This harness design makes the parallel multi-agent loop against the same frozen substrate commit works. Workers acting on disjoint editable regions produce patches that commute by construction so PRs can land in any order without merge conflict. Within its editable region, the Worker may *grow a local refinement DAG* consisting of fresh helper nodes before the target node which is the unique terminal, which aims to overcome the possible risk of under-decomposition brought from Blueprinter or Refiner.

2.2.4 Refiner

The Refiner repairs blueprint-level defects reported by the Target-Reviewer or Workers. Unlike a Worker, which is restricted to one proof node, the Refiner can edit the whole blueprint since a defect might affect multiple declarations.

Given the open issue(s), the Refiner first identifies the affected region, called the **illness** area: the smallest connected sub-DAG that must be inspected or changed to resolve the defect. We develop an MCP server called `dag-tracker` for agent to call for live parent/child identification.

The Refiner classifies each defect as either **blueprint drift** or a **source gap**. Blueprint drift means that the Lean blueprint has diverged from the source proof: for example, a statement was misformalized, a dependency was wrong, or the prose no longer describes the Lean declaration. A source gap means that the source proof itself is incomplete, ambiguous, or false at the level required for formalization. During the repair phase, the proof bodies are governed by the following decision tree node by node:

```
Is the node NEW or EXISTING?
├─ NEW: body as placeholder
└─ EXISTING: what is the current proof body's shape?
   ├─ placeholder:  keep and align with new dependency set if prose changed.
   └─ complete tactic proof:  still compiles after repair?
      ├─ YES: preserve the complete proof body byte-identical to the input blueprint.
      └─ NO: wholesale-replace with placeholder
```

Compilation is decided by Lean compiler, never by the agent. Wholesale replacement is recorded in the PR summary and never negotiated. There is no path that lets the Refiner partial-edit a complete proof body. Although a downstream Worker needs to prove the downgraded node again, the discipline keeps the Refiner's blast radius proportional to the issues it is closing, not to the file it is editing. Furthermore, the source gap should be fixed via reasoning and the new solution should be updated in the LaTeX fields.

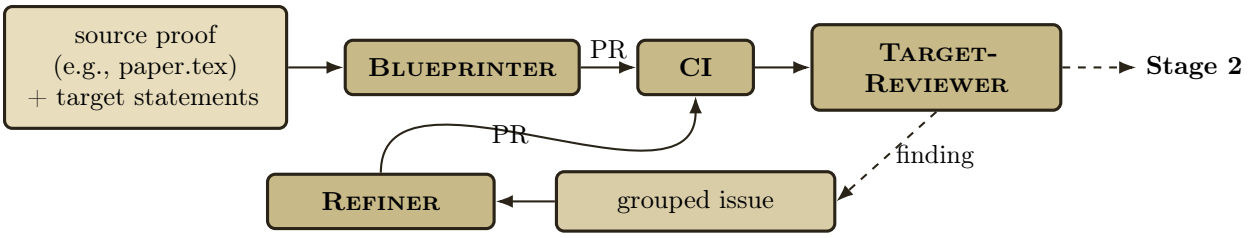

Figure 4: The orchestration of stage 1.

3 System Orchestration

After introducing harness infrastructure, we are ready to discuss our system orchestration in two stages. The first stage constructs and audits a faithful blueprint before formalization begins. The second stage discharges the proof DAG through parallel, CI-gated proof attempts.

3.1 Stage 1 – Cold Start and Target Review

We frame this stage as a **nested Ralph-Wiggum loop**, shown in Fig. 4, which is designed to digest the input natural language source into an initial Lean blueprint.

The Blueprinter produces an initial Lean blueprint in which all declarations elaborate and all proof bodies are placeholders which must pass the CI gate. The Target-Reviewer then compares the theorem nodes against the canonical targets and the blueprint's own LaTeX and Lean.

If the review is clean, our harness proceeds to Stage 2. If the Reviewer finds a target mismatch, missing hypothesis, incorrect dependency, or other blueprint-level defect, it files a grouped issue. The Refiner repairs the affected region and submits a repair PR. After the PR passes CI and merges, the Reviewer runs again. Stage 1 terminates only when the target review exits clean.

3.2 Stage 2 – DAG-orchestrated Loop

Stage 2 proves the blueprint via the pipeline in Fig. 5. Each round starts from the current `main` branch, which is the system of record. The orchestrator extracts the proof DAG from the blueprint and identifies the current dynamic leaves: unproved proof nodes whose dependencies have already been proved.

The orchestrator assigns each dynamic leaf to an independent Worker. Every Worker receives the same frozen substrate commit and a mechanically restricted editable region around its target node. Workers run in parallel. A successful Worker submits a proof PR; otherwise it files an issue instead.

Proof PRs are accepted only by the CI gate.Passing PRs are merged into main; failing PRs are rejected with diagnostics. After all Workers in a round have finished, the Refiner processes the accumulated issues, submits repair PR through the same CI gate, and updates the blueprint if the repairs pass.

Our harness repeats these rounds until every proof node in the DAG has a complete Lean proof. At termination, the main branch contains a fully proven blueprint with no remaining placeholders.

Why direct merge, not Github auto-merge. The orchestrator merges via `gh pr merge -squash` directly, not `gh pr merge -auto -squash`. Auto-merge requires a pending required check (branch protection) and serialises across parallel PRs, i.e., only the first PR to finish CI can enable auto-merge, and a later parallel PR finds `main` has moved and the GraphQL mutation refuses. Direct merge needs no pending check and lets every non-conflicting parallel PR land independently as its own CI completes.

3.3 Sustaining hours-long runs

The following composable properties keep a multi-hour run resumable without drift.

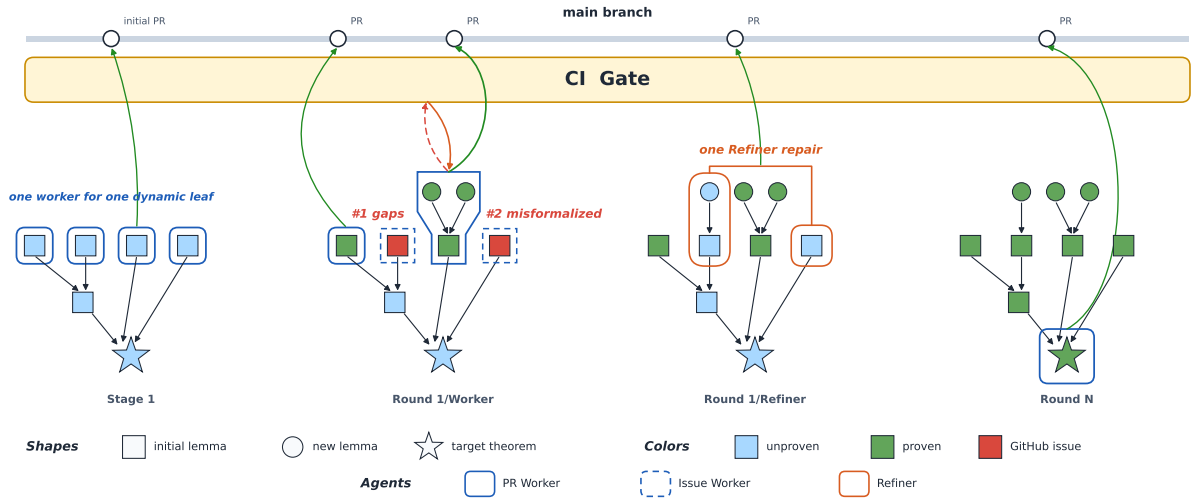

Figure 5: The DAG-based orchestration of stage 2.

CI verifier as continuous integration The CI verifier is the single gate for any merge request to `main` branch. The verifier encodes the blueprint contract (`blueprint-format.md`) as the following seven executable checks:

1. **Lean compilation**: every diagnostic must be either none, or `declaration uses 'sorry'` warning.
2. **Node well-formedness**: every `@[blueprint]` attribute has non-empty `statement` / `title` / `proof` fields. Placeholders are multi-line `by` followed by `sorry` or `sorry_using`. Incomplete proof body is prohibited.
3. `latexEnv` **consistency:** the Lean keyword and the `latexEnv` field must agree (e.g. `lemma` ↔ `lemma`, `theorem` ↔ `theorem`).
4. **Label-name normalization**: blueprint labels like `lem:foo-bar` normalize (`-` → `_`) to the actual Lean name.
5. **Unique labels**: each blueprint node's Lean naming should be unique.
6. **Proof-dependency parity**: Based on the proof DAG exatracted via Lean's elaborator metadata, a *two-way* parity then requires every Lean dep to be `\cref`-cited in the prose and every `\cref{lem:_}` or `\cref{thm:_}` to be a Lean dependency. `sorry_using` references additionally must precede the citing node in file order and must point at proof nodes only. We treat the definitional nodes as global context which are intentionally excluded from the proof DAG.
7. **Lemma closeness**: every `lemma` must be cited by some later `lemma` or `theorem`; a `theorem` is a target claim and is treated as a terminal sink, allowed to have no children. Equivalently, every non-terminal node of the proof DAG must have positive out-degree toward a target, so no `lemma` is an *orphan* that feeds nothing. Inspired by the *logical closeness* defined by Zhang et al. (2026a), which requires every non-sink node of a reasoning DAG to have positive out-degree, this check is a *structural* guard against task drift: an orphan lemma is exactly the footprint a drifting agent leaves when it proves machinery absent from the source, the failure mode that stalled our earlier harness (Section 4.7).

Either successful or failed verification triggers an upserted PR comment for live feedbacks. These checks are deliberately structural rather than stylistic/semantic. Lemma closeness in particular is the structural counterpart to feeding the Refiner the source proof (Section 2.2.4): source-anchored repair realigns proof

content to the paper, whereas lemma closeness enforces goal-directedness on the dependency *graph* alone, mechanically and without reading the mathematics.

This harness does not try to prescribe the exact tactic script a WORKER should discover; instead, it enforces the interfaces that make independent agents composable. Formal statements must compile, prose citations must match proof dependencies, graph edges must be visible and acyclic, and every pull request must preserve the blueprint as a coherent system of record. This is the main methodological advantage of our harness: it converts long-running agent work from an opaque sequence of prompts into a set of small, recoverable, mechanically checked transactions.

Stop-hook-driven self-recovery. Each PR-enabled agent's stop hook runs after the agent declares itself done. The hook validates `delivery.yml` as the terminal-delivery record; if the agent opened a PR, the hook polls GitHubfor either a successful merge or a CI failure. On a CI failure the hook extracts the verifier's upserted comment and the raw failed-job logs as context inside the worktree, and blocks the agent with a templated debug-fix-push instruction.

Bounded scope, deterministic enforcement. Every agent receives a contract that names the spans of the workspace it may edit. The contracts are not advisory: the `apply-patch` MCP server rejects every patch to a frozen span, the `read-only` Codex sandbox rejects every constructive operations, and the CI path-allowlist rejects every PR that touches a path outside blueprint. The principle averts the failure mode in which one agent's mistake silently corrupts another agent's working state. The worst outcome any misbehaving agent can produce is a rejected patch, never a poisoned PR.

Context management. All inter-agent communication flows through the on-disk Lean blueprint and the GitHub PR/issue stream. There is no scratch memory, no shared in-RAM channel, no out-of-tree file written by one agent and read by another. The principle averts *cross-agent context bleed*: with no hidden channel, an agent that misbehaves cannot poison another agent's context, and a Codex auto-compaction that wipes one agent's window does not corrupt anything any other agent will run on.

4 Experiments

We test LeanMarathon on two research papers (Barreto et al., 2026; Alexeev et al., 2026) on Erdős problems. Each input is the paper's LaTeX source together with a separate file of canonical target statements; each output is a Lean blueprint in which every proof node is proven (Section 2). The harness formalized all seven target theorems across the two papers, covering four Erdős problems (#1051, #1196, #164, #1217), with no `sorry` and under the seven-check CI contract of Section 3.3. It discharged 258 lemmas and theorems in total. Aristotle (Achim et al., 2025), the only commercial Lean agent we could access, failed on both papers after tens of hours.

4.1 Two research papers for evaluation

We chose two papers that are recent, research-level, and AI-assisted in their genesis. Difficulty rules out competition-style targets: both papers are analytic number theory with multi-page estimates, not some tricks. AI provenance makes formalization the decisive artifact: both proofs were *discovered* with AI help, and an AI-discovered proof is exactly the case where a machine-checked formalization, rather than prose, is the guarantee that the argument is correct.[1]

Erdős–Graham irrationality (Barreto et al., 2026). This paper answers Erdős Problem #1051, posed by Erdős and Graham in 1980 (Bloom, 2026): a double-exponential growth condition forces $\sum_n 1/(a_n a_{n+1})$ to be irrational. The headline case was solved autonomously by the agent *Aletheia* (Gemini Deep Think), one of four Erdős problems DeepMind reported solving autonomously in December 2025 (Feng et al., 2026). We

[1] In the broader Erdős-problem sweep that produced one of our targets, DeepMind reported that of 200 candidate solutions marked correct or incorrect, only 6.5% were "meaningfully correct" (Feng et al., 2026). This gap is what a `sorry`-free Lean proof closes.

Table 2: Formalization outcomes. Proof nodes counts `lemma` and `theorem` declarations; definitions are global context. The `Prim` row includes the 59-node #1196 blueprint reused as its seed.

Metric	Erdős–Graham (`ErdosGraham`)	#1196 (`Erdos1196`)	#164 & #1217 (`Prim`)
Target theorems	4	1	2
Lean lines	8,513	3,988	14,592
Nodes (def / lem / thm)	39 / 106 / 5	15 / 43 / 1	57 / 144 / 3
Proof nodes	111	44	147
Remaining `sorry`	0	0	0
Status	**complete**	**complete**	**complete**

stress that #1051 is the *easy* part. Its $d = 2$ golden-ratio instance follows from a short Borel-peak argument; the depth of the paper lies in two later results that we also formalize. The *general* irrationality theorem fixes a weight tuple $\mathbf{w}$, defines a growth threshold $c_{\mathbf{w}}$ as the unique positive root of a polynomial $P_{\mathbf{w}}$, and proves irrationality of a weighted series via Mahler's criterion, a local-peak selection lemma, and a three-regime tail analysis. The *construction* theorem is its sharp negative counterpart: for every $C > 1$ it builds an integer sequence whose weighted sum is rational, through a non-constructive nested-interval covering argument. The original question already drew formalization attention, but these two generalizations were unformalized.

Primitive sets and von Mangoldt chains (Alexeev et al., 2026). This paper introduces a method, suggested by output of GPT-5.4 Pro, that bounds Erdős sums of primitive sets using Markov chains with von Mangoldt weights on the divisibility poset. The single method resolves several conjectures at once. We target three. Erdős–Sárközy–Szemerédi #1196 (a 1966 conjecture) asserts $f(A) \le 1 + O(1/\log x)$ for a primitive set $A \subseteq [x, \infty)$, sharpening the previous record of ≈ 1.399 to the conjectured constant 1. The Erdős Primitive Set Conjecture #164 asserts $f(A) \le f(\mathbb{N}_1) = 1.6366\ldots$. Erdős–Sárközy–Szemerédi #1217 produces an infinite divisibility chain inside any set of positive doubly-logarithmic density. The proofs use the sub-invariance of the doubly-harmonic weight under the von Mangoldt chain, Mertens-type estimates, and the monotonicity of the Dirichlet eta function. This paper is a strong reference point because parts of it were already formalized by others: #1196 by Math Inc.'s *Gauss* agent in roughly 4,000 lines of Lean (Math, Inc., 2026), and #164 by Boris Alexeev using Codex (Alexeev et al., 2026). Conjecture #1217 had not been formalized by anyone.

4.2 Setup

We ran the harness three times; every agent runs on Codex (GPT-5.5-xhigh, 258K, read-only, no web access). For Erdős–Graham, one run (`ErdosGraham`[2]) formalizes all four target theorems. For the primitive-sets paper we split the work into two runs to test *incremental development*: whether a finished formalization can be extended by adding targets to the problem file and rerunning the harness, rather than starting over. A first run (`Erdos1196`[3]) formalizes #1196. We then seed a second repository (`Prim`[4]) with the final #1196 blueprint, add #164 and #1217 to the target statements, and rerun the harness; both build on the #1196 infrastructure. Each run receives only the paper source and the target statements; success means the `main` branch reaches a state where every blueprint proof node carries a complete proof, no `sorry` or `sorry_using` remains, and CI passes all seven structural checks (Section 3.3).

4.3 Results

Every target is formalized. LeanMarathon discharged all seven target theorems with complete, machine-checked proofs (Table 2): every run is `sorry`-free, with no `axiom` or `native_decide`.

[2] `https://github.com/YuanheZ/ErdosGraham`
[3] `https://github.com/YuanheZ/Erdos1196`
[4] `https://github.com/YuanheZ/Prim`

Incremental development works. Seeding `Prim` with the final #1196 blueprint, the harness reused all 59 nodes unchanged and added 145 new nodes (103 new proof obligations) to discharge #164 and #1217, reaching 14,592 lines over 204 nodes. It extended a finished formalization to new targets rather than restarting, confirming the incremental-development mode Section 4.2 set out to test. Across the three runs the harness proved 258 distinct lemmas and theorems: 111 for Erdős–Graham, 44 for #1196, and 103 more for #164 and #1217.

Formalization feedback sharpens the mathematics. The Lean compiler is not only a checker but also can provide ground-truth mathematical signal, and the issue-to-Refiner loop turns that signal into better mathematics. Across the three runs essentially every blocked-node issue is grounded in a concrete formalization artifact rather than a prose disagreement, and three kinds recur. The compiler *refutes false statements*: a Worker instantiates a tail estimate and collapses its conclusion to $1 \leq 0$ (issue #431), or a two-window inequality to $4/3 \leq 2/3$ (issue #465), each forcing a corrected hypothesis. Mathlib's totalization conventions *expose vacuous targets* that prose hides: a non-summable real `tsum` is defined to be 0, so the formalized #1196 bound held even for a divergent series until a `Summable` hypothesis was added (issue #2), and a real `limsup` of an unbounded count returns 0, so the divisibility-chain density target was unsound until it was recast in `ENNReal` (issue #102). And failed tactic probes together with Mathlib-absence findings *locate the missing mathematics*: when `positivity` cannot sign the derivative of the Dirichlet eta function and no monotonicity lemma exists (issue #16), and the naive route then reduces to a false inequality between Gamma measures (issue #20), the Refiner is driven to the paper's actual argument, eta monotonicity via stochastic domination of Gamma laws. A human reads "$f(A) \leq \cdots$" charitably, assuming the series converges; the formalization does not, and that mechanized skepticism, surfaced as an issue and resolved by the Refiner, is where much of the harness's mathematical value lies.

The runs are autonomous. Table 3 reports the orchestration cost of each fully autonomous run. The Erdős–Graham run took 19 rounds, launched 58 Workers and 7 Refiners, and cost $257 in GPT-5.5 API-equivalent tokens; the #1196 run took 17 rounds and $189; the #164/#1217 run was the largest, at $624. The CI gate is selective rather than ceremonial: of the 58 Erdős–Graham Workers, 44 produced PRs that passed all checks and merged, the rest being rejected before reaching `main`.

Parallel PRs never conflict. Across the three runs, 135 Worker pull requests landed on a continuously moving `main` by direct squash-merge, as many as 16 in a single round's parallel batch, and not one produced a merge conflict. This realizes the central guarantee of the frozen editable region (Section 2.2): Workers acting on disjoint spans produce patches that commute, so PRs land in any order without collision.

Human observation tightens the harness. We observe the agent's working trace to constantly reshape the design. Early Erdős–Graham runs fixed the `maxHeartbeats` to 0, which disables Lean's deterministic timeout. Reading the Worker traces, we found a strong preference for automation (`nlinarith`, `simp`, `aesop`) over explicit tactic scripts. With no deterministic ceiling, such a search runs until a coarse external timeout cuts the whole build, stalling a node with no reproducible signal. We changed the header to a fixed 500K-budget, restoring a per-declaration deterministic ceiling: a runaway search now fails fast and locally instead of exhausting wall-clock time, and the agent's reliance on unbounded automation dropped sharply. To make efficient proofs the path of least resistance, we gave the Worker two skills, `proof-refactoring.md` and `performance-optimization.md`, from lean4-skills (Freer, 2025). The first refactor under the new budget shows the effect: PR #447 replaced broad `nlinarith`/`simp` search in a peak-bound lemma with direct order arguments and hoisted shared sub-proofs, exactly that discipline.

4.4 Case study: Erdős–Graham

The Erdős–Graham run shows where formalization effort concentrates. Its four target theorems were proven in 19 rounds, yet every one of the 16 refinement issues fell into just two families: Proposition 9's three-case tail bound, the analytic core of the general theorem (8 issues, spanning all seven refiner rounds), and the

Table 3: Orchestration statistics for the three runs, computed from the agents' session histories. Cost is GPT-5.5 API-equivalent ($5 / $0.50 / $30 per million input / cached-input / output tokens). Critical path is agent compute under parallelization; times are `hh:mm:ss`.

Metric	**Erdős–Graham**	**ESS #1196**	**#164 & #1217**
Rounds	19	17	40
WORKERs launched	58	33	111
REFINERs	7	6	25
Merged PRs	53	32	93
Critical path (excl. CI wait)	11:38:23	11:32:40	40:43:21
Aggregate agent active time	21:29:41	16:26:32	71:16:52
Stop-hook / CI wait time	07:15:10	02:15:33	07:05:26
Tool-call parsed wall time	10:16:25	06:16:40	49:08:03
Total tool calls	5,273	4,067	12,204
Total tokens	308M	245M	796M
GPT-5.5 API-equiv. cost	$257.17	$189.43	$623.54

nested-interval covering lemma behind the construction theorem (8 issues, across 6 rounds). No other node ever blocked. These are exactly the two results the baseline leaves unproven (Section 4.8).

The REFINER converged each family by splitting the failed node into progressively finer sub-nodes that WORKERs discharged bottom-up. Most repairs corrected blueprint drift: a dyadic tail estimate that was false as written, its hypotheses letting the right-hand logarithmic factor vanish while the finite sum stayed positive so the claim collapsed to $1 \le 0$ (issue #431); a Lean index convention realigned to the paper's (issue #480); and a strengthened statement that forced a complete proof to be downgraded to a placeholder rather than left unsound (issue #468), the wholesale-replacement rule of Section 2.2 in action.

The run also caught a genuine gap in the published proof. In Case C of Proposition 9 the paper picks a Borel peak R and the largest exponential failure $P < R$ beneath it, then uses $a_n \ge e^n$ throughout $[P+1, R+1]$. But the peak condition bounds only a_{R+1}, not a_R, and Case C explicitly permits $a_R < e^R$: the chosen peak can itself be a failure, in which case $P = R$ and the block breaks at its lower end. A WORKER flagged the unjustified step (issue #469) and the REFINER repaired it by weakening the selection lemma to admit a failure at the peak (PR #473), recovering the conclusion. This step is the analytic heart of the general theorem and exactly the kind of gap a machine-checked formalization is built to surface.

4.5 Case study: #1196

The #1196 run shows the opposite pattern: one deep cascade rather than many independent defects, and no node refiled twice. Over five successive rounds the REFINER drilled downward from the surface estimate to the genuinely missing core, inserting one deeper upstream lemma each round. The surface node needed the bound $-\zeta'(1+u)/\zeta(1+u) \le \log 2/(2^u - 1)$; that reduced to monotonicity of the Dirichlet eta function; that reduced to stochastic domination of Gamma distributions in the shape parameter together with a Mellin representation of η. None of these are in Mathlib, so the harness built the entire probabilistic argument the paper compresses into a single sentence.

Once the chain reached the Mathlib-absent facts it converged with no further refinement: WORKERs proved the inserted leaves in dependency order, closing the original round-one defect seven rounds later from the bottom up. Five of the 6 refiner rounds and 6 of the eight issues targeted this one analytic spine, and the repairs added missing lemmas rather than rearranging existing ones, since the paper states in a line what a proof assistant needs a Gamma-coupling argument to establish. The eta-monotonicity step is precisely the blocker the baseline could not discharge (Section 4.8).

Table 4: The most-refiled blueprint nodes in the `Prim` run, with the span and number (n) of distinct Refiner rounds that reopened each. Labels are abbreviated: the path-data, model, and reverse-Fatou nodes form the #1217 adjoint-chain cluster (`mangoldt-adjoint-*`); the last is an EPS node (`eps-*`).

Blueprint node (abbreviated)	Target	Rounds (n)
`kernel-path-data-exists`	#1217	14–28 (10)
`constructed-path-data-exists`	#1217	13–28 (8)
`random-model-exists`	#1217	10–28 (8)
`chain-density-selection`	#1217	9–26 (7)
`reverse-fatou-path-extraction`	#1217	10–26 (7)
`eps-modified-chain-subinvariant`	#164	1–6 (4)

4.6 Case study: #164 and #1217

The `Prim` run is the harness's hardest case and its sharpest illustration of the refinement loop. It ran 40 rounds and merged 93 pull requests (one Blueprinter, 66 Worker, and 26 Refiner) while filing and closing 46 issues. It is by far the harness's largest run: its agents spent 71 hours of active compute, 41 of them along the critical path under parallelization, and $624 in GPT-5.5-equivalent tokens (Table 3). The two targets behaved very differently: #164, the Erdős primitive set conjecture, was proven early, while #1217, the divisibility-chain theorem, occupied the bulk of the run. The Refiner ran in 24 of the 40 rounds; the last nine were refiner-free and completed the cleaned blueprint.

Drift dominates source-gaps. The Refiner classifies every illness area as *drift* (the blueprint diverged from a sound source proof) or *source-gap* (the paper's argument is itself incomplete); see Section 2.2. Its reports record 32 illness areas, split 26 to 6 in favor of drift. The drift defects are telling. A Worker's misformalized sub-invariance statement for #164 was satisfied by the trivial identity kernel `P n m = [n=m]`, so it constrained nothing (issue #5). Several statements asserted a real `tsum` or `limsup` of a divergent or unbounded quantity, which Lean silently totalizes to 0; this hid the missing summability and forced a real-to-`ENNReal` reformulation of the chain-density definitions (issues #2, #33, #102). The 6 source-gaps are where the paper compresses: the invariant-weight asymptotic the authors call "a routine calculation" (issue #76: "not syntactic rewrites… substantive analytic obligations"), the occupation identity it derives by stating that "an induction gives" the result, and Mertens' theorem invoked as a black box (issue #127).

Hard nodes converge over many rounds. A few nodes were refiled and repaired repeatedly before converging (Table 4). The cluster behind #1217 (an adjoint upward Markov chain, an occupation-measure identity, a uniform second-moment bound, and a reverse-Fatou extraction) dominated; its existence lemma `kernel-path-data-exists` was reopened in 10 distinct rounds. The recurrence came in two waves. Through round 21 the Refiner pushed the construction upstream and bundled the analytic content (visit identity, second moment, reverse Fatou) into a single path-data package, so each downstream node could project what it needed (issue #69: "over-bundled for its position in the DAG"). At round 25 it reversed course and unbundled the package to match the paper's derivation order; removing those fields invalidated four downstream projection proofs at once, each replaced wholesale with a placeholder. Round 26 then corrected the real-versus-`ENNReal` limsup semantics, and round 31 supplied the remaining analytic bridges: a measurability fix, a hit-count moment interface, and a Mertens estimate, the run's last source-gap. The node stabilized and the run closed.

The discipline holds. The run produced 12 complete-proof downgrades across 7 rounds: when a parent's statement or type changed, the dependent proof was replaced wholesale, never partially edited (Section 2.2). The churn stayed safe because the downgrade is mechanical, decided by the Lean compiler, not negotiated by the agent.

Formalization expands the compressed steps. Comparing the converged blueprint against the paper quantifies the gap. The #1217 proof occupies about 62 lines of paper prose with no intermediate lemmas; the blueprint discharges it in roughly 84 nodes, a sixteen-fold expansion in lemma count. The single "routine calculation" for the von Mangoldt weight became about 14 explicit lemmas (a sum-integral interchange, a reciprocal-zeta derivative identification, and an integration-by-parts endpoint evaluation), and the phrase "an induction gives" became a 22-node construction of the probability space and its occupation measure. The expansion concentrates not in the number theory the authors wrote out carefully, but at the boundary where the paper defers to standard probability and analysis, exactly where a machine-checked proof cannot.

4.7 Ablation: which design choices matter

The Erdős–Graham paper was first attempted with an earlier version of the harness, giving a controlled ablation on the same target. That run (PRs #46–#429) differs from the one above in exactly two design choices, and it never finished: over roughly twelve days it filed 137 issues and restarted its blueprint about eight times, ending with a *larger* file (12,910 lines) that still carried 26 `sorry` placeholders and none of the four target theorems proven.

The Refiner must see the source. In the earlier harness only the Blueprinter ever read the source proof; the Refiner saw only the blueprint DAG. Unable to separate blueprint drift from a genuine source gap, it answered blocked nodes by inventing upstream machinery the paper never contained, and the blueprint drifted further from the source each round. GitHub gave us enough observability to watch the run stall; we then intervened by hand, launching two human-in-the-loop passes that were given the source proof and tried to realign the blueprint (PRs #405, #429). They diagnosed the damage exactly, finding that the blueprint had "drifted toward early-prefix and moving-band chain-cover machinery as if those were the source construction" (PR #429) and that an invented predicate was "not the paper's Case (C)" and "forced the wrong trajectory" (PR #405), but the drift was too deep to undo. That failure motivated the redesign and a fresh restart from scratch at PR #430, in which the Refiner reads the source proof and must classify every defect against it as drift or source gap. The current run closed all 16 issues with 7 source-aware repairs in a single pass.

The Worker needs a principled stopping rule. The earlier harness wrote its "cannot formalize" test directly into the Worker spec as a budget of physical Lean lines: a node could be declared un-formalizable once its estimated proof exceeded a fixed line count. Workers gamed this rule, filing blocked-node issues that cited size rather than a real defect ("exceeds 1000 physical lines"), at nearly one block per Worker PR. The current Worker spec removes every size signal. Its four-phase audit permits an issue only on a concrete defect, e.g., a misformalized or false statement, a numerical counterexample, a Lean contradiction, or an invalid input, and explicitly forbids filing "merely because the proof is substantial." A proof that is merely long, or that needs a helper the blueprint lacks, must instead be discharged inside the node's mechanically frozen edit region by growing a local refinement DAG, because "a missing or out-of-order upstream helper is never a blocker." None of the new run's 16 issues cite size, and every one names a real defect.

Takeaway. The two changes are complementary and both necessary. A source-blind Refiner turns repair into guesswork that compounds into drift; a budget-based Worker turns "cannot formalize" into an escape hatch. Together they explain a twelve-day run that never converged; removing them turned the same paper into a fast, fully proven formalization.

4.8 Baseline: Aristotle

We gave Aristotle the same inputs, the paper source and the target statements, and let it run to its own stopping point. It failed on both papers (Table 6). On Erdős–Graham it ran seven turns over 40 hours and delivered 751 lines with two `sorry`s. The two gaps are precisely the paper's deepest content: Proposition 9, the three-regime tail bound at the heart of the general theorem, and the nested-interval construction. Aristotle's own report calls both the paper's "deepest mathematical content," beyond what it could automate. On #1196 it ran four times over 24 hours and delivered a 24-line file whose single theorem is a `sorry`; its

Table 5: Ablation on Erdős–Graham: the same paper under the earlier harness (REFINER blind to the source; WORKER governed by a physical-line budget) versus the current harness.

Metric	Earlier harness	Current harness
Outcome	stalled	**complete**
Wall-clock	∼12 days	∼3 days
Blueprint restarts	∼8	1
Issues filed	137	16
citing a line budget	14	0
Source proof given to REFINER	no	yes
Final `Main.lean`	12,910 ln, 26 `sorry`	8,513 ln, 0 `sorry`

report identifies the missing piece as the flow inequality, equivalently the monotonicity of the Dirichlet eta function, and notes that the published proof "has been formalized in Lean by Math Inc.," reading this as evidence that the task "typically requires a dedicated team effort." LeanMarathon closed exactly these gaps autonomously, proving every target with no `sorry`.

Table 6: Head-to-head with Aristotle on identical inputs. Aristotle ran >40 h on Erdős–Graham and >24 h on #1196 without eliminating its `sorry`s.

	Erdős–Graham		**ESS #1196**	
	Aristotle	LeanMarathon	Aristotle	LeanMarathon
Targets proven	0 / 3	3 / 3	0 / 1	1 / 1
Lean lines delivered	751	8,513	24	3,988
Remaining `sorry`	2	0	1	0
Outcome	failed	**complete**	failed	**complete**

4.9 Failure case: the unit-distance disproof

We tried to formalize an OpenAI's recent disproof of the Erdős unit-distance conjecture (OpenAI, 2026). The disproof's clever geometric trick relies deep algebraic number theory, almost none of which exists in Mathlib. Our harness lets a proof stay incomplete only through explicit `sorry` placeholders and forbids assuming results as axioms per CI gate rejects, so with that theory missing the BLUEPRINTER had no honest foothold. The run instead faked the number theory: it modeled a number field as a dummy record and discharged the key step with placeholder values. This type-checks and passes CI, but proves nothing real. The run then stalled, stuck on the same node round after round, because the later geometric steps needed real objects the fake record could not supply, and the target was never reached. The lesson is a scope boundary, not about formalization capability: when the hardest part lives in prerequisites the library **significantly** lacks, the harness can organize the work but cannot fill the missing results since it is far away from the distribution of proof source.

5 Conclusion

We present LeanMarathon, a long-running multi-agent harness that formalizes entire research papers into Lean 4. Treating long-horizon autoformalization as a problem of *agent durability*, it decomposes a paper into an evolving proof DAG and contains faults behind four contract-scoped agents and a deterministic CI gate, turning one brittle multi-day run into many short, recoverable, parallel ones. On two 2026 papers spanning four Erdős problems it formalizes all seven target theorems with no `sorry`, while a commercial agent baseline fails. LeanMarathon is a step toward AI co-mathematicians whose long-running work stays legible, recoverable, and verifiable.

Acknowledgments and AI disclosure

Y. Z. is supported by Warwick Chancellor's International Scholarship and RIKEN-AIP Overseas Student Collaboration Program. JDL acknowledges support of Open Philanthropy, NSF IIS 2107304, NSF CCF 2212262, ONR Young Investigator Award, NSF CAREER Award 2144994, and NSF CCF 2019844. TS was partially supported by JSPS KAKENHI (24K02905) and JST CREST (JPMJCR2015). YS was partially supported by the National Science Foundation under awards 2027737, 2113373, 2414918 and a gift from OpenAI. This research is supported by the National Research Foundation, Singapore and the Ministry of Digital Development and Information under the AI Visiting Professorship Programme (award number AIVP-2024-004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and the Ministry of Digital Development and Information.

ChatGPT was used to generate several images (e.g., Fig. 1) and proofread the paper.

References

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, et al. Aristotle: IMO-level Automated Theorem Proving. *arXiv preprint arXiv:2510.01346*, 2025.

Boris Alexeev, Kevin Barreto, Yanyang Li, Jared Duker Lichtman, Liam Price, Jibran Iqbal Shah, Quanyu Tang, and Terence Tao. Primitive sets and von Mangoldt chains: Erdős Problem #1196 and beyond, 2026. URL `https://arxiv.org/abs/2605.00301`.

Axiom Math. Axiom Math. Website, 2025. URL `https://axiommath.ai/`. Accessed: 2026-06-01.

Kevin Barreto, Jiwon Kang, Sang-hyun Kim, Vjekoslav Kovač, and Shengtong Zhang. Irrationality of rapidly converging series: a problem of Erdős and Graham, 2026. URL `https://arxiv.org/abs/2601.21442`.

Lars Becker, María Inés de Frutos-Fernández, Leo Diedering, Floris van Doorn, Sébastien Gouëzel, Asgar Jamneshan, Evgenia Karunus, Edward van de Meent, Pietro Monticone, Jasper Mulder-Sohn, Jim Portegies, Joris Roos, Michael Rothgang, Rajula Srivastava, James Sundstrom, Jeremy Tan, and Christoph Thiele. A blueprint for the formalization of Carleson's theorem on convergence of Fourier series, 2024. URL `https://arxiv.org/abs/2405.06423`.

Thomas F. Bloom. Erdős problems. `https://www.erdosproblems.com`, 2026. Accessed 2026-05-30.

Matthew Bolan, Joachim Breitner, Jose Brox, Nicholas Carlini, Mario Carneiro, Floris van Doorn, Martin Dvořák, Andrés Goens, Aaron Hill, Harald Husum, Hernán Ibarra Mejia, Zoltan A. Kocsis, Bruno Le Floch, Amir Livne Bar-on, Lorenzo Luccioli, Douglas McNeil, Alex Meiburg, Pietro Monticone, Pace P. Nielsen, Emmanuel Osalotioman Osazuwa, Giovanni Paolini, Marco Petracci, Bernhard Reinke, David Renshaw, Marcus Rossel, Cody Roux, Jérémy Scanvic, Shreyas Srinivas, Anand Rao Tadipatri, Terence Tao, Vlad Tsyrklevich, Fernando Vaquerizo-Villar, Daniel Weber, and Fan Zheng. The Equational Theories Project: Advancing collaborative mathematical research at scale, 2025. URL `https://arxiv.org/abs/2512.07087`.

Sébastien Bubeck, Christian Coester, Ronen Eldan, Timothy Gowers, Yin Tat Lee, Alexandru Lupsasca, Mehtaab Sawhney, Robert Scherrer, Mark Sellke, Brian K. Spears, Derya Unutmaz, Kevin Weil, Steven Yin, and Nikita Zhivotovskiy. Early science acceleration experiments with GPT-5, 2025. URL `https://arxiv.org/abs/2511.16072`.

Kevin Buzzard et al. The Fermat's Last Theorem project. `https://github.com/ImperialCollegeLondon/FLT`, 2024. Lean 4 formalization project, Imperial College London. Accessed 2026-05-30.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, et al. Seed-Prover: Deep and broad reasoning for automated theorem proving, 2025. URL `https://arxiv.org/abs/2507.23726`.

Yaël Dillies, Terence Tao, et al. Formalization of the polynomial Freiman-Ruzsa conjecture of Marton. https://github.com/teorth/pfr, 2023. Lean 4 formalization project. Accessed 2026-05-30.

Tony Feng, Trieu Trinh, Garrett Bingham, Jiwon Kang, Shengtong Zhang, Sang-hyun Kim, Kevin Barreto, Carl Schildkraut, Junehyuk Jung, Jaehyeon Seo, Carlo Pagano, Yuri Chervonyi, Dawsen Hwang, Kaiying Hou, Sergei Gukov, Cheng-Chiang Tsai, Hyunwoo Choi, Youngbeom Jin, Wei-Yuan Li, Hao-An Wu, Ruey-An Shiu, Yu-Sheng Shih, Quoc V. Le, and Thang Luong. Semi-autonomous mathematics discovery with Gemini: A case study on the Erdős problems, 2026. URL https://arxiv.org/abs/2601.22401.

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

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. Herald: A natural language annotated Lean 4 dataset. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.10878.

Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, Olli Järviniemi, Matthew Barnett, Robert Sandler, Matej Vrzala, Jaime Sevilla, Qiuyu Ren, Elizabeth Pratt, Lionel Levine, Grant Barkley, Natalie Stewart, Bogdan Grechuk, Tetiana Grechuk, Shreepranav Varma Enugandla, and Mark Wildon. FrontierMath: A benchmark for evaluating advanced mathematical reasoning in AI, 2024. URL https://arxiv.org/abs/2411.04872.

Fabian Gloeckle, Ahmad Rammal, Charles Arnal, Remi Munos, Vivien Cabannes, Gabriel Synnaeve, and Amaury Hayat. Automatic textbook formalization. *arXiv preprint arXiv:2604.03071*, 2026.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://arxiv.org/abs/2102.06203.

Sidharth Hariharan, Christopher Birkbeck, Seewoo Lee, Ho Kiu Gareth Ma, Bhavik Mehta, Auguste Poiroux, and Maryna Viazovska. A milestone in formalization: The sphere packing problem in dimension 8. *arXiv preprint arXiv:2604.23468*, 2026.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651:607–613, 2026. doi: 10.1038/s41586-025-09833-y. URL https://doi.org/10.1038/s41586-025-09833-y.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.12283.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurélien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. HyperTree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2205.11491.

Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-STaR: Learning to interleave thinking and proving. In *International Conference on Learning Representations (ICLR)*, 2025a. URL https://arxiv.org/abs/2407.10040.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin

Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025b. URL https://arxiv.org/abs/2508.03613.

Patrick Massot. leanblueprint: A plasTeX plugin to build formalization blueprints for Lean. https://github.com/PatrickMassot/leanblueprint, 2020. Accessed 2026-05-30.

Math, Inc. Erdos1196: A Lean formalization of Erdős Problem #1196. https://github.com/math-inc/Erdos1196, 2026. Formalized by the Gauss autoformalization agent. Accessed 2026-05-30.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. An OpenAI model has disproved a central conjecture in discrete geometry. https://openai.com/index/model-disproves-discrete-geometry-conjecture/, 2026. Accessed 2026-06-01.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. URL https://arxiv.org/abs/2009.03393.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Peter Scholze. Liquid tensor experiment. *Experimental Mathematics*, 31(2):349–354, 2022. doi: 10.1080/10586458.2021.1926016. URL https://doi.org/10.1080/10586458.2021.1926016.

Sho Sonoda, Kazumi Kasaura, Yuma Mizuno, Kei Tsukamoto, and Naoto Onda. Lean formalization of generalization error bound by rademacher complexity. *arXiv preprint arXiv:2503.19605*, 2025.

Terence Tao. Machine-assisted proof. *Notices of the American Mathematical Society*, 72(1):6–13, 2025. doi: 10.1090/noti3041. URL https://doi.org/10.1090/noti3041.

Terence Tao. The three components of problem solving: proof generation, proof verification, and proof digestion. Mathstodon post, 2026. URL https://mathstodon.xyz/@tao/116450581967483825. Accessed: 2026-06-01.

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024. doi: 10.1038/s41586-023-06747-5. URL https://doi.org/10.1038/s41586-023-06747-5.

George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, Lei Yu, Aja Huang, Miklós Z. Horváth, Andrew Ferrauiolo, Henryk Michalewski, Codrut Grosu, Thomas Hubert, Matej Balog, Pushmeet Kohli, and Swarat Chaudhuri. Advancing mathematics research with AI-driven formal proof search, 2026. URL https://arxiv.org/abs/2605.22763.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, et al. Kimina-Prover preview: Towards large formal reasoning models with reinforcement learning, 2025. URL https://arxiv.org/abs/2504.11354.

Zichen Wang, Wanli Ma, Zhenyu Ming, Gong Zhang, Kun Yuan, and Zaiwen Wen. M2f: Automated formalization of mathematical literature at scale. *arXiv preprint arXiv:2602.17016*, 2026.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2205.12615.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search, 2024. URL https://arxiv.org/abs/2408.08152.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2306.15626.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. Lean workbook: A large-scale lean problem set formalized from natural language math problems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2406.03847.

Shangtong Zhang. Towards formalizing reinforcement learning theory. *arXiv preprint arXiv:2511.03618*, 2025.

Yuanhe Zhang, Ilja Kuzborskij, Jason D. Lee, Chenlei Leng, and Fanghui Liu. DAG-Math: Graph-of-Thought Guided Mathematical Reasoning in LLMs. In *International Conference on Learning Representations (ICLR)*, 2026a. URL https://arxiv.org/abs/2510.19842.

Yuanhe Zhang, Jason D Lee, and Fanghui Liu. AI4SLT: Empirical Processes in Lean 4 for Formal Statistical Learning Theory. In *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=dfqmQ9WhCP.

Daniel Zheng, Ingrid von Glehn, Yori Zwols, Iuliya Beloshapka, Lars Buesing, Daniel M Roy, Martin Wattenberg, Bogdan Georgiev, Tatiana Schmidt, Andrew Cowie, et al. AI Co-Mathematician: Accelerating Mathematicians with Agentic AI. *arXiv preprint arXiv:2605.06651*, 2026.

Thomas Zhu, Pietro Monticone, Jeremy Avigad, and Sean Welleck. LeanArchitect: Automating blueprint generation for humans and AI, 2026. URL https://arxiv.org/abs/2601.22554.

Contents

A Agent Knowledge-Store Layouts

Each agent runs in an isolated git worktree whose knowledge store is fixed before launch; it reads nothing outside it. Figs. 6 to 9 give the four layouts, and the file roles are as follows.

Runtime. `AGENTS.md` is each agent's charter: its purpose, hard boundaries, inputs, and ordered workflow. `.codex/config.toml` fixes the model, the read-only sandbox, and the exact MCP tools the agent may call; `rules/default.rules` (shared verbatim) forbids `git`, `gh`, and network commands; and `hooks/ralph_wiggum_stop.py` is the stop hook that blocks exit until `delivery.yml` records a merged PR, polling CI and re-injecting failed-job logs (the Worker's variant also accepts an issue as terminal). The Target-Reviewer is single-shot and carries no hook.

Inputs and state. `docs/inputs.yml` lists the run's file paths and `owner`/`repo`/`branch`; it carries `proof_file`, the raw source proof, only for the Blueprinter and Refiner. `docs/delivery.yml` is the terminal-delivery record the stop hook validates. Each agent maintains a tiny `state.md` with summaries per phase and per delivery path. Statuses are classified into `none`, `in-progress`, `pass`, `fail`, `complete`. The recovery rules after an auto-compaction are explicit and ordered: if delivery is `complete`, exit; if exactly one row is `in-progress`, resume it; if a phase is `fail`, jump to issue delivery; if the last execution phase is `pass`, jump to PR delivery; otherwise start the earliest `none` phase. State never overrides the agent's `AGENTS.md`, the phase contracts, or the Lean diagnostics. The combination guarantees that no compaction re-runs a phase that already passed, and no compaction skips a phase that already failed.

Contracts. `contracts/blueprint-format.md` is the `@[blueprint]` formatting contract the CI enforces (a placeholder-only variant for the Blueprinter, a complete-proof variant for the Worker and Refiner); `contracts/latex-quality.md` sets the prose rules with an agent-specific honesty-about-source clause. The Worker alone also carries `contracts/edit-constraints.md`, which partitions the file into frozen and editable line spans enforced by the `apply-patch` server, and `contracts/local-refinement.md`, which governs the local helper-lemma DAG it may grow before its target.

Execution phases. `docs/exec-phases/` holds one ordered `TASK.md` per phase: `understand`/`grounding`/`draft` for the Blueprinter, a single audit phase for the Target-Reviewer, `misformalization`/`numeric`/`polish`/`formalization` for the Worker (Fig. 3), and `scope`/`refine` for the Refiner, whose `scope` phase classifies each illness area against `proof_file` as drift or source gap.

Delivery and references. `docs/deliver/pr.md` and `issue.md` are the PR and issue templates; the Target-Reviewer files only issues, the Worker may do either, and the Blueprinter and Refiner only open PRs. `grounding/SKILL.md` and its worked examples drive just-in-time Mathlib retrieval; the remaining `references/` are read-on-demand know-how, including the Blueprinter's `decomposition.md` (the node-decomposition rubric) and `reframing.md`, the Worker's `formalization-style.md`, `lean-lsp-tools.md`, `proof-refactoring.md`, and `performance-optimization.md`, and the shared `numeric-tools.md`, `compute.md`, and `pdf-reading.md`.

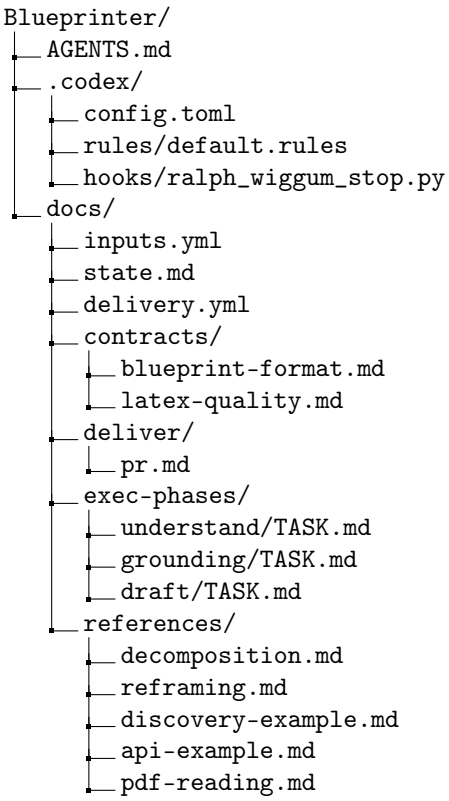

Figure 6: Knowledge store of the BLUEPRINTER.

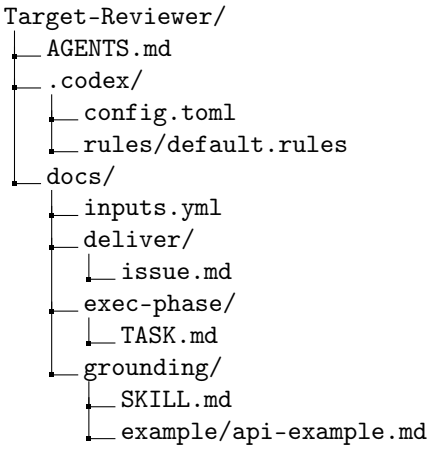

Figure 7: Knowledge store of the TARGET-REVIEWER. It carries no `proof_file` and no PR delivery path: it audits and files issues only.

```
Worker/
├── AGENTS.md
├── .codex/
│   ├── config.toml
│   ├── rules/default.rules
│   └── hooks/ralph_wiggum_stop.py
└── docs/
    ├── inputs.yml
    ├── state.md
    ├── delivery.yml
    ├── contracts/
    │   ├── blueprint-format.md
    │   ├── edit-constraints.md
    │   ├── local-refinement.md
    │   └── latex-quality.md
    ├── deliver/
    │   ├── issue.md
    │   └── pr.md
    ├── exec-phases/
    │   ├── misformalization/TASK.md
    │   ├── numeric/TASK.md
    │   ├── polish/TASK.md
    │   └── formalization/TASK.md
    ├── grounding/
    │   ├── SKILL.md
    │   ├── examples/api-example.md
    │   └── examples/discovery-example.md
    └── references/
        ├── formalization-style.md
        ├── lean-lsp-tools.md
        ├── proof-refactoring.md
        ├── performance-optimization.md
        ├── numeric-tools.md
        └── compute.md
```

Figure 8: Knowledge store of the per-node WORKER. The four `exec-phases` mirror Fig. 3; `edit-constraints.md` and `local-refinement.md` instruct the mechanically enforced editable region.

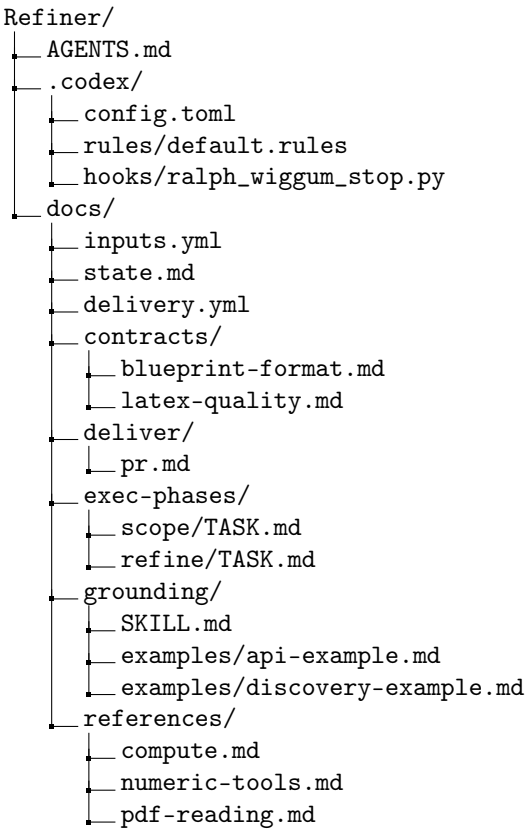

Figure 9: Knowledge store of the REFINER. Its `docs/inputs.yml` carries a `proof_file` (the source proof) and an `issues_file`; its two `exec-phases` are `scope` (locate the illness area) and `refine`.