Technical Report for AI4Math-2026 Track 2: Mechanically Orchestrated LLM Sub-Agents for Theorem Proving in Lean 4

Byung-Hak Hwang [1] **Joonhyun La** [2] **Chul-hee Lee** [1] **Hyojae Lim** [1]

Abstract

This technical report describes a pipeline for formal proof generation in Lean 4 that emulates the workflow of a working mathematician: decompose the theorem into sub-lemmas, probe for counterexamples, prove each sub-lemma in isolation, construct from scratch what the library lacks, and assemble the verified pieces under kernel control. Six sub-agents carry out the cognitive work; a deterministic shell state machine handles every coordination decision between them. Diagnostic outputs pass between sub-agents byte-identically, with no interpretive layer. Applied to the ICML 2026 AI4Math Challenge 2 (Theoretical Computer Science Proving in Lean), the pipeline solves 30 of the 34 challenges (600 of 1000 points) in our local Lean verification with no human-in-the-loop interaction at any sub-agent dispatch or coordination decision.

1. Introduction

Large language models (LLMs) have made rapid progress on mathematical reasoning, achieving competitive performance on contest mathematics and assisting in research workflows. Their generative capability, however, is inseparable from their tendency to hallucinate: they produce plausible-looking arguments whose steps do not actually compose into a valid proof. A single logically invalid step destroys the value of an entire theorem in mathematics. The role of formal verifiers therefore grows alongside LLM capability. Under such a verifier the difficulty shifts: soundness is guaranteed by construction, so the open challenge is not detecting invalid proofs but producing a valid one the verifier will accept.

Writing Lean 4 proofs is itself an open-ended task, and prompting an LLM to "write a Lean 4 proof of theorem T" rarely succeeds outside the simplest cases. Hard theorems require the LLM to decompose the goal, prove sub-lemmas independently, probe for counterexamples, reach for library facts, and construct from scratch what the library lacks. This is the same iterative pattern a working mathematician follows.

This technical report describes a pipeline for formal proof generation in Lean 4 that explicitly emulates that workflow. Six sub-agents (Generator, Refuter, Formalizer Coordinator, Formalizer Node, Inliner, Campaign) carry out the cognitive work. A deterministic shell state machine carries out every coordination decision between them; no LLM participates. Diagnostic outputs between sub-agents are passed byte-identically, with no interpretive layer. Applied to the ICML 2026 AI4Math Challenge 2 (Theoretical Computer Science Proving in Lean), the pipeline solves 30 of the 34 challenges (600 of 1000 points) in our local Lean verification.

Our contributions are:

(1) a six-sub-agent pipeline mechanized by a deterministic shell state machine (Section 4.3), making every coordination decision reproducible and auditable; (2) a verbatim-preservation principle for sub-agent diagnostic transfer (Sections 4.2 and 4.3); and (3) kernel-verified solutions to 30 of the 34 challenges of the ICML 2026 AI4Math Challenge 2 benchmark (Section 6); Section 6.5 catalogues the four failure modes the pipeline encountered along with their resolutions.

2. Related Work

Our work sits at the intersection of three lines of research: LLM-based theorem proving for Lean 4, hierarchical decomposition pipelines, and autoformalization with premise retrieval. We summarize each briefly, noting where our pipeline aligns with or departs from prior practice.

[1]Center for Artificial Intelligence and Natural Sciences, Korea Institute for Advanced Study, Seoul, South Korea [2]School of Mathematics, Korea Institute for Advanced Study, Seoul, South Korea. Correspondence to: Byung-Hak Hwang <bhwang@kias.re.kr>.

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).

2.1. LLM-Based Theorem Provers for Lean 4

The LeanDojo benchmark and ReProver model (Yang et al., 2023) established the modern open-source baseline: a retrieval-augmented LLM that selects premises from Mathlib and emits tactic-level proofs. DeepSeek-Prover V1 (Xin et al., 2024) and Goedel-Prover (Lin et al., 2025b) scaled this pattern with large synthetic datasets, reaching 46.3% (pass@64) and 57.6% (pass@32) on miniF2F respectively. Kimina-Prover Preview (Numina and Kimi Team, 2025) introduced an RL-driven formal-reasoning pattern on a 72B Qwen base, achieving 80.7% on miniF2F at pass@8192. Lean-STaR (Lin et al., 2025a) showed that interleaving informal "thoughts" with formal tactic steps improves whole-proof generation. Lean Copilot (Song et al., 2025) embedded LLM tactic suggestion inside Lean itself. These systems all target whole-proof or tactic-level generation; none addresses the decomposition stage as a first-class component of the system.

2.2. Decomposition Pipelines

The Draft-Sketch-Prove (DSP) framework (Jiang et al., 2023) introduced the modern decomposition pattern: an informal proof drafted by a human or LLM is converted by a sketch model into a formal proof structure with verifiable steps, and the remaining gaps are filled by an automated prover. DeepSeek-Prover-V2 (Ren et al., 2025) adopted recursive subgoal decomposition under reinforcement learning, achieving 88.9% on miniF2F-test by prompting a large model to decompose problems into subgoals with `sorry` stubs and using a 7B prover to fill them. Prover Agent (Baba et al., 2025) combined an informal-reasoning LLM, a formal-prover model, and Lean feedback in a tighter coordination loop, reaching 86.1% on miniF2F. APOLLO (Ospanov et al., 2025) developed an automated LLM–Lean collaboration loop driven by repair-on-error feedback. ProofCompass (Wischermann et al., 2025) added an LLM guidance layer over specialized provers. Our pipeline shares the decomposition motif with all of these systems but differs in that coordination between the decomposition and proof stages is fully mechanized rather than LLM-driven (Section 4.3).

2.3. Autoformalization, Premise Selection, and Adjacent Tools

Autoformalization, the task of translating natural-language statements into formal Lean code, is itself a substantial subtask within any LLM–Lean pipeline. LoC-Decomp (Shi et al., 2026) integrates a semantic consistency checker with iterative Lean feedback for miniF2F autoformalization. KELPS (Zhang et al., 2025a) proposes a multi-language autoformalization framework with semantic-syntactic alignment. HERALD (Gao et al., 2025) releases a natural-language annotated Lean 4 dataset for training autoformalization models. Lean Finder (Lu et al., 2025) addresses premise selection with semantic search over Mathlib that respects user intent. Our pipeline draws on premise retrieval indirectly through the +1-hop scoped API index (Section 5) but does not perform autoformalization itself: the challenge statements are given in formal Lean, and our task is to fill the single `:= sorry` in each.

Beyond formal theorem proving, our framing of LLMs as "valuable assistants" rather than autonomous solvers aligns with positions articulated by working mathematicians on the role of AI in research-level mathematics (Tao, 2025). The mechanical orchestration discussed in Section 4.3 directly supports the human-in-the-loop intervention pattern that view recommends, by exposing each coordination decision as a named, addressable point in a shell state machine.

The CSLib library underlying Challenge 2 was itself introduced through a workshop paper (Zhang et al., 2025b) that proposed theoretical computer science as a scalable formalization domain. We adopt CSLib as fixed background and focus on the LLM–Lean pipeline rather than on dataset construction.

3. Background: ICML 2026 AI4Math Challenge 2

The ICML 2026 AI4Math Challenge 2 (Theoretical Computer Science Proving in Lean) provides 34 Lean 4 theorem statements, across seven topics, drawn from CSLib's TCS modules covering graph theory and combinatorics, computational complexity, and algorithm correctness. Each statement is a top-level theorem whose body is a single `sorry` placeholder, and the participant's task is to replace that `sorry` with a proof that survives the challenge's official grader. The challenge imposes three submission constraints that any participating pipeline must satisfy; a fourth, implicit constraint also follows from the pinned toolchain.

Byte-pristine submission. Each submitted Lean file must be byte-identical to its pristine challenge form except for the proof term that replaces the single `sorry`. No challenge-file imports, no `Def_*` content, no headers, no `lakefile.toml`, and no `lean-toolchain` may be modified.

Three permitted axioms. Running `#print axioms <theorem>` on the submitted proof must yield an axiom closure that is a subset of {`propext`, `Quot.sound`, `Classical.choice`}. Any other dependency in the printed closure, such as `sorryAx` (introduced by an unfilled `sorry`) or a user-introduced axiom, is disallowed.

Official judge. Submissions are graded by SafeVerify (GasStationManager, 2025), the official verifier for the challenge, which compiles each submitted file against the pinned toolchain and accepts it only if the Lean kernel accepts the proof, the theorem statement exactly matches the original challenge specification, and the proof's transitive axiom closure is a subset of the three permitted axioms. The official verifier additionally rejects metaprogramming patterns that could escape its sandbox; the pipeline is therefore designed to avoid tactics whose output may differ across `lake build` invocations.

Implicit constraint: heartbeat budget. The pinned library set under which all submissions are evaluated is Lean 4.28.0 with Mathlib v4.28.0 and CSLib v4.28.0. Lean 4.28.0 enforces a command-level 200,000-heartbeat budget on each top-level declaration. The challenge specification does not state this budget, but every submission must compile under the pinned toolchain, so the budget is binding in practice. We refer to it as an *implicit constraint* to distinguish it from the three submission constraints above. The pipeline confines all four constraints to its validation gate (Section 4.5).

4. Pipeline

Our pipeline replaces the conventional "single LLM proof writer" with a six-sub-agent system whose coordination is fully mechanized. The remainder of this section describes the working-mathematician analogue that motivates the design (Section 4.1), the role and I/O contract of each sub-agent (Section 4.2), the deterministic shell state machine that orchestrates them (Section 4.3), the decomposition strategy (Section 4.4), and the way the assembled proof is validated against the challenge's submission contract (Section 4.5). The components described in Sections 4.1 to 4.4 are pipeline-inherent and applicable to any Lean-based formal-proof workflow. The validation step in Section 4.5 is the only place where ICML 2026 AI4Math Challenge 2 specific constraints enter. No sub-agent prompt or contract was tuned per challenge: the same six sub-agent contracts and the same dispatch template set drive every one of the 34 challenges, with challenge-specific content entering only through the pristine challenge file and a per-challenge scoped library index (Section 5).

4.1. Working-Mathematician Analogue and Overview

A working mathematician proving a theorem rarely writes it as a single linear argument. They split the statement into intermediate lemmas, attempt each in isolation, suspect a counterexample when a step resists, search the standard references for relevant facts, prove from scratch what is not in the references, and finally glue the pieces back into a single proof which they read end-to-end to confirm correctness. Our pipeline implements this loop, with each phase delegated to a dedicated sub-agent and each transition mechanized by a shell state machine.

Figure 1 shows the overall flow. The *Generator* emits an informal blueprint (the α-document) and a skeletal Lean 4 file (the β-skeleton) whose top-level theorems are each marked `:= sorry`; this corresponds to the mathematician's first pass, naming the intermediate lemmas. The *Refuter* then probes each lemma for small counterexamples before committing effort to its proof. The *Formalizer Coordinator* dispatches one *Formalizer Node* per surviving lemma in topological DAG-layer waves, each producing a kernel-verified proof term for its `sorry`. The *Inliner* assembles all proven nodes back into the original challenge file, measures the unfolded kernel cost, and either accepts the assembled file or selects the most cost-dominant nodes to be re-proven at a tighter style level. The *Campaign* sub-agent drives this loop over the full challenge worklist, fresh per campaign. The six sub-agents cover the cognitive work; the orchestrator described in Section 4.3 covers every coordination decision between them.

4.2. Sub-Agent Roles

Each sub-agent is dispatched in a fresh LLM context behind a context firewall: it receives only the file paths and short parameters it needs, never the full output of other sub-agents. Outputs are short machine-readable verdicts; proof terms, error tails, and refutation certificates live on disk in `work/<challenge>/` and never enter the orchestrator's memory. Each sub-agent's model assignment and the contract sentinel that enforces it are detailed in Section 5.

Generator [$\alpha + \beta$ emitter]. Reads the pristine challenge file, its `Def_*` imports, and the scoped API index `api.txt`; emits `blueprint.yaml` (a topologically ordered lemma DAG) and a Lean 4 skeleton `solution.lean` with every body `:= sorry`. Re-dispatched up to $G_{\max} = 3$ times per trigger; if exhausted, the orchestrator may perform at most $\mathit{Pivot}_{\max} = 2$ strategy pivots, each resetting the per-trigger counter.

Refuter [bounded per-node counterexample search]. Dispatched once per pregate-OK β over every `skeleton_typechecks` node. Tries a fixed minimal-instance ladder ($J = 2$); returns a kernel-checked counterexample certificate or `NO-REFUTATION`. Cap $A_3 = 1$. A `REFUTED` verdict triggers a `G-REFUTE` α-regen with the certificate passed verbatim.

Formalizer Coordinator [per-challenge DAG driver]. Walks the DAG in topological layer waves; layer N's proven node bodies are spliced byte-exactly into the canon-

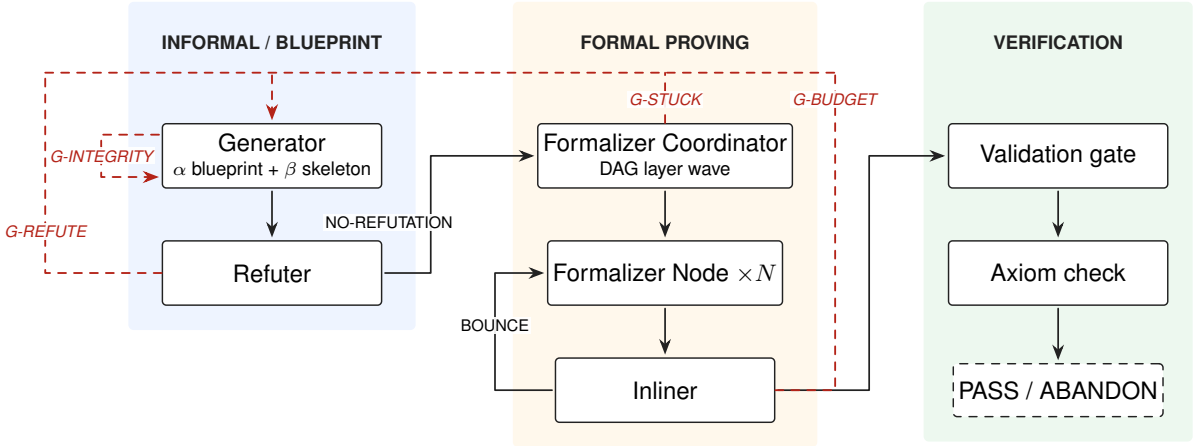

Figure 1. Pipeline workflow for a single challenge. Five of six sub-agents are shown; Campaign (the outer worklist driver, one level above the per-challenge cycle diagrammed here, Section 4.2) is omitted for clarity. Solid arrows are main dispatch; dashed arrows are α-regen feedback to the Generator, labeled at each merge point with one of four triggers (G-INTEGRITY, G-REFUTE, G-STUCK, G-BUDGET; **G** = Generator re-dispatch). BOUNCE is the Inliner's local return path that re-proves the top-$K_{\max}$ cost-dominant nodes at a higher style level.

ical β before layer $N+1$ dispatches. Tracks per-node `style_level` $\in \{1,2,3,4\}$ (free / no-simp / no-match / aux-free), which the Inliner bumps on BOUNCE.

Formalizer Node [single-node proof writer]. Takes one node, the per-node scratch β, the `api.txt`, the current style level, and (on re-dispatch) the prior term plus fold-cost. Returns `solved` or `stuck:<reason>`. Per-node attempt cap $B_1 = 4$ across the four style levels.

Inliner [assemble $\rightarrow$ measure $\rightarrow$ BOUNCE-or-gate]. Splices proven nodes into the gradeable position, measures the unfolded kernel cost, and then either BOUNCEs (the top $K_{\max} = 3$ cost-dominant nodes are re-proven at a higher style level) or invokes the validation gate (Section 4.5). Outer cap $C_1 = 10$. Script-first: LLM reasoning never substitutes for a script verdict.

Campaign [outer worklist driver]. The main session, running depth-1: it dispatches leaf sub-agents directly. Reads the challenge order from `tools/triage.md`, banks `proved` results, and surfaces `ABANDONED` challenges. Each campaign starts with a fresh workspace: prior `work/`, `kb/`, and `results/` artifacts are archived under `results/<prior_campaign_id>/` before the campaign begins.

4.3. Mechanical Orchestration

All coordination decisions between the six sub-agents (when to dispatch the Refuter, when to bump a node's style level, when to BOUNCE versus gate, when to trigger an α-regeneration, when to declare a strategy pivot) are made by a deterministic shell state machine. The orchestrator mechanizes the per-challenge Step 0–9 protocol as a phase set with three terminal states (PASS, ABANDON, SURFACE). No LLM participates in any of these decisions.

The state machine follows an exit-resume pattern. At each dispatch point it serializes the current phase, sub-state, last verdict, and any extra parameters to a state TSV on disk, then exits with code 10 and writes one line `DISPATCH:<agent>:<prompt-path>:<verdict-path>` to stdout. The main session reads the line, dispatches the indicated sub-agent with the rendered prompt, writes the sub-agent's verdict to the indicated path, and re-invokes the orchestrator. The orchestrator reads the state TSV and resumes from the exact phase it exited. Dispatch prompts are produced from one of seven `envsubst` templates with a fixed whitelist of nine substitution variables; rendering fails closed if any unwhitelisted variable is referenced. Verdicts flowing back from sub-agents are validated by a single per-agent grammar: the Formalizer Node's verdict must match `solved | stuck:<node>:<reason>` and the Refuter's verdict file must list one `<node>\s+(REFUTED|NO-REFUTATION)` per line. Out-of-grammar verdicts are treated as protocol violations and surface upward as `SURFACE`.

A central design choice is that the diagnostic outputs of sub-agents (the Refuter's counterexample certificate, the Formalizer Node's `stuck:<reason>` body, the blueprint-integrity verdict, the fold-cost trace) are passed to the next sub-agent **byte-identically**, with no interpretive layer. The implementation pairs a pure concatenator (which emits a single bundle file with begin/end markers around each verbatim source file) with a byte-equality check run after every bundle assembly. Only the orchestrator-side commentary

is removed; nothing the sub-agent observed is summarized, paraphrased, or trimmed before reaching the agent that consumes it. This is the verbatim-preservation principle, and it is the single most important invariant for reproducing or auditing a campaign.

Removing the LLM from coordination might appear to weaken the system. We do not claim that mechanical orchestration maximizes proof-discovery rate (an LLM coordinator might in principle route better); we claim that it maximizes the workflow's *auditability* and its *compositionality with future human intervention*. First, given identical sub-agent verdicts, the orchestrator produces an identical dispatch sequence: every coordination decision is a deterministic function of the state TSV. Second, every dispatch point doubles as a well-defined intervention point. A future mathematician applying this pipeline to a research-level theorem could read `.orch.state.tsv`, identify which verdict caused a stall, and intervene at that point, for example by replacing the verdict, editing the blueprint, or supplying a manual lemma. The rest of the state is left undisturbed. The benchmark reported here does not exercise this affordance, since its problems are tractable to fully automated proof generation. In research-level mathematics the human-in-the-loop role grows with difficulty (Tao, 2025). The LLM's reasoning autonomy is preserved inside each sub-agent's fresh context, while mechanical orchestration keeps the boundary between human and AI judgment explicit, so that the same pipeline composes with future human intervention without redesign.

4.4. Decomposition

Decomposition is the working mathematician's primary tool against complexity: any theorem too large to attack directly is split into intermediate lemmas that can be attacked individually. Our Generator emits the lemma DAG at challenge entry, and the Formalizer Coordinator processes it in topological layer waves. The blueprint records each node as a tuple `(id, statement, deps, status)`; edges are explicit through `deps`, which lists every prior node whose proven body is needed inside this node's proof. Layer N's proven node bodies are spliced byte-exactly into the canonical β before layer $N+1$'s nodes are dispatched, so each Formalizer Node sees its predecessors as ordinary referenceable lemmas, not as `sorry` placeholders.

For each lemma the Formalizer Node first attempts library reuse: it greps the scoped API index `api.txt` (Section 5) and applies `exact?`, `apply?`, and library search. We use the recorded pin Lean 4.28.0 / Mathlib v4.28.0 / CSLib v4.28.0; in practice a non-trivial challenge always requires at least one fact that has no direct counterpart in this pin. When library reuse fails, the Formalizer Node is instructed by its contract to construct the missing fact from first principles via nested `have ... := by ...` blocks inside the current proof body, rather than reporting the node stuck. The construction must be local and inline: the contract explicitly disallows polluting Mathlib, CSLib, or the persistent `kb/` knowledge base with new top-level declarations. The same rule applies recursively: if the construction requires a sub-fact that is itself not in the library, that sub-fact is also constructed inline. The pattern mirrors what a working mathematician does when a needed fact is not in a textbook: they prove it as a sub-lemma in their notes, not as a published theorem.

4.5. Validation under Challenge Constraint

The final responsibility of the pipeline is to satisfy the submission contract of the ICML 2026 AI4Math Challenge 2 (described in detail in Section 3): each submitted file must be byte-identical to its pristine version except for the single `:= sorry` filled in, must compile under `lake build` against the pinned library set, and must list only the three permitted axioms, `propext`, `Quot.sound`, and `Classical.choice`, when printed with `#print axioms`. We enforce these as gate conditions, not as trust assumptions; the pipeline only declares a challenge `proved` after its local validation gate returns OK.

The gate, invoked by the Inliner as the final step of every successful inline cycle, executes three checks in sequence: (a) a byte-diff against the pristine challenge file, restricting all changes to the proof-term region; (b) `lake build` under the pinned toolchain; and (c) `#print axioms <theorem>` against the allow-list. Any single failure short-circuits the gate; only an all-pass gate flips the challenge status to `proved`. The gate mirrors the official SafeVerify conditions (Section 3)—kernel acceptance, byte-level statement preservation, and the permitted-axiom closure—so the pipeline can reject invalid candidates locally without a network round-trip on each inline cycle, while the official SafeVerify run remains the final scoring authority. Implementation details of the gate scripts are deferred to Section 5.

These constraints (Section 3) are challenge-bound. Porting the pipeline to a different Lean workflow requires only swapping the gate.

5. Implementation Details

This section reports the concrete settings under which the pipeline of Section 4 was run: the model and dispatch contract, the breadth of library information offered to each sub-agent, the precise format of feedback bundles between sub-agents, and the design choices that keep the pipeline portable to Lean workflows beyond this challenge.

Model and dispatch contract. The Generator was dispatched at `model: opus` (Claude Opus 4.7); the other five sub-agents (Refuter, Formalizer Coordinator, Formalizer Node, Inliner, Campaign) were dispatched at `model: sonnet` (Claude Sonnet 4.6). The sub-agent contracts mandate `opus` or an equivalent-capability model and forbid a silent downgrade; the sentinel is embedded as a single line inside each of the seven sub-agent contract files, and a contract linter regression-checks the sentinel and ten companion contract clauses on every commit. The Generator carries the heaviest reasoning load: lemma DAG design, library reach, and recovery from refutations. We therefore concentrate the strongest model on the Generator and route the other sub-agents to a smaller equivalent-capability model. This isolates the model-capability question to one well-defined dispatch point per challenge.

Library breadth offered to sub-agents. Sub-agents are not given full Mathlib or CSLib content; they are given a scoped, declaration-header-only API index, regenerated per challenge. The index, written to `work/<challenge>/api.txt`, is computed as the exact +1-hop closure of the challenge's `Def_*` imports: hop 0 is the set of `Def_*` files the challenge imports directly, hop 1 is the modules those `Def_*` files import, and hop 2 is the modules those hop 1 modules import; nothing deeper is included. Source roots are restricted to the challenge's topic `Def_*` plus `.lake/packages/{mathlib,cslib}`; modules resolving to batteries, aesop, Cli, Qq, or Lean core are skipped. The index records only header lines for `theorem`, `lemma`, `def`, `abbrev`, `structure`, `inductive`, `class`, and `instance` declarations, with their file paths and line numbers. In practice the index ranges from tens of lines (e.g. ten lines for `Challenge_Dijkstra_3`) to a few thousand (e.g. five thousand for `Challenge_Kruskal`). Sub-agents are instructed to grep this file rather than read it wholesale, and to refrain from naming any library symbol that does not appear in it. The +1-hop boundary is a deliberate trade-off: wide enough to expose the immediate ambient theory, narrow enough to prevent the LLM from invoking unrelated Mathlib lemmas it could not have seen here.

Feedback bundle format. The four α-regeneration triggers all re-dispatch the Generator (hence the **G** prefix): `G-INTEGRITY` on mutation-grammar violation, `G-REFUTE` on a kernel-checked counterexample, `G-STUCK` on a Formalizer Node exhausting its style levels, and `G-BUDGET` on the Inliner's fold-cost overflow. Each emits a bundle whose header (`# TYPE: G-<trigger>`, `# wd:`, `# verdict:`, optional `# node:`) is followed by the verbatim source diagnostic files framed by `=== begin file <label> ===` / `=== end file ===` markers. A byte-equality check fails closed on any deviation.

Generic-by-construction prompt design. The seven dispatch templates that produce sub-agent prompts contain no challenge-specific content. They reference inputs only through nine `envsubst` whitelisted variables (path, working directory, node id, style level, feedback bundle path, regen type, regen counter, pivot counter, top-K node list); rendering fails closed on any unwhitelisted reference. The seven sub-agent contract files likewise speak only of generic structures (DAG nodes, layer waves, style levels, fold-cost traces, refutation certificates); all challenge-specific information enters the dispatch exclusively through `api.txt` (regenerated per challenge) and the pristine challenge file.

Pipeline implementation. The pipeline is implemented as a set of deterministic shell scripts: the orchestrator (`tools/orchestrate.sh`), the seven dispatch templates, the seven sub-agent contracts, and helper scripts (`feedback_bundle.sh`, `verbatim_check.sh`, `api_index.sh`, `inline.sh`, `fold_cost.sh`, `gate.sh`, and others), together with the `kit/` baseline and the `tests/` suite. All reported results were obtained under the exact frozen-6 manifest that pins the verification logic.

6. Results

This section reports the per-challenge outcome of our pipeline on the ICML 2026 AI4Math Challenge 2 benchmark (Section 6.1), a per-task-family summary (Section 6.2), the dispatch profile (Section 6.3), a focused case study on the three hardest challenges (Section 6.4), and the failure modes the pipeline encountered along with how each was addressed (Section 6.5).

6.1. Setup and Overall Result

The benchmark consists of 34 challenges drawn from CSLib (the TCS-in-Lean library underlying Challenge 2), spanning seven task families: binary heap and Dijkstra (7 challenges), minimum spanning tree (a single Kruskal challenge), segment tree (4 challenges), treap (17 challenges), spanner (1 challenge), and splay tree (4 challenges). Each challenge is scored binary: full credit for a submission that passes the validation gate (Section 4.5), zero otherwise.

Our pipeline solved 30 of the 34 challenges (600 of 1000 points) in our local Lean verification. Every solve produced a PASS verdict from the local validation gate (Section 4.5) and an axiom-closure check listing only the three permitted axioms {`propext`, `Quot.sound`, `Classical.choice`}. Every solved challenge was independently re-verified by re-running the validation gate from a fresh checkout, which

reproduced the same PASS verdict and axiom closure for all 30. All 30 solutions were produced end-to-end without human intervention; the only human inputs to a campaign were the worklist and the start signal. The four unsolved challenges all belong to the splay-tree family; two of them are unresolved conjectures in the dynamic-data-structures literature (the Sleator–Tarjan Deque and Traversal Conjectures).

6.2. Per-Task-Family Breakdown

Table 1 summarizes the per-family outcome. Three challenges required hard-challenge mechanisms: Dijkstra_3, Kruskal, and Spanner (the latter two both pinned-library-gap cases; all detailed in Section 6.4).

6.3. Dispatch Profile

The pipeline routes every challenge through the identical orchestrator phase set (Section 4.3); the variation across challenges is in how deep into the phase set a challenge proceeds and how many α-regenerations are triggered.

Fast-path challenges (27 of the 30 solved) followed the linear path with no α-regeneration: one Generator emitting α and β, one Refuter batch over every skeleton-typechecked node, a handful of Formalizer Node dispatches walking the DAG layers in topological order, and one Inliner pass concluding with a PASS verdict from the validation gate. DAG width was typically 2–8 nodes; style-level escalation was rarely needed (most nodes were provable at style level 1, free tactic mode).

Among the non-fast-path solves, `Dijkstra_3` and `Kruskal` deviated in two distinguishable ways. (a) Both consumed all three α-regenerations under $G_{\max} = 3$; the triggers observed were `G-INTEGRITY` (mutation-grammar violation) and `G-STUCK` (Formalizer Node escalated through all four style levels without success). Neither triggered `G-REFUTE` on its final blueprint, though an earlier prototype attempt on Dijkstra_3 did surface a `REFUTED` verdict that exposed a false-node class and forced an α-regen. (b) Both walked a deeper DAG: Dijkstra_3 settled at 11 nodes (one strategy pivot), and Kruskal at 26 nodes (three split cycles from a 16-node DAG). The third entry, `Spanner`, was a repair-heavy run in the same pinned-library-gap class as Kruskal.

6.4. Case Study: Hard-Challenge Resolution

Two challenges, `Dijkstra_3` and `Kruskal`, were initially blocked by two distinct obstructions that an earlier prototype of the pipeline did not handle. Resolving each with a targeted pipeline strengthening closed each gap. A third hard challenge, `Spanner`, was solved in a later run by the same pinned-library-gap mechanism as Kruskal, and is described at the end of this section.

Dijkstra_3: implicit constraint, heartbeat budget. The earlier prototype's Formalizer Coordinator produced a complete proof of `dijkstra_correctness` over an 11-node DAG: every node returned `solved`, the Refuter cleared every skeleton-typechecked node, and the per-node axiom check PASSED. However, the prototype's Inliner emitted the assembled term as a chained-`have` body of the form `theorem dijkstra_correctness ...:= by have...; have...; ...; exact ...`, whose cumulative elaboration overflowed Lean 4.28.0's command-level 200,000-heartbeat budget near the sixteenth `have`. In-term `set_option maxHeartbeats` is empirically ignored under Lean 4.28.0, so a per-term raise was not available. The final pipeline's Inliner instead transforms the chain into a `where`-clause structure: the theorem body becomes a single application of a top-level helper, and each per-node fact is recast as a `where`-bound helper elaborated in isolation against its own minimal context rather than the accumulated context of a monolithic `have` chain. In this run that structured layout kept the assembled proof's elaboration within the command-level budget, and the prototype's final 11 proven node bodies pass the gate without modification of any individual proof term.

Kruskal: pinned-library gap. The pinned Mathlib v4.28.0 contains an `IsAcyclic` predicate on `SimpleGraph` and several adjacent lemmas, but lacks the graphic-matroid / MST / cycle-exchange theory required for a full Kruskal correctness proof. An earlier prototype reported stuck after exhausting library search for these facts. The final pipeline instead constructs the missing chain inline as a sub-proving task (Section 4.4). The Generator emits a 16-node initial DAG decomposing the proof into edgeSet/subgraph, acyclic-invariant, reaches, cycle/exchange, and isMST chains; during DAG-layer-wave execution the Formalizer Coordinator opens three split cycles (an acyclic-invariant + cycle-length split into four helpers; a build-step swap into three helpers), settling at 26 nodes total. Each node is proven at style level 1 from first principles via nested `have ... := by ...` blocks within its own proof body, and the full `kruskal_computes_MST` theorem then passes the gate with the three permitted axioms only.

Spanner: pinned-library gap. `Spanner` presented the same obstruction class as Kruskal: the pinned library lacks the high-girth extremal edge bound (the Moore / Bondy–Simonovits bound, $\mathrm{girth} > 2t \Rightarrow \#\mathrm{edges} \leq n^{1+1/t}$) required for the $(2t-1)$-spanner sparsity argument. The pipeline constructs the missing fact inline by the same sub-proving mechanism (Section 4.4); the run was repair-heavy

Table 1. Per-task-family outcome on the ICML 2026 AI4Math Challenge 2 benchmark, by our local Lean verification. *Fast path*: solved in a single forward pass with no α-regeneration feedback loop (Section 6.3). *Hard mech.*: required one or more of the hard-challenge mechanisms described in Section 6.4.

Task family	Challenges	Solved	Fast path	Hard mech.
Binary heap & Dijkstra	7	7	6	1
Kruskal (MST)	1	1	0	1
Segment tree	4	4	4	0
Treap	17	17	17	0
Spanner	1	1	0	1
Splay tree	4	0	0	0
Total	**34**	**30**	**27**	**3**

but terminated with the full theorem passing the gate under the three permitted axioms.

6.5. Failure Modes Encountered (and How Each Was Addressed)

The pipeline's path across the 34-challenge benchmark surfaces four classes of failure that any LLM–Lean system on this benchmark should expect. We summarize them here both as a record of what occurred and as a guide to the pipeline mechanisms that resolve each.

(i) Implicit constraint: heartbeat budget. Assembled proofs that are individually fine can collectively overflow Lean 4.28.0's 200,000-heartbeat command-level cap. Addressed by the Inliner's `where`-clause transformation (Section 6.4).

(ii) Library gap. The pinned library set does not in general contain every fact required for a non-trivial challenge. Addressed by inline construction (Section 4.4).

(iii) Under-hypothesized blueprint nodes. A Generator-emitted node may be FALSE under the available hypotheses, the "false-node class" trap. This typically occurs when the Generator's informal blueprint uses a variable type or hypothesis that is not in scope at the node's position in the dependency DAG. Addressed by the Refuter (Section 4.2): a `REFUTED` verdict triggers a `G-REFUTE` α-regen with the kernel certificate passed verbatim.

(iv) Generator mutation-grammar violation. A Generator's α-regen output may violate the blueprint mutation-field enum. Addressed by the `blueprint_check` integrity gate, which on FAIL triggers a `G-INTEGRITY` α-regen with the integrity verdict bundled verbatim.

7. Discussion and Future Work

The pipeline's solving of 30 of the 34 challenges on this benchmark is, in our reading, less a statement about LLM capability per se and more about how to structure the workflow around an LLM so that its creativity and the verifier's strictness do not interfere with each other. Two implications follow.

First, mechanical orchestration (Section 4.3) was conceived for reproducibility but proves at least as valuable for human intervention: every dispatch point is a location at which a future mathematician can audit, supply a lemma, or redirect strategy without disturbing the rest of the state. The current benchmark was tractable to fully automated proof generation for the instances we solved; in actual research the human-in-the-loop role grows precisely as the mathematics becomes harder (Tao, 2025), and our pipeline extends naturally to that regime.

Second, the four failure modes catalogued in Section 6.5 are not specific to this benchmark but representative of the failure classes any LLM–Lean pipeline should expect to encounter, and each pipeline mechanism resolving one is portable as written.

Limitations and future work. The pipeline is currently tied to the proprietary LLMs identified in Section 5 and one library pin; portability to open-weight models and out-of-domain benchmarks remains open. Natural extensions include a human-in-the-loop research workflow with intervention exposed at each dispatch, and a generalized validation gate (Section 4.5) for Lean workflows beyond binary-scoring submission. Beyond the campaign reported here, we have since extended the same deterministic-coordination principle with typed repair routing, strategy-health diagnosis, and closure-aware re-derivation, as ongoing design evolution not evaluated here.

8. Conclusion

This technical report has described a pipeline for formal proof generation in Lean 4 that emulates the working-mathematician workflow of decomposition, counterexample probing, sub-lemma isolation, gap construction, and kernel-checked assembly. The pipeline's six sub-agents are

coordinated by a deterministic shell state machine, not by an LLM coordinator; diagnostic outputs flow between them byte-identically. Applied to the 34-challenge ICML 2026 AI4Math Challenge 2 benchmark, the pipeline solves 30 of the 34 challenges in our local Lean verification. We believe its structural separation is a useful template for the human–AI collaboration that harder mathematics will demand. LLM reasoning is encapsulated inside each sub-agent's fresh context; coordination judgment is mechanized in shell.

Acknowledgements

Byung-Hak Hwang was supported by KIAS Individual Grant (AP098201) via the Center for Artificial Intelligence and Natural Sciences at Korea Institute for Advanced Study (KIAS). Hyojae Lim was supported by KIAS Individual Grant (AP109701) via the Center for Artificial Intelligence and Natural Sciences at Korea Institute for Advanced Study (KIAS).

Impact Statement

This paper presents work whose goal is to advance the field of AI for mathematics. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Baba, K., Liu, C., Kurita, S., and Sannai, A. Prover agent: An agent-based framework for formal mathematical proofs. In *ICML Workshop on AI for Math*, 2025. URL https://openreview.net/forum?id=sPdQfGQccH.

Gao, G., Wang, Y., Jiang, J., Gao, Q., Qin, Z., Xu, T., and Dong, B. HERALD: A natural language annotated Lean 4 dataset. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=Se6MgCtRhz.

GasStationManager. SafeVerify: A trustworthy judge for Lean proofs. https://github.com/GasStationManager/SafeVerify, 2025.

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. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=SMa9EAovKMC.

Lin, H., Sun, Z., Welleck, S., and Yang, Y. Lean-STaR: Learning to interleave thinking and proving. In *International Conference on Learning Representations (ICLR)*, 2025a. URL https://openreview.net/forum?id=SOWZ59UyNc.

Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., and Jin, C. Goedel-Prover: A frontier model for open-source automated theorem proving. arXiv preprint arXiv:2502.07640, 2025b.

Lu, J., Emond, K., Yang, K., Chaudhuri, S., Sun, W., and Chen, W. Lean Finder: Semantic search for Mathlib that understands user intents. In *ICML Workshop on AI for Math*, 2025. URL https://openreview.net/forum?id=5SF4fFRw7u.

Numina and Kimi Team. Kimina-Prover preview: Towards large formal reasoning models with reinforcement learning. arXiv preprint arXiv:2504.11354, 2025.

Ospanov, A., Farnia, F., and Yousefzadeh, R. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning. arXiv preprint arXiv:2505.05758, 2025.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., and Ruan, C. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. arXiv preprint arXiv:2504.21801, 2025.

Shi, J., Zhang, Z., Ma, B., Zhao, S., Yuan, Y., and Wang, G. LoC-Decomp: LLM autoformalization via logical concept decomposition and iterative feedback correction. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=0KFQ4F9YEH.

Song, P., Yang, K., and Anandkumar, A. Lean Copilot: Large language models as copilots for theorem proving in Lean. In *Proceedings of the International Conference on Neuro-symbolic Systems (NeuS)*, volume 288 of *PMLR*, pp. 1–26, 2025.

Tao, T. Machine-assisted proof. *Notices of the American Mathematical Society*, 72(1):6–12, 2025. doi: 10.1090/noti3041. URL https://www.ams.org/notices/202501/rnoti-p6.pdf.

Wischermann, N., Mayrink Verdun, C., Poesia, G., and Noseda, F. ProofCompass: Enhancing specialized provers with LLM guidance. In *ICML Workshop on AI for Math*, 2025. URL https://openreview.net/forum?id=M6bQVHesCr.

Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., and Liang, X. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. In *NeurIPS Workshop on Mathematical Reasoning and AI (MATH-AI)*, 2024. arXiv:2405.14333.

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=g7OX2sOJtn.

Zhang, J., Zhong, C., Xu, H., Li, Q., and Zhou, Y. KELPS: A framework for verified multi-language autoformalization via semantic-syntactic alignment. In *ICML Workshop on AI for Math*, 2025a. URL https://openreview.net/forum?id=U640PVSSoR.

Zhang, T. J., Jiang, W., Liu, R., Wang, Y., Wang, N., Yang, J., Huang, Y., and Sachan, M. Lean meets theoretical computer science: Scalable synthesis of theorem proving challenges in formal-informal pairs. In *ICML Workshop on AI for Math*, 2025b. URL https://openreview.net/forum?id=snoHekTbpd.