LEANSWARM: AN AGENT SWARM FOR LONG-HORIZON LEAN 4 THEOREM PROVING AND COUNTEREXAMPLE DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon theorem proving requires more than solving isolated proof states: large developments involve evolving proof plans, auxiliary lemmas, failed decompositions, and dependencies that must persist across many proof-assistant interactions. We present **LEANSWARM**, a guide-centric multi-agent system for autonomous long-horizon theorem proving in Lean 4. Specialized agents handle proof planning, construction, repository search, research, cycle analysis, and repair, while durable proof state is maintained outside any individual agent context. When ordinary proof search stalls, the system can escalate to deeper analysis, revise the proof structure, or diagnose that a target is false or under-specified.

We evaluate LEANSWARM on the 600-problem Lean split of NTP4VC and on long-horizon correctness proofs from STRATA. On NTP4VC, LEANSWARM proves **563/600 (93.8%)** verification conditions, which to the best of our knowledge is the highest reported end-to-end solve rate on this split. In the STRATA case studies, controlled comparisons against a standalone Claude Code agent using the same primary model show faster proof development, including 25% lower wall-clock time, 12% lower cost, and 17% shorter proofs on a substantial transformation-correctness task. LEANSWARM also closes a previously open SMT correctness target, discovers proof architectures different from an existing human development, and identifies false or under-specified specifications that the standalone Claude Code agent continues trying to prove. These results suggest that durable state, specialization, and structured escalation substantially extend strong language models on long-horizon formal reasoning.

1 INTRODUCTION

Formal verification uses mathematically specified properties and machine-checkable proofs to establish that software or hardware behaves as intended (Clarke & Wing, 1996). Interactive theorem provers such as Lean 4 (Moura & Ullrich, 2021) provide a particularly expressive foundation for this approach: programs, semantics, specifications, and correctness arguments can all be represented in the same logic and checked by a small trusted kernel. Recent progress in language-model-based Lean theorem proving has moved from retrieval-augmented and feedback-driven proof search toward specialized prover models, self-correction, and agentic proof planning (Yang et al., 2023; Thakur et al., 2024; Ren et al., 2025; Lin et al., 2026; Leang et al., 2026; Li et al., 2026). These systems have demonstrated that strong models, proof-assistant feedback, retrieval, and search can solve mathematical statements that previously required substantial human effort.

Formal mathematics has already begun to expose the challenges of *long-horizon* theorem proving: substantial developments require introducing intermediate lemmas, managing dependencies, choosing appropriate induction hypotheses, and recovering from proof plans that later prove inadequate (Zhang et al., 2026). We observe an analogous challenge in software verification, but with an additional source of complexity: the proof structure is induced not only by a mathematical argument, but also by the semantics and implementation structure of the program being verified. This is especially pronounced for *program transformations*, which rewrite a program into a different representation while preserving a specified notion of behavior (program semantics). Proving such

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

transformations correct often requires relating source and target executions through simulation arguments and invariants that span syntax, environments, control flow, and well-formedness conditions. The resulting proofs may demand different induction principles, strengthened invariants, or even simultaneous and mutually recursive arguments, so repeatedly decomposing a theorem into independent smaller obligations can be actively misleading. At this scale, discovering *how the proof itself should be organized* becomes part of the theorem-proving problem.

We study this problem in STRATA (Strata contributors, 2026), a Lean-based verification framework whose program transformations are implemented and proved correct within Lean. These proofs are substantial proof-engineering tasks: a single top-level result can depend on hundreds of interlocking lemmas, and progress may require discovering appropriate invariants, induction schemes, and dependency structure as the proof unfolds.

Long-horizon verification requires more than local tactic generation: an autonomous prover must manage failed decompositions, dependencies, repository context, and ambiguous failures across an evolving proof. Multi-agent LLM systems have shown that complex workflows can benefit from specialized roles and explicit coordination (Wu et al., 2024; Hong et al., 2024; Chen et al., 2024); we therefore ask whether long-horizon theorem proving can similarly be organized as specialized agents operating over durable shared proof state.

We present **LEANSWARM**, a guide-centric multi-agent system for autonomous long-horizon theorem proving in Lean 4. Rather than treating proving as a sequence of isolated local attempts, LEANSWARM maintains an evolving proof project: specialized agents develop and revise proof structure while durable external state preserves dependencies, failures, and progress across agent contexts. This enables the system not only to sustain long proof developments whose decomposition must be discovered over time, but also to escalate failed proof search into diagnosis, returning counterexamples or identifying missing hypotheses when an obligation is false or under-specified. We evaluate LEANSWARM on NTP4VC and substantial Strata program-transformation proofs; Sections 3 and 5 provide the technical details and evaluation.

Our contributions are:

- **Long-horizon proof engineering.** We study theorem proving in which decomposition, induction structure, invariants, and dependencies must be discovered and revised during proof construction. We evaluate this setting through substantial Strata verification developments that require long-horizon reasoning across many interdependent obligations.
- **Guide-centric multi-agent architecture.** We introduce a system that separates proof strategy, proof writing, repository search, research, cycle analysis, and repair among specialized agents, while keeping orchestration and Lean validation outside the models. On NTP4VC, we evaluate whether this architecture also provides strong general theorem-proving performance beyond the Strata case studies.
- **Durable, revisable proof state with diagnosis and repair.** A persistent proof-obligation DAG, communication history, and non-progress detection preserve reasoning across agent contexts and allow failed decompositions to be restructured rather than forgotten. When proof search stalls, privileged agents can investigate deeper failures and identify false or under-specified targets; our Strata case studies demonstrate both successful proof restructuring and actionable specification diagnoses.

2 PROBLEM SETTING: LONG-HORIZON VERIFICATION IN STRATA

STRATA is a Lean 4 verification framework for defining intermediate verification languages and proving the correctness of transformations between them. A transform maps statements from one language to another, $T : L_1.\mathrm{StmtT} \to \mathrm{Option}; L_2.\mathrm{StmtT}$, while the surrounding language definitions provide the program state, execution semantics, and well-formedness conditions needed to state semantic correctness. For the transformations studied in this paper, correctness is expressed as *overapproximation*: every relevant behavior of the source program must remain represented by the transformed program, so properties established after transformation remain sound for the source. Concretely, STRATA requires preservation of reachable terminal and exiting states, failure behavior, and initial-state well-formedness; Figure 1 shows the corresponding Lean definition.

```
-- Simplification: we omit information about the new prefix of fresh identifiers
def Overapproximates (L1 L2 : Lang P) (T : L1.StmtT -> Option L2.StmtT) : Prop :=
  forall (st : L1.StmtT) (st' : L2.StmtT),
    -- Transforming st is successful, and the result is st'
    T st = some st' ->
    forall (rho0 : Env P), L1.initEnvWF st rho0 ->
      -- Every terminating trace in src also appears in the tgt pgm
      (forall rho', (L1.star (L1.stmtCfg st rho0) (L1.terminalCfg rho') ->
         L2.star (L2.stmtCfg st' rho0) (L2.terminalCfg rho')) /\
      -- Every exit also appears in the target
      (forall lbl, L1.star (L1.stmtCfg st rho0) (L1.exitingCfg lbl rho') ->
         L2.star (L2.stmtCfg st' rho0) (L2.exitingCfg lbl rho')) /\
      -- If the source program may raise an assertion failure, so may the target
      (CanFail L1 st rho0 -> CanFail L2 st' rho0) /\
      -- The initial state is also well-formed for the transformed program
      L2.initEnvWF st' rho0
```

Figure 1: Abridged version of STRATA's *overapproximation* criterion for transformation correctness. For every transformed statement and well-formed initial environment, the target must preserve the source's reachable terminal and exiting states, failure behavior, and well-formedness.

These correctness proofs form the long-horizon setting studied in this work. Their difficulty is not only scale: the proof structure itself often has to be discovered during development. A successful argument may require strengthened invariants, different induction principles, simultaneous or mutually recursive reasoning, and repeated revision of earlier decompositions. A locally plausible helper lemma can therefore turn out to be no easier than its parent or introduce an unproductive dependency cycle. LEANSWARM is designed for this setting: rather than treating theorem proving as a fixed sequence of independent subgoals, it maintains and revises an evolving proof structure as the proof develops. Our evaluation includes substantial STRATA transformation-correctness developments together with obligations from its SMT and type-system layers (Section 4).

3 THE LEANSWARM SYSTEM

LEANSWARM is a multi-agent theorem-proving system for developing long, interdependent Lean proofs. Its central design separates *proof strategy* from *workflow control*: a stateful *proof guide* decides how each proof obligation should be attacked, while a deterministic orchestrator manages the evolving proof state, invokes the appropriate agents and tools, and advances obligations through a fixed control flow. Agents execute in an interleaved, single-threaded workflow rather than concurrently. Figure 2 combines these two views of the system: the participating agents and persistent infrastructure, and the per-obligation state machine through which the orchestrator drives proof development.

3.1 SPECIALIZED AGENTS AND ROLES

We use the term *agent* to mean an LLM-driven process with its own role-specific instructions, conversational state, and permitted tools, following the common abstraction used in multi-agent LLM systems (Wu et al., 2024). Concretely, each LEANSWARM agent is an independent instance of a general-purpose coding-agent environment, similar in spirit to systems such as OpenAI Codex and Anthropic Claude Code (OpenAI, 2025; Anthropic, 2025). Agents need not use different underlying models; they are distinguished primarily by their assigned role, retained context, and tool privileges.

LEANSWARM contains both *stateful* and *stateless* agents. Stateful agents retain conversational context across repeated interactions and are used where continuity is valuable. Stateless agents are instantiated for focused tasks and discarded afterward, providing fresh contexts for specialized analysis, validation, and repair. Table 1 summarizes the principal roles.

The most important separation is between proof planning and proof execution. The stateful *proof guide* maintains the proof plan and reasons about decomposition, dependencies, failures, research, and escalation, but cannot edit Lean files. The stateful *proof writer* executes that plan, edits the current proof, and reacts to compiler feedback. A persistent *search agent* serves as an evolving repository expert, retaining knowledge about definitions, existing lemmas, and code structure so

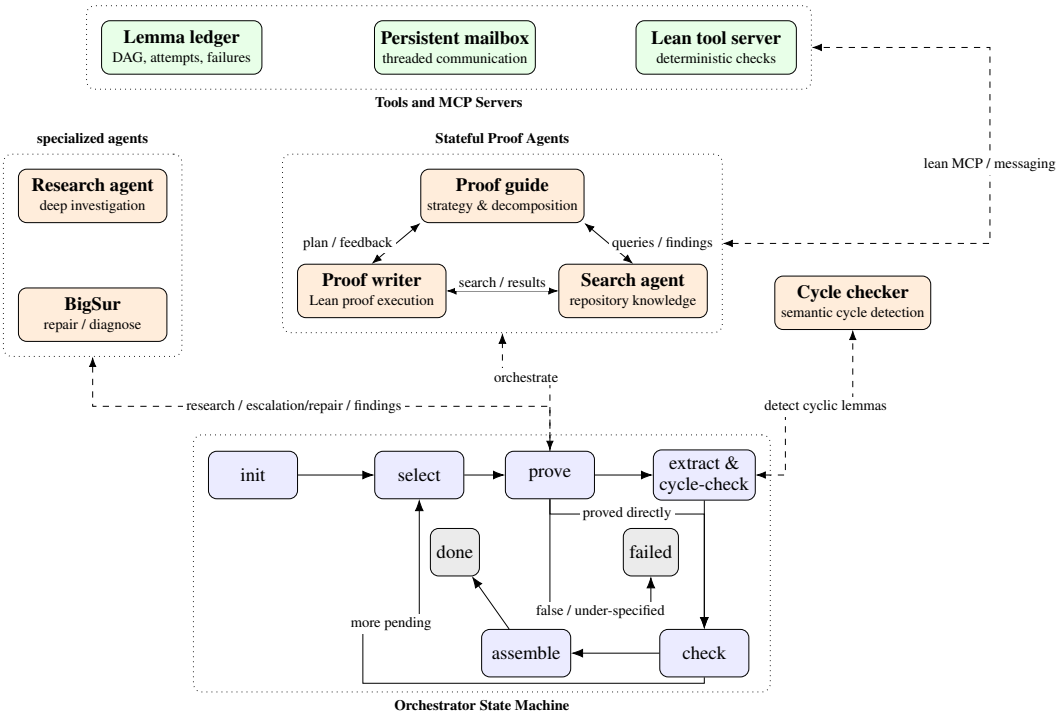


Figure 2: LEANSWARM architecture and per-obligation control flow. The deterministic orchestrator repeatedly selects a proof obligation and routes it through proving, decomposition and cycle detection, validation, and assembly. During proving, the stateful *proof guide* determines strategy and directs the *proof writer*, with a persistent *search agent* providing repository knowledge. Difficult obligations can invoke stateless specialists, including the *research agent*, *cycle checker*, and *BigSur* repair. Durable state and verification live outside the agents in the lemma ledger, persistent mailbox, and long-lived Lean tool server. Directly proved obligations bypass decomposition and cycle detection; failed strategies may be repaired and resumed or terminate with an explicit diagnosis. Only representative infrastructure edges are shown.

repeated questions do not require rediscovering the same context. Focused tasks are delegated to stateless specialists: the *research agent* performs deeper repository-wide investigation when a strategy is blocked; the *cycle checker* analyzes potentially circular decompositions; splitter, merger, validation, and compilation helpers perform narrow mechanical tasks; and *BigSur* provides privileged last-resort repair when the ordinary proving workflow cannot recover, and can return a concrete counterexample when the target cannot be proved as stated.

3.2 ORCHESTRATOR AND STATE MACHINE

Each run begins from a user-specified *root theorem*. As proof development proceeds, the guide may decompose an obligation into auxiliary lemmas, which become new proof obligations and induce an evolving dependency DAG. The orchestrator repeatedly chooses a pending node from this DAG and advances it through the control flow shown in Figure 2.

The orchestrator prioritizes pending obligations (in the `select` phase of Figure 2) based on their estimated generality, favoring lemmas whose proofs are likely to expose reusable auxiliary results that can help discharge obligations elsewhere in the proof. A selected obligation then enters the proving stage, where the proof guide and proof writer attempt either to discharge it directly or to reduce it to auxiliary lemmas. If the obligation is proved directly, it bypasses decomposition and cycle detection and proceeds immediately to validation. Otherwise, the newly introduced obligations are extracted, checked for circular dependencies, and recorded in the proof DAG. The original obligation then becomes *contingent* on its children, and the orchestrator returns to selection to work

Agent	Kind	Role
proof guide	stateful	Maintains proof strategy; decides whether to continue, decompose, research, restructure, or give up. Cannot edit Lean files.
proof writer	stateful	Executes the guide's plan in Lean, edits the current proof, and reacts to compiler feedback.
search agent	stateful	Persistent repository expert that accumulates knowledge of definitions, lemmas, and code structure, reducing repeated searches and tool calls.
research agent	stateless	Performs focused, whole-repository investigation when the current proof strategy is blocked and returns a recommendation.
BigSur (repair)	stateless[†]	Last-resort repair agent with broader privileges to restructure the proof state and diagnose false or under-specified targets.
cycle checker	stateless	Detects semantically circular decompositions (Section 3.3).
splitter / merger	stateless	Decomposes proof files into obligations and reassembles completed proofs.

Table 1: Principal agents in LEANSWARM. Stateful agents preserve context across repeated interactions, while stateless specialists are instantiated for focused tasks. [†]BigSur retains state within one repair invocation but is re-spawned for each escalation. Additional stateless helpers support extraction, compilation repair, validation, and assembly.

on another pending obligation. Once the required dependencies are discharged, completed proofs are validated and assembled back into their parents. This process continues until the root theorem is fully discharged or the workflow reaches an explicit failure.

The orchestrator itself contains no proof strategy. Its responsibilities are mechanical: selecting obligations, invoking agents, extracting generated lemmas, updating proof state, running deterministic Lean-side checks, and assembling completed proofs (see Figure 2). Choices about how to prove an obligation—including decomposition, repository research, restructuring, and giving up—remain with the proof agents. This division keeps the execution protocol deterministic even though the proof plan and dependency structure evolve dynamically over a long run.

3.3 PERSISTENT PROOF MEMORY AND VERIFICATION

LEANSWARM externalizes proof state so long developments do not depend on any single agent context. A persistent *lemma ledger* records the root theorem, auxiliary obligations, dependency edges, statuses, attempts, and failures. Failed decompositions remain visible, allowing fresh agents to resume from prior work rather than rediscovering the same dead ends. Agents coordinate through a persistent threaded mailbox, so decisions and failure information survive context rotation and agent replacement.

A long-lived Lean tool server provides deterministic checks for compilation, theorem status, signatures, dependencies, and remaining `sorry`s, grounding LLM-generated proposals in Lean-side validation. This infrastructure also prevents *non-progressing decompositions*: after filtering obvious repetitions, a separate cycle checker identifies candidate semantic cycles between new obligations and their ancestors, which are then verified with lightweight Lean derivability checks. Confirmed cycles are recorded and pruned, and the affected ancestor is returned to the proving workflow for a different decomposition.

3.4 ESCALATION, RECOVERY, AND ROBUSTNESS

When the proof guide and proof writer stop making progress, LEANSWARM changes strategy rather than repeatedly retrying the same local proof. The guide may invoke a stateless *research agent* to inspect the surrounding repository for relevant definitions, missing lemmas, proof patterns, or evidence that the current strategy is unsound. Persistent failure can escalate further to **BigSur**, a

privileged repair agent that may restructure the decomposition and associated proof state while preserving the root theorem signature. Successful repairs return control to proving; if the target is false or under-specified, BigSur can instead return a counterexample or identify missing assumptions.

Recovery is supported by durable external state and restricted execution. Independent obligations are isolated into separate files and contexts, while mutually recursive declarations remain together. The lemma ledger and persistent mailbox allow fresh agent instances to resume from recorded proof state after context rotation or interruption, and the orchestrator checkpoints progress throughout the run. All proof development occurs inside a sandbox with restricted write permissions, preventing the proof writer from modifying protected parts of the task or weakening the original obligation. In addition, a separate validation agent independently checks the proved obligation against the original target to detect accidental drift or trivialization before the result is accepted. Together, these mechanisms allow LEANSWARM to recover from stalled proof attempts while preserving the integrity of the theorem being proved.

4 EXPERIMENTAL SETUP

Models and compute. LEANSWARM runs as a single-threaded workflow on a single machine. The proof guide and proof writer use the primary frontier Claude model for each experiment, while the persistent SearchAgent may use the immediately preceding model version. No model is fine-tuned; specialization comes entirely from role-specific prompts, tool access, persistent context, and orchestration.

Evaluation settings. We evaluate LEANSWARM in two complementary settings. First, we use NTP4VC (Xu et al., 2026), an open benchmark of 600 Lean verification conditions, to measure broad theorem-proving performance under a fixed agentic workflow. Second, we study substantially longer proof-development tasks from STRATA, where the challenge is not only to close individual goals but to discover and revise auxiliary lemmas, induction structure, invariants, and dependencies over an evolving proof. These case studies include transformation-correctness obligations as well as proofs from STRATA's SMT and type-system layers.

Claude Code baseline. For the STRATA case studies, our primary baseline is a standalone Claude Code agent operating on the same Lean repository and toolchain. The standalone agent and LEANSWARM use the same primary Claude model for proof construction and receive the same target theorem and repository. Claude Code can inspect, edit, and compile Lean code through a thin harness, while LEANSWARM augments the same primary model with specialized roles, persistent proof state, repository search, non-progress detection, and escalation. Where multiple baseline runs are used, we report them explicitly.

Metrics. On NTP4VC, we report task-level proof success under a fixed evaluation budget. For the long-horizon STRATA case studies, we report proof size, wall-clock time, model-usage cost, and whether the proof completes without human intervention. We additionally record qualitative outcomes that aggregate benchmark accuracy does not capture, including alternative proof structures and actionable diagnoses of false or under-specified specifications.

5 RESULTS

We evaluate LEANSWARM in two complementary settings. NTP4VC measures broad theorem-proving performance over a standardized set of Lean verification conditions, while the STRATA case studies examine the long-horizon behavior that aggregate benchmark accuracy cannot capture: evolving decompositions, alternative proof structures, persistent memory, escalation, and diagnosis of incorrect specifications.

5.1 NTP4VC

NTP4VC (Xu et al., 2026) contains 600 verification conditions translated to Lean from real-world program-verification tasks; Appendix A.1 shows a representative binary-search instance. We eval-

System	Solved	Success rate	Setting
LEANSWARM (Opus 5)	**563/600**	**93.8%**	agentic, 4h timeout
DeepSeek-Prover-V2-671B (Xu et al., 2026)	18/600	3.0%	pass@8
Union of NTP models (Xu et al., 2026)	34/600	5.7%	union across evaluated systems

Table 2: Results on the 600-problem Lean split of NTP4VC. To the best of our knowledge, LEANSWARM achieves the highest reported end-to-end solve rate on this split. The published baselines use different inference regimes and budgets, so the comparison should be interpreted as benchmark coverage rather than a matched pass@k evaluation.

	Cost ($)	Time (min)	Proof lines
Standalone (3 runs)	7.44	22.9	536
LEANSWARM (3 runs)	**6.54**	**17.1**	**446**
Difference	-12.1%	-25.4%	-16.8%

Table 3: `detToKleene_overapproximates` with Claude Opus 5. Means over three unfiltered independent runs per condition; all six runs completed successfully with Lean-checked, sorry-free proofs.

uate LEANSWARM on the complete Lean benchmark using Claude Opus 5 with a four-hour per-problem timeout. LEANSWARM proves **563/600 (93.8%)** verification conditions.

As shown in Table 2, LEANSWARM proves 563/600 (93.8%) Lean verification conditions, which, to the best of our knowledge, is the highest reported end-to-end solve rate on the Lean split of NTP4VC. In comparison, the union of all NTP models evaluated in the original benchmark solves 34/600 (5.7%). Although these systems operate under different inference budgets, the comparison shows that long-horizon interactive proof development can cover a substantially larger fraction of the benchmark than independent proof generation. Full evaluation settings, token usage, cost, and latency statistics are reported in Appendix B.

5.2 LONG-HORIZON STRATA CASE STUDIES

The STRATA experiments study proof-development tasks that extend beyond isolated verification conditions. Such proofs require discovering auxiliary lemmas, choosing and revising proof structure, searching the repository for relevant facts, and sometimes determining that the target itself is incorrect or under-specified.

Autonomous proof rediscovery on `detToKleene`. We evaluate LEANSWARM on `detToKleene_overapproximates`, a substantial transformation-correctness theorem with an existing 1,262-line human-written proof. We compare against a standalone agent using the same Claude Opus 5 model and Lean harness. All runs in both conditions produced Lean-checked, sorry-free proofs.

Table 3 compares three independent LEANSWARM runs with three independent baseline Claude Code runs. LEANSWARM is 12% cheaper, 25% faster, and produces 17% shorter proofs on average. We observe the same speed advantage with Claude Opus 4.8: across three LEANSWARM runs and three baseline Claude Code runs, LEANSWARM averaged 87 ± 9 minutes versus 203 ± 61 minutes, with comparable cost (51±$5 versus $45±5). Across both model generations, the clearest benefit is therefore substantially lower proof-development time at similar model cost.

The autonomous runs also rediscover the proof rather than merely reproducing the human development. Across Opus 5 runs, LEANSWARM found both step-indexed and relation-based simulation arguments, while the human proof uses a larger mutually recursive structure. This variation across successful runs indicates that LEANSWARM explores alternative proof architectures for the same correctness theorem.

Previously open SMT correctness target. We next evaluate LEANSWARM on `Factory.quant_correct`, a previously open correctness theorem for quantifier coalesc-

ing in STRATA's SMT layer (Barrett et al., 2010). With Claude Opus 5, LEANSWARM found the proof faster, completing autonomously in 17.6 minutes with 120 lines of Lean at a cost of $7.37; a standalone run required 21.2 minutes and 198 lines at $5.24. The proof hinges on transporting across variable contexts that are propositionally, but not definitionally, equal (Appendix B.3).

Counterexamples and specification repair. Long-horizon proof failure does not always indicate that a stronger tactic is needed. In one STRATA case study, escalation revealed that a call-elimination correctness statement was false as formulated: the available source-side assumptions did not expose enough information to establish the target obligation. Rather than continuing proof search, LEANSWARM returned an actionable diagnosis of the specification failure.

In a separate type-preservation case, the theorem was under-specified rather than false. LEANSWARM identified missing well-formedness assumptions, including the requirement that the type context already contain the `"bool"` type; adding these hypotheses made the intended theorem provable.

This behavior differed qualitatively from a standalone Claude Code baseline using Opus 4.8. On both targets, the standalone agent continued attempting to prove the original statement and did not consider searching for a counterexample, whereas LEANSWARM, using the same model generation, diagnosed the specification issue in each case. These examples show that persistent proof failure can trigger semantic diagnosis and specification repair rather than merely producing a timeout or further proof attempts.

Discussion. Across these case studies, LEANSWARM consistently reduces proof-development time, discovers alternative proof structures, and, when proof search should stop, diagnoses false or under-specified specifications instead of continuing blindly. The results suggest that its main advantage comes from coordinating long-horizon proof development—persistent state, repository search, and targeted escalation—rather than from stronger local proof generation alone.

6 RELATED WORK

Long-horizon formal reasoning. Recent systems move beyond isolated theorem proving toward constructing and revising larger proof developments. Prover Agent combines informal reasoning, Lean feedback, and auxiliary-lemma generation (Baba et al., 2025); LeanMarathon uses multiple agents around an evolving shared blueprint for research-scale formalization (Zhang et al., 2026); and Goedel-Architect constructs and iteratively refines a dependency graph whose open lemmas are discharged by prover agents (Chung et al., 2026). These are the closest methodological comparisons to LEANSWARM.

LEANSWARM differs in treating the evolving proof graph as *durable search state*, not only as a proof plan. Its ledger preserves dependencies, failed decompositions, and proof status across agent contexts, while a persistent replayable mailbox lets specialized agents communicate findings, requests, and failures without relying on a single shared conversation history. Explicit cycle detection guards against auxiliary lemmas that reconstruct ancestor obligations, and escalation can move beyond proof repair to *specification diagnosis*: when an obligation is false or under-specified, LEANSWARM can return a counterexample or identify missing hypotheses. Our setting is additionally repository-scale program verification, where proofs must be recovered from an existing semantic development and may span many dependent declarations.

Neural and agentic theorem proving. Earlier work developed retrieval-augmented proof environments, whole-proof generation and repair, and agentic feedback-driven search (Yang et al., 2023; First et al., 2023; Jiang et al., 2023; Thakur et al., 2024). More recent prover models improve individual-goal solving through decomposition, reinforcement learning, and verifier-guided self-correction (Ren et al., 2025; Wang et al., 2025; Lin et al., 2026). These methods are complementary to LEANSWARM: they strengthen local proof generation, whereas our focus is maintaining, communicating, and revising proof structure beyond a single theorem or model context.

Program verification. Verified compilers established simulation and semantic preservation as foundations for program-transformation correctness (Leroy, 2009). More recently, NTP4VC stud-

ies neural proving of verification conditions from real verification workloads (Xu et al., 2026), while CLEVER evaluates Lean-certified specification and implementation generation (Thakur et al., 2025). LEANSWARM addresses a different scale: constructing the interdependent proof development needed to establish transformation correctness, while also diagnosing obligations that cannot be proved as stated.

7 LIMITATIONS

LEANSWARM uses stateful, single-threaded proof search to preserve failure history and evolving proof structure, trading parallelism for coherent long-horizon reasoning. This can increase wall-clock time relative to systems optimized for broad parallel exploration.

Our long-horizon STRATA experiments are also small-scale and drawn from a single verification framework, so cost and latency comparisons should be interpreted as case-study evidence rather than broad performance estimates. Finally, LEANSWARM combines several mechanisms—persistent memory, specialization, repository search, non-progress detection, and escalation—whose individual contributions are not isolated here; systematic ablations remain future work.

8 AI STATEMENT

Generative AI tools were used during this work for research ideation and execution, including code generation, debugging, literature discovery, and writing assistance (including assistance in drafting this statement itself). All experimental results, Lean proofs, quantitative claims, and final manuscript content were checked and reviewed by the authors.

9 CONCLUSION

We presented LEANSWARM, a guide-centric multi-agent system for long-horizon Lean 4 theorem proving. Its core design combines durable externalized proof state, specialized repository search, and structured escalation so that proof development can persist across long contexts and terminate informatively when a target is false or under-specified. On STRATA, LEANSWARM consistently reduced proof-development time relative to standalone baselines using the same primary model, rediscovered substantially different proof architectures, closed a previously open SMT correctness target, and diagnosed specification failures that the standalone agent continued trying to prove. On NTP4VC, LEANSWARM proved 563/600 (93.8%) Lean verification conditions, which, to the best of our knowledge, is the highest reported end-to-end solve rate on the Lean split. These results suggest that coordinating proof search over long horizons can substantially extend what strong language models can accomplish beyond single-agent proof generation.

REFERENCES

Anthropic. Claude code: Best practices for agentic coding. https://www.anthropic.com/engineering/claude-code-best-practices, 2025.

Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. Prover Agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025. URL https://arxiv.org/abs/2506.19923.

Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, 2024.

Jui-Hui Chung, Ziyang Cai, Zihao Li, Qishuo Yin, Rohit Agarwal, Simon Park, Rodrigo Porto, Narutatsu Ri, Ziran Yang, Shange Tang, Xingyu Dang, Hongzhou Lin, Mengdi Wang, Danqi Chen, Chi Jin, Liam H. Fowl, and Sanjeev Arora. Goedel-Architect: Streamlining formal theorem proving with blueprint generation and refinement. *arXiv preprint arXiv:2606.06468*, 2026. URL https://arxiv.org/abs/2606.06468.

Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. doi: 10.1145/242223.242257.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1229–1241, 2023. doi: 10.1145/3611643.3616243. URL https://doi.org/10.1145/3611643.3616243.

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

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=SMa9EAovKMC.

Joshua Ong Jun Leang, Zheng Zhao, Mihaela Catalina Stoian, Qiyuan Xu, Haonan Li, Wenda Li, Shay B. Cohen, and Eleonora Giunchiglia. Pythagoras-prover: Advancing efficient formal proving via augmented lean formalisation. *arXiv preprint arXiv:2606.12594*, 2026.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, 2009. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Jinzheng Li, Zeru Zhu, and Yuanjie Ren. Merlean-prover: A recursive looping harness for lean 4 theorem proving, 2026.

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

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction (CADE)*, 2021.

OpenAI. Addendum to o3 and o4-mini system card: Codex. https://cdn.openai.com/pdf/8df7697b-c1b2-4222-be00-1fd3298f351d/codex_system_card.pdf, 2025.

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. *arXiv preprint arXiv:2504.21801*, 2025. URL https://arxiv.org/abs/2504.21801.

Strata contributors. Strata: An extensible framework for building verification tools in Lean. https://github.com/leanprover/strata, 2026. Open-source Lean 4 project.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=V7HRrxXUhN.

Amitayush Thakur, Jasper C.H. Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3857. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/a7f67788f7b4d77fa7cd6887de3dcbe7-Abstract-Datasets_and_Benchmarks_Track.html.

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, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025. URL https://arxiv.org/abs/2504.11354.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *Conference on Language Modeling*, 2024.

Qiyuan Xu, Xiaokun Luan, Renxi Wang, Joshua Leang, Peixin Wang, Haonan Li, Wenda Li, and Conrad Watt. Neural theorem proving for verification conditions: A real-world benchmark. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=MfDyickxQA.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html.

Yuanhe Zhang, Yuekai Sun, Taiji Suzuki, Jason D. Lee, and Fanghui Liu. LeanMarathon: Toward reliable ai co-mathematicians through long-horizon lean autoformalization. *arXiv preprint arXiv:2606.05400*, 2026. URL https://arxiv.org/abs/2606.05400.

A NTP4VC EXAMPLES

A.1 REPRESENTATIVE VERIFICATION CONDITION

NTP4VC translates verification conditions generated from Why3 and Frama-C developments into Lean 4. Each benchmark file contains exactly one theorem corresponding to one verification condition (Xu et al., 2026). The following excerpt (see Figure 3) from a binary-search verification condition illustrates the machine-generated structure encountered by the prover; repetitive generated branches are elided for readability.

Unlike typical mathematical theorem-proving benchmarks, such goals arise directly from program verification and often expose large collections of program-state assumptions, arithmetic side conditions, invariants, and generated auxiliary definitions.

B EXPERIMENTAL DETAILS

B.1 NTP4VC EVALUATION

We evaluate LEANSWARM on all 600 Lean verification conditions in NTP4VC using Claude Opus 5 with a 1M-token context window. Each instance is run through the same fixed agentic workflow, with a maximum wall-clock budget of four hours. The evaluation was executed through an evaluation harness running on AWS ECS, with LEANSWARM as the proof engine.

Across the full benchmark, LEANSWARM proves 563/600 verification conditions (93.8%). The run consumed 4.4B input tokens and 73.9M output tokens, for a total model cost of $5,385.27. Median wall-clock latency was 2,287 s (38.1 minutes), and the 90th-percentile latency was 3,924 s (65.4 minutes).

Setting / metric	**Value**
Benchmark size	600 VCs
Model	Claude Opus 5 (1M context)
Per-instance timeout	4 hours
Solved	563 / 600
Success rate	93.8%
Input tokens	4.4B
Output tokens	73.9M
Total model cost	$5,385.27
Median latency	2,287 s (38.1 min)
90th-percentile latency	3,924 s (65.4 min)

Table 4: Full-run settings and aggregate statistics for the NTP4VC evaluation.

Unlike the independent proof-generation setting used in the original NTP4VC evaluation, LEANSWARM treats each theorem as an interactive proof-development task. During a run, agents may repeatedly invoke Lean, search the repository, retain intermediate proof state, introduce auxiliary lemmas, and revise the proof strategy until the theorem is proved or the time budget is exhausted. Accordingly, the NTP4VC comparison in the main text should be interpreted as a comparison of end-to-end benchmark coverage under different inference regimes, rather than as a matched pass@k experiment.

B.2 DETTOKLEENE CONTROLLED COMPARISON

For the controlled `detToKleene_overapproximates` experiment, both LEANSWARM and the standalone baseline use Claude Opus 5 and the same Lean-facing harness. We report three unfiltered independent runs per condition. All six runs terminate successfully with Lean-checked, sorry-free proofs; no failed runs are excluded from the reported averages.

The corresponding Claude Opus 4.8 experiment consists of three LEANSWARM runs and three baseline runs. These experiments are reported separately from NTP4VC because they are intended to

```
import Why3.Base
import Why3.why3.Ref.Ref
open Classical Lean4Why3

namespace binary_search_Complexity_binary_searchqtvc

axiom log2 : ℤ -> ℤ
axiom log2'def (n : ℤ) :
  if n ≤ 1 then log2 n = 0
  else log2 n = 1 + log2 (Int.tdiv n 2)

noncomputable def f (n : ℤ) :=
  if n = 0 then 0 else 1 + log2 n

theorem binary_search'vc
    (a : List ℤ) (v : ℤ)
    (fact0 :
      ∀ (i1 i2 : ℤ),
        0 ≤ i1 ∧ i1 ≤ i2 ∧ i2 < Int.ofNat a.length →
        a[Int.toNat i1]! ≤ a[Int.toNat i2]!) :
    let o1 : ℤ := Int.ofNat a.length

    ((0 ≤ o1 ∧ o1 ≤ Int.ofNat a.length) ∧
     (∀ i : ℤ,
       0 ≤ i ∧ i < 0 ∨
       o1 ≤ i ∧ i < Int.ofNat a.length →
       ¬a[Int.toNat i]! = v) ∧
     f o1 ≤ f (Int.ofNat a.length))

    ∧

    (∀ (hi lo time : ℤ),
      (0 ≤ lo ∧ lo ≤ hi ∧ hi ≤ Int.ofNat a.length) ∧
      (∀ i : ℤ,
        0 ≤ i ∧ i < lo ∨
        hi ≤ i ∧ i < Int.ofNat a.length →
        ¬a[Int.toNat i]! = v) ∧
      time + f (hi - lo) ≤ f (Int.ofNat a.length)
      →
      if lo < hi then
        let mid := lo + Int.tdiv (hi - lo) 2
        -- generated obligations for:
        --   * bounds on mid
        --   * interval preservation
        --   * exclusion of v outside the interval
        --   * decrease of the complexity measure
        ...
      else
        -- unsuccessful-search postcondition
        ...)
  := sorry

end binary_search_Complexity_binary_searchqtvc
```

Figure 3: Excerpt from a binary-search verification condition in NTP4VC. The generated theorem encodes sortedness, interval invariants, functional correctness, and a logarithmic complexity measure. Repetitive generated branches are elided for readability.

isolate the effect of the agentic architecture under a matched-primary-model setting, whereas the NTP4VC experiment measures large-scale end-to-end coverage.

B.3 Quantifier-Coalescing Correctness

Target. `Factory.quant` performs a representational optimization when constructing SMT terms. If a term being wrapped in a single-variable quantifier is already a quantifier of the same kind with a simple, pattern-free outer trigger, the implementation coalesces the two binders into a single multi-variable quantifier rather than producing nested quantifiers. The theorem `Factory.quant_correct` states that this transformation is denotation-preserving under `denoteTerm`. This target had previously remained open due to the difficulty of relating the two representations.

Proof difficulty. The coalesced and nested forms induce variable contexts that are propositionally equal but not definitionally equal. In particular, the relevant contexts take the forms

$$(\langle x, ty\rangle :: args_2).\mathsf{reverse} ++ \Gamma$$

and

$$args_2.\mathsf{reverse} ++ (\langle x, ty\rangle :: \Gamma).$$

These agree after list-associativity reasoning but cannot be identified by `rfl`. The proof therefore requires explicit transport arguments to move `denoteTerm` results across this context equality.

Illustrative run. Using Claude Opus 5, LEANSWARM completed the proof autonomously in 17.6 minutes, producing 120 lines of Lean proof at a model cost of $7.37. The corresponding standalone run completed in 21.2 minutes with a 198-line proof at a cost of $5.24. The LEANSWARM run completed within a single writer chunk and invoked the repository SearchAgent four times during proof construction.