COST-EFFICIENT THEOREM PROVING VIA AGENT ORCHESTRATION IN PROGRAM VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

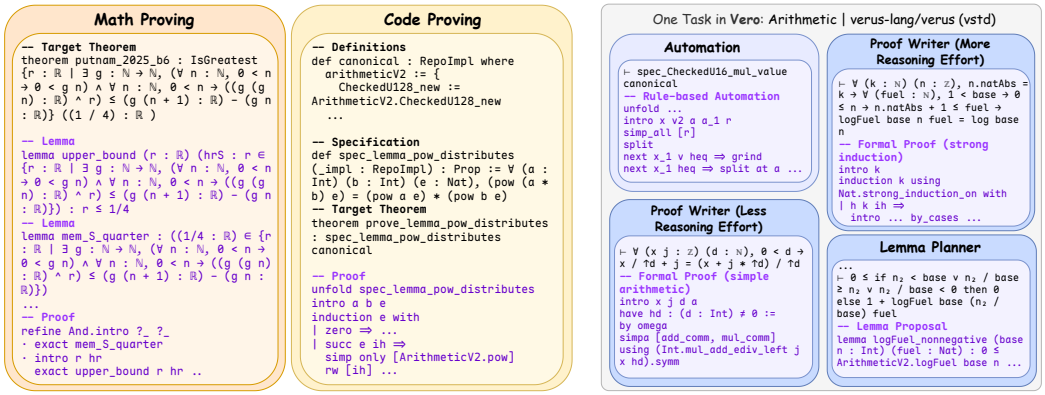
Program verification establishes software correctness through machine-checkable proofs constructed in theorem provers. It's a guarantee especially valuable for code generated by large language models (LLMs), which is fluent but carries no assurance of correctness. Almost all existing provers, however, pursue pass rates alone at whatever sampling or search budget it takes, and overlook the *success-vs-cost frontier*; yet real software often carries hundreds of interdependent proof obligations, so what matters at scale is not whether one theorem can be proved, but how many can be proved economically. We introduce *CoCo-Prover*, which formalizes cost-efficient program proving as *metalevel decision-making under cost, grounded on two-level proof graphs*: an AND/OR proof hypergraph within each declaration is joined to a lemma-dependency graph across declarations; and at each step, it answers two questions: which open goals to select, and which actions to purchase on these goals. Selection stays symbolic as a topological pass over the proof graphs. Action choice is agent orchestration via metalevel decision-making: an agentic router treats every bounded specialist invocation as a separately priced, best-effort computation, matching heterogeneous specialist agents together with configurations, under evolved routing rules as evidence accumulates. On five program verification benchmarks in Lean 4 including function-level CLEVER, VERINA, and AlgoVeri, and repository-level NTP4VC and Vero, we show that CoCo-Prover achieves a better success-vs-cost frontier than baselines including frontier coding agents and state-of-the-art LLM-based provers: it achieves the best solve rate on every benchmark and up to 100% on two benchmarks. It also reduces cost by up to 30.9% compared to the strongest baseline with the strongest LLM in our evaluation.

1 INTRODUCTION

Program verification uses formal mathematical reasoning to establish that a program satisfies specified correctness properties (Floyd, 1993; Hoare, 1969), providing guarantees that testing, which examines only finitely many executions, cannot offer. Historically, obtaining such guarantees has been expensive, requiring substantial expert effort to construct machine-checkable proofs in interactive theorem provers (ITPs) such as Lean 4 (Moura & Ullrich, 2021), Rocq (The Rocq Development Team, 2025), and Isabelle/HOL (Nipkow et al., 2002). Large language models (LLMs) create an opportunity to reduce this burden: while they can now generate sophisticated programs from natural-language specifications (Chen et al., 2021; Alharbi & Alshayeb, 2026), they may also automate the costly process of constructing formal proofs. This possibility has motivated *verified code generation* (Sun et al., 2024; Yang et al., 2024), where a model generates a program with a machine-checkable proof proving it satisfies the specification. Nevertheless, proof generation remains considerably harder than code generation shown in recent benchmarks VERINA (Ye et al., 2026b), CLEVER (Thakur et al., 2025), and AlgoVeri (Zhao et al., 2026a). Proof generation has therefore emerged as a central bottleneck in program verification and verified code generation.

Real software magnifies this bottleneck. A repository can contain hundreds or thousands of proof obligations with sharply different proof economics (Xin et al., 2026; Ye et al., 2026a). At the same time, a generated proof serves only to verify its program: the checker returns exactly one of two outcomes, and every accepted proof certifies equally. Thus, the natural objective for scalable program verification is the *success–vs–cost frontier*—the Pareto frontier between proof success and token

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.



(a) Math proving vs. code proving.

(b) Goals within one code-proving task.

Figure 1: **The difference between math proving and code proving; and the difference between goals within one code proving task.** (a) A theorem in math proving (theorem from PutnamBench (Tsoukalas et al., 2024), proof from GoedelArchitect (Chung et al., 2026)) is stated and proved in the vocabulary of an existing library and strongly dependent on lemma decompostion, whereas a theorem in code proving (theorem from Vero (Ye et al., 2026a)) is stated over definitions the task itself introduces so the proof follows program syntax rather than library priors. (b) Obligations of a single Vero task need different strategies and reasoning efforts: one goal closes under rule-based automation with no model call, two need formal proofs at different reasoning efforts (simple arithmetic versus strong induction), and a recurring fact is better stated as a shared helper lemma.

cost (*or* monetary expenditure practically), measured as success under matched cost limits and cost at matched success.

Recent LLM-based theorem provers have shown substantial promise, particularly in mathematical reasoning (Ren et al., 2025; Lin et al., 2026; Achim et al., 2025; Varambally et al., 2026; Chung et al., 2026). Code proving, however, presents a different challenge Figure 1a. Each task introduces a fresh formal context (*e.g.*, the implementation, specification, and often new data types and definitions) and proving correctness requires reasoning about the program's intermediate states. While program syntax often determines the overall proof structure and many side conditions can be discharged mechanically, a few critical obligations require discovering facts not explicit in the program or specification (*e.g.*, the right loop invariant, a sufficiently generalized induction hypothesis). These discoveries often determine whether the entire proof succeeds, yet they are precisely the steps that require substantial semantic reasoning.

This asymmetry creates a distinctive opportunity for code proving: most obligations are routine, but a small number are both difficult and decisive. Existing code provers (He et al., 2026; Tu et al., 2026; Zhao et al., 2026b; Li et al., 2026c), however, largely ignore this asymmetry, pursuing provability through large sampling or iterative repair budgets and applying similar model-driven procedures across subgoals. Such strategies are particularly costly with frontier general-purpose LLMs and waste substantial computation on obligations that theorem-prover automation or inexpensive model calls could solve directly (Figure 1). Effective code proving therefore requires not simply more reasoning, but identifying where deep reasoning matters and concentrating model computation there.

To address this challenge, we introduce CoCo-Prover, a cost-aware prover that adapts its strategy and reasoning effort to the evolving proof state. Rather than applying the same proof routine to every obligation, CoCo-Prover repeatedly decides what to prove next and how to prove it: whether an obligation can be discharged mechanically, needs direct proof generation, benefits from introducing a reusable helper lemma, or requires recovery from an unproductive proof path. It also controls how much reasoning effort to spend on each action. To make these decisions globally, CoCo-Prover maintains a two-level proof graph that tracks both fine-grained proof progress and dependencies among lemmas. We formalize this process as *metalevel decision-making under cost*: each proving action is a computation with a cost and an uncertain, verifier-judged outcome, and the prover must decide which computation is worth performing given the current proof state (Russell & Wefald,

1991). CoCo-Prover realizes this formulation through an agentic router and four specialized proving agents, using the proof graph and accumulated verifier feedback to adapt its strategy as proving progresses. This turns code proving from a fixed proof-search routine into adaptive orchestration of heterogeneous proving strategies under cost.

We evaluate CoCo-Prover on five Lean program verification benchmarks (function-level CLEVER (Thakur et al., 2025), VERINA (Ye et al., 2026b), and AlgoVeri (Zhao et al., 2026a), and repository-level NTP4VC (Xu et al., 2026) and Vero (Ye et al., 2026a)) against state-of-the-art open-source agentic provers, and each model's official coding agent, with three frontier LLMs under matched caps. No baseline exceeds CoCo-Prover's solve rate on any benchmark and backbone: it solves up to 100% of function-level benchmarks and 60.5% of the large repository-level benchmark Vero while reduces cost by up to 30.9% against the strongest baseline, Humanize on the strongest backbone LLM, GPT-5.6-Terra.

In summary, our contributions are: i) **Cost-efficient code proving.** We pose cost efficiency, rather than provability alone, as the objective for theorem proving in program verification at the scale of real-world software. ii) **Metalevel formulation under cost over two-level proof graphs.** We are the first to formalize cost-efficient program proving as metalevel decision-making under cost, grounded on two-level proof graphs. And we instantiate it in CoCo-Prover with agent orchestration. iii) **Success–cost evaluation.** On five Lean benchmarks and three LLMs, no baseline exceeds CoCo-Prover's solve rate in any setting, and it reduces cost by up to 30.9% against the strongest baseline.

2 Related Work

LLM-based provers. Fine-tuned provers generate whole Lean proofs and buy success through *distributional sampling*: correct proofs are spread across many samples and extracted at pass@k budgets by redrawing on unchanged context (Lin et al., 2025; Wang et al., 2025; Lin et al., 2026), with decomposition, where present, confined to a single generation's chain of thought (Ren et al., 2025); refinement agents temper this with verifier feedback yet still spend fixed per-theorem iteration budgets (Thakur et al., 2024; Requena Pozo et al., 2026; Zhou et al., 2025; Ma et al., 2026; Liu et al., 2026b). CoCo-Prover instead treats every bounded invocation as best-effort given its evidence and state, and resamples only as a separate decision justified by changed evidence. Structured provers including stepwise search over AND/OR tactic states (Lample et al., 2022; Hubert et al., 2026; Limperg & From, 2023) and lemma decomposition through pools, sketches, or blueprints (Chen et al., 2025b;a; Achim et al., 2025; Varambally et al., 2026; Chung et al., 2026; Zhang et al., 2026a) suffer from *coarse rollback or rare reuse*: a falsified conjecture triggers wholesale sketch or blueprint regeneration or outright branch failure, the rollback unit being the whole attempt rather than exactly the invalidated dependents, and helpers are rarely shared across distinct targets. CoCo-Prover's persistent two-level proof graph instead provide fine-grained control over the rollback unit and the helper reuse. Code provers inherit both modes: NSPG runs one stepwise search per obligation under a flat timeout (He et al., 2026), AutoRocq maintains a per-lemma proof tree (Tu et al., 2026), LemmaNet discovers helper lemmas from program semantics into a flat corpus (Zhao et al., 2026b), and Goedel-Code-Prover decomposes each verification task in isolation (Li et al., 2026c), while the general coding agents merge these techniques but cost much (Sosso et al., 2026; Thakur et al., 2025).

Cost-aware inference and agent orchestration. Query-level routers and cascades choose a model from surface features of an isolated query (Chen et al., 2023; Ong et al., 2025), and budget-aware tree search anneals exploration within a single query's reasoning tree under an LLM critic (Li et al., 2026b). In the area of formal reasoning and theorem proving: Optimizing-Lean-Agent (Rögnvaldsson et al., 2026) aims for math proving by routing over a binary action space using a logistic model over surface features of the failed attempts. This coarse continue-or-restart routing selects no goal and no proof strategy, and never conditions on decomposition structure or on other obligations. EconProver (Li et al., 2026a) also targets math proving, but on the training side: reinforcement learning teaches a single prover to switch reflective chain-of-thought off on easy theorems and to diversify its parallel samples, so they decide neither which obligation to work on nor which kind of action to buy. Danus (Liu et al., 2026a) maintains a fact graph with transitive revocation for mathematical research agents but routes work to homogeneous general-purpose agents rather than specialists matched to goal types and makes no cost decision over that structure.

3 METHODOLOGY

3.1 PROBLEM FORMULATION

We formulate cost-efficient program proving as *metalevel decision-making under cost*. At each step, the prover must determine (1) which goals to work on next, and (2) what proving strategy and configuration to use. The first is structural and the latter one requires semantic reasoning and incur model cost. Importantly, cost is considered globally at the task level: an expensive action on one goal may reduce total cost, for example by producing a helper lemma reused by several others. We measure cost as monetary expenditure rather than token count, since different token types are priced differently.

Proof state and structural control. Given a fixed Lean environment Γ_0 and target theorems $L_\star = \{\ell_1, \ldots, \ell_n\}$, the prover maintains the two-level proof graph in Figure 2. Within each declaration, an AND/OR hypergraph records verified proof steps and their resulting subgoals; across declarations, a dependency graph connects target and helper lemmas. Together, they preserve partial progress, expose reusable results, and track which work depends on provisional lemmas.

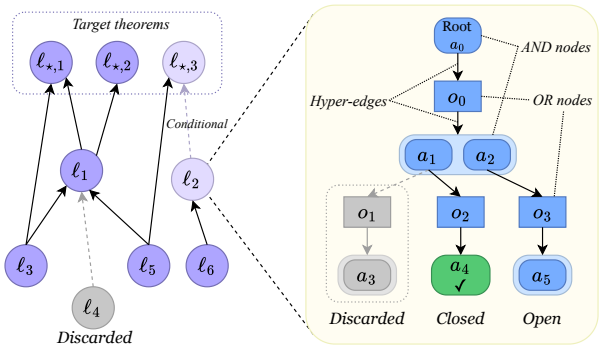


Figure 2: **The two-level proof graph.** *Left*: the lemma-dependency DAG $\mathcal{D} = (L, E_\mathcal{D})$, with an edge $u \to v$ when the proof or plan of v uses helper lemma u, the targets $L_\star \subseteq L$ as roots, and each lemma stable, provisional, or discarded. *Right*: within one declaration, the AND/OR hypergraph $\mathcal{H}_\ell = (A_\ell, O_\ell, E_\ell)$ over goal states (AND nodes A_ℓ) and proof steps (OR nodes O_ℓ), where an executed step attaches a verified hyperedge in E_ℓ over the subgoals Lean returns.

At decision epoch t,

$$\begin{aligned} s_t &= (\mathfrak{G}_t, \mathcal{F}_t, \sigma_t, \mathcal{M}_t, \mathcal{L}_t), \\ U_t &= \mathrm{Sched}(s_t), \end{aligned} \tag{1}$$

where $\mathfrak{G}_t = (\mathcal{D}_t, \{\mathcal{H}_{\ell,t}\}_{\ell \in L_t})$ is the two-level graph, σ_t records artifact status, $\mathcal{M}_t$ is shared memory, $\mathcal{L}_t$ is the computation ledger, and $\mathcal{F}_t$ is the *dependency-ready frontier*: open goals whose active prerequisites have been constructed. A lemma becomes stable once all of its prerequisites are stable; goals depending on provisional lemmas may still proceed, with those dependencies tracked for rollback (Section B.5). An incremental topological pass in the style of Kahn's algorithm (Kahn, 1962) maintains $\mathcal{F}_t$ as lemmas are added, stabilized, or retracted. The scheduler exposes $U_t = \mathcal{F}_t$, restricting the action space without spending model computation.

Action selection as purchased computation. Given U_t, the router decides *where and how to spend computation*: it selects a group of goals, a proving strategy, and its reasoning budget. Following metareasoning principles (Russell & Wefald, 1991), we represent this as

$$c_t = (G_t, o_t, \theta_t) \in \mathcal{C}(s_t, U_t), \qquad Y_t \sim P_{c_t}(\cdot \mid s_t, U_t), \qquad s_{t+1} = T(s_t, U_t, c_t, Y_t), \tag{2}$$

where $G_t \subseteq U_t$ is the selected goal group, o_t is a specialist agent, and θ_t specifies its configuration, such as reasoning effort or refinement budget. The outcome Y_t incurs cost $\kappa_t = \kappa(s_t, U_t, c_t, Y_t)$. The router's own inference cost is accounted for separately.

Metalevel objective. The cheapest immediate action is not necessarily the most cost-efficient: a more expensive computation may increase the chance of completing the task or reduce future work. For $\lambda \geq 0$, we therefore score a run by $\mathbf{1}[\mathrm{solve}] - \lambda C$, where C is total realized cost. The optimal value satisfies

$$Q^*_\lambda(s, U, c) = \mathbb{E}_Y\left[-\lambda\kappa + V^*_\lambda(s')\right], \quad V^*_\lambda(s) = \max_{c \in \mathcal{C}(s,U)} Q^*_\lambda(s, U, c), \tag{3}$$

writing $\kappa = \kappa(s, U, c, Y)$ and $s' = T(s, U, c, Y)$, with $U = \mathrm{Sched}(s)$, $V^*_\lambda = 1$ on solved states, and a free stop action of value 0. This objective values a computation by both its immediate cost and

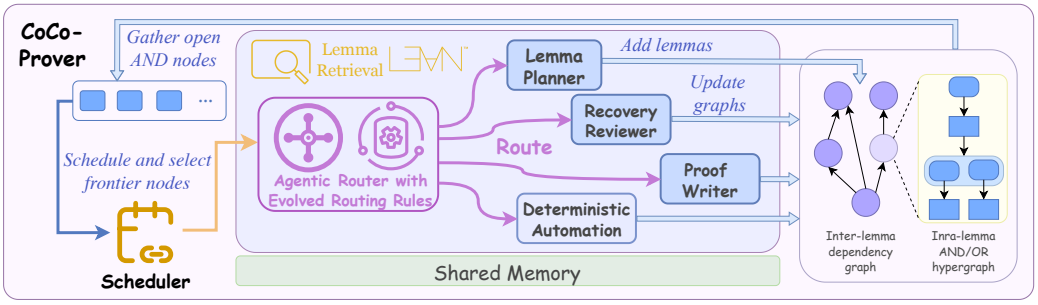


Figure 3: **Thw workflow of CoCo-Prover.** *Left*: the scheduler gathers open AND nodes from the proof graphs and hands the router every dependency-ready goal, grouping none of them. *Middle*: the agentic router clusters the frontier it receives, then reasons about one group and routes it to one of four specialist agents together with the configuration. The routing rules are revised by the router itself as evidence packets accumulate. The agents alone interact with Lean checker, lemma retrieval, and shared memory. *Right*: every agent writes back to the two-level graphs.

its effect on the remaining verification task. Q^*_λ is normative rather than computed explicitly. The router uses the current proof state and accumulated evidence to approximate its induced preference over computations. We operate at a small positive λ, so successful verification dominates while cost distinguishes actions with comparable prospects of success. Finally, the regret of a routing decision is

$$\delta_\lambda(s, U, c) = V^*_\lambda(s) - Q^*_\lambda(s, U, c) \geq 0. \tag{4}$$

Adaptive routing is beneficial when the decision regret it avoids exceeds its own inference overhead; we formalize this result in Section A.

3.2 System Design of CoCo-Prover

Guided by the metalevel analysis of Section 3.1, CoCo-Prover is an agent framework built around one principle: LLM agents make the semantic decisions and dispatch the decisions to the specialist agents as shown in Figure 3. CoCo-Prover maintains the two-level proof graph together with a shared memory of artifact status, failure records, and the realized resource ledger. A *scheduler* selects the frontier goals on AND nodes symbolically according to discussion in Section 3.1. An *agentic router* reasons about the selected goals and dispatches them to one of four specialist action agents. Each action agent writes its verified transitions, diagnostics, and realized cost back to the graph and shared memory.

Evidence-grounded agentic router. The scheduler hands the router the whole dependency-ready frontier of open AND nodes. The router first groups it using k-means clustering as described in Section B.4, so that semantically related goals are routed together. A static-analysis check then runs before any model call: a goal that is *definition-nested* is sent directly to Deterministic Automation. At each decision point it receives an *evidence packet*, instantiating $\mathrm{ev}(s, U)$: the selected Lean goals with their tactic states and relevant definitions, snapshots of earlier proof states, the dependent provisional lemmas, and the execution history with Lean diagnostics and counterexamples from the shared memory, together with its current routing rules. The router approximates the metalevel ordering of Equation (3) in the small-λ regime of Proposition A.1. From this packet it returns (G, o, θ, j): the group G of the goals it decides to spend on next, drawn from its own clustering of the frontier; a specialist option o; a configuration θ which sets the invocation's reasoning effort and working budget (*e.g.*, the maximum number of refinement iterations); and the preliminary analysis j it produces, so that the action agents can continue the reasoning from it. The routing rules are initialized with one priority, *same provability, cheap-first*: among the actions equally likely to leave every remaining root provable, take the one with the lowest total expected cost of finishing them all. Both terms are task-level rather than goal-level, which means the comparison is predicted over the whole remaining task. This explains why proposing a helper lemma can be the cheapest choice when several goals will use it, even though proving it once costs more than one direct proof. The initial rules are deliberately simple: a short natural-language description of each specialist including what kind of goal it suits together with the priority and a few worked routing examples. The rules

then *evolve*: after observing an evidence packet and the outcome of its previous decision, the router may amend them. For instance, if the Proof Writer repeatedly fails on a kind of goal, the rules may evolve to increase reasoning effort or propose a shared helper lemma to decompose the goal.

Specialist agents. The four specialists exist to handle different kinds of proof goals. ***Deterministic Automation:*** this action doesn't involve any model call. It extends Aesop Limperg & From (2023), a white-box rule-based proof search methods, which automatically tries different tactics in Lean to solve the current goal. ***Proof Writer:*** it's an agent interacting with Lean checker. It first continue on the reasoning results from the router and then write formal Lean code under a iterative refinement budget from the router. When the budget is exhausted, it keeps the partial successful proof and update the proof graphs. ***Lemma Planner:*** it proposes typed helper lemmas with explicit root-relevant consumers. A candidate is added to the dependency graph only after it elaborates and preserves acyclicity, it is checked by both random testing and LLM-based structural checking. Its proof obligations enter the frontier as new open goals, and control returns to the router with analysis tags. ***Recovery Reviewer:*** it diagnoses why an attempt failed, distinguishing two cases. First, the goal may be *unprovable*: a wrong proof step or an incorrect helper lemma can leave a goal that is false, so the reviewer constructs concrete instances and evaluates them to look for a counterexample to disprove the goal. Second, the goal may be provable but on a *hard path*: the current route has grown expensive while a cheaper one likely exists, so the reviewer compares continuing with switching on the work each still requires: the expected remaining cost, revalidation included, and whether either route looks less likely to close at all. The judgment is the model's rather than a certified optimality test, since a cheaper route can also be a less provable one. When it decides to recover, it backtracks to the best point in the proof graphs (*i.e.*, the nearest state whose verified work does not depend on the failure) so only the invalidated work is discarded.

4 EVALUATION

In this section, we first evaluate CoCo-Prover on the performance of solve rate, cost and success-vs-cost frontier under a budget sweep. And then we conduct ablation studies to show the contribution of different components. Section C details the experiment setup, and gives further results. Besides, we select some cases to showcase the workflow and benefits of CoCo-Prover in Section D.

Benchmarks. We evaluate on five Lean 4 program verification benchmarks. Three are function-level: CLEVER (Thakur et al., 2025), VERINA (Ye et al., 2026b), and AlgoVeri (Zhao et al., 2026a), which provide function-level verifiable code generation problems. Two are repository-level, where obligations are derived from upstream code repositories. NTP4VC (Xu et al., 2026) focuses on verification conditions, generated from C and Why3 programs, and released in Lean 4, Rocq, and Isabelle, where we use the Lean 4 track. Vero (Ye et al., 2026a) is a repository-level verifiable code generation benchmark, where we use proof-only tasks. The target theorems are constructed as follows. VERINA and Vero ship both ground-truth specifications and ground-truth implementations, which we use the formed theorem tasks directly. CLEVER and AlgoVeri provide ground-truth specifications but no ground-truth implementations, so we generate each implementation with GPT-5.6-Sol at high reasoning effort and manually check its correctness against the specification before any prover runs; the checked implementations are then fixed and shared by all methods. NTP4VC's theorems are already well-formed verification conditions but it's a large benchmark so we only use the first 300 tasks in alphabetical order. In every case the theorem statements are fixed inputs and code or specification synthesis is never a part of the measured task.

Baselines. We compare against three baseline families under identical backbones, providers, and caps. We use three frontier models: Gemini-3.5-Flash, GPT-5.6-Terra, and DeepSeek-V4-Flash. *Ax-Prover-Base* (Requena Pozo et al., 2026) is a minimal iterative agent—propose a Lean proof, compile, review, and retry with a memory of recent failures and library search—representative of refinement agents and reported to be strongly cost-effective relative to repeated sampling. *Humanize* (Humanfia, 2026) is an open-source state-of-the-art agentic Lean 4 prover based on coding agents like Codex OpenAI (2026) and Claude Code Anthropic (2026), which achieves 100% solve rate on PutnamBench (Tsoukalas et al., 2024). *Coding agents* pair each backbone with its official general-purpose agent interface, equipped with tooling (*i.e.,* Lean toolchains, retrieval)—Codex for GPT model, Gemini CLI (Gemini CLI, 2026) for Gemini model, and DeepSeek Harness (DeepSeek, 2026) for DeepSeek model. Ax-Prover-Base and Humanize prove one self-contained theorem per

Table 1: **Solve rates across function- and repository-level benchmarks.** A task counts as solved only when Lean accepts the root declaration and no generated axioms are used.

Solve-Rate Comparison Across 2 Repository-Level Benchmarks						
	GPT-5.6-Terra		Gemini-3.5-Flash		DeepSeek-V4-Flash	
Method	**NTP4VC**	**Vero**	**NTP4VC**	**Vero**	**NTP4VC**	**Vero**
Coding Agent	85.3%	2.3%	79.7%	7.0%	39.0%	0%
CoCo-Prover	**89.3%**	**53.5%**	**94.7%**	**60.5%**	**95.0%**	**14.0%**

Solve-Rate Comparison Across 3 Function-Level Benchmarks									
	GPT-5.6-Terra			Gemini-3.5-Flash			DeepSeek-V4-Flash		
Method	**CLEVER**	**VERINA**	**AlgoVeri**	**CLEVER**	**VERINA**	**AlgoVeri**	**CLEVER**	**VERINA**	**AlgoVeri**
Ax-Prover-Base	89.4%	85.7%	81.8%	85.7%	72.5%	76.6%	73.9%	66.1%	58.4%
Humanize	97.5%	98.9%	85.7%	97.5%	95.2%	80.5%	91.9%	73.5%	66.2%
Coding Agent	91.9%	87.3%	67.5%	94.4%	87.8%	71.4%	78.9%	58.2%	79.2%
CoCo-Prover	**100%**	**100%**	**89.6%**	**100%**	**98.4%**	**80.5%**	**96.9%**	**100%**	**93.5%**

episode and do not operate over repository-scale dependency structure, so the repository-level comparison includes only the coding agents (Table 1). We do not compare against other open-source provers such as Hilbert (Varambally et al., 2026), Goedel-Code-Prover (Li et al., 2026c), Wybe-Coder (Gloeckle et al., 2026), and P^3 (Li et al., 2026d) because each is designed for a different setting, and Section C.1 gives the reason for each.

4.1 PERFORMANCE

Solve rates. As shown in Table 1, no baseline exceeds CoCo-Prover's solve rate in any benchmark–LLM pair. On function-level benchmarks with a frontier LLM there is little headroom: Humanize, the strongest baseline, already proves almost every CLEVER and VERINA problem, and CoCo-Prover closes the remainder to 100% on CLEVER and VERINA; on AlgoVeri with Gemini-3.5-Flash the two tie. This also reflects CoCo-Prover's advantage on different LLMs and its scalability. The second is repository scale: on Vero the coding agents solve at most 7.0% of tasks, against up to 60.5% for CoCo-Prover. Vero's obligations are spread across a repository and share definitions and helper lemmas, so a prover that treats all obligations in a single context will be overwhelmed, whereas CoCo-Prover uses proof graphs and agent orchestration to handle them effectively.

Cost at matched success. We compare cost against one baseline at a time, and only on the problems that this baseline and CoCo-Prover both solve. Costs on unsolved problems are not comparable: a method that gives up early, or that reward-hacks a hard problem such as closing it with an axiom, weakening or rewriting the theorem statement (*i.e.*, the behavior we observe most often in the coding agents) spends little precisely where an honest proof is expensive, so counting those runs would reward failure. On the commonly solved problems, CoCo-Prover is cheaper than every baseline in all nine function-level benchmark–LLM pairs (Table 2). The cost saving is largest against Ax-Prover-Base (up to 85.7%), which re-samples a whole proof after each failure, and against every baseline under DeepSeek-V4-Flash, where failures are most frequent. It is smallest against Humanize (from 1.3% to 30.9% under GPT-5.6-Terra and Gemini-3.5-Flash), the strongest baseline, since for most easy problems, Humanize and the basic coding agents can simply propose and refine proofs so that our gain is small. The one case where a baseline is cheaper is Vero with GPT-5.6-Terra, where Codex solves a single task that CoCo-Prover also solves. This is because this is only one task problem which only has a few easy obligations so it doesn't weaken CoCo-Prover's advantage. The two settings therefore differ in what CoCo-Prover buys. On isolated function-level obligations, it principally reduces proof cost. On repository-level verification, persistent proof structure can additionally change what is practically provable: on Vero, CoCo-Prover solves up to 26 of 43 tasks while the coding agents solve at most three, which is also why too few commonly solved tasks remain there for a meaningful cost comparison.

Success–cost frontier under a budget sweep. Figure 4 asks a complementary question to Table 2: sweeping the per-problem cap from \$0 to the \$30 of the main evaluation, how many problems does each method solve? For each LLM we sum the problems solved over the three function-level benchmarks (CLEVER, VERINA, and AlgoVeri) at every budget, so each curve is one method's

Table 2: **Cost comparison on commonly solved problems.** For each baseline, we compare expenditure only on the intersection of problems solved successfully by both CoCo-Prover and that baseline, yielding a like-for-like comparison of the cost of valid proofs.

Solve-Cost on CLEVER Benchmark (Total 161 Problems)

	GPT-5.6-Terra			Gemini-3.5-Flash			DeepSeek-V4-Flash		
Method	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**
Ax-Prover-Base	144	69.9	**63.0** ▼ 9.8%	138	364.6	**140.3** ▼ 61.5%	115	53.7	**14.3** ▼ 73.4%
Humanize	157	136.6	**94.4** ▼ 30.9%	157	271.9	**199.6** ▼ 26.6%	143	64.9	**29.0** ▼ 55.3%
Coding Agent	148	90.4	**72.5** ▼ 19.8%	152	258.7	**176.4** ▼ 31.8%	124	56.9	**24.1** ▼ 57.7%
CoCo-Prover	161	–	100.2	161	–	235.6	156	–	38.7

Solve-Cost on VERINA Benchmark (Total 189 Problems)

	GPT-5.6-Terra			Gemini-3.5-Flash			DeepSeek-V4-Flash		
Method	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**
Ax-Prover-Base	162	143.1	**106.0** ▼ 25.9%	137	376.5	**175.5** ▼ 53.4%	125	117.2	**16.7** ▼ 85.7%
Humanize	187	299.5	**244.5** ▼ 18.4%	180	498.9	**403.4** ▼ 19.1%	139	108.2	**25.4** ▼ 76.5%
Coding Agent	165	127.4	**111.8** ▼ 12.2%	166	407.4	**237.9** ▼ 41.6%	110	50.4	**15.2** ▼ 69.8%
CoCo-Prover	189	–	245.4	186	–	537.4	189	–	80.3

Solve-Cost on AlgoVeri Benchmark (Total 77 Problems)

	GPT-5.6-Terra			Gemini-3.5-Flash			DeepSeek-V4-Flash		
Method	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**
Ax-Prover-Base	63	113.8	**55.5** ▼ 51.2%	55	189.9	**90.0** ▼ 52.6%	45	17.4	**10.7** ▼ 38.4%
Humanize	66	67.4	**66.6** ▼ 1.3%	58	157.3	**112.1** ▼ 28.7%	51	29.9	**15.2** ▼ 49.1%
Coding Agent	52	35.3	**32.0** ▼ 9.5%	51	94.4	**79.8** ▼ 15.5%	61	58.1	**29.8** ▼ 48.6%
CoCo-Prover	69	–	74.6	62	–	167.7	72	–	48.2

Solve-Cost on NTP4VC Benchmark (Total 300 Problems)

	GPT-5.6-Terra			Gemini-3.5-Flash			DeepSeek-V4-Flash		
Method	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**
Coding Agent	247	186.5	**162.2** ▼ 13.0%	239	608.2	**343.9** ▼ 43.5%	117	30.7	**18.6** ▼ 39.6%
CoCo-Prover	268	–	241.3	284	–	619.9	285	–	116.1

Solve-Cost on Vero Benchmark (Total 43 Problems)

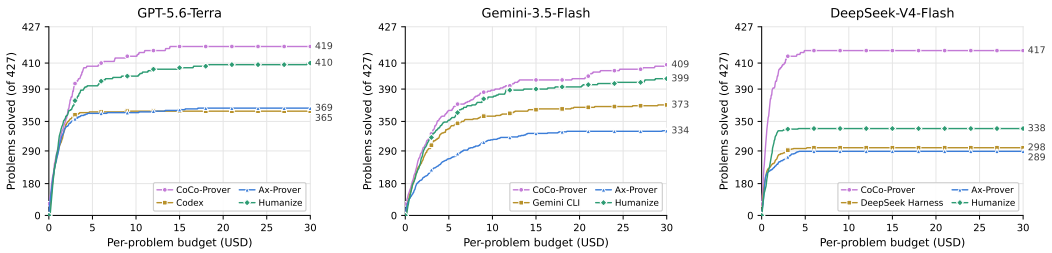
	GPT-5.6-Terra			Gemini-3.5-Flash			DeepSeek-V4-Flash		
Method	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**	**# Commonly Solved**	**Cost in USD (Baseline)**	**Cost in USD (CoCo-Prover)**
Coding Agent	1	1.6	**3.6** ▲ 131.2%	3	55.0	**24.6** ▼ 55.2%	0	0.0	0.0
CoCo-Prover	23	–	515.7	26	–	859.5	6	–	28.4

success–cost frontier. As shown in Figure 4, CoCo-Prover achieves the highest final solve count under all three backbones and leads over much of the evaluated budget range. Moreover, the baselines flatten early. Under GPT-5.6-Terra, Codex and Ax-Prover-Base gain relatively few additional solves beyond roughly $5 per problem, while CoCo-Prover reaches 419. This is because the extra money buys more of the same discarded work.

4.2 Ablation Study

In this section, we conduct some ablation studies to understand the contribution of each component in CoCo-Prover. Since rerunning all five benchmarks for every variant is too expensive, we use a fixed subset: 10 problems (randomly chosen, which span difficulty levels from easy to hard) from each of CLEVER, VERINA, and AlgoVeri, 30 in total.

We first conduct a run-to-run variance analysis to understand the stability of CoCo-Prover since model outputs are stochastic. We firstly run the complete CoCo-Prover three times with Gemini-3.5-Flash on 3 function-level benchmarks and then we run the complete CoCo-Prover three times with each backbone LLM on these 30 problems and details are shown in Section C.4.



(a) GPT-5.6-Terra. (b) Gemini-3.5-Flash. (c) DeepSeek-V4-Flash.

Figure 4: **Success–cost frontier on the 3 benchmarks.** Each curve gives the number of problems a method solves when every problem is capped at the per-problem budget on the horizontal axis, total 427 problems summed over CLEVER, VERINA, and AlgoVeri.

Table 3: **Component ablation across function-level benchmarks.** Using Gemini-3.5-Flash, we report total cost in USD and the number of problems solved (#S) on a subset of the problems. Parenthesized percentages give the cost change relative to the complete system.

Ablated Version	**CLEVER**		**VERINA**		**AlgoVeri**		**Total**	
	Cost (USD)	**#S**	**Cost (USD)**	**#S**	**Cost (USD)**	**#S**	**Cost (USD)**	**#S**
CoCo-Prover w/o Router	41.02 (+43.2%)	10	60.38 (+38.8%)	9	53.80 (+38.6%)	10	155.20 (+39.9%)	29
CoCo-Prover w/o Automation	31.93 (+11.5%)	10	48.89 (+12.4%)	9	36.57 (-5.8%)	10	117.39 (+5.8%)	29
CoCo-Prover w/o Lemma Planner	37.72 (+31.7%)	10	66.48 (+52.9%)	8	52.86 (+36.2%)	9	157.06 (+41.6%)	27
CoCo-Prover	28.64	10	43.49	10	38.82	10	110.94	30

Component ablation. To attribute the gains to individual components, we disable one component at a time and rerun CoCo-Prover on the 30 problems with Gemini-3.5-Flash. Table 3 compares each variant against the complete system, whose cost is the mean of its three Gemini-3.5-Flash runs in Table 7. *CoCo-Prover w/o Router* removes adaptive action choice: Deterministic Automation still runs first on every scheduled goal, and the action agents then follow one fixed order over the same two-level graph. Total cost rises by 39.9% and one VERINA problem is no longer solved. This is because the fixed routine purchases more unnecessary action invocations. *CoCo-Prover w/o Lemma Planner* removes shared-helper discovery, so every goal must be proved in isolation. It is the costliest variant, at 41.6% above the complete system, and it solves the fewest problems (27 of 30), losing two on VERINA and one on AlgoVeri. This mirrors the failure the case study isolates in the baselines (Section D): without lemma proposal, a missing fact cannot become a separately provable subgoal, so it is re-derived inside monolithic attempts instead of proved once and reused. *CoCo-Prover w/o Automation* removes the automation pass. Its effect is smaller: cost rises by 11.5% on CLEVER and 12.4% on VERINA, where many goals close mechanically, but falls by 5.8% on AlgoVeri, whose algorithmic goals rarely close by automation and frequent automation will split amonolithic goal into multiple meaningless subgoals, which increases cost. Overall, the component ablation study supports the theoretical analysis and system design of CoCo-Prover.

5 Conclusion

We presented CoCo-Prover, an agent-orchestration framework for cost-efficient theorem proving in program verification with Lean 4. CoCo-Prover formalizes proving as metalevel decision-making under cost, grounded on two-level proof graphs. A symbolic scheduler selects every dependency-ready goal, and an evidence-grounded router groups them and purchases actions from four bounded specialist agents with Lean as the sole proof authority. Across five Lean program verification benchmarks and three backbones, CoCo-Prover achieves highest solve rate in any setting, and it reduces cost on commonly solved problems by up to 30.9% against the strongest baseline, Humanize. Ablations also demonstrates the advantages of the system design with the shared-lemma discovery and adaptive routing.

AI USE STATEMENT

We use AI (LLM) as the following 4 roles. *Writing*: surveying and summarizing related work, reorganizing and editing manuscript drafts, and tightening exposition. *Implementation*: helping to draft and refactor parts of the implementation of CoCo-Prover and of the experiment tooling. *Data collection*: helping to build the scripts that run the experiments, parse execution traces, and aggregate token counts and costs into the reported tables. We also use frontier models to generate some implementation code for CLEVER and AlgoVeri to compose the target theorems, and we use the frontier models on coding agents to review the generated proof in evaluation. But these are further checked by human reviewers. *Figures and plots*: helping to polish the figures, plots, and code listings. All scientific claims, formal statements and proofs, citations, experimental designs, measurements, and conclusions were independently verified by the authors, who take full responsibility for the content of this paper.

ETHICS STATEMENT

This work involves no human subjects or personal data, and uses five public benchmarks under their licenses without redistributing them (Table 5). Formal verification is a defensive technology. Its main risk is misplaced trust, since a machine-checked proof certifies only its specification, and every reported result passes a review that rejects modified statements and added axioms (Section C.1).

REPRODUCIBILITY STATEMENT

We provide detailed engineering workflows in Section B, experimental settings in Section C and prompt templates in Section E for reproducing the implementation and results in the paper. We are also working on open-sourcing our code and data on GitHub.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: Imo-level automated theorem proving, 2025. URL https://arxiv.org/abs/2510.01346.

Maha Alharbi and Mohammad Alshayeb. Automatic code generation techniques: A systematic literature review. *Automated Software Engineering*, 33(1):4, 2026.

Anthropic. Claude code. https://claude.com/product/claude-code, 2026.

Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 1995.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025a. URL https://arxiv.org/abs/2512.17260.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance, 2023. URL https://arxiv.org/abs/2305.05176.

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, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen

Zhu. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025b. URL https://arxiv.org/abs/2507.23726.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

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, 2026. URL https://arxiv.org/abs/2606.06468.

DeepSeek. Deepseek harness. https://www.deepseek.com/harness/en/, 2026.

Robert W. Floyd. *Assigning Meanings to Programs*, pp. 65–81. Springer Netherlands, Dordrecht, 1993. ISBN 978-94-011-1793-7. doi: 10.1007/978-94-011-1793-7_4. URL https://doi.org/10.1007/978-94-011-1793-7_4.

Gemini CLI. Gemini cli. https://geminicli.com/, 2026.

Fabian Gloeckle, Mantas Baksys, Darius Feher, Kunhao Zheng, Amaury Hayat, Sean B. Holden, Gabriel Synnaeve, and Peter O'Hearn. Wybecoder: Verified imperative code generation, 2026. URL https://arxiv.org/abs/2603.29088.

Baoding He, Zenan Li, Wei Sun, Yuan Yao, Taolue Chen, Xiaoxing Ma, and Zhendong Su. Neuro-Symbolic proof generation for scaling systems software verification. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 2533–2550, Seattle, WA, July 2026. USENIX Association. ISBN 978-1-939133-55-7. URL https://www.usenix.org/conference/osdi26/presentation/he-baoding.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

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(8106):607–613, 2026. ISSN 1476-4687. doi: 10.1038/s41586-025-09833-y. URL https://doi.org/10.1038/s41586-025-09833-y.

Humanfia. Humanize. https://github.com/humanfia/humanize, 2026.

A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962. ISSN 0001-0782. doi: 10.1145/368996.369025. URL https://doi.org/10.1145/368996.369025.

Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pp. 267–274, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. ISBN 1558608737.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 26337–26349. Curran Associates, Inc., 2022. doi: 10.52202/068431-1910. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/a8901c5e85fb8e1823bbf0f755053672-Paper-Conference.pdf.

Mukai Li, Linfeng Song, Zhenwen Liang, Jiahao Xu, Shansan Gong, Qi Liu, Haitao Mi, and Dong Yu. EconProver: Towards more economical test-time scaling for automated theorem proving. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 45741–45753, San Diego, California, United States, July 2026a. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.2121. URL https://aclanthology.org/2026.acl-long.2121/.

Yushu Li, Wenlong Deng, Jiajin Li, and Xiaoxiao Li. Spend less, reason better: Budget-aware value tree search for llm agents, 2026b. URL https://arxiv.org/abs/2603.12634.

Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-code-prover: Hierarchical proof search for open state-of-the-art code verification, 2026c. URL https://arxiv.org/abs/2603.19329.

Zenan Li, Ziran Yang, Peiyang Song, Zhaoyu Li, and Kaiyu Yang. P^3: Joint program-and-proof planning for verified code generation, 2026d. URL https://arxiv.org/abs/2608.09277.

Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pp. 253–266, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700262. doi: 10.1145/3573105.3575671. URL https://doi.org/10.1145/3573105.3575671.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover: A frontier model for open-source automated theorem proving, 2025. URL https://arxiv.org/abs/2502.07640.

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 C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 11793–11818, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/13e8be77982beb73d7ed0bbf122f9f3c-Paper-Conference.pdf.

Jihao Liu, Guoxiong Gao, Zeming Sun, Bin Wu, Shurui Liu, Jiedong Jiang, Haocheng Ju, Leheng Chen, Ronnie Cheng, Xiping Zhang, and Bin Dong. Danus: Orchestrating mathematical reasoning agents with fact-graph memory, 2026a. URL https://arxiv.org/abs/2607.06447.

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics, 2026b. URL https://arxiv.org/abs/2601.14027.

Jialin Lu, Kye Emond, Kaiyu Yang, Swarat Chaudhuri, Weiran Sun, and Wuyang Chen. Lean finder: Semantic search for mathlib that understands user intents. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 155005–155040, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/fad8e1915f66161581bb127ccf01092e-Paper-Conference.pdf.

David Ma, Kaijing Ma, Shawn Guo, Yunfeng Shi, Enduo Zhao, Jiajun Shi, Zhaoxiang Zhang, Gavin Cheung, Jiaheng Liu, and Zili Wang. Oprover: A unified framework for agentic formal theorem proving, 2026. URL https://arxiv.org/abs/2605.17283.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

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

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data, 2025. URL https://arxiv.org/abs/2406.18665.

OpenAI. Codex. https://chatgpt.com/codex/, 2026.

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.

Borja Requena Pozo, Austin Letson, Krystian Nowakowski, Izan Beltran Ferreiro, and Leopoldo Sarra. A minimal agent for automated theorem proving. 2026. URL https://arxiv.org/abs/2602.24273.

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49(1):361–395, 1991. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(91)90015-C. URL https://www.sciencedirect.com/science/article/pii/000437029190015C.

Kári Rögnvaldsson, Chenhao Sun, Jasper Dekoninck, and Martin Vechev. Optimizing the cost-quality tradeoff of agentic theorem provers in lean, 2026. URL https://arxiv.org/abs/2606.04883.

Sho Sonoda, Shunta Akiyama, and Yuya Uezato. Why agentic theorem prover works: A statistical provability theory of mathematical reasoning models, 2026. URL https://arxiv.org/abs/2602.10538.

Alessandro Sosso, Akhil Arora, and Bas Spitters. Agentic proving for program verification, 2026. URL https://arxiv.org/abs/2605.23772.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In Guy Avni, Mirco Giacobbe, Taylor T. Johnson, Guy Katz, Anna Lukina, Nina Narodytska, and Christian Schilling (eds.), *AI Verification*, pp. 134–155, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-65112-0.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving, 2024. URL https://arxiv.org/abs/2310.04353.

Amitayush Thakur, Jasper 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 D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-3857. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a7f67788f7b4d77fa7cd6887de3dcbe7-Paper-Datasets_and_Benchmarks_Track.pdf.

The Rocq Development Team. *The Rocq Prover Reference Manual*, 2025. URL https://rocq-prover.org.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 11545–11569. Curran Associates, Inc., 2024. doi: 10.52202/079017-0368. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/1582eaf9e0cf349e1e5a6ee453100aa1-Paper-Datasets_and_Benchmarks_Track.pdf.

Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. Agentic verification of software systems. *Proc. ACM Softw. Eng.*, 3(FSE), June 2026. doi: 10.1145/3808164. URL https://doi.org/10.1145/3808164.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 38998–39031, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/41b95e8ee0e054dab145a7564d9cd58a-Paper-Conference.pdf.

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, 2025. URL https://arxiv.org/abs/2504.11354.

Yutong Xin, Qiaochu Chen, Greg Durrett, and Işil Dillig. Verisoftbench: Repository-scale formal verification benchmarks for lean, 2026. URL https://arxiv.org/abs/2602.18307.

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 C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 39209–39235, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/41efc6e1f29cf7c6bf7c6d9909850761-Paper-Conference.pdf.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai, 2024. URL https://arxiv.org/abs/2412.16075.

Zhe Ye, Hantao Lou, Yuechun Sun, Peiyang Song, Zhengxu Yan, Timothe Kasriel, Qingyang Zhang, Kaiyu Yang, Soonho Kong, Jingxuan He, and Dawn Song. Vero: Can ai agents build formally verified software repositories?, 2026a. URL https://arxiv.org/abs/2608.13522.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 38933–38972, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/41b8c80f9113b9f8e2e129447221682a-Paper-Conference.pdf.

Ning Zhang, Nongyu Di, Zenan Li, Yuan Yao, and Xiaoxing Ma. Planning to hammer: Difficulty-aware decomposition for automating rocq proofs, 2026a. URL https://arxiv.org/abs/2606.17981.

Youyuan Zhang, Jialiang Sun, Hangrui Bi, Chuqin Geng, Wenjie Ma, Zhaoyu Li, and Xujie Si. Dreamprover: Evolving transferable lemma libraries via a wake-sleep theorem-proving agent, 2026b. URL https://arxiv.org/abs/2604.26311.

Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V. Veeravalli, Aarti Gupta, and Sanjeev Arora. Algoveri: An aligned benchmark for verified code generation on classical algorithms, 2026a. URL https://arxiv.org/abs/2602.09464.

Huan Zhao, Haoxin Tu, Zhengyao Liu, Martin Rinard, and Abhik Roychoudhury. Automated lemma discovery in agentic program verification, 2026b. URL https://arxiv.org/abs/2603.22114.

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, Haocheng Wang, Luong Trung, Rong Ye, Phan Nhat Hoang, Huishuai Zhang, Peng Sun, and Hang Li. Solving formal math problems by decomposition and iterative reflection, 2025. URL https://arxiv.org/abs/2507.15225.

A Additional Theoretical Analysis

Roadmap. Section A.1 states the metalevel process of Section 3.1 in full, and Section A.2 proves Theorem 1 and Corollary 2. Section A.1 also names the operating point our protocol targets and shows that, in a finite abstraction of the process, the induced ordering is constant there (Propositions A.1 and A.2); that the deployment realizes the abstraction is assumed, not shown. Sections A.3 to A.7 establish structural properties of the two-level representation: it preserves reachability, graph information weakly—and on some instances strictly—raises the attainable value, the hard filters preserve the optimal value under their stated assumptions, sharing a helper can save a factor K, and returned proofs are kernel-sound.

A.1 The metalevel process

States, selection, and computations. A metalevel state is $s = (\mathfrak{G}, \mathcal{F}, \sigma, \mathcal{M}, \mathcal{L})$ as in Equation (1). It doesn't invovle the router's rules. The analysis below needs only a common environment to every policy. Theorem 1 conditions on the history, so a router carrying internal state is covered. Sched maintains the dependency-ready frontier (Section B.3) and hands the whole $U = \mathcal{F}$ to the router. The admissible computations $\mathcal{C}(s, U)$ are the triples (G, o, θ) of a group G. The epoch's update records it in $\mathcal{M}'$ afterwards. The contracts fix a finite grid of reasoning-effort and working-budget settings. That grid alone does not make $\mathcal{C}(s, U)$ finite, since θ also carries text: we assume in addition that every text-valued field of the action (*e.g.*, the analysis j and the prepared context alike) is drawn from a finite vocabulary under a fixed token bound, which the specialist contracts impose. Under that assumption, $\mathcal{C}(s, U)$ is finite and the maximum in Equation (3) is attained.

Outcomes, costs, and transitions. A decision epoch begins at the pre-routing state s and ends at s', and two different things happen inside it. The first is *deliberation*: the router's calls, the decisions its guards reject, and the amendment of its own rules. This is policy-internal and leaves no trace in the state at all. Its charge h goes to the router's own spending record. Over a run these charges accumulate as H_π. The second is the purchased computation itself. The environment returns $Y \sim P_c(\cdot \mid s, U)$: the specialist's output (*e.g.*, a proof script, proposed helper lemmas with their elaboration results, a diagnosis, or a recovery proposal) together with Lean's verdict on it. Confining Y to the environment's response is what makes P_c common to every policy. Its execution cost is $\kappa(s, U, c, Y)$, which never includes deliberation. Total spending is $C = \sum_{t<\tau} \kappa_t + H_\pi$, so each charge is counted once. The controller and Lean then apply the update $s' = T(s, U, c, Y)$. We assume:

(A1) *Markov outputs*: given (s, U, c), the law of Y depends neither on the rest of the history nor on which router produced c. This holds because everything the specialist is shown is carried by c;

(A2) *bounded computations*: $\kappa \le \bar{c}$ for every computation, and $\kappa \ge \kappa_{\min} > 0$ for every paid one, that is, one that invokes a model. Some computations have $\kappa = 0$ like Deterministic Automation;

(A3) *verifier determinism*: given (s, U, c, Y), the update T is deterministic. Neither rule amendment nor deliberation spending needs a clause here.Sched and the router's grouping are deterministic.

The stop computation costs nothing and moves to an absorbing state of value 0. Free computations would allow a policy to cycle forever at no charge, so throughout we restrict to *proper* policies, those reaching an absorbing state with probability one and in finite expected time; the global iteration limit of Algorithm 1 enforces this in the implementation. A solved state, in which every root and every generated dependency of its proof are stable, is absorbing with value 1.

Value and Bellman equation. For a policy π choosing $c_t \in \mathcal{C}(s_t, U_t)$ from the history, let τ be the number of computations before it enters an absorbing state, and let $C = \sum_{t<\tau} \kappa_t + H_\pi$ be its total realized cost, including its own inference cost H_π. Its value is

$$V_\lambda^\pi(s_0) = \Pr_\pi[\text{solve}] - \lambda\, \mathbb{E}_\pi[C], \tag{5}$$

and $V_\lambda^* = \sup_\pi V_\lambda^\pi$, taken over policies with zero inference cost, so that the optimum charges nothing for deliberation. Because stopping is always admissible, $0 \le V_\lambda^* \le 1$ holds here. Writing $W = 1 - V$ turns the problem into one with nonnegative stage costs $\lambda\kappa$ and terminal cost 0 (solved) or 1 (stopped). Under this positivity condition, the optimal value satisfies the Bellman equation (3) (Bertsekas, 1995).

The frontier and the evaluation metric. Each policy induces a point $(\Pr_\pi[\text{solve}], \mathbb{E}_\pi[C])$. Maximizing Equation (5) for a fixed λ selects a point on the success–cost boundary of the achievable set, and sweeping λ recovers its supported points, *i.e.*, those on its convex hull. The evaluation instead reports the threshold metric $\mathrm{Pass}_R(B)$ (*i.e.*, the fraction of tasks a policy R solves before its cumulative cost reaches an external cap B that is never given to the controller) together with cost on commonly solved problems. V_λ^* is the normative object behind these measurements, not a quantity we estimate.

The router as an ordinal approximation. At each state, Q_λ^* induces a latent ordering of the admissible computations, $c \succ_s c'$ iff $Q_\lambda^*(s, U, c) > Q_\lambda^*(s, U, c')$, and an optimal policy picks a maximal element. The router is asked to predict a maximal element of this ordering from $\mathrm{ev}(s, U)$.

The operating point. Our protocol does not choose an interior trade-off between success and cost. The caps of are set generously and the prover works best-effort until it proves the target or exhausts its admissible computations. The preference is a small but positive λ one, which means success dominates, and cost separates continuations that are equally likely to succeed. The cap B is a safety limit rather than a control input. It is never given to the controller, and $\mathrm{Pass}_R(B)$ records how often best-effort proving finishes beneath it. The Proposition A.1 shows that, in a finite abstraction of this process, it is the regime (not a numerical value of λ) that determines the ordering the router has to predict.

Proposition A.1 (Constant ordering in the small-λ regime)**.** *Assume (A1)–(A3) and restrict to proper policies. Assume further that the reachable state space is finite; $\mathcal{C}(s, U)$ is finite by the contracts. Then there is $\lambda_0 > 0$ such that for every state s, every frontier U, and every $c, c' \in \mathcal{C}(s, U)$, the sign of $Q_\lambda^*(s, U, c) - Q_\lambda^*(s, U, c')$ is the same for all $\lambda \in (0, \lambda_0]$. The ordering $\succ_s$ is therefore one fixed ordering on that interval, and its maximal elements are the computations that first maximize the attainable success probability and, among those, minimize expected remaining cost.*

Proof. Under the finiteness hypothesis, the metalevel process is a finite stochastic shortest-path problem over proper policies, for which deterministic stationary policies suffice (Bertsekas, 1995). There are finitely many of them and each one is proper, so its success probability $p_\pi(s)$ and expected cost $c_\pi(s)$ are finite. By Equation (5) each has value $V_\lambda^\pi(s) = p_\pi(s) - \lambda\, c_\pi(s)$, affine in λ, so $V_\lambda^*(s) = \max_\pi V_\lambda^\pi(s)$ is a maximum of finitely many affine functions: piecewise linear and convex in λ, with finitely many breakpoints. Since $Q_\lambda^*(s, U, c) = -\lambda\, \mathbb{E}_Y[\kappa] + \mathbb{E}_Y\left[V_\lambda^*(T(s, U, c, Y))\right]$ is an affine term plus a finite convex combination of such functions, it is piecewise linear in λ with finitely many breakpoints, and so is every difference $Q_\lambda^*(s, U, c) - Q_\lambda^*(s, U, c')$. Each difference therefore changes sign at finitely many values of λ. As there are finitely many tuples (s, U, c, c'), taking λ_0 below the smallest positive such value gives the first claim. For the second, note that $Q_\lambda^*(s, U, c)$ is itself the value of the best continuation that begins with c, hence of the form $p(s, U, c) - \lambda\, \bar{c}(s, U, c)$ on each linear piece, where p is the success probability attainable after c and $\bar{c}$ is the expected remaining cost of the continuation attaining it. Letting $\lambda \downarrow 0$, a computation whose p is not maximal is

strictly beaten for all small enough λ, and among those attaining the largest p the value is decreasing in $\bar{c}$, so the maximizers on $(0, \lambda_0]$ are exactly the lexicographic ones. □

Proposition A.2 (The routing rules target this ordering)**.** *Fix $\lambda \in (0, \lambda_0]$.* (i) *Among computations that leave the attainable success probability unchanged, $\succ_s$ ranks by expected remaining cost, so when no admissible computation raises that probability the cheapest is maximal. In particular, let c close the remaining work outright with probability p_c at cost κ_c and, under the proviso of Section A.5, leave the state value-equivalent when it fails. Write $J^*(s)$ for the optimal expected remaining cost at s, so that $V^*_\lambda(s) = 1 - \lambda J^*(s)$ when success is attainable with certainty. Then*

$$\delta_\lambda(s, U, c) = \lambda\big(\kappa_c - p_c\, J^*(s)\big), \tag{6}$$

so c is maximal exactly when $\kappa_c \le p_c\, J^*(s)$. For a paid computation the condition is a genuine restriction, and being the cheapest admissible computation does not imply it.

Proof. The first sentence of *(i)* is the second claim of Proposition A.1 restricted to the computations that tie on attainable success. For Equation (6), the proviso makes the failure successor value-equivalent to s, so $Q^*_\lambda(s, U, c) = -\lambda\kappa_c + p_c + (1 - p_c)V^*_\lambda(s)$; substituting $V^*_\lambda(s) = 1 - \lambda J^*(s)$ and subtracting from $V^*_\lambda(s)$ gives the identity. □

Within this abstraction the two initial routing rules are statements about one ordering rather than heuristics attached to a particular λ. A working budget θ is an operational constraint: once it is exhausted, continuing on the same goal is simply outside $\mathcal{C}(s, U)$, and the controller returns to the router, which may re-decide. Whether this is also value-preserving is a comparison of whole continuations: writing h_{ret} for the routing overhead, returning dominates continuing exactly when $p_{\mathrm{ret}} - \lambda\big(h_{\mathrm{ret}} + \mathbb{E}[C_{\mathrm{ret}}]\big) \ge p_{\mathrm{cont}} - \lambda\,\mathbb{E}[C_{\mathrm{cont}}]$. Retaining the partial proof and the router's analysis, so that work is resumed rather than redone, keeps p_{ret} and $\mathbb{E}[C_{\mathrm{ret}}]$ close to their continuing counterparts, but it does not by itself establish the inequality: if one further refinement would have closed the goal for 1, a return costing 0.2 that resumes the same refinement totals 1.2. We therefore treat the bounded contract as a restriction of the action space, justified operationally by retention, and do not claim it is free.

A.2 Regret decomposition: proofs of Theorem 1 and Corollary 2

This subsection states and proves the result summarized at the end of Section 3.1: adaptive routing is beneficial when the decision regret it avoids exceeds its own inference overhead. Both statements are written in terms of the metalevel regret δ_λ of Equation (4).

Theorem 1 (regret decomposition). Let router R reach an absorbing state after finitely many computations in expectation, with value V^R_λ, finite expected inference cost H_R counted in C, and the cap encoded as an explicit stop. Then

$$V^*_\lambda(s_0) - V^R_\lambda(s_0) = \mathbb{E}_R\Big[\textstyle\sum_{t<\tau} \delta_\lambda(s_t, U_t, c_t)\Big] + \lambda\,\mathbb{E}_R[H_R], \tag{7}$$

where τ is the number of computations.

Corollary 2 (when routing pays for itself). Router R outperforms a fixed routine R_0 exactly when the regret it avoids exceeds its extra decision-making cost:

$$\mathbb{E}_{R_0}\Big[\textstyle\sum_t \delta_\lambda\Big] - \mathbb{E}_R\Big[\textstyle\sum_t \delta_\lambda\Big] \;>\; \lambda\big(\mathbb{E}_R[H_R] - \mathbb{E}_{R_0}[H_{R_0}]\big). \tag{8}$$

Cost efficiency is therefore the allocation of purchased computations with low cumulative regret.

Proof of Theorem 1. Fix a router R with $\mathbb{E}_R[\tau] < \infty$, write $\delta_t = \delta_\lambda(s_t, U_t, c_t)$, and let $\mathcal{H}_t$ be the history up to the choice of c_t. By Equation (3) and (A1), on $\{t < \tau\}$,

$$\mathbb{E}_R\big[Q^*_\lambda(s_t, U_t, c_t) \,\big|\, \mathcal{H}_t\big] = \mathbb{E}_R\big[-\lambda\kappa_t + V^*_\lambda(s_{t+1}) \,\big|\, \mathcal{H}_t\big],$$

so that $\mathbb{E}_R[\delta_t \mid \mathcal{H}_t] = \mathbb{E}_R[V^*_\lambda(s_t) - V^*_\lambda(s_{t+1}) + \lambda\kappa_t \mid \mathcal{H}_t]$. Summing over $t < \tau$ and taking expectations telescopes to

$$\mathbb{E}_R\Big[\sum_{t<\tau} \delta_t\Big] = V^*_\lambda(s_0) - \mathbb{E}_R\big[V^*_\lambda(s_\tau)\big] + \lambda\,\mathbb{E}_R\Big[\sum_{t<\tau} \kappa_t\Big],$$

where exchanging sum and expectation is justified because every summand is bounded ($0 \le V^*_\lambda \le 1$, and $\kappa \le \bar{c}$ by (A2)) and $\mathbb{E}_R[\tau] < \infty$. The final state is solved, with value 1, or stopped, with value 0, so $\mathbb{E}_R[V^*_\lambda(s_\tau)] = \Pr_R[\text{solve}]$. Substituting $\Pr_R[\text{solve}] = V^R_\lambda(s_0) + \lambda\,\mathbb{E}_R[\sum_{t<\tau}\kappa_t] + \lambda\,\mathbb{E}_R[H_R]$ from Equation (5) gives Equation (7). □

Proof of Corollary 2. Apply Equation (7) to R and to R_0 from the same s_0. The common term $V^*_\lambda(s_0)$ cancels, so $V^R_\lambda(s_0) > V^{R_0}_\lambda(s_0)$ if and only if Equation (8) holds. □

Theorem 1 is the performance-difference identity of reinforcement learning (Kakade & Langford, 2002), specialized to purchased computations with an explicit charge for deliberation. What it fixes is the target of the design: a router is worthwhile precisely to the extent that its choices lose less continuation value than a fixed routine's, net of its own inference cost.

Truncated runs. The telescoping argument assumes the trajectory reaches the modeled absorbing state and that τ counts the transition into it. A run stopped by an external time or expenditure cap ends instead at a state s_T that may still carry positive continuation value, and for the truncated value $V^R_{\text{trunc}}(s_0) = \mathbb{E}[Z_T] - \lambda\,\mathbb{E}[\sum_{t<T}\kappa_t + H_R]$, with Z_T the solve indicator, the same calculation gives

$$V^*_\lambda(s_0) - V^R_{\text{trunc}}(s_0) = \mathbb{E}_R\Big[\textstyle\sum_{t<T}\delta_t\Big] + \lambda\,\mathbb{E}_R[H_R] + \mathbb{E}_R\big[V^*_\lambda(s_T) - Z_T\big]. \tag{9}$$

The last term is the value abandoned at the cap and cannot be dropped: truncating an unsolved state of value 0.9 before any computation gives a gap of 0.9 with no recorded regret. Equation (7) is therefore the identity for runs that end in the modeled stop, which we obtain by appending an explicit stop computation at the cap and charging it the abandoned value as its regret. Both statements additionally require $\mathbb{E}_R[H_R] < \infty$, which $\mathbb{E}_R[\tau] < \infty$ and (A2) do not imply: the outer horizon bounds completed epochs, so a router that keeps returning malformed decisions could deliberate without end. We therefore assume a finite cap on router retries within an epoch, with a defined fallback when it is reached.

A.3 Faithful two-level graph states preserve reachability

This subsection concerns primitive Lean steps which are deterministic. For a sequence $w = (a_1, \ldots, a_k)$ of primitive steps, let $F^w(s)$ be the state reached from s and $C(w)$ its cost; τ is reserved for the number of computations in Section A.1. Following the verifier-defined reachability view (Sonoda et al., 2026), define

$$\text{Reach}(s) = \mathbf{1}\{\exists w :\ F^w(s) \in \mathcal{G}\}, \qquad C^\star(s) = \inf_{w:\,F^w(s)\in\mathcal{G}} C(w), \tag{10}$$

where $\mathcal{G}$ is the solved set and $C^\star(s) = \infty$ when no successful sequence exists.

Let $\mathcal{R}$ be the lower-level Lean search space with verifier transition $F_\mathcal{R}$ and solved set $\mathcal{G}_\mathcal{R}$. Let $e : \mathcal{R} \to \mathcal{S}$ encode a lower-level state in CoCo-Prover's two-level state space $\mathcal{S}$, whose verified primitive transition is $F_\mathcal{S}$ and whose solved set is $\mathcal{G}_\mathcal{S}$.

Proposition A.3 (Faithful reachability)**.** *Suppose that, for every reachable lower-level state r and primitive step a, the encoding commutes with the verifier transition:*

$$e(F_\mathcal{R}(r, a)) = F_\mathcal{S}(e(r), a). \tag{11}$$

Suppose further that solved states are preserved in both directions, that the two spaces admit the same primitive steps at corresponding states, and that the encoding does not alter step costs. Then

$$\text{Reach}^\mathcal{R}(r) = \text{Reach}^\mathcal{S}(e(r)), \qquad C^\star_\mathcal{R}(r) = C^\star_\mathcal{S}(e(r)). \tag{12}$$

Proof. If a lower-level sequence reaches a solved state, induction over the transition-preservation identity in Equation (11) maps every prefix to the two-level representation, and solved-set preservation maps its endpoint to $\mathcal{G}_\mathcal{S}$. Conversely, the same argument lifts a successful sequence in $\mathcal{S}$ through the faithful encoding. Because corresponding sequences have equal cost, taking the infimum over successful sequences gives the cost equality in Equation (12). This is the faithful-abstraction argument of the verifier-defined reachability view (Sonoda et al., 2026) specialized to the two-level graph. □

Every certificate reachable by primitive Lean steps is reachable in the representation of the two-level proof graphs at the same primitive cost.

A.4 Graph information raises the attainable value

Let $I_{\mathrm{loc}}(s, G)$ contain the goals of the supplied group and their local attempt history. A target-local router has the form $R_{\mathrm{loc}}(s, G) = f(I_{\mathrm{loc}}(s, G))$, whereas a graph-informed router may use the full state s, including other goals, dependency edges, artifact status, and realized spend. Let Π_{loc} and Π_{graph} denote the corresponding policy classes, and $V^*_{\lambda,\Pi}(s) = \sup_{\pi\in\Pi} V^\pi_\lambda(s)$.

Proposition A.4 (Information-class dominance)**.** *$V^*_{\lambda,\Pi_{\mathrm{graph}}}(s) \geq V^*_{\lambda,\Pi_{\mathrm{loc}}}(s)$ for every state s and every $\lambda \geq 0$.*

Proof. A graph-informed router can ignore its additional inputs and process them at no extra inference cost, so $\Pi_{\mathrm{loc}} \subseteq \Pi_{\mathrm{graph}}$, and a supremum over a superset is no smaller. If reading graph evidence were itself more expensive, inclusion of decision rules would not suffice and the comparison would have to charge that difference. □

The gap between the classes shows up when a single policy must serve instances that look alike locally: the value of a helper lemma depends on how many consumers it serves. The next result is a minimax statement about one fixed local policy—on each instance separately, a local policy hardcoded for that instance can be optimal.

Proposition A.5 (Strict separation)**.** *Consider two instances. In instance A, $K \geq 2$ consumers $g_1, \ldots, g_K$ each need a fact that can be supplied by one helper lemma h; in instance B, only g_1 exists. The symbolic rule supplies the group $\{g_1\}$ at* every *decision until the router commits—either closing g_1 directly or proposing the helper—and every computation before that commitment returns the same output in both instances. So the whole observation history I_{loc} admits up to the commitment, and the actions available along it, are identical in the two instances; what differs is the root set, which only a graph-informed router reads. Supplying $\{g_1\}$ only at the first decision would not suffice: a rule that moved to g_2 after a free failed attempt would hand a local router the second consumer, and reading it would be enough to choose correctly on both instances. Every computation is deterministic: the Proof Writer closes a consumer directly at cost a, the Lemma Planner proposes and proves h at cost b, each consumer then closes from h at residual cost $d < a$, and every other computation makes no progress and returns the same output on both instances. Assume $a < b + d$ and $b + Kd < Ka$, so the helper is a losing trade for one consumer but a winning trade for K, and $\lambda Ka < 1$, so solving is always worth its cost. Then one router in Π_{graph} is optimal on both instances, while every* single *router $R \in \Pi_{\mathrm{loc}}$ satisfies*

$$\max_{I\in\{A,B\}} \Big(V^*_{\lambda,\Pi_{\mathrm{graph}}} - V^R_\lambda\Big)(I) \;\geq\; \lambda\,\frac{\Delta_B\,\Gamma_A}{\Delta_B + \Gamma_A}, \tag{13}$$

where $\Delta_B = b + d - a$ and $\Gamma_A = \min\big(a - d,\ Ka - b - Kd\big)$ are positive by assumption.

Proof. A graph-informed router reads the number of consumers from $\mathcal{D}$ and chooses the helper in A (cost $b + Kd$) and the direct proof in B (cost a), both optimal under the assumptions. By hypothesis the group supplied and every output observed agree across the instances until the commitment, so a local router reaches that decision on g_1 with the same observation history in both, and chooses the Lemma Planner there with the same probability p. In B, that choice costs at least $b + d$, an excess of Δ_B over the optimum. In A, choosing the Proof Writer on g_1 costs a for g_1 and at least $\min\big(b + (K-1)d,\ (K-1)a\big)$ for the rest, an excess of at least Γ_A over $b + Kd$. Because $\lambda Ka < 1$, stopping loses at least as much as either excess, and among solving runs value differences equal λ times cost differences, so the local router's worst-case value loss is at least $\lambda \max\big(p\Delta_B, (1-p)\Gamma_A\big)$, which is minimized at $p = \Gamma_A/(\Delta_B + \Gamma_A)$ and gives Equation (13). □

For example, with $K = 2$, $a = 5$, $b = 6$, and $d = 0$, the helper costs 6 against 5 from either consumer's view, yet 6 against 10 globally, and every local router loses at least $4\lambda/5$ on one of the two instances. The separation is therefore uniform over the two instances, and it needs $\lambda > 0$: at $\lambda = 0$ both classes attain success probability one.

A.5 SAFE FILTERING

CoCo-Prover shrinks $\mathcal{C}(s, U)$ with two hard filters. The first is the *no identical deterministic repetition* clause of GUARDS (Algorithm 1), which the Deterministic Automation branch supports by caching a no-progress result against reruns on an unchanged state. The second is the subtraction of ROOTDISC$(\mathcal{D}_t)$ from the frontier, together with the Lemma Planner's requirement that a proposed helper name root-relevant consumers (Section 3.2), so that work with no path to a root is never scheduled. This subsection asks when discarding those options is free, that is, when it leaves $V^*_\lambda(s)$ unchanged. Both rely on the same proviso, which we state as *erasability*: the filtered computation can be deleted from any trajectory that contains it, leaving the subsequent admissible actions, their output laws and prices, the scheduling, and the terminal behavior unchanged, so in particular $V^*_\lambda(s') = V^*_\lambda(s)$ for its successor s'. Equality of the optimal value alone is not enough. A free computation with $\delta_\lambda = 0$ can be the only route to the optimal continuation, which is a free step that sets a flag on which a cheaper proof action depends ties with the optimum, yet deleting it strictly lowers the value. Thus, the deletion argument needs the stronger, behavioral condition.

Lemma A.6 (Deterministic no-progress elimination)**.** *Suppose a computation c has a deterministic output and an earlier invocation from the same normalized proof state, environment, and configuration left that state unchanged. Then, for $\lambda > 0$, $\delta_\lambda(s, U, c) \geq \lambda\kappa_{\min} > 0$, if c is paid, so removing it from $\mathcal{C}(s, U)$ does not change $V^*_\lambda(s)$. If c is free, $\delta_\lambda(s, U, c) = 0$: it is a maximizer, and removing it preserves the value because erasability makes it a self-loop on everything the continuation can observe. It is removed to keep policies proper.*

Proof. By determinism, the repeated invocation again leaves the proof state unchanged, so under the proviso $Q^*_\lambda(s, U, c) = -\lambda\kappa + V^*_\lambda(s)$. For a paid c, (A2) gives $\kappa \geq \kappa_{\min}$ and hence $Q^*_\lambda(s, U, c) \leq V^*_\lambda(s) - \lambda\kappa_{\min}$; for a free c the same identity gives $Q^*_\lambda(s, U, c) = V^*_\lambda(s)$. Thus a paid c is never maximal, and the maximum in Equation (3) is attained without it. A free c is maximal; erasability is what lets the optimal continuation be executed in its absence, so dropping it leaves $V^*_\lambda(s)$ unchanged. □

Lemma A.7 (Root-disconnected work elimination)**.** *A proposed declaration with no dependency path to an active consumer in any root's proof region does not occur in the dependency closure returned from the* current *graph. Assume in addition that such work can be postponed until a consumer is attached, at no greater cost and with the same success distribution. Then, under the proviso, a paid computation that works only on such a declaration has $\delta_\lambda \geq \lambda\kappa_{\min}$, so deferring it does not change $V^*_\lambda(s)$.*

Proof. Every generated lemma used by a returned root proof has, by definition, a path through its consumers to that root, so a lemma with no such path is absent from every closure and cannot affect whether a state is solved. A later plan may attach a consumer, which is why the postponement assumption is needed where it makes deferral. Under the proviso the computation then leaves V^*_λ unchanged while costing at least $\kappa_{\min}$, and the argument of Proposition A.6 applies. □

Consumer-first admission is therefore a search restriction justified by the postponement assumption. The remaining semantic choices are not certified as safe and are left to the router with bounded trials.

A.6 SHARED-LEMMA COST SEPARATION

The previous two subsections compare routers on the same shared graph. The next result compares representations: a shared lemma-dependency graph against isolated proof hypergraphs.

Proposition A.8 (Shared-lemma cost separation)**.** *Fix an abstract instance in which K consumers each need a fact supplied only by an absent helper h, producing h costs c_h, and consumer i then closes at residual cost d_i. Compare two* stipulated *strategies: an isolated-graph prover that cannot transfer generated artifacts, and a graph prover that proves h once. Their costs are*

$$C_{\text{local}} = Kc_h + \sum_{i=1}^{K} d_i, \qquad C_{\text{graph}} = c_h + \sum_{i=1}^{K} d_i. \tag{14}$$

When $c_h > 0$ and $d_i = 0$ for every consumer, $C_{\text{local}}/C_{\text{graph}} = K$; with $c_h = 0$ both totals equal $\sum_i d_i$, so the ratio is 1, or undefined when the residuals vanish too.

Proof. The accounting is immediate from the stipulated costs. An isolated-graph prover that cannot transfer generated artifacts pays c_h inside each graph. The graph-informed prover proves h once, records one node, and connects it to all consumers. Deterministic use of the checked declaration incurs no additional model-generation cost. Substitution in Equation (14) gives the stated ratio when the residual costs vanish. □

Consequently, for any external cap B satisfying

$$c_h + \sum_i d_i \le B < K c_h + \sum_i d_i, \tag{15}$$

the stipulated graph strategy completes within the cap while the stipulated isolated strategy does not. This compares the two strategies, not every strategy each representation admits. Failed helper attempts and rollback appear as the realized term r_h in Equation (18).

Lemma planning as nonlocal value of computation. Measured against V^*_λ, no purchase can show a gain: $Q^*_\lambda(s, U, c_{\mathrm{LP}}) - V^*_\lambda(s) = -\delta_\lambda(s, U, c_{\mathrm{LP}}) \le 0$, since the optimum already includes the option of buying the helper. The amortization argument is therefore stated against a *direct-only* reference policy π_{dir}, which never proposes a helper and proves every consumer separately. For two strategies the value difference is $V_{\mathrm{share}} - V_{\mathrm{dir}} = (p_{\mathrm{share}} - p_{\mathrm{dir}}) + \lambda\big(\mathbb{E}[C_{\mathrm{dir}}] - \mathbb{E}[C_{\mathrm{share}}]\big)$, so a cost saving settles the comparison only when the two succeed equally often. Assume therefore that sharing does not change the success probability, write ΔJ_i for the expected cost that h saves consumer i relative to π_{dir}, and treat the consumers as otherwise independent. Then proposing and proving h improves on π_{dir} by roughly

$$\lambda\Big(\sum_{i \in \mathrm{Cons}(h)} \Delta J_i - c_h - \mathbb{E}[r_h]\Big), \tag{16}$$

where r_h is the realized cost of failed helper attempts and rollback. The equal-success assumption is a real restriction rather than a formality: a helper that turns out false, too hard, or abandoned lowers p_{share}, and r_h prices those attempts without pricing their effect on whether the task is solved at all. A strategy that spends 1 and succeeds half the time beats one that spends 10 and always succeeds on cost, yet their values are $\frac{1}{2} - \lambda$ and $1 - 10\lambda$, so the cheap strategy loses on value exactly when $\lambda < 1/18$ and wins above it. For instance, in the $K = 2$, $a = 5$, $b = 6$, $d = 0$ instance of Section A.4 with $\lambda = 0.1$, this reads $0.1(10 - 6) = 0.4$ against the direct-only policy, while the advantage over V^*_λ is zero. One granularity caveat: the implemented Lemma Planner only proposes and elaborates h, returning control to the router (Section 3.2), so its successor value still carries the cost of proving h. Equation (16) prices the macro-action that includes that proof, and c_h must not be charged twice. A lemma can thus be a losing purchase from each consumer's view and a winning one globally, as in the $K = 2$ example of Section A.4. This is why the router's evidence packet includes consumers and conditional descendants.

A.7 Kernel-relative soundness

Proposition A.9 (Kernel-relative soundness)**.** *Assume that (i) the declared Lean environment Γ_0 is accepted, (ii)* CoCo-Prover *promotes a generated lemma only after Lean accepts its proof, (iii) every generated dependency of root $\ell \in L_\star$ is stable, and (iv) no prohibited placeholder or unapproved axiom occurs. If* CoCo-Prover *returns a stable proof of ℓ, then $\Gamma_0 \vdash \ell$.*

Proof. Topologically order the generated dependency DAG. Induction over this order shows that each stable generated declaration has a kernel-accepted proof using only Γ_0 and earlier stable declarations. Applying the argument to the accepted proof of ℓ yields the result. □

B Operational Details

B.1 Two-level proof graphs

This subsection gives the full definition of the two-level proof graphs summarized in Section 3.1 and drawn in Figure 2.

Inter-lemma dependency graph. Across declarations, the lemma-dependency graph $\mathcal{D} = (L, E_{\mathcal{D}})$ is a directed acyclic graph that carries an edge $u \to v$ whenever the current proof or plan for v uses helper lemma u. The targets $L_\star \subseteq L$ are its roots, so helper-lemma work is useful only through a dependency path to some root.

Intra-lemma AND/OR hypergraph. Within every target theorem $\ell_{\star,i} \in L_\star$ or generated helper lemma ℓ, a lazily constructed AND/OR proof hypergraph $\mathcal{H}_\ell = (A_\ell, O_\ell, E_\ell)$ is a tree rooted at the lemma's initial goal. *AND nodes* A_ℓ are normalized Lean goal states, all of whose subgoals must be proved; *OR nodes* O_ℓ are candidate proof steps (a tactic or tactic sequence), siblings being alternatives for the same goal. Executing a step o attaches the verified hyperedge $e = (o, \{g_1, \ldots, g_k\}) \in E_\ell$ recording exactly the subgoals Lean returns, with $k = 0$ when the step closes its goal. A goal keeps at most one valid alternative at a time; an alternative discarded by the Recovery Reviewer (Section 3.2) is retained as failure evidence while a new one is added.

Artifact status. Each artifact is labeled *open*, *provisional*, *conditionally solved*, *stable*, or *rejected*, and a goal with no admissible action is *blocked*. Only a Lean-accepted declaration whose generated dependencies are stable can itself become stable, and conditional plans guide search but never certify a root.

Nonlocal effects. This structure makes the two nonlocal effects of our setting explicit: one stable helper lemma serving several consumers, and a failed provisional helper lemma invalidating exactly its conditional descendants, so a failure rolls back to the nearest safe point instead of discarding whole attempts.

B.2 End-to-end algorithm

Algorithm 1 states the complete control loop of CoCo-Prover (Section 3.2): each epoch builds the frontier symbolically, lets the router group it and pick one group, purchases one routed action, and writes its verified result back, with initialization, evidence construction, decision validation, the two write-back paths, admissibility failure, and termination made explicit.

The operators are as follows. PROPAGATE closes artifact status over $\mathcal{D}_t$: a stabilized helper lemma releases its consumers, a rejected or changed provisional artifact marks its conditional descendants invalidated. OPEN($\mathfrak{G}_t$) is the set of open AND nodes; PREREQ(g) is the set of active prerequisites in $\mathcal{D}_t$; ELABORATED(σ_t) is the set of declarations with an elaborated statement, stable or provisional, against which a proof can be constructed, while STABLE(σ_t) is the smaller set that may be promoted; ROOTDISC($\mathcal{D}_t$) is the set of lemmas with no dependency path to a root, which are discarded; SCHED is the structural stage of Equation (1): it maintains the frontier and hands *all* of it to the agentic router, $U_t = \mathcal{F}_t$. Grouping is the router's: CLUSTER partitions U_t into Γ_t by embedding k-means (Section B.4), the static-analysis check sends definition-nested groups straight to Deterministic Automation, and the router then chooses which group $G_t \in \Gamma_t$ to buy a computation for; T is the deterministic update of Equation (2). DEFNESTED is the static-analysis check: a goal whose obstruction is only unfolded definitions is dispatched to Deterministic Automation without a routing call, but only when that action itself passes the guards—after a no-progress automation attempt on an unchanged goal the guards exclude it, and the goal is routed instead— and AMEND is the router's revision of its own rules ρ_t after observing an outcome. GUARDS evaluates the mechanically checkable admissibility conditions (contract membership, acyclicity, available identifiers, no identical deterministic repetition); $r_{\max}$ bounds the router calls in one epoch, so deliberation is finite even when decisions are repeatedly malformed. DEFAULT is the deterministic fallback used when that bound is reached: the admissible triple with the lowest contract cost bound (*i.e.*, Deterministic Automation whenever it is admissible) with groups ordered by d_σ then statement hash, agents by a fixed order, and an empty analysis j. Everything the chosen specialist sees is part of the decision: besides j_t, the excerpt of the router's conversation that the invocation carries is a field of θ_t, and the update records it in $\mathcal{M}_{t+1}$ afterwards. Deliberation charges go to H, so T stays a function of its four arguments. One pass of the loop is a *decision epoch*: it recomputes the frontier, reclusters it, and purchases exactly one computation on one group. A *controller iteration* is the reporting unit the case studies count (Section D): a boundary in the log that groups consecutive epochs, imposing no restriction on which actions they may take. In $\mathrm{ev}(s_t, U_t)$, $N_{\mathfrak{G}_t}(U_t)$ is the graph neighborhood of U_t (prerequisites, consumers, blocking relations, conditional descendants), $\mathcal{M}_t\restriction_{U_t}$ is the failure memory restricted to U_t, and $X_t(\cdot)$ is the at-risk spend of Equation (19). O abbreviates the four specialist

Algorithm 1 CoCo-Prover: metalevel decision-making under cost on the two-level proof graph

Require: environment Γ_0; targets $L_\star = \{\ell_1, \dots, \ell_n\}$; specialist agents $O = \{\mathrm{DA}, \mathrm{PW}, \mathrm{LP}, \mathrm{RR}\}$ with contracts $\{\Theta_o\}_{o \in O}$; horizon $t_{\max}$ (also stopped by the run's wall-clock and expenditure caps)

$\mathcal{D}_0 \leftarrow (L_\star, \emptyset)$; $\mathcal{H}_{\ell,0} \leftarrow \{\mathrm{root}(\ell)\}$ and $\sigma_0(\ell) \leftarrow$ *open* for each $\ell \in L_\star$; $\mathcal{M}_0, \mathcal{L}_0 \leftarrow \emptyset$
$s_0 \leftarrow (\mathfrak{G}_0, \mathcal{F}_0, \sigma_0, \mathcal{M}_0, \mathcal{L}_0)$; $\rho_0 \leftarrow$ initial routing rules ▷ ρ_t is the router's own variable, not part of s_t
for $t = 0, 1, 2, \dots$ **while** $s_t \notin \mathcal{G}$ **and** $t < t_{\max}$ **do**
  $\sigma_t \leftarrow \textsc{Propagate}(\mathfrak{G}_t, \sigma_t)$ ▷ status closure over $\mathcal{D}_t$
  $\mathcal{F}_t \leftarrow \{g \in \textsc{Open}(\mathfrak{G}_t) : \textsc{Prereq}(g) \subseteq \textsc{Elaborated}(\sigma_t)\} \setminus \textsc{RootDisc}(\mathcal{D}_t)$ ▷ selection: incremental Kahn pass
  if $\mathcal{F}_t = \emptyset$ **then return** $\textsc{Report}(\sigma_t, \mathcal{M}_t, \mathcal{L}_t)$
  end if
  $U_t \leftarrow \textsc{Sched}(s_t) = \mathcal{F}_t$ ▷ the whole dependency-ready frontier, handed over in full
  $\Gamma_t \leftarrow \textsc{Cluster}(U_t)$; $\mathcal{C}(s_t, U_t) \leftarrow \{(G, o, \theta) : G \in \Gamma_t,\ o \in O,\ \theta \in \Theta_o,\ \textsc{Guards}(G, o, \theta, s_t) = \top\}$ ▷ router-side grouping, no LLM call
  if $\mathcal{C}(s_t, U_t) = \emptyset$ **then** $\sigma_t(U_t) \leftarrow$ *blocked*; **continue**
  end if
  if some $G \in \Gamma_t$ has $\textsc{DefNested}(G) \ \wedge\ (G, \mathrm{DA}, \theta_{\mathrm{DA}}) \in \mathcal{C}(s_t, U_t)$ **then** ▷ static analysis: no model call
    $(G_t, o_t, \theta_t, j_t) \leftarrow (G, \mathrm{DA}, \theta_{\mathrm{DA}}, \varnothing)$ ▷ definition-nested goals go straight to automation
  else
    $\mathrm{ev}(s_t, U_t) \leftarrow \big\langle U_t,\ N_{\mathfrak{G}_t}(U_t),\ \mathcal{M}_t\restriction_{U_t},\ X_t(\cdot),\ \mathcal{L}_t,\ \{\Theta_o\}_{o \in O} \big\rangle$ ▷ environmental evidence; the router adds its own rules ρ_t
    for $r = 1, \dots, r_{\max}$ **do** ▷ evidence packet; ρ_t: routing rules ▷ action: metalevel decision under cost
      $(G_t, o_t, \theta_t, j_t) \leftarrow \textsc{Router}\big(\mathrm{ev}(s_t, U_t), \Gamma_t, \rho_t\big)$; charge the call to H ▷ pick a group, an agent, a configuration; j_t: preliminary analysis
      break if $(G_t, o_t, \theta_t) \in \mathcal{C}(s_t, U_t)$ ▷ malformed or guard-violating decisions are rejected
    end for
    if no admissible decision after $r_{\max}$ retries **then** $(G_t, o_t, \theta_t, j_t) \leftarrow \textsc{Default}(\mathcal{C}(s_t, U_t))$ ▷ finite deliberation per epoch
    end if
  end if
  $Y_t \leftarrow \textsc{Exec}_{o_t}(G_t; \theta_t, j_t)$ ▷ environment response on the chosen group; execution cost κ_t, deliberation charged to H
  if $o_t = \mathrm{DA}$ **then** ▷ automation: Lean-side search, no model call
    attach the verified steps of Y_t to $\mathcal{H}_{\ell,t}$; cache a no-progress result against reruns on the unchanged state
  else if $o_t = \mathrm{PW}$ **then** ▷ proof writer: iterative refinement within θ_t
    attach the verified steps of Y_t, keeping a partial proof when the refinement budget is exhausted
  else if $o_t = \mathrm{LP}$ **then** ▷ lemma planner: elaborate, then back to the router
    $\mathcal{D}_{t+1} \leftarrow \mathcal{D}_t \cup \textsc{Elab}_{\circlearrowleft}(Y_t)$;
    new obligations $\to \mathcal{F}$
  else if $o_t = \mathrm{RR}$ **then** ▷ recovery: nearest-safe-point backtracking
    if Y_t refutes some $g \in G_t$ **then** reject that g alone **else** switch path when judged no less provable and cheaper to finish
    end if
    if Y_t retracts artifact u **then** reopen $\mathrm{CondDesc}_t(u)$; retain unaffected verified subgraphs
    end if
  end if
  $\rho_{t+1} \leftarrow \textsc{Amend}(\rho_t, U_t, c_t, Y_t)$ if the epoch was routed, else ρ_t ▷ rules evolve on the outcome just observed
  $s_{t+1} \leftarrow T(s_t, U_t, c_t, Y_t)$ ▷ attach verified (hyper)edges; update $\sigma, \mathcal{M}$; append κ_t to $\mathcal{L}$
  for all v proposed for promotion **do**
    $\sigma_{t+1}(v) \leftarrow$ *stable* **iff** $\textsc{Kernel}(v) = \top\ \wedge\ \textsc{Deps}(v) \subseteq \textsc{Stable}(\sigma_{t+1})$
  end for
end for
return $\{\ell \in L_\star : \sigma(\ell) =$ *stable*$\}$ with dependency closures; $\textsc{Report}(\sigma, \mathcal{M}, \mathcal{L})$ for the rest

agents; $\textsc{Elab}_{\circlearrowleft}(Y_t)$ keeps only proposed lemmas that elaborate, preserve acyclicity, and survive random testing and the structural check; $\textsc{Kernel}(v)$ is Lean kernel acceptance of v's proof and $\textsc{Deps}(v)$ are its generated dependent lemmas. $\textsc{Report}$ emits the structured failure and diagnostics.

B.3 DEPENDENCY-AWARE SCHEDULING

The scheduler discharges the selection half of the metalevel symbolically: it maintains the dependency-ready frontier and hands it to the router *in full*, so no model call is spent on selection. Algorithm 2 states the maintenance precisely. The frontier is kept by an *incremental Kahn pass* (Kahn, 1962) over the dynamic dependency graph $\mathcal{D}_t$: for every lemma/theorem ℓ, the scheduler keeps the *unelaborated in-degree* $d(\ell)$ and the frontier $\mathcal{F}_t$ consists exactly of the open AND nodes of declarations with $d(\ell) = 0$, minus root-disconnected work. A second counter, the *unstable in-degree* $d_\sigma(\ell)$, counts every prerequisite that is not yet stable, elaborated or not: it does not gate the frontier, but $d_\sigma(\ell) = 0$ marks the declarations whose proofs can be promoted, and the scheduler orders the frontier by it, so unconditional work is preferred to conditional work. Kahn's algorithm processes zero-in-degree nodes of a static graph but here the graph changes as lemmas are added, elaborated, stabilized, and retracted, so the counters are updated per event at cost proportional to the affected out-degree. Elaborating u decrements $d(\ell)$ for each consumer ℓ and releases those reaching zero; stabilizing u decrements $d_\sigma(\ell)$ and enables promotion; retracting u reopens exactly its conditional descendants and restores their counters rather than re-sorting the graph. Two properties of the frontier follow from its definition and carry the scheduler's cost rationale: work with no dependency path to a root never enters it, and no goal is opened beneath a prerequisite that has not even been elaborated. Work beneath a *provisional* prerequisite is admitted and its exposure is what Equation (19) measures and the Recovery Reviewer unwinds.

Algorithm 2 Dependency-aware scheduling: incremental Kahn maintenance of the frontier

Maintained state: $d(\ell) = \big|\{u : (u \to \ell) \in E_{\mathcal{D},t},\ u \text{ not elaborated}\}\big|$ and $d_\sigma(\ell) = \big|\{u : (u \to \ell) \in E_{\mathcal{D},t},\ \sigma(u) \neq \textit{stable}\}\big|$ for every $\ell \in L_t$; $\mathcal{F} = \{g \in \textsc{Open}(\mathfrak{G}) : d(\mathrm{decl}(g)) = 0\} \setminus \textsc{RootDisc}(\mathcal{D})$

procedure ONEDGEADDED($u \to \ell$)
  if u not elaborated **then** $d(\ell) \leftarrow d(\ell) + 1$; remove ℓ's goals from $\mathcal{F}$
  end if; **if** $\sigma(u) \neq$ *stable* **then** $d_\sigma(\ell) \leftarrow d_\sigma(\ell) + 1$
end procedure
procedure ONELABORATED(u)
  for all $(u \to \ell) \in E_{\mathcal{D},t}$ **do** $d(\ell) \leftarrow d(\ell) - 1$; **if** $d(\ell) = 0$ **then** add ℓ's open goals to $\mathcal{F}$
  end for
end procedure
procedure ONSTABILIZED(u)
  for all $(u \to \ell) \in E_{\mathcal{D},t}$ **do** $d_\sigma(\ell) \leftarrow d_\sigma(\ell) - 1$ ▷ $d_\sigma(\ell) = 0$: ℓ may be promoted
  end for
end procedure
procedure ONRETRACTED(u)
  reopen $\mathrm{CondDesc}(u)$, retaining unaffected verified subgraphs; restore $d(\cdot)$ and $d_\sigma(\cdot)$ on the reopened region; update $\mathcal{F}$
end procedure
procedure SCHED(s)
  return $\mathcal{F}$ in full, ordered by d_σ ▷ grouping and the static check are the router's (Section 3.2)
end procedure

Groups that reach the Lemma Planner are additionally validated for compatibility: a valid group requires compatible environments and similarity in normalized targets, hypotheses, source definitions, or recursion structure. This restriction prevents surface similarity from becoming an unchecked dependency.

B.4 GOAL CLUSTERING BY SEMANTIC EMBEDDING

Batching reduces the cost of router and specialist invocations: a single call jointly analyzes several related goals, and proposals for shared helpers draw evidence from multiple potential consumers. These savings, however, depend on grouping goals that are genuinely related. The router forms such groups, on the frontier the scheduler hands it, by applying k-means clustering to semantic

embeddings of the normalized goal statements. DreamProver (Zhang et al., 2026b) uses the similar clustering method but they use it to evolve lemmas.

Embedding and similarity. Each open goal $g \in \mathcal{F}_t$ is rendered as its normalized statement using the tactic state and embedded with the Lean-statement embedding model of Lean Finder (Lu et al., 2026), which is trained on formal Lean statements. Writing $\phi(g) \in \mathbb{R}^d$ for the raw embedding, we store the unit-normalized vector $x_g = \phi(g)/\|\phi(g)\|_2$ in an index keyed by the statement hash, so embeddings are computed once per normalized statement and reused as the frontier evolves. Similarity between goals is cosine similarity: $\mathrm{sim}(g, g') = \langle x_g, x_{g'} \rangle$.

k-means on the unit sphere. Given the current frontier embeddings $\{x_g\}_{g \in \mathcal{F}_t}$, spherical k-means seeks unit centroids $\mu_1, \ldots, \mu_k$, with $\|\mu_j\|_2 = 1$, and an assignment into clusters $C_1, \ldots, C_k$ minimizing

$$\sum_{j=1}^{k} \sum_{g \in C_j} \left\| x_g - \mu_j \right\|_2^2, \tag{17}$$

Because both x_g and μ_j lie on the unit sphere and an unconstrained mean centroid has $\|x_g - \mu_j\|_2^2 = 1 + \|\mu_j\|_2^2 - 2\langle x_g, \mu_j \rangle$, we have $\|x_g - \mu_j\|_2^2 = 2 - 2\langle x_g, \mu_j \rangle$, so minimizing Equation (17) is equivalent to maximizing within-cluster cosine similarity, under the similarity the index already uses. We run Lloyd's iteration: initialize centroids by k-means++ seeding from a fixed seed derived from the hash of the frontier, so that the grouping is a deterministic function of the state. We repeat: (i) assign each x_g to its nearest centroid and (ii) update each μ_j to the mean of its cluster, re-normalized to the unit sphere; stop when assignments are unchanged. An empty cluster, or one whose mean vanishes, is reseeded deterministically with the frontier goal furthest from its assigned centroid, ties broken by statement hash, so that the grouping stays a total deterministic function. The cluster count is set from the batch budget, $k = \lceil |\mathcal{F}_t|/b \rceil$ for a fixed target batch size b, and singleton clusters are permitted.

B.5 Sharing and rollback accounting

Shared helper lemmas can reduce duplicate generation, but failed helper lemmas create rollback costs. CoCo-Prover exposes both effects without asking the router to fill unknown quantities. If K consumers would otherwise rediscover helper lemma h, define

$$\begin{aligned} C_{\mathrm{iso}}(h) &= \sum_{i=1}^{K} (c_{h,i} + d_i), \\ C_{\mathrm{share}}(h) &= c_h + \sum_{i=1}^{K} d_i + r_h, \end{aligned} \tag{18}$$

where $c_{h,i}$ is the work spent generating helper lemma h for consumer i, c_h is the realized cost of proving the shared helper lemma once, d_i is the remaining work for consumer i, and r_h is expenditure on failed attempts to prove the helper lemma and on rollback. By Equation (18), sharing saves money exactly when $C_{\mathrm{share}}(h) < C_{\mathrm{iso}}(h)$.

The lemma-dependency graph also makes current exposure measurable. For provisional artifact u, CoCo-Prover reports

$$X_t(u) = \sum_{v \in \mathrm{CondDesc}_t(u)} c_t^{\mathrm{sunk}}(v), \tag{19}$$

where $\mathrm{CondDesc}_t(u)$ is the set of conditional descendants that require revalidation if u changes, and $c_t^{\mathrm{sunk}}(v)$ is the realized cost already charged to descendant v by epoch t. A large value in Equation (19) is evidence for resolving u before further deepening. This quantity enters the router's evidence packet in Section 3.2. When a helper lemma is rejected, CoCo-Prover reopens its conditional descendants while retaining verified subgraphs unaffected by the failed dependency.

B.6 Message passing between the router and the action agents

CoCo-Prover treats the router's context as the head of the conversation that the chosen agent continues, rather than as a separate request. Concretely, a routing decision hands the action agent the

router's message history together with the preliminary analysis j that the router produced while making the decision. The agent appends its own turns to that history instead of opening a fresh one, so the shared prefix is served from the provider's prompt cache at the cache-hit price, and the agent inherits the router's reading of the goals rather than re-deriving it. The ledger records cached and uncached input tokens separately, so the realized effect of this reuse is visible in the reported costs rather than assumed.

B.7 Structural checking of proposed lemmas

A helper lemma proposed by the Lemma Planner enters the dependency graph only after it elaborates and preserves acyclicity, but elaboration only establishes that the statement is well-formed, not that it is true. Committing to a false helper lemma is expensive: consumers are proved against it, and the work beneath it must be discarded when Lean eventually refuses the proof. CoCo-Prover therefore tries to refute a candidate before any proof effort is spent on it, in two escalating steps.

Step 1: random testing. For a candidate whose statement is executable, we sample concrete values for its universally quantified variables, instantiate the statement, and evaluate it. This is cheap, requires no model call, and catches the common failures of a plausible-looking conjecture such as a missing side condition, an off-by-one bound, or a hypothesis that the planner assumed but never stated. A single falsifying instance disproves the lemma outright.

Step 2: LLM-proposed structured instances. Random sampling explores the space uniformly and rarely reaches the corner cases where verification conditions actually fail: a singleton, an index at the boundary, an overflow value, or the state just before and after a loop's last iteration. Constructing those instances takes reasoning about the statement rather than sampling, so when Step 1 finds no counterexample, we ask a model to propose a small set of structured instances, and evaluate them the same way. Step 2 runs only if Step 1 passes, so its cost is paid only for candidates that have already survived the free check.

If either step produces a counterexample, the candidate is rejected. The counterexamples, with the instance that triggered it, are returned to the Lemma Planner, which refines the statement, most often by strengthening a hypothesis or restricting a bound, and proposes a revised lemma. If both steps pass, the lemma is admitted to the dependency graph as *provisional*. Passing these checks is evidence rather than a guarantee: testing can refute a candidate but cannot promote it, and only Lean acceptance makes a declaration stable.

C Additional Experiment Results

C.1 Additional Explanation of Experiment Setup

Experiment setup. Every method runs under the same external limits: each function-level problem is capped at 5 hours of wall-clock time and $30 of expenditure (including NTP4VC since one task has one obligation in its benchmark design), and each repository-level task at 8 hours and $60. All methods access each backbone model through the same official provider: Gemini-3.5-Flash via Google Vertex AI, GPT-5.6-Terra via OpenAI, and DeepSeek-V4-Flash via DeepSeek, so token prices and caching behavior are identical across methods. Cost is measured the same way for every method: we collect the token counts of every model call by type (uncached input, cached input, cache write, and output tokens, with reasoning tokens billed as output) and convert them to US dollars under the provider's official pricing (Table 4). All experiments run on a machine with 16 vCPUs and 256 GB of memory, which hosts Lean checking and all local computation while model inference runs through the provider APIs, with 12 concurrent prover runs. We use Lean Finder (Lu et al., 2026) as the retrieval tool for the methods and deploy it as a service on a machine with 4 vCPUs, 16 GB of memory and one NVIDIA L4 GPU. After each run, the generated Lean proof is checked by the independent Lean proof reviewer checker described below.

Model pricing. Table 4 lists the official prices used to convert token counts into the dollar costs reported throughout the paper. We fetched the GPT and Gemini prices from the providers' official pricing pages on July 19, 2026, and the DeepSeek prices on August 17, 2026. For GPT and Gemini we use the standard-tier prices; DeepSeek prices vary by time of day, and we use the peak-hour rates (off-peak rates are half), so no DeepSeek cost is understated. Every run of every method is

costed under this single price schedule, so cost comparisons do not depend on when a run was executed. The three backbones are shared by CoCo-Prover and all baselines; GPT-5.6-Sol is used only to generate the CLEVER and AlgoVeri implementations before any prover runs, so its cost is not charged to any method.

Table 4: **Official model prices**, in US dollars per million tokens. *Input* applies to uncached input tokens, *Cache hits* to input tokens read from the provider's prompt cache, and *Output* to output tokens, including reasoning tokens. Sources: OpenAI, `https://developers.openai.com/api/docs/pricing`, and Google, `https://ai.google.dev/gemini-api/docs/pricing`, both fetched with standard tier on July 19, 2026; DeepSeek, `https://api-docs.deepseek.com/quick_start/pricing/` with peak-hour rates, fetched on August 17, 2026.

Model	Provider	Input	Cache hits	Cache write	Output
GPT-5.6-Terra	OpenAI	2.50	0.250	3.125	15.00
Gemini-3.5-Flash	Google Vertex AI	1.50	0.150	–	9.00
DeepSeek-V4-Flash	DeepSeek	0.44	0.014	–	1.32

Excluded baselines. We do not compare against four other open-source provers, each for a different reason.

Hilbert (Varambally et al., 2026) was built for MiniF2F and PutnamBench, where a problem is a self-contained theorem: the file header holds only `import Mathlib`, `open` declarations, and `set_option` lines, and everything the theorem mentions is Mathlib vocabulary the model already knows. Program verification benchmarks break this assumption. Their header carries the actual mathematical content—`def implementation` and `def problem_spec`—and the theorem is a single line that refers to these definitions. Hilbert's decomposition does not treat the header as part of the problem, and this also causes the wasted assemblies we observed: when stating a subgoal, the model adds hypotheses about `implementation`, not recognizing it as a concrete definition it could unfold, so the subgoal's signature no longer matches its use site and the sketch fails to assemble.

Goedel-Code-Prover (Li et al., 2026c) relies mainly on its own fine-tuned model; its agentic framework is a simple pipeline of lemma decomposition and iterative refinement whose effectiveness comes from sampling many model answers. Under our protocol—one attempt per problem (pass@1) with a frontier backbone such as Gemini-3.5-Flash—it failed to prove problems within a $5 budget that both CoCo-Prover and the coding agents prove for under $0.50 each. Given this gap, which stems from a framework designed around a fine-tuned model and repeated sampling rather than from the backbone, a matched-budget comparison would not be informative.

WybeCoder (Gloeckle et al., 2026) and P^3 (Li et al., 2026d) target end-to-end verifiable code generation, in which generating the implementation and generating its proof guide each other. In our setting the implementation and specification are fixed inputs (Section 4), so their central mechanism does not apply to pure proof generation.

Benchmark licenses. Table 5 lists the license of each benchmark we evaluate on, as stated in its official repository. We use all five benchmarks only for research evaluation, in accordance with their licenses, and credit each through its original paper; we do not redistribute any benchmark data. The implementations we generate with GPT-5.6-Sol for CLEVER and AlgoVeri are derived from their specifications and remain subject to those benchmarks' licenses. NTP4VC's tooling is MIT-licensed, while each verification condition, including its Lean 4 translation, keeps the license of the C project it is derived from. Each Vero task is a Lean 4 translation of a third-party repository: tasks derived from permissively licensed upstream projects are distributed under Vero's Apache-2.0 license with attribution, while tasks derived from copyleft upstream projects keep their upstream licenses.

Baseline versions. Table 6 pins every baseline to the exact release or commit we ran, so each comparison can be reproduced against the same code. The three coding agents are installed from npm at the listed versions, each driving the backbone of its own provider; Humanize and Ax-Prover-Base are run from the listed commits of their public repositories.

Table 5: **Benchmark licenses.** ‡NTP4VC's tooling is released under MIT; its verification conditions keep the licenses of their upstream C projects, which include GPL-2.0, GPL-3.0, LGPL-2.1, BSD-3-Clause, and MIT. †Vero's pipeline, specifications, and permissively licensed tasks are released under Apache-2.0; tasks derived from copyleft upstream projects keep their upstream licenses: `flocq` and `portion` (LGPL-3.0-or-later) and `huffman` (LGPL-2.1-or-later).

Benchmark	License	Repository
CLEVER (Thakur et al., 2025)	MIT	`github.com/trishullab/clever`
VERINA (Ye et al., 2026b)	Apache-2.0	`github.com/sunblaze-ucb/verina`
AlgoVeri (Zhao et al., 2026a)	Apache-2.0	`github.com/haoyuzhao123/algoveri`
NTP4VC (Xu et al., 2026)	Mixed‡	`github.com/xqyww123/NTP4VC`
Vero (Ye et al., 2026a)	Apache-2.0†	`github.com/sunblaze-ucb/vero`

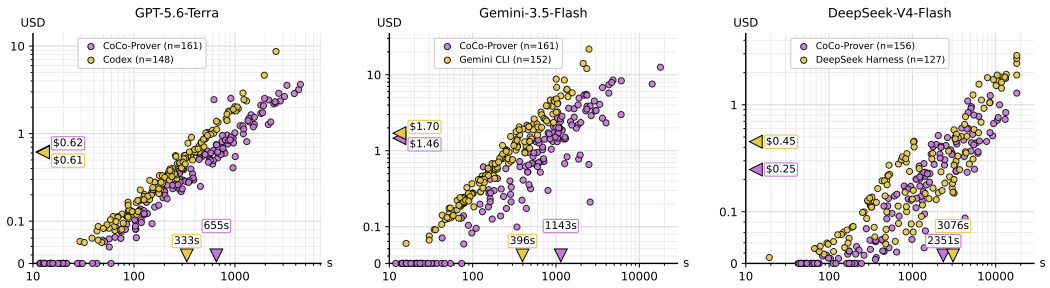
Lean proof reviewer checker. A run counts as solved only if its proof survives an independent review, applied identically to CoCo-Prover and to every baseline. The review is needed because a failing agent can end a run in ways that compile but prove nothing: leaving a placeholder, quietly editing the theorem it was asked to prove, or assuming the goal as a new axiom. After a run finishes, we take the produced Lean file and apply five checks, the first three mechanical, the last two aimed at proofs that pass the mechanical checks yet are not genuine.

1. **No unsolved goals.** We re-elaborate the file with the Lean checker and reject any run whose target theorems still contain an unproved hole, whether written as `sorry` or `admit`, or left open by a placeholder tactic such as `apply?`; the elaborator's own `sorry`-containing and error warnings are treated as failures rather than diagnostics.
2. **Unmodified statements.** The theorem the agent proves must be the theorem it was given. We compare each target declaration against the benchmark original by string match on the statement and by re-elaborating it in the benchmark environment, so a weakened hypothesis, a renamed constant, a changed binder, or a restated conclusion is rejected even when the accompanying proof compiles.
3. **No new axioms.** We ask the Lean checker for the axioms each target theorem's proof actually depends on and require that set to lie within the benchmark's own environment. A proof that introduces an `axiom`, admits a declaration through an axiomatic definition, or otherwise extends the trusted base is rejected.
4. **Automated review.** The three checks above are syntactic, so we additionally run Codex with GPT-5.6-Sol over each accepted proof, with the original statement and the produced file, and ask it to report placeholders, unproved obligations, and proofs that are vacuous or only appear to establish the statement. Any proof it flags is re-examined by hand.
5. **Manual audit.** Finally, we sample passing proofs at random across benchmarks, methods, and backbones and inspect them manually to confirm that they are genuine and that the preceding checks are not missing a failure mode.

Every solve rate and cost reported in this paper counts only runs that pass all five checks.

Table 6: **Baseline versions.** Coding agents are pinned by npm release; Humanize and Ax-Prover-Base by Git commit, abbreviated to seven characters (each entry links to the full commit). Full commit hashes: Humanize `8e21ff211daa731fa826c491e1f12ed284135c0d`; Ax-Prover-Base `06dfadc9ab439755af5efcfe0add95bfef2733c7`.

Baseline	Backbone	Package / GitHub repository	Version
Codex	GPT-5.6-Terra	`@openai/codex`	0.146.0
Gemini CLI	Gemini-3.5-Flash	`@google/gemini-cli`	0.55.1
DeepSeek Harness	DeepSeek-V4-Flash	`@deepseek-ai/dsh`	0.1.2-alpha.2
Humanize	all three	`humanfia/putnambench`	`8e21ff2`
Ax-Prover-Base	all three	`Axiomatic-AI/ax-prover-base`	`06dfadc`



(a) GPT-5.6-Terra: CoCo-Prover vs. Codex.

(b) Gemini-3.5-Flash: CoCo-Prover vs. Gemini CLI.

(c) DeepSeek-V4-Flash: CoCo-Prover vs. DeepSeek Harness.

Figure 5: **Per-problem cost and wall-clock time of every solved problem in CLEVER benchmark, for CoCo-Prover and each coding agent.** Each point is one problem solved and plotted at its wall-clock time and monetary cost. Triangles on the axes mark each system's mean cost (◀) and mean time (▼) over its own solved set.

C.2 Per-Problem Cost and Time

Figure 5 opens the CLEVER results into one point per solved problem, comparing CoCo-Prover with the coding agent of each backbone. Two patterns hold across backbones. At comparable wall-clock time its points also lie below the coding agent's and avoid their high-cost tail, which reaches about $30 per problem under Gemini-3.5-Flash against about $13 for CoCo-Prover. The average cost also covers the harder problems the baselines never solve so it is higher than the coding agents on GPT model. CoCo-Prover has a higher mean wall-clock time under GPT-5.6-Terra and Gemini-3.5-Flash, but a lower mean under DeepSeek-V4-Flash. Thus, wall-clock time is the one axis where CoCo-Prover does not win: a coding agent writes a whole proof in one shot, whereas CoCo-Prover selects, routes, and Lean-checks one goal before purchasing the next. That sequential, verified progress is what keeps cost low but it takes longer, which is also discussed in Section F.

C.3 Discussion of DeepSeek Model on Baselines

The gap between CoCo-Prover and the baselines is widest under DeepSeek-V4-Flash, and the reason is that the baselines are not built for how this model spends its reasoning budget. Two of the three baseline families run inside an agent scaffold designed around another provider's model. Humanize is built on general coding agents such as Codex and Claude Code, so running it on DeepSeek-V4-Flash means driving a Codex-shaped loop with a model the loop was not tuned for. The coding-agent baseline uses DeepSeek's own harness, which is the intended scaffold but still gives the model one large context holding the whole task. Both arrangements share the same assumption: the model is asked to reason about a long context and produce a proof in one turn.

DeepSeek-V4-Flash usually has long reasoning traces and spends much of its wall-clock budget inside them, and the effect compounds in an agentic setting, where each turn carries the accumulated transcript. We observe two resulting failure modes. In the first, a run is still reasoning when the wall-clock cap expires. In the second, the model finishes reasoning and then declines the task, reporting that the goal is too hard or that it cannot complete the proof. Both leave the run unsolved after real spending, which is why the cost saving on commonly solved problems is largest there.

However, CoCo-Prover's fine-grained control on proof graphs supplies exactly that: the router fixes a reasoning effort and working budget per invocation, and each specialist sees only its own goals with the prepared context it needs. This is why the framework can transfer across backbone LLMs.

C.4 Run-to-Run Variance

Model outputs are stochastic, so a single run can overstate or understate a system's cost. We measure this variance at two scales: over the three function-level benchmarks in full, which tells us whether the headline results rest on a lucky run, and over the 30-problem ablation subset, which sets the

noise floor against which a component ablation must be read. Both use the same provider, caps, and proof-review procedure as the main evaluation.

Full function-level benchmarks. We run the complete CoCo-Prover three times with Gemini-3.5-Flash on all of CLEVER, VERINA, and AlgoVeri, 427 problems in total. Total cost over the three benchmarks is $969.8 on average with a standard deviation of $89.5, a coefficient of variation of 9.2%. The spread is therefore an order of magnitude smaller than the cost gaps the comparison rests on (Table 2), so the reported savings are not an artifact of which run we report. The solved set is identical across the three runs on all three benchmarks, so the solve rates of Table 1 are stable and the variance is in cost alone.

Ablation subset across backbones. We then run the complete CoCo-Prover three times with each of the three backbones on the 30 ablation problems of Section 4.2—10 each from CLEVER, VERINA, and AlgoVeri, spanning difficulty levels from easy to hard. Table 7 reports, for each backbone and benchmark, the mean and standard deviation of cost over the three runs, together with the coefficient of variation (CV, the standard deviation divided by the mean); every run solves all 30 problems, so the spread is in cost alone and not in what was proved.

Six of the nine cells have a CV below 9%, and the two largest, AlgoVeri with GPT-5.6-Terra (34.2%) and CLEVER with DeepSeek-V4-Flash (22.0%), arise for different reasons. The AlgoVeri figure is genuine cost dispersion: the standard deviation is $5.29 on a $15.47 mean, the largest absolute spread in the table, and AlgoVeri's algorithmic goals admit proofs of very different lengths, so a single unlucky decomposition changes the bill substantially. The DeepSeek figure is an artifact of scale: its standard deviation is only $0.93 but DeepSeek-V4-Flash is cheap enough ($3.00–$7.66 per benchmark) that a small absolute spread is a large relative one. Absolute variation is therefore modest everywhere, and the ranking of backbones by cost is stable across all three runs.

The Gemini-3.5-Flash means are the complete-system figures used in the component ablation (Table 3), and their CVs (2.4%–8.3%) indicate how large an ablation difference must be to exceed run-to-run noise. The router and Lemma Planner ablations, at 39.9% and 41.6% above the complete system, are far outside this range; the automation ablation, at 5.8% overall, is comparable to it, and we therefore read that component's effect as small rather than precisely measured.

Table 7: **Run-to-run variance of CoCo-Prover.** We run the complete system three times with each backbone on the 30 ablation problems (10 each from CLEVER, VERINA, and AlgoVeri) and report the mean $\pm$ standard deviation of total cost in USD. Parenthesized percentages are coefficients of variation (standard deviation divided by the mean). Every run solves all 30 problems. The Gemini-3.5-Flash means are the complete-system figures used in Table 3.

Backbone	CLEVER Cost (USD)	VERINA Cost (USD)	AlgoVeri Cost (USD)
GPT-5.6-Terra	17.85 ± 0.80 (4.5%)	30.32 ± 2.54 (8.4%)	15.47 ± 5.29 (34.2%)
Gemini-3.5-Flash	28.64 ± 1.61 (5.6%)	43.49 ± 1.06 (2.4%)	38.82 ± 3.22 (8.3%)
DeepSeek-V4-Flash	4.22 ± 0.93 (22.0%)	7.66 ± 0.91 (11.9%)	3.00 ± 0.23 (7.7%)

C.5 More Component Ablation Discussion

The component ablation of Section 4.2 removes one action agent at a time, or the router's adaptive choice among them, while keeping the rest of the system intact. Two further components of CoCo-Prover admit no such ablation, and we explain here why we do not report one rather than leaving the omission unexplained.

The two-level proof graph. The graph is not a component that the rest of the system uses. It is the state the rest of the system is defined over. Removing it leaves nothing to select from: there is no dependency-ready frontier to maintain, no topological order in which goals become available, no consumers to share a helper lemma with, and no conditional descendants to reopen when a provisional lemma is retracted. The scheduler, the router's grouping, and the Recovery Reviewer's nearest-safe-point backtracking all read the graph, so a variant without it is not CoCo-Prover with one part disabled but a different system—one that proves each obligation in isolation, which is what the baselines already do. The controlled comparison for the graph is therefore the baseline

comparison of Section 4.1, and the case studies of Section D show concretely what the graph buys on individual tasks.

The adaptive routing rules. Removing the explicit rules does not produce a router that stops adapting. The evidence packet still carries the execution history from shared memory so a router without written rules can read the same signals and change its behavior in response to them, only implicitly and without a logged, versioned artifact we can inspect. Such an ablation would therefore measure the value of *writing the rules down*, not the value of adaptation, and a difference in either direction would be hard to attribute: a null result would not show that adaptation is unnecessary, and a positive one would not separate the rules from the extra tokens that carrying them adds to every packet. What the experiments do isolate is adaptive action choice as a whole: *CoCo-Prover w/o Router* replaces it with one fixed order over the same graph and raises cost by 39.9% (Table 3).

D Case Studies

We walk through two tasks end to end: a function-level task, CLEVER `prob_15` (Section D.1), where all systems face one obligation and the question is cost; and a repository-level task, Vero `prob_14` (Section D.2), with 56 interdependent obligations, where the question is whether provability can be reached at all.

D.1 CLEVER `prob_15`: a function-level cost comparison

CoCo-Prover proves CLEVER `prob_15` for $2.36; the strongest baselines prove it for $8.67 (Codex) and $10.74 (Humanize), and Ax-Prover-Base spends $5.48 and fails. All four systems use the same LLM (GPT-5.6-Terra) at the same provider prices under the caps of the main evaluation.

D.1.1 The task

`prob_15` is HumanEval's `string_sequence`: return the numbers $0..n$ joined by single spaces.

```
def problem_spec
 (implementation: Nat → String)
 (n: Nat) :=
let spec (result: String) :=
  let result_nums := result.splitOn " ";
  result_nums.length = n + 1 ∧
  ∀ i, i < n + 1 → result_nums[i]! = i.repr;
∃ result, implementation n = result ∧ spec result

def implementation (n: Nat) : String :=
  " ".intercalate ((List.range (n + 1)).map (fun i => i.repr))

theorem correctness (n: Nat) : problem_spec implementation n := sorry
```

The specification reads the output back with `String.splitOn " "`, so the proof must show that splitting on the separator undoes `String.intercalate` whenever no piece contains the separator and Lean 4 core has no such theorem: `String.splitOn` is a byte-position scanner (`String.splitOnAux`) using well-founded recursion over UTF-8 offsets; `String.intercalate` is implemented by a private accumulator worker `String.intercalate.go`; the only ready-made fact is the list-level `List.splitOn_intercalate`; and the proof also needs "`Nat.repr` never produces a space," which requires reasoning about `Nat.toDigits`. A proof therefore has to connect three layers: the string API, character lists, and the byte-level scanner. Every baseline named exactly this connection as its obstacle (Section D.1.4).

D.1.2 Head-to-head outcome

Table 8 summarizes the four runs. Two observations frame the rest of the section. First, the two successful baselines reach essentially the same mathematics: Codex's helper lemma set mirrors our lemma graph so the $3.7\times$–$4.5\times$ cost gap is organizational. Second, the failure is organizational too:

Table 8: **CLEVER `prob_15` under identical LLM, prices, and caps.** Ax-Prover-Base's 288,669 output tokens include 228,209 reasoning tokens. Codex and Humanize find proofs of similar size and structure to ours; most of their extra cost is cache reads from re-sending an ever-growing context at every step. Ax-Prover-Base keeps its context small but spends $4.33 on output, mostly reasoning, re-deriving a monolithic proof each iteration, and never compiles a complete proof.

	CoCo-Prover	Codex	Humanize	Ax-Prover-Base
Outcome	**proved**	proved	proved	failed (`sorry`)
Total cost (USD)	**2.36**	8.67 (3.7×)	10.74 (4.5×)	5.48
Wall-clock time (s)	2,816	2,550	4,538	6,893
Uncached input tokens	425,737	408,271	1,000,932	452,135
Cache-read tokens	2,115,584	26,675,200	23,777,280	77,568
Total prompt tokens	**2.54M**	27.08M	24.78M	0.53M
Output tokens	51,310	65,566	152,920	288,669
Proof size	191 lines, 9 lemmas	149 lines, 8 helpers	184 lines, 13 helpers	—
Cost: uncached input	1.06	1.02	2.50	1.13
Cost: output	0.77	0.98	2.29	4.33
Cost: cache reads	0.53	**6.67**	**5.94**	0.02

Ax-Prover-Base correctly identified every missing fact yet had no mechanism to turn a blocker into a separately provable goal.

D.1.3 How CoCo-Prover proved it

The run took 8 controller iterations and 18 routed agent invocations: 7 planning invocations (router reasoning and the Lemma Planner, at a high-thinking configuration θ) and 11 Proof Writer invocations (at a low-thinking θ). It had zero backtracks, zero retracted lemmas, and zero rejected proof scripts: every lemma was proposed once and proved once, every Proof Writer invocation closed its target, so no target needed a second invocation, and nothing that was proved was ever redone.

Table 9: **Iteration timeline** (declaration numbers from Table 10). Each iteration pushes the open gap exactly one layer down: from the specification, to the string API, to character lists, to the byte scanner. Deterministic Automation opened iteration 1 by producing the five-tactic skeleton of `correctness` at zero model cost; on the eight lower-level goals it failed fast (heartbeat timeout, no progress, or a partial script below the quality threshold) and the controller moved on without spending model budget there.

Iter.	Open goal(s) at start	New helper lemmas proposed	Closed by Proof Writer	Cost ($)
1	`correctness`	`implementation_splitOn`	both conjuncts of the spec	0.263
2	`implementation_splitOn`	round-trip (#4) + no-space (#3)	`implementation_splitOn`	0.262
3	#3, #4	`toList`/`splitOn` bridges (#5, #6)	#4 (`calc` chain); #3	0.453
4	#5, #6	private worker (#7) + predicate split (#8)	#5, #6	0.266
5	#7, #8	byte-scanner agreement (#9)	#7 (induction generalizing `acc`); #8	0.263
6	#9	UTF-8 offset lemma (#10)	#9 (25 calls, $0.325: hardest step)	0.612
7	#10	— (plan via `String.next_of_valid`)	#10	0.128
8	—	**proof complete** at 2,815 s (run ended at 2,816 s)		

Table 9 gives the timeline, and Figure 6 the lemma-dependency graph it builds: each iteration extends a nine-lemma chain, pushing the open gap one layer further down, from the specification to the string API, then to character lists, and finally to the byte-level scanner and its UTF-8 offset arithmetic.

Lemma-by-lemma Lean code. We list every declaration in discovery order, from the top of the DAG down: its role, the planner's plan (quoted from the transcript), and the final Lean code.

1. `correctness`: the target theorem. Deterministic Automation applied a partial script at zero model cost (static analysis check in rounter), leaving the two conjuncts open; the first Lemma Planner invocation (11 calls, $0.229) proposed `implementation_splitOn` and Proof Writer closed both conjuncts using exactly the planned tactics.

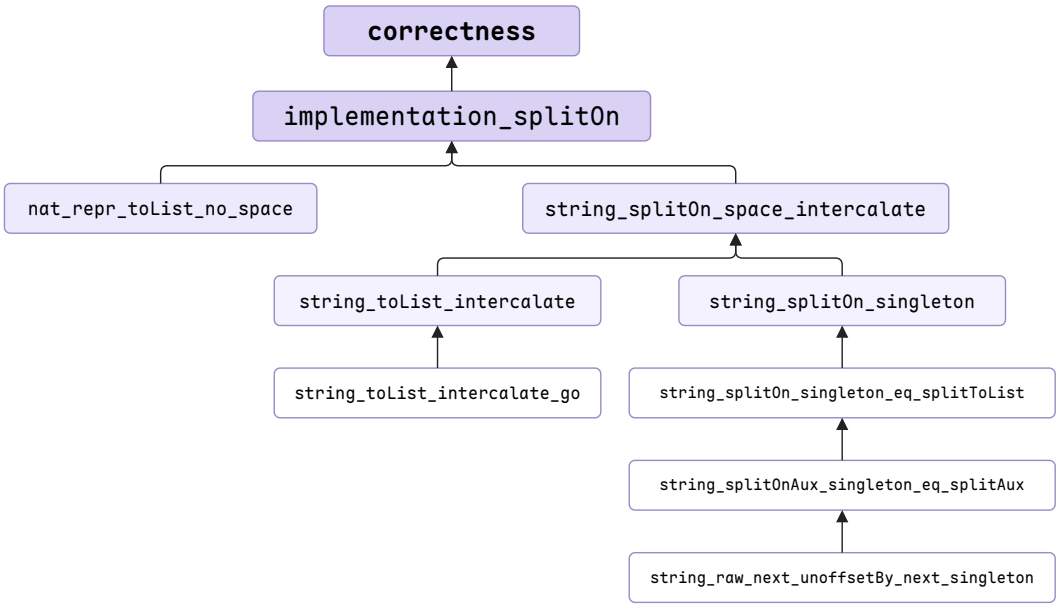


Figure 6: **Lemma-dependency graph of the final proof of CLEVER `prob_15`.** The root is the specification-level `correctness` theorem; every edge points from a helper lemma to the declaration whose proof consumes it. Depth corresponds to the layer of abstraction each iteration reached: `implementation_splitOn` instantiates the specification to the implementation, the round-trip and no-space lemmas work at the string API, the `toList` and `splitOn` bridges move to character lists (including the private accumulator worker), and the lowest branch descends through the byte-level scanner agreement to UTF-8 offset arithmetic. Each declaration was proposed with a `sorry` body, checked for counterexamples, and proved exactly once (Table 10).

Table 10: **Every declaration in the final proof.** Each lemma was proposed with a `sorry` body and a structured header (`@description`, `@lib_depends_on`, `@used_in`, and `@depends_on` once it has generated dependencies), tested for counterexamples before acceptance, and proved exactly once, in the iteration after it was proposed.

#	Lemma/Theorem	Proposed	Counterexample check	Lines
1	`correctness`	target theorem	—	14
2	`implementation_splitOn`	iter. 1	no counterexample	9
3	`nat_repr_toList_no_space`	iter. 2	no counterexample	9
4	`string_splitOn_space_intercalate`	iter. 2	inconclusive (timeout)	28
5	`string_toList_intercalate`	iter. 3	no counterexample	9
6	`string_splitOn_singleton`	iter. 3	no counterexample	5
7	`string_toList_intercalate_go`	iter. 4	no counterexample	11
8	`string_splitOn_singleton_eq_splitToList`	iter. 4	no counterexample	5
9	`string_splitOnAux_singleton_eq_splitAux`	iter. 5	inconclusive (not testable)	55
10	`string_raw_next_unoffsetBy_next_singleton`	iter. 6	inconclusive (not testable)	15

```
theorem correctness
(n: Nat)
: problem_spec implementation n := by
  unfold problem_spec          -- Deterministic Automation skeleton
  unfold implementation        --    (no LLM tokens)
  intro spec
  simp_all [spec]
  apply And.intro
  · rw [implementation_splitOn]                    -- plan, conjunct 1
    simp only [List.length_map, List.length_range]
  · intro i a                                      -- plan, conjunct 2
    rw [implementation_splitOn]
```

```
    rw [List.getElem?_map, List.getElem?_range a]
    rfl
```

2. `implementation_splitOn`: specialize the round-trip to the implementation. Every lemma is proposed with a structured header; this one reads:

```
/-
@name: implementation_splitOn
@description: Splitting the space-intercalated decimal representations of the
  numbers in `List.range (n + 1)` recovers exactly that mapped range.
@lib_depends_on: [String.splitOn, String.intercalate, Nat.repr, List.range]
@used_in: [correctness]
-/
```

The iteration-2 Lemma Planner proposed the two lemmas the plan relies on (#3 and #4). The Proof Writer closed it in 3 calls:

```
lemma implementation_splitOn (n : Nat) :
    (" ".intercalate ((List.range (n + 1)).map (fun i => i.repr))).splitOn =
      (List.range (n + 1)).map (fun i => i.repr) := by
  apply string_splitOn_space_intercalate
  · intro s hs
    rw [List.mem_map] at hs
    rcases hs with ⟨i, hi, rfl⟩
    exact nat_repr_toList_no_space i
  · simp
```

3. `nat_repr_toList_no_space`: decimal digits contain no space. This is Ax-Prover-Base's `hnoSpace` blocker (Section D.1.4). The plan argues by contradiction through the *library* fact `Nat.isDigit_of_mem_toDigits` instead of unfolding `Nat.toDigits`—unfolding is exactly where Ax-Prover-Base got stuck. The Proof Writer wrote `(10 : Nat)` explicitly and added a final `contradiction`:

```
lemma nat_repr_toList_no_space (n : Nat) : (' ' : Char) ∉ n.repr.toList := by
  intro h
  have hm : (' ' : Char) ∈ (10 : Nat).toDigits n := by
    simpa [Nat.repr] using h
  have hd : (' ' : Char).isDigit = true :=
    Nat.isDigit_of_mem_toDigits (by norm_num) (by norm_num) hm
  norm_num [Char.isDigit] at hd
  contradiction
```

4. `string_splitOn_space_intercalate`: the string round-trip. The central string-level fact; Humanize rounds 1–2 reported it as missing from the library. The Lemma Planner first proposed the two character-list bridge lemmas (#5 and #6), then the Proof Writer used the four-step `calc` structure and fixed three details against Lean: `rintro` became `rcases List.mem_map.mp`, `String.singleton_eq` became `String.singleton_eq_ofList`, and a final `simp [List.map_map]` that did not close the goal was replaced by an explicit induction using `String.ofList_toList`:

```
lemma string_splitOn_space_intercalate (ss : List String)
    (hspace : ∀ s ∈ ss, (' ' : Char) ∉ s.toList) (hne : ss ≠ []) :
    (" ".intercalate ss).splitOn = ss := by
  have hspace' : ∀ l ∈ ss.map String.toList, (' ' : Char) ∉ l := by
    intro l hl
    rcases List.mem_map.mp hl with ⟨s, hs, rfl⟩
    exact hspace s hs
  have hne' : ss.map String.toList ≠ [] := by
    simpa using hne
  calc
    -- (a) String.splitOn → List.splitOn on characters      [lemma #6]
    (" ".intercalate ss).splitOn =
```

```
        (List.splitOn (' ' : Char) ((" ".intercalate ss).toList)).map
          String.ofList := by
      simpa [String.singleton_eq_ofList] using
        (string_splitOn_singleton (" ".intercalate ss) (' ' : Char))
  -- (b) String.intercalate → List.intercalate on characters  [lemma #5]
  _ = (List.splitOn (' ' : Char)
      (([' '] : List Char).intercalate (ss.map String.toList))).map
        String.ofList := by
    rw [string_toList_intercalate]
    rfl
  -- (c) the list-level round-trip from the library
  _ = (ss.map String.toList).map String.ofList := by
    rw [List.splitOn_intercalate (ss.map String.toList) (' ' : Char)
      hspace' hne']
  -- (d) back to strings
  _ = ss := by
    rw [List.map_map]
    clear hspace' hne' hspace hne
    induction ss with
    | nil => rfl
    | cons s ss ih =>
      simp only [List.map_cons, Function.comp_apply]
      rw [String.ofList_toList, ih]
```

5. `string_toList_intercalate`: `String.intercalate` as a list operation. AxProver-Base's `hintercalate` blocker. The Lemma Planner first proposed #7, a lemma about the private worker, then split on the list of strings; the Proof Writer completed:

```
open private String.intercalate.go from Init.Data.String.Defs in
lemma string_toList_intercalate (sep : String) (ss : List String) :
    (sep.intercalate ss).toList = sep.toList.intercalate (ss.map String.toList) := by
  cases ss with
  | nil =>
    rfl
  | cons a as =>
    simpa only [String.intercalate, List.map_cons] using
      (string_toList_intercalate_go a sep as)
```

6. `string_splitOn_singleton`: `String.splitOn` as a list operation. Part of AxProver-Base's `hbridge` blocker. The Lemma Planner separated the byte-scanner part into #8 and connected the rest to the library theorem `String.splitToList_of_valid`. The Proof Writer completed:

```
lemma string_splitOn_singleton (s : String) (c : Char) :
    s.splitOn (String.singleton c) = (s.toList.splitOn c).map String.ofList := by
  rw [string_splitOn_singleton_eq_splitToList]
  rw [String.splitToList_of_valid]
  rfl
```

7. `string_toList_intercalate_go`: a lemma about a private function. The Proof Writer strengthened a plan "induct on `ss`, generalizing `acc`, since `String.intercalate.go` advances precisely by replacing the accumulator with `acc ++ sep ++ head`." with a `cases` split and two extra `simp` lemmas:

```
open private String.intercalate.go from Init.Data.String.Defs in
lemma string_toList_intercalate_go (acc sep : String) (ss : List String) :
    (String.intercalate.go acc sep ss).toList =
      sep.toList.intercalate (acc.toList :: ss.map String.toList) := by
  induction ss generalizing acc with
  | nil =>
    simp [String.intercalate.go, List.intercalate]
  | cons s ss ih =>
```

```
    rw [String.intercalate.go]
    rw [ih (acc := acc ++ sep ++ s)]
    cases ss <;> simp [String.toList_append, List.intercalate,
      List.intersperse, List.append_assoc]
```

8. `string_splitOn_singleton_eq_splitToList`: pattern split = predicate split. The Lemma Planner pushed the scanner itself into lemma #9; the Proof Writer used the lemma as written:

```
lemma string_splitOn_singleton_eq_splitToList (s : String) (c : Char) :
    s.splitOn (String.singleton c) = s.splitToList (fun d => d == c) := by
  unfold String.splitOn String.splitToList
  rw [if_neg (by simpa only [beq_iff_eq] using String.singleton_ne_empty)]
  exact string_splitOnAux_singleton_eq_splitAux s c 0 0 []
```

9. `string_splitOnAux_singleton_eq_splitAux`: the byte-level scanners agree. The hardest step. Ax-Prover-Base's attempts to unfold this recursion hit "recursion-depth failures"; Humanize's round 1 called the scanner "irreducible." The Lemma Planner invocation separated the offset arithmetic into lemma #10. Then the Proof Writer planned the four cases of a well-founded `String.splitOnAux.induct`, "retaining the invariant only for the reachable singleton-separator state $j = 0 \ldots$ because the recursor also exposes unreachable states having a nonzero separator position." The Proof Writer then turned the prose plan into 55 lines, discovering the exact `if`/`dif` rewrite sequences case by case:

```
lemma string_splitOnAux_singleton_eq_splitAux
    (s : String) (c : Char) (b i : String.Pos.Raw) (r : List String) :
    s.splitOnAux (String.singleton c) b i 0 r =
      s.splitAux (fun d => d == c) b i r := by
  refine String.splitOnAux.induct s (String.singleton c)
    (motive := fun b i j r => j = 0 →
      s.splitOnAux (String.singleton c) b i j r =
        s.splitAux (fun d => d == c) b i r)
    ?_ ?_ ?_ ?_ b i 0 r rfl
  -- case 1: at the end of `s`
  · intro b i j r hi hj
    simp [String.splitOnAux.eq_def, String.splitAux.eq_def, hi]
  -- case 2: character matches; the (one-character) separator is complete
  · intro b i j r hi hmatch i1 j1 hend ih hj
    subst j
    have hsingle : String.Pos.Raw.get (String.singleton c) 0 = c := by
      rw [← String.front_eq_get, String.front_eq]
      simp [String.toList_singleton]
    have hnext : String.Pos.Raw.next (String.singleton c) 0 =
        { byteIdx := c.utf8Size } := by
      simpa only [String.singleton_eq_ofList, List.nil_append,
        List.cons_append, String.utf8Len_nil, Nat.zero_add] using
        String.next_of_valid [] c []
    have hsingle_end : String.Pos.Raw.atEnd (String.singleton c)
        (String.Pos.Raw.next (String.singleton c) 0) = true := by
      rw [hnext]
      simp [String.Pos.Raw.atEnd, String.utf8ByteSize_singleton]
    have hget : String.Pos.Raw.get s i = c := by
      simpa [hsingle] using hmatch
    rw [string_raw_next_unoffsetBy_next_singleton s i c hget] at ih
    rw [String.splitOnAux.eq_def]
    rw [String.splitAux.eq_def]
    rw [if_neg hi, if_pos hmatch, if_pos hsingle_end]
    simp only [hget]
    rw [string_raw_next_unoffsetBy_next_singleton s i c hget]
    rw [dif_neg hi, if_pos (by simp)]
    exact ih rfl
  -- case 3: match, but separator incomplete (impossible for a singleton)
  · intro b i j r hi hmatch i1 j1 hend ih hj
    subst j
```

```
    have hnext : String.Pos.Raw.next (String.singleton c) 0 =
        { byteIdx := c.utf8Size } := by
      simpa only [String.singleton_eq_ofList, List.nil_append,
        List.cons_append, String.utf8Len_nil, Nat.zero_add] using
        String.next_of_valid [] c []
    have hsingle_end : String.Pos.Raw.atEnd (String.singleton c)
        (String.Pos.Raw.next (String.singleton c) 0) = true := by
      rw [hnext]
      simp [String.Pos.Raw.atEnd, String.utf8ByteSize_singleton]
    exact False.elim (hend hsingle_end)
  -- case 4: no match; separator position resets to 0
  · intro b i j r hi hmatch ih hj
    subst j
    have hsingle : String.Pos.Raw.get (String.singleton c) 0 = c := by
      rw [← String.front_eq_get, String.front_eq]
      simp [String.toList_singleton]
    have hnomatch : ¬(String.Pos.Raw.get s i == c) = true := by
      simpa [hsingle] using hmatch
    have hunoffset : i.unoffsetBy 0 = i := by
      apply String.Pos.Raw.ext
      simp [String.Pos.Raw.unoffsetBy]
    rw [String.splitOnAux.eq_def]
    rw [String.splitAux.eq_def]
    rw [if_neg hi, if_neg hmatch]
    simp only [hsingle, hunoffset]
    rw [dif_neg hi, if_neg hnomatch]
    exact ih rfl
```

10. `string_raw_next_unoffsetBy_next_singleton`: UTF-8 offset arithmetic. The Proof Writer first reasoned a plan which reduces position equality to byte-index equality, computes the singleton advance with the library theorem `String.next_of_valid`, and cancels the subtraction with `Nat.add_sub_cancel`. The Proof Writer then added `String.Pos.Raw.mk_zero` and `Nat.zero_add` to the planned `simpa only` list:

```
lemma string_raw_next_unoffsetBy_next_singleton
    (s : String) (i : String.Pos.Raw) (c : Char)
    (hget : String.Pos.Raw.get s i = c) :
    (String.Pos.Raw.next s i).unoffsetBy
        (String.Pos.Raw.next (String.singleton c) 0) = i := by
  have hsingleton : String.Pos.Raw.next (String.singleton c) 0 =
      { byteIdx := c.utf8Size } := by
    simpa only [String.singleton_eq_ofList, List.nil_append,
      String.utf8Len_nil, String.Pos.Raw.mk_zero, Nat.zero_add] using
      (String.next_of_valid [] c [])
  apply String.Pos.Raw.ext
  rw [String.Pos.Raw.byteIdx_unoffsetBy, hsingleton]
  simp only [String.Pos.Raw.next, hget, String.Pos.Raw.addChar_eq]
  exact Nat.add_sub_cancel _ _
```

What the lemma-by-lemma view shows. The Lemma Planner decomposed subgoals. Every blocker became a lemma one layer down including #7 (the private worker), #8/#9 (the byte scanner), #10 (offset arithmetic). The Proof Writer often reasoned about the mathematics then fixes the Lean details. And library facts were found and used when they existed (`Nat.isDigit_of_mem_toDigits`, `List.splitOn_intercalate`, `String.splitToList_of_valid`, `String.splitOnAux.induct`, `String.next_of_valid`). New lemmas were written only for facts missing from the library.

D.1.4 How the baselines spent their budgets

Codex: $8.67, proved. Codex worked in one continuous turn of about 193 tool steps—101 shell commands (92 of them Lean checks) and 92 scratch-file edits—plus 5 progress messages. Its mes-

sages show it reached the same mathematical plan as ours: "the remaining work is the string round-trip fact for `intercalate`/`splitOn` ... reducing `splitOn " "` to the library's character-splitting semantics. The proof is now at the low-level auxiliary recursion, with termination decreasing by UTF-8 position." Its final proof has a matching shape (excerpt):

```
theorem splitOn_space_aux (s : String) (b i : String.Pos.Raw) (r : List String) :
    s.splitOnAux " " b i 0 r = s.splitAux (fun c => c == ' ') b i r := by
  ...
termination_by s.utf8ByteSize - i.byteIdx
decreasing_by
  all_goals
    apply dec_next s i
    simp_all [String.Pos.Raw.atEnd]

theorem splitOn_space (s : String) :
    s.splitOn " " = s.splitToList (fun c => c == ' ') := ...

theorem toList_intercalate (sep : String) (xs : List String) :
    (sep.intercalate xs).toList = sep.toList.intercalate (xs.map String.toList) := ...

theorem space_not_mem_repr (n : Nat) : ' ' ∉ n.repr.toList := ...

theorem split_intercalate_repr (n : Nat) :
    (" ".intercalate ((List.range (n + 1)).map (fun i => i.repr))).splitOn " " =
      (List.range (n + 1)).map (fun i => i.repr) := ...
```

For the private `String.intercalate.go`, Codex wrote a custom meta-level tactic that rebuilds the private declaration's internal mangled name by hand:

```
open Lean Elab Tactic Meta
elab "unfoldStringIntercalateGo" : tactic => liftMetaTactic fun goal => do
  let name := Name.anonymous.str "_private" |>.str "Init" |>.str "Data"
    |>.str "String" |>.str "Defs" |>.num 0 |>.str "String"
    |>.str "intercalate" |>.str "go"
  let goal ← unfoldTarget goal name
  return [goal]
```

This works but depends on the private name's internal encoding. Where the money went: every step re-sent the whole transcript, including all earlier Lean outputs and edits—27.08M prompt tokens, of which 26.68M were cache reads ($6.67, 77% of its cost), roughly 140k prompt tokens per step against our 15.8k per call (2.54M over 161 calls). The mathematics was the same and the difference is context management, which demonstrates the advantage of the metalevel decision-making on agent orchestration.

Humanize: $10.74, proved on round 9 of 9. Humanize runs a fresh worker each round, with gate checks and a final reviewer. Rounds 1–8 all ended without a proof, costing $9.05 (84% of the total); each worker restarted the exploration and hit the same wall (Table 11).

Two structural problems stand out. First, progress is not kept: round 3 proved the string-to-character splitting bridge but left the candidate file unchanged ("The candidate remains unchanged"), so rounds 4–8 rediscovered it. Second, a blocker is never turned into a goal: four of the five rounds 4–8 named the private `String.intercalate.go` as the obstacle, where CoCo-Prover stated it as a checked lemma *about* the private function and proved it for $0.037. (A second, earlier Humanize run also proved the task, for $6.01 over 3 worker rounds—still $2.5\times$ our cost.)

Ax-Prover-Base: $5.48, failed. Ax-Prover-Base repeatedly proposes a complete proof of `correctness` and compiles it: over 50 iterations and 151 LLM calls, 49 of 50 builds failed to compile, and the run ended by exhausting its iteration limit. Its final self-assessment lists three blocking inline `have`s that it kept trying to prove inside the body of `correctness`. The sketch below shows that the three `have` statements and the quoted failure reasons are verbatim, and the surrounding lines follow its described strategy:

```
theorem correctness (n: Nat) : problem_spec implementation n := by
```

Table 11: **Humanize's nine rounds.** The eight failed rounds name the same two obstacles that our lemmas #7–#10 dissolve.

Round	Cost	Worker's stated blocker (from the run log)
1	\$0.83	"`String.splitOn`'s irreducible positional scanner; the needed semantic bridge is absent"
2	\$0.98	"`String.splitOn`'s irreducible UTF-8 scanner; no applicable semantic split/intercalate lemma"
3	\$2.77	established the `splitOn " "` $\leftrightarrow$ character-splitting bridge, but "the remaining obstacle is … the opaque/internal `String.intercalate` implementation"
4	\$0.99	"the recursive join helper is inaccessible for induction"
5	\$0.79	"the required semantic bridge … is not exported; `List.splitOn_intercalate` cannot be applied"
6	\$0.99	"the unresolved piece is a public recursive lemma for this Lean version's private `String.intercalate` helper"
7	\$0.88	"its recursive worker is private/opaque"
8	\$0.83	"an unexported/private recursive helper behind `String.intercalate`"
9	\$1.54 + \$0.15 reviewer	proof completed; gates pass

```
unfold problem_spec
refine ⟨implementation n, rfl, ?_⟩
-- stable parts: htoList_inj, hmap_inj, the List.splitOn_intercalate
-- step, both conjuncts
have hbridge : (s.splitOn " ").map String.toList = s.toList.splitOn ' ' := by
  ...  -- "recursive unfolding caused recursion-depth failures"
have hintercalate : (" ".intercalate xs).toList =
    [' '].intercalate (xs.map String.toList) := by
  ...  -- "simplification did not reduce its base and inductive cases"
have hnoSpace : ' ' ∉ i.repr.toList := by
  ...  -- "got stuck on the conditional implementation of `Nat.toDigits`"
...
```

Each blocker corresponds to a lemma or chain CoCo-Prover proved separately: `hbridge` to lemmas #6, #8, #9, and #10; `hintercalate` to #5 and #7; and `hnoSpace` to #3. A monolithic attempt fails if *any* part fails, so the parts that did work (`htoList_inj`, `hmap_inj`, and the use of `List.splitOn_intercalate`) were never kept, and the model spent \$4.33 on output, including 228k reasoning tokens, re-deriving the whole proof each iteration.

D.1.5 Why CoCo-Prover is cheaper

Six mechanisms, each visible in the trace, account for the gap; each is a design component of Section 3.2 doing exactly the job it was built for. *(1) Small, focused contexts—two packets, neither of them the whole search.* The router's packet is frontier-wide but shallow: tactic states, the graph neighborhood, and failure memory restricted to the open goals, with no proof transcripts. The specialist's is narrow and deep: the group the router chose, its preliminary analysis, and the excerpt of the router's conversation the invocation carries. Neither packet grows with the search, so each call carries 15.8k prompt tokens, against $\sim$140k per Codex step and 1.4M–7.6M cache reads per Humanize round. This alone explains the $10\times$ gap in prompt tokens at similar proof sizes. *(2) Verified progress is never repeated—the persistent two-level proof graph.* Kernel-checked promotion commits every proved lemma to the dependency graph, and the Kahn frontier hands later invocations only what remains open, so all 18 invocations contributed to the final proof. However, Humanize, restarting a fresh worker per round, spent 84% of its budget re-exploring, and Ax-Prover-Base, keeping only a memory of recent failures rather than persistent proof state, discarded 49 of 50 builds. *(3) Blockers become subgoals—the Lemma Planner.* Every baseline correctly *named* the missing facts but had no agent whose job is to act on them. The Lemma Planner states each blocker as a checked lemma and admits it to the dependency graph only after elaboration and acyclicity, whereupon its proof obligations enter the frontier as ordinary goals small enough for the Proof Writer to close. *(4) Expensive reasoning where it matters—the router's configuration choice θ.* Because every action is a

separately priced purchase (G, o, θ), the router bought lemma planning at a high-thinking and script synthesis (proof writer) at a low-thinking θ. *(5) Free goals stay free—Deterministic Automation under the same-provability cheap-first rule.* It produced the opening skeleton of `correctness` at zero model cost and failed fast everywhere else, so no LLM budget was ever spent on a goal that symbolic procedures could open or close. *(6) Conjectures are tested before they are trusted—the counterexample check under the tests-do-not-certify invariant.* All 9 proposed lemmas were tested for $0.12 before anything was built on them. Tests may reject a candidate but never promote one, so refutable statements are filtered before proof effort, while soundness stays with the kernel.

D.2 VERO PROB_14: INTERVAL-SET ALGEBRA, 56 SPECIFICATIONS

Vero `prob_14` asks for proofs of 56 specification theorems about a small interval-set library. CoCo-Prover, running on GPT-5.6-Terra, proved all 56, and every proof is checked by Lean. It built 55 helper lemmas along the way, organized into a dependency hierarchy up to six levels deep. The coding-agent baseline on the same backbone LLM proved 5 of the 56. This subsection explains how full provability was reached: how the targets were split up, which lemmas carry the load, and how the proofs of the hardest facts were constructed.

D.2.1 THE TASK

An interval `Iv` is a half-open range $[lo, hi)$ of integers; a `List Iv` denotes the union of its intervals. The library under verification has three operations:

```
structure Iv where
  lo : Int
  hi : Int
deriving DecidableEq, Repr, Inhabited

-- frozen point-set semantics that every spec is written against
def covered (x : Int) (ivs : List Iv) : Bool :=
  ivs.any (fun iv => decide (iv.lo ≤ x ∧ x < iv.hi))

def disjointSortedNonempty (ivs : List Iv) : Prop :=
  (∀ iv ∈ ivs, iv.lo < iv.hi) ∧
  (∀ (i : Nat) (h : i + 1 < ivs.length), (ivs[i]).hi < (ivs[i+1]).lo)

def Intervaltree.overlaps : Intervaltree.OverlapsSig :=
  fun a b => decide (a.lo < b.hi ∧ b.lo < a.hi)

-- mergeOverlaps = insertion sort by `lo` (dropping empty intervals),
-- then coalescing
def Intervaltree.insertByLo (iv : Iv) : List Iv → List Iv
  | [] => if iv.lo < iv.hi then [iv] else []
  | y :: ys =>
    if iv.lo < iv.hi then
      if iv.lo ≤ y.lo then iv :: y :: ys
      else y :: Intervaltree.insertByLo iv ys
    else y :: ys

def Intervaltree.sortByLo (ivs : List Iv) : List Iv :=
  ivs.foldr Intervaltree.insertByLo []

def Intervaltree.coalesce : List Iv → List Iv
  | [] => []
  | [x] => [x]
  | x :: y :: rest =>
    if y.lo ≤ x.hi then
      Intervaltree.coalesce
        ({ lo := x.lo, hi := if x.hi < y.hi then y.hi else x.hi } :: rest)
    else
      x :: Intervaltree.coalesce (y :: rest)
  termination_by l => l.length
```

```
def Intervaltree.mergeOverlaps : Intervaltree.MergeOverlapsSig :=
  fun ivs => Intervaltree.coalesce (Intervaltree.sortByLo ivs)

-- chop b e = remove [b, e) from every interval, splitting straddlers
def Intervaltree.chopOne (b e : Int) (iv : Iv) : List Iv :=
  let left  : List Iv :=
    if iv.lo < b then [{ lo := iv.lo, hi := if iv.hi < b then iv.hi else b }] else []
  let right : List Iv :=
    if e < iv.hi then [{ lo := if iv.lo < e then e else iv.lo, hi := iv.hi }] else []
  (left ++ right).filter (fun p => decide (p.lo < p.hi))

def Intervaltree.chop : Intervaltree.ChopSig :=
  fun b e ivs => ivs.flatMap (Intervaltree.chopOne b e)
```

The 56 targets fall into three groups: *overlaps* (6 targets: symmetry, self-overlap, a characterization by a common point, and one target relating `overlaps` to runs of merged intervals); *mergeOverlaps* (39 targets: coverage, canonical form, uniqueness, invariance under permutation and reversal, idempotence, measure bounds, hull endpoints, and point-count properties); and *chop* (11 targets: pointwise removal, idempotence, preservation of canonical form, and interaction with merging). Each target has the form `theorem prove_X : spec_X canonical`; some representative statements:

```
def spec_merge_coverage (impl : RepoImpl) : Prop :=
  ∀ (ivs : List Iv) (x : Int),
    covered x (impl.intervaltree.mergeOverlaps ivs) = covered x ivs

def spec_merge_disjoint_strict (impl : RepoImpl) : Prop :=
  ∀ (ivs : List Iv),
    let out := impl.intervaltree.mergeOverlaps ivs
    (∀ iv ∈ out, iv.lo < iv.hi) ∧
    (∀ (i : Nat) (h : i + 1 < out.length), (out[i]).hi < (out[i+1]).lo)

def spec_merge_canonical_unique (impl : RepoImpl) : Prop :=
  ∀ (ivs c : List Iv),
    disjointSortedNonempty c →
    (∀ x, covered x c = covered x ivs) →
      impl.intervaltree.mergeOverlaps ivs = c

def spec_merge_perm_eq_any (impl : RepoImpl) : Prop :=
  ∀ (xs ys : List Iv),
    List.Perm xs ys →
      impl.intervaltree.mergeOverlaps xs = impl.intervaltree.mergeOverlaps ys

def spec_chop_canonical_merge_fixpoint (impl : RepoImpl) : Prop :=
  ∀ (b e : Int) (ivs : List Iv),
    b < e →
      disjointSortedNonempty ivs →
        impl.intervaltree.mergeOverlaps (impl.intervaltree.chop b e ivs)
          = impl.intervaltree.chop b e ivs
```

Four features make the problem hard. *Three different recursion schemes*: `insertByLo` is structural recursion, `sortByLo` is a fold, and `coalesce` is well-founded recursion whose merge branch recurses on a list whose head has been changed. Unfolding them by hand gives nested `match`/`if` terms with no usable induction hypothesis. *Semantic specifications*: the specs are phrased through `covered` (a point set) and `disjointSortedNonempty` (index-based gap conditions), not through the implementation's structure, so every proof must connect the recursive code to these frozen predicates. *Genuinely global targets*: uniqueness and permutation-invariance need an extensionality principle (*e.g.*, two canonical lists covering the same points are equal) which no single unfolding of `mergeOverlaps` gives. *Scale*: 56 targets share structure, and proving each independently would repeat the same inductions dozens of times.

D.2.2 Outcome

The complete run used about $19 of model usage, roughly $0.35 per target theorem.

Table 12: **Vero `prob_14` outcome.** The final 2,471-line proof file passes the checker: all 56 targets proved with no `sorry` and no axioms beyond Lean's standard ones, and all 56 target statements identical to the originals (no restated targets).

	CoCo-Prover (GPT-5.6-Terra)	Coding-agent baseline (GPT-5.6-Terra)
Targets proved	**56 / 56**	5 / 56
Full file checks in Lean	**yes**	no

D.2.3 How CoCo-Prover proved all 56 targets

The proof was built over 34 controller iterations; the number of proved lemmas/theorems grew steadily: 6 at the start of iteration 2, 25 at iteration 5, 60 at iteration 15, 91 at iteration 26, and all 111 (56 targets + 55 lemmas) by iteration 34. Deterministic Automation ran first on every open goal: it closed 24 goal nodes outright (counted per goal, not per declaration as in Table 13) and 59 times applied a partial script that a specialist then completed. The run made 175 routed invocations—72 planning (router reasoning and the Lemma Planner) and 103 Proof Writer—and accepted 56 helper lemmas, each first passing the counterexample check; 55 appear in the final proof, and one ended up unnecessary. Unlike the zero-backtrack CLEVER run, this run made **52 backtracks**, each with a written reason accepted by the Recovery Reviewer: the controller routinely rewound a branch where automation had unfolded a recursive definition into an unmanageable term, and replaced it with an induction or a lemma application at the nearest safe node (Section D.2.4).

Table 13: **How each declaration was closed.** The 8 targets closed by Deterministic Automation alone are local facts, including `overlaps` symmetry, self-overlap, disjointness, `merge_nil`, `merge_singleton`, `chop_removes`, and `chop_append_coverage`; everything else is built through lemmas, and 7 of the 46 lemma-built targets are themselves reused as lemmas by later targets.

	Closed by Automation	Closed by an LLM script (no new lemma)	Closed using lemmas or earlier targets
56 target theorems	8	2	**46**
55 helper lemmas	12	20	23

For almost every non-trivial target the planner followed one pattern: (1) avoid unfolding the implementation, instead unfold only the spec and the `canonical` record, down to `coalesce (sortByLo ivs)`; (2) split the claim along the implementation's pipeline: a fact about `insertByLo`, lifted to `sortByLo`, combined with a fact about `coalesce` that assumes a sortedness certificate (`Pairwise (·.lo ≤ ·.lo)`); (3) prove each layer by the induction matching its recursion: structural induction for `insertByLo` and `sortByLo`, the functional induction principle `coalesce.induct` for `coalesce`; (4) close the target in a few lines by rewriting with the layer lemmas. The plan for `prove_merge_coverage` states it directly: "the existing coverage lemmas exactly factor merge into sorting and coalescing; do not unfold either implementation."

D.2.4 Key proof episodes, with Lean code

The two central targets: coverage and canonical form. Almost every `mergeOverlaps` target depends on `prove_merge_coverage` (used directly by 15 declarations) or `prove_merge_disjoint_strict` (used directly by 11); both reduce to lemmas about the implementation's layers:

```
theorem prove_merge_coverage : spec_merge_coverage canonical := by
  intro ivs x
  change covered x (Intervaltree.coalesce (Intervaltree.sortByLo ivs)) = covered x ivs
  rw [coalesce_coverage_of_pairwise_lo (Intervaltree.sortByLo ivs) x
    (sortByLo_pairwise_lo ivs)]
  exact sortByLo_coverage ivs x

theorem prove_merge_disjoint_strict : spec_merge_disjoint_strict canonical := by
  intro ivs
  change disjointSortedNonempty
```

```
    (Intervaltree.coalesce (Intervaltree.sortByLo ivs))
  exact coalesce_disjointSortedNonempty (Intervaltree.sortByLo ivs)
    (sortByLo_output_nonempty ivs) (sortByLo_pairwise_lo ivs)
```

The sorting layer is handled by structural induction; `sortByLo_coverage` lifts the one-step fact `insertByLo_coverage` through the fold:

```
lemma sortByLo_coverage (ivs : List Iv) (x : Int) :
    covered x (Intervaltree.sortByLo ivs) = covered x ivs := by
  induction ivs with
  | nil =>
    rfl
  | cons iv ivs ih =>
    calc
      covered x (Intervaltree.sortByLo (iv :: ivs)) =
          covered x (Intervaltree.insertByLo iv (Intervaltree.sortByLo ivs)) := rfl
      _ = covered x (iv :: Intervaltree.sortByLo ivs) :=
        insertByLo_coverage iv (Intervaltree.sortByLo ivs) x
      _ = covered x (iv :: ivs) := by
        simpa only [covered, List.any_cons] using
          congrArg (fun z : Bool => decide (iv.lo ≤ x ∧ x < iv.hi) || z) ih
```

The coalescing layer is where the difficulty lies. The first attempt on `prove_merge_coverage` let automation unfold `coalesce` and case-split; the Recovery Reviewer rewound it, recording: "the current branches have unfolded `Intervaltree.coalesce` repeatedly into arbitrarily deep recursive matches, but carry no induction hypothesis or sortedness invariant; closing them would require reconstructing a global induction invariant from the expanded terms." The Lemma Planner then proposed `coalesce_coverage_of_pairwise_lo`, planned as a functional induction over `coalesce` in which x and the sortedness certificate are *reverted*, so the induction hypothesis applies to the merged list:

```
lemma coalesce_coverage_of_pairwise_lo (ivs : List Iv) (x : Int)
    (hsorted : ivs.Pairwise (fun a b => a.lo ≤ b.lo)) :
    covered x (Intervaltree.coalesce ivs) = covered x ivs := by
  revert x hsorted
  induction ivs using Intervaltree.coalesce.induct with
  | case1 =>
    intro x _
    rw [Intervaltree.coalesce.eq_1]
  | case2 a =>
    intro x _
    rw [Intervaltree.coalesce.eq_2]
  | case3 a b rest hab ih =>                          -- merge branch: b.lo ≤ a.hi
    intro x hs
    rcases (List.pairwise_cons.mp hs) with ⟨ha, htail⟩
    have hablo : a.lo ≤ b.lo := ha b (by simp)
    by_cases hhi : a.hi < b.hi
    · have hsMerged :
          ({ lo := a.lo, hi := b.hi } :: rest).Pairwise
            (fun u v => u.lo ≤ v.lo) := by
              rw [List.pairwise_cons]
              constructor
              · intro r hr
                exact ha r (by simp [hr])
              · exact List.Pairwise.of_cons htail
      rw [Intervaltree.coalesce.eq_3, if_pos hab]
      simp only [dif_pos hhi, if_pos hhi] at ih ⊢
      rw [ih x hsMerged]
      by_cases hal : a.lo ≤ x <;>
        by_cases hax : x < a.hi <;>
        by_cases hbl : b.lo ≤ x <;>
        by_cases hbx : x < b.hi <;>
        simp [covered, hal, hax, hbl, hbx] <;> omega
    · have hsMerged :
```

```
          ({ lo := a.lo, hi := a.hi } :: rest).Pairwise
            (fun u v => u.lo ≤ v.lo) := by
              rw [List.pairwise_cons]
              constructor
              · intro r hr
                exact ha r (by simp [hr])
              · exact List.Pairwise.of_cons htail
      rw [Intervaltree.coalesce.eq_3, if_pos hab]
      simp only [dif_neg hhi, if_neg hhi] at ih ⊢
      rw [ih x hsMerged]
      by_cases hal : a.lo ≤ x <;>
        by_cases hax : x < a.hi <;>
        by_cases hbl : b.lo ≤ x <;>
        by_cases hbx : x < b.hi <;>
        simp [covered, hal, hax, hbl, hbx] <;> omega
  | case4 a b rest hab ih =>                        -- gap branch: a.hi < b.lo
    intro x hs
    rcases (List.pairwise_cons.mp hs) with ⟨ha, htail⟩
    rw [Intervaltree.coalesce.eq_3, if_neg hab]
    have htailCov := ih x htail
    simpa [covered] using
      congrArg (fun q : Bool =>
        (decide (a.lo ≤ x ∧ x < a.hi) : Bool) || q) htailCov
```

The counterexample check rejects a false lemma. While planning `prove_merge_disjoint_strict`, the Lemma Planner proposed that every interval produced by `insertByLo` is nonempty:

```
lemma insertByLo_output_nonempty (iv : Iv) (ys : List Iv) :
    ∀ o ∈ Intervaltree.insertByLo iv ys, o.lo < o.hi
```

This is false: when `iv` is empty, `insertByLo` returns `ys` unchanged, so an empty interval already in `ys` survives. The counterexample check rejected the lemma before any proof effort was spent on it:

```
Counterexample 1:
iv := { lo := 0, hi := 0 }, ys := [{ lo := 0, hi := 0 }]
issue: refuted deductively; the statement is not evaluable (no cocheck verdict)
note: verified by tactic ladder, not by cocheck_refute
```

In the same iteration the Lemma Planner restated it with the missing invariant as a hypothesis which was exactly the form the fold in `sortByLo` needs, since the list built so far is known to contain only nonempty intervals:

```
lemma insertByLo_output_nonempty (iv : Iv) (ys : List Iv)
    (hys : ∀ o ∈ ys, o.lo < o.hi) :
    ∀ o ∈ Intervaltree.insertByLo iv ys, o.lo < o.hi := by
  induction ys with
  | nil =>
    intro o ho
    unfold Intervaltree.insertByLo at ho
    split at ho
    · simp_all
    · simp_all
  | cons y ys ih =>
    have hy : y.lo < y.hi := hys y (by simp)
    have htail : ∀ z ∈ ys, z.lo < z.hi := fun z hz => hys z (by simp [hz])
    intro o ho
    unfold Intervaltree.insertByLo at ho
    split at ho
    · rename_i hp
      split at ho
      · rename_i hle
        simp only [List.mem_cons] at ho
```

```
        rcases ho with rfl | rfl | ho
        · exact hp
        · exact hy
        · exact htail o ho
      · rename_i hle
        simp only [List.mem_cons] at ho
        rcases ho with rfl | ho
        · exact hy
        · exact ih htail o ho
    · rename_i hp
      exact hys o (by simpa using ho)

lemma sortByLo_output_nonempty (ivs : List Iv) :
    ∀ o ∈ Intervaltree.sortByLo ivs, o.lo < o.hi := by
  induction ivs with
  | nil =>
    simp [Intervaltree.sortByLo]
  | cons iv ivs ih =>
    simpa [Intervaltree.sortByLo, List.foldr_cons] using
      (insertByLo_output_nonempty iv (Intervaltree.sortByLo ivs) ih)
```

`sortByLo_output_nonempty` is used directly by 7 declarations.

Canonical form of `coalesce`. `coalesce_disjointSortedNonempty` is the canonical-form half of the core: coalescing a sorted list of nonempty intervals gives a canonical list. The Proof Writer reasoned about a plan which split off a small invariant first, `coalesce_lower_bound` (a lower bound on every input `lo` is preserved in the output), and then planned a four-case functional induction, e.g., "in the nonmerge branch, use `Int.lt_of_not_ge hyx` to obtain `x.hi < y.lo` ... for gap index zero, the index hypothesis makes the coalesced tail nonempty; apply the reusable lemma `coalesce_lower_bound (y :: rest) y.lo` to its first element ... for a successor index, reduce the list indexing with `simp` and invoke IH.2 at the predecessor index":

```
lemma coalesce_disjointSortedNonempty (ivs : List Iv)
    (hnonempty : ∀ iv ∈ ivs, iv.lo < iv.hi)
    (hsorted : ivs.Pairwise (fun a b => a.lo ≤ b.lo)) :
    disjointSortedNonempty (Intervaltree.coalesce ivs) := by
  revert hnonempty hsorted
  induction ivs using Intervaltree.coalesce.induct with
  | case1 =>
    intro hnonempty hsorted
    unfold disjointSortedNonempty
    simp [Intervaltree.coalesce]
  | case2 x =>
    intro hnonempty hsorted
    unfold disjointSortedNonempty
    simp [Intervaltree.coalesce, hnonempty x (by simp)]
  | case3 x y rest hyx ih =>                    -- merge branch
    intro hnonempty hsorted
    have hx : x.lo < x.hi := hnonempty x (by simp)
    have hy : y.lo < y.hi := hnonempty y (by simp)
    have hp := List.pairwise_cons.mp hsorted
    have hxy : x.lo ≤ y.lo := hp.1 y (by simp)
    have hp' := List.pairwise_cons.mp hp.2
    let u : Iv := { lo := x.lo, hi := if x.hi < y.hi then y.hi else x.hi }
    have hu : u.lo < u.hi := by
      dsimp [u]
      split <;> omega
    have hnu : ∀ iv ∈ u :: rest, iv.lo < iv.hi := by
      intro iv hiv
      rcases List.mem_cons.mp hiv with rfl | hiv
      · exact hu
      · exact hnonempty iv (by simp [hiv])
    have hpu : List.Pairwise (fun a b : Iv => a.lo ≤ b.lo) (u :: rest) := by
      apply List.pairwise_cons.mpr
```

```
      constructor
      · intro z hz
        dsimp [u]
        exact Int.le_trans hxy (hp'.1 z hz)
      · exact hp'.2
    have hres := ih hnu hpu
    simpa [Intervaltree.coalesce, hyx, u] using hres
  | case4 x y rest hyx ih =>                        -- gap branch
    intro hnonempty hsorted
    have hx : x.lo < x.hi := hnonempty x (by simp)
    have hp := List.pairwise_cons.mp hsorted
    have hgap : x.hi < y.lo := Int.lt_of_not_ge hyx
    have htailnon : ∀ iv ∈ y :: rest, iv.lo < iv.hi := by
      intro iv hiv
      exact hnonempty iv (by simp [hiv])
    have hres := ih htailnon hp.2
    have hlower : ∀ iv ∈ y :: rest, y.lo ≤ iv.lo := by
      intro iv hiv
      rcases List.mem_cons.mp hiv with rfl | hiv
      · exact Int.le_refl _
      · exact (List.pairwise_cons.mp hp.2).1 iv hiv
    simp only [Intervaltree.coalesce, hyx, ↓reduceIte]
    unfold disjointSortedNonempty at hres ⊢
    constructor
    · intro iv hiv
      rcases List.mem_cons.mp hiv with rfl | hiv
      · exact hx
      · exact hres.1 iv hiv
    · intro i hi
      cases i with
      | zero =>
        simp only [List.getElem_cons_zero, List.getElem_cons_succ]
        simp only [List.length_cons] at hi
        have hzero : 0 < (Intervaltree.coalesce (y :: rest)).length := by
          omega
        have hzlo := coalesce_lower_bound (y :: rest) y.lo hlower
          (Intervaltree.coalesce (y :: rest))[0] (List.getElem_mem hzero)
        exact lt_of_lt_of_le hgap hzlo
      | succ i =>
        simp only [List.length_cons] at hi
        simpa using hres.2 i (by omega)
```

Uniqueness of canonical form: the extensionality lemma. The deepest mathematical step is independent of the implementation: two canonical interval lists that cover the same points are equal. The Lemma Planner stated it as a standalone lemma in iteration 13. With it, the uniqueness, determination, and permutation-invariance targets follow in a few lines each. The proof plan in Proof Writer is an induction on the first list, generalizing the second: (1) the heads' lower endpoints agree, because the smallest covered point of each canonical list is its head's `lo`; (2) the upper endpoints agree, because the point at a canonical head's `hi` is *uncovered*, which rules out either head extending past the other; (3) the tails cover the same points, so the induction hypothesis applies.

```
lemma disjointSortedNonempty_ext_of_coverage (c d : List Iv)
    (hc : disjointSortedNonempty c) (hd : disjointSortedNonempty d)
    (hcov : ∀ x : Int, covered x c = covered x d) :
    c = d := by
  induction c generalizing d with
  | nil =>
    cases d with
    | nil =>
      rfl
    | cons b ds =>
      exfalso
      have hb : b.lo < b.hi := hd.1 b (by simp)
```

```
      have hdtrue : covered b.lo (b :: ds) = true :=
        (covered_eq_true_iff b.lo (b :: ds)).2 ⟨b, by simp, le_rfl, hb⟩
      have heq := hcov b.lo
      simp [covered] at heq
      omega
  | cons a cs ih =>
    cases d with
    | nil =>
      exfalso
      have ha : a.lo < a.hi := hc.1 a (by simp)
      have hctrue : covered a.lo (a :: cs) = true :=
        (covered_eq_true_iff a.lo (a :: cs)).2 ⟨a, by simp, le_rfl, ha⟩
      have heq := hcov a.lo
      simp [covered] at heq
      omega
    | cons b ds =>
      have ha := hc.1 a (by simp)
      have hb := hd.1 b (by simp)
      have hcgap := disjointSortedNonempty_pairwise_gap (a :: cs) hc
      have hdgap := disjointSortedNonempty_pairwise_gap (b :: ds) hd
      -- every member's lo is bounded below by the head's lo
      have hclo : ∀ v ∈ a :: cs, a.lo ≤ v.lo := by
        intro v hv
        rcases List.mem_cons.mp hv with rfl | hv
        · exact le_rfl
        · have hgapv := (List.pairwise_cons.mp hcgap).1 v hv
          omega
      have hdlo : ∀ v ∈ b :: ds, b.lo ≤ v.lo := by
        intro v hv
        rcases List.mem_cons.mp hv with rfl | hv
        · exact le_rfl
        · have hgapv := (List.pairwise_cons.mp hdgap).1 v hv
          omega
      -- (1) the heads have the same lower endpoint
      have hba : b.lo ≤ a.lo := by
        have hatrue : covered a.lo (a :: cs) = true :=
          (covered_eq_true_iff a.lo (a :: cs)).2 ⟨a, by simp, le_rfl, ha⟩
        have hbtrue : covered a.lo (b :: ds) = true := by
          rw [← hcov a.lo]
          exact hatrue
        obtain ⟨v, hv, hvlo, hvhi⟩ := (covered_eq_true_iff a.lo (b :: ds)).1 hbtrue
        have hlow := hdlo v hv
        omega
      have hab : a.lo ≤ b.lo := by
        have hbtrue : covered b.lo (b :: ds) = true :=
          (covered_eq_true_iff b.lo (b :: ds)).2 ⟨b, by simp, le_rfl, hb⟩
        have hatrue : covered b.lo (a :: cs) = true := by
          rw [hcov b.lo]
          exact hbtrue
        obtain ⟨v, hv, hvlo, hvhi⟩ := (covered_eq_true_iff b.lo (a :: cs)).1 hatrue
        have hlow := hclo v hv
        omega
      have hlo : a.lo = b.lo := le_antisymm hab hba
      -- (2) a canonical head's upper endpoint is uncovered
      have hcaHi : covered a.hi (a :: cs) = false := by
        apply Bool.eq_false_of_not_eq_true
        intro ht
        obtain ⟨v, hv, hvlo, hvhi⟩ := (covered_eq_true_iff a.hi (a :: cs)).1 ht
        rcases List.mem_cons.mp hv with rfl | hv
        · omega
        · have hgapv := (List.pairwise_cons.mp hcgap).1 v hv
          omega
      have hdbHi : covered b.hi (b :: ds) = false := by
        apply Bool.eq_false_of_not_eq_true
        intro ht
```

```
      obtain ⟨v, hv, hvlo, hvhi⟩ := (covered_eq_true_iff b.hi (b :: ds)).1 ht
      rcases List.mem_cons.mp hv with rfl | hv
      · omega
      · have hgapv := (List.pairwise_cons.mp hdgap).1 v hv
        omega
    have hbia : b.hi ≤ a.hi := by
      by_contra h
      have hlt : a.hi < b.hi := lt_of_not_ge h
      have hbt : covered a.hi (b :: ds) = true :=
        (covered_eq_true_iff a.hi (b :: ds)).2 ⟨b, by simp, by omega, hlt⟩
      have hct : covered a.hi (a :: cs) = true := by
        rw [hcov a.hi]
        exact hbt
      rw [hcaHi] at hct
      simp at hct
    have haib : a.hi ≤ b.hi := by
      by_contra h
      have hlt : b.hi < a.hi := lt_of_not_ge h
      have hat : covered b.hi (a :: cs) = true :=
        (covered_eq_true_iff b.hi (a :: cs)).2 ⟨a, by simp, by omega, hlt⟩
      have hdt : covered b.hi (b :: ds) = true := by
        rw [← hcov b.hi]
        exact hat
      rw [hdbHi] at hdt
      simp at hdt
    have hhi : a.hi = b.hi := le_antisymm haib hbia
    have habeq : a = b := by
      cases a
      cases b
      simp_all
    subst b
    -- (3) the tails cover the same points; apply the IH
    have hcTail : disjointSortedNonempty cs := disjointSortedNonempty_tail a cs hc
    have hdTail : disjointSortedNonempty ds := disjointSortedNonempty_tail a ds hd
    have htailcov : ∀ x : Int, covered x cs = covered x ds := by
      intro x
      by_cases hx : a.lo ≤ x ∧ x < a.hi
      · have hcsfalse : covered x cs = false := by
          apply Bool.eq_false_of_not_eq_true
          intro ht
          obtain ⟨v, hv, hvlo, hvhi⟩ := (covered_eq_true_iff x cs).1 ht
          have hgapv := (List.pairwise_cons.mp hcgap).1 v hv
          omega
        have hdsfalse : covered x ds = false := by
          apply Bool.eq_false_of_not_eq_true
          intro ht
          obtain ⟨v, hv, hvlo, hvhi⟩ := (covered_eq_true_iff x ds).1 ht
          have hgapv := (List.pairwise_cons.mp hdgap).1 v hv
          omega
        rw [hcsfalse, hdsfalse]
      · have hhead : decide (a.lo ≤ x ∧ x < a.hi) = false :=
          decide_eq_false_iff_not.mpr hx
        have hhcov := hcov x
        change (decide (a.lo ≤ x ∧ x < a.hi) || covered x cs) =
          (decide (a.lo ≤ x ∧ x < a.hi) || covered x ds) at hhcov
        rw [hhead, Bool.false_or, Bool.false_or] at hhcov
        exact hhcov
    have htail := ih ds hcTail hdTail htailcov
    exact congrArg (fun t : List Iv => a :: t) htail
```

Combined with the two central targets, extensionality gives several targets almost for free, and the agent proved permutation and reversal invariance *semantically*: a permutation preserves coverage (*i.e.*, `covered_perm`, closed by Deterministic Automation), and extensionality turns equal coverage into equal canonical outputs, avoiding any reasoning about how `insertByLo` handles ties:

```
theorem prove_merge_coverage_determines : spec_merge_coverage_determines canonical :=
    by
  unfold spec_merge_coverage_determines
  intro a b hab
  apply disjointSortedNonempty_ext_of_coverage
  · exact prove_merge_disjoint_strict a
  · exact prove_merge_disjoint_strict b
  · intro x
    calc
      covered x (canonical.intervaltree.mergeOverlaps a) = covered x a :=
        prove_merge_coverage a x
      _ = covered x b := hab x
      _ = covered x (canonical.intervaltree.mergeOverlaps b) :=
        (prove_merge_coverage b x).symm

lemma mergeOverlaps_perm (xs ys : List Iv) (hperm : xs.Perm ys) :
    Intervaltree.mergeOverlaps xs = Intervaltree.mergeOverlaps ys := by
  apply disjointSortedNonempty_ext_of_coverage
  · simpa [spec_merge_disjoint_strict, canonical] using
      (prove_merge_disjoint_strict xs)
  · simpa [spec_merge_disjoint_strict, canonical] using
      (prove_merge_disjoint_strict ys)
  · intro x
    calc
      covered x (Intervaltree.mergeOverlaps xs) = covered x xs := by
        simpa [spec_merge_coverage, canonical] using (prove_merge_coverage xs x)
      _ = covered x ys := covered_perm x xs ys hperm
      _ = covered x (Intervaltree.mergeOverlaps ys) := by
        symm
        simpa [spec_merge_coverage, canonical] using (prove_merge_coverage ys x)

theorem prove_merge_reverse_eq : spec_merge_reverse_eq canonical := by
  unfold spec_merge_reverse_eq
  unfold canonical
  unfold Intervaltree.overlaps Intervaltree.mergeOverlaps Intervaltree.chop
  try unfold Intervaltree.coalesce
  try unfold Intervaltree.chopOne
  unfold Intervaltree.sortByLo
  intro ivs
  rw [← Intervaltree.coalesce.eq_def, ← Intervaltree.coalesce.eq_def]
  exact mergeOverlaps_perm ivs ivs.reverse (List.reverse_perm ivs).symm
```

The deepest chain: `chop` preserves merge fixpoints. `prove_chop_canonical_merge_fixpoint` sits at dependency depth 6, the deepest point in the proof; it combines both halves of the library (chopping a canonical list keeps it canonical; merging a canonical list is the identity): Figure 7 shows its dependency graph.

Here the agent kept the mathematical statement and the syntactic shape of the goal separate. `chop_merge_fixpoint_public` is stated over the public definitions and proved in two lines from two verified results:

```
lemma chop_merge_fixpoint_public (b e : Int) (ivs : List Iv)
    (hbe : b < e)
    (hnonempty : ∀ iv ∈ ivs, iv.lo < iv.hi)
    (hconsecutive : ∀ (i : Nat) (h : i + 1 < ivs.length),
      (ivs[i]).hi < (ivs[i + 1]).lo) :
    Intervaltree.mergeOverlaps (Intervaltree.chop b e ivs) =
      Intervaltree.chop b e ivs := by
  apply mergeOverlaps_eq_self_of_disjointSortedNonempty
  exact prove_chop_canonical_preserves_canonical b e ivs hbe ⟨hnonempty, hconsecutive⟩
```

The target's goal, however, had already been unfolded by the automation prefix into an explicit `match`/`flatMap` term, and an earlier attempt to `change` the goal back to the public form had

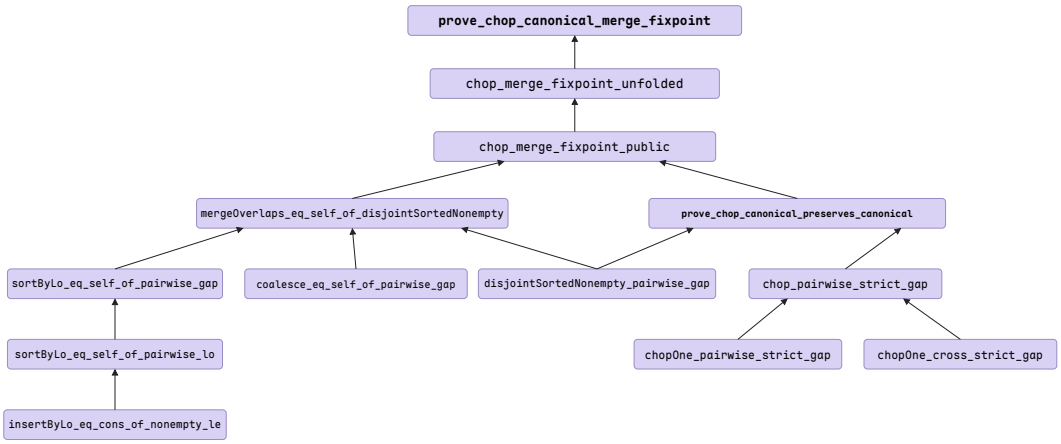


Figure 7: **Lemma-dependency graph of `prove_chop_canonical_merge_fixpoint`**, the deepest target in the Vero `intervaltree` run; edges point from a helper lemma to the declaration whose proof consumes it. Below the target, `chop_merge_fixpoint_unfolded` bridges to the unfolded goal shape left by the automation prefix and `chop_merge_fixpoint_public` restates it over the public definitions; the proof then splits into the two halves of the library, merging a canonical list is the identity (left) and chopping preserves canonicity (right). Two forms of reuse are visible. `prove_chop_canonical_preserves_canonical` (bold) is another benchmark target that this proof consumes as a lemma, and `disjointSortedNonempty_pairwise_gap` is a single node feeding both halves rather than a fact re-derived in each.

failed (recorded in the plan as "the failed earlier approach"). The Lemma Planner therefore proposed a bridge lemma, `chop_merge_fixpoint_unfolded`, stated in exactly that unfolded form and proved by unfolding the public lemma's statement the same way, which reduces the target to a single `exact`:

```
theorem prove_chop_canonical_merge_fixpoint : spec_chop_canonical_merge_fixpoint
    canonical := by
  unfold spec_chop_canonical_merge_fixpoint
  unfold disjointSortedNonempty canonical
  unfold Intervaltree.overlaps Intervaltree.mergeOverlaps Intervaltree.chop
  try unfold Intervaltree.coalesce
  try unfold Intervaltree.chopOne
  unfold Intervaltree.sortByLo
  intro b e ivs a a_1
  simp_all
  exact chop_merge_fixpoint_unfolded b e ivs a a_1.1 a_1.2
```

Backtracking to an induction point. All 52 backtracks in the run follow one pattern: automation or an eager script unfolds a recursive definition several levels deep and loses the induction structure; the router and Recovery Reviewer recognize this, rewinds to a node where the recursive object is still abstract (*i.e.*, the nearest safe point) and proposes an induction there or a lemma to apply. Two representative recorded reasons: for `insertByLo_length`, "the repeated `unfold`/`split` path has exposed several fixed recursive layers of `insertByLo` but introduced no induction hypothesis for the arbitrary remaining tail `ys`; closing the displayed goal would require proving the same length property for `ys`, whereas node 66234 permits structural induction on `ys`, yielding exactly that induction hypothesis"; and for `prove_merge_coverage`, "at node 65680 both `covered` and `sortByLo` have already been unfolded, so the available coverage lemmas no longer syntactically match … at node 65541, unfolding only the specification and the canonical record preserves it sufficiently to apply the three existing semantic lemmas directly." Because every backtrack keeps a record of why the abandoned path failed, later planning passes over the same region do not repeat it.

D.2.5 DEPENDENCY RELATIONSHIPS

The 111 lemmas/theorems form the dependency graph of maximum depth 6, layered from single-step facts up through the pipeline:

```
Layer 0: facts about one interval / one insertion step
         insertByLo_coverage, insertByLo_output_nonempty, mem_insertByLo,
         chopOne_coverage, covered_eq_true_iff, covered_perm,
         disjointSortedNonempty_tail, disjointSortedNonempty_pairwise_gap, ...
Layer 1: lifted through the fold / flatMap; functional induction on coalesce
         sortByLo_coverage, sortByLo_output_nonempty,
         coalesce_coverage_of_pairwise_lo, coalesce_disjointSortedNonempty,
         chop_pairwise_strict_gap, disjointSortedNonempty_ext_of_coverage
Layer 2: sortedness certificate
         sortByLo_pairwise_lo, mem_sortByLo_iff, sortByLo_eq_self_of_pairwise_gap
Layer 3: the central targets
         prove_merge_coverage, prove_merge_disjoint_strict, prove_merge_is_canonical,
         mergeOverlaps_eq_self_of_disjointSortedNonempty
Layer 4: consequences of coverage + canonicality (+ extensionality)
         prove_merge_canonical_unique, prove_merge_coverage_determines,
         mergeOverlaps_perm, ...
Layer 5: consequences of determination / permutation invariance
         prove_merge_reverse_eq, prove_merge_perm_eq_any, prove_merge_chop_merge, ...
Layer 6: prove_chop_canonical_merge_fixpoint
```

Table 14: **The most reused declarations.** A handful of layer lemmas (sortedness, nonemptiness, the strict-gap conversion) feed more than half of the 56 targets, and seven proved targets are themselves reused as lemmas—this is why decomposing into lemmas scales here: each hard induction is done once and then reused.

Declaration	Kind	Direct users	Targets depending on it (transitively)
`prove_merge_coverage`	target	15	18
`disjointSortedNonempty_pairwise_gap`	lemma	11	24
`prove_merge_disjoint_strict`	target	11	18
`sortByLo_pairwise_lo`	lemma	7	31
`sortByLo_output_nonempty`	lemma	7	28
`prove_chop_pointwise`	target	5	6
`prove_merge_coverage_determines`	target	5	5
`coalesce_disjointSortedNonempty`	lemma	4	23
`disjointSortedNonempty_ext_of_coverage`	lemma	4	10
`covered_eq_true_iff`	lemma	4	14
`disjointSortedNonempty_tail`	lemma	4	14

Of the 56 targets, 10 close at depth 0 (no helper lemma), and the rest sit at depths 1–6 with direct dependency lists of one to four declarations each.

D.2.6 WHAT MADE ALL 56 PROVABLE

Five mechanisms, each visible in the trace, account for this outcome. As in the CLEVER study, each is a design component of Section 3.2 doing its intended job, yet here the components carry *provability* rather than cost. *(1) Decomposition along the implementation—the Lemma Planner writing into the dependency DAG.* Every `mergeOverlaps` fact was split into an `insertByLo` step, a `sortByLo` lift, and a `coalesce` fact taking a sortedness certificate. Because each layer lemma is a DAG node served to every consumer, the certificate (`sortByLo_pairwise_lo`) and the nonemptiness invariant (`sortByLo_output_nonempty`) were proved once and reused by 31 and 28 of the 56 targets, respectively. *(2) The right induction for each recursion scheme—the router's planning.* The agent built for structural obstructions chose structural induction for insertion and the fold and the functional induction principle `coalesce.induct` for the well-founded coalescing, reverting or generalizing hypotheses so the induction hypothesis applies to the modified recursive argument; its plans then guided the Proof Writer's scripts. *(3) Semantic lemmas instead of syntactic unfolding—structure-before-repetition.* Where redrawing on an unfolded goal could only fail again, the router escalated to statement-level structure: the extensionality lemma turns uniqueness, determination, and permutation-invariance targets into short applications of the two central results, with

no reasoning about the sorting algorithm. *(4) Proved targets reused as lemmas—the DAG makes targets first-class nodes.* Nothing in the graph distinguishes a stable target from a stable helper, so seven proved targets serve as dependencies of later targets and the proof reaches depth 6 without duplicating work (*i.e.*, cross-target reuse that no per-theorem pool, sketch, or blueprint expresses). *(5) Checking and backtracking—the counterexample check and the Recovery Reviewer.* The check refuted a false candidate lemma before any proof effort was staked on it, and the Reviewer accepted 52 backtracks to the nearest safe node, each with a recorded reason entering the failure memory, so later planning passes over the same region never repeated a dead end.

E PROMPT TEMPLATES

This section lists the prompt templates of the agents shown in Figure 3 and described in Section 3.2.

System Prompt

````
You are a Lean 4 proof analyst inside an automated theorem-proving agent.

The agent explores a proof tree: each node is one tactic application, and each open (
unproved) node carries a `sorry` placeholder standing for the goal that remains to be
 proved at that point.

The overall task may involve multiple theorems and lemmas with dependency
relationships among them. Each lemma / theorem is introduced by an annotation doc
string in the following format:

```lean
/-
@name: String
@namespace: String
@description: String
@proved: Boolean
@depends_on: List
@lib_depends_on: List
@used_in: List
-/
lemma one_lemma (param: String): spec param := by
  sorry
```

The annotation fields:

- `@name`: The name of the lemma / theorem.
- `@namespace`: The namespace of the lemma / theorem; empty means the global
namespace.
- `@description`: A natural language description of what the lemma / theorem states,
typically in one or two sentences.
- `@proved`: Whether the lemma / theorem has been proved.
- `@depends_on`: The proposed lemmas / theorems (library lemmas excluded) that the
proof may depend on (actually depends on when proved; otherwise the dependency is
analyzed probably).
- `@lib_depends_on`: The library lemmas / theorems (Lean Core, Std, Mathlib) that the
 lemma / theorem depends on (actually depends on when proved, otherwise the
dependency is analyzed probably).
- `@used_in`: The theorems / lemmas that use this lemma / theorem.

A lemma / theorem may additionally carry a **header prefix**: one or more `... in`
modifier lines written right before its declaration and scoped to it alone, e.g.

```lean
open private Foo.bar.loop from Init.Data.Foo.Basic in
set_option maxHeartbeats 400000 in
lemma one_lemma (param: String): spec param := by
  sorry
```

The prefix is rendered by the harness before that one declaration in every check and
in the final proof file; it is not inherited by the lemmas that use it. Its main use
is `open private <name> from <Module> in`: Lean core hides many worker functions as
PRIVATE declarations | they appear in goals with a dagger (`Foo.bar.loop†`), `#check`
answers `Unknown constant`, and their names cannot be written in source | so a proof
about a definition that unfolds to such a worker can only proceed after the header
prefix makes the name resolvable (then `rw [<name>]`, `rw [<name>, dif_pos h]`, `
````

```
fun_induction <name> ...`, `<name>.eq_def` work inside that lemma). The harness tells
 you the exact line whenever it recognizes a private name in a goal or in an "Unknown
 constant" answer. Other allowed prefix lines: `open <Namespace> in`, `set_option
maxHeartbeats <n> in`, `attribute [local simp] <lemmas> in`.

The Lean context is the following, including imported libraries, definitions, and
possible helper lemmas / theorems:

'''lean
{lean_context}
'''

All theorems and lemmas being proved live under the following working namespace (
empty means the global namespace):

'''lean
{namespace}
'''

Plan first, then call tools to carry out the tasks given in the user prompt.

When the tasks are complete, do not output a long summary | output any final artifact
 the user prompt explicitly asks for, then a single sentence stating that you are
done.
```

Agentic Router Prompt

```
You are given a group of proof tree nodes.

Do the following tasks in sequence:

1. **Information gathering**: For the nodes in the group, get their tactic states and
 their previous proof snapshots; also get all the lemmas proposed so far in the lemma
 dependency graph (library lemmas excluded), whose annotation doc strings carry the
dependency information. Note that the proposed lemmas are not all correct: only those
 marked `@proved: true` are verified, while the rest are unproved proposals that may
turn out to be wrong | judge each lemma's statement yourself and select which ones to
 rely on.
2. **Rule-based Routing:** based on the information you gathered, route each nodes
into different action specialist based on the routing rules {rules} with its
configuration including the possible reasoning effort and working budget:
   - Deterministic Automation
   - Lemma Planner
   - Proof Writer
   - Recovery Reviewer
```

Lemma Planner Prompt

```
Based on existing proposed lemmas, decide for each goal whether to prove it directly
or to propose new general lemmas to support it.

  - **Prefer reusable lemmas**: ones that serve several nodes | in this group,
elsewhere in the task, or in goals that will recur later as the same definitions are
unfolded. State a lemma at the generality of the definitions involved, not at the
shape of one goal (quantify over the arguments, drop unneeded hypotheses, split a
combined fact into its independent parts). Check the pool first: reuse or generalize
an existing lemma rather than adding a near-duplicate.
  - **State the side conditions, then test the corners in your head**: a lemma about
indices, ranges, slices or windows is false without its well-formedness hypotheses |
spell out every needed `0 ≤ i`, `lo ≤ hi`, `i < xs.length`, non-emptiness or length
relation EXPLICITLY. Before adding, mentally instantiate the statement at the corners
: the empty structure, `lo = hi` and `lo > hi`, an index at / past the length, a
negative integer under a `toNat` coercion, equal or swapped indices. A statement that
 fails any of these will be refuted mechanically and its proof budget wasted | fix
the statement first.
  - Proposed lemmas must be provable and must do real work | do not trivially restate
 a tactic state as a lemma.
  - Each proposed lemma must be a self-contained top-level `lemma` declaration, not
an internal step inside another proof.
```

```
  - Do not write a proof body for a proposed lemma: give only its declaration header
ending in `:= by` with an indented `sorry` placeholder as the body | each proposed
lemma will be proved later as its own proof task.
  - Annotate every proposed lemma with a complete annotation doc string: `@name`, `
@namespace`, `@description`, `@proved: false`, `@depends_on` (the proposed lemmas it
is expected to depend on), `@lib_depends_on` (the library lemmas it is expected to
depend on), and `@used_in` (the theorems / lemmas that will use it).
  - **Analyze the dependency graph before adding**: the annotations you write become
edges of the lemma dependency graph, and only what is declared there is available to
the prover. So (a) `@used_in` must name every theorem / lemma whose goal (in this
group) the new lemma is meant to serve | otherwise the prover of that goal cannot use
 it; (b) `@depends_on` must list every existing pool lemma the new lemma's proof will
 need; (c) a lemma must never be self-referential or circular: it must not appear in
its own `@depends_on`, and none of the lemmas it depends on (directly or recursively
through their `@depends_on`) may be one of the lemmas / theorems in its `@used_in` |
check the existing annotations before you write the new ones.
  - You can retrieve similar lemmas from the proof databases for reference.
  - Add each new lemma to the lemma pool, one at a time.
  -  **Helper definitions**: when a goal becomes much easier to state or prove once
an auxiliary function exists, add that function as a helper definition (a `def` with
its complete body | never `sorry`, since a definition is elaborated, not proved)
before the lemmas that use it. Use this sparingly: it changes the Lean context every
later step is checked against.
```

Proof Writer Prompt

```
Propose Lean 4 code that solves the goal, given the current tactic state and the
natural language plan below.

IMPORTANT:

- Propose ONLY the tactic lines to run at the current goal. Your code is inserted
into a proof that is ALREADY OPEN, at the `sorry` standing for this goal | so it must
 not be a standalone declaration. Never wrap it in `theorem`/`lemma`/`def`/`example`,
 never restate the goal or re-introduce the hypotheses (they are already in scope, as
 shown in the tactic state), and never emit `import` lines. Doing so closes the
surrounding proof and the check fails before your tactics run.
- You may need to propose a step-by-step natural language plan for proving a goal if
needed. And then you may follow the natural language plan.
- Follow Lean 4 indentation rules: indent nested tactics by 2 spaces relative to
their parent, e.g. the tactics under a case arm (`| succ n ih =>`), a focus bullet (`·
`), or a `by` sub-proof. Indentation determines block structure, so a wrongly
indented line changes which goal a tactic applies to.
- Fill in concrete terms and names from the tactic state when an action requires them
.
- You can inspect the definitions and lemmas you intend to use via the Lean execution
 engine.
- **Library lemma names | never guess.** Use ONLY library lemma names you have
confirmed exist (via `#check`) or that came back from the retrieval tool. When you
need a library fact but are unsure of its exact name, or the Lean checker reports `
Unknown constant` / `unknown identifier` for a name you used, call the retrieval tool
 with a description or the statement you need (e.g. `(!b) = true ↔ b = false`, or the
 misspelled name itself) | it returns the closest existing declarations with their
statements. Do NOT re-`#check` a name that Lean has already reported as unknown, and
do not retry a failed name with small spelling variations: retrieve, then use a
returned name.
- **Search tactics are not proof steps.** `apply?`, `exact?`, `rw?`, `simp?`, `hint`
may only be used to DISCOVER a lemma; never leave them in submitted code. `apply?` in
 particular admits the goal with `sorry` when it cannot close it | the checker now
rejects such code as "ADMITTED, not proved". Replace a suggestion with the concrete
lemma it names, re-verify, then submit.
- Verify the proposed code with the Lean checker; if it fails, fix the errors and
verify again until the code passes.
- **Submission**: verifying is not answering. Once a candidate passes, submit that
candidate as your final answer. Submitting only records the code | it does not check
it | so submit exactly once, and only code you have already verified. Then stop and
reply with exactly `Code submitted.` | no summary, no restatement of the plan or the
code.
```

Recovery Reviewer Prompt

```
Judge, for each node, whether it is on a viable proving path.

  - If every node is viable, skip.
  - Backtrack a node only when analysis shows its goal is mathematically **unprovable
**, or is **clearly much harder** to prove than what an earlier node on its path
allows. Backtracking is costly: it throws away every tactic tried below the node you
rewind to.
  - To backtrack, rewind ONE node to ONE previous node on its own path, giving the `
reason` (`unprovable` or `hard`) and the evidence in `reason_details`: for `
unprovable`, concrete counterexamples that satisfy every hypothesis and falsify the
goal | they are checked, and a rewind whose counterexamples do not refute the goal is
 refused; for `hard`, what makes this goal much harder than the one you rewind to.
  - Prefer the nearest previous node that puts the proof back on a viable path. If it
 is still not viable, backtrack again from there.
  - Rewinding discards everything below the node you rewind to, so pass the other
group nodes in `checking_nodes`: the result tells you which of them were dropped with
 it.
  - **Lemma abandon**: only when a backtrack is needed, additionally consider whether
 the LEMMA itself should be abandoned | when the lemma is mathematically unprovable
or very hard to prove AND its parent lemma / theorem can be proved without it.
Abandoning is more severe than backtracking: the lemma is removed from the task for
good and every proof path that used it (in any lemma / theorem) is discarded | so it
must be taken carefully, only on strong evidence. Like a backtrack it is reviewed (
for `unprovable`, the counterexamples target the lemma's own hypotheses and goal, not
 a node's tactic state), and it takes `checking_nodes` the same way. The top theorem
can never be abandoned.
  - **Failure memory**: a proof snapshot may end with a "Failure memory" section |
the paths and lemmas already given up around that node, with the reviewed reasons.
Treat them as settled: do not retry a rewound path or re-propose an abandoned lemma.
A `[Failure memory: plan not executable]` entry is different in kind: the GOAL is
still open, but the plan quoted there was already handed to the tactic writer and its
 scripts could not close it | propose a different technique or a decomposition that
avoids the step those scripts died on, never the same strategy reworded.
```

F LIMITATIONS

Benchmark coverage. Our evaluation covers five Lean 4 benchmarks, but only two of them, NTP4VC and Vero, are repository-level, so the evidence on repository-scale lemma reuse rests on fewer tasks than the function-level comparison. All benchmarks are in Lean 4 since we don't implement CoCo-Prover on Rocq or Isabelle/HOL.

Proof-only setting. Implementations and specifications are fixed inputs since CoCo-Prover only operates on proofs as a theorem prover in program verification.

Conditional theory. In theory, we do not prove that the implemented LLM router must attain low regret. The analysis fixes the target of the design and explains when orchestration pays off. And our experiment results demonstrate that the implemented agentic router can attain low regret.

Reliance on LLM judgment. Routing, lemma proposal, planning, and recovery all depend on the semantic judgment of the backbone model. The symbolic guards are syntactic and do not judge whether a decision is good.

Wall-clock time and cost accounting. CoCo-Prover is slower than the coding agents in mean wall-clock time under GPT-5.6-Terra and Gemini-3.5-Flash, which we attribute to checking each bounded invocation with Lean before choosing the next one. Our cost model counts only model tokens at official API prices; Lean checking, retrieval, embedding, and other local computation run on our own machine and are not charged. Dollar costs also depend on each provider's caching behavior, which means the baselines like coding agents using one single conversation will take advantage of the caching cost.