Beyond Retry: A Budget-Controlled Ablation of Compiler Feedback for Lean Proof Repair

Anonymous authors
Paper under double-blind review

Abstract

Formal theorem-proving agents frequently feed Lean compiler diagnostics back to language models after failed proof attempts. Yet existing improvements from compiler-guided repair do not establish which part of that feedback is responsible: semantic error information, a binary failure verdict, additional prompt context, or simply another opportunity to generate. We present a budget-controlled evaluation that isolates the informational value of compiler feedback in iterative Lean proof repair. Starting from 219 failed attempts of a public checkpoint (deepseek-ai/DeepSeek-Prover-V1.5-RL) on a version-pinned Lean 4 benchmark (LeanDojo Benchmark 4 (v10)), we compare four feedback conditions: full diagnostics, binary success/failure, no feedback, and length-matched non-diagnostic placebo text. All conditions use identical model checkpoints, theorem splits, decoding seeds, maximum generated tokens, prompt-length caps, and Lean verifier-call budgets; the primary comparisons are full diagnostics against the placebo and against the binary verdict, which differ from it only in the feedback text. At the pre-specified budget of 8 verifier calls and 3 seeds, the paired full-minus-placebo difference in solved rate is +0.9 percentage points (95% interval [-2.4, +4.0]) and the paired full-minus-binary difference is +1.2 ([-2.1, +4.4]); both intervals include zero. The secondary full-minus-none difference, which also changes the state context, is -9.9 ([-14.5, -5.3]): independent resampling from the same initial failure reached accepted proofs more often than any condition that showed the model its previous attempt. For this model and benchmark, the length-matched placebo and the binary verdict recover whatever the repair loop adds, and we find no evidence that diagnostic semantics add to retry and context effects at equal budgets. By checker-observable failure category, no bucket shows a primary difference whose interval excludes zero; bucket slices are descriptive. We release scripts, environment locks, prompts, seeds, raw per-attempt records and complete checker traces; one command regenerates every table and figure.

1 Introduction

Language-model provers for Lean 4 (de Moura & Ullrich, 2021) rarely succeed on the first attempt, and a common remedy is to try again with the compiler's response in the prompt: the failed candidate, the error messages, and often the remaining goals are fed back to the model, which proposes a revised proof. Systems that use compiler feedback at repair time report gains over their non-repairing configurations (Lin et al., 2025; Ospanov et al., 2025); repair from diagnostics is also a supervised training target in its own right (Wang et al., 2026); and provers trained with proof-assistant feedback as a reward are common (Xin et al., 2024b). However, an end-to-end comparison cannot determine whether those gains come from the *semantic content* of the diagnostics. A repair loop differs from its single-shot baseline in at least four ways at once: the model learns that the previous attempt failed; it sees a longer prompt; it sees the previous candidate; and it is simply given another sample. Each of these can raise the success rate on its own. Attributing the improvement to "compiler feedback" therefore names the intervention, not the mechanism.

We ask a narrow causal question. **Under equal Lean verifier-call, generated-token, decoding, proof-attempt, and prompt-length budgets, how much of iterative Lean proof repair comes from the semantic content of compiler diagnostics rather than a binary failure verdict, extra context, or another retry?** We treat the feedback block as the manipulated variable and hold every-

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.

thing else fixed: one public checkpoint, one frozen initial failing attempt for every arm of an episode, one prompt wrapper, one decoding configuration and seed schedule, one input- and output-token cap, one repair-round cap, one Lean toolchain and Mathlib commit, and the same number of verifier calls. Success is defined only by acceptance from the pinned Lean checker.

We compare four conditions. FULL provides the authentic diagnostics and goal state of the previous attempt. BINARY provides only the verdict that it failed. NONE provides no feedback and resamples independently. PLACEBO provides non-diagnostic filler whose token length matches what FULL would have shown, separating prompt length from diagnostic content. The primary comparisons are FULL against PLACEBO and FULL against BINARY, because those arms differ from FULL only in the feedback text. FULL against NONE is a secondary operational baseline: NONE also withholds the previous candidate, so it measures everything the repair loop adds over independent resampling. A shuffled-diagnostic control is specified in the protocol but is not part of this study: it enters only if a pre-specified leakage audit passes (Section 3.5).

The protocol, budgets, primary outcomes, exclusion policy, and analysis were pre-specified before the locked evaluation in a versioned document whose hash is written to every record, and every later change is logged with a rationale. The two possible outcomes have different interpretations: a FULL advantage over PLACEBO and BINARY that survives paired, theorem-level uncertainty would locate the benefit of compiler feedback in its semantics; a null contrast indicates that, for this model and benchmark, the benefit of the feedback loop is attributable to retry and context effects that cheaper controls reproduce. On 219 held-out episodes and 3 seeds at $B = 8$, the result corresponds to the second outcome: the paired FULL $-$PLACEBO and FULL $-$BINARY differences are +0.9 and +1.2 pp with intervals [-2.4, +4.0] and [-2.1, +4.4], while independent resampling (NONE) exceeded FULL by -9.9 pp with interval [-14.5, -5.3] (Section 5).

Contributions.

1. A budget-controlled protocol for isolating the informational value of compiler feedback in proof repair, with an exact verifier-call accounting rule (covering parse and elaboration failures, timeouts, duplicates, cache reads and infrastructure retries), a length-matched placebo, and a frozen-initial-failure episode design that makes every arm start from the same state (Section 3).
2. A matched-budget evaluation of FULL, BINARY, NONE and PLACEBO on 219held-out, version-pinned Lean 4 tasks with one public checkpoint, 3seeds, and paired theorem-level intervals, reported as success-versus-verifier-call curves and as effects by checker-observable failure category (Section 5), together with a descriptive account of candidate repetition that bears on the secondary contrast.
3. A complete, anonymised artifact: environment locks, data manifest, prompts, seeds, raw per-attempt JSONL records with checker logs, and a single command that regenerates every table and figure from those records (Section 7).

Scope. The claims concern one public model, one benchmark, one prompt wrapper and the Lean 4 diagnostics of one pinned toolchain, not language models, proof assistants or provers trained to read feedback in general. We propose no new prover and report no state-of-the-art comparison.

2 Related work

Neural provers for Lean and their feedback loops. LeanDojo (Yang et al., 2023) provides the interaction toolkit, retrieval-augmented tactic generator (ReProver) and the mathlib-derived benchmark that later Lean 4 work builds on and that our data manifest reuses. DeepSeek-Prover (Xin et al., 2024a) scales whole-proof generation with synthetic data; DeepSeek-Prover-V1.5 (Xin et al., 2024b) adds reinforcement learning from proof assistant feedback and a Monte-Carlo tree search variant, and DeepSeek-Prover-V2 (Ren et al., 2025) applies reinforcement learning to subgoal decomposition. Kimina-Prover (Wang et al., 2025) and Goedel-Prover-V2 (Lin et al., 2025) continue this line; Goedel-Prover-V2 in particular trains the model to revise its own proofs using Lean compiler feedback and reports that its self-correction mode improves over its standard mode on miniF2F (Zheng et al., 2021). In all of these systems the compiler is used as a reward or as an oracle during search;

our study is orthogonal in that it holds the checker fixed and manipulates only what the model is told about a failure.

Compiler-guided repair at inference time. APOLLO (Ospanov et al., 2025) is a model-agnostic pipeline that uses Lean to locate failing sub-lemmas, applies syntax fixes and automated solvers, and re-invokes the model on remaining goals, reporting large gains at low sampling budgets. APRIL (Wang et al., 2026) recasts repair as supervised learning: from 260,000 systematically generated failing proofs paired with compiler diagnostics and verified repairs, it fine-tunes models to produce a corrected proof (and optionally a natural-language diagnosis) and evaluates single-shot repair accuracy. These works establish that repair conditioned on diagnostics can be effective; they ask how well a system or a trained model repairs, not which component of the feedback the repair depends on when budgets are equalised.

Verifier signals during training and search. LeanListener (Rajaee et al., 2025) uses step-level Lean feedback as a reward during online policy optimisation of a tactic generator, and Lean-Progress (George et al., 2025) trains a model to predict the number of remaining proof steps to guide best-first search. Both use the verifier to shape what the model learns or explores; neither varies the content of failure feedback shown at repair time, which is the variable this study isolates.

Feedback content and budgeted evaluation in code repair. Our methodological precedents come from program synthesis. Self-Debugging (Chen et al., 2023) compares feedback variants for code repair, including execution results and model-generated explanations. Olausson et al. (2023) evaluate self-repair for code generation under an accounting that charges the repair samples, and report that gains are often modest once that cost is included. Huang et al. (2023) examine intrinsic self-correction without external feedback and find limited benefit. Our design applies the budget-accounting stance of these studies to formal proof, where the verifier is exact, and adds a length-matched placebo to separate semantic content from prompt length. Pass@k style repeated sampling (Chen et al., 2021) is the baseline that our NONE arm instantiates under the same verifier-call budget as the feedback arms.

3 A BUDGET-CONTROLLED FEEDBACK PROTOCOL

3.1 TASK, EPISODE AND TRAJECTORY

A *task* is a Lean 4 theorem statement with the compile context (imports and opens) needed to elaborate it in a pinned environment, identified by a hash of its source location and statement. An *episode* is a task with one *frozen initial failing candidate*: a proof attempt produced once by the study model, verified once, and rejected by the checker. The episode identifier hashes task and candidate and is written to every record of every arm, so all conditions start from a byte-identical state: same theorem, same failed candidate, same diagnostics available to be shown or withheld.

A *trajectory* is one (episode, condition, seed): a sequence of rounds in which the model receives a prompt, emits one candidate, and the candidate is submitted to the checker exactly once. A trajectory ends only at a recorded stop: the checker accepts a candidate (`solved`); the verifier-call budget B, the generated-token cap or the repair-round cap is exhausted; the wrapper plus theorem plus state alone exceed the input-token cap; or a recorded exclusion applies. No condition may stop for any other reason.

3.2 CONDITIONS

The prompt wrapper has four slots: a fixed instruction, the theorem statement, a *state* block and a *feedback* block. Only the last two depend on the condition (Table 1); the per-condition template hash and a hash of the complete frozen configuration are stored on every attempt record.

Contrasts have fixed roles, recorded on every comparison and checked by the validator. *Primary:* FULL VS. PLACEBO isolates semantic content from prompt length, and FULL VS. BINARY isolates it from the bare failure signal; both controls share FULL's state block and differ only in the feedback text. *Secondary:* FULL VS. NONE measures everything the loop adds over independent resampling at the same verifier-call budget; because NONE also withholds the previous candidate it changes two inputs at once and is never read as a diagnosis-only contrast. A conditional SHUFFLED contrast (another

Table 1: The manipulated variable. "Previous candidate" is the text of the last failed attempt. The PLACEBO filler is a repeated theorem-agnostic word whose token count equals the FULL text that this arm's own previous failure would have produced, before and after the shared truncation policy.

Condition	State block	Feedback block
FULL	previous candidate	authentic diagnostics + goal state of the previous candidate
BINARY	previous candidate	the fixed sentence "The previous attempt failed."
NONE	static marker, identical everywhere	empty; independent resample
PLACEBO	previous candidate	length-matched non-diagnostic filler

theorem's authentic diagnostics from the same error bucket and length bin) is defined in the protocol and enters only if a pre-specified bucket and leakage audit passes on the evaluation data; it is not part of the present study.

Controls obey an exact-form leakage rule: BINARY feedback is the fixed sentence byte-for-byte, NONE feedback is empty, and PLACEBO feedback is the filler unit alone, which may not contain theorem names, identifiers, source paths, diagnostic vocabulary or tactic tokens from fixed lists; the validator rejects violations and adversarial tests confirm the rejection.

3.3 Resource contract

All arms of a run share one public checkpoint and revision, tokenizer, inference stack, benchmark manifest, frozen initial failure per episode, decoding configuration and seed schedule, verifier-call budget B, repair-round cap, generated- and input-token caps, checker timeout, Lean toolchain, Mathlib commit, checker command and prompt wrapper, hashed into a single *frozen configuration hash* on every record; the validator rejects a run whose records carry more than one such hash.

The input-token policy is a per-call cap with one declared truncation function. The feedback budget of a round is the cap minus the token count of the prompt with an empty feedback slot, and the feedback block of *every* arm is cut to the longest word prefix within it. PLACEBO is built to the untruncated FULL length of the same arm's own previous failure and passed through the same function, so FULL and PLACEBO record identical pre-policy counts, post-policy counts and truncation events, all stored on the record and checked by the validator. A truncated trajectory is flagged, excluded from the primary analysis and counted in the excluded fraction.

3.4 Verifier-call, token and cache semantics

A *verifier call* is the submission of one candidate for one trajectory to the checking path. Every submission is charged exactly one call whatever its outcome: accepted, parse failure, elaboration failure, unsolved goals, failed tactic, kernel rejection, `sorry`, or per-call timeout (Appendix A). A candidate identical by hash to an earlier one in the same trajectory is not re-run; the stored result is reused, the reuse is recorded, and it is still charged one call, because otherwise a low-entropy condition would obtain extra search for free. Infrastructure failures of the checker process trigger a bounded number of uncharged, logged re-runs; reaching the bound flags the trajectory for exclusion. Verification of the frozen initial candidate happens once, outside B.

A candidate succeeds only if it is attached to the *original* statement (a candidate restating a different statement is not accepted), compiled in a fresh namespace of the pinned environment, and the Lean process exits successfully, emits no error and no `sorry` warning, reports no axiom outside `propext`, `Classical.choice` and `Quot.sound`, and does not reference the library's own copy of the target theorem. Model self-reports, script exit codes and similarity to a reference proof are not success. Token counts come from the checkpoint's own tokenizer; every record carries prompt length, generated length and the trajectory's cumulative totals. Wall time and checker CPU time are recorded separately from, and never merged with, the call budget.

3.5 Pre-specified analysis

Outcomes. For each condition: solved rate at every budget $b \in \{1, \ldots, B\}$; normalised area under the solve-versus-call curve, AUC $= \frac{1}{B} \sum_{b=1}^{B} \text{solved}(b)$; calls to first accepted proof; generated and input tokens; wall and checker time. Compliance counts (exclusions by reason, flagged trajectories, infrastructure retries, timeouts) are reported before any outcome.

Paired uncertainty. Every contrast is paired at the episode level: for FULL vs. X we take the per-episode difference in the solved indicator at B and report a percentile bootstrap interval over episodes, resampling all arms and seeds of an episode together; the resample count and bootstrap seed are recorded in the aggregate.

Splits, seeds and gates. A 20-episode, one-seed smoke test validates accounting and supports no claim. A development split, disjoint by source group from every other split, fixes B and the held-out size by a mechanical rule from measured throughput (Section 4); these are written into a versioned protocol manifest whose hash appears on every record before any held-out call. The held-out evaluation uses 219 episodes, 3 seeds and the four core conditions. SHUFFLED may be added only if a pre-specified donor-coverage and bucketing audit passes on the evaluation data; the scheduler refuses to start it otherwise. Any protocol change after a locked run is observed creates a new protocol version with a logged rationale, and runs under different versions are never pooled. Although not publicly preregistered, the protocol is pre-specified before the locked evaluation and its dated history is part of the artifact.

Exclusions. An (episode, condition) may be excluded only with a reason code from a fixed list, a stage, a recorder and a timestamp, every exclusion is listed in the analysis output, and an exclusion in any arm removes the episode from every arm's paired analysis. Exclusion decisions may not depend on the arm's outcome.

4 Experimental setup

Model. We use one public checkpoint, deepseek-ai/DeepSeek-Prover-V1.5-RL, at immutable revision `40a760138c68`, with its own tokenizer for every token count. Inference is local and sequential with batch size one; stack, version, data type and device are recorded in the model lock and hashed into the frozen configuration. Decoding uses the checkpoint's published sampling configuration (parameters in the run record), one run seed, and a per-call seed derived from the run seed and the round index, identical across conditions. No hosted API is used.

Benchmark and environment. Tasks are drawn from LeanDojo Benchmark 4 (v10)at commit `29dcec074de1`. Statements are compiled in a single pinned environment, Lean v4.10.0-rc1with Mathlib at `29dcec074de1`, built from a locked dependency manifest, with the checker invoked as a fresh process per candidate. The environment lock records toolchain and library identities, the checker command template, the per-call timeout, host identity, and the outcome of a fixture compiled twice to confirm determinism; its hash appears on every record. The model's public repository pins a Lean 4 toolchain one minor release earlier (v4.9.0-rc1) than the checker's v4.10.0-rc1; the gap is shared by all arms and can only lower absolute rates, not favour a condition.

Selection and splits. Selection uses only metadata available before any model or checker output: theorem name, source file, statement, and the length of the human proof. Eligible theorems are ordered by a salted hash fixed in the protocol and taken with at most one per source file and, for the held-out split, at most three per source directory. Splits are disjoint by source directory: the 20smoke and 100development episodes (from the benchmark's validation pool; used only to fix the budget and measure throughput) share none with the 219held-out episodes, drawn from the union of the benchmark's two test partitions (a dated pre-run amendment; Appendix F). Nothing is tuned on the held-out split. A theorem whose frozen initial candidate is accepted is replaced by the next in salted order; a failing initial candidate that reproduces the benchmark's human proof verbatim excludes the theorem as suspected leakage, also with replacement. Every theorem examined and every exclusion is in the selection log.

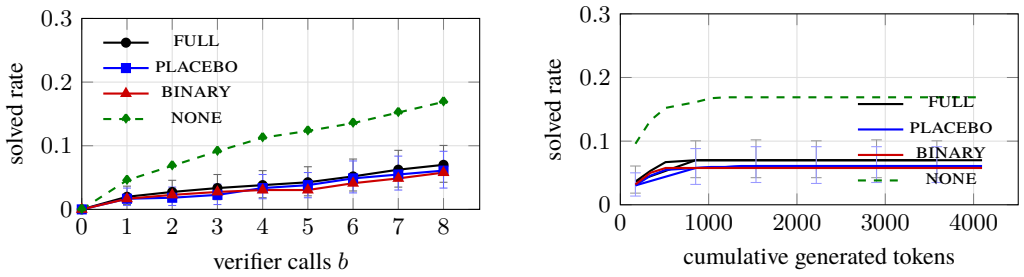

Figure 1: Left: solved rate (episode-level mean over three seeds) against verifier calls $b = 1..8$ for the four core conditions, with 95% episode-level bootstrap intervals for FULL and PLACEBO. Right: the same against cumulative generated tokens (the additional view chosen before the held-out run; rank stability is in Appendix G). Drawn from the generated CSVs.

Budgets. The smoke test uses $B = 4$ with one seed for accounting validation only; every pre-specified stop condition passed and all 240 of its checker calls were reproduced independently (Appendix E); it supports no claim. The held-out evaluation uses a primary budget of 8 verifier calls and 3 seeds, fixed by a mechanical rule from a throughput test on the first 40 development episodes at eight calls: the rule sets B, the held-out size and the seed count from measured wall time under a 40-hour compute cap, stops for a human decision if fewer than 5% of development trajectories reach an accepted proof, and its output was written into the protocol manifest before any held-out call. Per-call output cap, per-trajectory token cap, per-call input cap and truncation policy are identical across conditions and recorded in the run record.

Raw record provenance. Before the first generation of a run, a pre-run manifest fixes every fingerprint the run depends on (protocol, code commit and tree hash, data, selection, environments, model, checker, frozen configuration, templates); the run record must agree with it. Every attempt is one JSON line carrying these hashes, the condition, seed, round and budget snapshot, the feedback template and token accounting, the candidate hash and cache status, the exact checker command, exit code, normalised outcome, error bucket, raw-log hash, timings and stop reason (schema in Appendix D). Aggregates are recomputed from these records by the analysis script, and the validator rejects an aggregate that does not match. Hardware (one arm64 workstation, 18 CPU cores, Darwin; model inference on the Apple-silicon GPU via MPS in bfloat16) is recorded; no outcome depends on it except the shared per-call timeout.

Diagnostic buckets. Failed checks are assigned to one of six pre-specified buckets (syntax error, unknown identifier, type mismatch, unsolved goals, tactic failed, timeout) by a versioned ordered rule table applied to the first error message (Appendix C). Bucket-level slices are descriptive unless the bucket audit passes.

5 Results

5.1 Protocol compliance

The three held-out runs (seeds 0, 1, 2; $B = 8$) cover the same 219 episodes, 2628 trajectories in total. No episode was excluded in any arm or seed, no trajectory carries a protocol flag, no feedback block was truncated, and no infrastructure-class checker failure occurred; 5 checker calls timed out and are charged as failures. All three runs share one frozen-configuration hash, each run record agrees with its pre-run manifest, and the standalone validator accepts every record. The runs were interrupted and resumed for reasons unrelated to the model or checker; every event and the two outcome-blind trajectory discards are dated in the run log shipped with the artifact. An independent audit re-executed the checker on every accepted proof and on 200 seeded failures per seed (Appendix H).

Table 2: Primary endpoint at $B = 8$ over 219 episodes and three seeds: solved trajectories out of $657 = 219 \times 3$; solved rate with 95% episode-level percentile bootstrap interval; paired difference to FULL (pp) with its interval; median calls to first proof among solved trajectories; mean input and generated tokens per trajectory; violations. †NONE also differs from FULL in the state block, so FULL −NONE measures everything the repair loop adds over independent resampling.

Cond.	Solved/657	Rate at B (%) [CI]	Δ vs FULL (pp) [CI]	Calls_{1st}	In tok	Gen tok	Viol.
FULL	46	7.0 [4.3, 10.2]	–	4.0	3246	721	0
BINARY	38	5.8 [3.2, 8.8]	+1.2 [-2.1, +4.4]	4.0	2419	753	0
NONE	111	16.9 [12.3, 21.8]	-9.9 [-14.5, -5.3]†	3.0	1576	716	0
PLACEBO	40	6.1 [3.5, 9.0]	+0.9 [-2.4, +4.0]	4.0	3283	694	0

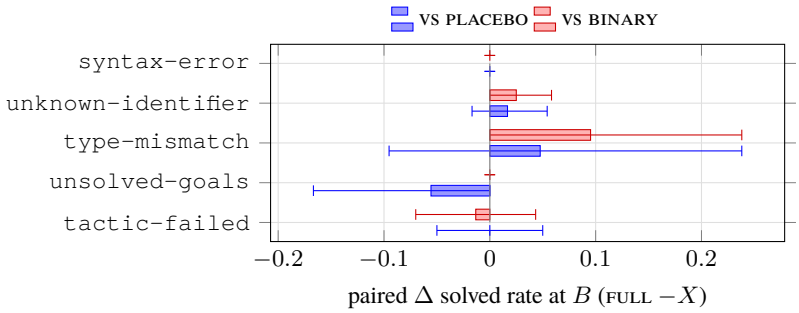

Figure 2: Paired FULL $-X$ difference in solved rate at B for the two primary contrasts, by error bucket of the frozen initial failure, with episode-level bootstrap intervals. Episode counts per bucket are in Table 7. No held-out initial failure fell in the timeout bucket. Bucket slices are descriptive: the shuffled-control bucket audit was not run.

5.2 Primary endpoint: success under a matched verifier-call budget

Table 2 and Figure 1 report the pre-specified contrasts in their fixed order. The paired FULL −PLACEBO difference in solved rate at B is +0.9 pp with 95% interval [-2.4, +4.0] (FULL had the higher solve rate over seeds on 14 episodes, PLACEBO on 9). The paired FULL −BINARY difference is +1.2 pp, interval [-2.1, +4.4] (15 vs. 10 episodes). Both primary intervals include zero. The rates themselves are low: FULL 7.0%, PLACEBO 6.1%, BINARY 5.8%. The secondary FULL −NONE difference is -9.9 pp, interval [-14.5, -5.3]: independent resampling from the frozen initial failure (NONE, 16.9%) reached an accepted proof more often than the repair loop with full diagnostics, and its interval excludes zero. Per seed, the two primary differences range from -0.5 to $+2.3$ pp and every per-seed interval includes zero; the FULL −NONE difference is between -9.1 and -11.0 pp in each seed (`per_seed.csv`). Across the 24 (seed, budget) cells, NONE ranks first in every cell and FULL ranks first in none (Appendix G); the order among FULL, PLACEBO and BINARY varies from cell to cell.

5.3 Effect by checker-observable failure category

Table 7 (appendix) and Figure 2 slice the primary contrasts by initial-failure bucket; two buckets carry most episodes (tactic failed, 100; unknown identifier, 80). Every primary interval contains zero or has it as an endpoint, the latter only in buckets with at most eight discordant episodes, so these reflect small counts, not detected effects. NONE has the highest rate in every bucket.

5.4 Token frontier

The generated-token budget was shared and rarely binding (one trajectory per seed stopped on it), and the token frontier (Figure 1, right) shows the same picture as the call frontier: NONE has the highest rate at every one of the 24 token levels, FULL is second at each, and BINARY and PLACEBO exchange places. Feedback conditions did not trade tokens for calls differently.

5.5 A descriptive observation on candidate repetition

After the primary analysis, we examined a quantity in the raw records that was not pre-specified and is relevant to the secondary result. In the three conditions that show the model its previous candidate (FULL, BINARY, PLACEBO), between 27.0% and 31.9% of generate-and-verify rounds re-proposed a candidate already tried in the same trajectory, and about 92.0% of those repeats were the immediately preceding candidate (FULL 27.7%). Under the protocol a repeat is served from the per-trajectory cache and still costs one verifier call. NONE, which re-samples independently, repeated in 1.8% of rounds. The mean number of distinct candidates per eight-call trajectory was 5.6 (FULL), 5.7 (BINARY), 5.3 (PLACEBO) and 7.1 (NONE). Lost calls account for part, but not all, of the NONE advantage, since FULL had about 78% as many distinct candidates as NONE but about 41% of its solved rate. Whether the remainder reflects conditioning on a failed candidate narrowing the model's proposals is a hypothesis these data cannot test, and we do not claim it.

5.6 Case studies selected by rule

Three episodes were selected by the rule fixed before the held-out run, ordered by task hash and never by outcome (Table 5, Appendix G): *benefit*, the smallest task hash among episodes solved by FULL in every seed and by no control in any; *null*, solved by every condition in every seed with identical calls to first proof (tier 1; tier 2 drops the call-count requirement and the tier used is recorded); *misleading or inconclusive*, solved by PLACEBO or NONE in at least one seed and by FULL in none. All three categories were non-empty. In the benefit case the initial failure was a stalled `simp` and FULL alone reached a proof; in the misleading case an invalid field-notation error was followed by a proof from NONE in every seed and from none of the conditions that saw the previous candidate.

6 Limitations

Model and template dependence. The study evaluates one public checkpoint under one prompt wrapper, so its conclusions concern the informational value of diagnostics *for this model under this wrapper*. The checkpoint was, by its report (Xin et al., 2024b), refined by reinforcement learning with verifier acceptance as the reward and was not trained to read compiler diagnostics; a model trained on diagnostic-conditioned repair data, or a differently worded wrapper, could change how any condition is used.

Benchmark representativeness and contamination. Tasks come from one public benchmark extracted from one Mathlib commit. Mathlib and its human proofs are very likely in the pre-training data of any public Lean model, so absolute success rates are not estimates of performance on unseen mathematics. Because every contrast is within-episode, contamination shifts the base rate for all conditions rather than favouring one, but it could interact with feedback content (if diagnostics cue memorised lemma names). Near-duplicate theorems within the benchmark remain a source of correlation that paired resampling by episode does not remove.

Diagnostic heterogeneity and control mismatch. Lean diagnostics vary in length, structure and specificity across error classes. PLACEBO matches token length but not structure, and a shuffled-diagnostic control that would match structure too was not enabled because its identity-leakage audit is a precondition we do not assume. A null FULL $-$PLACEBO contrast could mean diagnostics that are uninformative for this model, or a placebo that is the wrong control for a particular bucket; bucket slices are exploratory. Truncation of long diagnostics is flagged and excluded rather than analysed (no trajectory was truncated here). NONE is not a clean diagnosis-only control, since it also withholds the previous candidate; FULL $-$NONE is an operational baseline only.

Timeouts and what acceptance means. Timeouts are charged as failures, so a slow but correct proof counts as wrong for every condition that produces it; wall-clock timeouts depend on hardware and load. Acceptance by a pinned toolchain and Mathlib commit means that the candidate elaborates to a term of the stated type without `sorry` or unexpected axioms; it does not mean the proof is idiomatic or would compile under another Mathlib version.

Statistical scope and low base rates. With 219 episodes and 3 seeds the study was designed to detect moderate differences in solved rate at the pre-specified budget; the primary intervals span roughly ± 3 pp, so a diagnostic effect smaller than that, or one that appears only at budgets beyond eight calls, is outside its power. Solved rates were low (7.0% for FULL), which limits how much any contrast could move. All slices beyond the primary contrasts are descriptive, as is the candidate-repetition analysis of Section 5.5, which was not pre-specified.

Wall-time measurements. The runs were interrupted and resumed several times (machine sleep, one memory incident, one concurrent audit window), each dated in the run log; trajectories spanning an interruption carry inflated wall times. Outcomes, call counts and token counts are unaffected; times in Table 6 are upper bounds.

7 Conclusion

Compiler-guided proof repair is widely used, but "feedback helps" conflates at least four mechanisms: a failure signal, a longer prompt, visibility of the previous attempt, and another sample. Our protocol separates them under an exactly matched verifier-call, token, decoding and attempt budget, with success defined only by a pinned Lean checker, and applied it to one public checkpoint on 219 held-out failures with three seeds. The paired FULL $-$PLACEBO and FULL $-$BINARY differences at $B = 8$ are +0.9 and +1.2 pp with intervals [-2.4, +4.0] and [-2.1, +4.4]; the secondary FULL $-$NONE difference is -9.9 pp with interval [-14.5, -5.3]. For this model, wrapper and benchmark, the cheaper controls recover whatever the repair loop adds, and showing the model its failed attempt, with or without diagnostics, was associated with fewer distinct candidates and fewer accepted proofs than resampling from the initial failure. A model trained on diagnostic-conditioned repair data could behave differently; the released artifact lets the protocol be applied to it.

AI use statement

In this work we used generative AI tools (a large-language-model coding assistant operating inside a code editor) for the following tasks that the ICLR 2027 AI policy lists as requiring disclosure: designing and providing feedback on the research methodology and experimental protocol (the condition definitions, budget accounting rules, exclusion policy and analysis plan were drafted with the assistant from the authors' specification); implementing methods (the scheduler, accounting library, validator, checker and model adapters, and analysis scripts were written with the assistant); generating synthetic data (the deterministic toy fixtures used to test the pipeline, which are labelled synthetic and are not used as evidence); interpreting results (the assistant ran the pre-specified analysis script on the locked records and drafted the prose reading its output, including the post-hoc repetition descriptives; every number in that prose is a macro emitted by the script); and cleaning or reformatting data (the assistant wrote the scripts that archive, validate and re-aggregate the raw records; no record was edited by hand). We also used the assistant for tasks with recommended disclosure: drafting and editing parts of this manuscript, suggesting its structure, identifying and formatting references, summarising existing literature, and creating figures from the recorded data (the figure sources are drawn by PGFPlots from generated CSVs). We have not used generative AI tools to formulate or prove mathematical claims, and no such claims appear in this work. All AI-assisted work was checked before submission: every bibliographic record was compared against its arXiv or DOI record; the code is covered by the unit and validation tests released in the artifact; every number in the paper is generated from the raw records and was re-derived by a second, independent script; and every accepted proof was re-checked by the pinned Lean checker in an independent audit. We take responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

Reproducibility statement

Every quantitative claim in this paper is derived from raw per-attempt records by one analysis command, and every input to those records is pinned. The artifact (supplementary material, anonymised) maps each class of claim to its evidence. *Protocol claims* (Section 3): the machine-readable protocol manifest, its prose form and dated change log, and the hash of the manifest on every record.

Setup claims (Section 4): the data proposal and selection manifest with benchmark version, commit, license, salted selection rule, episode list and replacement log; the environment lock with Lean toolchain, Mathlib commit, dependency manifest hash, checker command and timeout, and the determinism fixture result; the model lock with checkpoint revision, tokenizer hash, inference stack version and decoding configuration; the exact prompt wrapper and per-condition feedback templates with their hashes; all run seeds; and the pre-run manifest of every run. *Result claims* (Section 5): the raw JSONL attempt records with budget snapshots, provenance hashes, feedback policy fields and stop reasons; the checker logs addressed by hash; every exclusion record; the shuffled-control audit record if that condition was run; the JSON schemas and the validator that checks the accounting invariants (budget, cache, initial-failure identity, frozen configuration, feedback policy and placebo length, control leakage, success oracle, exclusion coverage and visibility, contrast roles); and the analysis code that regenerates every table and figure. The mapping from each sentence of the paper to these files is kept in a claim–evidence ledger shipped with the artifact. Running `make validate` on the released records reproduces the compliance verdict, and running the analysis script on the released records reproduces Tables 2 and 5 and Figures 1 and 2 byte-for-byte; re-running the experiment itself requires the pinned environment and checkpoint and took about 44 hours of summed per-trajectory wall time for the three seeds (Table 6; an upper bound, since the runs were interrupted and resumed) on one arm64 workstation, 18 CPU cores, Darwin; model inference on the Apple-silicon GPU via MPS in bfloat16. Synthetic fixtures used to test the pipeline are labelled as such in every record and are not Lean evidence.

References

Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 58 authors; introduces pass@k.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023. Venue not yet verified against the proceedings page; cite as preprint until the claim ledger row is closed.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Robert Joseph George, Suozhi Huang, Peiyang Song, and Anima Anandkumar. LeanProgress: Guiding search for neural theorem proving via proof progress prediction. *Transactions on Machine Learning Research*, 2025. arXiv:2502.17925; author order as listed by the arXiv API.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023. Venue not yet verified against the proceedings page; cite as preprint until the claim ledger row is closed.

Yong Lin et al. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025. 20 authors; first author verified via arXiv API.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896*, 2023. Venue not yet verified against the proceedings page; cite as preprint until the claim ledger row is closed.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2505.05758.

Sara Rajaee, Kumar Pratik, Gabriele Cesa, and Arash Behboodi. Local look-ahead guidance via verifier-in-the-loop for automated theorem proving. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025. doi: 10.18653/v1/2025.findings-acl.825. LeanListener; arXiv:2503.09730.

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

Evan Wang, Simon Chess, Daniel Lee, Siyuan Ge, Ajit Mallavarapu, Jarod Alper, and Vasily Ilin. Learning to repair Lean proofs from compiler feedback. *arXiv preprint arXiv:2602.02990*, 2026. Introduces the APRIL dataset.

Haiming Wang et al. Kimina-Prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025. 40 authors; first author verified via arXiv API.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024a.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024b.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023. arXiv:2306.15626.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021. Venue not yet verified against the proceedings page; cite as preprint until the claim ledger row is closed.

A Verifier-call accounting rule

Table 3 is the normative counting rule; the released validator enforces it on every record.

Table 3: Charged verifier calls per event.

Event	Charged calls
Fresh check, accepted	1
Fresh check, parse failure	1
Fresh check, elaboration failure	1
Fresh check, checker failure (goals remain, tactic failed, kernel rejection, `sorry`)	1
Fresh check, wall-clock or heartbeat timeout	1
Duplicate candidate (same hash, same trajectory): result reused, source logged	1
Cache read from any other scope (source logged)	1
Cache write	0
Infrastructure re-run of the same submission (bounded; reaching the bound flags the trajectory)	0
Verification of the frozen initial candidate (once, before arms exist)	0
Budget refusal (logged action, nothing executed)	0

B Prompt wrapper and condition templates

The wrapper is byte-identical across conditions apart from the state and feedback slots, which sit in two Lean block comments immediately before the theorem. The text below is emitted from the code by `paper/gen_appendix.py`.

Wrapper. Identifier `prompt-complete-lean4-v3`. The completion is the tactic block after `:= by`.

````
Complete the following Lean 4 code:

```lean4
{context}
/- {state_block} -/
/- {feedback_block} -/
{statement} := by
````

State slot. `Previous attempt:\n<previous candidate>` for FULL, BINARY and PLACEBO; the static marker `[RETRY]` for NONE. No round counter is shown to any arm.

Feedback slot. FULL: the checker's diagnostic text followed by the goal state, verbatim (template `fb-full-v1`). BINARY: the fixed sentence `The previous attempt failed.` (`fb-binary-v1`). NONE: empty (`fb-none-v1`). PLACEBO: the unit `filler` repeated to the token length of this arm's own untruncated FULL text for the same round, then passed through the same truncation function (`fb-placebo-filler-v1`). Truncation policy `truncate-tail-words-v1`: tail words are dropped until the block fits the per-call input cap; the event is recorded.

C Diagnostic bucket rules

Ordered rule table (first match on the first error message wins), version recorded in every checker manifest; emitted from the code. Rule table `bucket-rules-v0.1`; applied in order to the first error message, first match wins.
````

Order	Bucket	Regular expression on the first error message
1	`syntax_error`	`unexpected token|expected token| unterminated|unexpected end of input| expected term|expected command`
2	`unknown_identifier`	`unknown (identifier|constant|namespace| tactic)|unknown constant`
3	`type_mismatch`	`type mismatch|has type .* but is expected to have type|application type mismatch| failed to synthesize|numerals are polymorphic`
4	`unsolved_goals`	`unsolved goals`
5	`timeout`	`deterministic timeout|maximum recursion depth|\{\}(deterministic\{\}) timeout| heartbeats`
6	`tactic_failed`	`.*`

Shuffled-control audit. The protocol defines a shuffled-diagnostic control and an audit it must pass (coverage of recipients with an admissible donor, donor diversity per bucket and length bin, no identity overlap, agreement of an independent human bucketing with the rule table). The audit was not passed on the evaluation data at the time of writing, so the control is not part of this study and no result involving it is reported.

D Record schema

Schema version `0.3.1` (`attempt.schema.json`, sha256 `91cc2967d5aed832`); 61 fields including nested objects. Every record is one JSON line; unknown fields are rejected.

```
action_type
attempt_id
attempt_index
budget_B
budget_after
budget_after.cache_reads
budget_after.cache_writes
budget_after.generated_tokens_cap
budget_after.generated_tokens_used
budget_after.infra_retries
budget_after.input_tokens_cap_per_call
budget_after.input_tokens_used
budget_after.rounds_cap
budget_after.rounds_used
budget_after.verifier_calls_cap
budget_after.verifier_calls_used
budget_after.wall_ms_cumulative
candidate
candidate_index
condition
decoding
decoding.max_new_tokens
decoding.temperature
decoding.top_p
episode_id
feedback
initial_failure_hash
initial_failure_id
instance_id
is_synthetic
model
model.inference_stack
model.inference_version
model.model_checkpoint
model.revision
model.tokenizer_id
prompt_hash
protocol_flags
provenance
provenance.code_commit
provenance.dataset_manifest_hash
provenance.environment_manifest_hash
provenance.feedback_template_hash
provenance.frozen_config_hash
provenance.model_manifest_hash
provenance.prompt_template_hash
provenance.protocol_manifest_hash
provenance.protocol_version
provenance.selection_manifest_hash
provenance.source_environment_hash
provenance.source_group
provenance.task_hash
provenance.theorem_name
round_index
run_id
schema_version
seed
stop_reason
synthetic_label
tokens
verifier
```

E Smoke test outcomes (not evidence)

The 20-episode smoke test (one seed, $B = 4$, four core conditions, 80 trajectories, 320 generate-and-verify actions) was run once to completion under the locked model, environment and protocol. Every pre-specified stop condition passed: the environment fixture was deterministic; the validator reported zero issues on the raw records; no arm exceeded its verifier-call, round or token caps; every one of the 80 cache reads was logged with its source and charged; no control-arm feedback failed

the leakage scan; FULL and PLACEBO feedback matched in post-policy length in every episode with no truncation event; there were no checker timeouts, infrastructure failures or retries; the run record agreed with the pre-run manifest on every fingerprint; and the aggregate regenerated byte-for-byte from the raw records. An independent audit script re-executed all 240 distinct checker calls in the pinned environment and reproduced the recorded outcome, acceptance and error bucket in 240 of 240 cases.

Descriptive counts, labelled *smoke test* and not used for any claim: 1 of 80 trajectories reached an accepted proof (NONE, round 4); the three arms whose prompt shows the previous candidate regenerated it verbatim in 28–38% of rounds against 1% for NONE, each such round being a charged cache read. Median wall time was 38 s per trajectory (checker 3.7 s per executed call).

F HELD-OUT POOL AMENDMENT (BEFORE ANY HELD-OUT CALL)

The protocol first named one of the benchmark's two test partitions as the held-out pool. Under the unchanged selection rules (directory-level disjointness from the smoke and development draws, one theorem per file, at most three per directory, human proof of 1–10 tactic lines, salted order) the first draw stopped after examining 81 candidates with 14 episodes: the smoke and development draws had used 62 of that partition's 98 source directories, and 1769 of its 2000 theorems fell in one of them. The pool was widened to the union of both test partitions (3972 unique theorems), from which the same rules admit 457 candidates. This was the only rule changed, it was made before any held-out model or checker call under any budget, and the failed draw's log, the dated protocol history entry and the change-log rationale ship with the artifact. The two partitions differ only in how the benchmark authors assigned theorems to train, validation and test for retrieval experiments; this study trains nothing.

G ADDITIONAL ANALYSES

Rank stability. Table 4 counts the 24 (seed, budget) cells by the full ordering of solved rates and lists the per-seed paired contrasts. NONE is first in every cell; FULL is first in none.

Table 4: Rank stability across seeds and budgets, and per-seed paired contrasts at $B = 8$.

Order of solved rate (over 3 seeds $\times$ 8 budgets $=$ 24 cells)	Cells
NONE $>$ FULL $>$ PLACEBO $>$ BINARY	12
NONE $>$ BINARY $>$ FULL $>$ PLACEBO	8
NONE $>$ FULL $>$ BINARY $>$ PLACEBO	4

Seed	Contrast	Solved FULL/ X	Δ (pp)	95% CI
0	FULL $-$PLACEBO	13 / 13	+0.0	[-3.2, +3.7]
1	FULL $-$PLACEBO	17 / 14	+1.4	[-2.7, +5.5]
2	FULL $-$PLACEBO	16 / 13	+1.4	[-2.7, +5.0]
0	FULL $-$BINARY	13 / 14	-0.5	[-4.6, +3.2]
1	FULL $-$BINARY	17 / 12	+2.3	[-1.4, +5.9]
2	FULL $-$BINARY	16 / 12	+1.8	[-2.3, +5.9]
0	FULL $-$NONE	13 / 37	-11.0	[-16.4, -5.9]
1	FULL $-$NONE	17 / 37	-9.1	[-14.2, -4.6]
2	FULL $-$NONE	16 / 37	-9.6	[-15.1, -4.1]

Case studies. The three rule-selected episodes of Section 5.6.

Resources and repetition. Table 6 gives checker and wall time per trajectory, cache reads (repeat candidates served from the per-trajectory cache, each charged one verifier call), timeouts, the area under the solved-versus-calls curve, and the post-hoc repetition descriptives of Section 5.5. Wall times include the interruptions listed in the run log and are upper bounds.

Table 5: Rule-selected case studies (Section 5.6). The diagnostic is the first line of the checker output shown to FULL, verbatim; the last two columns count seeds (of three) in which the arm reached an accepted proof.

Kind	Initial bucket	Initial diagnostic (first line)	FULL	PLACEBO
benefit	tactic_failed	`simp made no progress`	3/3	0/3
null	tactic_failed	`simp made no progress`	3/3	3/3
misleading	tactic_failed	`invalid field notation, type is not of the form (C ...) where C is a constant`	0/3	0/3

Table 6: Per-condition resources. The last two columns were added after the primary analysis and are descriptive.

Condition	Checker s	Wall s	Cache reads	Timeouts	AUC	Repeat (%)	Distinct
FULL	19.4	62.4	1407	1	0.043	27.7	5.6
BINARY	19.2	58.1	1380	1	0.034	27.0	5.7
NONE	24.2	61.4	88	1	0.112	1.8	7.1
PLACEBO	18.0	60.9	1629	2	0.037	31.9	5.3

Bucket slices. Table 7 gives solved rates and the two primary paired differences by bucket of the initial failure (Figure 2).

Table 7: Solved rate at B (%) by error bucket of the frozen initial failure, with paired differences (pp) and 95% episode-level intervals. Descriptive; the bucket audit was not run.

Initial bucket	n	FULL	BINARY	NONE	PLACEBO	Δ FULL $-$PLACEBO	Δ FULL $-$BINARY
syntax_error	12	0.0	0.0	16.7	0.0	+0.0 [+0.0, +0.0]	+0.0 [+0.0, +0.0]
unknown_identifier	80	5.4	2.9	12.5	3.8	+1.7 [-1.7, +5.4]	+2.5 [+0.0, +5.8]
type_mismatch	21	14.3	4.8	17.5	9.5	+4.8 [-9.5, +23.8]	+9.5 [+0.0, +23.8]
unsolved_goals	6	0.0	0.0	11.1	5.6	-5.6 [-16.7, +0.0]	+0.0 [+0.0, +0.0]
tactic_failed	100	8.0	9.3	20.7	8.0	+0.0 [-5.0, +5.0]	-1.3 [-7.0, +4.3]

H Independent audit of the held-out runs

An audit script independent of the runner re-read every record of each completed run and checked 23 conditions per run: manifest precedence and agreement, validator verdict, coverage of all four arms on every episode, identity of the frozen initial failure across arms, budget and cap adherence, one frozen configuration, token counters present, every cache read logged and charged, every solved trajectory backed by a checker-accepted record, no accepted record with a non-success outcome, control-arm leakage scan, FULL/PLACEBO length matching, truncation and exclusion visibility, infrastructure-failure fraction, and the existence and hash of every raw checker log. All checks passed for all three seeds (69/69). It then re-executed the pinned checker on every accepted proof and on a seeded sample of 200 failures per seed: 835 re-executions, 835/835 agreeing on outcome, acceptance and error bucket, 0 missing logs, 0 log-hash mismatches; all 235 accepted proofs were re-accepted. The audits ran after all generation had finished, with no other checker or model process on the host. (The audit's 'infra-class' counter is the number of records whose checker error class is not 'none'; across the three seeds it equals the number of records whose error class is `timeout`, so no infrastructure-class failure was recorded.)

Seed	Run (`run-…`)	Verdict	Checks	Records	Cache reads	Timeouts	Re-executed	Agree
0	`fb1a8c36ec22`	PASS	23/23	6677	1668	2	277	277/277
1	`7aefcf155ddc`	PASS	23/23	6697	1444	0	280	280/280
2	`3f2700844ce1`	PASS	23/23	6687	1392	3	278	278/278

I Full exclusion log

No episode was excluded in any arm or seed.