FROM NATURAL-LANGUAGE PROPOSITIONS TO LEAN PROOFS: PROOF-SEARCH FEEDBACK FOR SOURCE-GROUNDED RE-FORMALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

End-to-end formalization and proving transforms a natural-language mathematical proposition into a Lean theorem together with a kernel-checked proof. In the proposition-only setting, no proof is supplied, so the system must discover the proof itself. Once a formalization is accepted, however, many pipelines keep it fixed during proof search, leaving proof failures unable to expose semantic defects in the statement. We call the resulting rate of self-accepted but unproved formalizations the operational semantic–proof closure gap. We study whether concrete proof-failure information produced by a model can help that same model discover such defects. We instantiate a verify-before-revise loop at the proposition-only Lean formalization–proof boundary: when proof search produces a suspected-false signal, a separate semantic audit compares the root formalization with the original proposition, and a revision is allowed only when the audit identifies a concrete, proposition-grounded discrepancy. Across two models and two datasets, the closed loop reduces the closure gap and improves externally validated end-to-end success, including BEq-validated success.

1 INTRODUCTION

Proof assistants such as Lean, Isabelle/HOL, and Coq express mathematics formally and verify proofs with small trusted kernels or proof checkers (de Moura & Ullrich, 2021; Nipkow et al., 2002; The Coq Development Team, 2023). End-to-end formalization and proving must translate a natural-language proposition into a faithful formal statement and construct its checked proof (Singh et al., 2025; Xu et al., 2026). Existing pipelines connect formalization and proving under two information settings. Some receive a supplied proof or solution route that coordinates formalization and proving, whereas proposition-only pipelines receive only the natural-language problem and must find a proof themselves (Jana et al., 2026; Zhang et al., 2026; Liu et al., 2025b; Yu et al., 2026). In both settings, the selected formal statement is typically fixed after initial checks, so proof outcomes do not trigger a renewed comparison between the root formalization and the natural-language source. We address this fixed-statement limitation in proposition-only proving.

This design leaves a measurable mismatch: a formalization can pass the system's semantic self checker yet remain unproved under the available budget. We observe this mismatch in our fixed-statement baseline and call its all-input rate the **operational semantic–proof closure gap**. With the same base model used for semantic self-checking and proof search, the gap is positive in all four evaluated model–dataset combinations and reaches 14.8 percentage points.

These observations suggest reopening the comparison with the natural-language source when proof search exposes a problem, rather than treating the initial formalization as final. However, proof failure is ambiguous: it may reflect a semantic defect or simply an incomplete search. Unconstrained revision also creates a shortcut, since strengthening assumptions, weakening the conclusion, or dropping a constraint can make a distorted theorem easier to prove. Thus proof failure can motivate a renewed semantic check, but cannot by itself justify changing the theorem. This motivates a controlled feedback loop in which proof diagnoses reopen the source comparison while an independent audit retains revision authority.

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.

We instantiate this controlled loop as a verify-before-revise policy at the proposition-only Lean formalization–proof boundary. The system first generates and validates a Lean formalization. A knowledge-accumulating pre-pruning module then screens proposed decompositions for suspected falsity, circularity, or similarity to prior failures, using rejected proposals to guide later search. The prover searches recursively for a proof, and when a suspected-false judgment reaches the root, its diagnosis string is passed to a fresh semantic audit as auxiliary evidence. The diagnosis cannot authorize revision: only an audit-reported, proposition-grounded discrepancy permits a change to the root formalization, and every replacement is revalidated before proof search resumes. In this way, proof failure prompts source-grounded re-examination without making ease of proof the semantic criterion. This design lets us test whether concrete, model-generated failure signals help the same model locate semantic vulnerabilities that a generic failure signal leaves unspecified.

Across two models and two datasets, matched comparisons show that the loop reduces the gap relative to keeping the initial formalization fixed. Fixed offline ASCC-3-MV (reference-free) and BEq (reference-based bidirectional derivability) validators provide post hoc semantic assessments (Shi et al., 2026; Liu et al., 2025a). Validated end-to-end success increases in all four settings, including BEq+Proof, and no decrease in final-formalization pass rates is observed. The Full–Signal-only comparison is designed to test whether the model-generated diagnosis adds value beyond an uninformative failure signal; Full is best or tied in all eight setting–validator comparisons. The single-annotator paired audit of 137 root-reaching inputs finds no harmful *Full* transition among the five initially faithful cases (0/5), whereas *Gate-off* has 2/5 harmful transitions. This supports the role of the semantic gate in limiting weakening, although the small denominator does not rule out rare harms. Experimental results and code are available at https://anonymous.4open.science/r/proof_reform_anonymous-840D/.

1.1 Contributions

We make four main contributions:

- **An instantiation of verify-before-revise control for proposition-only Lean formalization.** Downstream proof diagnoses trigger renewed comparison with the natural-language proposition but cannot directly authorize theorem revision; every replacement is revalidated.
- **An empirical study of the operational semantic–proof closure gap.** In our fixed-statement baseline, we measure gaps of 6.5–14.8 percentage points across two models and two datasets, and use held-out ASCC-3-MV and BEq to assess how much of each gap receives external semantic support.
- **A knowledge-accumulating pre-pruning module.** Before expanding a decomposition, the module assesses its plausibility, circularity, and similarity to recorded failed strategies; rejected proposals inform later search.
- **A controlled cross-setting evaluation.** We compare the complete closed-loop policy (*Full*) with a baseline that keeps the formalization fixed (*Fixed*), starting from the same initial formalization with matched prover models and token budgets. Two fixed offline semantic validators (ASCC-3-MV and BEq) show consistent gains in externally validated end-to-end success.

2 Related Work

Autoformalization and proof construction. Autoformalization systems improve statement generation through retrieval, decomposition, learned traces, semantic consistency, process supervision, iterative correction, tool feedback, and consistency rewards from unlabeled data (Lu et al., 2025b; Min et al., 2026; Wu et al., 2026; Shi et al., 2026; Chen et al., 2026; Lu et al., 2025a; Li et al., 2024; Wu et al., 2022; Lu et al., 2024; Pathak, 2024; Zhang et al., 2025; Wang et al., 2026; Gallardo et al., 2025; Huang et al., 2025). Semantic evaluators assess natural–formal alignment with reference-free or reference-based judgments, while the *faithfulness gap* captures mismatch between a natural-language source and a type-correct, provable statement (Shi et al., 2026; Liu et al., 2025a; Mohammad & Sheikh, 2026). Model-centric and agentic provers, proof-structure editors, and Pythagoras-Prover target decomposition, lemma generation, dependency management, curriculum training, and

proof-trace filtering over formal targets (Wang et al., 2025; Chen et al., 2025; Lin et al., 2026; Ospanov et al., 2025a; Varambally et al., 2026; Xin et al., 2026; Chung et al., 2026; Xiao et al., 2026; Lin et al., 2025; Xin et al., 2025; Leang et al., 2026); related systems are benchmarked on miniF2F and PutnamBench (Ospanov et al., 2025b; Tsoukalas et al., 2024). Sonoda et al.'s *provability gap* compares finite-budget proof policies on fixed formal states (Sonoda et al., 2026), whereas our *closure gap* compares semantic self-acceptance with proof completion within one system.

End-to-end formalization under different information settings. End-to-end pipelines either formalize a theorem paired with a proof or solution route (ProofBridge and monotonic reference-free refinement) or construct formal problem-solving data from an informal problem, answer, and solution (CoPA) (Jana et al., 2026; Zhang et al., 2026; Liu et al., 2025b). Proposition-only systems receive only the informal problem; Mathesis selects a compiler-valid Lean statement through semantic assessment before proving it (Yu et al., 2026). Because the formal statement is usually fixed after initial checks, we use proof-stage evidence to audit and revise the root theorem only when an audit confirms a source-grounded discrepancy.

Verification-guided refinement and evidence-gated generation. Our control design is informed by prior verification-guided refinement and evidence-gated generation, including RARR, CRITIC, Chain-of-Verification, and related checker- or tool-gated orchestration (Gao et al., 2023; Gou et al., 2024; Dhuliawala et al., 2024; Zhou et al., 2024). We adapt this verify-before-revise pattern to proposition-only Lean formalization and proof: proof-search feedback triggers a fresh comparison between the root theorem and the natural-language proposition, while only a source-grounded audit can authorize revision.

3 Problem Setting

3.1 Task and Success Criteria

Given only a natural-language problem P, the system must produce a Lean formalization F that expresses P and a Lean-kernel-checked proof of F. A formalization enters proof search only after Lean validation and acceptance by the online semantic self checker S_M, instantiated with the setting's base model M. This acceptance defines the system's within-run boundary and is not an external correctness judgment. The fixed offline validators ASCC-3-MV and BEq never enter the loop; they assess the final formalization after the trajectory.

A run reaches *within-system semantic–proof closure* when its final formalization passes S_M and the same model-instantiated prover produces a kernel-checked proof. For a held-out validator V, *externally validated end-to-end success* additionally requires $V(F_{i,\mathrm{end}}) = 1$ together with that proof; runs failing the initial validation checks are unsuccessful. Passing S_M does not certify semantic equivalence, since a true-but-unfaithful formalization may still be proved. The closed loop uses proof-stage feedback to expose such mismatches. Expanded definitions and validator conventions appear in Supplementary Section A.1.

3.2 Operational Semantic–Proof Closure Gap

Let a be the all-input self-acceptance rate and c the all-input rate that is both self-accepted and proved. The *operational semantic–proof closure gap* (**closure gap**) is

$$\mathrm{CGAP} = a - c = \frac{|\mathrm{Self} \setminus \mathrm{Proved}|}{N}.$$

It measures self-accepted formalizations that remain unproved under the specified prover, Lean environment, and budget. The offline ASCC-3-MV and BEq validators are not part of this definition; they test whether closing the gap also improves final semantic quality. RQ1 measures the gap under *Fixed*.

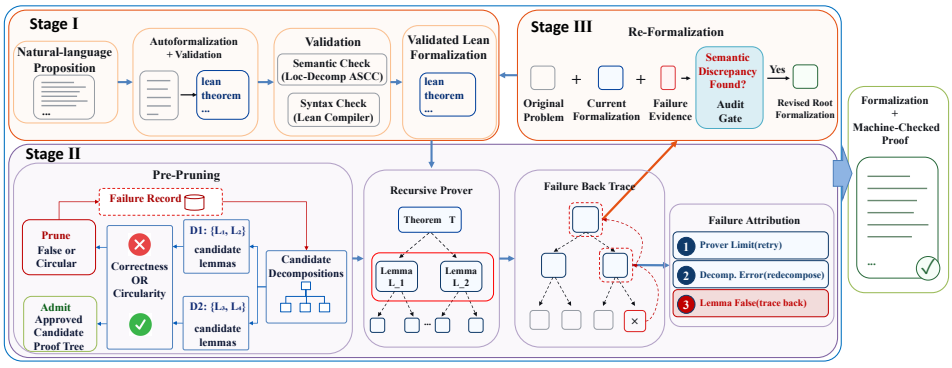


Figure 1: Proposition-only closed-loop formalization and proving with audit-gated re-formalization.

4 Test-Time Closed-Loop System

The system operates in three stages[1]. Stage I formalizes the natural-language problem and validates the result. Stage II runs recursive proof search; when a subtree fails, a diagnosis module attributes the failure to prover limitation, a bad decomposition, or a suspected-false proposition. Stage III is reached only when a suspected-false diagnosis reaches the formalization layer. The semantic audit then compares the formalization with the original proposition. A repair is attempted only if the audit confirms a concrete discrepancy. Otherwise, the formalization is retained and the failure is handled at the proof level.

Figure 1 shows this loop. The system receives a natural-language proposition without a proof, formalizes it, validates the statement, and discovers a proof route through recursive search. If search fails, it diagnoses the failed path and may propagate a suspected-false judgment toward the formalization. The formalization is then audited against the original proposition. If the audit confirms a discrepancy, the system repairs and revalidates the formalization before proof search resumes.

This audit gate remains fixed throughout the loop: proof evidence may guide the audit, but only a proposition-grounded discrepancy permits repair.

4.1 Stage I: Autoformalization and Validation

Given a natural-language problem P, the formalization module generates a Lean formalization F using a structured, compilable template that separates mathematical objects, constraints, auxiliary definitions, and the target proposition. The generated formalization is then validated by two checks.

Lean verifies that F is syntactically and elaborationally valid. The semantic self checker then compares the natural-language problem with a natural-language rendering of the Lean formalization. If either check fails, the formalization module makes local corrections until the retry budget is exhausted.

The output is a system-accepted initial formalization. Stage I acceptance is only an entry condition for proof search. It is not end-to-end success. We call this online component the *semantic self checker*; the held-out evaluators used in the reported results are *semantic validators*.

4.2 Stage II: Proof Search and Failure Propagation

The system-accepted formalization is passed to a Lean prover that follows a recursive decomposition strategy. Because no natural-language proof is supplied, the system generates every decomposition and intermediate lemma during search. For each goal, the prover begins with a direct proof attempt by querying Lean for goal states and type information and searching Mathlib for relevant lemmas. If that attempt fails, it proposes candidate sub-lemmas and tries to prove accepted candidates recursively.

[1]The complete control flow appears in Supplementary Algorithm 1; implementation settings appear in Supplementary Section C.

Each failed node leads to one of three operational actions. The prover may continue locally with a new proof strategy. It may reject the current decomposition and try another one. Or it may propagate a suspected-false judgment to the parent. These are heuristic controller decisions, not logical certificates.

4.3 Pre-Pruning Module

A proposed decomposition that compiles in Lean must still pass the unified pre-pruning module. If it passes, it is attached to the proof tree and expanded. If it fails, it is retained only as historical context for later candidates and is not searched. The module inspects each child lemma along three axes:

- **Truth check.** Whether the sub-lemma is plausibly true under its hypotheses, so that obviously false or unsupported claims are pruned before any proof budget is spent on them.
- **Circularity check.** Whether the sub-lemma and the lemma being proved form a circular argument—for example, proving the child would require the parent or one of its ancestors as a fact.
- **Knowledge accumulation.** A retained record of prior failures—pruned proposals, dead-end decompositions, and recurring error patterns—lets the module reject new proposals that repeat or closely resemble a path already known to fail, so the search does not repeat the same mistake.

The module uses a $2{+}1$ vote. The first two judgments are independent calls to the setting's base model with Lean-environment tools; a third call to the same model is requested only if they disagree. A child is admitted only if no axis rejects it.

4.4 Heuristic Falsity Diagnosis Across Layers

Suppose a child proposition cannot be proved and its decompositions are rejected. The parent may have chosen a bad child, or the parent's own statement may be wrong under its hypotheses. The diagnosis module revisits the parent with the child report, failed attempts, Lean feedback, and ancestor context. The same reasoning repeats at each higher layer.

It returns one of three operational categories. *Prover limitation*: the proposition is not judged wrong, but the prover cannot find a proof or workable strategy, so it retries locally with a new idea. *Local decomposition error*: a child of the current decomposition is judged false, unsuitable, or not progressive, so the child is discarded and the parent is re-decomposed. *Suspected false*: the controller judges the current proposition possibly false under its hypotheses, and the judgment travels upward because its source may be an ancestor statement.

Prover limitation and Local decomposition error trigger local proof-level handling. A Suspected false judgment moves one layer upward. When it reaches the formalization, the system hands the accumulated audit focus to Stage III. No heuristic judgment is treated as a proof of falsity.

4.5 Stage III: Proof-Informed Re-Formalization

Stage III assesses whether the formalization diverges from the original proposition. Jointly regenerating the statement and proof from a failure signal could weaken the target rather than repair it. Stage III therefore separates proof diagnosis, semantic audit, and repair, and treats the natural-language proposition as the semantic authority.

The semantic self checker begins by comparing the current Lean formalization with the original proposition. Following LoC-Decomp (Shi et al., 2026), it separates the formalization into semantic sections: mathematical objects and types, auxiliary functions or predicates, constraints, and the final target. It aligns each section with its natural-language counterpart instead of judging the file as one block. The propagated diagnosis is only auxiliary evidence about where a mismatch may occur. The checker must report a concrete discrepancy grounded in the original proposition. We call such a system-confirmed discrepancy an *operational formalization error*. If no discrepancy is found, the formalization is retained and the failure returns to proof-level handling.

Only an audit-confirmed discrepancy triggers targeted repair. The repair step changes the formalization, not the proof. The revised candidate then enters an iterative validation loop. The semantic self checker compares it with the original proposition, and Lean checks its syntax and elaboration. A failed check causes another targeted correction. A new root proof tree starts for the revised formalization only after a candidate passes both checks.

5 EXPERIMENTS

5.1 DATASETS AND CROSS-MODEL SETTINGS

We evaluate on miniF2F (Zheng et al., 2022) and FormalMath500, a formalized 387-problem subset of MATH-500 (Lightman et al., 2024; Liu et al., 2026; Hendrycks et al., 2021). We use DeepSeek V4 Flash and MiMo v2.5 as base models (Xu et al., 2026; Xiaomi MiMo Team, 2026). For answer-seeking FormalMath500 samples, the supplied answer is included in the natural-language proposition. Only this informal proposition is supplied; reference formalizations and proofs are withheld. Eligibility and denominator details appear in Supplementary Section A.2.

5.2 EVALUATION PROTOCOL

Every reported success requires a Lean-kernel-checked proof. We reject `sorry`, `admit`, `sorryAx`, newly introduced axioms, and other untrusted declarations. ASCC-1 is the online self checker; fixed offline ASCC-3-MV and BEq provide post hoc semantic validation of final formalizations and validated end-to-end outcomes.

All policies use the same pre-intervention execution, current formalization, prover, and endpoint. A single non-resetting budget $B^* = 2 \times 10^6$ completion tokens covers proof search, diagnosis, audit, repair, and validation. This ceiling is a maximum for the combined proving and semantic-control process, rather than a semantic-checking budget; 78.2–89.6% of eligible trajectories finish within 0.25M tokens, consistent with the resource scale used by recursive proof systems such as Hilbert (Varambally et al., 2026). Detailed validator configurations, endpoint rules, token accounting, and statistical tests appear in Supplementary Section A and Supplementary Section A.4.

5.3 BASELINES AND VARIANTS

We compare three policies. *Fixed* keeps the initial formalization unchanged and spends the budget on proof search. *Signal-only* runs the same audit-and-revision loop but replaces the prover diagnosis with the fixed string *"Proof failed under budget." Full* passes the concrete diagnosis generated during proof search to the audit. The policies share the same initial formalization, pre-intervention execution, audit and repair procedure, revalidation, prover, history handling, and global budget. The *Full–Signal-only* comparison tests whether problem-specific proof-failure information provides value beyond the occurrence of proof failure; these paired contrasts are descriptive because the branches share a prefix. Exact trigger, branching, and budget rules appear in Supplementary Section A.4.

5.4 RQ1: HOW LARGE IS THE OPERATIONAL SEMANTIC–PROOF CLOSURE GAP?

We measure the closure gap under *Fixed*, the open-loop baseline that never revisits the formalization. In each setting, it is the all-input rate of initial formalizations that pass the base model's ASCC-1 self checker but remain unproved by $\mathrm{Prover}(M_s)$ at the endpoint. Table 1 reports this within-system quantity. The closure gap is positive in all four settings: 8.4 and 6.5 pp for DeepSeek V4 Flash, and 14.8 and 14.7 pp for MiMo v2.5. The values are larger for MiMo on both datasets because more self-accepted formalizations remain unproved. Self-acceptance is uniformly high (97.5–99.7%), so the gap counts proof failures among self-accepted formalizations rather than checker rejections. The last two columns apply the fixed offline validators to the gap formalizations. Together, the results suggest that the gap is heterogeneous. Some cases remain semantically plausible under an independent LLM judgment, whereas a smaller subset passes BEq's bidirectional equivalence check against the withheld reference formalization.

Setting	Self-accepted	Proved	Closure gap	ASCC in gap	BEq in gap
DSV4Flash–miniF2F	97.5%	89.1%	8.4	78.0%	29.3%
DSV4Flash–FormalMath	99.0%	92.5%	6.5	72.0%	12.0%
MiMov2.5–miniF2F	97.5%	82.8%	14.8	59.7%	20.8%
MiMov2.5–FormalMath	99.7%	85.0%	14.7	56.1%	15.8%

Table 1: Operational semantic–proof closure gap under *Fixed*. *Self-accepted* is the all-input ASCC-1 acceptance rate; *Proved* is the all-input rate that is both self-accepted and kernel-proved; their difference is the closure gap. *ASCC in gap* and *BEq in gap* are the two offline validators' conditional pass rates among the gap formalizations (self-accepted but unproved). They are reported as post hoc semantic assessments and are not part of the gap definition.

5.5 RQ2: Does the Closed Loop Improve Operational Closure?

RQ2 compares *Full* with *Fixed* over every dataset input. Within each setting, we first ask whether *Full* leaves fewer formalizations outside semantic–proof closure. We then evaluate both policies with the same fixed offline ASCC-3-MV and BEq validators. These columns help assess whether more proofs come with aggregate semantic degradation.

For setting s, the reported within-system reduction is

$$\Delta\mathrm{CGAP}_s = \mathrm{CGAP}_{\mathrm{fixed},s} - \mathrm{CGAP}_{\mathrm{full},s}.$$

A positive value means that the closed loop improves budgeted closure within that setting. The system-level claim is supported when this reduction accompanies positive externally validated ASCC+Proof and BEq+Proof E2E effects and nonnegative external final-formalization semantic effects.

Table 2 reports these effects for all four settings, with problem-level paired-bootstrap 95% confidence intervals for the two validated E2E outcomes. The within-system closure-gap reduction is positive throughout (4.1–9.3 pp, largest for MiMo), and the externally validated ASCC+Proof and BEq+Proof gains are positive with intervals excluding zero (ASCC+Proof 3.3–7.0 pp, BEq+Proof 2.1–7.0 pp). The final-formalization semantic effects are nonnegative—BEq strictly positive throughout and ASCC nonnegative. Across these four settings, we observe the claimed direction of improvement without an observed decrease in aggregate semantic pass rates. The single-annotator audit does not observe a harmful transition under Full among the five initially faithful root-reaching cases; the small denominator means rare harms remain possible. Detailed paired tests appear in Supplementary Table 9.

	Within system	External Semantic gain		External E2E gain	
Setting	Gap reduction	ASCC	BEq	ASCC+Proof	BEq+Proof
DSV4F–miniF2F	4.7	0.0	2.5	3.3 [1.8, 4.9]	2.9 [1.4, 4.5]
DSV4F–FormalMath	4.1	0.5	1.8	3.6 [1.8, 5.7]	2.1 [0.8, 3.6]
MiMo–miniF2F	7.4	2.9	4.7	6.4 [4.3, 8.6]	4.9 [3.1, 7.0]
MiMo–FormalMath	9.3	2.6	6.5	7.0 [4.7, 9.6]	7.0 [4.7, 9.6]

Table 2: All-input *Full*-minus-*Fixed* effects in percentage points. Gap reduction is oriented as $\mathrm{CGAP}_{\mathrm{Fixed}} - \mathrm{CGAP}_{\mathrm{Full}}$ so that positive values consistently favor *Full*. External Semantic gain is the change in final-formalization semantic pass rate measured by the fixed offline ASCC-3-MV and BEq validators; it does not require proof completion. Brackets in the two E2E columns give problem-level paired-bootstrap 95% confidence intervals. The exact and Holm-adjusted McNemar p-values are moved to Supplementary Table 9, where Exact p is the two-sided exact value and Holm p is the step-down family-wise correction across the eight E2E comparisons. Stage I failures remain in every denominator.

Table 3 decomposes the loop's end-to-end improvement across the three policies. The rates follow *Fixed* $\leq$ *Signal-only* $\leq$ *Full* in every setting and under both validators. *Signal-only* already improves on *Fixed*. *Full* is higher in seven of the eight setting–validator comparisons and tied in one, so it is best or tied in all eight comparisons. The *Full–Signal-only* contrast tests whether the concrete model-generated diagnosis provides additional information beyond the uninformative failure signal.

The observed pattern is consistent with such information helping the model locate semantic vulnerabilities, although each contrast uses one unseeded trajectory and is therefore descriptive rather than causal. Figure 2 shows the same ordering across the budget grid.

We do not interpret proof completion as a semantic certificate. The two external validators measure complementary notions of consistency, so their agreement is an additional empirical consistency signal when a formalization ground truth is unavailable, rather than a reliability guarantee; a residual successful-deviation blind spot remains.

	Fixed		*Signal-only*		*Full*	
Setting	ASCC+P	BEq+P	ASCC+P	BEq+P	ASCC+P	BEq+P
DSV4F–miniF2F	82.6%	78.5%	85.5%	80.7%	85.9%	81.4%
DSV4F–FormalMath	83.2%	76.7%	86.0%	78.8%	86.8%	78.8%
MiMo–miniF2F	73.4%	71.9%	77.7%	76.0%	79.7%	76.8%
MiMo–FormalMath	70.5%	73.1%	74.9%	77.3%	77.5%	80.1%

Table 3: End-to-end success rates for the three policies across the four settings. Underlines mark the best result for each setting and validator; ties share the same rank. Here, ASCC+P denotes ASCC-3-MV validation plus a kernel-checked proof, and BEq+P denotes BEq validation plus a kernel-checked proof.

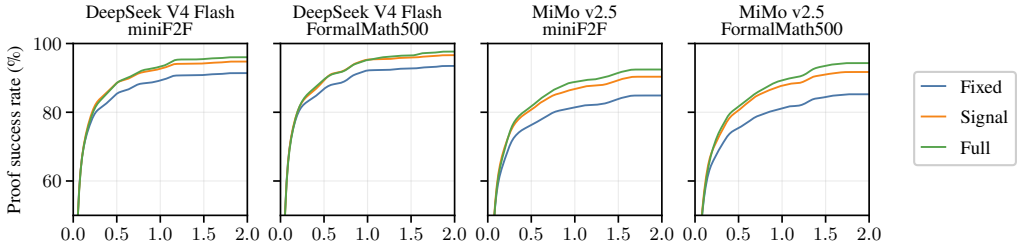


Figure 2: Proof success rate versus cumulative completion-token budget across the four settings. Each panel compares the *Fixed*, *Signal-only* (labeled "Signal"), and *Full* policies, conditional on Stage I acceptance; Stage I failures are excluded from these curves. *Full* consistently attains the highest rate, and *Signal-only* exceeds *Fixed*.

Beyond the within-system loop, we benchmark the formalization method against the external Mathesis baseline (Yu et al., 2026). To control for prover-side differences, we use Mathesis's released autoformalizer and scorer with our prover, and use the same post-initialization proof budget. This matches proof-stage compute after a formalization is available, but not total inference compute: Mathesis candidate-generation and scoring, and our initial-formalization generation, are excluded from Table 4. Both conditions use DeepSeek V4 Flash as the prover. Mathesis generates 6 candidates and selects the highest-scoring compilable candidate whose LeanScore is strictly above 0.6; our Stage I generates one initial candidate (full setup in Supplementary Section A.7). Under this reproduced shared-prover setup, *Full* reaches 81.4% on miniF2F and 78.8% on FormalMath, versus 69.5% and 64.1% for Mathesis—gains of 11.9 and 14.7 pp.

	miniF2F		FormalMath	
	Mathesis	Our system (*Full*)	Mathesis	Our system (*Full*)
BEq+Proof end-to-end success	69.5%	81.4%	64.1%	78.8%

Table 4: End-to-end comparison against a Mathesis-style formalization baseline.

5.6 RQ3: What Is the Joint Effect of the Pre-Pruning Package?

The pre-pruning package filters each proposed decomposition before it enters the proof tree. To assess its joint effect, we disable it inside *Full* with DeepSeek V4 Flash on both datasets. Disabling

the package removes the plausibility, circularity, and recorded-failure-history checks. It also omits failure history from ordinary proof prompts. All other settings remain unchanged; the exact ablation scope is given in Supplementary Table 17.

Metric	miniF2F, w/	miniF2F, w/o	FormalMath, w/	FormalMath, w/o
ASCC+Proof	85.9	83.8	86.8	82.5
BEq+Proof	81.4	79.8	78.8	77.0
Tokens (K)	150	244	133	205

Table 5: Grouped pre-pruning ablation under *Full* with DeepSeek V4 Flash. "w/" and "w/o" denote the full pre-pruning package enabled and disabled. "Tokens (K)" reports mean completion-token use in thousands.

In the reported trajectories, enabling the full pre-pruning package is associated with higher validated end-to-end success and lower average completion-token use (Table 5). Because the grouped ablation changes several checks and history handling together, it does not isolate the contribution of any individual component.

5.7 RQ4: Does the Closed Loop Repair or Weaken Formalizations?

To assess whether root-triggered revisions repair semantic defects or weaken faithful statements, we audit all 137 shared inputs whose *Full* trajectory backtracked to the root, pairing the initial formalization with the final *Full* and *Gate-off* endpoints. To isolate the semantic gate, *Gate-off* starts from the same initial formalizations and shared first proof tree, and uses the same prover and retry mechanism as *Full*; it bypasses the semantic consistency check and uses the proof-failure diagnosis directly to request a revision, retaining only structural and Lean checks. Human labels show that 132/137 inputs were initially inconsistent and 5/137 were initially faithful. *Full* repaired 96/132 initially inconsistent inputs (72.7%) and caused no harmful transition (0/5), whereas *Gate-off* repaired 81/132 (61.4%) and produced 2/5 harmful transitions (40.0%). Thus the semantic gate is associated with both a higher repair yield and fewer harmful changes. Repair-rate point estimates favor *Full* in all four settings; the pooled paired bootstrap estimates an advantage of 11.4 percentage points (95% CI: [0.8, 22.0]). Within this audited root-reaching sample, 96.4% of inputs contained an audited initial discrepancy, which is consistent with a low empirical false-positive share for recursive traceback. Setting-level rates appear in Table 6, and the complete transition breakdown is given in Supplementary Table 11.

		Full		*Gate-off*	
Setting	N	Repair rate	Harm rate	Repair rate	Harm rate
DSV4F–miniF2F	27	81.5% (22/27)	0/0	66.7% (18/27)	0/0
DSV4F–FormalMath	18	88.2% (15/17)	0/1	70.6% (12/17)	1/1
MiMo–miniF2F	50	62.5% (30/48)	0/2	50.0% (24/48)	1/2
MiMo–FormalMath	42	72.5% (29/40)	0/2	67.5% (27/40)	0/2
Total	137	72.7% (96/132)	0/5	61.4% (81/132)	2/5

Table 6: Human audit comparing *Full* with the *Gate-off* control on the same 137 root-reaching inputs. Under *Full*, 129 inputs are modified and 8 are retained unchanged; *Gate-off* revises all 137 inputs. Repair-rate denominators are initially inconsistent inputs, and harm-rate denominators are initially faithful inputs.

6 Conclusions

We instantiate and evaluate a failure-triggered, audit-authorized re-formalization loop for proposition-only Lean proving. Across two models and two datasets, it reduces the operational semantic–proof closure gap and improves externally validated end-to-end success relative to *Fixed*. The improvement in BEq-validated end-to-end success indicates that higher within-system closure can coincide with better external equivalence validation.

7 AI USE STATEMENT

This study utilized Large Language Models (LLMs) in two distinct roles:

1. Language Polishing and Editing: LLMs were employed as an assistive tool to enhance the language quality and clarity of the manuscript. The initial draft was entirely authored by the researcher, after which the LLM provided suggestions to improve sentence fluency, grammar, and academic tone. All conceptual contributions, arguments, and factual assertions originated from the author. The final manuscript was thoroughly reviewed and approved by the author, who takes full responsibility for its content.

2. LLM as a Research Subject: A key aspect of this research involves the evaluation of LLM capabilities. The outputs generated by the model were systematically analyzed as a primary focus of the study. The corresponding methodology is elaborated in the main text.

8 REPRODUCIBILITY STATEMENT

To ensure the reproducibility of our work, we have made our complete codebase, along with the raw experimental results and detailed instructions for setting up the environment, publicly available. All materials have been submitted to an anonymous repository for blind peer review and will be retained upon publication.

REFERENCES

G. Chen, J. Wu, X. Chen, W. X. Zhao, R. Song, C. Li, K. Fan, D. Liu, and M Liao. ReForm: Reflective autoformalization with prospective bounded sequence optimization, 2026. In *Proceedings of the International Conference on Learning Representations*.

J. Chen, W. Chen, J. Du, J. Hu, Z. Jiang, A. Jie, X. Jin, X. Jin, C. Li, W. Shi, et al. Seed-Prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025. *arXiv preprint arXiv:2512.17260*.

J.-H. Chung, Z. Cai, Z. Li, et al. Goedel-Architect: Streamlining formal theorem proving with blueprint generation and refinement, 2026. *arXiv preprint arXiv:2606.06468*.

L. de Moura and S Ullrich. The Lean 4 theorem prover and programming language, 2021. In *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, 625–635.

DeepSeek-AI. DeepSeek-V3 Technical Report, 2024. *arXiv preprint arXiv:2412.19437*.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models, 2024. In *Findings of the Association for Computational Linguistics: ACL 2024*, 3563–3578.

P. Gallardo, M. Raissi, K. Zhang, and S Murthy. Agentic Lean Autoformalization (ALA): An LLM collaborative approach to autoformalization in Lean, 2025. In *NeurIPS 2025 Workshop on Evaluating the Evolving LLM Lifecycle*.

Luyu Gao, Zhuyun Dai, Panupong Pasupat, Anthony Chen, Arun Tejasvi Chaganty, Yicheng Fan, Vincent Y. Zhao, Ni Lao, Hongrae Lee, Da-Cheng Juan, and Kelvin Guu. RARR: Researching and revising what language models say, using language models, 2023. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing, 2024. In *Proceedings of the International Conference on Learning Representations*.

D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J Steinhardt. Measuring mathematical problem solving with the MATH dataset, 2021. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Y. Huang, X. Jin, S. Liang, P. Li, and Y Liu. FormaRL: Enhancing autoformalization with no labeled data, 2025. In *Proceedings of the Second Conference on Language Modeling*.

P. Jana, K. Kale, A. E. Tanriverdi, C. Song, S. Vishwanath, and V Ganesh. ProofBridge: Autoformalization of natural language proofs in Lean via joint embeddings, 2026. In *Proceedings of the International Conference on Learning Representations*.

J. O. J. Leang, Z. Zhao, M. C. Stoian, Q. Xu, H. Li, W. Li, S. B. Cohen, and E Giunchiglia. Pythagoras-Prover: Advancing efficient formal proving via augmented lean formalisation, 2026. *arXiv preprint arXiv:2606.12594*.

Z. Li, Y. Wu, Z. Li, X. Wei, X. Zhang, F. Yang, and X Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency, 2024. In *Advances in Neural Information Processing Systems*, volume 37, 53598–53625.

H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K Cobbe. Let's verify step by step, 2024. In *Proceedings of the International Conference on Learning Representations*.

Y. Lin, S. Tang, B. Lyu, J. Wu, H. Lin, K. Yang, J. Li, M. Xia, D. Chen, S. Arora, and C Jin. Goedel-Prover: A frontier model for open-source automated theorem proving, 2025. In *Conference on Language Modeling*.

Y. Lin, S. Tang, B. Lyu, Z. Yang, J.-H. Chung, H. Zhao, L. Jiang, Y. Geng, J. Ge, J. Sun, J. Wu, J. Gesi, X. Lu, D. Acuna, K. Yang, H. Lin, Y. Choi, D. Chen, S. Arora, and C Jin. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2026. In *Proceedings of the International Conference on Learning Representations*.

Q. Liu, X. Zheng, X. Lu, Q. Cao, and J. Yan. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach, 2025a. In *Proceedings of the International Conference on Learning Representations*.

Q. Liu, X. Zheng, R. Xia, Q. Cao, and J. Yan. Bootstrapping hierarchical autoregressive formal reasoner with chain-of-proxy-autoformalization, 2025b. In *Advances in Neural Information Processing Systems*, volume 38, 20206–20251.

Q. Liu, X. Zheng, R. Xia, X. Qi, Q. Cao, and J Yan. Beyond theorem proving: Formulation, framework and benchmark for formal problem-solving, 2026. In *Proceedings of the International Conference on Machine Learning*.

J. Lu, Y. Wan, Z. Liu, Y. Huang, J. Xiong, C. Liu, J. Shen, H. Jin, J. Zhang, H. Wang, Z. Yang, J. Tang, and Z Guo. Process-driven autoformalization in Lean 4, 2024. *arXiv preprint arXiv:2406.01940*.

J. Lu, Y. Wan, Y. Huang, J. Xiong, Z. Liu, and Z. Guo. FormalAlign: Automated alignment evaluation for autoformalization, 2025a. In *Proceedings of the International Conference on Learning Representations*.

W. Lu, L. Du, S. Li, K. Weng, H. Sun, H. Liu, M. Yu, T. Zhang, and G. Yu. Automated formalization via conceptual retrieval-augmented LLMs, 2025b. In *Proceedings of the International Conference on Learning Representations*, 2026; *arXiv preprint arXiv:2508.06931*.

M. J. Min, Y. Gao, W. Sy, Z. Li, X. Si, and O Bastani. Divide and abstract: Autoformalization via decomposition and abstraction learning, 2026. In *Proceedings of the International Conference on Learning Representations*.

N. I. S. Mohammad and T Sheikh. The faithfulness gap: Certifying semantic equivalence between natural-language and formal mathematical statements, 2026. In *6th Muslims in ML Workshop at the 43rd International Conference on Machine Learning*.

T. Nipkow, L. C. Paulson, and M Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, 2002. Springer.

A. Ospanov, F. Farnia, and R Mohit. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning, 2025a. In *Advances in Neural Information Processing Systems*, volume 38, 41599–41633.

A. Ospanov, F. Farnia, and R Mohit. miniF2F-Lean revisited: Reviewing limitations and charting a path forward, 2025b. In *Advances in Neural Information Processing Systems*, volume 38, 79964–80002.

S Pathak. GFLean: An autoformalisation framework for Lean via GF, 2024. *arXiv preprint arXiv:2404.01234*.

J. Shi, Z. Zhang, B. Ma, S. Zhao, Y. Yuan, and G Wang. LoC-Decomp: LLM autoformalization via logical concept decomposition and iterative feedback correction, 2026. In *Proceedings of the International Conference on Learning Representations*.

A. Singh, A. Fry, A. Perelman, A. Tart, A. Ganesh, A. El-Kishky, A. McLaughlin, A. Low, A. J. Ostrow, A. Ananthram, et al. OpenAI GPT-5 system card, 2025. *arXiv preprint arXiv:2601.03267*.

S. Sonoda, S. Akiyama, and Y Uezato. Why agentic theorem prover works: A statistical provability theory of mathematical reasoning models, 2026. In *Proceedings of the Forty-third International Conference on Machine Learning*.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2023.

G. Tsoukalas, J. Lee, J. Jennings, J. Xin, M. Ding, M. Jennings, A. Thakur, and S Chaudhuri. PutnamBench: Evaluating neural theorem-provers on the Putnam Mathematical Competition, 2024. In *Advances in Neural Information Processing Systems*, volume 37.

S. Varambally, T. Voice, Y. Sun, Z. Chen, R. Yu, and K Ye. Hilbert: Recursively building formal proofs with informal reasoning, 2026. In *Proceedings of the International Conference on Learning Representations*.

H. Wang, M. Unsal, X. Lin, M. Baksys, J. Liu, M. D. Santos, F. Sung, M. Vinyes, Z. Ying, Z. Zhu, et al. Kimina-Prover preview: Towards large formal reasoning models with reinforcement learning, 2025. *arXiv preprint arXiv:2504.11354*.

Z. Wang, W. Ma, Z. Ming, G. Zhang, K. Yuan, and Z Wen. M2F: Automated formalization of mathematical literature at scale, 2026. In *3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents*.

Y. Wu, A. Q. Jiang, W. Li, M. N. Rabe, C. Staats, M. Jamnik, and C Szegedy. Autoformalization with large language models, 2022. In *Advances in Neural Information Processing Systems*, volume 35, 32353–32368.

Y. Wu, D. Huang, R. Wan, Y. Peng, S. Shang, C. Cao, L. Qi, R. Zhang, X. Zhang, Z. Du, et al. StepFun-Formalizer: Unlocking the autoformalization potential of LLMs through knowledge-reasoning fusion, 2026. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, 33980–33988.

Z. Xiao, H. Wang, M.-H. Guo, S.-M. Hu, and S.-T Yau. Editable proof sketch for automated theorem proving, 2026. In *Proceedings of the International Conference on Machine Learning*.

Xiaomi MiMo Team. MiMo-V2.5, 2026. https://huggingface.co/collections/XiaomiMiMo/mimo-v25.

H. Xin, Z. Z. Ren, J. Song, Z. Shao, W. Zhao, H. Wang, B. Liu, L. Zhang, X. Lu, Q. Du, W. Gao, H. Zhang, Q. Zhu, D. Yang, Z. Gou, Z. F. Wu, F. Luo, and C Ruan. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search, 2025. In *Proceedings of the International Conference on Learning Representations*.

R. Xin, Z. Zheng, Y. Nie, K. Yuan, and X Xiao. Scaling up multi-turn off-policy RL and multi-agent tree search for LLM step-provers, 2026. In *Proceedings of the International Conference on Machine Learning*.

A. Xu, B. Lin, B. Xue, B. Wang, B. Xu, B. Wu, B. Zhang, C. Lin, C. Dong, C. Ling, et al. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. *arXiv preprint arXiv:2606.19348*.

X. Yu, J. Zhong, Z. Feng, P. Zhai, R. Yousefzadeh, W. C. Ng, H. Liu, Z. Shou, J. Xiong, Y. Zhou, et al. Mathesis: Towards formal theorem proving from natural languages, 2026. In *Proceedings of the International Conference on Learning Representations*.

J. Zhang, C. Zhong, H. Xu, Q. Li, and Y Zhou. KELPS: A framework for verified multi-language autoformalization via semantic-syntactic alignment, 2025. *arXiv preprint arXiv:2507.08665*.

L. Zhang, M. Valentino, and A Freitas. Monotonic reference-free refinement for autoformalization, 2026. *arXiv preprint arXiv:2601.23166*.

K. Zheng, J. M. Han, and S Polu. miniF2F: a cross-system benchmark for formal olympiad-level mathematics, 2022. In *Proceedings of the International Conference on Learning Representations*.

J. P. Zhou, C. Staats, W. Li, C. Szegedy, K. Q. Weinberger, and Y Wu. Don't trust: Verify—grounding LLM quantitative reasoning with autoformalization, 2024. In *Proceedings of the International Conference on Learning Representations*.

SUPPLEMENTARY MATERIAL: PROOF-SEARCH FEEDBACK FOR SOURCE-GROUNDED RE-FORMALIZATION

A DETAILED EXPERIMENTAL PROTOCOL

A.1 TASK AND SUCCESS DEFINITIONS

Given only a natural-language problem P, an end-to-end system must (a) produce a Lean formalization F that faithfully expresses P, and (b) construct a Lean-kernel-checked proof of F. The first condition is semantic; the second is operational. Neither alone suffices: a faithful statement without a proof is incomplete, and a proof of a distorted statement is irrelevant.

The failure mode studied in this paper occurs when the system's semantic checker accepts F but the same system's prover cannot produce a proof under a fixed token budget. Some such cases reflect genuine prover limitations; others may reflect a mismatch between F and P that the self checker missed but failed proof attempts reveal. The controller must distinguish these cases without weakening the theorem.

The online semantic self checker S_M, instantiated with the trajectory's base model M, screens formalizations during the run and defines that setting's acceptance boundary. Fixed offline validators ASCC-3-MV and BEq never enter the loop; they provide post hoc assessments of final-formalization fidelity. Stage I admits a formalization to proof search only after Lean validation and ASCC-1 acceptance. This gate reduces malformed targets reaching the prover, but acceptance by the same model is not a guarantee of semantic equivalence: a true-but-unfaithful formalization may pass ASCC-1 and receive a proof.

For a natural-language problem P, the system receives no natural-language proof, reference formalization, or formal proof. A formalization F must compile in Lean and pass the online semantic self checker before it enters proof search. An end-to-end run succeeds under a held-out semantic validator V only when its final statement is accepted by V and it has a kernel-checked proof:

$$\mathbf{1}[V(F_{i,\mathrm{end}}) = 1 \wedge \mathrm{Proof}(F_{i,\mathrm{end}}) = 1]\,.$$

Overall success is the mean of this indicator over every dataset input; Stage I failures count as unsuccessful. Proofs using `sorry`, additional axioms, or other unsound shortcuts are rejected.

A run reaches within-system semantic–proof closure when the self checker accepts the final formalization and the same model-instantiated prover delivers a kernel-checked proof. Let a be the all-input self-acceptance rate and c the all-input closure rate. The operational semantic–proof closure gap is

$$\mathrm{CGAP} = a - c = \frac{|\mathrm{Self} \setminus \mathrm{Proved}|}{N}.$$

It is comparable only within the same prover, self checker, Lean environment, and budget. Offline ASCC-3-MV and BEq are not part of this definition; they test whether closing the gap also improves final semantic quality.

A.2 Datasets, Eligibility, and Cross-Model Settings

miniF2F is the official 488-problem Olympiad-level benchmark (Zheng et al., 2022). FormalMath500 is the 387-problem formalized subset of MATH-500 released with the cited benchmark paper (Lightman et al., 2024; Liu et al., 2026); we retain that release's sample identifiers and its Algebra, Intermediate Algebra, Number Theory, Prealgebra, and Precalculus coverage.

For answer-seeking problems $P(a)$, we combine the problem with its supplied answer $\bar{a}$ to form the complete proposition $P(\bar{a})$. This removes answer discovery from the task and is held fixed across policies. The system receives only this informal proposition; neither the reference Lean statement nor its proof is exposed during generation, proof search, diagnosis, or re-formalization.

A problem enters a paired policy-effect comparison only when Stage I produces a syntax- and semantics-valid F_0. We freeze this eligible set before branching and use it for *Fixed*, *Signal*, and *Full* in every setting. Different models may therefore have different eligible denominators, and all policy effects are paired within a setting rather than pooled into a common leaderboard. Overall end-to-end success is computed separately on every dataset input; Stage I failures remain in that denominator and count as unsuccessful. Stage I permits at most five correction rounds when the generated Lean file fails validation.

The primary DeepSeek V4 Flash–miniF2F setting uses endpoint $B^* = 2 \times 10^6$ completion tokens. For every setting, the complete candidate grid $\mathcal{G}_s = \{B_{s,1}, \ldots, B_{s,K_s}\}$ and its maximum endpoint $B_s^* = B_{s,K_s}$ are frozen from the pre-run compute schedule before any *Fixed*, *Full*, or *Signal* outcome is inspected; no observed success rate or policy contrast is used to choose an endpoint.

A.3 Semantic Evaluators

We use two kinds of semantic evaluator for different purposes. A *semantic self checker* is online, runs inside the loop on the trajectory's own base model, and defines that setting's semantic-acceptance boundary for the operational semantic–proof closure gap. A run reaches within-system semantic–proof closure when this checker accepts the final formalization and the same model-instantiated prover produces a kernel-checked proof. A *semantic validator* is offline, held fixed across settings, and never fed back into the trajectory; it evaluates whether improved operational closure also improves final semantic consistency and validated end-to-end success.

Semantic validators (offline). The ASCC validator decomposes and back-translates a generated formalization, merges the resulting semantic description, and compares it with the informal proposition (Shi et al., 2026). ASCC-3 repeats this judgment three times. ASCC-3-MV accepts when at least two judgments accept; its published calibration against human labels reports precision 0.90, recall 0.82, and F1 0.86. We recompute every reported ASCC outcome after the trajectory with `Pro/deepseek-ai/DeepSeek-V3` (DeepSeek-AI, 2024), the fixed evaluator used by LoC-Decomp. It is distinct from both trajectory backends and is held fixed across all four settings. Each vote runs with thinking disabled; neither a seed nor temperature/top-p is sent, so sampling uses the provider defaults. A vote of `not_pass` or `invalid` is rejecting, and at least two `pass` votes are required. The second validator, BEq, attempts to prove the single Lean theorem $F \leftrightarrow F_{\text{ref}}$ (Liu et al., 2025a). It uses `deepseek/deepseek-v4-pro` in direct-proof mode, one root attempt, no completion-token cutoff, at most 20 proof-action attempts and 20 environment-query calls, a 180-second API timeout, and a 120-second Lean-RPC timeout. There is no hand-written tactic whitelist: the model may construct a Mathlib proof, but `sorry`, `admit`, new axioms, resource-guard overrides, and proofs that bypass either implication direction are rejected. A wrapper/elaboration failure or a completed search without an accepted bidirectional proof is negative; infrastructure failures are resumed rather than silently scored. Both validators are offline and are never fed back to the system.

For ASCC, the cache key is the SHA-256 of canonical JSON containing the stable sample ID, exact informal proposition, and exact Lean formalization. For BEq, imports are removed before the candidate and reference are placed in separate namespaces; the key is the SHA-256 of that exact wrapper plus the equivalence theorem. Thus byte-identical endpoints reuse the same offline result

Model	Dataset	Both pass	ASCC only	BEq only	Both fail	Agreement
DeepSeek V4 Flash	miniF2F	380	55	27	14	82.8%
DeepSeek V4 Flash	FormalMath500	279	63	28	13	76.2%
MiMo v2.5	miniF2F	344	71	45	16	75.6%
MiMo v2.5	FormalMath500	266	49	51	20	74.1%

Table 7: ASCC-3-MV–BEq agreement on *Full* final formalizations among eligible inputs. Counts are reported independently of proof completion; Agreement is the fraction in the Both-pass and Both-fail cells.

across policies, while no undocumented statement normalization is applied. The compact artifact releases one ASCC-3-MV endpoint summary per primary setting in `results/ascc3_mv/`. Each summary retains the three published policies' per-input final decision, three vote labels, evaluation key, formalization hash, proof outcome, and aggregate counts. It also retains the evaluator fields used in the correction audit. Raw provider responses and materialized BEq wrappers are not distributed.

Semantic self checker (online). The semantic self checker is ASCC-1, run online with the same base model as the corresponding DeepSeek V4 Flash or MiMo v2.5 trajectory. It filters proposed formalizations in Stage I and performs the proof-informed audit and post-repair validation in Stage III. For setting s, the operational semantic–proof closure gap is the all-input rate of final formalizations accepted by this checker but left unproved by the same model-instantiated prover at budget B_s. It is the portion of the setting's self-accepted mass that remains outside budgeted semantic–proof closure. Because ASCC-1 shares the trajectory's model, this quantity describes the setting's own acceptance–proof boundary. The offline validators above are therefore used for all reported final semantic-quality and validated-E2E outcomes.

ASCC end-to-end success requires fresh ASCC-3-MV acceptance and a kernel-checked proof. Independently, BEq end-to-end success requires positive BEq and a kernel-checked proof, without an ASCC prerequisite. Applying the two validators separately to every final formalization without requiring a proof yields the corresponding final-formalization semantic pass rates.

Because the two validators capture different notions, we report their agreement rather than treating one as ground truth for the other. Table 7 cross-tabulates their judgments on *Full* final formalizations. In this sample, some successfully proved formalizations still disagree with at least one independent validator, so proof completion alone is not a semantic certificate; this observation is consistent with a residual successful-deviation blind spot. ASCC-3-MV and BEq are imperfect and measure different notions of consistency, so these are external disagreement rates rather than human-confirmed semantic error rates. The compact artifact does not include a separate release of all sample-level disagreement records.

A.4 Budget Accounting, Shared Prefixes, and Statistics

The post-initialization budget begins after the shared validated F_0 has been produced. The same 2M-token ceiling is used in all four model–dataset settings. It accumulates the provider-reported `usage.completion_tokens` from every successful response in proof search, including the prover's own failure diagnosis, and from re-formalization and syntax and semantic validation of revised candidates. Input tokens, wall-clock time, and a provider request that fails before returning a successful usage record are not charged. Any reasoning tokens are included only insofar as the provider includes them in `completion_tokens`. A successful response that crosses the ceiling is charged and persisted, but its content is withheld from the controller. The budget never resets between root trees, and there is no per-tree token budget. At an audit-eligible termination, the shared prover trajectory has already produced a diagnosis string d_i. *Full* forwards d_i to the semantic audit. *Signal* replaces only this field with the fixed string *"Proof failed under budget."* No additional diagnostic model call is made by either policy. Both then follow the same audit, repair, validation, history, and budget-accounting path. *Fixed* bypasses revision and spends its budget on proof search. Initial formalization cost is reported separately.

Post-initialization cost summaries are conditioned on eligible inputs, whereas overall end-to-end success retains the complete dataset denominator. Input-context tokens and tool execution time are not equated with charged completion tokens.

Each policy may construct at most five root proof trees. A tree terminates when a proof is kernel-checked or the prover actively exhausts its available strategies and emits a terminal diagnosis; only a Suspected false diagnosis that reaches the formalization can trigger the controlled semantic audit, and the full category routing appears with the failure taxonomy in Section C. Either continuation can start another tree only if global budget remains. Global-budget exhaustion instead terminates the run directly: it does not emit the *Signal* message or invoke a further audit, repair, or tree. All policies use the same Lean invocation mechanism.

For the *Fixed*, *Signal*, and *Full* comparison, each problem–model pair remains one physical stochastic execution until the prover emits an audit-eligible terminal record $D^\dagger$. Every earlier root tree, Prover limitation or Local decomposition error judgment, and proof-level continuation is therefore shared. If the shared execution proves the formalization, exhausts the global budget, or uses all root trees before $D^\dagger$ occurs, that common terminal outcome is assigned to every policy and the input never enters the conditional comparison. Otherwise, after charging the shared execution through $D^\dagger$, we copy $\mathcal{C}^\dagger = (P, F_0, \tau^\dagger, D^\dagger, K^\dagger, B^\dagger, r^\dagger)$ into isolated *Fixed*, *Signal*, and *Full* branches. Here $\tau^\dagger$ and $K^\dagger$ contain the complete pre-intervention trace and controller history, $B^\dagger$ is remaining charged budget, and $r^\dagger$ is the number of root trees already used. Thus no policy-specific decoding occurs before the audit-input intervention, and branching creates neither fresh budget nor fresh root-tree capacity.

Trajectories are extended monotonically over $B_1 < \cdots < B_K = B^*$, so the endpoint records checkpointed execution rather than an interpolated run. The main paper reports B_s^*.

The designated run is the primary DeepSeek V4 Flash–miniF2F setting; each setting uses one unseeded trajectory per problem–policy configuration. The API receives no seed parameter. Robustness is assessed across the four model–dataset settings rather than through repeated seeded runs. RQ2 uses paired all-input *Full*-minus-*Fixed* differences in the within-system operational semantic–proof closure gap, defined by the setting's ASCC-1 checker and model-instantiated prover. A positive reduction means that more inputs reach budgeted semantic–proof closure at the endpoint. Its semantic-quality and validated-E2E effects are evaluated separately with fixed offline ASCC-3-MV and BEq. We also report charged completion tokens, root trees, and revisions per input and per successful pair, including failed trajectories in cost numerators.

For the two validated E2E outcomes, uncertainty intervals are percentile 95% intervals from 10,000 problem-level paired-bootstrap resamples over the complete dataset denominator. The implementation derives a deterministic bootstrap RNG state from the SHA-256 of the named contrast and ordered paired outcome vector; this analysis seed does not alter model sampling. Significance is assessed with a two-sided exact McNemar test on each paired binary E2E endpoint. A *Full* win denotes an input that satisfies the named semantic validator and receives a kernel-checked proof only under *Full*; a *Fixed* win is defined conversely. We apply Holm's step-down correction jointly to the eight confirmatory E2E tests (four settings by two validators). Closure-gap reduction and final-formalization semantic pass rates are reported descriptively rather than entered into this method-comparison test family.

For the RQ4 human audit, the resampling unit is one complete paired root-reaching record containing the initial, *Full*, and *Gate-off* audit labels. Each bootstrap replicate samples 137 records with replacement, preserving the within-record pairing. In each replicate, the repair rate is recomputed with the number of initially inconsistent records as its denominator, and the harm rate is recomputed with the number of initially faithful records as its denominator; the reported contrasts are *Full* repair minus *Gate-off* repair and *Gate-off* harm minus *Full* harm. Replicates with a zero denominator for a contrast are omitted for that contrast. We use 10,000 replicates and report the percentile interval from the 2.5th and 97.5th percentiles. This procedure yields the RQ4 intervals reported in the main text, while accounting for the small initially faithful stratum (5/137).

A.5 ALL-INPUT *Full–Fixed* SYSTEM OUTCOMES

Table 8 provides the raw rates underlying the RQ2 *Full*-minus-*Fixed* effects. Every rate uses the complete dataset denominator. Self-accepted, Closure, and Closure gap use the setting's online ASCC-1 checker and model-instantiated prover. Final ASCC, Final BEq, ASCC+Proof, and BEq+Proof instead use the fixed offline validators.

Model	Dataset	Policy	Self-accepted	Closure	Closure gap	Final ASCC	Final BEq	ASCC+Proof	BEq+Proof
DeepSeek V4 Flash	miniF2F	*Fixed*	97.5%	89.1%	8.4%	89.1%	80.9%	82.6%	78.5%
		Full	97.5%	93.9%	3.7%	89.1%	83.4%	85.9%	81.4%
DeepSeek V4 Flash	FormalMath500	*Fixed*	99.0%	92.5%	6.5%	87.9%	77.5%	83.2%	76.7%
		Full	99.0%	96.6%	2.3%	88.4%	79.3%	86.8%	78.8%
MiMo v2.5	miniF2F	*Fixed*	97.5%	82.8%	14.8%	82.2%	75.0%	73.4%	71.9%
		Full	97.5%	90.2%	7.4%	85.0%	79.7%	79.7%	76.8%
MiMo v2.5	FormalMath500	*Fixed*	99.7%	85.0%	14.7%	78.8%	75.5%	70.5%	73.1%
		Full	99.7%	94.3%	5.4%	81.4%	81.9%	77.5%	80.1%

Table 8: All-input raw *Full* and *Fixed* rates for RQ2. Closure is the rate of formalizations that are both self-accepted and proved; Closure gap is Self-accepted minus Closure. These are endpoint quantities under the stated budget, not asymptotic convergence measures. Final-formalization ASCC and BEq do not require proof completion; their corresponding E2E rates also require a kernel-checked proof. The online quantities define the within-system gap, whereas the offline quantities evaluate its semantic consequences. Stage I failures count as unsuccessful in every column. Effects are computed from unrounded counts, so subtracting two displayed one-decimal rates can differ by 0.1 percentage points.

Setting	Endpoint	*Full* wins	*Fixed* wins	Ties	Exact p	Holm p
DSV4Flash–miniF2F	ASCC+Proof	16	0	472	$< .001$	$< .001$
	BEq+Proof	14	0	474	$< .001$	$< .001$
DSV4Flash–FormalMath	ASCC+Proof	14	0	373	$< .001$	$< .001$
	BEq+Proof	8	0	379	.008	.008
MiMov2.5–miniF2F	ASCC+Proof	31	0	457	$< .001$	$< .001$
	BEq+Proof	24	0	464	$< .001$	$< .001$
MiMov2.5–FormalMath	ASCC+Proof	27	0	360	$< .001$	$< .001$
	BEq+Proof	27	0	360	$< .001$	$< .001$

Table 9: Paired *Full*-versus-*Fixed* inference for the two validated E2E outcomes in the main-paper RQ2 table. Wins count discordant all-input pairs for which only the named policy satisfies both semantic validation and proof completion; ties include both-pass and both-fail pairs. Exact p is the two-sided exact McNemar value. Holm p applies Holm's step-down family-wise correction across the eight E2E comparisons. Stage I failures remain in the denominator and enter as both-fail ties. Values displayed as $< .001$ are retained at full precision for correction.

A.6 Closed-Loop Intervention Funnel

Table 10 opens the *Full* trajectory into the distinct-input transitions that connect a failed proof tree to the final endpoint. All intervention and endpoint columns are computed from *Full*; *Fixed* is used only to identify the BEq status of the shared initial formalization F_0. Each input is counted once in each column, irrespective of how many internal audit or repair attempts occurred.

How to read each column. **Setting** identifies the base-model–dataset combination. **Eligible** is the number of inputs for which Stage I produced a syntax- and semantics-valid initial formalization. **Audit triggered** counts inputs on which a suspected-false diagnosis reached the formalization and invoked the semantic audit at least once; an ordinary proof failure or global-budget exhaustion does not count. **Modified inputs** counts inputs whose accepted final formalization differs from their initial formalization. This defines the modified-input intervention-summary population; it is not the full paired human-audit population, and an audit trigger that retains the current statement is not counted as a modification. **BEq repairs** counts modified inputs whose initial formalization was BEq-negative and whose *Full* endpoint is BEq-positive. **Modified and proved** counts modified inputs that eventually receive a kernel-checked proof, without additionally requiring BEq. **BEq regressions** counts the reverse semantic transition, from a BEq-positive initial formalization to a BEq-negative *Full* endpoint.

Formalization audit is selective: it is triggered for 5.7%, 4.4%, 10.5%, and 10.9% of eligible inputs in table order. Not every trigger changes the statement: the final number of distinct modified inputs is 27, 17, 48, and 37. Among those modified inputs, 14/27 (51.9%), 8/17 (47.1%), 25/48 (52.1%), and 26/37 (70.3%) move from BEq-negative to BEq-positive. The corresponding modified-and-proved

Setting	Eligible	Audit triggered	Modified inputs	BEq repairs	Modified and proved	BEq regressions
DSV4Flash–miniF2F	476	27	27	14	23	2
DSV4Flash–FormalMath	383	18	17	8	16	1
MiMov2.5–miniF2F	476	50	48	25	37	2
MiMov2.5–FormalMath	386	42	37	26	35	1

Table 10: *Full*-policy intervention funnel at the reported endpoint. All entries are distinct-input counts. The corresponding initial-BEq-positive populations for the regression column are 395, 300, 366, and 292 in row order, giving regression rates of 0.5%, 0.3%, 0.5%, and 0.3%.

yields are 23/27 (85.2%), 16/17 (94.1%), 37/48 (77.1%), and 35/37 (94.6%). By contrast, only 2, 1, 2, and 1 initially BEq-positive inputs regress at the *Full* endpoint. The table therefore exposes both sides of the intervention without treating every audit trigger as an actual re-formalization.

A.7 Mathesis Formalization Baseline

Mathesis releases no prover (Yu et al., 2026), so we prove its selected statement with the same DeepSeek V4 Flash prover used for our DeepSeek settings. The proof phase uses the same 2M completion-token budget, five-root limit, local retry limit, and recursive depth. As in our system, statement-generation cost is outside this post-initialization proof budget; Mathesis candidate-generation and scoring tokens are therefore not charged to the 2M proof budget.

We reproduce the released selection pipeline with huawei-ai4math/Mathesis-Autoformalizer-HPO (Hugging Face revision `a24fad1`). It samples six candidates in one request with temperature 0.7, top-p 0.8, top-k 20, repetition penalty 1.1, a 4,096-token output cap, and a 600-second timeout; no seed is set. Each candidate is checked in Lean with a 120-second timeout. Every compilable candidate is evaluated by the released LeanScorer using Pro/deepseek-ai/DeepSeek-V3: one call decomposes the informal statement and a second compares that decomposition with the candidate. Each scorer call is retried at most four times after transient failure, uses a 60-second timeout, and leaves decoding at the provider defaults. The compilable candidate with the highest LeanScore strictly above 0.6 is selected, with the original candidate index breaking ties. An input is unsuccessful if no candidate meets this threshold. The reproduction uses Mathesis repository commit prefix `48c72753b7c7`. The compact artifact includes the reproduction script and LeanScorer in `baselines/mathesis/`, frozen selected inputs in `data/mathesis/`, and the retained endpoint results and BEq summaries.

The selection stage uses `python baselines/mathesis/reproduce_mathesis_pipeline.py --input-json <dataset> --output <checkpoint.jsonl> --resume`. Its selected statements are supplied to `configs/miniF2F_mathesis_deeseek_v4f.json` and `configs/formalMATH500_mathesis_deeseek_v4f.json`. The corresponding manifests, row-level proof endpoint records, and aggregate BEq outputs are included; raw provider logs and intermediate API traces are omitted.

A.8 Human Audit Protocol

The audit is performed by a single annotator and is used to calibrate the automatic semantic evaluations, not as a third independent benchmark metric. Its unit is one shared root-reaching input, represented by the initial formalization and the final *Full* and *Gate-off* endpoints. Thus an input contributes once even if the controller attempted several intermediate revisions, and the eight *Full* endpoints that were retained unchanged remain in the paired analysis. Each packet shows the original natural-language proposition and the three formalizations in randomized method order, with method identities hidden. The annotator does not see ASCC-3-MV results, BEq results, proof outcomes, or policy identity. Labels are assigned under the following criteria:

1. Mathematical objects. The formalized statement must discuss the same mathematical objects as the original problem. Redundant auxiliary objects are permitted, and individual objects may be expressed as sets.

Analysis	Transition	D–miniF2F	D–FormalMath	M–miniF2F	M–FormalMath	Total
Full	I→F	22	15	30	29	96 (70.1%)
	I→I	5	2	18	11	36 (26.3%)
	F→F	0	1	2	2	5 (3.6%)
	F→I	0	0	0	0	0 (0.0%)
Gate-off	I→F	18	12	24	27	81 (59.1%)
	I→I	9	5	24	13	51 (37.2%)
	F→F	0	0	1	2	3 (2.2%)
	F→I	0	1	1	0	2 (1.5%)
Full→*Gate-off*	F→F	16	11	15	23	65 (47.4%)
	F→I	6	5	17	8	36 (26.3%)
	I→F	2	1	10	6	19 (13.9%)
	I→I	3	1	8	5	17 (12.4%)
Per-setting total		27	18	50	42	137

Table 11: Detailed single-annotator audit transitions over the 137 paired root-reaching inputs. Here D denotes DSV4F, M denotes MiMo, I denotes Inconsistent, and F denotes Faithful. The first block compares the initial label with the accepted *Full* endpoint, the second compares the initial label with the ungated endpoint, and the third compares the *Full* endpoint directly with the *Gate-off* endpoint. Percentages in the Total column use the shared denominator 137.

2. Logical constraints. The constraints of the original problem must be correctly expressed: under the stated hypotheses, all object properties and relationships are consistent with the original. Constraint expressions that are not universally valid but hold in the current context are still accepted.

3. Truth-value correspondence. The truth value of the formalized statement must correspond exactly to that of the original problem—for any concrete instantiation of the objects (including a solution in the case of a solve-type problem), the formal statement holds if and only if the original problem is true.

4. Solution form. No additional requirements are imposed on the form of the solution. For example, if the original problem asks for an angle in degrees, a formalization that accepts radians is considered correct as long as the two are practically equivalent. This accommodates Lean 4's limited support for symbolic computation (e.g., requiring simplest-form expressions).

5. Equivalence without loss of generality. A formalization that analyzes representative cases, where the final criterion is satisfied in each case and the general case follows by straightforward deduction, is accepted. Similarly, formulations that rely on equivalent conversions—such as transforming a canonical equation into general form—or that hold only in the specific context even if not generally valid are regarded as faithful without loss of generality.

Only two labels are used. A formalization satisfying all applicable criteria is labeled `pass`; any concrete discrepancy in objects, constraints, or truth-value correspondence is labeled `not_pass`.

In Table 11, *Faithful* corresponds to `pass`, and *Inconsistent* corresponds to `not_pass`. The released audit records contain these labels for all pairs, but do not include free-text discrepancy rationales. The merged audit contains the same 137 root-reaching inputs for both methods: 27, 18, 50, and 42 from DSV4F–miniF2F, DSV4F–FormalMath, MiMo–miniF2F, and MiMo–FormalMath, respectively. It records the initial, accepted *Full*, and root-level *Gate-off* labels for every input. Under *Full*, 129 inputs changed and 8 reached the root but were retained unchanged. The table below reports all transitions, including the paired transition from the *Full* label to the *Gate-off* label.

B Checkpoint Case Studies

We select two complete checkpoints from the primary DeepSeek V4 Flash–miniF2F setting. The first is a *Fixed*-failure/*Full*-success pair; the second is a failed *Full* trajectory in which the audit–repair loop is invoked repeatedly. They are diagnostic examples rather than a representative sample, and the aggregate comparisons remain the basis of the quantitative claims. The listings reproduce all

semantically operative declarations from the checkpoints; common imports, notation, and template comments are omitted.

B.1 Successful Repair: the Missing Radius Domain

Original proposition. Sample `mathd_algebra_107` asks: "Find the radius of the circle with equation $x^2 + 8x + y^2 - 6y = 0$. Show that it is 5."

Initial formalization. Stage I accepted the following root statement. The circle is represented extensionally by a biconditional between its two equations.

```
structure MySystem where
  center_x : ℝ
  center_y : ℝ
  radius : ℝ
  proof_goal : Prop

def circle_equation_constraint (sys : MySystem) : Prop :=
  ∀ x y : ℝ,
    x^2 + 8*x + y^2 - 6*y = 0 ↔
    (x - sys.center_x)^2 +
      (y - sys.center_y)^2 = sys.radius^2

def proof_goal_constraint (sys : MySystem) : Prop :=
  sys.proof_goal = (sys.radius = 5)

def problem_statement : Prop :=
  ∀ sys : MySystem,
    circle_equation_constraint sys ∧
    proof_goal_constraint sys →
    sys.proof_goal
```

Prover diagnosis and semantic audit. The first proof tree terminated with the following checkpoint diagnosis:

> "The theorem is false because we can construct a counterexample with `radius=-5`. The circle equation is equivalent to $(x+4)^2 + (y-3)^2 = 25$, which also matches radius -5 because $(-5)^2 = 25$. The hypotheses hold but the conclusion fails."

Thus the initial universal statement admits a concrete counterexample. This information did help the semantic self checker locate a proposition-grounded mismatch: the informal object is a circle radius, whose domain is nonnegative, but the formal object was an unrestricted real. The audit authorized only that missing domain condition; it did not change the target value.

Repaired formalization. The accepted replacement added `radius_nonneg_constraint` and otherwise retained the initial representation:

```
structure MySystem where
  center_x : ℝ
  center_y : ℝ
  radius : ℝ
  proof_goal : Prop

def circle_equation_constraint (sys : MySystem) : Prop :=
  ∀ x y : ℝ,
    x^2 + 8*x + y^2 - 6*y = 0 ↔
    (x - sys.center_x)^2 +
      (y - sys.center_y)^2 = sys.radius^2

def radius_nonneg_constraint (sys : MySystem) : Prop :=
  0 ≤ sys.radius

def proof_goal_constraint (sys : MySystem) : Prop :=
  sys.proof_goal = (sys.radius = 5)

def problem_statement : Prop :=
  ∀ sys : MySystem,
    circle_equation_constraint sys ∧
    radius_nonneg_constraint sys ∧
    proof_goal_constraint sys →
    sys.proof_goal
```

Trajectory analysis. *Fixed* retained the initial statement for all five root trees. Each tree returned essentially the same negative-radius counterexample, so no amount of proof-level replanning could close the false root. *Full* shared the first failed tree, performed one audit and one accepted repair, and succeeded on its second root tree. The generated direct proof evaluates the biconditional at $(0, 0)$, $(-8, 0)$, and $(0, 6)$; these instances yield the center $(-4, 3)$ and `radius^2 = 25`. The added non-negativity condition then selects `radius = 5` rather than -5. The final proof is kernel-checked, and the repaired statement passes both offline ASCC-3-MV and BEq. This trajectory illustrates the intended boundary: the proof diagnosis supplied a counterexample, but the root changed only after the audit connected that counterexample to an implicit semantic property of "radius."

B.2 Failed Repair: Confusing Unit Values with Amounts

Original proposition. Sample `mathd_algebra_398` states: "On planet Larky, 7 ligs = 4 lags, and 9 lags = 20 lugs. How many ligs are equivalent to 80 lugs? Show that it is 63."

Initial formalization. The accepted root treated the three fields as numerical amounts, imposed `lugs = 80`, and asked for `ligs = 63`:

```
structure MySystem where
  ligs : ℚ
  lags : ℚ
  lugs : ℚ
  proof_goal : Prop

def ligs_lag_constraint (sys : MySystem) : Prop :=
  7 * sys.ligs = 4 * sys.lags

def lags_lug_constraint (sys : MySystem) : Prop :=
  9 * sys.lags = 20 * sys.lugs

def target_lugs_constraint (sys : MySystem) : Prop :=
  sys.lugs = (80 : ℚ)

def proof_goal_constraint (sys : MySystem) : Prop :=
  sys.proof_goal = (sys.ligs = (63 : ℚ))

def problem_statement : Prop :=
  ∀ sys : MySystem,
    ligs_lag_constraint sys ∧
    lags_lug_constraint sys ∧
    target_lugs_constraint sys ∧
    proof_goal_constraint sys →
    sys.proof_goal
```

Prover diagnosis and semantic audit. The first tree returned the following checkpoint diagnosis:

> "`lugs=80` gives `9*lags=1600`, hence `lags=1600/9`; then `7*ligs=6400/9`, hence `ligs=6400/63`, which is not 63. All constraints can hold while `proof_goal` is false, so the root theorem is false."

The diagnosis therefore helped the semantic self checker detect that the accepted root did not express the requested equivalence. This detection was only partially successful, however. The audit initially recommended reversing coefficients in the conversion constraints, rather than recognizing that the fields in those equations denote unit values and that "80 lugs" should appear as a coefficient in the target relation. The subsequent repair calls did not implement a faithful relational representation.

Final attempted formalization. The first three accepted revision outputs changed only whitespace and left the false root semantically unchanged. The fourth replaced the fixed-value premise with an implication inside the goal:

```
structure MySystem where
  ligs : ℚ
  lags : ℚ
  lugs : ℚ
  proof_goal : Prop

def ligs_lag_constraint (sys : MySystem) : Prop :=
  7 * sys.ligs = 4 * sys.lags
```

```
def lags_lug_constraint (sys : MySystem) : Prop :=
  9 * sys.lags = 20 * sys.lugs

def proof_goal_constraint (sys : MySystem) : Prop :=
  sys.proof_goal =
    (sys.lugs = (80 : ℚ) → sys.ligs = (63 : ℚ))

def problem_statement : Prop :=
  ∀ sys : MySystem,
    ligs_lag_constraint sys ∧
    lags_lug_constraint sys ∧
    proof_goal_constraint sys →
    sys.proof_goal
```

Trajectory analysis. This edit still does not follow from the conversion equations: they imply `ligs = (80/63) * lugs`, so setting `lugs = 80` again yields `6400/63`. The fifth root tree rediscovered exactly this counterexample, and the run stopped at the five-root limit rather than at global-budget exhaustion. A faithful representation would express the requested equivalence itself—for example, retain the two conversion relations and prove `63 * ligs = 80 * lugs`—instead of assigning the numeral 80 to a field that denotes a unit value.

The final statement passed offline ASCC-3-MV but did not pass BEq and received no kernel-checked proof. This evaluator disagreement is informative: the prover diagnosis consistently exposed falsity, while both online rechecks and one reference-free offline validator accepted a repair that preserved the same representational confusion. The failure therefore lies after useful diagnostic localization, in converting that evidence into a semantically faithful replacement. It also shows why the system-level results require both proof completion and external semantic evaluation rather than treating an audit-triggered edit as a successful repair.

C Implementation and Reproducibility Details

C.1 Runtime Control Flow and Diagnostic Records

Algorithm 1 describes policy continuation from the shared audit-eligible checkpoint $\mathcal{C}^\dagger$. The shared-prefix, branching, and budget rules that produce $\mathcal{C}^\dagger$ are specified in Section A, so the algorithm shows only the policy-specific audit-input intervention and the subsequent audit, repair, and proof loop. The checkpoint already contains the prover-driven diagnosis string d_i. *Fixed* bypasses revision. *Full* supplies d_i to the audit, whereas *Signal* replaces only d_i with *"Proof failed under budget."* Internally, each root tree stores a recursive list of proof-node attempts. A node record contains its theorem, proposed decomposition, lemma slots and dependencies, direct-proof and decomposition attempts, Lean feedback, accumulated helpful definitions, and terminal result. Only declarations from successfully checked child subtrees are reused as known facts. Failed subtrees are not imported as axioms; instead, their summaries enter the layer history used to pre-prune later proposals. When a Suspected false judgment reaches the formalization, the prover serializes the available failure category, failed goal or lemma, Lean feedback, and propagated explanation into the diagnosis string d_i. This string is part of the terminal prover record, not the output of a separate semantic-diagnosis call. The policy history H contains prior formalizations, feedback strings supplied to the audit, and audit outcomes. The three terminal categories and their routing are tabulated in Table 13.

The implementation persists the raw formalization diagnostic shown in Table 12. We preserve raw runtime event names in released logs and separately normalize them into the three paper-level reporting categories. This avoids implying that an LLM judgment such as `lemma_false` is a kernel proof of falsehood.

The formalization semantic audit emits `consistency_level` $\in \{$`level_1`, `level_2`, `level_3`$\}$ together with lists of concrete `discrepancies` and paired `recommendations`. Level 1 is fully consistent, level 2 is accepted as an equivalent or without-loss-of-generality formulation, and level 3 is inconsistent. Before repair, one separate semantic-self-check verification call uses the setting's same base model and a fixed recheck prompt to compare the complete proposed discrepancy list with P and the separated semantic sections of F. Its completion tokens are charged to the same global budget. Only discrepancy–recommendation

Field	Type	Meaning
`failure_type`	string	Raw terminal runtime event, retained for audit and mapped to a paper-level category only during analysis.
`failure_reason`	string	Model- and Lean-grounded explanation accumulated at the terminal proof node.
`lemmas`	list of Lean declarations	Candidate or current decomposition relevant to the terminal failure.
`failed_lemma`	Lean declaration or empty string	The child proposition whose failure initiated local recomposition or upward attribution.

Table 12: Persisted prover-diagnostic schema serialized into the diagnosis string d_i. The policy-history schema for H_i contains prior formalizations, feedback strings supplied to the audit, and audit outcomes.

pairs accepted by this verification call reach repair; if all are rejected, the audit is reset to level 1 and the formalization is retained.

Algorithm 1 Policy continuation from a shared audit-eligible trigger

Require: checkpoint $\mathcal{C}^\dagger = (P, F_0, \tau^\dagger, D^\dagger, K^\dagger, B^\dagger, r^\dagger)$
Require: policy $\pi \in \{\pi_{\text{fixed}}, \pi_{\text{signal}}, \pi_{\text{full}}\}$; model M; root-tree limit R
$F \leftarrow F_0$; $K \leftarrow K^\dagger$; $D \leftarrow D^\dagger$
$\tau \leftarrow \tau^\dagger$; $B \leftarrow B^\dagger$; $r \leftarrow r^\dagger$
$H \leftarrow \textsc{RootHistory}(K^\dagger)$
while $r \leq R$ **do**
  if the global budget is exhausted **then**
    return $(F, \text{unsuccessful})$
  end if
  if $D.\text{type} = \textsc{SuspectedFalse}$ **and** $\pi \neq \pi_{\text{fixed}}$ **then**
    $d \leftarrow D.\text{diagnosis_string}$ {prover-produced}
    if $\pi = \pi_{\text{signal}}$ **then**
      $d \leftarrow$ "Proof failed under budget."
    end if
    $(a, e) \leftarrow \textsc{SelfCheck-Audit}(P, F; d, H)$
    append (F, d, a) to H
    if a identifies a discrepancy **then**
      $F' \leftarrow \textsc{Repair}(P, F, e)$
      $v \leftarrow \textsc{Validate}(P, F')$ {semantic + Lean}
      if $v = \text{pass}$ **then**
        $F \leftarrow F'$
      end if
    end if
  end if
  if $r = R$ **or** the global budget is exhausted **then**
    return $(F, \text{unsuccessful})$
  end if
  $(z, \tau) \leftarrow \textsc{RecursiveProve}(\text{Prover}(M), F, K)$
  $r \leftarrow r + 1$
  if z is a kernel-checked proof **then**
    return (F, z)
  end if
  if the global budget is exhausted **then**
    return $(F, \text{unsuccessful})$
  end if
  $D \leftarrow \tau.\text{terminal_diagnosis}$
  append (F, D) to K
end while
return $(F, \text{unsuccessful})$

Paper category	Representative runtime events	Controller interpretation
Prover limitation	`unable_to_prove`, `local_retry_exhausted`, `depth_exceeded`	The current proposition is not judged wrong; retry locally or end the current tree.
Local decomposition error	`recompose`, layer-level `decomposition_error`	Reject the failed child/decomposition and regenerate the parent decomposition.
Suspected false	`lemma_false`, `traceback_up`	Propagate the controller's suspected-false judgment upward for re-attribution; this is not a refutation.
Not a proof diagnosis	global budget exhaustion; infrastructure or program error	Terminate directly or retain a resumable checkpoint; do not emit the *Signal* message and do not classify the statement.

Table 13: Normalization from implementation events to the three reporting categories. A formalization error is not in this table because it is recorded only after upward evidence reaches the formalization and the semantic self checker confirms a discrepancy against the informal proposition.

Prompt	Supplied fields	Required action/output
Direct proof and decomposition	Current Lean file, target, Lean goal, helpful definitions, kernel-checked child facts, prior failed strategies, and optional parent plan	Return a complete proof, or direct sub-lemmas used by a parent proof. Proposed declarations are parsed and checked before recursive expansion.
Unified pre-pruning	Candidate and parent propositions, hypotheses, ancestor stack, Lean environment queries, and recorded failure history	Decide whether to retain or prune the candidate based on plausibility, progress, circular dependence, and similarity to failed paths. Two judgments are made; a third is requested only on disagreement.
Layer diagnosis	Current proposition, failed child, failure explanation, decomposition, Lean feedback, and accumulated layer history	Decide whether the parent is suspected false or the failure belongs to its decomposition; produce a concise reason and decomposition advice when local.
Semantic self-check audit	Original proposition, current Lean formalization separated into semantic sections, policy history, and one feedback-string field	Compare objects/types, auxiliary predicates, constraints, and final goal; emit a consistency level, concrete discrepancies, and corresponding recommendations. *Full* fills the field with d_i; *Signal* substitutes the fixed string *"Proof failed under budget."*
Semantic repair	Original proposition, current Lean file, only audit-confirmed discrepancies and recommendations, and the mandatory file template	Return a complete replacement file faithful to the original proposition; recommendations are evidence, not permission to weaken the target.
Syntax repair	Current Lean file and Lean diagnostics	Correct elaboration or syntax only; declaration names, parameters, types, and semantics must remain unchanged.

Table 14: Prompt interfaces and information flow. *Full* and *Signal* use the same semantic-self-check audit, semantic-repair, syntax-repair, and validation templates. Their only intervention difference is the value of the audit's feedback-string field.

C.2 Prompt Interfaces

Table 14 specifies the information boundary and required output for every decision that can alter proof search or the formalization. These interfaces are instantiated with dataset-specific formalization templates, but the audit, repair, and validation templates are fixed across policies.

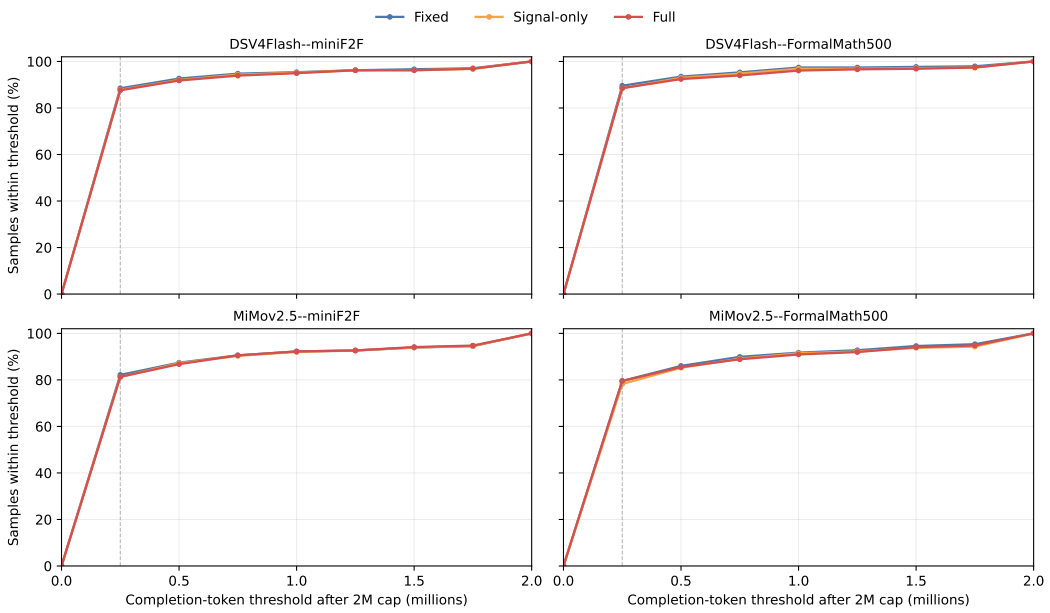


Figure 3: Empirical cumulative distribution of completion-token expenditure for *Fixed*, *Signal*-only, and *Full* across the four settings. For each 0.25M-token threshold x, the curve gives the percentage of eligible input–policy trajectories with usage no greater than x; the dashed line marks 0.25M. The corresponding means and medians are reported in Table 15. The logical total includes inherited shared-prefix usage and all subsequent proof, diagnosis, audit, repair, and validation completions; Stage I formalization cost is excluded.

	Fixed		*Signal*-only		*Full*	
Setting	Mean	Median	Mean	Median	Mean	Median
DSV4Flash–miniF2F	137.8	4.9	146.7	4.9	150.0	4.9
DSV4Flash–FormalMath	113.3	5.4	126.4	5.4	133.0	5.4
MiMov2.5–miniF2F	221.6	10.0	227.6	10.0	227.4	10.0
MiMov2.5–FormalMath	227.0	8.9	243.8	8.9	241.6	8.9

Table 15: Mean and median completion-token expenditure in thousands of tokens (K) over eligible trajectories. Each trajectory is capped at 2M tokens before aggregation, matching Figure 3. Expenditure includes inherited shared-prefix usage and all subsequent proof, diagnosis, audit, repair, and validation completions; Stage I formalization cost is excluded.

The Stage III audit uses the following normalized template; capitalization and dataset-specific Lean skeletons are omitted, but all decision-relevant fields and output constraints are shown:

> **Input:** original proposition P; current Lean file F; back-translated sections for mathematical objects and types, auxiliary definitions, constraints, and final problem statement; auxiliary feedback q.
> **Task:** identify concrete omissions, type mismatches, semantic alterations, incomplete constraints, or problem-solving computation not licensed by P. Treat the mandatory Lean template itself as intentional. Use q only to locate a possible mismatch and ground every claimed discrepancy in P. Proof failure alone is not a discrepancy.
> **Output:** {`"consistency_level": level_1|level_2|level_3, "discrepancies": [...], "recommendations": [...]`}. Each recommendation must correspond to one listed discrepancy; both lists are empty for level 1.

Policy feedback string. At an audit-eligible termination, the recursive prover has already emitted the diagnosis string d_i as part of its terminal record. No additional LLM call transforms this record before the semantic audit. *Full* places d_i verbatim in the audit's feedback field. *Signal* places *"Proof*

Setting	Modified inputs	Trigger cost mean (K)	Median / P95 (K)
DSV4Flash–miniF2F	27	47.0	6.5 / 216.0
DSV4Flash–FormalMath	17	25.3	12.0 / 117.9
MiMov2.5–miniF2F	48	52.9	44.5 / 209.3
MiMov2.5–FormalMath	37	89.4	42.0 / 573.3
Overall	129	58.5	15.9 / 216.0

Table 16: Completion-token usage before the first audit-eligible diagnosis reached the formalization layer for distinct inputs whose accepted final formalization changed. Stage I formalization cost is excluded; values include the shared recursive proof and diagnosis execution.

failed under budget." in the same field. All other audit inputs, the prompt template, output schema, decoding configuration, history update, repair procedure, validation procedure, and budget rules are identical. The audit must still ground every discrepancy in the original proposition; neither string authorizes a revision.

Once an audit confirms a discrepancy, the repair prompt supplies P, F, the verified discrepancy–recommendation pairs, and the fixed Lean template. The resulting file is reparsed, re-audited, and syntax-checked before another proof tree begins.

Figure 3 reports the empirical cumulative distribution of completion-token expenditure. At threshold x, the vertical value is the percentage of eligible trajectories whose capped usage is at most x. The reporting cap is 2M tokens: every recorded total above 2M is set to 2M. Across the four settings, 78.2–89.6% of trajectories use at most 0.25M tokens, so the 2M ceiling is substantially larger than what most inputs require. *Full* and *Signal*-only also have nearly overlapping resource curves. As Table 15 shows, their mean expenditures differ by only $+3.3$K, $+6.6$K, -0.3K, and -2.2K tokens in table order, respectively, and their medians are identical within every setting. Thus *Full*'s additional validated recoveries cannot be attributed solely to systematically larger completion-token expenditure.

Table 15 gives the corresponding setting-specific numerical summary. The median is identical across policies within each setting because more than half of the inputs terminate inside the shared execution prefix, before any policy-specific formalization intervention. The much larger means are caused by the comparatively small high-cost tail visible in Figure 3. *Full* and *Signal*-only remain close in mean expenditure, with *Full* slightly lower in both MiMo settings.

C.3 Successful Proofs and Audit-Trigger Costs

The endpoint records allow two additional checks relevant to the scope of the failure-triggered loop. First, among inputs that finish with a kernel-checked proof and have zero recorded revision attempts, the final formalization disagrees with the independent ASCC-3-MV validator on 32/435 (7.4%), 36/358 (10.1%), 46/403 (11.4%), and 57/330 (17.3%) inputs in the DSV4Flash–miniF2F, DSV4Flash–FormalMath, MiMo–miniF2F, and MiMo–FormalMath settings, respectively. The corresponding BEq disagreements are 52/435 (12.0%), 61/358 (17.0%), 53/403 (13.2%), and 46/330 (13.9%) in the same setting order. These are validator disagreements rather than human-confirmed semantic deviations; they provide a conservative estimate of the successful-proof blind spot because the compact endpoint records do not identify every audit that retained the current statement.

Second, for the 129 distinct inputs whose accepted final formalization changed, the charged shared-prefix completion usage before the first audit-eligible diagnosis reached the formalization layer has an overall mean of 58.5K tokens, median of 15.9K, and 95th percentile of 216.0K. These correspond to 2.9%, 0.8%, and 10.8% of the 2M ceiling. The setting-level values are shown in Table 16. This cost is charged to the shared recursive proof execution and diagnosis; it is not an additional 2M-token semantic-checking pass. The same execution also performs the proof search that establishes whether the current target can be closed.

C.4 Reproducibility Record

The experiment driver freezes an input manifest containing stable sample identifiers, source-row hashes, and initial-formalization hashes. A run identity additionally records the dataset, model iden-

tifier, enabled methods, selection rule, local retry limit, and proof-strategy depth threshold. The released checkpoints and endpoint records retain the formalizations, retained proof-tree state, terminal status, diagnosis and revision decisions, and aggregate token fields needed to inspect the executed trajectories. For the policy comparisons, branch metadata verifies the shared input and pre-intervention state. Atomic writes and identity checks prevent a checkpoint from being silently reused with another input or run identity.

The compact supplementary artifact contains frozen inputs, manifests, prompts, configurations, aggregate endpoint records, and retained proof checkpoints for the evaluated *Fixed*, *Signal*, and *Full* policies. Every sample has a shared-prefix checkpoint; a policy-specific branch checkpoint is retained when execution proceeds beyond that shared prefix, while every branch endpoint is recorded in its `results.jsonl`. It also contains the 137 paired initial–Full–Gate-off audit records and their locked `pass`/`not_pass` labels. To keep the release compact, it does not include raw provider request/response logs, API timing or cache ledgers, a complete event-by-event tool transcript, raw ASCC model outputs, materialized BEq wrappers, or free-text audit rationales. The released files therefore support inspection of the frozen data, endpoints, and retained proof states, rather than replay of every paid call or reconstruction of every transient event.

Final-proof acceptance applies structural checks before the Lean kernel result is recorded. Generated declarations are parsed into declaration and expression trees; a final theorem is rejected if its proof contains `sorry`, `admit`, or a direct `sorryAx` application, and generated `axiom` declarations are not admitted into the proof context. Intermediate decomposition stubs may contain placeholders while they denote unproved subgoals, but only separately kernel-checked child declarations can later be reused as facts. Consequently, no unresolved stub or newly generated axiom can enter the accepted theorem–proof artifact. The released records retain the final acceptance status and proof outcome; complete parser output and raw Lean diagnostic traces are not part of the compact release.

Frozen identities. The four formal configs are:

```
configs/miniF2F_deeseek_v4f_test_and_valid.json
configs/formalMATH500_deeseek_v4f.json
configs/miniF2F_mimo_v2.5_test_and_valid.json
configs/formalMATH500_mimo_v2.5.json
```

Their frozen manifest SHA-256 display prefixes, in the same order, are `f90d9a5bffb6`, `cec90a29eec2`, `c4ce19e69cdd`, and `6706fa6d91e9`. These manifest identities are recorded in the released configuration and manifest files. The compact release does not provide a separate repository-wide `SHA256SUMS` index or evaluator snapshot.

Execution entry points. From the repository root, the relevant entry points are:

```
python -m experiments.cli prepare --config <config>
python -m experiments.cli all --config <config>
python -m experiments.cli aggregate --config <config>
```

A one-sample config selects one stable manifest ID. The first two commands execute its trajectory, while `aggregate` produces the proof and cost summary. Replacing it with any of the four configs above selects a complete setting. The compact ASCC summaries provide all published-policy endpoint decisions and vote labels; BEq provides aggregate result JSONL files. Their raw evaluator inputs, provider outputs, and materialized Lean wrappers are not distributed in the compact artifact.

Calls cap outputs at 8,192 tokens (15,000 with reasoning); tools use routine-specific limits. These guard responses, not proof trees. Usage is checked before each paid call; over-budget responses are withheld and cannot trigger audit or revision. Retry limits are nine for capacity/rate failures, three for other requests, four for Lean operations, and three top-level resumptions. Uncharged failures and retries preserve the checkpoint, budget, and root count. The compact release retains endpoint-level token summaries rather than provider-level timing, cache, or usage ledgers.

Component	Reported configuration	Reproduction artifact
Lean environment	Lean `v4.25.0-rc2`; Mathlib revision `766e19e781a09923 44fd4f24d2662858b6b87250`	Version and revision are recorded here; `lean-toolchain` and `lake-manifest.json` are not bundled.
Model assignment	Internal ID `deepseek/deepseek-v4-flash` maps to the DeepSeek API model `deepseek-v4-flash`; `xiaomimimo/mimo-v2.5` is called through the PPIO OpenAI-compatible API. The setting model is used for formalization, proof, diagnosis, ASCC-1 audit, and repair	Requested model identifiers are recorded in configs and run metadata. Provider timing, response-usage, and cache ledgers are not released. The aliases are not date-versioned; all reported calls were made in July 2026.
Decoding	Main-system calls do not send temperature, top-p, or seed. Formalization, direct proof, decomposition, audit, and repair disable thinking; proof-strategy and heuristic truth-analysis calls enable provider reasoning at `medium`	The implementation and prompt files specify the interface and output limits; complete provider request/response logs and timestamps are not released.
Proof controller	Stage I allows five correction rounds. A trajectory allows five root trees, five local continuation/recomposition attempts per root, and recursion through depth 30. A direct-proof dialogue allows 20 proof-action attempts and 20 environment queries; initial decomposition allows 30 action attempts, later re-decomposition 20, with at most 50 environment queries	Driver code, prompts, run metadata, endpoint records, and retained checkpoints for checkpointed proof nodes.
Budget	One non-resetting 2M completion-token total per eligible input and policy in all four settings. Successful provider-reported completions are charged; failed requests without usage are not. Prover diagnosis, audit, repair, and validation are included	Frozen grid manifest, endpoint histories, and aggregate token fields; the charged-event ledger is not released.
Audit feedback string	The shared prover emits d_i before branching; *Full* forwards it verbatim, whereas *Signal* substitutes only *"Proof failed under budget."*; no separate diagnostic model call is made	Retained terminal and revision fields in the proof checkpoints; raw provider calls and per-call usage are not released.
Shared comparison prefix	Proof search and non-audit routing stay shared until an audit-eligible $D^{\dagger}$ is produced; branches then copy $\mathcal{C}^{\dagger}$ with remaining budget and tree capacity	Shared-prefix results and branch checkpoint metadata; not a complete API-event trace.

Table 17: Core implementation settings and artifacts required to reconstruct policy assignment, resource use, and control flow.

Component	Reported configuration	Reproduction artifact
Offline ASCC	Three non-thinking Pro/deepseek-ai/DeepSeek-V3 calls; provider-default sampling; no seed; majority pass; `invalid` is rejecting	Four compact endpoint summaries with per-input decisions, vote labels, keys, hashes, proof outcomes, and aggregate counts; the correction-audit pairs retain the same fields. Raw evaluator outputs and cache records are not bundled.
Offline BEq	Direct deepseek/deepseek-v4-pro; one root; no token cutoff; 20 action attempts; 180-second API and 120-second Lean timeouts; both implication directions must pass	Aggregate result JSONL files, including retained outcome fields and hashes where available; namespaced wrappers, full diagnostic traces, and BEq checkpoints are not bundled.
Pre-pruning ablation	Simultaneously removes truth, circularity, and recorded-failure-history checks and omits failure history from ordinary proof prompts; model, F_0, budget, root limit, repair, and validation remain fixed	`ablations/pre_pruning/` contains the ablation code snapshot, two configurations, frozen manifests, row-level endpoint results, and compact ASCC/BEq summaries. Complete prompt/checkpoint traces are not part of the compact release.
Failures and resumption	API/Lean infrastructure interruptions remain resumable and are excluded from mathematical failure categories; global-budget exhaustion is terminal	Checkpoint run status is retained; append-only error and API timing/usage ledgers are not released.
Randomness	One unseeded stochastic trajectory per problem–policy; no seed parameter is sent. Robustness is evaluated across four model–dataset settings, not repeated seeds	Run identifier, exact model alias, and setting are retained; per-call timestamps are not released.

Table 18: Evaluator, ablation, failure, and randomness settings. Per-call output caps are API safeguards and do not reset or replace the global completion-token budget.