FYAN: A HUMAN–AI HARNESS WITH SEMANTIC AUDITING FOR DOCUMENT-LEVEL FORMALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We present FYAN, a human–AI harness for document-level mathematical formalization. Rather than treating theorems in isolation, FYAN coordinates an end-to-end workflow spanning specification, proof planning, logical review, Lean proof construction, knowledge curation, and validation, with support for independent supervision and human guidance. A central component is evidence-grounded semantic auditing, which assesses whether formal statements faithfully preserve their informal specifications. A language model constructs structured evidence over local correspondences, omissions, scope, and logical relations, while a deterministic validator checks this evidence and produces reproducible judgments. When a substantive but admissible deviation is accepted, FYAN requires an explicit proof-transfer obligation connecting the formal statement back to a source-facing interpretation. With the same model (DeepSeek-V4.1-Flash) in every stage, FYAN proves 86 of 143 FormalTCS theorems under a strict Lean check, against 69 for a general agent harness, and raises the natural-language proof score from 0.501 to 0.851. On ConsistencyCheck, its semantic audit catches more inconsistent statements than a direct LLM judge, both on labels verified against the source (recall 0.777 vs. 0.636) and on the original labels (0.873 vs. 0.820), and localizes each mismatch it reports to a specific hypothesis, conclusion, or scope. FYAN also built ODENumLib, a 9,355-line Lean library for the numerical analysis of ordinary differential equations.

1 INTRODUCTION

Formal proof assistants such as Lean provide machine-checkable guarantees for mathematical arguments, but formalization still requires substantial human effort to translate informal statements, identify intermediate lemmas, locate relevant library results, and construct and repair proofs. Recent advances in large language models (LLMs) are changing this workflow. LLM-based systems can generate mathematical arguments, translate informal statements into formal specifications, construct proof-assistant code, and refine proofs using compiler feedback (Wu et al., 2022; Yang et al., 2023; Jiang et al., 2023; Xin et al., 2025; An et al., 2026). As these capabilities improve, automated formalization is expanding from isolated theorems toward research papers, textbooks, and other collections of interdependent mathematical results (Math, Inc., 2025; Achim et al., 2025; Zhang et al., 2026b; Gloeckle et al., 2026; Rammal et al., 2026; Wang et al., 2026b).

Scaling from a single theorem to a mathematical document changes the problem substantially. A system must identify the theorem units in the source, recover their dependencies, determine a valid formalization order, reuse previously established results, and maintain a coherent formal project. These requirements naturally motivate agentic workflows in which different components handle specification, reasoning, planning, proving, feedback, and reuse. Such workflows extend proof-search capability, but they also make intermediate formalization decisions increasingly consequential: once a generated theorem is accepted as a dependency, later results may build on it automatically.

Semantic faithfulness is a central concern. A proof assistant checks the theorem, not whether it faithfully represents the source. A formal statement may compile and admit a proof while omitting a hypothesis, changing a quantified domain, or replacing a source object with a different formal construction. At the document level, such mismatches can propagate through downstream dependencies while the project remains formally verified. We therefore distinguish *proof capability*,

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.

whether a proof can be found; *semantic faithfulness*, whether the formal statement preserves the source meaning; and *formal correctness*, whether the artifacts satisfy machine-checkable requirements.

Existing work evaluates informal–formal alignment through bidirectional provability, learned or prompted judgments, structural comparisons, expert references, and targeted consistency checks (Zhang et al., 2026a; Liu et al., 2025a; Han et al., 2026; Ammanamanchi et al., 2026). These methods provide useful theorem-level signals, but formalizing an entire document also requires turning this evidence into document-level decisions: whether a generated statement may advance, whether a deviation from the source is justified, and what evidence is required before dependent theorems can safely build on it or the project can be considered complete.

We introduce FYAN, a human–AI harness for document-level mathematical formalization. FYAN decomposes a source into dependency-aware theorem units, formalizes them in dependency order, and allows later units to reuse proved upstream results. It places a semantic audit between statement generation and proof search, while allowing models and human experts to guide the process without directly deciding whether a theorem may advance. Our contributions are:

1. **A dependency-aware workflow for document-level formalization.** FYAN organizes theorem units in a document-level dependency DAG and formalizes them in topological order, so proved upstream modules can be reused by downstream theorems (Figure 1). Components exchange persistent project artifacts rather than relying on transient agent context, while the ORCHESTRATOR tracks dependencies and project progress. Users and domain experts can inspect the project and provide guidance without directly editing protected statements or forcing stage transitions.
2. **An evidence-grounded semantic audit with one standard of equivalence.** Before proof search proceeds, FYAN compares the candidate Lean statement with its source and records local correspondences, omissions, and semantic differences (Figure 2). An LLM proposes the comparison evidence, while deterministic code checks coverage, scope, polarity, and admissible transfer directions and computes the grade. Only statements whose agreement with the source is evident are accepted directly; the rest return for repair or, if admitted, carry a bridge obligation that must be discharged in Lean.
3. **Empirical evaluation across proving, faithfulness, and document-level formalization.** With the model, toolchain, and graders fixed, FYAN proves 86 of 143 FormalTCS theorems under a strict Lean check against 69 for a general agent harness and raises the natural-language proof score from 0.501 to 0.851. Its semantic audit catches more inconsistent statements on ConsistencyCheck than a direct LLM judge (recall 0.777 vs. 0.636 on verified labels and 0.873 vs. 0.820 on the original labels). FYAN also built ODENumLib, 20 dependency-linked theorem units and 9,355 lines of Lean, including error bounds for θ-methods and the consistency conditions of linear multistep methods.

2 METHOD

FYAN maps PDF/TeX/Markdown sources into a dependency-aware Lean project with provenance, semantic-review, proof, and completion records. It combines SPECIFIER, REASONER, PLANNER, FORMALIZER, CURATOR, and AUDITOR. Components communicate through persistent artifacts and explicit control boundaries: models and experts may reason and guide, while validated evidence determines stage advancement and acceptance. Details appear in Appendix F.

2.1 WORKFLOW AND CONTROL BOUNDARIES

Figure 1 shows the end-to-end workflow. Intake preserves file and page provenance, and SPECIFIER converts the reviewed source into self-contained `problem.tex` units and a *document-level dependency DAG*. The ORCHESTRATOR follows this DAG in topological order, starting a theorem only after its dependencies are complete. Proved upstream units then become importable Lean modules for downstream theorems, avoiding unnecessary re-proving and enabling reuse across the project.

For each ready unit, REASONER, built on the open-source QED system (An et al., 2026), develops a natural-language proof while respecting the document-level dependency structure. The resulting `proof.md` preserves source provenance and may reuse previously proved upstream units. PLANNER then translates this argument into a *lemma-level DAG*, a formalization plan, and a compiling `Statement.lean` scaffold. The document-level DAG determines *which theorem is formalized*

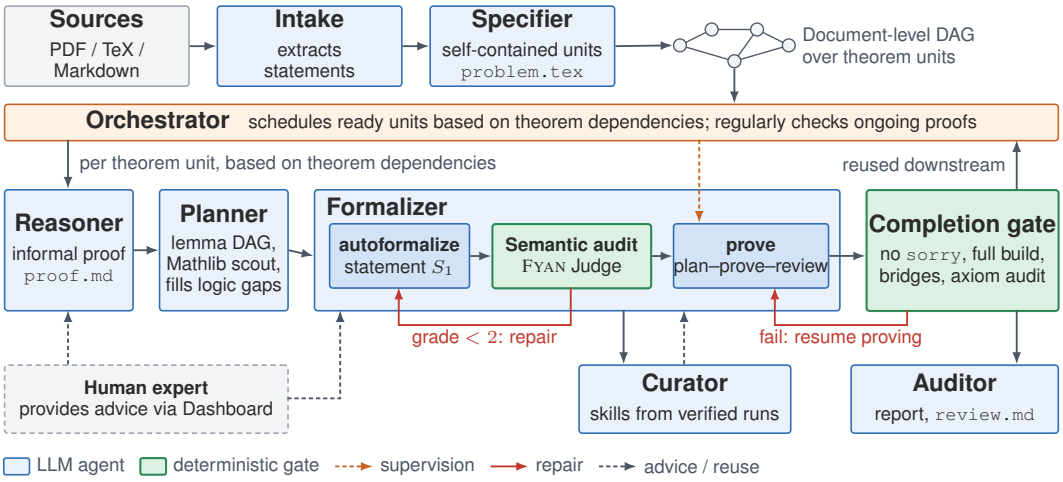


Figure 1: **The FYAN workflow for document-level formalization.** SPECIFIER turns the source into theorem units and a document-level DAG; the ORCHESTRATOR schedules ready units based on theorem dependencies and regularly checks ongoing proofs. Within each unit, models propose and deterministic gates decide: the semantic audit (Figure 2) freezes the statement produced by FORMALIZER or returns it for repair, and the completion gate accepts the proof, whose module is then reused downstream. Human guidance and CURATOR skills are advice only; AUDITOR writes a report for human inspection.

next, whereas the lemma-level DAG organizes *how it is proved*. These artifacts are kept read-only during proof search, preventing the prover from silently altering the task it is meant to solve.

FORMALIZER constructs Lean proofs through a plan–prove–review loop in a Pi-based coding-agent runtime (Zechner & Earendil Inc., 2025). It implements the scaffold in `Basic.lean`, may introduce helper lemmas and revise proof strategies, and can directly use completed upstream modules. CURATOR then mines verified runs for reusable proof patterns, useful failures, and search strategies; these skills may guide later runs but do not count as proof evidence. AUDITOR combines deterministic scans with a model-written `review.md` for human inspection.

Through the Dashboard, users can inspect progress, review blockers, and inject mathematical or strategic guidance during a run; the ORCHESTRATOR regularly cross-checks ongoing proofs against the argument, specification, and plan, and may activate critics or escalate questions. Such guidance is advice-only: it cannot edit executable Lean, modify a protected statement, or force a stage transition.

Across the workflow, persisted artifacts rather than transient agent conversation carry state, and a gateway is the sole writer of stage state. This separates *capability* from *acceptance*: reasoners, provers, retrieved skills, ORCHESTRATOR supervision, and human experts may all help find a proof, but validated evidence alone determines what counts as a faithful and completed formalization. The next two subsections describe the semantic audit and the proof-transfer and bridge obligations.

2.2 EVIDENCE-GROUNDED SEMANTIC AUDITING

A formal statement may compile, and even be equivalent to its source as a whole, while altering quantified objects, assumptions, or dependencies. FYAN therefore treats semantic faithfulness as an *auditable relation* between source and formal statement.

Structured semantic record. The FYAN Judge compares the source N with the candidate Lean statement S_1 in its actual library environment, starting from the source. Before any Lean is shown, one call extracts a formula tree of N whose definitions and atoms are anchored by literal source quotations, and an independent call reviews it. The host reads the Lean tree from the elaborated declaration, with the definitions it reaches, and cuts both trees into the same *slots*: binder domains (*objects*), hypothesis conjuncts (*hypotheses*), goal conjuncts (*claims*), and used definitions, each with host-computed scope and polarity. The FYAN Judge fills one row per group of paired slots with its

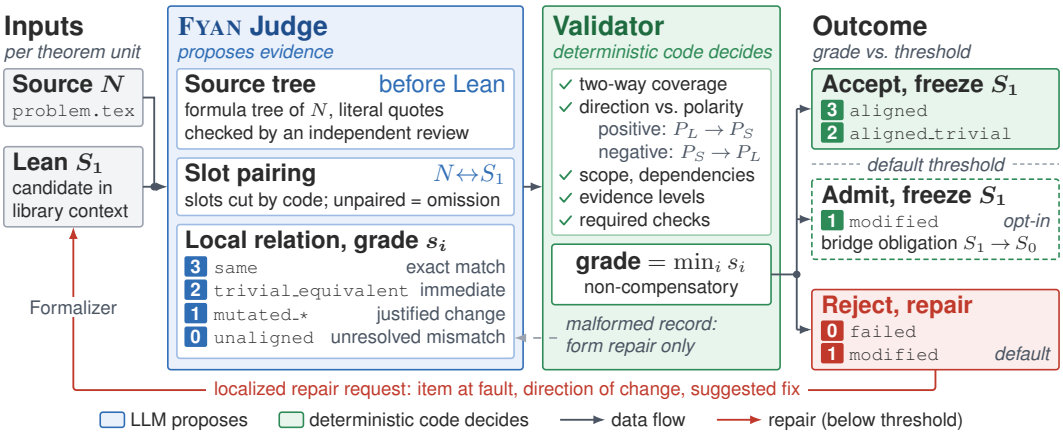


Figure 2: **The FYAN semantic audit.** The FYAN Judge, an LLM, first extracts a quoted, independently reviewed formula tree of N, then pairs the slots (objects, hypotheses, claims) that code cuts from it and from S_1, with a grade s_i per paired group. Code derives polarity, checks the record, and routes S_1 by the statement grade $\min_i s_i$: at or above the threshold (default 2) it is frozen for the document-level project, else repaired; at threshold 1, grade 1 is admitted with a Lean bridge obligation to a source-facing S_0.

Table 1: Local semantic relations. The `mutated_*` family includes equivalent, stronger, and weaker changes; one-way changes are admissible only in the appropriate logical polarity.

Relation	Interpretation	Grade
`same`	Same mathematical content and structure	3
`trivial_equivalent`	Immediate equivalence after notation or checked definitions	2
`mutated_*`	Substantive change supported by a bounded equivalence or directional proof	1
`unaligned`	Unsupported, mismatched, unresolved, or requiring a complex comparison	0

semantic relation and comparison evidence. Restrictions carried by types, subtypes, instances, or implicit parameters are treated as mathematical content.

Because both sides are cut into slots, source content that has disappeared entirely remains visible: it leaves a source slot that no row pairs, which the host scores as an omission. For example, if the source assumes both $x > 0$ and $x < 1$, while the Lean statement contains only $x > 0$, the remaining condition is paired and the unpaired slot for $x < 1$ is recorded as an omitted hypothesis.

Local semantic relations. Once the slots have been paired, FYAN classifies each local correspondence into one of the four relation families of Table 1.

The admissibility of a substantive change depends on its polarity in the complete source formula. For corresponding source and Lean predicates P_S and P_L, conjunction, disjunction, and fixed-domain quantifiers preserve polarity, while negation and implication antecedents reverse it. Hence the admissible one-way directions are

$$\text{positive: } P_L \to P_S, \qquad \text{negative: } P_S \to P_L.$$

Thus candidate conclusions may be stronger, while candidate assumptions may be weaker. For example, replacing $(A \wedge H) \to G$ by $A \to G$ is recoverable, whereas replacing $A \to (G \wedge H)$ by $A \to G$ is not in general. Omissions are handled by the same polarity principle.

Deterministic judgment. The language model fills the structured evidence record but does not choose the verdict. A deterministic validator checks coverage, relation admissibility, scope and

dependencies, evidence thresholds, and required semantic checks. Each valid row receives a grade $s_i \in \{0, 1, 2, 3\}$, and FYAN computes the statement grade

$$\mathrm{grade}(R) = \min\big(\{3\} \cup \{s_i : i \in I(R)\}\big),$$

corresponding to 3 (`aligned`), 2 (`aligned_trivial`), 1 (`modified`), and 0 (`failed`). The minimum is non-compensatory: a single unresolved, unknown, or overly complex comparison cannot be offset by well-aligned items. A `modified` verdict records a substantive but locally admissible change; Section 2.3 shows how FYAN verifies its whole-statement recoverability through explicit proof-transfer obligations and, when needed, a Lean-checkable bridge.

2.3 Proof Transfer and Bridge Obligations

A directionally admissible change is not thereby faithful. FYAN instead asks whether a proof of the candidate statement can be transferred back to a Lean rendering of the source specification.

Global proof transfer. Let

$$S_0 := \forall x : X,\ A_0(x) \to G_0(x), \qquad S_1 := \forall y : Y,\ A_1(y) \to G_1(y)$$

denote the source-facing Lean statement and the audited candidate, respectively. A recoverable modification supplies

$$e : X \to Y, \qquad a : \forall x : X,\ A_0(x) \to A_1(e(x)), \qquad g : \forall x : X,\ A_0(x) \to G_1(e(x)) \to G_0(x).$$

Here e maps each admissible source object into the candidate domain, a provides the assumptions needed to apply the candidate theorem, and g recovers the source conclusion from the candidate conclusion. Hence every proof $p : S_1$ induces the canonical transfer

$$\kappa : S_1 \to S_0, \qquad \kappa(p)(x)(h) = g\,x\,h\big(p\,(e(x))\,(a\,x\,h)\big). \tag{1}$$

These three requirements correspond to the `object_coverage`, `hypothesis_transfer`, and `claim_transfer` obligations recorded by the audit.

From local evidence to global transfer. The condition above is stated at the level of the complete theorem, whereas the FYAN Judge in Section 2.2 records local correspondences. For the logical fragment covered by our audit, these two levels are compatible: scope-aware local certificates can be composed along the logical structure of the statement according to their polarity. In particular, certificates for aligned objects and admissible local changes under conjunction, disjunction, implication, and quantifiers can be combined to construct the global maps e, a, and g, provided that coverage, scope, and witness dependencies are preserved. Thus the directional rules used by the FYAN Judge are not merely local heuristics; they form a sufficient certificate discipline for whole-statement proof transfer (Theorem E.1, Appendix E.2).

Bridge obligations. A recoverable change is still a semantic change. FYAN does not reclassify a `modified` statement as `aligned` merely because a transfer exists. By default, grades 2 and 3 are accepted directly, whereas a grade 1 statement is returned for repair. If the acceptance threshold is lowered to grade 1, the pipeline instead freezes S_1 and records a bridge obligation to a source-facing statement S_0. The prover must discharge this obligation by establishing $S_1 \to S_0$ in Lean, and the bridge statement is itself re-audited before being frozen. Thus any accepted semantic deviation is backed by a formal proof rather than by the FYAN Judge's natural-language evidence alone.FYAN therefore keeps semantic classification separate from proof-theoretic recoverability

The bridge certifies only the formal implication $S_1 \to S_0$; whether S_0 faithfully represents the source remains the responsibility of the semantic audit in Section 2.2. Appendix E gives the full compositional transfer theorem and its proof.

3 Evaluation

We ask whether the harness improves verified proof construction with the model and verifier held fixed, what the semantic audit reveals beyond direct judgment and reference equivalence, and whether FYAN sustains a dependency-linked formalization project.

3.1 EXPERIMENTAL SETUP

Stage-wise evaluation. FormalTCS (Wang et al., 2026a) contains 143 theorems and separately evaluates statement translation (NT2FT), natural-language proof generation (T2NP), and formal proof construction (FT2FP), preventing errors from propagating across stages. We compare FYAN with a general-purpose agent harness using the same DeepSeek-V4.1-Flash model, Lean environment, and released graders, with one output per item. NT2FT is evaluated by BEq^+, T2NP by the benchmark rubric judge, and FT2FP by strict Lean verification, including compilation, signature preservation, axiom checking, and fresh kernel replay. Full configurations are given in Appendix B. A stand-alone run of the FORMALIZER proof engine on ProofNet# is reported in Appendix B.2.

Semantic-audit evaluation. We evaluate the relation-based audit on 300 ConsistencyCheck statements, balanced between expert-labelled consistent and inconsistent pairs. The baseline is a direct judge using the same model and per-call reasoning setting. The audit instead performs source extraction, local alignment, and deterministic validation, so the comparison isolates the auditing procedure. Details appear in Appendix B.

Document-level evaluation. We use ODENumLib, a dependency-linked Lean development built with FYAN, to evaluate the workflow beyond isolated theorem proving. The project contains multiple theorem units that reuse proved upstream results, allowing us to examine project-scale dependency management and theorem reuse. We report its number of units, Lean artifact size, completion and replay status under the pinned environment.

3.2 VERIFIED PROOF PRODUCTION

Table 2: **Controlled comparison on FormalTCS** (143 items; one output per item).

Task	Metric	Control	FYAN	Difference
FT2FP	strict Lean pass	69/143 (48.3%)	**86/143 (60.1%)**	+17
T2NP	rubric mean	0.501	**0.851**	+0.350
NT2FT	BEq^+ equivalent	5/143 (3.5%)	**8/143 (5.6%)**	+3

FYAN improves both formal proof construction and natural-language reasoning (Table 2). The NT2FT scores remain low for both systems, largely because BEq^+ requires the generated statement to match the reference formalization closely enough for equivalence to be established. In particular, it does not identify independently declared types, which appear in many FormalTCS references (Section 3.3). This limitation affects all systems evaluated with the same metric: the strongest result reported by the benchmark authors, obtained with Claude Opus 5 and eight samples per item, is 11.2% (Wang et al., 2026a), whereas both systems here produce a single output using a smaller model. We therefore evaluate statement faithfulness separately using the semantic audit.

3.3 SEMANTIC AUDITING FOR FAITHFULNESS

We next ask whether structured semantic auditing detects errors that are missed by a direct, one-shot judgment. What a gate in front of a reusable library must deliver is reliability: an unfaithful formalization has to be caught and sent back to be rebuilt. The two kinds of error are therefore far from symmetric. A false alarm, even on a statement that is in fact equivalent to its source, costs one more autoformalization round: the statement returns with a localized repair request and the autoformalizer revises it. A missed inconsistency is silent: the altered statement is frozen, proved, and imported by downstream units, no later check revisits it, and every proof that builds on it certifies a theorem other than the one the document states. In a large project the first cost is routine and the second is unacceptable. We therefore treat recall on inconsistent statements as the primary metric, well ahead of precision.

Fewer missed inconsistencies. Under the original labels the audit already leads on recall (Table 3); with its runs without a valid record counted as acceptances, recall is 0.827 against 0.820 (Appendix C.2). Each rejection also names the source object, hypothesis, claim, or scope at fault

Table 3: **Detecting inconsistent statements on ConsistencyCheck** (300 items; inconsistent is the positive class). Missed errors are inconsistent statements judged consistent. The verified labels come from re-verifying all 300 statements against a single criterion (Section 3.3): 56 statements that the original labels call consistent become inconsistent, and none moves the other way. Bold marks the better method in each column; shaded rows are the FYAN judge. Confusion counts, F1, and bootstrap intervals are in Appendix B.3.

	Recall ↑	Missed errors ↓	Precision ↑	False alarms ↓	Accuracy ↑
Original labels					
Direct judge	0.820	27	**0.939**	**8**	**0.883**
FYAN judge	**0.873**	**19**	0.771	39	0.807
Verified labels					
Direct judge	0.636	75	**1.000**	**0**	0.750
FYAN judge	**0.777**	**46**	0.941	10	**0.813**

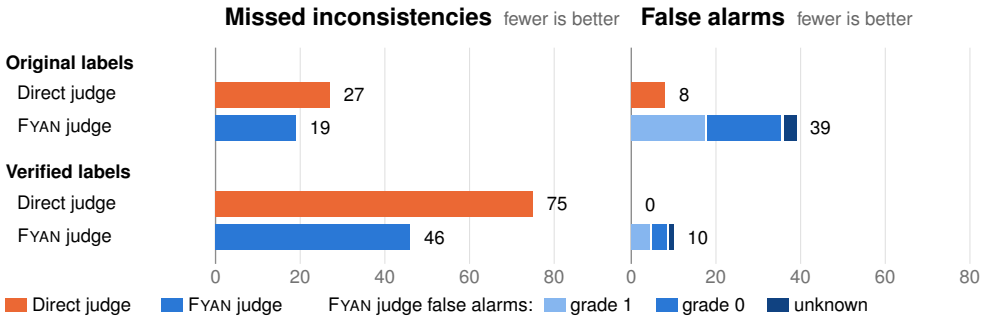


Figure 3: **Errors on ConsistencyCheck under the original and the verified labels** (Table 3). The FYAN judge's false alarms are split by grade; *unknown* marks runs without a valid record.

rather than returning a single bit. Of the 22 items that only the audit classifies correctly under these labels, 6 are structural errors in scope, quantification, or intervals that a holistic reading passes over, of the kind shown by the development item in case (a) of Table 4.

Grade-1 false alarms. Of the 39 statements that the original labels call consistent but the audit rejects, 18 receive grade 1. Re-verification (below) finds that 13 of these are not the text's statement: 9 are equivalent to it only by an argument—an enlarged domain that the other hypotheses restore, a representation that needs a translation, a codomain widened from a subspace to the whole space—and 4 rest on a reading that the text does not state or differ from it outright; in `exercise_1_13a` the Lean statement omits the connectedness of the domain and is false. The original labels accept these statements; the audit, which certifies alignment, grades them as `modified`. The other 5 grade-1 reports and 5 further rejections are false alarms of the audit itself (Appendix C.2).

Verified labels. We re-verified all 300 statements against one criterion: a formal statement is consistent only if it restates the text, differing at most by conversions a reader sees at a glance. Of the 150 statements that the original labels call consistent, 94 meet it; 36 are equivalent to the text only by an argument, 12 rely on a reading or an assumption that the text does not state, and 8 are not equivalent to it. In `exercise_3_2_21`, for instance, the hypothesis that σ and τ disturb no common element becomes the requirement that every element is moved by exactly one of them, which together with $\sigma\tau = e$ is unsatisfiable, so the Lean theorem is vacuous. None of the 150 statements labelled inconsistent meets the criterion, as expected: the criterion is stricter than the benchmark's notion of consistency, so an inconsistent label could move only if it were simply wrong. Relabelling the 56 statements as inconsistent gives the verified labels (Table 3 and Figure 3). The criterion is the one the direct judge is instructed to apply: the same objects and domains, hypotheses, and conclusion, with nothing missing, added, weakened, or strengthened (Appendix B.3). It nevertheless accepts 48

Table 4: **One audit, three yardsticks** (real records, abridged). Each yardstick returns one bit; the audit localizes the difference, grades it, and routes the statement. Relations and quoted arguments come from the FYAN Judge; polarity and grades are computed by code, and $(-)$ marks negative polarity. Case (a) is from the diagnostic pool, not the evaluation sample.

Case	Yardstick	Audit record (abridged)	Grade
(a) Arithmetic series (miniF2F)	Direct judge: *consistent* Experts: *inconsistent*	$\sum$ `k in range 5, a + k * d = 70` parses as $(\sum_k a) + kd$ with a free `k`. Hypothesis slot $(-)$, `unaligned`: "$5a + kd = 70$ constrains a different object; $a = 28$, $kd = -70$ satisfies the candidate's hypotheses but not the source."	**0** $\rightarrow$ repair
(b) Floor equation (miniF2F)	Experts: *consistent* Direct judge: *consistent*	Source: $a = p/q$ with p, q relatively prime positive integers. Binder slot `a : ℚ` $(-)$ also admits $a \leq 0$: `mutated_weaker`, so at negative polarity "the candidate statement implies the source statement (a proof transfers), but it says something else"; repair: restrict `a` to the positive rationals.	**1** $\rightarrow$ repair or bridge
(c) Product measure bounds (FormalTCS)	BEq$^+$: `not_proven`	Every row names its conversion: $n \geq 2$ as `n : ℕ`, $n \geq 2$ (condition moved into a type); $D_1, \ldots, D_n$ as `Fin n` $\rightarrow$ `ProbabilityMeasure ℝ` (reindexing); the product law as `productLaw` (Mathlib vs. local definition); both bounds `same` or reindexed. Premise: a division relies on Lean's total division, as the text never excludes zero.	**2** $\rightarrow$ accept

of the 56 relabelled statements, whereas the audit, which encodes the criterion in its closed list of conversions and its grade threshold, rejects 29. Under the verified labels the audit leads on recall and accuracy, and its false alarms fall from 39 to 10 (Table 3, Figure 3). It is correct alone on 43 statements and the direct judge on 24 (exact McNemar $p = 0.027$; Appendix C.2). Every one of the 46 inconsistencies that the audit misses is a grade-2 acceptance, in which the Judge recorded a difference as one of the listed conversions, and all 6 statements graded 3 are consistent. Grade 3 is rare, so the audit's remaining errors sit at the grade-2 boundary, in the list of admissible conversions.

The criterion decides cases in both directions. In `exercise_3_1_22a` the text asks to show that the intersection of two normal subgroups is normal and the Lean states `Subgroup.Normal (H ⊓ K)`; the experts and the direct judge accept it, while the audit grades it 1 because its Judge records Mathlib's infimum of subgroups as an argued equivalence to intersection rather than as its standard name, and the statement remains a false alarm. In `exercise_7_11` the text asks to prove that every normal operator on a complex inner-product space has a square root, and the Lean adds `[FiniteDimensional ℂ V]`. The experts and the direct judge accept it, reading in the finite dimension that the source textbook assumes throughout; the audit rejects it because that hypothesis is not in the statement. By the criterion the audit is right: a condition that is not in the text must not be assumed, however standard it is in the text's source. Statements that are stronger than the text, or equivalent to it only by a non-trivial argument, are likewise not the same statement, even when a proof of one yields the other. Were they counted as the same, one would have to decide how long a bridge proof may be and still witness equivalence, and no such line can be drawn without ambiguity. The audit therefore accepts only agreement that is evident, grades argued equivalences and one-way changes 1, and leaves them to a Lean-checked bridge rather than to the Judge's word; its aim is to make every acceptance as certain as possible, with one standard fixed before any statement is judged and applied uniformly.

One audit, three yardsticks. Table 4 sets the audit against the three yardsticks in current use, on one real record each. The direct judge passes (a) as a whole, although a misplaced summation delimiter leaves a free variable and makes the candidate false; the audit exhibits an instance on which a hypothesis slot and its source sentence disagree. The experts pass (b), which the audit grades 1 with a proposed repair. Such cases are where a binary label must make a policy choice: of the 51 statements graded 1, the experts call 18 consistent and 33 inconsistent, whereas the grade names the modification and leaves the decision to an explicit threshold. BEq$^+$ does not certify (c), although each

conversion is routine and named in the record, because it does not identify independently declared types. Of the 143 FormalTCS references, 91 declare a `structure`, `inductive`, or `class`, and all 5 NT2FT outputs of the control harness that BEq^+ certifies lie among the other 52 (one-sided Fisher $p = 0.0056$; Appendix C.1). A score against any of these yardsticks therefore cannot by itself serve as the acceptance criterion of a library; the graded audit is designed to be one.

3.4 Document-Level Formalization

ODENumLib, built with FYAN, formalizes the numerical analysis of ordinary differential equations in 20 dependency-linked theorem units (Table 5): discrete Grönwall inequalities, Taylor and interpolation remainders, local existence and uniqueness, and the error analysis of standard solvers.

Table 5: **ODENumLib at the document level.** A unit is formalized when it contains no `sorry` and introduces no new axioms in its original environment. Replay is measured under the currently pinned Lean and Mathlib revisions.

Measure	Result
Formalized theorem units	20
Replaying under the current environment	17
Requiring library porting	3
Lean artifact	9,355 lines; 34 theorems; 118 lemmas

Several units are substantial formalizations in their own right. They include error bounds for the whole family of θ-methods (1,551 lines of Lean), the moment conditions that characterize consistency of linear multistep methods (1,439 lines), stability regions and L-stability (713 lines), a multistep Grönwall inequality, and the well-posedness and convergence of the implicit backward-Euler and trapezoidal rules. Units are formalized in dependency order and import the modules they depend on rather than re-proving them; the complete inventory is in Appendix D.

4 Conclusion and Limitations

We introduced FYAN, a human–AI harness for document-level mathematical formalization. Its design separates proof capability from semantic faithfulness and formal correctness: models and human experts propose and guide, while deterministic gates decide what enters the library. With the same model, FYAN proves 86 of 143 FormalTCS theorems under a strict Lean check against 69 for a general agent harness, and raises the natural-language proof score from 0.501 to 0.851. Its semantic audit accepts only statements whose agreement with the source is evident, grades every other difference, and names the object, hypothesis, claim, or scope at fault; on ConsistencyCheck it catches more inconsistent statements than a direct judge under both the original and the verified labels. FYAN also built ODENumLib, 20 dependency-linked theorem units of numerical analysis.

For a library that later proofs import, reliability comes first. A false alarm costs one more autoformalization round, whereas an accepted misformalization silently undermines every proof built on it. Fixing one standard of semantic equivalence in advance, and leaving every non-evident change to a Lean-checked bridge, makes acceptance reproducible and as certain as the evidence allows.

Our evaluation uses one model and one trajectory per item, ODENumLib covers a single domain, and the audit costs more per statement than a single judgment. Future work will extend the evaluation to more models and domains, ablate CURATOR and ORCHESTRATOR supervision, and replace more of the Judge's conversions with mechanically checked rules.

AI USE STATEMENT

Generative AI tools are an integral part of the FYAN system studied in this work and are used in the reported experiments for mathematical reasoning, semantic analysis, and Lean proof construction. During the research process, the authors also used generative AI tools to assist with software implementation and debugging, literature summarization, and drafting and editing the manuscript. AI-assisted mathematical arguments, code, and manuscript content were reviewed by the authors. The authors take responsibility for the final content, claims, and artifacts reported in this work.

ETHICS STATEMENT

This work does not involve human subjects or private or sensitive personal data. FYAN uses autonomous agents that may access executable tools, files, and external information sources; such capabilities can introduce security and resource-use risks if deployed without appropriate isolation. Our implementation therefore scopes tool permissions, separates model-generated actions from deterministic acceptance decisions, and is intended to run in an isolated, low-privilege environment. Public benchmark data are used subject to their respective licenses, and the source material used in the document-level case study is open.

REPRODUCIBILITY STATEMENT

We provide detailed protocols and evaluation settings in Appendix B, additional semantic-audit analyses in Appendix C, and the complete ODENumLib case-study inventory in Appendix D. Appendix E gives the technical details of semantic validation and proof transfer, Appendix F describes the system implementation and trust boundaries, and Appendix G records the principal configuration, prompt construction, and independent verification procedures. The accompanying anonymous repository contains the implementation, prompts, configurations, and supporting artifacts needed to reproduce the reported experiments: `https://anonymous.4open.science/r/fyan-release-78C6`.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, et al. Aristotle: IMO-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025. URL `https://arxiv.org/abs/2510.01346`.

Pawan Sasanka Ammanamanchi, Siddharth Bhat, and Stella Biderman. Faults in our formal benchmarking: Dataset defects and evaluation failures in Lean theorem proving. *arXiv preprint arXiv:2606.29493*, 2026. URL `https://arxiv.org/abs/2606.29493`.

Chenyang An, Qihao Ye, Minghao Pan, and Jiayaun Zhang. QED: An open-source multi-agent system for generating mathematical proofs on open problems. *arXiv preprint arXiv:2604.24021*, 2026. URL `https://arxiv.org/abs/2604.24021`.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023. URL `https://arxiv.org/abs/2302.12433`.

Matthew Bolan, Joachim Breitner, Jose Brox, Nicholas Carlini, Mario Carneiro, et al. The Equational Theories Project: Advancing collaborative mathematical research at scale. *arXiv preprint arXiv:2512.07087*, 2025. URL `https://arxiv.org/abs/2512.07087`.

Joachim Breitner. Loogle: Mathlib search tool. `https://loogle.lean-lang.org/`, 2023. Source: `https://github.com/nomeata/loogle`; hosted by the Lean FRO; accessed 2026-09-03.

Rafael Cabral, Tuan Manh Do, Xuejun Yu, Wai Ming Tai, Zijin Feng, and Xin Shen. ProofFlow: A dependency graph approach to faithful proof autoformalization. In *International Conference on Learning Representations (ICLR)*, 2026. URL `https://arxiv.org/abs/2510.15981`.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023. URL https://arxiv.org/abs/2303.04910.

FrenzyMath. Archon: AI-assisted Lean project automation with DAG blueprints, proof orchestration, and multi-agent coding/proving workflows. https://github.com/frenzymath/Archon, 2026a. Apache-2.0; PKU AI4Math; v0.2.0 (2026-05) is the base of our fork; accessed 2026-09-03.

FrenzyMath. Archon: An agentic system for formalizing research-level mathematics. https://frenzymath.com/blog/archon-firstproof/, 2026b. Blog post; FirstProof Problems 4 and 6 formalized; accessed 2026-09-03.

Guoxiong Gao, Haocheng Ju, Jiedong Jiang, Zihan Qin, and Bin Dong. A semantic search engine for Mathlib4. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024. URL https://arxiv.org/abs/2403.13310.

Fabian Gloeckle, Ahmad Rammal, Charles Arnal, Rémi Munos, Vivien Cabannes, Gabriel Synnaeve, and Amaury Hayat. Automatic textbook formalization. *arXiv preprint arXiv:2604.03071*, 2026. URL https://arxiv.org/abs/2604.03071.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*, 2025. URL https://arxiv.org/abs/2507.17746.

Hojae Han, Jongyoon Kim, Sanghyeok Park, Dongwook Cheon, Yeachan Park, Myeong Jae Jeon, Sunjong Choe, Soonho Kong, Wonseok Hur, Seung-won Hwang, and Donghoon Hyeon. ShadowBench: Toward reliable automatic evaluation of semantic alignment in autoformalization. *arXiv preprint arXiv:2608.29270*, 2026. URL https://arxiv.org/abs/2608.29270.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.12283.

Haocheng Ju, Guoxiong Gao, Jiedong Jiang, Bin Wu, Zeming Sun, Shurui Liu, Leheng Chen, Yutong Wang, Yuefeng Wang, Zichen Wang, Wanyi He, Peihao Wu, Liang Xiao, Ruochuan Liu, Bryan Dai, and Bin Dong. Automated conjecture resolution with formal verification. *arXiv preprint arXiv:2604.03789*, 2026. URL https://arxiv.org/abs/2604.03789.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, et al. Prometheus: Inducing fine-grained evaluation capability in language models. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.08491.

Adarsh Kumarappan, Mo Tiwari, Peiyang Song, Robert Joseph George, Chaowei Xiao, and Anima Anandkumar. LeanAgent: Lifelong learning for formal theorem proving. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.06209.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025. URL https://arxiv.org/abs/2508.03613.

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

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *International Conference on Learning Representations (ICLR)*, 2025a. URL https://openreview.net/forum?id=hUb2At2DsQ.

Xiaoyang Liu, Tao Zhu, Zineng Dong, Yuntian Liu, Qingfeng Guo, Zhaoxuan Liu, Yu Chen, and Tao Luo. ASSESS: A semantic and structural evaluation framework for statement similarity. In *International Conference on Learning Representations (ICLR)*, 2026b. URL https://arxiv.org/abs/2509.22246. Introduces the TransTED (Transformation Tree Edit Distance) similarity.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. URL https://arxiv.org/abs/2303.16634.

Yuntian Liu, Tao Zhu, Xiaoyang Liu, Yu Chen, Zhaoxuan Liu, Qingfeng Guo, Jiashuo Zhang, Kangjie Bao, and Tao Luo. Generalized tree edit distance (GTED): A faithful evaluation metric for statement autoformalization. In *AI for Math Workshop at ICML*, 2025b. URL https://arxiv.org/abs/2507.07399.

Jianqiao Lu, Yingjia Wan, Yinya Huang, Jing Xiong, Zhengying Liu, and Zhijiang Guo. FormalAlign: Automated alignment evaluation for autoformalization. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.10135.

Patrick Massot. leanblueprint: A plasTeX plugin to build formalization blueprints. https://github.com/PatrickMassot/leanblueprint, 2023. Software; accessed 2026-09-03.

Math, Inc. Introducing Gauss, an agent for autoformalization. https://www.math.inc/gauss, 2025. Strong Prime Number Theorem formalization, https://github.com/math-inc/strongpnt; accessed 2026-09-03.

OpenClaw contributors. OpenClaw: an open agent gateway and runtime. https://www.npmjs.com/package/openclaw, 2026. Used as the supervisor runtime; accessed 2026-09-03.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. doi: 10.52202/085713-1389. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/3b77109ad4dd4ba82d07cacd4b24207e-Abstract-Conference.html.

Ahmad Rammal, Niket Patel, Fabian Gloeckle, Amaury Hayat, Julia Kempe, Rémi Munos, Charles Arnal, and Vivien Cabannes. Formalizing mathematics at scale. *arXiv preprint arXiv:2605.29955*, 2026. URL https://arxiv.org/abs/2605.29955.

Zhizhou Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025. URL https://arxiv.org/abs/2504.21801.

Peter Scholze. Liquid tensor experiment. *Experimental Mathematics*, 31(2):349–354, 2022. doi: 10.1080/10586458.2021.1926016.

Terence Tao. Formalizing the proof of PFR in Lean4 using Blueprint: a short tour. Blog post, https://terrytao.wordpress.com/2023/11/18/formalizing-the-proof-of-pfr-in-lean4-using-blueprint-a-short-tour/, 2023. Project page: https://teorth.github.io/pfr/.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. *arXiv preprint arXiv:2310.04353*, 2023. URL https://arxiv.org/abs/2310.04353.

Vijay Viswanathan, Yanchao Sun, Xiang Kong, Meng Cao, Graham Neubig, and Sherry Wu. Checklists are better than reward models for aligning language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. doi: 10.52202/085713-3824. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/a6837c1dd021f76f1b4098e3722052a8-Abstract-Conference.html.

Dingzirui Wang, Xuanliang Zhang, Keyan Xu, Qingfu Zhu, and Wanxiang Che. FormalTCS: Benchmarking end-to-end frontier formal theoretical computer science research of large language models. *arXiv preprint arXiv:2608.20153*, 2026a. URL https://arxiv.org/abs/2608.20153.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-Prover: Neural theorem proving with growing libraries. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/85dca46374dc0f27b4bb5f265b3d17f0-Abstract-Conference.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

Zichen Wang, Wanli Ma, Zhenyu Ming, Gong Zhang, Kun Yuan, and Zaiwen Wen. M2F: Automated formalization of mathematical literature at scale. *arXiv preprint arXiv:2602.17016*, 2026b. URL https://arxiv.org/abs/2602.17016.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2205.12615.

Huajian Xin, Zhizhou Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, et al. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2408.08152.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.15793.

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

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Position: Formal mathematical reasoning—a new frontier in AI. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *PMLR*, pp. 82384–82398, 2025. URL https://arxiv.org/abs/2412.16075.

Mario Zechner and Earendil Inc. pi: AI agent toolkit – unified LLM API, agent loop, TUI, coding agent CLI. https://github.com/earendil-works/pi, 2025. MIT license; npm package @earendil-works/pi-coding-agent; first released August 2025; accessed 2026-09-03.

Ke Zhang, Patricio Gallardo Candela, Sudhir Murthy, Yi Xie, Zhi Wang, and Maziar Raissi. Beyond compilation: Evaluating faithful natural-language-to-Lean statement formalization. *arXiv preprint arXiv:2606.31002*, 2026a. URL https://arxiv.org/abs/2606.31002.

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

A Related Work

Mathematical reasoning and formalization. Language models assist mathematics through both informal reasoning and formal proof construction. Informal systems generate, decompose, and critique natural-language arguments, including multi-agent proof workflows (An et al., 2026; Yang et al., 2025). Formal systems translate statements into proof-assistant languages (Wu et al., 2022; Azerbayev et al., 2023), generate proofs or tactics (Yang et al., 2023; Xin et al., 2025; Lin et al., 2025), and combine informal sketches with formal search (Jiang et al., 2023; Ren et al., 2025). Compiler-in-the-loop agents add diagnosis and repair (Thakur et al., 2023; First et al., 2023; Ospanov et al., 2025; Liu et al., 2026a). Yet a checkable proof establishes only the supplied formal statement, not that the statement preserves its informal source.

Semantic faithfulness. This distinction motivates work on semantic faithfulness. Evaluations include expert assessment and curated informal–formal pairs (Wu et al., 2022; Azerbayev et al., 2023), bidirectional provability (BEq/BEq$^+$ (Liu et al., 2025a)), learned or prompted alignment scorers (Lu et al., 2025; Zhang et al., 2026a), structural distance (Liu et al., 2025b; 2026b), expert-authored shadow statements (Han et al., 2026), and deterministic checks for vacuity and benchmark defects (Ammanamanchi et al., 2026). These signals are complementary: expert evaluation is costly, structural similarity need not imply semantic equivalence, and bidirectional provability of closed propositions need not imply semantic faithfulness. Structured evaluators and checklists decompose a global verdict into explicit criteria (Liu et al., 2023; Kim et al., 2024; Viswanathan et al., 2025; Gunjal et al., 2025), but still require controls for omitted comparisons and unsupported judgments.

Agents and document-level harnesses. Agentic proving and software-engineering systems combine decomposition, tool use, feedback, repair, and reusable experience (Thakur et al., 2023; Wang et al., 2024; Kumarappan et al., 2025; Yang et al., 2024; Wang et al., 2025). Beyond single theorems, Archon (FrenzyMath, 2026a; Ju et al., 2026) coordinates proof work across dependent declarations, while related efforts target libraries, textbooks, and papers (Math, Inc., 2025; Zhang et al., 2026b; Gloeckle et al., 2026; Rammal et al., 2026; Wang et al., 2026b; Cabral et al., 2026). Blueprint-style workflows similarly support large human formalization projects (Massot, 2023; Scholze, 2022; Tao, 2023; Bolan et al., 2025). These settings link statement assessment with artifact versions, scheduling, and completion criteria. Fyan integrates these ideas through semantic auditing, checked transfer obligations, and deterministic acceptance conditions around existing reasoning and proving systems.

B Experimental Details

This appendix provides the experimental protocols and outcome accounting for Section 3. Additional analyses of semantic-audit behavior are reported in Appendix C.

B.1 FormalTCS

FormalTCS evaluates three stages of formalization independently: NT2FT maps a natural-language theorem to a Lean statement, T2NP generates a natural-language proof from a theorem and its Lean statement, and FT2FP constructs a Lean proof from a fixed statement. Because the benchmark provides annotated inputs separately for each task, errors do not propagate across stages.

We compare Fyan with a general-purpose agent harness using the same model, Lean environment, released graders, and one candidate per item. Table 6 summarizes the experimental configuration.

The benchmark's default eight samples are reduced to one in both arms.

Outcome accounting. For NT2FT, the control produces 137 `not_proven` outcomes, five `equivalent` outcomes, and one ill-typed truncated file.

B.2 ProofNet#

ProofNet# contains 371 statements across its validation and test splits. Under Lean 4.32.2, 343 elaborate in the pinned environment and are included in the evaluation. We run only the Formalizer

Table 6: FormalTCS experimental configuration. Shared settings apply to both the control and FYAN.

Setting	Configuration
Model	DeepSeek-V4.1-Flash (`deepseek-flash`)
Sampling	T2NP: $T = 0$, top-$p = 1$; NT2FT and FT2FP: $T = 0.6$, top-$p = 0.9$; one candidate per item, submitted once to the grader; no resampling
Lean	Lean 4.32.2; Mathlib revision `905b958`; prebuilt cache unchanged
Graders	Released benchmark graders: BEq$^+$ for NT2FT, the FormalTCS Qwen3.8-Max rubric for T2NP, and strict Lean verification for FT2FP
Control	General-purpose agent using the benchmark prompts verbatim
FYAN	REASONER for T2NP and the FORMALIZER for NT2FT/FT2FP, with the benchmark brief and stage preamble
Reference access	Benchmark references are unavailable to both systems; reference files and credentials are hidden from the FYAN agent by a `bwrap` sandbox

proof engine used for FormalTCS, without the REASONER and PLANNER stages, with one trajectory and at most five prover iterations per theorem.

A theorem counts as solved only if it compiles, preserves the supplied declaration, contains neither `sorry` nor a new axiom, and passes a transitive axiom audit. The engine solves 216 of the 343 statements (63.0%). The remaining outcomes are 109 proof failures, seven changed signatures, two compile failures, and nine infrastructure failures; every non-pass is counted as a failure.

Published ProofNet results use a different Lean port and substantially larger Pass@N budgets, so we report this experiment only as an absolute cross-corpus evaluation and do not use published results as a numerical baseline.

B.3 CONSISTENCYCHECK

ConsistencyCheck contains 859 informal–formal pairs: 488 from miniF2F and 371 from ProofNet. Each statement is placed under its corresponding benchmark header in Lean 4.32.2 with Mathlib revision `905b958`. We update retired finite-sum notation in 66 statements, applying the same rewritten statement to both evaluation methods. Ten items fail to elaborate because of library drift, and 14 ProofNet entries are definitions rather than propositions, leaving 835 eligible items.

The reported evaluation uses 300 items sampled with a fixed seed, balanced between 150 expert-labelled consistent and 150 inconsistent pairs. The FYAN judge runs the same source extraction, Lean-statement extraction, local alignment, and deterministic validation used by FYAN, with grades 2 and 3 treated as consistent. All other outcomes, including a missing valid verdict, are treated as inconsistent. The direct baseline receives the same source and Lean statement in a single model call, with the instruction to decide whether the Lean statement faithfully formalizes the text: "the same objects and domains, the same hypotheses and the same conclusion, with nothing missing, added, weakened or strengthened." Both methods use DeepSeek-V4.1-Flash at low reasoning effort, temperature 0.6, and top-p 0.9.

Under the original labels, the FYAN judge is correct alone on 22 items, including 6 structural errors, and the direct judge on 45; under the verified labels, on 43 and 24 items respectively. The main text reports recall first because the audit is used as a gate, where a missed inconsistency propagates to downstream units while a false alarm triggers another repair round; 18 of the audit's 39 false alarms are grade-1 reports (Appendix C.2). The audit is also more expensive: its estimated model cost is about 0.15 CNY per item, compared with 0.007 CNY for a direct judgment. Further analysis of these disagreements, grade 1 cases, and representative failure modes is given in Appendix C.

C ADDITIONAL SEMANTIC-AUDIT ANALYSIS

This appendix provides additional analyses supporting Section 3.3. We focus on three issues: the limitations of reference-based equivalence, the main sources of disagreement in the ConsistencyCheck evaluation, and representative semantic mismatches that can survive compilation or holistic judgment.

Table 7: ConsistencyCheck confusion counts and aggregate metrics under the original and the verified labels (Section 3.3). "Inconsistent" is the positive class. Invalid or missing verdicts of the FYAN judge are treated as predictions of inconsistency and are already included in TP or FP; "No verdict" reports this diagnostic subset rather than an additional outcome category. Brackets give 95% bootstrap intervals.

Method	TP	FN	TN	FP	No verdict	Accuracy	F1
Original labels							
Direct judge	123	27	142	8	0	0.883 [0.847–0.917]	0.875
FYAN judge	131	19	111	39	10	0.807 [0.760–0.850]	0.819
Verified labels							
Direct judge	131	75	94	0	0	0.750 [0.700–0.797]	0.777
FYAN judge	160	46	84	10	10	0.813 [0.767–0.853]	0.851

C.1 LIMITS OF REFERENCE-BASED EQUIVALENCE

We first examine the behavior of the released BEq$^+$ grader used for the FormalTCS NT2FT task. A reference file graded against itself receives `equivalent`, whereas renaming a separately declared structure in an otherwise identical copy can produce `not_proven`. The grader elaborates the reference and candidate in separate namespaces and shares only their longest verbatim common prefix, so independently declared structures, inductives, and classes need not be identified automatically.

As an additional calibration, each of 21 references and 21 candidates graded against itself receives `equivalent`. However, 91 of the 143 FormalTCS reference preambles introduce a `structure`, `inductive`, or `class`, and all 5 BEq$^+$ successes of the control harness on NT2FT lie among the other 52 items; under random placement this has probability $\binom{52}{5}/\binom{143}{5} = 0.0056$. Table 8 shows three representative cases in which a semantically reasonable candidate is not certified as equivalent to the benchmark reference.

Table 8: Representative FormalTCS candidates reported as `not_proven` by BEq$^+$. Lean code is abridged.

Claim	Reference	Candidate	Reason for failure
Expected number of cycles	average using Lean's total division	explicit zero branch for an empty family	equivalence requires a case split outside the tactic ladder
ReLU networks	inductive representation over affine maps	inductive representation over matrices and biases	independently declared representations cannot be identified
Best loss in a hypothesis class	bounded infimum whose empty fibers contribute zero	infimum over the image of the class	the reference does not encode the source infimum faithfully

For the first case, a separate Lean proof establishes equivalence by splitting on whether the averaging family is empty. In the third, a direct Lean check shows that the bounded infimum in the reference may receive a zero contribution from elements outside the intended class, whereas the candidate takes the infimum over the class itself. These examples illustrate two distinct limitations: failed proof search need not indicate semantic mismatch, and a benchmark reference need not be a perfect semantic ground truth. We therefore use BEq$^+$ as a complementary benchmark signal rather than as the sole criterion for semantic faithfulness.

C.2 CONSISTENCYCHECK ERROR ANALYSIS

The FYAN judge and the direct judge make different types of errors. Under the original labels (Appendix B.3), the audit reduces false negatives (missed inconsistencies) but produces more false positives (false alarms): 39, of which 18 at grade 1, 18 at grade 0, and 3 without a valid record.

Re-verification (below) finds that 29 of them are not the text's statement—13 of the grade-1 reports, 14 of the grade-0 rejections, and 2 of the runs without a record—and that 10 are false alarms of the audit itself.

Grade 1 is where a binary label must make a policy choice. Among the 51 grade-1 records, the original labels call 18 consistent and 33 inconsistent. Of the 18, 9 are equivalent to the text only by an argument—for example a codomain widened from a subspace to the whole space, or a binder domain enlarged beyond what the text allows and restored only through the other hypotheses—and 4 rest on a reading that the text does not state or differ from it. The original labels accept these statements, whereas the audit, which certifies alignment, records them as substantive changes. Lowering the acceptance threshold to 1 would pass them, but also the 33 statements that the original labels call inconsistent: under those labels, recall would fall from 0.873 to 0.653 (98/150) while precision would rise to 0.824. FYAN therefore keeps grade 2 as the direct-acceptance threshold and admits a grade-1 statement only through a Lean-checked bridge.

The 10 false alarms that remain under the verified labels—5 at grade 1, 4 at grade 0, and 1 without a valid record—are attributable to relation-direction errors, source parsing, slot pairing, or verdict production. These are implementation failures rather than semantic disagreements about the label, and they are the target of future improvements to the audit. All 46 inconsistencies that the audit misses under the verified labels are grade-2 acceptances, in which the Judge recorded each difference as one of the listed conversions.

Re-verification of the labels. All 300 statements were re-verified. Each was first screened by Claude Opus 5.5, which saw only the source text and the Lean statement, without the original label or either method's prediction, and was assigned to one of four classes: the same statement up to conversions a reader sees at a glance; equivalent only by an argument; dependent on a reading or an assumption that the text does not state; or not equivalent. The final class of every statement was confirmed by the authors. Of the 150 statements that the original labels call consistent, 94 are the same as the text, 36 are equivalent only by an argument, 12 depend on an unstated reading or assumption (among them the finite dimension that the linear-algebra exercises `exercise_7_9` and `exercise_7_11` take from their textbook), and 8 are not equivalent; none of the 150 statements labelled inconsistent is the same as its text. Relabelling the 56 statements outside the first class as inconsistent leaves 206 inconsistent and 94 consistent statements. Under these labels the FYAN judge has TP/FN/TN/FP 160/46/84/10, accuracy 0.813 [0.767–0.853], precision 0.941, recall 0.777 [0.717–0.833], and F1 0.851; the direct judge has 131/75/94/0, accuracy 0.750 [0.700–0.797], precision 1.000, recall 0.636 [0.571–0.702], and F1 0.777 (brackets: 95% bootstrap intervals). The FYAN judge is correct alone on 43 statements and the direct judge on 24 (exact McNemar $p = 0.027$). These are the verified rows of Table 3.

Runs without a verdict. All 10 runs without a verdict end the same way: the Judge fails three consecutive times to produce a record that passes validation, and the fail-closed rule records a rejection. 7 of them fall on expert-inconsistent items and 3 on expert-consistent ones. Excluding them, its recall is 0.867 (124/143) and its precision 0.775; counting all 10 as acceptances, as a fail-open gate would, gives recall 0.827 (124/150) against 0.820 (123/150) for the direct judge, with accuracy 0.793.

Two of the 3 expert-consistent items sit on the boundary that the audit is designed to expose. In `675_exercise_7_9`, the source states that a normal operator on a complex inner-product space is self-adjoint if and only if all its eigenvalues are real, and the Lean statement adds finite-dimensionality. The source textbook assumes finite dimension throughout, but the statement itself does not, and without that assumption the reverse direction can fail; the expert label therefore rests on context outside the statement. In `145_amc12a_2013_p7`, the source constrains ten terms $S_1, \ldots, S_{10}$ by a recurrence from $n \geq 3$, whereas the Lean sequence $s : \mathbb{N} \to \mathbb{R}$ satisfies the recurrence for every n, including indices outside the source range. The two are equivalent, because any ten terms satisfying the source recurrence extend to a sequence satisfying it everywhere, but establishing this needs an extension argument across a change of indexing, the renumbering case that the Judge prompt treats explicitly. In both cases the validator refused a record it could not check rather than accept the statement on the model's word; in the pipeline, such a statement returns for another round instead of entering the library unverified.

This behavior explains why the audit uses more than a binary consistent/inconsistent decision. Grades 2 and 3 indicate alignment up to trivial equivalence, grade 1 records a substantive but potentially recoverable change, and grade 0 indicates an unsupported mismatch. A grade-1 statement is therefore not accepted merely because it appears mathematically reasonable; it requires the proof-transfer mechanism described in Section 2.3.

The classes used in this appendix come from the same re-verification as the verified labels (Appendix C.2): screening by Claude Opus 5.5, blind to the labels and to both methods' predictions, followed by author confirmation of every class.

C.3 Representative Semantic Failure Cases

A localized scope error. The arithmetic-series example, case (a) of Table 4, illustrates a mismatch that survives both compilation and holistic judgment. The source requires

$$\sum_{k=0}^{4}(a+kd)=70, \qquad \sum_{k=0}^{9}(a+kd)=210,$$

whereas the candidate's misplaced delimiter yields expressions of the form $5a + kd = 70$ and $10a + kd = 210$, with k outside the summation. The candidate therefore constrains different mathematical objects even though the overall statement remains syntactically plausible. The direct judge accepts the statement, while the FYAN judge rejects the affected hypothesis rows and localizes the missing occurrence of the common-difference term.

A definition-sensitive mismatch. A second example concerns Herstein's Exercise 4.3.25. The source refers to an ideal of the ring of 2×2 real matrices, where the intended notion is two-sided. In Lean, however, `Ideal` over a noncommutative semiring denotes a left ideal. Under the source notion, the stated conclusion holds; under the Lean notion, it does not.

The evaluation record already contained Mathlib evidence indicating that `Ideal` is a left ideal, but the comparison did not force the record to check whether this formal notion matched the source notion of "ideal." The resulting judgment therefore accepted a statement with a semantically different mathematical object. This example illustrates an important limitation of retrieval alone: relevant evidence may be available without affecting the verdict unless the audit explicitly requires the corresponding comparison. Definition-sensitive objects must therefore be treated as part of the semantic alignment rather than as incidental implementation details.

D ODENumLib Case Study

ODENumLib is a dependency-linked Lean development generated from numerical-analysis lecture notes. It was built before the semantic audit was added to FYAN, so its statements were accepted on compilation and the completion gate alone. The project contains 23 theorem units covering discrete Grönwall inequalities, Taylor and interpolation remainders, one- and multistep convergence, and stability of standard ODE solvers. Table 9 gives the complete unit-level inventory.

The "Statement" column compares the final main theorem with the retained pre-proof scaffold: $=$ denotes an identical statement, $\approx$ a change in syntax or notation only, Δ a substantive mathematical change, and – indicates that no pre-proof scaffold was retained. "Current" reports replay under Lean 4.32.2 and Mathlib revision `905b958`.

Replay status. In the original environment, 20 of the 23 units were closed without `sorry` or newly introduced axioms. Under the pinned environment, 17 replay directly. The three previously closed units `taylor/lagrange`, `trapezoidal`, and `multistep_theory` require porting because of library-level type changes, while the remaining three units are still open. We therefore distinguish original completion from current-environment replay rather than treating them as the same notion of success.

Statement changes. Among the 19 units with a retained pre-proof scaffold, nine undergo a substantive change to the main statement. These changes are not uniformly problematic: some repair

Table 9: ODENumLib unit inventory and replay status under the pinned Lean/Mathlib environment.

Unit	Lines	Statement	Current	Main result
exponential_bounds	111	$\approx$	pass	product–exponential bound
taylor/remainder	186	$\approx$	pass	Taylor remainder
ode	62	Δ	pass	local existence and uniqueness
gronwall/constant	162	$=$	pass	constant-coefficient discrete Grönwall
gronwall/variable	223	$=$	pass	variable-coefficient discrete Grönwall
taylor/lagrange	637	Δ	port	interpolation error
method/consistency_stability	198	Δ	pass	forward-Euler step stability
gronwall/multistep	520	$=$	pass	multistep Grönwall
gronwall/sum_form	176	$\approx$	pass	sum-form Grönwall
method/absolute_stability	713	Δ	pass	stability region and L-stability
multistep_theory	1,439	$=$	port	moment conditions for consistency
convergence/single_step	273	Δ	pass	one-step convergence bound
expert	451	Δ	pass	stiff-ODE example
convergence/dahlquist	652	–	pass	one direction of the convergence theorem
forward_euler	393	$=$	pass	first-order global error
runge_kutta_2	261	Δ	pass	RK2 consistency and convergence
runge_kutta_4	340	$=$	pass	fourth-order convergence; local defect assumed
backward_euler	486	Δ	pass	well-posedness and convergence
trapezoidal	521	–	port	well-posedness and convergence
theta_method	1,551	Δ	pass	error bounds for θ-methods
adams_bashforth	–	–	open	never built; propagator bound assumed
adams_moulton	–	–	open	two `sorry` terms in the main theorem
bdf	–	$=$	open	six `sorry` terms

an inadequate scaffold, while others make the final theorem weaker or easier than the source-facing result. Such weakening can also enter before proof search, as two further units illustrate. The scaffold of `runge_kutta_4`, kept unchanged, already assumes the local truncation estimate that the source result is intended to derive. In `convergence/dahlquist`, for which no scaffold was retained, the final theorem proves only one direction of the stated convergence result.

These examples explain why project completion cannot be assessed solely by whether individual Lean files close. Once a theorem becomes a dependency, any substantive change in its statement becomes part of the interface seen by downstream units, motivating semantic review before reuse.

E Semantic Audit and Proof Transfer: Technical Details

This appendix provides the technical details behind the semantic audit and proof-transfer mechanism described in Sections 2.2 and 2.3. The audit separates two roles. A language model constructs localized evidence relating the source specification to the candidate Lean statement, while deterministic code validates the structure, coverage, logical direction, and internal consistency of that evidence. When an accepted formalization differs substantively from its source-facing interpretation, the resulting transfer obligation must ultimately be discharged by Lean.

E.1 Structured Evidence and Deterministic Validation

Audit input and semantic inventory. The FYAN Judge compares the source statement with the candidate theorem signature in its actual Lean environment. Its input contains the source text; the source tree, extracted and independently reviewed before any Lean is shown, with literal quotations; the Lean tree that the host reads from the elaborated declaration, with the bodies of the definitions it reaches; and the slots cut from both trees, with host-computed scope and polarity. Candidate-authored comments never reach the FYAN Judge, because they constitute neither source evidence nor instructions to the audit.

Table 10: Principal fields of the semantic evidence record.

Field	Purpose
`source_graph`, `candidate`	Reviewed source tree and the host-extracted Lean tree.
`source_slots`, `candidate_slots`	Binder, hypothesis, and goal slots cut by the host from each tree.
`scope`, `polarity`	Logical position of each slot and the binders or premises available to it, computed by the host.
`relation`, `conversion`, `argument`, `difference`, `evidence`	Local relation of a group of paired slots or used definitions; the named conversion, argument, or concrete difference supporting it; and an optional `repair`.
`affects`, `omissions`	Downstream effects of changed objects, and unpaired slots, which the host scores as omissions.
`checks`, `transfer`	Required semantic checks and, when needed, object, hypothesis, and claim transfer evidence.

Table 11: Local semantic relations and their evidence requirements.

Relation	Evidence level	Interpretation
`same`	none	Same mathematical content and structure, up to consistent renaming.
`trivial_equivalent`	immediate	Immediate equivalence after notation or checked definitions are expanded.
`mutated_equivalent`	bounded local proof	Substantive reformulation with evidence in both directions.
`mutated_stronger`	bounded local proof	Candidate implies source.
`mutated_weaker`	bounded local proof	Source implies candidate.
`mutated`	bounded local evidence	Substantive object or representation change whose effects are tracked downstream.
`unaligned`	unresolved or over budget	Mismatch, unsupported comparison, or relation not established within the allowed comparison budget.

Rather than producing a free-form verdict, the FYAN Judge fills a structured evidence record. Its principal fields are summarized in Table 10.

The comparison is source-first and covers both sides. The source tree is fixed and reviewed before any Lean is shown, and the host cuts both trees into slots, so a slot on either side that no row pairs remains visible and is scored by the host as an omission under the rule given below. Many-to-many correspondence is allowed.

Restrictions encoded through types, subtypes, instances, implicit parameters, or definitions are treated as mathematical content. When an object or representation changes, the record also identifies the downstream hypotheses and claims whose meaning depends on that change.

Local semantic relations. The relation assigned to a local correspondence records what conversion is required between the source and Lean item, rather than whether the two complete theorems happen to share a truth value. Table 11 gives the relation vocabulary together with its required evidence.

A bounded local proof is a short derivation from the source context, checked definitions, and identified conversion lemmas, with the relevant side conditions stated explicitly. Reproving the original theorem or invoking a major result that bypasses the local comparison does not qualify. Consequently, `unaligned` means that the required relation has not been established; it does not by itself prove mathematical inequivalence.

Polarity is computed relative to the complete theorem. At positive polarity, a one-way change must support candidate → source; at negative polarity the direction reverses. A mixed-polarity occurrence requires both directions unless a larger enclosing comparison directly establishes the required transfer.

Omissions follow the same rule. A deleted conjunct is represented locally by replacement with $\top$, while a deleted disjunct is represented by $\bot$. The omitted item retains its original scope and polarity, so omission is not handled by a separate heuristic such as "dropping assumptions is safe."

Deterministic validation. The model does not choose the final grade or pipeline transition. Before grading, deterministic code checks the record schema, identifiers, cross-references, scope, admissible relation values, evidence levels, and required fields. It also checks that the source tree passed its independent review, that every slot lies in a row or is scored as an omission, that each relation is admissible at the host-computed polarity of its slots, that each `trivial_equivalent` row names its conversions from a closed list that includes standard definition unfolding, that substantive and one-way rows state their argument and `unaligned` rows their concrete difference, and that no placeholder stands in for evidence. The FYAN Judge's audit of the original text covers stated hypotheses, hidden extra conditions, conclusion strength, realized notions, domains and types, and non-vacuity. Consistency between the recorded evidence and transfer directions is also checked.

When a substantive change or omission is present, the record additionally supplies three transfer obligations: `object_coverage`, `hypothesis_transfer`, and `claim_transfer`. These correspond to the maps e, a, and g introduced below. Transfer evidence cannot override a failed local comparison.

Malformed reports are distinct from semantic failures. If deterministic validation rejects the structure of a record, the same FYAN Judge may repair its form against the unchanged source and candidate. This is form repair rather than repeated semantic voting. Once a valid unfavorable record has been produced, it is not resampled merely to obtain a different judgment.

Every validated item, omission, required check, and required transfer contributes to the non-compensatory minimum aggregation described in Section 2.2, producing grades 3 (`aligned`), 2 (`aligned_trivial`), 1 (`modified`), and 0 (`failed`). These are ordered policy labels rather than probabilities or mathematical truth values; in particular, unresolved or over-budget comparisons may also receive grade 0.

E.2 From Local Evidence to Global Proof Transfer

The semantic record is local, whereas a reusable theorem requires a whole-statement guarantee. We now state a sufficient condition under which scope-aware local certificates compose into a proof transformer.

Let

$$S_0 := \forall x : X,\ A_0(x) \to G_0(x), \qquad S_1 := \forall y : Y,\ A_1(y) \to G_1(y),$$

where S_0 is a source-facing Lean statement and S_1 is the audited candidate.

Suppose there are terms

$$e : X \to Y, \qquad a : \forall x : X,\ A_0(x) \to A_1(e(x)),$$

and

$$g : \forall x : X,\ A_0(x) \to G_1(e(x)) \to G_0(x).$$

Here e maps admissible source objects into the candidate domain, a supplies the assumptions needed to invoke the candidate theorem, and g recovers the source conclusion from the candidate conclusion.

Proposition 1 (Proof transfer)**.** *Given e, a, and g above, every proof $p : S_1$ induces*

$$\kappa(p)(x)(h) = g\,x\,h\big(p\,(e(x))\,(a\,x\,h)\big), \tag{2}$$

and hence a canonical proof transformer $\kappa : S_1 \to S_0$. Such transfers compose.

Proof. For $x : X$ and $h : A_0(x)$, the term $a\,x\,h$ has type $A_1(e(x))$. Applying p yields $G_1(e(x))$, and then $g\,x\,h$ yields $G_0(x)$.

For composition, suppose $S_2 \to S_1$ is witnessed by (e_{21}, a_{21}, g_{21}) and $S_1 \to S_0$ by (e_{10}, a_{10}, g_{10}). Then

$$e_{20}(x) = e_{21}(e_{10}(x)),$$
$$a_{20}(x, h) = a_{21}(e_{10}(x), a_{10}(x, h)),$$

and

$$g_{20}(x, h, k) = g_{10}(x, h, g_{21}(e_{10}(x), a_{10}(x, h), k))$$

have the required types. □

The directions are forced by consequence: source assumptions must suffice to apply the candidate theorem, whereas the candidate conclusion must suffice to recover the source conclusion.

Signed local certificates. For corresponding source and candidate propositions P and Q, define

$$\mathrm{Dir}^+(P, Q) := Q \to P, \qquad \mathrm{Dir}^-(P, Q) := P \to Q.$$

A signed comparison $P \rightsquigarrow_\sigma Q$ therefore requests a certificate of type $\mathrm{Dir}^\sigma(P, Q)$. The theorem root has positive polarity; conjunction, disjunction, and quantifier bodies preserve polarity, while negation and implication antecedents reverse it.

The comparison record forms a finite well-founded tree, or an acyclic shared graph. Certified leaves form a non-overlapping comparison frontier; a leaf may represent an atomic correspondence or a larger many-to-many group. Unchanged regions contribute identity certificates.

Certificates are scope-aware. A certificate below a binder must hold uniformly over that binder and may use only variables and premises available at its declared location. It may not depend on sibling branches, future witnesses, the candidate theorem being proved, or the source conclusion being recovered. Dependencies among reused certificates must be acyclic.

For structured objects, aligned dependent telescopes and well-typed field adapters construct the outer object map

$$e : X \to Y.$$

Theorem E.1 (Scope-preserving compositional transfer)**.** *Assume that:*

1. *the object adapters are well typed and define $e : X \to Y$;*
2. *for every $x : X$, there is a complete scope-aware comparison*

$$C_A(x) : A_0(x) \rightsquigarrow_- A_1(e(x));$$

3. *for every $x : X$ and $h : A_0(x)$, there is a complete scope-aware comparison*

$$C_G(x, h) : G_0(x) \rightsquigarrow_+ G_1(e(x));$$

4. *all comparison leaves are covered and their certificates are valid in their declared scopes.*

Then structural composition constructs

$$a : \forall x : X,\ A_0(x) \to A_1(e(x))$$

and

$$g : \forall x : X,\ A_0(x) \to G_1(e(x)) \to G_0(x),$$

and therefore the transformer $\kappa : S_1 \to S_0$ of Equation (2).

Proof. The object adapters construct $e(x)$ in dependency order. The negative comparison $C_A(x)$ compiles to $A_0(x) \to A_1(e(x))$, yielding $a(x)$, whereas the positive comparison $C_G(x, h)$ compiles to $G_1(e(x)) \to G_0(x)$, yielding $g(x, h)$. Substitution into Equation (2) gives the required transformer. □

Lemma E.2 (Signed certificate compilation)**.** *Every well-formed comparison tree $P \rightsquigarrow_\sigma Q$ compiles to a term of $\mathrm{Dir}^\sigma(P, Q)$ in its declared context.*

Table 12: Direction-sensitive adapters for changed quantifier domains.

Binder	Sign	Adapter	Required body transfer
$\forall$	$+$	$d : U \to V$	$Q(d(u)) \to P(u)$
$\forall$	$-$	$d : V \to U$	$P(d(v)) \to Q(v)$
$\exists$	$+$	$d : V \to U$	$Q(v) \to P(d(v))$
$\exists$	$-$	$d : U \to V$	$P(u) \to Q(d(u))$

Proof. The proof is by structural induction. Certified leaves use their supplied arrows and unchanged leaves use identity. Conjunction composes child maps componentwise, disjunction by case analysis, and fixed-domain quantifiers require the body certificate uniformly over the bound variable.

The essential reversal occurs at implication. At positive polarity, maps

$$t_1 : P_1 \to Q_1, \qquad t_2 : Q_2 \to P_2$$

give

$$\lambda f\, p.\, t_2\big(f(t_1(p))\big) : (Q_1 \to Q_2) \to (P_1 \to P_2).$$

At negative polarity both arrows reverse. Because every recursive call stays within the scope established by its parent, binder and witness dependencies are preserved. □

The construction is sufficient rather than complete. A local comparison that cannot be decomposed by these structural rules may still be recoverable through an explicit certificate for a larger enclosing expression.

E.3 Omissions and Changed Domains

Omissions. An omission remains explicit in the comparison tree. Deleting a conjunct is modeled by replacing it with $\top$, whereas deleting a disjunct is modeled by $\bot$. The elementary maps

$$P \to \top, \qquad \bot \to P$$

then determine admissibility through polarity. For example,

$$(A \wedge H) \to G \quad \rightsquigarrow \quad A \to G$$

is recoverable: removing H weakens the candidate assumptions. In contrast,

$$A \to (G \wedge H) \quad \rightsquigarrow \quad A \to G$$

does not recover the original conclusion. Any normalization that removes the introduced $\top$ or $\bot$ leaves the omitted item and its original scope recorded in the audit.

Changed quantifier domains. Suppose the source body $P(u)$ ranges over U and the candidate body $Q(v)$ over V. A domain change is admissible only when the transfer has an appropriately directed witness adapter (Table 12).

The directions depend on whether the transfer must construct or consume a quantified witness. Adapters may depend on variables already in scope, but not on future witnesses. Injectivity or surjectivity is not required by the logical rule itself; whether the adapter preserves the intended mathematical representation remains a separate semantic question.

E.4 Bridge Obligations and Deterministic Completion

The structured audit does not itself turn a non-identical formalization into a source-faithful theorem. When the stage policy admits a non-identical reading, the candidate root S_1 is frozen and the system records an explicit bridge obligation to a source-facing statement S_0. The audit record stores the semantic differences and transfer evidence motivating this obligation.

Freezing binds the accepted theorem to its normalized signature and to the source and local definitions used during the audit. Subsequent changes to a protected signature or one of its bound inputs make the corresponding certification stale rather than silently transferring approval to the modified statement.

The prover then proves S_1 without changing its frozen signature and, for each bridge obligation, states a theorem of the form

$$\texttt{theorem <root>_faithful :} \quad S_0$$

and proves it from the frozen root. The symbolic composition described above may guide this proof, but it does not discharge the obligation: only the compiled Lean theorem establishes the formal implication

$$S_1 \to S_0.$$

Before it is frozen, the bridge statement is itself subjected to semantic review so that proving an implication to an incorrectly stated S_0 cannot silently satisfy the workflow.

Completion gate. A model or prover cannot declare a theorem complete merely by emitting a completion token. Final acceptance is determined by a deterministic gate that makes no new semantic-model call. For each completion root, the gate requires:

1. the target-scope `sorry` count is available and zero;
2. the project builds successfully under the configured Lean environment;
3. every frozen theorem signature and its bound audit inputs remain current;
4. every required bridge obligation has been discharged and the frozen bridge declaration is still present;
5. every completion root carries the required semantic certification; and
6. a transitive axiom audit completes without `sorryAx` laundering or an axiom outside the configured allow-list.

These checks separate proof search from acceptance. Reasoners, retrieved skills, ORCHESTRATOR supervision, and human guidance may influence how a proof is found, but they cannot bypass the frozen statement, bridge, build, or axiom requirements used to decide completion.

E.5 EXECUTABLE FRAGMENT AND GUARANTEE BOUNDARY

The compositional checker implements a sufficient fragment of the construction above. Its input consists of ordered source and candidate object telescopes, explicit object adapters, paired formula trees, a non-overlapping comparison frontier, local certificates, and optional certified normalizations.

The supported logical fragment contains $\top$, $\bot$, conjunction, disjunction, implication, and fixed-domain universal and existential quantifiers. The checker recomputes polarity, verifies coverage and scope, and checks that each certificate has the direction required by its occurrence. Groups appearing at both polarities require evidence in both directions. Unsupported constructors, incomplete coverage, malformed scope, cyclic or undeclared dependencies, incorrectly directed certificates, or a nonidentity item labelled `same` cause refusal.

For omissions, the checker supports the explicit $\wedge\top$, $\top\wedge$, $\vee\bot$, and $\bot\vee$ cases rather than performing unrestricted logical simplification. General changed-domain witnesses are not synthesized automatically; they must be provided explicitly or justified by a larger enclosing certificate.

On success, the checker constructs symbolic recipes for e, a, and g, together with the recomputed polarities and reindexings. These recipes are conditional evidence, not Lean terms or kernel verdicts. In particular, deterministic validation can verify that required evidence is present, well-scoped, internally consistent, and used in the correct logical direction, but it cannot establish the truth of an arbitrary free-form mathematical explanation supplied by the model.

Three boundaries are especially important. First, polarity cannot be ignored: a relation valid in a conclusion may require the opposite direction when it appears in an implication antecedent. Second, local agreement does not justify changes in witness dependency; for example,

$$\forall x\, \exists y,\ y = x$$

does not in general justify

$$\exists y\, \forall x,\ y = x.$$

Third, a map between domains is not sufficient by itself. Although the inclusion $\mathbb{Z} \hookrightarrow \mathbb{R}$ may support an appropriate universal restriction, it cannot transfer

$$\exists r \in \mathbb{R},\ 2r = 1$$

to

$$\exists z \in \mathbb{Z},\ 2z = 1$$

without a valid reverse witness construction.

Finally, proof transfer and semantic faithfulness remain distinct guarantees. If S_0 faithfully expresses the source, S_1 has been proved, and Lean checks a bridge $S_1 \to S_0$, then a proof of S_0 is machine-checked. The bridge does not, however, establish that S_0 is the correct interpretation of the original natural-language source. That responsibility remains with the semantic audit. Thus the structured evidence, the compositional transfer record, and the Lean bridge certify different parts of the overall formalization claim.

F SYSTEM IMPLEMENTATION

This appendix describes the implementation details needed to understand how FYAN maintains provenance, dependency-aware execution, protected theorem interfaces, and advice-only interaction across a document-level formalization project. The emphasis is on the boundaries that affect the claims in the main text; model prompts, numerical defaults, and reproduction commands are collected separately in Appendix G.

F.1 PROJECT CONSTRUCTION, PROVENANCE, AND SCHEDULING

SPECIFIER and theorem units. SPECIFIER converts a source document into self-contained theorem units `problem.tex` and a document-level dependency DAG. Each extracted statement is stored with source provenance, including its source file, page range, and anchor. Persisted statement catalogues are never truncated; truncation is used only when constructing bounded model contexts.

An architect model groups statements into theorem units and proposes dependencies using the document structure together with a deterministic reference graph extracted from explicit cross-references. The proposed DAG is then canonicalized and checked independently. In particular, deterministic validation requires every source statement to belong to exactly one unit, checks that dependency targets exist, rejects cycles, and recomputes the topological layering from the dependency relation rather than trusting a model-generated ordering.

After the plan is accepted, units are generated layer by layer. Each `problem.tex` contains the mathematical setup, dependencies, hypotheses, and claim needed by downstream stages. Source provenance is inserted by code rather than by the generator. A validation pass compares each generated unit with its assigned source statements using a structured rubric for statement coverage, hypotheses, added assumptions, conclusions, and dependencies. Independent deterministic lints detect selected high-risk changes, including lost biconditionals, altered strict inequalities, newly introduced asymptotic terms, malformed provenance, and undeclared cross-unit dependencies.

Critical findings trigger bounded regeneration rather than silent acceptance. If validation discovers an undeclared dependency on another theorem unit, the corresponding DAG edge may be added automatically only after the updated graph has again passed cycle and layering checks. Thus models propose the unit decomposition, while coverage, graph well-formedness, provenance insertion, and selected consistency properties are handled deterministically.

Persistent state and scheduling. A gateway is the sole writer of theorem-stage state. Stage advancement is derived from persisted artifacts and exit status rather than from an agent's self-report, so a failed stage remains active rather than being recorded as complete.

A separate Python ORCHESTRATOR schedules theorem units in topological order. A unit becomes ready only after all of its declared dependencies are complete. Starting a single theorem restricts execution to the transitive closure of its dependencies, while project-wide execution dispatches independent ready units up to a configured parallelism limit.

All theorem units belong to a single Lake project, but each proving run uses a per-theorem workspace. Toolchain and package files are shared, while the current project aggregate is copied into the workspace and sibling theorem units are linked. An unfinished theorem may therefore be imported during its own proof search without being exposed as a completed module to the project as a whole.

Its import into the product aggregate is activated only after completion. Downstream units receive upstream results only from proved output modules, not from draft statement scaffolds.

F.2 REASONING, PLANNING, AND LEAN PROOF CONSTRUCTION

REASONER reasoning. REASONER is based on the open-source QED system (An et al., 2026) for natural-language proof generation. We preserve its decomposition, proving, and adversarial-review structure, while making two changes needed for document-level formalization. First, the provenance block from `problem.tex` is propagated into the resulting `proof.md`. Second, a proof step may treat an already proved upstream theorem unit as a black box rather than re-proving it.

The adapted stage also tests selected plan assertions by searching for counterexamples and counting them only when they have been executed in a sandbox. Repeated executed refutations force plan revision, while verified proof fragments independent of the rejected step may be retained.

PLANNER and the lemma-level DAG. PLANNER turns `problem.tex` and the natural-language proof into three main artifacts: a lemma-level dependency DAG, a human-readable `Blueprint.md`, and a compiling `Statement.lean` scaffold.

The proof is first reviewed for logical gaps and translated into a lemma-level DAG at a controlled granularity. The resulting graph is checked for dangling edges, self-loops, target reachability, and source-step references. Candidate Mathlib lemmas are collected from semantic search, LeanSearch, and Loogle (Gao et al., 2024; Breitner, 2023); every candidate name is then checked against the live Lean environment before being passed back to the model. Already proved upstream project lemmas are retrieved from their actual Lean declarations.

The statement scaffold deliberately leaves theorem bodies as `:= by sorry`. Its purpose is to freeze the intended interfaces before proof search, not to obtain premature proofs by weakening difficult conclusions. A bounded compile-and-quality pass checks that the required root declaration exists, rejects vacuous targets such as `True`, and verifies that every blueprint node has a corresponding tagged declaration.

Three dependency graphs therefore coexist at different levels. The *document-level DAG* schedules theorem units; the *lemma-level DAG* organizes the planned argument inside one unit and becomes read-only during proving; and the *proof DAG* may grow during proof construction as the prover introduces helper declarations. These roles are distinct: document dependencies determine what may be reused, while proof-local decomposition determines how a particular theorem is attacked.

FORMALIZER proving loop. FORMALIZER started as a fork of ARCHON v0.2.0 (FrenzyMath, 2026a;b; Ju et al., 2026) and has since been substantially rewritten: it runs on a Pi-based coding-agent runtime (Zechner & Earendil Inc., 2025), and every consequential state transition is decided outside the model. Little of the original remains beyond the plan–prove–review structure of its proving loop.

A theorem progresses through autoformalization, proving, and polishing stages. Within an iteration, the plan agent proposes proof objectives, prover agents work on those objectives under a global concurrency limit, deterministic post-processing validates touched Lean modules and synchronizes proof metadata, and a reviewer summarizes progress for the next iteration.

Provers interact with Lean through goal-state inspection, diagnostics, hover and declaration lookup, tactic trials, a code runner, axiom checking, and several search interfaces. These tools may guide search but do not determine theorem acceptance. Provers also do not directly control the project build or completion state.

F.3 PROTECTED ARTIFACTS AND DETERMINISTIC COMPLETION

After a candidate root statement passes semantic review, FYAN stores its normalized signature together with bindings to the source-facing input and the evidence used to certify it. Subsequent modification of the protected theorem signature or a bound input invalidates that certification rather than silently transferring approval to a changed theorem. Bridge obligations for admitted non-identical statements are protected in the same way; their semantic and logical role is described in Appendix E.

An agent-generated `COMPLETE` token is therefore only a request. Final completion is decided by deterministic code. For every completion root, the system requires a zero target-scope `sorry` count, a successful current project build, intact protected theorem signatures and semantic certifications, discharged bridge obligations when required, and a transitive axiom audit with no `sorryAx` or disallowed custom axiom.

Several other operations are likewise deliberately outside model control, including objective parsing, `sorry` counting, axiom auditing, synchronization of deterministic proof-status markers, and the final completion decision. Missing or stale evidence fails closed.

The system distinguishes this acceptance evidence from human-facing summaries. After a unit finishes, AUDITOR combines deterministic scans of declarations, `sorry`, `admit`, and `axiom` occurrences with a model-written review. The prose report may aid inspection, but deterministic counts take precedence and the report does not participate in theorem acceptance.

F.4 CURATED REUSE, SUPERVISION, AND HUMAN GUIDANCE

CURATOR. CURATOR extracts reusable proof experience from completed runs. It reads the final Lean source together with the structured run log and records proof patterns, tactic patterns, successful decomposition strategies, and informative failure events. Related episodes are clustered and may be synthesized into reusable prose skills.

Because these skills are model-generated, they are treated as search guidance rather than proof evidence. Every lemma named by a candidate skill is checked against the live Lean environment, and only fully verified, actionable skills pass the strict retrieval gate. A model may subsequently narrow this admitted set for a particular proof obligation, but it cannot introduce a skill that failed deterministic admission.

The prover receives skill metadata and may open a skill explicitly when it is relevant to the current obligation. Lemma names, tactics, imports, and side conditions found in a skill remain suggestions to be checked against Lean. Consequently, CURATOR can change which proof search is attempted but cannot change whether the resulting theorem satisfies the completion gate.

We do not attribute an independent empirical effect to CURATOR in this paper.

ORCHESTRATOR and human guidance. In addition to scheduling theorem units, the ORCHESTRATOR independently checks ongoing FORMALIZER proofs automatically and regularly, cross-checking the mathematical argument, specification, and proof plan. Its supervision component uses the OpenClaw runtime (OpenClaw contributors, 2026) and is intentionally advice-only. Its model has no direct filesystem, shell, web, or executable-Lean write capability; file mutation associated with supervision is performed by a restricted sidecar.

The sidecar monitors coarse signs of stagnation, including an unchanged positive `sorry` count, repeated blockers, and iterations with no prover edits. Such events may trigger review or enable additional critic agents. These event-triggered reviews supplement the regular automatic checks; FORMALIZER may also explicitly request a review.

ORCHESTRATOR advice is inserted only as non-executable comments. Questions to a human operator are routed through the Dashboard, and the returned answers enter the proving context through the same comment-only channel. Human and ORCHESTRATOR guidance can therefore affect strategy and search, but neither can directly edit a protected theorem statement, inject executable proof code, or bypass the completion gate.

We do not evaluate ORCHESTRATOR supervision as an independent capability intervention. Its role here is architectural: it provides an external observation and communication channel without becoming another authority over proof acceptance.

F.5 PIPELINE TRUST BOUNDARIES

Table 13 summarizes the separation between components that propose mathematical content or search actions and mechanisms that decide whether artifacts may advance.

Table 13: Principal trust boundaries in the FYAN pipeline.

Component	May influence	Cannot determine directly
SPECIFIER models	theorem grouping, candidate dependencies, generated unit text	graph validity, deterministic provenance, final downstream semantic acceptance
REASONER / PLANNER	proof strategy, lemma decomposition, candidate scaffold	final protected theorem interface or proof completion
FORMALIZER agents	proof plans, helper lemmas, executable proof search	completion state, protected signatures, axiom acceptance
FYAN Judge	structured comparison evidence	deterministic record validity or Lean proof validity
CURATOR	retrieved proof patterns and search suggestions	whether a resulting proof is accepted
ORCHESTRATOR / human	strategic advice, questions, critic activation	executable Lean, protected statements, or completion
Deterministic gates	stage transitions and acceptance from persisted evidence	mathematical interpretation beyond the evidence they validate

The FYAN Judge uses no research interfaces: each audit call receives only its own input. The proving agent may use Lean inspection and theorem search tools. Retrieved information is never itself an acceptance decision. When required evidence, tool output, or protected state is unavailable, the corresponding deterministic check fails closed.

This division is the central implementation principle of the harness: model-based components expand the space of candidate formalizations, arguments, and proof searches, whereas persisted evidence and deterministic checks control what is allowed to become a reusable project artifact.

G REPRODUCIBILITY AND CONFIGURATION

This appendix records the pipeline configuration and reproduction information that is not already specified in the experimental protocols of Appendix B. Experimental model, sampling, benchmark, and Lean settings are reported with the corresponding evaluations; here we focus on the defaults that govern the FYAN workflow, semantic audit, and document-level execution.

G.1 SYSTEM CONFIGURATION

All project-level settings may be overridden in `.archon/config.json`. Table 14 summarizes the principal FORMALIZER and semantic-audit defaults used by the system.

The semantic policy distinguishes the threshold used for direct acceptance from the treatment of admitted non-identical statements. Under the default `bridge` policy, a substantive deviation that is allowed to advance must still satisfy the bridge mechanism described in Appendix E; configuration does not override protected signatures or the deterministic completion gate.

Table 15 gives the main defaults for source decomposition and curated reuse.

These defaults specify operational budgets rather than correctness guarantees. Changing them may alter search effort or the frequency of review, but final theorem acceptance remains subject to the protected-artifact, build, bridge, and axiom checks described in Appendix F.

G.2 PROMPTS AND SEMANTIC-AUDIT INPUT

The semantic audit uses the prompts in `.archon-src/prompts/coverage/`. Each stage is a fresh model call that receives only its own input: `source.md` extracts the source tree with literal quotations; `source_review.md` reviews it independently; `source_repair.md` and `source_repair_review.md` repair a rejected tree and review the repair; `alignment.md` is the FYAN Judge; `pairing.md` is appended to the unchanged FYAN Judge input for one re-pairing

Table 14: Principal FORMALIZER and semantic-audit defaults.

Setting	Default
Maximum proving iterations	50
Maximum concurrent provers, including nested subagents	4
Maximum dispatched objectives per iteration	10
Block execution on incomplete dependencies	true
Semantic protection	enabled
Faithfulness threshold	`trivial`
Non-identical-statement policy	`bridge`
Maximum statement-repair attempts	3
Maximum bridge-repair attempts	3
Structured-record form-repair attempts	3 per FYAN Judge call; 5 for source-tree preparation
Source-tree repair rounds	2
Re-pairing calls for unpaired slots	1
Research interfaces available to the FYAN Judge	none
Maximum complete FYAN Judge input	700,000 bytes, never truncated
Maximum FYAN Judge output	provider maximum for DeepSeek models (393,216 tokens); otherwise 32,768 tokens
FYAN Judge timeout	none (no local deadline)
Enabled proving subagents	none by default
Maximum concurrently scheduled theorem units	3
ORCHESTRATOR retry budget	1

Table 15: Principal SPECIFIER and CURATOR defaults.

Setting	Default
SPECIFIER maximum units generated per call	6
SPECIFIER maximum prior-context budget	6,000 tokens
SPECIFIER maximum verification revisions	2
SPECIFIER maximum units repaired in validation	10
Maximum unit TeX size admitted to validation	40,000 characters
CURATOR promotion threshold	3 projects
CURATOR minimum observations	3
Single-project minimum skill size	180 characters / 2 lemmas
Maximum retrieved skills per run	20
Maximum stuck-triggered retrievals per run	5
Minimum review confidence before rejection	0.3
Cluster merge thresholds	Jaccard ≥ 0.7 automatic; $[0.4, 0.7)$ reviewed

call when slots remain unpaired; and `discrepancy.md` routes a rejection to source-tree repair, candidate re-extraction, statement repair, or source clarification. The user message assembled for each FYAN Judge call contains:

1. the source text;
2. the reviewed source tree;
3. the candidate Lean tree extracted from the elaborated declaration, with the bodies of the local definitions it reaches; and
4. the host-computed source and candidate slots with their polarities, and the polarity of each source definition.

The prompt asks the model to produce the structured semantic evidence record described in Appendix E; its full text is reproduced at the end of this subsection.

If a generated record fails deterministic schema or consistency validation, a bounded form-repair attempt receives the validator complaint and the rejected record as untrusted input. These retries repair the structure of the record; a valid unfavorable semantic judgment is not repeatedly sampled as a vote.

The proving pipeline uses separate prompts for statement construction and polishing, including `prover-autoformalize.md` and `prover-polish.md`. SPECIFIER uses `system_validator.md` for model-assisted unit validation. The exact prompt files used for a reported run should be retained together with the corresponding run artifacts rather than reconstructed from their names alone.

Judge prompt. The complete system prompt of the FYAN Judge (`alignment.md`) follows verbatim. In it, `equivalent`, `stronger`, and `weaker` name the relations `mutated_equivalent`, `mutated_stronger`, and `mutated_weaker` of Table 11.

```
Compare a Lean statement with the natural-language source text ("source"). You
receive the reviewed source tree ("source_graph"), the actual Lean candidate
extracted from its elaborated Expr ("candidate"), and host-computed slots. Neither
tree can be edited. The original text is authoritative for meaning. Do not test
whether the theorem is true. Compare what the two statements say.

Slots. The host strips the outer telescope of each statement into slots: "binder" (a
quantified variable and its domain), "hypothesis" (one conjunct of an assumption)
and "goal" (one conjunct of the conclusion); binders and hypotheses restrict the
claim (negative), goals assert it (positive). Curried and conjoined hypotheses give
the same slots; structure nested inside a slot is one unit. Each slot lists id,
kind, polarity and "text" (source) or "lean" (candidate); "definition_polarity"
gives each source definition the sign of its uses ("neutral" = used as data;
"unused" = the formula never uses it: it needs no group). Write one row per group of
corresponding slots (a group means the conjunction of its slots under the paired
binders). Pair binders, hypotheses and goals in separate rows, unless one side is a
single slot that the other unpacks. Leave a slot with no counterpart out of every
row; the host scores it (a dropped source hypothesis or an extra candidate
conclusion makes a stronger theorem; anything else missing or extra is a
difference). A candidate binder marked "context" needs a row only if it realizes a
source variable; pair every other binder, e.g. [Field K] bundled with its carrier K.
A source hypothesis the candidate's types enforce joins that binder's row (condition
moved into a type). A candidate binder naming a value the source quantifies inside a
hypothesis ((exists L, P L) -> Q is forall L, P L -> Q) joins that hypothesis row
(currying); one whose hypotheses realize a source definition (initial values, a
recurrence) joins the row of the source slots using it (inlining); it must occur in
a candidate hypothesis of its row and in no candidate hypothesis outside it. List a
"defined_by" binder (v = t) realizing a used source definition, without its
hypothesis (placed with it), in that group's "candidate_slots"; the group is at best
trivial_equivalent (inlining).

Relations, for a slot group or a definition group (P = source part, L = Lean part):
- same: identical meaning and representation, one slot per side of the same kind
  (bound-variable renaming and de Bruijn shifts from proof binders are allowed).
- trivial_equivalent: identical meaning. The representation changes only by
  conversions from this list, named in "conversion" (comma-separated): bundling,
  currying, binder order, argument order, condition moved into a type, Mathlib vs
  local definition, inlining, implicit context, renaming, reindexing, standard
  unfolding. A row that regroups slots names bundling, condition moved into a type
  or standard unfolding (or the currying or inlining above); splitting a
  conjunction or a chained inequality (0 < d <= n) into several slots is bundling.
  If the definition groups already record every change a slot row needs, the row
  names those conversions. Standard unfolding: a difference an undergraduate who
  knows the concepts sees at a glance: elementary logic (not exists vs forall not,
  currying), a standard definition unfolded (units digit vs % 10, percent vs /100,
  degrees vs radians, abelian vs forall a b, a*b = b*a, a Mathlib predicate vs its
  defining formula), or a change of domain that an accompanying bound neutralizes
  (n : Int with n >= 9 vs n : Nat with 9 <= n; exists! k : Nat vs exists! k : Int
  when k > 0 is forced). Name it and give the one-line unfolding in "argument".
- equivalent: L and P are equivalent by a real argument beyond the listed
  conversions. Give it in "argument".
- stronger: L implies P, uniformly in the enclosing variables. For a binder the
  Lean domain is SMALLER (Nat for "integer"); a LARGER Lean domain is weaker (Nat
  for "positive integer" adds 0; Complex for a real; Int for Nat). Give "argument".
  A binder and the hypothesis that bounds it form one group: judge them together
  in one row (bundling, condition moved into a type or standard unfolding).
- weaker: P implies L, uniformly in the enclosing variables. Give "argument".
- unaligned: the meaning differs or cannot be established. Give "difference":
  what the source says versus what the candidate says, preferably an instance
  where they disagree (an index shift that changes which elements are covered,
  a flipped inequality, a different range or set, a missing case).
Choose the relation by what it takes to get from P to L, not by whether the theorem
survives. Any row may add "repair": the change that would make it same. Mixed rows
(joining negative and positive slots) and neutral definitions admit only same,
```

```
trivial_equivalent or equivalent. Be generous with genuine re-encodings; reserve
unaligned for changes in which situations the claim covers or what it says. Text
indexing a sequence from 1 vs Lean Nat -> A: hypothesis instances that only
constrain indices outside the source range (a recurrence at n = 0) are reindexing if
the goal never reads those indices and every source sequence has a value there
satisfying them (name it, e.g. a 0 = 0); otherwise they restrict.

Definitions. Each source definition the formula uses belongs to exactly one group
{"source_ids","candidate_names","members","parameters",relation,...}. Members assign
each source definition its candidate declaration roots (their union is
candidate_names; an empty list means inlined or missing, never same). Parameter
pairs map a source parameter to the FULL candidate declaration name and the index of
its lambda binder; use parameters=[] when none correspond (e.g. bundling). Read
every local candidate body: a definition means what its body computes, not what its
name says. A named notion the text does not expand is compared with what the text,
or the standard meaning of that name, says: its body must be that notion (same) or
it is unaligned. Realizing it concretely is never a conversion. Only if nothing
constrains the notion may you record the realization in "premises". Candidate names
may also be exact names from external_primitives: imported constants carry their
documented meaning (their bodies are not exported, which is not missing evidence).
An imported anchor is same only for a source symbol, alone or applied to its
definition's parameters in order, with no dependency on another definition; an
expanded source construction realized by an imported definition is
trivial_equivalent, "Mathlib vs local definition". List every local candidate
declaration except the target exactly once in "declarations" with its linked source
ids; empty links only for generated or inert machinery.

"premises": plain sentences you assumed but could not check (an imported constant's
meaning, the tree's reading of a recorded ambiguity). They never block. A premise
may never assume that a local definition means what the text says. Doubt is not a
finding: a believed difference is a row or a defect.

"meaning_audit" reads the ORIGINAL TEXT for what the rows cannot show, in this
order, each {"check","status":"ok"|"defect","evidence"}: hypotheses_present (each
condition the text states is in a source slot), no_extra_conditions (no candidate
condition hides in a definition, type or module that no row states),
conclusion_strength (each conclusion the text asserts is a source goal slot, with
its quantifier scope), notions_realized (each notion the text defines or constrains
is a definition with that body, not a free parameter, field, axiom, arbitrary choice
or trivially true body; name each; an unconstrained field of a locally declared
structure is such a defect: quote the text, name the field), domains_types (Nat
contains 0 and truncates n - 1, x / 0 and x % 0: if the source domain excludes 0,
compare the n = 0 case), non_vacuity (hypotheses satisfiable, conclusion not
trivial). A difference that a row or an unmatched slot records is not a defect. A
defect adds "source_quote" (verbatim text) and "lean", and states the difference.

A candidate field {"$ref":"E0001"} is an entry of the lossless expression_pool;
expand it recursively. Author no score or verdict; a valid unfavorable row is final.

Return JSON:
{"definitions":[{"source_ids":["D1"],"candidate_names":["Foo"],"relation":"same",
 "evidence":"...","members":[{"source_id":"D1","candidate_names":["Foo"]}],
 "parameters":[{"definition":"D1","source":"x","candidate_name":"Foo","candidate_index":0}]}],
 "declarations":[{"name":"Foo","source_ids":["D1"],"reason":"..."}],
 "slots":[{"source_slots":["s1"],"candidate_slots":["c1"],"relation":"same","evidence":"..."},
  {"source_slots":["s2"],"candidate_slots":["c2","c3"],"relation":"trivial_equivalent",
   "conversion":"condition moved into a type","evidence":"..."},
  {"source_slots":["s4"],"candidate_slots":["c5"],"relation":"weaker","argument":"...",
   "repair":"...","evidence":"..."}],
 "premises":[],
 "meaning_audit":[{"check":"hypotheses_present","status":"ok","evidence":"..."}, ...]}
```

G.3 Reproduction Records and Independent Verification

A reproducible run requires more than the final generated theorem. For each reported result, the reproduction record should bind the output to the corresponding source corpus, configuration, composed prompts, model and provider settings, random seed where applicable, Lean and Mathlib revisions, validation code, aggregation code, retained run logs, and independent verification output. Cryptographic hashes may be used to identify these artifacts unambiguously.

Proof claims are independently checked after generation. The verifier recompiles each claimed theorem in the configured Lean environment, checks that the protected declaration agrees with its frozen signature, rejects `sorry` and `admit`, and audits the transitive axioms of every completion root. When a bridge is required, the bridge theorem is included in the same verification procedure.

Semantic evidence and formal verification are retained separately. The former records why the system considered a statement aligned with, or recoverable from, its source; the latter establishes that the resulting Lean declarations compile and depend only on permitted axioms. Reproducing only the final Lean file therefore does not reproduce the full pipeline acceptance decision. Resource accounting follows the same principle. If runtime usage is unavailable for a run, that run is excluded from aggregate cost claims rather than being assigned zero cost.