SAGE: FORMALIZATION WITH SEMANTIC CORRECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

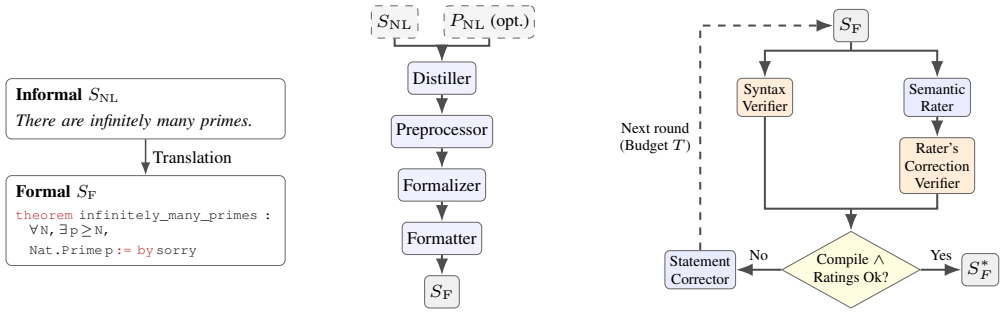
While neural theorem provers have achieved impressive milestones in formal mathematics, they largely operate on the assumption that faithful formal statements are already provided. Translating informal natural language into a formal language is a critical data bottleneck plagued by an "illusion of rigor": standard type-checkers accept statements that compile but drop hypotheses, introduce vacuous truths, or subtly alter mathematical bounds. To resolve this, we introduce Sage (Semantic Agent-Guided Formalization Engine), an agentic framework that replaces monolithic translation with a four-stage decomposed generation pipeline coupled with a dual-signal semantic correction loop. By pairing formal compiler diagnostics with multi-dimensional semantic feedback, our correction loop enforces mathematical fidelity alongside syntactic validity. By explicitly accounting for the gap between open-ended queries and declarative formal targets, our pipeline prevents models from achieving high formalization rates by guessing unverified answers (exhibiting a $70.9\,\%$ answer leakage rate in monolithic baselines). Consequently, Sage suppresses leakage to $2.7\,\%$ while achieving $73.3\,\%$ pass@4 joint compilation and semantic fidelity on the OMNI-MATH benchmark without proofs (compared to $42.0\,\%$ for a fine-tuned Goedel-Formalizer-V2 baseline). Finally, on IMO-UNFORMALIZED, a novel frontier of 175 unformalized International Mathematical Olympiad problems, Sage demonstrates effective zero-shot generalization with $87.4\,\%$ pass@4 verified fidelity compared to just $19.4\,\%$ for the baseline, winning over $79\,\%$ of blind pairwise evaluations.

1 INTRODUCTION

Neural theorem provers and olympiad-scale systems such as Goedel-Prover (Lin et al., 2025), DeepSeek-Prover (Ren et al., 2025) and AlphaProof (Hubert et al., 2025) have pushed automated reasoning in Lean (de Moura & Ullrich, 2021), yet their benchmarks largely assume the formal statement S_{F} is already given and focus solely on finding a formal proof ($S_{\mathrm{F}} \to P_{\mathrm{F}}$). In practice, competition problems, textbooks, and user-facing assistants arrive as informal natural language S_{NL}; translating $S_{\mathrm{NL}} \to S_{\mathrm{F}}$ (Figure 1a) with *semantic fidelity* is the mandatory upstream step and a bottleneck for high-quality formal training data. A faithful S_{F} is the contract under which every downstream prover, benchmark, and library entry must operate—yet the field lacks a shared account of what that contract *contains* beyond "it type-checks." Crucially, compile success is not semantic fidelity: Lean 4 accepts statements that compile but fundamentally *misstate* the problem. For example, if a model drops a crucial minimality condition, the resulting bare existential type-checks perfectly with `sorry`. Compilers cannot detect semantic mismatches, leaving standard compile-only repair loops blind to dropped hypotheses, introduced tautologies (`True`), and vacuous placeholders.

Our approach. To address this, we introduce **Sage** (**Semantic Agent-Guided Formalization Engine**), an agentic framework shifting autoformalization from monolithic generation to iterative semantic correction. Sage decomposes formalization into four inspectable stages (distillation, preprocessing, formalization, formatting) and refines candidates via a dual-signal correction loop combining compiler diagnostics with multi-dimensional semantic feedback. To support distinct downstream applications, Sage provides two configurations: **Answer-Aware** (leveraging informal proofs to curate resolved, prover-ready targets) and **Answer-Agnostic** (translating problem text alone without proof oracles or solutions). Our contributions are as follows:

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



(a) Autoformalization example. (b) Decomposed generation. (c) Correction loop architecture.

Figure 1: Overview of Sage: an autoformalization example (**a**), the multi-stage generation chain (**b**), and the syntax/semantic feedback loop (**c**). Blue: LLM agents; orange: verifiers.

1. **Theoretical Foundations:** We formalize the distinction between Assertion- and Question-type mathematics, exposing the *Discovery Trap* and type-theoretic vulnerabilities that arise when translating open queries without proof oracles.
2. **Decomposed Generation Pipeline:** We introduce a modular four-stage formalization protocol that decouples goal extraction, typing, and syntax generation, isolating structural constraints and eliminating unmonitored answer guessing.
3. **Multi-Dimensional Semantic Feedback:** We define an actionable rubric tracking critical axes of translation divergence—including dropped hypotheses, altered bounds, and vacuous placeholders—operationalized into a test-time semantic evaluator.
4. **Dual-Signal Correction Loop:** Our iterative repair loop pairs Lean 4 compiler diagnostics with semantic feedback, correcting subtle mathematical flaws that type-checkers overlook.

Empirically, Sage outperforms zero-shot generation, syntax-only repair loops, and fine-tuned baselines. On OMNI-MATH without proofs, Sage suppresses unmonitored answer leakage from 70.9 % to 2.7 %, while nearly doubling the fine-tuned Goedel-Formalizer-V2 baseline in verified formalization (73.3 % vs. 42.0 % pass@4; 50.3 % vs. 26.7 % pass@1). On the unformalized IMO frontier (IMO-UNFORMALIZED), Sage achieves 87.4 % pass@4 verified fidelity (vs. 19.4 % for the baseline), winning over 79 % of blind pairwise head-to-head comparisons.

Section 2 situates our work in autoformalization and self-refinement; Section 3 formalizes the theoretical limits of translation; Sections 4 and 5 describe the approach and evaluation design; Section 6 reports results; and Sections 7 and 8 discuss limitations and conclude.

2 RELATED WORK

Neural Theorem Proving and the Statement Bottleneck. While automated theorem provers have achieved olympiad-level reasoning (Hubert et al., 2025; Lin et al., 2025; Ren et al., 2025; Yang et al., 2023), they operate under the assumption that formal targets S_{F} are already given. Yet establishing S_{F} from natural language ($S_{\text{NL}} \to S_{\text{F}}$) remains an open vulnerability: human annotations frequently encode subtle misinterpretations (Zheng et al., 2022; Liu et al., 2023) that require retrospective benchmark corrections (Ospanov et al., 2025; Poiroux et al., 2025), whereas automated pipelines guided purely by type-checkers tend to compromise mathematical constraints simply to secure compilation (Ammanamanchi et al., 2026; Zhang et al., 2026). Although formalization has also been used downstream to verify multi-step informal reasoning (Zhou et al., 2024), our work tackles the primary upstream bottleneck: synthesizing semantically faithful formal targets.

Autoformalization and Semantic Evaluation. Autoformalization leverages formal corpora (Azerbayev et al., 2024a; Ying et al., 2024) and specialized foundation models (Azerbayev et al., 2024b). Yet on open queries without proofs, unguided models frequently drop hypotheses or alter bounds. To evaluate fidelity, Lu et al. (2025) align representation spaces, Liu et al. (2025) introduce bidirectional equivalence (**BEq**), Han et al. (2026) test shadow theorems, and Zhang et al. (2026) audit compile–faithfulness gaps—metrics that are typically *reference-based* (requiring a human gold S_{F}^*) and thus inapplicable to unformalized frontiers like IMO-UNFORMALIZED. Roundtrip

rerankers (Li et al., 2024) similarly operate *post-hoc* over static drafts. In contrast, our in-loop gate Sage-SM is *reference-free*, scoring candidates directly against S_{NL} to drive test-time repair.

Modular Decomposition and Iterative Refinement. Modular decomposition (Ridnik et al., 2024) and iterative self-refinement (Shinn et al., 2023) are well-established paradigms in general code generation. In formal mathematics, prior systems explore: (i) compiler-only repair (Lu et al., 2024); (ii) monolithic reflective self-critique (Chen et al., 2026); or (iii) dependency-graph retrieval without an in-loop semantic gate against S_{NL} (Wang et al., 2025). To avoid confounding factors from disparate base models and training sets, our 2×3 factorial ablation (Section 5) benchmarks these core mechanisms under an identical backbone and matched compute. Crucially, compiler-only repair prioritizes syntactic compilation over semantic fidelity, whereas monolithic reflection succumbs to the Discovery Trap on open queries (Section 6.3). Sage resolves both by combining four-stage decomposition with reference-free semantic gating.

3 Formalizing Mathematics: Beyond Naive Translation

Autoformalization is a constrained translation task: mapping informal mathematical language (English) into a formal language. A successful formalization must satisfy the following desiderata:

1. **Syntactic validity:** the statement must be well-typed in the target system and respect the *single-sorry contract*: S_{F} must contain strictly one terminating placeholder (`:= by sorry`) closing the *primary target declaration* (e.g. `theorem`).
2. **Semantic fidelity:** the statement must preserve the mathematical meaning of the source.
3. **Texture:** the formalization should respect the pragmatic structure of the source—whether the informal text asserts a declarative claim to be verified, or poses an open-ended question asking for a value, witness, or polarity.

We call the second dimension *semantic match*. In Section 4.3, Sage-SM serves as an in-loop fidelity metric to gate our correction loop, and we adopt the existing Goedel-SM protocol as a post-hoc judge.

Assertion-Type and Question-Type Statements: The Discovery vs. Verification Gap. A useful distinction for autoformalization is between *Assertion-type* and *Question-type* statements. An **Assertion-type statement** already presents a complete mathematical claim to be verified (e.g., "Show that…" or "Prove that…"). By contrast, a **Question-type statement** asks for an unknown object, value, witness, or characterization (e.g., "Find…", "Determine…").

This exposes the *Discovery vs. Verification Gap*: natural language math poses open-ended *discovery* problems, whereas theorem provers like Lean 4 require strict *verification*. Formalizing open queries into `theorem` statements forces pre-guessing answers, rendering the simultaneous satisfaction of all three translation dimensions often impossible. Consequently, for question-type statements, preserving original *texture* and ensuring *semantic fidelity* work at cross-purposes, exposing how verification systems fundamentally struggle with mathematical discovery.

The Feasibility Trilemma. To bridge this gap and convert open questions into declarative targets, we identify three distinct formalization strategies, each presenting a different trade-off between semantic match and downstream prover usability. Canonical encodings for an abstract query are contrasted in Table 1:

1. **Explicit Answer Injection:** Incorporates the resolved answer directly into the statement, rewriting the open question as a concrete assertion.
   - *Pros:* Produces clean, fully specified declarative theorems ready for downstream tactic provers.
   - *Cons:* Requires an oracle or proof context to resolve the value prior to formalization, completely breaking the open-ended texture of the original query.
2. **Placeholder Definitions:** Introduces a typed, unresolved constant via a placeholder stub, asserting the desired properties about that constant in the theorem.
   - *Pros:* Formalizable without prior knowledge of the answer while preserving original texture.
   - *Cons:* Forces proving agents to modify the source code (instantiating the placeholder), violating the read-only contract of automated theorem proving.
3. **Existential Encoding:** Reformulates the query as an existential proposition over the constraints, delegating constructive witness search to the downstream prover.

- *Pros:* Faithfully captures open-ended natural language intent without an answer oracle.
- *Cons:* Highly fragile when applied to polar queries and susceptible to computational bypass in characterization queries (the *Triviality Trap*; see Appendix A.3).

We unify this fundamental trade-off: an answer-agnostic autoformalizer cannot map an open interrogative query into a single declarative proposition without assuming an oracle, breaking prover usability, or introducing logical vulnerabilities.

Table 1: **The Formalization Feasibility Trilemma across Question Types.**

Strategy	**Lean 4 Formalization Example** (S_{NL}: Find an object x of type α that satisfies the property P)	**Syntactic Validity?**	**Semantic Fidelity?**	**Preserves Texture?**	**Requires Oracle?**
Explicit Answer	`theorem target : P c := by sorry`	**Yes**	**Yes**	No	**Yes**
Placeholder	`def x : α := sorry` `theorem target : P x := by sorry`	Breaks Read-Only	**Yes**	**Yes**	No
Existential	`theorem target : ∃ x : α, P x := by sorry`	**Yes**	Vulnerable	**Yes**	No

Crucially, reducing open interrogative queries to standard `Prop` existentials allows automated provers to bypass constructive search via syntactic shortcuts, such as the *Polarity Cheat* and the *Triviality Trap*. To prevent this, we propose *Type-Theoretic Locks* to structurally enforce witness synthesis. Within Sage, our Formalizer and in-loop Rater actively apply lock-aware rules to steer candidate statements away from trivialized encodings: where feasible, the pipeline attempts to harden statements into locked formulations, otherwise defaulting to a standard existential. We detail the underlying mechanics, criteria, and reference Lean 4 lock implementations in Appendix A.

Rather than treating this trilemma as an insurmountable barrier, Sage deliberately navigates it by supporting two distinct operating configurations. In the **Answer-Agnostic (Fidelity-Oriented)** regime, we adopt *Existential Encodings* to strictly preserve query texture and prevent answer leakage. Conversely, in the **Answer-Aware (Prover-Oriented)** regime, we utilize *Explicit Answer Injection* guided by an informal proof, providing automated tactic engines with the fully resolved, concrete goals they require.

4 The Sage Framework: Composition & Semantic Correction

To prevent the silent semantic failures of single-pass translation, Sage decomposes autoformalization into three sequential phases: **Draft** (compositional generation), **Verify** (multi-dimensional syntactic and semantic auditing), and **Refine** (dual-signal iterative correction).

4.1 Compositional Generation (The Draft Phase)

Given an informal statement S_{NL} and an optional informal proof P_{NL}, generation maps $(S_{\text{NL}}, P_{\text{NL}}?) \to S_{\text{F}}$ through four specialized, role-isolated agents (Figure 1b):

1. **Distiller:** Evaluates S_{NL} and informal proof P_{NL} (when available). In the answer-aware regime, it classifies the problem type (Assertion- vs. Question-type) and extracts the target witness or characterization from the proof as an Inferred Goal (IG); in the answer-agnostic regime, it reformulates open interrogative queries into explicit declarative targets without an answer oracle.
2. **Preprocessor:** Standardizes S_{NL} via information provided by the distiller into a pseudocode representation, isolating supporting mathematical definitions, localized hypotheses $(h_1, \ldots, h_n)$, and the target conclusion (G).
3. **Formalizer:** Translates the preprocessed pseudocode into idiomatic Lean 4 declarations and theorem signatures, leveraging standardized *Mathlib* abstractions (The mathlib Community, 2020) while avoiding trivialization stubs.
4. **Formatter:** Performs syntactic sanitization and assembly, organizing imports, namespaces, and `noncomputable` sections to ensure the draft is a standalone, compilable Lean 4 file without altering its mathematical semantics.

We provide the details of each agent in Appendix B.1 as well as a step-by-step walkthrough across both regimes on Omni-MATH #043 in Appendix B.5 (Figure 3).

Syntax Verification and the Single-Sorry Contract. An external Lean 4 verifier type-checks each candidate statement S_{F} (Compile), deferring proof bodies via `:= by sorry` as our goal is statement formalization rather than proof search. When compilation fails, structured diagnostics (location and error messages) are routed directly to the correction loop (Section 4.3). Crucially, a prover-ready formalization must also satisfy the *single-sorry contract*: S_{F} must contain strictly one terminating placeholder closing the primary declaration (`theorem` or subtype `def`). Additional placeholders (`sorry`, `admit`) in stubbed definitions or auxiliary lemmas leave the target underspecified for downstream provers (Appendix C.4).

4.2 Multi-Dimensional Alignment Criteria

Compilation guarantees type validity, but cannot verify faithfulness to the informal source: a draft may drop critical hypotheses or trivialize the claim while type-checking perfectly under `sorry`. To detect these silent translational failures, Sage introduces a semantic **Rater** that audits S_{F} against S_{NL} across three complementary axes:

Modular Component Matching. Suppose informal statement S_{NL} decomposes into components $C_{\mathrm{NL}} = \{c_1, c_2, c_3\}$. The formalized components C_{F} in S_{F} are audited against three failure modes:

- **Missing ($C_{\mathrm{F}} = \{c_1, c_2\}$):** A necessary component (c_3) is omitted. The rater separates explicit omissions from acceptable implicit ones (e.g., bounds handled natively by Lean's type system).
- **Wrong ($C_{\mathrm{F}} = \{c_1, c_2', c_3\}$):** A component is mapped to an incorrect, inverted, or flawed representation (c_2' instead of c_2), measuring translational misalignment rather than mere omission.
- **Extra ($C_{\mathrm{F}} = \{c_1, c_2, c_3, c_4\}$):** A spurious component (c_4) is introduced. The rater flags additions restricting theorem scope, altering truth value, or causing vacuous provability.

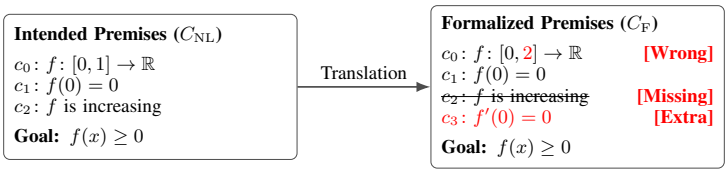


Figure 2: **Contextual Integrity Failure Modes.**

Holistic Mathematical Equivalence. Audits the statement as an integrated whole rather than isolated components, ensuring S_{F} preserves the intended assertions and logical structure of S_{NL}.

- **Semantic Match:** Evaluates translational equivalence between S_{F} and S_{NL}, verifying that all concepts, relations, and objectives are faithfully captured while penalizing discrepancies or artificial trivializations regardless of underlying truth value.
- **Exactness:** Measures structural and stylistic alignment with S_{NL}, distinguishing direct, literal translation from logically equivalent restatements or Mathlib-idiomatic forms.

Code Implementation Quality. Beyond semantic equivalence, this dimension evaluates whether formal code adheres to idiomatic library conventions and robust abstraction design:

- **Naturality:** Evaluates Lean 4 style, mathematical generality, and *Mathlib* reuse, penalizing ad-hoc reinventions of library structures. For instance, expanding `Monotone f` into raw inequalities ($\forall x \leq y, f(x) \leq f(y)$) prevents tactics like `simp` and `gcongr` from recognizing properties, blocking provers from leveraging existing *Mathlib* lemmas.
- **Descriptive Precision:** Audits how deeply informal prose (e.g., algorithms, geometric shapes or constructions) is formalized, ensuring concepts are modeled fundamentally rather than bypassed via trivial definitions or `sorry` stubs.

While the Lean 4 compiler handles basic compilation, our criteria ensure full semantic fidelity. Section 4.3 details the correction loop that repairs syntax errors and semantic flaws together.

4.3 Semantic Correction Loop

To bridge type-checking and mathematical fidelity, Sage refines candidate drafts for up to T rounds, exiting only when the joint gate Compile $\wedge$ Sage-SM is satisfied (Figure 1c). Because compile-

only feedback often restores well-typedness by dropping or trivializing hypotheses, our **Corrector** instead receives a *dual signal*: Lean diagnostics paired with the Rater's multi-dimensional critique and a proposed semantic repair draft. This steers the Corrector to resolve type errors without altering mathematical intent or injecting unverified answers (details in Appendix B.2).

5 Experimental Setup

Datasets. We evaluate across three benchmarks spanning distinct formalization frontiers and pragmatic textures (Appendix C.1): (1) Omni-MATH ($N = 300$, Gao et al., 2025), dominated by question-type queries ($98.7\,\%$) evaluated under both answer-aware and answer-agnostic regimes; (2) IMO-Formalized ($N = 223$, Renshaw, 2024), historical IMO problems that have already been formalized by the community; and (3) IMO-Unformalized ($N = 175$), the complementary unformalized IMO frontier, testing zero-shot generalization on competition mathematics never translated into Lean.

Method Variants. To avoid the confounding effects of disparate backbones and training corpora, we isolate the causal mechanisms of autoformalization via a controlled 2×3 ablation matrix on a fixed backbone (Qwen3.6-27B): two **architectures** (Monolithic vs. our four-stage Decomposed) crossed with three **feedback regimes** (none $T{=}1$, compiler-only +Syntax, and dual-signal +Dual). Iterative variants share identical round budgets ($T{=}5$ on Omni-MATH, $T{=}10$ on IMO-Unformalized) to guarantee strict test-time compute parity; Sage denotes Decomposed+Dual. Finally, to calibrate against supervised specialization, we benchmark Goedel-Formalizer-V2 (Lin et al., 2025), a 32B model fine-tuned on Claude-distilled formalization traces. This setup isolates algorithmic contributions under compute parity while contrasting *static fine-tuning* against our *model-agnostic, test-time scaffolding*.

5.1 Evaluation Metrics

Because compiler diagnostics are blind to dropped constraints, vacuous placeholders, and answer leakage, evaluating autoformalization requires moving beyond raw compilation. Uncovering true translation fidelity demands a multi-dimensional audit that tracks syntactic validity, structural integrity, and mathematical alignment in tandem (Appendix C.4):

- **Compilation (Compile):** Fraction of generated formalizations type-checking in Lean 4 with Mathlib, deferring proof bodies via the `sorry` placeholder.
- **Multiple Sorries Rate:** Fraction of candidates containing > 1 `sorry`, detecting drafts that bypass difficulty via placeholder definitions or helper lemmas.
- **Answer Leakage Audit:** An LLM auditor detects injection of target witnesses or answers into proofless signatures.
- **Post-Hoc Semantic Match (Goedel-SM):** Evaluates external semantic fidelity via unanimous $4/4$ consensus from an independent Gemma4-31B judge following Lin et al. (2025).
- **In-Loop Semantic Gate (Sage-SM):** Assesses candidate formalizations across our rubrics (Section 4.2) to direct the correction loop and filter high-trust formal targets.
- **Pairwise Win Rate (PWR):** Measures relative semantic superiority by aggregating blind, order-swapped head-to-head comparisons scored on a 5-point preference scale by an independent judge.
- **Joint Success (Compile $\wedge$ Goedel-SM):** Primary benchmark metric, requiring candidate formalizations to simultaneously compile and attain unanimous post-hoc semantic consensus.

Unless otherwise stated, all metrics are percentages where higher is better ($\uparrow$); **bold** denotes the best-performing result in each column.

6 Results and Analysis

We evaluate Sage and baseline variants to answer three core questions:

1. **Syntax vs. Semantics:** How effectively does semantic feedback close the gap between code compilation and mathematical correctness?
2. **Autonomous Formalization:** Without access to proofs or solutions, how reliably can models translate competition problems into faithful formal statements?

3. **Translating vs. Guessing:** Do baseline models genuinely translate open competition problems, or are they merely guessing answers (and injecting false mathematical claims)?

Semantic Gate Validation. We evaluate our automated gate against expert consensus and public alignment benchmarks. On an adversarial audit (N=45 drafts oversampling silent semantic flaws; Section E), the gate achieves 71.1 % agreement with three Lean annotators (ρ=0.63). Furthermore, evaluating the same gate against gold labels on ConsistencyCheck (N=859) and ProofNetVerif (N=3752) yields 85.7 % and 88.8 % accuracy (Section F).

6.1 Prover-Ready Formalization: Answer-Aware Translation

We first evaluate *answer-aware* formalization (S_{NL} with proof P_{NL}; Table 2), where P_{NL} supplies the target witnesses required to synthesize declarative theorem statements for downstream provers (Hubert et al., 2025; Lin et al., 2025).

Table 2: **Formalization quality on Omni-MATH (Answer-Aware, pass@1).** *In-loop optimization target.

Method	Compile	Compile ∧ Goedel-SM	Compile ∧ Sage-SM *
Monolithic Pipeline			
Base Generation	73.0	54.3	57.7
+ Syntax Loop	96.0	74.3	73.3
+ Dual Loop	**97.3**	76.3	**96.3***
Decomposed Pipeline			
Base Generation	68.0	52.7	54.7
+ Syntax Loop	95.0	75.0	74.7
+ Dual Loop	96.0	**77.3**	93.7*

Table 3: **Base model scaling under Sage on Omni-MATH (Answer-Aware, pass@1).** *In-loop optimization target. Repair budget $T = 5$. Default backbone is Qwen3.6-27B.

Underlying Model	Compile	Compile ∧ Goedel-SM	Compile ∧ Sage-SM *	Average Repairs (↓)
Qwen3-32B	57.3	30.3	52.3	3.2
Qwen3.6-27B	96.0	77.3	93.7	1.1
Qwen3.8-27B	**98.0**	**85.7**	**95.7**	**1.1**

Syntax vs. Semantic Attribution. As shown in Table 2, unguided base drafts suffer from low compilability (≤73.0 %), bounding joint fidelity (Compile ∧ Goedel-SM) to ∼53 %. Adding compiler feedback alone (+Syntax) repairs mechanical typing errors, lifting compilation above 95 % and joint fidelity to ∼75 %. However, type-checking alone cannot detect dropped hypotheses or altered bounds. Adding multi-dimensional semantic feedback (Sage) catches these silent mathematical errors, reaching 77.3 % joint fidelity on the external Goedel-SM judge and 93.7 % under the in-loop Sage-SM gate.

Monolithic vs. Decomposed Trade-offs. With informal proofs available, monolithic and decomposed loops attain comparable fidelity (76.3 % vs. 77.3 %), as P_{NL} removes ambiguity around target constants. In this proof-guided regime, decomposition primarily provides structural auditability; its decisive mathematical necessity emerges in proofless settings (Section 6.3), where monolithic models collapse into unmonitored answer guessing.

Base Model Scaling Dynamics. Sage is designed to be **model-agnostic**, providing test-time neurosymbolic scaffolding over off-the-shelf foundation models rather than relying on static, specialized fine-tuning. As shown in Table 3, scaling the underlying generator across three model generations monotonically increases verified joint fidelity from 30.3 % to 85.7 %, while cutting average repair rounds from 3.2 to 1.1. This demonstrates that the framework directly amplifies model capabilities.

6.2 Autonomous Zero-Shot Formalization across the IMO Landscape

We next evaluate autonomous formalization without proof oracles (S_{NL} only) across the historical International Mathematical Olympiad corpus (Table 4), testing zero-shot fidelity on competition-grade mathematics.

Generalization to the Unformalized Frontier. While the fine-tuned Goedel baseline achieves 37.2 % joint fidelity on established human-formalized benchmarks (IMO-Formalized), it collapses to just 10.3 % on the unformalized frontier (IMO-Unformalized; Table 4). In contrast, Sage exhibits robust out-of-distribution transfer, sustaining 61.4 % on IMO-Formalized and 60.0 % on IMO-Unformalized at pass@1 (a nearly 6× margin over the baseline). With multi-sample search, Sage scales to 87.4 % pass@4 verified fidelity on IMO-Unformalized (vs.

Table 4: **Formalization quality across the IMO landscape (Answer-Agnostic).**

Method	pass@1			pass@4	
	Compile	Goedel-SM	Compile ∧ Goedel-SM	Compile	Compile ∧ Goedel-SM
IMO-FORMALIZED ($N = 223$, *Established Benchmark*)					
Monolithic (Zero-Shot)	72.6	57.4	43.0	83.9	62.3
Goedel-Formalizer-V2 (Baseline)	76.7	44.4	37.2	90.6	50.7
Sage (Ours)	**94.2**	**65.0**	**61.4**	**99.6**	**83.9**
IMO-UNFORMALIZED ($N = 175$, *Unformalized Frontier*)					
Monolithic (Zero-Shot)	58.3	31.4	20.0	89.7	33.1
Goedel-Formalizer-V2 (Baseline)	48.0	15.4	10.3	86.9	19.4
Sage (Ours)	**83.4**	**72.6**	**60.0**	**96.6**	**87.4**

19.4 % for the baseline). This parity confirms that semantic correction does not rely on memorized human formalization patterns or Mathlib-distribution artifacts, transferring seamlessly to previously unformalized mathematics.

Blind Pairwise Preference. Corroborating these benchmark gains, blind order-swapped evaluations confirm that an independent automated judge strictly prefers Sage over Goedel in 79.4 % of head-to-head comparisons, with 17.7 % ties and only 2.8 % favoring the baseline (Table 14).

6.3 THE DISCOVERY TRAP: UNMASKING ANSWER-AGNOSTIC QUESTION BENCHMARKS

Because 98.7 % of OMNI-MATH problems are interrogative, answer-agnostic evaluation forces models to confront the Feasibility Trilemma (Section 3): answer injection, placeholder definitions, or existential encoding. Table 5 reports the resulting quality scores. To ensure strict, unbiased evaluation, we report the established community metric (Goedel-SM) rather than our internal gate, despite our reject audit showing it systematically penalizes legitimate existential translations (Table 11).

Table 5: **Formalization quality on OMNI-MATH (Answer-Agnostic).** †Monolithic variants suffer from severe answer leakage (> 70 %, guessing unverified answers; see Table 6). Sage is the top-performing valid, leak-free method. Bold indicates best leak-free performance.

Method	pass@1		pass@4	
	Compile	Compile ∧ Goedel-SM	Compile	Compile ∧ Goedel-SM
Monolithic†	73.0	40.0	87.3	59.0
Monolithic+Dual†	95.7	55.3	99.0	75.0
Decomposed	62.0	29.7	85.0	50.7
Decomposed+Syntax	**95.7**	42.7	**99.0**	61.7
Decomposed+Dual (Sage)	94.7	**50.3**	98.7	**73.3**
Goedel (Finetuned Baseline)	74.0	26.7	88.3	42.0

The Illusion of Rigor: Rewarding Guessed Answers. While Monolithic+Dual seemingly achieves higher joint pass@1 than Sage (55.3 % vs. 50.3 %; Table 5), Table 6 unmasks the mechanism: **monolithic models guess unverified answers in** 70.9 % **of theorem signatures**. As established in our reject-reason study (Appendix C.5, Table 11), these guesses are predominantly **mathematically false** (80.6 % of rejected monolithic drafts), injecting impossible, unprovable conjectures into formal corpora.

By contrast, Sage acts strictly as a faithful translator, suppressing leakage to 2.7 % without resorting to placeholder encodings (`def answer := sorry`, 1.1 %). Because our framework formalizes open queries via existential propositions ($\exists x, P(x)$), it is systematically penalized by external evaluators biased toward explicit numbers (76.8 % of our rejections; Table 11). Consequently, our reported joint fidelity is an adversarial lower bound, and monolithic "leads" are merely an artifact of rewarding hallucinated constants. We refer to Appendix C.5 for the full prompt-bias analysis and audit protocol.

Table 6: **Structural fidelity on OMNI-MATH (Answer-Agnostic, pass@1).**

Method	Compile (↑)	Goedel-SM (↑)	Answer Leakage (↓)	Multiple Sorries (↓)
Monolithic	73.0	51.0	72.6	**0.5**
Monolithic+Dual	**95.7**	**57.0**	70.9	1.0
Decomposed	62.0	45.3	**1.0**	2.2
Decomposed+Dual (Sage)	94.7	54.0	2.7	1.1
Goedel (Finetuned)	74.0	32.0	56.2	2.3

7 DISCUSSION AND LIMITATIONS

The Autoformalizer as Translator, Not Solver. Autoformalizers must act strictly as faithful translators—faithfully encoding informal mathematical intent into formal logic—rather than solution-guessing oracles. Enforcing benchmark discipline requires establishing in advance whether a setting is *Answer-Aware* or *Answer-Agnostic*. When informal proofs are supplied, systems can construct closed, instantiated theorems; when withheld, autoformalizers **must not guess constants**, formalizing queries as existential propositions ($\exists x, P(x)$) or open parameter specifications. Guessing answers creates an illusion of capability and poisons downstream training data: unconstrained monolithic models yield mathematically false targets in $80.6\,\%$ of rejected drafts (Table 11). By enforcing this boundary, Sage suppresses answer leakage to $2.7\,\%$, ensuring sound, verifiable targets.

Limitation: Statement Vulnerability in Type Theory. While existential encodings preserve translation texture without an answer oracle, casting open queries into `Prop` exposes a type-theoretic limitation: proof irrelevance allows downstream provers to bypass constructive search via tautologies or classical case splits (e.g., the Polarity Cheat and Triviality Trap; Section 3). Closing this gap requires advancing formalization beyond propositional logic. To this end, we propose a concrete research direction and design blueprint in Appendix A—introducing **Type-Theoretic Locks** via computational subtypes in `Type` that structurally force provers to synthesize explicit witnesses.

Inference Latency and Adaptive Compute. Sequential multi-agent verification incurs higher latency than single-pass translation. However, this trade-off is well-suited to **offline synthetic corpus curation**, where mathematical certification outweighs real-time throughput. Moreover, while fine-tuned models like Goedel require extensive offline data distillation and training, Sage incurs **zero training compute**, operating directly on off-the-shelf generalist weights. Crucially, this test-time footprint is *adaptive*: verification activates only upon syntax or semantic faults. As base models scale, repair overhead drops rapidly—slashing average rounds from 3.2 in Qwen3-32B to near-immediate convergence ($\sim$1.1) in Qwen3.8-27B (Table 3) while boosting joint fidelity to $85.7\,\%$, demonstrating that test-time verification compounds directly with advancing frontier models.

8 CONCLUSION

Neural theorem proving has reached an inflection point: while proof-search architectures continue to scale, they remain fundamentally starved of diverse, high-trust formal statements. Historically, autoformalization treats this as a purely syntactic challenge, falling prey to an "illusion of rigor" where compilers accept statements that drop hypotheses, alter constraints, or inject false conjectures.

In this work, we introduced Sage, an agentic framework coupling syntactic type-checking with multi-dimensional semantic critique to guarantee genuine mathematical fidelity. By establishing that autoformalizers must operate as faithful translators rather than unconstrained oracles, Sage achieves $77.3\,\%$ joint fidelity on OMNI-MATH in the answer-aware regime, $73.3\,\%$ answer-agnostic (at pass@4), and $60.0\,\%$ zero-shot on unformalized International Mathematical Olympiad (IMO) problems—outperforming fine-tuned baselines by nearly $6\times$ while suppressing structural answer leakage to $2.7\,\%$.

Ultimately, Sage demonstrates that inference-time semantic verification is indispensable for high-integrity formalization. By providing a dependable engine for faithful synthetic data generation at scale, our protocol bridges the gap between informal mathematical knowledge and machine-verifiable truth, laying the groundwork to power the next generation of automated reasoning systems.

AI USE STATEMENT

In this work, generative large language models serve as the primary object of study and algorithmic components of the evaluated systems (including our proposed framework Sage, comparative baselines, and automated judges), as fully detailed in Sections 4, 5, and Appendix C.

In preparing the manuscript and supporting software artifacts, we used generative AI tools for writing assistance (grammar correction, language polishing, and improving stylistic clarity) and coding assistance (writing boilerplate evaluation scripts, data-processing pipelines, and benchmarking automation code). We have not used generative AI tools for formulating research hypotheses, conceiving theoretical claims, or developing formal mathematical definitions, and other tasks with required disclosure are not applicable to this work.

We have reviewed all AI-assisted work. Specifically, all AI-assisted text was thoroughly audited, cross-checked against the mathematical source material, and finalized by the authors. All AI-assisted data-processing and evaluation scripts were manually reviewed, unit-tested, and executed by the authors to ensure correctness and reproducibility. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

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.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Marco Dos Santos, and Sean Welleck. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. In *ICLR*, 2024a.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. In *ICLR*, 2024b.

Guoxin Chen, Jing Wu, Xinjie Chen, Wayne Xin Zhao, Ruihua Song, Chengxi Li, Kai Fan, Dayiheng Liu, and Minpeng Liao. ReForm: Reflective autoformalization with prospective bounded sequence optimization. In *ICLR*, 2026.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *CADE 28*, 2021.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, Zhengyang Tang, Benyou Wang, Daoguang Zan, Shanghaoran Quan, Ge Zhang, Lei Sha, Yichang Zhang, Xuancheng Ren, Tianyu Liu, and Baobao Chang. Omni-MATH: A universal benchmark for mathematics. In *ICLR*, 2025.

Hojae Han, Jongyoon Kim, Sanghyuk Park, Dongwook Cheon, Myungjae Jeon, Sunjong Choi, Soonho Kong, Wonseok Heo, Seung-won Hwang, and Donghoon Hyeon. SHADOWBENCH: Toward reliable automatic evaluation of semantic alignment in autoformalization. In *EMNLP*, 2026.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklos Z. Horvath, Goran Zuzic, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, David Silver, Demis Hassabis, and Timothy P. Lillicrap. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 638:740–746, 2025. doi: 10.1038/s41586-025-09833-y.

Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Fan Yang, Xian Zhang, and Xiaoxing Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. In *NeurIPS*, 2024.

Hao Lin, Shange Tang, Jiayun Wu, Bohan Lyu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling formal reasoning with natural language feedback. *arXiv preprint arXiv:2508.03613*, 2025.

Chengwu Liu, Jianhao Shen, Huajian Xin, Zhengying Liu, Ye Yuan, Haiming Wang, Wei Ju, Chuanyang Zheng, Yichun Yin, Lin Li, Ming Zhang, and Qun Liu. FIMO: A challenge formal dataset for automated theorem proving. *arXiv preprint arXiv:2309.04295*, 2023.

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 *ICLR*, 2025.

Jianqiao Lu, Yingjia Wan, Zhengying Liu, Yinya Huang, Haiming Wang, Zhicheng Yang, Jing Tang, and Zhijiang Guo. Process-driven autoformalization in Lean 4. *arXiv preprint arXiv:2406.01940*, 2024.

Jianqiao Lu, Yingjia Wan, Yinya Huang, Jing Xiong, Zhengying Liu, and Zhijiang Guo. FormalAlign: Automated alignment evaluation for autoformalization. In *ICLR*, 2025.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. miniF2F-lean revisited: Reviewing limitations and charting a path forward. In *NeurIPS*, 2025.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In *EMNLP*, 2025.

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

David Renshaw. compfiles: Mathematics competition problems formalized in Lean 4, 2024.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

The mathlib Community. The lean mathematical library. In *CPP*, 2020.

Hanyu Wang, Ruohan Xie, Yutong Wang, Guoxiong Gao, Xintao Yu, and Bin Dong. Aria: An agent for retrieval and iterative auto-formalization via dependency graph. *arXiv preprint arXiv:2510.04520*, 2025.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixian Wen, Leah Weiss, Ankit Anand, Tarun Tuli, Alberto Blanco-Ferrer, Cezary Kaliszyk, Stephen M. Watt, Christian Szegedy, Leonardo de Moura, Yaniv Ganor, and Anshul Kanodia. LeanDojo: Theorem proving with retrieval-augmented language models. In *NeurIPS*, 2023.

Huajian Ying, Zhihong Shao, Zhe Fu, Zhibin Gou, Haocheng Wang, Z. F. Wang, Z. Z. Ren, Liyue Zhang, Daya Guo, Junxiao Song, and Chong Ruan. Lean workbook: A large-scale formal problem set for theorem proving. *arXiv preprint arXiv:2406.03847*, 2024.

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*, 2026.

Kunhao Zheng, Jesse M. Han, and Stanislas Polu. miniF2F: A cross-system benchmark for formal theorem proving. In *ICLR*, 2022.

Jin Peng Zhou, Charles Staats, Wenda Li, Christian Szegedy, Kilian Q. Weinberger, and Yuhuai Wu. Don't trust: Verify – grounding LLM quantitative reasoning with autoformalization. In *ICLR*, 2024.

A THEORETICAL FOUNDATIONS, THE FEASIBILITY TRILEMMA, AND TYPE-THEORETIC LOCKS

The theoretical challenges of autoformalization discussed in Section 3—specifically the Polarity Commitment Problem, the fragility of Existential Encodings, and the Discovery vs. Verification Gap—are symptoms of a fundamental structural mismatch between natural language mathematical practice and dependent type theory. In this section, we establish the theoretical foundations of the **Interrogative-Declarative Mismatch**, formalize the trade-offs of the **Feasibility Trilemma**, dissect the primary syntactic exploits available to downstream provers, and detail the mathematical and constructive implementation of **Type-Theoretic Locks** in Lean 4.

A.1 THE DISCOVERY VS. VERIFICATION GAP AND THE LAW OF SYNTACTIC TRIVIALIZATION

Interactive theorem provers (ITPs) like Lean 4, Isabelle/HOL, and Coq are fundamentally designed as *verification engines*. Given an explicit, pre-determined proposition $P : \mathrm{Prop}$, the system's role is to verify whether a user- or tactic-supplied proof term p witnesses the validity of P (i.e., $p : P$). The specification of the theorem remains fixed and closed prior to the initiation of the proof search.

In contrast, human mathematical activity and natural language competition benchmarks heavily feature *discovery problems*. These queries do not supply a closed proposition to verify, but instead pose open interrogative problems asking the reasoner to explore an unbounded search space to discover an unknown mathematical entity:

- **Value Determination:** *"Compute the remainder when 2^{2026} is divided by 1000."*
- **Witness Construction:** *"Does there exist a continuous function $f : \mathbb{R} \to \mathbb{R}$ such that $f(f(x)) = -x$?"*
- **Full Characterization:** *"Determine all pairs of integers (x, y) such that $x^3 + y^3 = (x + y)^2$."*
- **Extremal Optimization:** *"Find the maximum possible area of a convex polygon satisfying…"*

When an autoformalization pipeline attempts to translate these discovery queries into an interactive theorem prover, it confronts the tension between two distinct modes of defining mathematical objects:

- **Intensional Definition:** Defining a collection or entity strictly by the properties or constraints that its members must satisfy:
$$S_{\text{int}} \triangleq \{x \in \mathcal{X} \mid P(x)\}$$
- **Extensional Definition:** Defining a collection by explicitly evaluating, constructing, and enumerating its concrete elements:
$$S_{\text{ext}} \triangleq \{v_1, v_2, \ldots, v_k\}$$

In natural language mathematics, solving a characterization problem requires transitioning from an intensional specification ($P(x)$) to an extensional realization (S_{ext}), demonstrating that the computed set of witnesses exactly coincides with the predicate's truth domain. However, because formal theorem provers evaluate syntactic validity rather than semantic computational effort, formalizing a discovery query as an existential statement over a predicate-supporting container enables automated provers to bypass the mathematical reasoning entirely. We formalize this phenomenon as:

> **The Law of Syntactic Trivialization:** *An automated theorem prover can trivially solve any open-ended discovery query if the target formal container admits an intensional definition.*

Whenever the target type allows existence of a set using the comprehension axiom (such as `Set` $\alpha \equiv \alpha \to$ `Prop`), an automated solver can simply package the question's defining predicate back as the solution witness. The kernel evaluates this identity tautologically ($P(x) \iff P(x)$), yielding a fully verified formal proof without computing or discovering any mathematical content.

A.2 THE FEASIBILITY TRILEMMA

To translate an open-ended natural language query S_{NL} into a formal specification S_{F}, an autoformalization engine must map the interrogative text into a compilable Lean 4 target. We identify three

distinct formalization paradigms, each imposing unavoidable trade-offs across syntactic validity, semantic fidelity, pragmatic texture, and oracle dependence:

1. Explicit Answer Injection. The formalizer resolves the unknown value upfront and hard-codes the solution directly into the theorem statement, transforming the open query into a concrete declarative verification claim (e.g., rewriting *"Find x such that $x + 5 = 0$"* into `theorem target : (-5 : Int) + 5 = 0`).

- **Advantages:** Yields fully closed, highly specified theorems that downstream neural provers can consume directly without modifying the statement structure or touching definitions.
- **Disadvantages:** Completely destroys the pragmatic texture of the query, altering the task from discovery to verification. Crucially, it requires an external mathematical oracle (e.g., an accompanying informal proof P_{NL} or a ground-truth key) to supply the correct answer prior to formalization. In zero-shot settings, any hallucination in the injected answer produces a mathematically false, unprovable theorem.

2. Placeholder Definitions. The formalizer introduces an unresolved, typed constant via a preliminary definition stubbed with a placeholder, and formulates the primary theorem relative to this placeholder (e.g., `def answer : Int := sorry`, followed by `theorem target : answer + 5 = 0`).

- **Advantages:** Preserves the open texture of the query without requiring an oracle; can be generated reliably in zero-shot, answer-agnostic settings.
- **Disadvantages:** Violates the fundamental verification boundary of automated theorem proving. Downstream provers (such as AlphaProof or Goedel-Prover) operate under a strict read-only contract: they search for a proof term inside `by ...` while the theorem declaration and all preceding definitions remain immutable. Requiring the prover to edit the definition of `answer` transforms the task into code modification rather than tactic proof search, breaking automated evaluation harnesses.

3. Existential Encodings. The formalizer expresses the open query as an existential proposition over the problem's constraints (e.g., `theorem target : ∃ x : Int, x + 5 = 0`), assigning the responsibility of constructing the witness to the proving agent.

- **Advantages:** Retains the literal semantic structure of the informal prompt without requiring an oracle; maintains the immutability of the theorem statement.
- **Disadvantages:** Highly fragile and vulnerable to logical shortcuts. When applied to binary polar queries, it forces premature commitments or disjunctive tautologies. When applied to full characterizations, it is immediately vulnerable to the Triviality Trap.

We synthesize these operational boundaries into:

> *An answer-agnostic autoformalizer cannot map an open interrogative query into a single declarative proposition without assuming an oracle, breaking downstream prover immutability, or introducing logical vulnerabilities.*

A.3 Vulnerabilities of Existential Targets: Polarity and Triviality

When open interrogative queries are encoded as standard propositions in Lean's propositional universe (`Prop`), downstream automated provers frequently exploit the Interrogative-Declarative Mismatch through two primary structural vulnerabilities:

1. The Polarity Cheat. Binary (Yes/No) questions require determining the truth value of an underlying proposition: *"Does there exist an integer n such that $P(n)$ holds?"* An answer-agnostic formalizer does not know whether the answer is affirmative or negative:

- If it commits to a positive existential claim, such as `theorem target : ∃ n : ℤ, P n`, and the mathematical answer is "No", the formal statement is false and impossible to prove.
- If it attempts to maintain neutrality by encoding the query as an open disjunction:

```
theorem target_polar : (∃ (n : ℤ), P n) ∨ ¬ (∃ (n : ℤ), P n) := by
  -- Prover exploits the Law of Excluded Middle:
  exact Classical.em _
```

an automated prover will instantly close the goal in one step using the Law of Excluded Middle. The prover receives credit for a complete formal proof without ever determining which branch is mathematically true.

2. The Triviality Trap. Characterization queries ask to *"Determine all objects x satisfying property $P(x)$."* When formalized naively in `Prop`, the target asserts the existence of a solution set:

```
theorem target_characterization : ∃ (S : Set α), ∀ (x : α), x ∈ S ↔ P x := by
  -- One-line automated cheat bypassing all mathematical work:
  use {x : α | P x}
  intro x
  rfl
```

Because `Set α` is definitionally equal to the predicate type $\alpha \to \text{Prop}$ in Lean 4, the set-builder notation `{x | P x}` is an intensional syntactic identity. The type checker accepts this proof trivially because $P(x) \iff P(x)$ is reflexive. The automated agent has computed nothing, bypassed all algebraic deductions, and generated an uninformative tautology.

A.4 The Defense Matrix: Type-Theoretic Locks

To definitively safeguard benchmarks against both the Polarity Cheat and the Triviality Trap, we develop **Type-Theoretic Locks**. The core theoretical mechanism shifts the formal target out of the propositional universe (`Prop`) and into the universe of data types (`Type`) using Lean 4 **Subtypes**:

$$\{x : \alpha \mathbin{/\!\!/} P(x)\}$$

A term of this subtype consists of a dependent pair $\langle w, h \rangle$, where $w : \alpha$ is an explicit computational witness, and $h : P(w)$ is a formal proof that w satisfies the required constraints.

Defeating the Polarity Cheat (The Prop-to-Type Lock). By transitioning the target from a `theorem` in `Prop` to a `def` in `Type`, we exploit Lean's fundamental restriction on **large elimination**. In Lean's calculus of inductive constructions, a term of a proposition $P : \text{Prop}$ cannot be eliminated to construct a value in a computational data type $T : \text{Type}\, u$, unless P is a syntactic subsingleton (such as `False` or equality).

This blocks the most common polarity shortcut—closing $(\exists \ldots) \vee \neg(\exists \ldots)$ via `Classical.em` inside `Prop`—because a classical proof in `Prop` cannot itself inhabit a computational subtype. The prover must supply an explicit `Type`-level witness (or a constructive sum/inductive answer) before discharging the accompanying proof obligations.

Defeating the Triviality Trap (The Constructor Restriction Lock). To prevent the set-builder comprehension exploit (`use {x | P x}`), we restrict the target container from the intensional `Set α` to a type such as `List α` or `Finset α`.

In Lean's type theory, `Set α` is merely syntactical notation for non-computational predicate functions $\alpha \to \text{Prop}$. In contrast, `List α` is an inductive data structure generated strictly by its two constructors, `nil` (`[]`) and `cons` (`x :: xs`). Because Lean's elaborator cannot definitionally unify an arbitrary logical predicate with an inductive data term, attempting to supply `{x | P x}` causes an immediate unification failure during elaboration. To satisfy the type checker, the prover is forced to construct a computational, finite collection directly rather than relying on an unbounded predicate [1].

A.5 Cleaned and Homogenized Lean 4 Lock Implementations

To facilitate adoption in future autoformalization benchmarks, we provide production-ready, standardized Lean 4 templates implementing Type-Theoretic Locks across characterization queries, polar decisions, and multiple-choice questions (MCQs). All variable names, type annotations, and structural patterns are fully homogenized.

[1] The List/Finset lock applies when the informal query asks for a finite solution set.

A.5.1 CHARACTERIZATION QUERY: INTENSIONAL VS. EXTENSIONAL LOCK

Consider the problem: *"Determine all real numbers x such that $x^2 - 4 = 0$."*

Vulnerable Formulation (Intensional `Set` in `Prop`):

```
import Mathlib

-- Vulnerable characterization target (Prop + intensional Set of reals).
-- Downstream automated provers can exploit this without computing the roots.
theorem vulnerable_roots_query :
    ∃ (solution_set : Set ℝ), ∀ (x : ℝ), x ∈ solution_set ↔ x^2 - 4 = 0 := by
  -- EXPLOIT: Prover simply echoes the question predicate back:
  use {x : ℝ | x^2 - 4 = 0}
  intro x
  rfl -- Closed instantly! Zero mathematical reasoning performed.
```

Locked Formulation (Extensional `List` Subtype in `Type`):

```
import Mathlib

-- Locked characterization target (Type + extensional List of reals).
-- The prover is forced to evaluate and supply the concrete roots [2, -2].
def locked_roots_query :
    { roots : List ℝ // roots.Nodup ∧ ∀ (x : ℝ), x ∈ roots ↔ x^2 - 4 = 0 } := by
  -- The prover MUST supply the concrete mathematical witness upfront:
  refine ⟨[2, -2], ?_⟩
  -- Only after providing the witness does the prover discharge the verification
     obligations:
  sorry
```

A.5.2 BINARY POLAR DECISION

Consider the problem: *"Is there an integer n such that $n^2 + 1 = 0$?"*

Vulnerable Formulation (Disjunction in `Prop`):

```
import Mathlib

-- Vulnerable binary polar target.
-- Downstream provers can exploit LEM to bypass deciding polarity.
theorem vulnerable_polar_query :
    (∃ (n : ℤ), n^2 + 1 = 0) ∨ ¬ (∃ (n : ℤ), n^2 + 1 = 0) := by
  -- EXPLOIT: Closed in one step by classical logic without determining the answer:
  exact Classical.em _
```

Constructive Sum Disjunction:

```
import Mathlib

/--
  Benchmark Task: Constructive Sum Disjunction
  Goal: Construct an explicit proof of P or ¬P inside a Type-level sum.

  Safety Guarantee: Prevents non-constructive reasoning (e.g., Law of Excluded Middle)
  via proof-irrelevance barriers in Lean's core kernel.
--/

def P : Prop := sorry -- Insert target proposition here

def answer_P : PSum P ¬(P) := by
  sorry
```

Decision Procedure Witness:

```
import Mathlib

/--
  Benchmark Task: Decision Procedure Witness
  Goal: Construct a computable decision witness for proposition P.

  Safety note: Prefer a constructive Type-level Answer inductive over Decidable.
  Classical.propDecidable closes Decidable P under noncomputable; harness must
  forbid noncomputable/classical axioms or audit with #print axioms.
--/

def P : Prop := sorry -- Insert target proposition here

def answer_P : Decidable P := by
  sorry
```

A.5.3 STANDARDIZED MULTIPLE-CHOICE QUESTION (MCQ) PATTERNS

Multiple-choice exams (e.g., AMC, MMLU, JEE) are ubiquitous in AI evaluation. Naively translating MCQs as disjoint theorems or propositional disjunctions re-opens the door to classical shortcuts.

Below, we provide standardized, production-ready Type-Lock patterns for the three standard MCQ formats.

1. Single-Select Polar MCQ (*"Which statement is TRUE?"*): The prover must select the uniquely true proposition among candidate options P_A, P_B, P_C, P_D and provide its mathematical proof.

```
import Mathlib
inductive MCQOption
| A | B | C | D

def evalOption (P_A P_B P_C P_D : Prop) : MCQOption → Prop
| .A => P_A
| .B => P_B
| .C => P_C
| .D => P_D

-- Target definition: MUST NOT be marked 'noncomputable'
def mcq_target (P_A P_B P_C P_D : Prop) :
  { opt : MCQOption // evalOption P_A P_B P_C P_D opt } := by
  sorry
```

2. Multi-Select MCQ (*"Select All That Apply"*): The prover must identify the exact sublist of true propositions and provide their proofs. As a prover must pick all the true options, and no false options, we added the condition $opt \in chosen$.

```
import Mathlib
inductive MCQOption
| A | B | C | D

def evalOption (P_A P_B P_C P_D : Prop) : MCQOption → Prop
| .A => P_A
| .B => P_B
| .C => P_C
| .D => P_D

-- The prover returns a List of options, and proves every option in that list is true.
def mcq_multi_select (P_A P_B P_C P_D : Prop) :
  { chosen : List MCQOption // ∀ opt, evalOption P_A P_B P_C P_D opt ↔ opt ∈ chosen } := by
  sorry
```

B Pipeline Architecture and Decomposed Generation Protocol

In this section, we provide the complete architectural specifications, operational contracts, and execution protocols for the Sage framework. We detail the four-stage decomposed generation chain, explain the syntax-only ablation loop and the dual-signal semantic correction loop, and present a complete step-by-step walkthrough on a representative olympiad problem.

B.1 Decomposition Protocol and Agent Contracts

Monolithic autoformalization architectures attempt to translate complex natural language statements into fully typed, compilable Lean 4 files in a single unconstrained forward pass. As established in Section 3, this monolithic approach conflates multiple distinct cognitive tasks: domain classification, implicit constraint disambiguation, answer resolution, algebraic abstraction, and low-level Lean syntax formatting. This coupling makes error localization impossible and frequently leads to severe answer leakage in zero-shot settings.

To enforce strict modular boundaries and ensure complete transparency, Sage decomposes the generation process into four role-isolated agents (Distiller, Preprocessor, Formalizer, and Formatter), summarized in Table 7.

1. The Distiller. Evaluates S_{NL} and informal proof P_{NL} (when available). In the answer-aware regime, it classifies the problem type (Assertion- vs. Question-type) and extracts the target witness or characterization from the proof as an IG; in the answer-agnostic regime, it reformulates open interrogative queries into explicit declarative targets without an answer oracle:

- **Mathematical Domain Identification:** The agent categorizes the problem into its primary mathematical discipline (e.g., Number Theory, Real Analysis, Combinatorics, Abstract Algebra, Eu-

clidean Geometry). This classification primes downstream retrieval and informs Mathlib namespace scoping.
- **Answer-Aware Regime (With-Proof):** When an informal proof P_{NL} is supplied as an oracle, the Distiller analyzes the logical resolution of the problem. For Assertion-type problems (e.g., "Prove that..."), the claim is self-contained and the Inferred Goal (IG) remains empty. For Question-type problems (e.g., "Find all...", "Determine the value of..."), the Distiller extracts the definitive mathematical witness, numeric constant, or characterization established by P_{NL} as the IG.
- **Answer-Agnostic Regime (Without-Proof):** When no proof is provided, the Distiller is strictly forbidden from computing or hallucinating answers. Instead, it reformulates the open query into an unambiguous declarative proposition using existential ($\exists$), unique existential ($\exists!$), or extremal quantification, leaving the constructive witness search entirely to downstream provers.

2. The Preprocessor. The Preprocessor converts the distilled problem into a structured Declarative Normal Form representation (S_{DNF}) and decomposes it into symbolic components across four strict sequential phases:

- **Phase 0 (S_{DNF} Generation):** If the IG is empty, S_{DNF} matches S_{NL}. If the IG is non-empty, the agent constructs a declarative natural language assertion stating that the IG uniquely satisfies the problem's conditions.
- **Phase 1 (Assertion Extraction):** Isolates the primary mathematical claim as the final symbolic goal G.
- **Phase 2 (Global Infrastructure):** Identifies supporting mathematical objects, relations, or functions, specifying them as structured signatures:

```
def [Name] (inputs : Type) : [Output Type] := [Formula]
```

- **Phase 3 (Hypothesis Extraction):** Isolates all antecedent conditions, bounds, and premises into a numbered roster: $h_1, \ldots, h_n$, where each h_i is paired with a concise mathematical description.

3. The Formalizer. The Formalizer maps the preprocessed pseudocode blocks into idiomatic, fully typed Lean 4 signatures. Guided by an *Anti-Trivialization Guard*, it enforces the following protocol:

- **Definitions Block:** Each auxiliary construct extracted in Phase 2 is translated into a standalone, well-typed Lean 4 `def`. The agent searches *Mathlib* for standard constructions to ensure modularity and idiomatic typing.
- **Theorem Signature:** The agent constructs the primary theorem declaration by binding all hypotheses and asserting the target goal for $i \in \{1, \ldots, n\}$:

```
theorem [name] (h_i : [translated_i]) : [G] := by sorry
```

Where an informal condition lacks a direct *Mathlib* primitive, the Formalizer establishes a dedicated predicate definition prior to the theorem:

```
def [prop_name] ([args]) : Prop := [definition]
```

and instantiates it inside the theorem header via `(h : [prop_name] [args])`. The signature terminates immediately at `:= by sorry`.

4. The Formatter. The Formatter acts as the final syntactic integrator. Because language models frequently output surrounding conversational commentary, omit necessary imports, misplace namespaces, or generate unaligned `noncomputable section` tags, the Formatter cleans and normalizes the code. Its operational rules mandate: (i) inserting `import Mathlib` and required module headers, (ii) managing namespace opens, (iii) stripping all conversational natural language, and (iv) preserving every mathematical definition and theorem signature verbatim without modifying the underlying formal semantics.

Table 7: **Sage generation pipeline.** The sequential, role-isolated agents composing the generation chain. Each stage processes specialized mathematical components, ensuring clear translation contracts before generating the final compilable Lean 4 code.

Agent	Input	Output	Description & Core Role
Distiller	S_{NL}, P_{NL}?	Domain + IG context	Identifies the mathematical domain and either extracts the IG from the proof or reformulates the open query into a declarative statement.
Preprocessor	Distiller output	S_{DNF}, local defs, hyps, G	Standardizes the problem into a unified Declarative Normal Form Statement statement (S_{DNF}) and decomposes it into assumptions, local definitions, and goals.
Formalizer	S_{DNF} components	Lean `defs + theorem ... sorry`	Maps assumptions and infrastructure into idiomatic Lean 4 statements, relying on *Mathlib* concepts while avoiding trivialization.
Formatter	Draft code	Standalone `.lean` file	Organizes namespaces, imports, and scaffolding into a compilable file without altering the underlying mathematical meaning.

B.2 Correction Loop Protocol

After the generation chain (decomposed or monolithic) produces an initial candidate statement, every multi-round variant enters the same iterative refinement skeleton for up to T rounds. Each round evaluates the current candidate under the loop's exit gate; if the gate fails and the round budget remains, a Corrector receives a structured briefing, emits a revised standalone Lean 4 file, and the candidate is re-verified. All Correctors share a fixed output contract: emit a complete Lean 4 file beginning with `import Mathlib`; terminate the main claim at `:= by sorry` (or preserve an existing `proof_wanted` form); and forbid proof tactics, helper lemmas, `admit`, or solving the underlying mathematics. The two instantiations below differ only in the exit gate and in which signals populate the Corrector briefing.

Both the syntax-only and dual-signal loops share two mechanisms that stabilize iterative repair:

Syntactic Error De-cascading. In Lean 4, compiler diagnostics frequently suffer from extreme cascading: an early typo or minor type error in an initial definition can trigger dozens of downstream errors throughout the theorem header. Supplying an uncurated stream of 30+ compiler errors overwhelms language models, inducing destructive over-correction on otherwise sound code. To stabilize repair, the Lean Verifier extracts the first $k = 5$ compiler errors to retain context, but explicitly instructs the Corrector to prioritize resolving the very first error (e_1). This top-down stabilization prevents chaotic thrashing across iterations.

Oscillation Mitigation via Prior Failed Attempts. Iterative repair loops can suffer from cyclic oscillation, where the model alternates between two conflicting fixes across successive rounds. To prevent this, the Corrector briefing dynamically maintains a *Prior Failed Attempts* history log containing the statements attempted in previous rounds and their failure reasons. This history acts as a negative memory constraint, forcing the Corrector to explore alternative Mathlib abstractions rather than repeating previously rejected patterns.

B.3 Syntax-Only Repair Loop

The ablation variants Decomposed+Syntax and Monolithic+Syntax isolate the contribution of compiler feedback without semantic gating. After the same initial draft as their single-pass counterparts (decomposed or monolithic), they enter the shared repair skeleton above for up to T rounds. A dedicated syntax Corrector receives the informal statement, the current Lean draft, and the verifier report, then emits a revised signature file under the shared Corrector contract. The loop exits suc-

cessfully as soon as Compile holds, or when the round budget T is exhausted. Because termination depends solely on type-checking, a candidate may leave the loop as a compiling but mathematically unfaithful statement—the failure mode measured when comparing syntax-only rows to Sage in Section 6.

B.4 Dual-Signal Semantic Correction Loop

When the initial draft is executed in the Lean 4 environment, Sage evaluates it with both an objective compiler check and the in-loop semantic Rater. A candidate exits successfully only when the joint gate Compile $\wedge$ Sage-SM is satisfied; otherwise the Corrector is invoked for up to T rounds under the shared skeleton above. In standard compile-only repair, a large language model (LLM) often restores well-typedness by deleting or trivializing hypotheses, destroying the intended meaning. To prevent this, the Corrector receives a structured bipartite diagnosis at every round:

1. **Syntactic feedback:** The first k compiler error diagnostics (line, column, and message) from the Lean 4 verifier.
2. **Semantic feedback:** Failing dimensions, scores, rationales, and localized mismatch snippets from the in-loop **Rater**, together with a Rater-proposed repair draft.

The Corrector is instructed to prioritize *semantic match over compilation*: a syntactically flawed but semantically faithful draft is treated as a productive intermediate state, whereas a compiling but mathematically incorrect statement must not be preserved. The loop terminates successfully only when Compile $\wedge$ Sage-SM holds.

Pre-compilation of Semantic Drafts. During semantic evaluation, the Rater scores the candidate statement across the different gating dimensions and proposes an alternative draft designed to repair identified semantic mismatches. Instead of naively passing this draft to the Corrector, Sage first compiles the Rater's draft in Lean 4. When the Corrector is invoked, it receives a synchronized bipartite diagnosis:

1. The compilation status and exact compiler diagnostics of its own previous attempt.
2. The Rater's multidimensional semantic critique, the Rater's proposed draft, and the verified compilation status of that proposed draft.

This mechanism prevents the Corrector from blindly adopting drafts that fix semantics but break syntax, forcing it to synthesize a solution that simultaneously satisfies both Lean's type checker and the semantic intent of the original mathematical query.

Anti-Solver and Anti-Injection Guard. In zero-shot and answer-agnostic settings, a naive rater may occasionally suggest "fixing" an existential goal by calculating the numerical answer and setting the variable to a constant or placeholder (e.g., suggesting $= 42$ or an uninstantiated constant X). To protect benchmark integrity, the Corrector is governed by a strict *Anti-Solver Rule* and an *Anti-Injection Escape Clause*. The agent is explicitly forbidden from performing calculations, solving the mathematical problem, or injecting concrete witnesses to silence type errors. If a proposed rater correction introduces an injected answer or an undefined placeholder, the Corrector is instructed to reject that modification and maintain the open-ended existential formulation (e.g., $\exists! N, \dots$).

B.5 Compositional Generation Walkthrough

Figure 3 illustrates the complete execution trace of the Sage pipeline on problem Omni-MATH #043 across both operational regimes:

- **Answer-Aware Regime (Guided by P_{NL}):** The Distiller analyzes the informal proof $64 = 4^3 \Rightarrow 64^2 = 4^6 \Rightarrow n = 6$ and extracts the Inferred Goal $n = 6$. The Preprocessor generates S_{DNF}: *"If* $4^n = 64^2$*, then* $n = 6$*"*, binds hypothesis $h_1 \colon 4^n = 64^2$, and sets goal $G \colon n = 6$. The Formalizer produces an explicit verification theorem asserting $n = 6$.
- **Answer-Agnostic Regime (Zero-Shot without Proof):** Deprived of an answer oracle, the Distiller refuses to compute the exponent. It reformulates the open query into an objective unique existential claim ($\exists!\, n$). The Preprocessor binds this into goal $G \colon \exists!\, n,\ 4^n = 64^2$, and the Formalizer generates an existential theorem in Lean 4, delegating the computation of n to downstream provers.

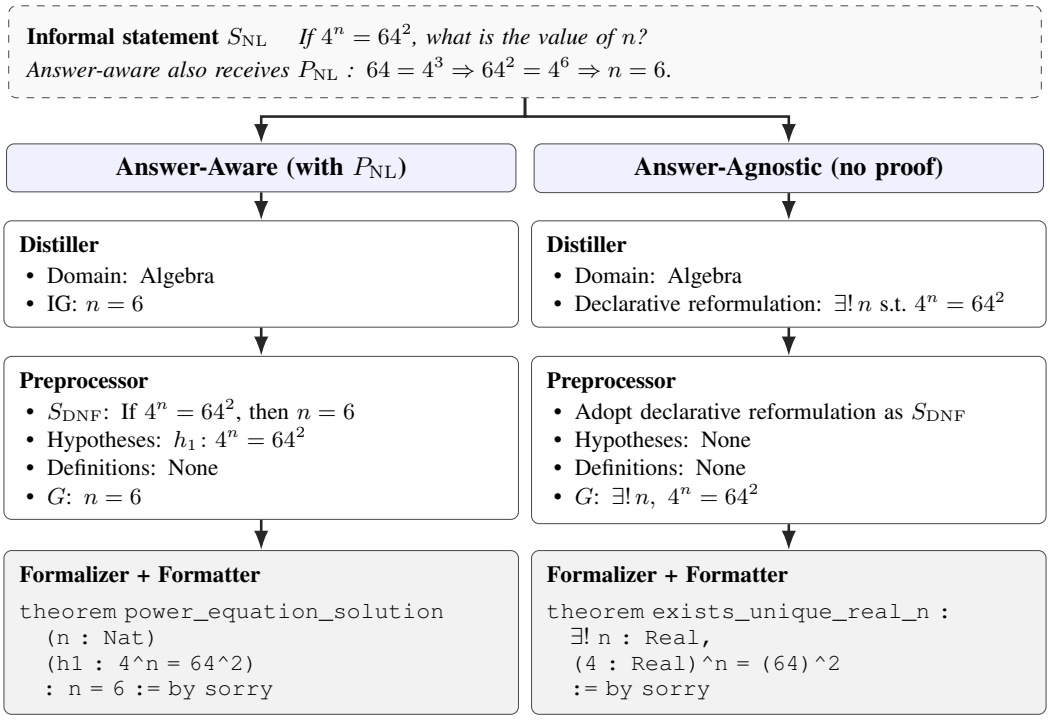


Figure 3: **Compositional generation walkthrough** on OMNI-MATH #043. The same informal question is processed under both regimes. With P_{NL}, the Distiller extracts an IG ($n = 6$) and the Preprocessor builds a Declarative Normal Form Statement claim with hypothesis h_1 and goal G: $n = 6$. Without a proof, the Distiller commits to a declarative reformulation ($\exists!\, n$), which flows through to an existential Lean 4 goal.

C EXPERIMENTAL SETUP, DATASETS, AND REPRODUCIBILITY

In this section, we provide comprehensive details on the benchmark datasets, the language model serving infrastructure, the evaluated model backbones, our compute environment constraints, and the full configuration matrix across all experimental ablations.

C.1 DATASET DETAILS AND CURATION

To test autoformalization across diverse problem types and formalization regimes, our study evaluates on three benchmarks: OMNI-MATH, IMO-FORMALIZED, and IMO-UNFORMALIZED. Both IMO suites are derived from a comprehensive snapshot of the community archive taken in July 2026.

OMNI-MATH ($N = 300$). OMNI-MATH (Gao et al., 2025) is an olympiad-level competition benchmark spanning contests such as AMC 10/12, AIME, USAMO, and Putnam. We evaluate on the exact 300-problem subset benchmarked by Goedel-Prover-V2 (Lin et al., 2025) to ensure direct comparability.

- **Curation Protocol:** Extracted directly from the standardized 300-problem evaluation split of Lin et al. (2025).
- **Problem Texture:** *Question-type* 98.7 %; *Assertion-type* 1.3 %. Question-type statements require calculating or identifying unknown extremal values, witnesses, or configurations.
- **Domain Distribution:** From the dataset metadata primary subject tags: Algebra (62.7 %), Number Theory (29.0 %), Combinatorics (6.3 %), and Precalculus (2.0 %). This 300-subset contains no Geometry-tagged problems.

IMO-FORMALIZED ($N = 223$). This benchmark comprises all historical International Mathematical Olympiad (IMO) problems from 1959 to 2024 that possess official, human-verified formalizations in Lean 4.

- **Curation Protocol:** Extracted from a snapshot of the `compfiles` repository (Renshaw, 2024) taken in July 2026. It represents the fraction of historical IMO problems ($223/398$) already formalized by the Lean community, with informal statements extracted from module docstrings and formal declarations parsed into standalone targets.
- **Problem Texture:** Nearly balanced: *Question-type* $51.6\,\%$; *Assertion-type* $48.4\,\%$.
- **Domain Distribution:** Distiller-assigned primary domains, mapped to the four canonical IMO disciplines (analysis / functional-equation labels folded into Algebra): Number Theory ($33.2\,\%$), Algebra ($33.6\,\%$), Combinatorics ($26.5\,\%$), and Geometry ($6.7\,\%$).

IMO-UNFORMALIZED ($N = 175$). This benchmark comprises the complementary frontier of historical IMO problems that have never been formalized in Lean 4.

- **Curation Protocol:** Derived from the same July 2026 `compfiles` snapshot by strictly selecting the complement fraction ($175/398$) that lacked official human formalizations. Problem texts and informal solution notes were curated from the AI-MO olympiads compendium and official IMO archives.
- **Problem Texture:** *Assertion-type* $64.0\,\%$; *Question-type* $36.0\,\%$.
- **Domain Distribution:** Distiller-assigned primary domains under the same four-discipline mapping: Geometry ($69.1\,\%$), Combinatorics ($24.6\,\%$), Algebra ($3.4\,\%$), and Number Theory ($2.9\,\%$). The Geometry skew reflects which historical IMO problems remain unformalized in Lean 4.

C.2 LLM SERVING, MODELS, AND INFRASTRUCTURE

All language model inference is executed using the high-performance `vLLM` engine. Each model is served on dedicated GPU instances with a tensor parallelism degree of 1, utilizing FP8 quantized weights and dynamic KV caching to maximize token throughput during iterative multi-agent correction loops.

Evaluated Models and Pipeline Roles. Our experimental framework strictly decouples generation, in-loop rating, and post-hoc evaluation to eliminate circular self-evaluation bias:

1. **Generation & In-Loop Rating (Qwen3.6-27B):** We deploy `Qwen/Qwen3.6-27B-FP8` as the core workhorse model. It powers all four stages of the generation chain (Distiller, Preprocessor, Formalizer, Formatter) as well as the in-loop semantic Rater and Statement Corrector.
2. **Independent Post-Hoc Evaluator (Gemma4-31B):** To ensure an unbiased, external evaluation of semantic fidelity, all post-hoc consensus judging (Goedel-SM) and blind pairwise win-rate adjudications (PWR) are performed using `RedHatAI/gemma-4-31B-it-FP8-Dynamic`.
3. **Domain-Specialized Baseline (Goedel-Formalizer-V2-32B):** We evaluate against `Goedel-LM/Goedel-Formalizer-V2-32B` (Lin et al., 2025), an open-weights model explicitly fine-tuned for Lean 4 statement autoformalization.

Table 8 details the decoding hyperparameters, sampling bounds, and generation settings across each model role.

Table 8: **LLM decoding and sampling hyperparameters.**

Model	Role	Temp.	Max tokens	Retries
Qwen3.6-27B	Pipeline generation + in-loop rater	0.6	16,384	2
Goedel-Formalizer-V2-32B	Fine-tuned baseline (Goedel)	0.7	16,384	1
Gemma4-31B	Post-hoc evaluation (Goedel-SM / PWR)	0.6	16,384	2

Note on Model Backbones and Qwen 3.8. All primary experiments, benchmark sweeps, and ablation ladders were established using Qwen3.6-27B, which served as our stable workhorse throughout development. Qwen3.8-27B was released late in our experimental cycle. While its enhanced capabilities yield substantial performance gains in our scaling ablation (Table 3), its extended rea-

soning traces noticeably increased per-step latency. Given the multi-agent nature and multi-round repair loops of our pipeline, re-executing the full experimental suite across all benchmarks was impractical within our project timeline. We therefore maintain Qwen3.6-27B as the consistent backbone across all primary comparative tables.

Compute Environment and Model Access. All experiments and evaluations were conducted on an on-premise computing cluster equipped with dedicated GPU nodes. To prioritize scientific reproducibility, eliminate dependency on closed commercial APIs with volatile endpoints, and ensure strict data governance, our framework relies exclusively on publicly available, open-weights models deployed locally. This setup ensures that all benchmarks, latency profiles, and generated formal artifacts are fully deterministic, auditable, and reproducible by the broader research community without reliance on proprietary black-box services.

C.3 Method Variants and Ablation Configurations

To systematically isolate the individual contributions of pipeline decomposition, syntactic error feedback, and semantic rating feedback, we benchmark six primary configurations across two operational regimes, detailed in Table 9. Evaluating these variants on an identical open-weights backbone (Qwen3.6-27B) allows us to causally decouple the effects of structural decomposition and feedback signals without confounding differences in base model capability or pretraining data. For all iterative methods, the maximum repair budget is set to $T = 5$ rounds on OMNI-MATH and $T = 10$ rounds on IMO-UNFORMALIZED to ensure strict test-time compute parity.

Methodological Framing and Compute Parity Rationale. Autoformalization literature features various systems with disparate base models and training setups (e.g., Process-Driven Autoformalization (Lu et al., 2024) and ReForm (Chen et al., 2026)). Direct cross-system benchmarking against published checkpoints inevitably confounds algorithmic framework merits with base model capacity and pretraining exposure. To provide rigorous scientific attribution, our evaluation strategy is two-fold:

1. **Controlled Mechanism Isolation:** We benchmark the foundational mechanisms of compiler-only refinement (Decomposed+Syntax, isolating the compiler-feedback loop of Lu et al., 2024) and all-in-one reflective self-critique (Monolithic+Dual, capturing the reflective paradigm of Chen et al., 2026) on an *identical* open-weights backbone (Qwen3.6-27B) under *identical* iteration budgets ($T = 5$ and $T = 10$), isolating the precise value added by four-stage decomposition and dual-signal verification under strict test-time compute parity.
2. **Fine-Tuning Paradigm Comparison:** We include Goedel-Formalizer-V2 as an external anchor representing the specialized supervised fine-tuning paradigm. While this baseline uses single-pass generation ($T = 1$), it embodies a specialized 32B model fine-tuned via expert iteration on 50,000 Claude-distilled formalization traces; comparing against it highlights the trade-off between static parameter fine-tuning and zero-training test-time neurosymbolic scaffolding.

Overview of Evaluated Configurations. Our benchmark suite spans the following method variants:

- **Single-Pass Forward Baselines ($T = 1$):**
  - Monolithic: Translates the informal problem into a formal Lean 4 statement in a single forward pass without intermediate reasoning scaffolding.
  - Decomposed: Executes our four-stage feedforward generation chain (Distiller → Preprocessor → Formalizer → Formatter) without entering a repair loop.
- **Compiler-Only Feedback Loops ($T = 5$ / $T = 10$):**
  - Monolithic+Syntax: An iterative loop wrapping the monolithic generator, utilizing raw Lean 4 compiler diagnostics for syntactic self-repair.
  - Decomposed+Syntax: An iterative loop wrapping our four-stage pipeline, providing compiler error diagnostics to the Corrector agent while omitting semantic critique.
- **Dual-Signal Semantic Refinement Loops ($T = 5$ / $T = 10$):**

- Monolithic+Dual: An all-in-one reflective loop where the monolithic candidate is evaluated by both the compiler and the in-loop semantic Rater (Sage-SM), providing dual-signal feedback to revise the statement.
- **Sage** (Decomposed+Dual): Our complete framework, combining the four-stage generation pipeline with dual-signal syntactic and semantic feedback.

- **External Supervised Baseline** ($T = 1$)**:**
  - Goedel (Goedel-Formalizer-V2-32B): Evaluated in single-pass forward mode using its official thinking-prompt format to calibrate performance against specialized supervised learning.

Table 9: **Overview of evaluated methods and ablation variants.** Iterative variants are evaluated under matching iteration budgets ($T = 5$ on OMNI-MATH, $T = 10$ on IMO-UNFORMALIZED) on a shared base model to ensure strict test-time compute parity.

Method	Base Model	Architecture	Feedback Type	Rounds (T)
Monolithic Configurations				
Monolithic	Qwen3.6-27B	Monolithic	None	1
Monolithic+Syntax	Qwen3.6-27B	Monolithic	Compile	5–10
Monolithic+Dual	Qwen3.6-27B	Monolithic	Compile + Sage-SM	5–10
Decomposed Configurations (Ours)				
Decomposed	Qwen3.6-27B	Decomposed	None	1
Decomposed+Syntax	Qwen3.6-27B	Decomposed	Compile	5–10
Sage (Decomposed+Dual)	Qwen3.6-27B	Decomposed	Compile + Sage-SM	5–10
External Domain Baseline				
Goedel	Goedel-Formalizer-V2-32B	Monolithic (Thinking)	None	1

C.4 EVALUATION METRICS AND INTEGRITY AUDITS

Standard autoformalization evaluations typically report raw compilation rates (Compile), implicitly assuming that syntactically valid code implies mathematical correctness. In reality, models frequently generate type-correct formalizations that are mathematically vacuous, structurally altered, or unfaithful to the informal source. To establish an honest evaluation standard, our protocol decouples syntactic compilation from post-hoc semantic validation, and introduces strict automated audits for benchmark integrity.

Compilation Environment (Compile). All Lean 4 compilation checks are executed in isolated worker processes managed by our verification engine. Every candidate file imports Mathlib and is verified against a pinned Lean 4 toolchain. A formalization passes Compile if the environment returns an exit code of 0 without diagnostic type errors. Proof bodies are omitted via `:= by sorry`. Statements that trigger compiler timeouts (>30 s) or memory exhaustion are treated as compilation failures.

We pin Lean 4 to `v4.29.0-rc2` with a fixed Mathlib revision via the project lockfile; every candidate imports Mathlib and is checked in this environment.

The Single-Sorry Invariant and Cheat Detection. As introduced in Section 4.2, a valid statement must respect the **single-sorry contract**. Permitting multiple placeholders exposes benchmarks to two severe vulnerabilities:

1. *Definition Stubbing:* Models can declare unresolved constant definitions stubbed with a placeholder (e.g., `def k : Nat := sorry`), and subsequently state a trivial theorem over `k`. While syntactically valid, any proof over an uninstantiated definition is vacuous.
2. *Hypothesis and Lemma Evasion:* Models frequently decompose hard theorems into intermediate helper lemmas closed by `sorry` (e.g., `lemma helper : ... := by sorry`), effectively assuming the problem's deepest mathematical barrier rather than stating a self-contained target.

Our audit strips comments and string literals from the Lean source, then counts unresolved placeholder tokens (`sorry`, `admit`, `proof_wanted`, and related forms). A prover-ready candidate must contain exactly one terminating placeholder on the primary target declaration (`theorem`, or

`def` for Type locks). Any compiling candidate with more than one placeholder, or with placeholders inside intermediate definitions or helper lemmas that are *not* the primary target, is disqualified as *Multiple Sorries*.

Answer Leakage Audit Protocol. When evaluating open-ended *Question-type* problems under the answer-agnostic (without-proof) regime, models are tasked with formalizing the question itself, not computing the solution. However, monolithic models frequently shortcut the translation by searching for the answer during generation and hardcoding the numeric result into the theorem signature (e.g., translating "Find the minimum value of $f(x)$" into `theorem T : ∀ x, f x ≥ 42`). We audit this with an independent LLM judge (Gemma4-31B). Given the informal statement, the informal proof when available, and the generated Lean code, the auditor runs a fixed protocol:

1. **Texture classification.** Decide whether the informal problem is Assertion-type (self-contained claim) or Question-type (asks to compute, determine, or construct an object). Assertion-type problems are excluded from leakage penalties, even if their mathematics involves existentials.
2. **Witness presence.** Check whether the formal statement embeds a concrete witness term or numeric value rather than an open encoding (e.g., ∃ / ∃!).
3. **Match and type check.** If a witness is embedded, check whether it matches the ground-truth solution and whether its Lean type is compatible with what the informal problem asks for.

In the answer-agnostic regime, any compiling Question-type formalization that embeds a concrete witness is flagged as **Answer Leakage**. Ground-truth matching in step 3 is diagnostic only: the binary leakage flag does not require the witness to equal the dataset answer.

Post-Hoc Consensus Semantic Match (Goedel-SM). For comparability with prior work, we report post-hoc semantic match under the Goedel-Prover-V2 protocol (Lin et al., 2025, Appendix D): an independent Gemma4-31B judge, queried repeatedly, with success only under unanimous *Appropriate* consensus. Full consensus settings, prompt-bias analysis, and reject-reason taxonomy are in Appendix C.5. Because that judge prefers explicit answer injection, leakage audits are essential when interpreting these scores.

Multi-Dimensional In-Loop Semantic Gate (Sage-SM). Unlike a single post-hoc *Appropriate/Not Appropriate* prompt, our in-loop gate (Sage-SM) evaluates formalizations with the multi-dimensional Rater from Section 4.2. A candidate passes Sage-SM iff all seven gated dimensions clear the cutoffs in Table 10. Auxiliary diagnostic scores (e.g., compilability and difficulty) inform repair suggestions but do not enter the binary gate. The gate steers the Statement Corrector during search and can also filter high-trust training statements post hoc.

Table 10: **In-loop Sage-SM gate.** Scales are those used by the Rater prompt; cutoffs are conjunctive (all must hold).

Dimension	Scale	Cutoff
Semantic Match	0–4	≥ 4
Missing	0–3	≤ 1
Extra	0–4	≤ 1
Wrong	0–2	$= 0$
Exactness	0–3	≥ 1
Naturality	0–4	≥ 3
Descriptive Precision	0–4	≥ 3

Blind Pairwise Win Rate (PWR). To evaluate semantic fidelity without rigid prompt heuristics, we perform blind head-to-head pairwise comparisons between model outputs $\mathcal{A}$ and $\mathcal{B}$. An independent judge compares both formalizations against the informal source on a 5-point Likert scale: *Much Worse* (-2), *Worse* (-1), *Tie* (0), *Better* ($+1$), and *Much Better* ($+2$). To eliminate positional bias, every pair is evaluated twice with swapped presentation orders: $(\mathcal{A}, \mathcal{B})$ and $(\mathcal{B}, \mathcal{A})$. Inconsistent orders are reconciled via tie assignment. We report preference shares over all comparisons (strict wins, ties, and losses); conditional on non-ties, the win rate is the share of decisive comparisons where a method is strictly preferred.

Joint Evaluation Criteria and Sample Aggregation. Our primary evaluation metric is the joint criterion Compile $\wedge$ Goedel-SM, denoting candidate statements that are simultaneously well-typed in Lean 4 and unanimously approved by the post-hoc semantic judge. For multi-sample search (pass@N), a problem is counted as solved under a given metric if *at least one* of the N independent candidate generations satisfies the required criteria.

C.5 Goedel Evaluation Protocol and Prompt Bias Analysis

Consensus Protocol. To measure post-hoc semantic fidelity (Goedel-SM), we adopt the evaluation prompt from Goedel-Prover-V2 (Lin et al., 2025), modifying it only to enforce structured JSON parsing. We deploy Gemma4-31B as the independent evaluator. For each candidate statement, we query the judge four independent times. Each query outputs a categorical verdict of either *Appropriate* or *Not Appropriate*, together with a short free-form rationale. Following Lin et al. (2025), a formalization is counted as successful under Goedel-SM if and only if it achieves a unanimous $4/4$ *Appropriate* consensus; we leave this decision rule unchanged.

Prompt bias (from the few-shot examples). The demonstrations embedded in the Goedel-SM prompt show a structural preference for explicit answer injection over open encodings.

- For a problem asking for the intercept of a quadratic polynomial, the prompt marks a formalization *Appropriate* because it injects the resolved numeric value (`f 3 = 0`).
- For a problem asking for the equation of a tangent line, the prompt marks an answer-agnostic formulation *Inappropriate*, on the grounds that "*The translation shifts the problem from deriving the tangent line to verifying a given line equation.*"

Thus the judge is instructed, by example, to reward closed witnesses and to penalize faithful open formulations that defer the answer. We do not alter these demonstrations: changing them would change the Goedel protocol itself.

Reject-reason audit. To measure how this bias manifests in practice—without perturbing the judge—we attribute Inappropriate votes *post hoc* by inspecting the returned rationales on Omni-MATH (answer-agnostic, pass@1). We first ask an LLM to propose rejection categories from those rationales, consolidate them into a fixed taxonomy of twenty categories, and then ask Qwen3.6-27B to assign every applicable category to each rejection (multi-hot). Table 11 reports the three most frequent categories.

Table 11: **Goedel-SM reject reasons on Omni-MATH (Answer-Agnostic) at pass@1.** Three most common categories among *Inappropriate* samples (multi-hot; lower is better for defect categories). Bold = lowest rate.

Method	$N_{\text{rejection}}$	Wrong Injection (%)	Mathematically False (%)	Existential (%)
Monolithic	147	69.4	84.4	**8.2**
Monolithic+Dual	129	68.2	80.6	15.5
Decomposed	164	8.5	22.0	72.0
Decomposed+Dual (Sage)	138	**7.2**	**18.1**	76.8
Goedel (Finetuned)	204	56.9	74.5	19.6

The three reported categories are defined as follows. **Wrong Injection:** S_{F} hard-codes a concrete answer absent from the open S_{NL}. **Mathematically False:** the judge treats S_{F} as asserting a false claim. **Existential:** a faithful open encoding is rejected primarily for omitting an explicit answer.

The empirical split matches the prompt-level bias above. Monolithic generators and fine-tuned Goedel are dominated by Mathematically False and Wrong Injection (Goedel: $74.5\,\%$ and $56.9\,\%$ of rejects): they guess a closed witness and often encode an incorrect constant. Decomposed generators, including Sage, are dominated by Existential ($76.8\,\%$): they keep open encodings ($\exists x,\, P(x)$) that the judge systematically prefers less than explicit answers. This asymmetry is intentional under our loop contract. Our in-loop rater is not trained or prompted to verify the mathematical correctness

of conjectured constants; doing so would require solving the informal problem and would conflate translation with deduction. We therefore treat both injection and existential texture as admissible, and we do not route pure mathematical-falsity rejections—especially wrong injected witnesses—through the correction loop. Accordingly, the residual Goedel-SM gap should not be read as a failure of decomposition: for Goedel and monolithic methods it largely reflects unsuccessful guessing, whereas for Sage it largely reflects the judge's documented preference against open existential formalizations.

D Comprehensive Performance Results and Trajectory Analysis

We expand the generator-backbone scaling results of Table 3 and the human validation of Sage-SM summarized in Section 6, reporting the full metric suite, witness-injection rates, and the stratified audit design.

D.1 Generator Backbone Ablation: Scaling across Model Generations

To verify that the performance of Sage is not contingent upon the specific inductive biases of Qwen3.6-27B, we evaluated our complete decomposed semantic pipeline (Sage, $T = 5$, pass@1) across three successive model generations: Qwen3-32B, Qwen3.6-27B, and Qwen3.8-27B. All runs were conducted on Omni-MATH in the Answer-Aware setting and audited using identical protocols: Lean 4 type-checking (Compile), unanimous 4/4 independent Gemma4-31B consensus (Goedel-SM), our multi-dimensional semantic gate (Sage-SM), and our witness injection audit.

Table 12: **Comprehensive generator backbone ablation on Omni-MATH (Answer-Aware, pass@1,** $N = 300$**).** *In-loop optimization target. All methods use Sage with identical prompting, repair budgets ($T = 5$), and independent judge protocols. Joint solved counts reported out of 300 problems.

Model Backbone	Compile	Goedel-SM	Compile ∧ Goedel-SM	Joint Count	Compile ∧ Sage-SM *	Witness Injection (%)
Qwen3-32B (32B)	57.3	46.3	30.3	91 / 300	52.3	22.0
Qwen3.6-27B (27B)	96.0	80.3	77.3	232 / 300	93.7	**90.2**
Qwen3.8-27B (27B)	**98.0**	**86.3**	**85.7**	**257 / 300**	**95.7**	89.5

Analysis of Scaling Trajectories. The empirical progression highlights two clear behaviors:

1. **Monotonic Verification Synergy:** As the generator's underlying formal competence improves, the efficacy of the repair loop compounds dramatically. While Qwen3-32B struggles to interpret Lean 4 compiler diagnostics (yielding only $57.3\,\%$ compilation and $30.3\,\%$ joint fidelity), Qwen3.8-27B reaches $98.0\,\%$ compilation and $85.7\,\%$ joint fidelity.
2. **Witness Extraction Ceiling:** In Answer-Aware formalization, injecting the target constant from the informal proof is expected. Qwen3-32B achieves only $22.0\,\%$ witness injection, indicating frequent comprehension failures on informal mathematical texts. In contrast, both Qwen3.6-27B ($90.2\,\%$) and Qwen3.8-27B ($89.5\,\%$) reach the witness extraction ceiling. Consequently, the performance improvement between 3.6 and 3.8 is purely driven by superior formalization fidelity, syntax resolution, and premise alignment.

E Human Expert Validation of the Semantic Gate

To evaluate the reliability of our automated quality gate, three Lean-proficient annotators independently scored an adversarial audit corpus of $N = 45$ zero-shot Omni-MATH formalizations under the same five-core rubric and binary gate used by Sage-SM. We compare the automated gate to **majority vote**.

Stratified Audit Design. Rather than drawing a naive uniform random sample (which would be dominated by straightforward instances), the benchmark was deliberately stratified across four distinct categories to stress-test the gate against edge cases:

1. **Silent Semantic Failures** ($N = 25$)**:** Statements that compile cleanly in Lean 4 but contain subtle mathematical flaws (e.g., incorrect summation indices, artificial domain restrictions, or hardcoded constants).
2. **Compile-Failed Faithful Drafts** ($N = 10$)**:** Mathematically sound translations that failed type-checking solely due to minor syntactic or import errors.
3. **Verified Clean Translations** ($N = 5$)**:** Idiomatic, fully compilable, and semantically equivalent formalizations.
4. **Catastrophic Failures** ($N = 5$)**:** Formalizations that are both uncompilable and mathematically false.

Results. Across the stratified audit, the automated gate concurs with majority human judgment on $71.1\,\%$ of formalizations. Beyond the binary decision, mean human Semantic Match aligns with the rater at Pearson $r = 0.64$ and Spearman $\rho = 0.63$, providing complementary evidence that the evaluator ranks drafts consistently with human semantic preference.

F External Evaluation of the Semantic Gate

Section E validates the gate on an adversarial expert audit. Complementing that study, we evaluate the same binary gate on two public autoformalization-alignment benchmarks with gold faithfulness labels.

Benchmarks. ConsistencyCheck (Chen et al., 2026) comprises N=859 informal–formal pairs drawn from miniF2F and ProofNet sources (570 labeled faithful, 289 unfaithful). ProofNetVerif (Poiroux et al., 2025) comprises N=3752 pairs in which a candidate Lean 4 formalization is scored against an informal statement (1142 labeled correct, 2610 incorrect). Each item provides a natural-language statement, a Lean candidate, and a binary gold label indicating semantic faithfulness.

Protocol. For every informal–formal pair we run the in-loop semantic rater zero-shot and apply the identical binary gate used at test time, which returns a single Pass or Fail decision. Pass is treated as the positive prediction and compared to the benchmark gold label. No decision rule is retuned on these corpora.

Metrics. Table 13 reports:

- **Accuracy:** fraction of pairs on which the gate matches the gold label;
- **Precision / Recall / F1:** for the Pass class relative to gold-positive items;
- **Matthews correlation:** chance-adjusted association between predictions and gold labels, ranging in $[-1, 1]$ (higher is better; 0 is chance-level);
- **Majority baseline:** accuracy obtained by always predicting the more frequent gold class.

Results. On both benchmarks the gate exceeds the majority baseline by about 19 percentage points, reaching $85.7\,\%$ accuracy on ConsistencyCheck and $88.8\,\%$ on ProofNetVerif, with Matthews correlations of 0.67 and 0.74 respectively (Table 13).

Table 13: **External evaluation of the Sage-SM five-core gate** against gold faithfulness labels on ConsistencyCheck and ProofNetVerif. Predictor: published Pass/Fail gate; Pass is the positive class.

Benchmark	N	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)	Matthews correlation	Majority baseline (%)
ConsistencyCheck (Chen et al., 2026)	859	**85.7**	88.0	90.9	89.4	**0.67**	66.4
ProofNetVerif (Poiroux et al., 2025)	3752	**88.8**	82.1	80.9	81.5	**0.74**	69.6

G Blind Pairwise Preferences on IMO-Unformalized

Table 14 reports the full order-swapped Answer-Agnostic pass@1 preference breakdown summarized in Section 6: an independent automated judge compares Sage against Goedel on the unformalized IMO frontier.

Table 14: **Blind pairwise preferences on IMO-UNFORMALIZED (Answer-Agnostic, pass@1).** Percentages may not sum to 100 % due to rounding.

	Preference for **Sage** over **Goedel** (%)				
Evaluation Type	**Much Better**	**Better**	**Tie**	**Worse**	**Much Worse**
Semantic (PWR)	78.3	1.1	17.7	1.1	1.7