NF-Bridge: Faithfully Bridging Natural and Formal Languages for Mathematical Reasoning

Anonymous authors
Paper under double-blind review

Abstract

Natural-language and formal-language reasoning and proving constitute two important paths for advancing mathematical theory: the former supports flexible mathematical expression, while the latter provides rigorous machine verification. However, faithfully converting between these two proof forms and effectively combining their respective strengths remains a central challenge. To address this problem, we propose NF-Bridge, a faithfulness-centered framework that connects natural-language mathematical reasoning with formal verification. NF-Bridge first converts a candidate natural-language solution into a semantic proposition DAG, which explicitly represents answer-relevant mathematical assertions and their dependency structure, while abstracting away rhetorical and repetitive content without discarding key mathematical commitments. The entire DAG is then formalized into Lean representations. Compilation-based verification and semantic-fidelity evaluation are subsequently used to assess, respectively, the syntactic and type-level validity of the formalization and its consistency with the original natural-language argument. We further demonstrate that NF-Bridge not only supports faithful autoformalization of natural-language proofs, but also enables the construction of formal-verification-grounded reward signals for improving mathematical reasoning. Across AIME24, AIME25, MATH-500, and BRUMO25, the verifier-based reward derived from NF-Bridge outperforms multiple learned reward-model baselines. As additional evaluations of its underlying proof-autoformalization capability, NF-Bridge also achieves strong results on ProofFlowBench and RobustPABench.

1 Introduction

Providing checkable evidence for the outputs of frontier AI systems Kimi Team et al. (2026); DeepSeek-AI et al. (2026) is an important step toward reliable reasoning on complex tasks. While self-consistency Wang et al. (2023) and learned verifiers Cobbe et al. (2021); Lightman et al. (2024) have improved reasoning performance, neuro-symbolic methods Olausson et al. (2023); Pan et al. (2023) offer a principled path to verifiability through explicit rules and formal logic. By combining the expressive flexibility of large language models (LLMs) with the rigorous verification capabilities of symbolic systems, such as Lean 4 de Moura & Ullrich (2021), these methods Liu et al. (2025) can improve output quality while providing explicit verification feedback to guide further optimization.

However, the reliability of formal verification does not automatically extend to natural-language reasoning. In mathematics, autoformalization may alter variable bindings, omit necessary assumptions, modify mathematical claims, or introduce unintended proof dependencies. Such deviations can cause verification failures or produce verifiable proofs that depart from the original argument. Lu et al. (2025) and Cabral et al. (2026) address this challenge through semantic alignment evaluation and proof dependency preservation, respectively. Nevertheless, maintaining faithfulness throughout autoformalization remains a key challenge. Moreover, even faithful formalizations are subject to the limitations of proof

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.

search and computational budgets. Verification outcomes therefore do not automatically provide effective signals for test-time scaling (TTS), resulting in sparse feedback for candidate selection. In this work, we focus on two central questions:

- ***How can natural-language reasoning be faithfully formalized?***
- ***How can formal verification be translated into effective signals for TTS?***

To address these challenges, we propose NF-Bridge, a three-stage framework that connects natural-language mathematical reasoning with formal verification: First, we formalize the mathematical objects (e.g., variables, sets, and functions) and the constraints specified by the original problem and the modeling choices in the candidate solution. These definitions remain fixed throughout subsequent stages to maintain a consistent interpretation of the problem. Second, we organize the solution into a directed acyclic graph (DAG) of mathematical claims, preserving answer-relevant reasoning and its logical dependencies while consolidating redundant descriptions. We then translate the entire graph into Lean and check both its semantic fidelity to the original solution and its syntactic and type correctness. When either check fails, we jointly refine the graph and its Lean representation. Finally, we perform dependency-aware verification in Lean, searching for proofs and formal refutations of individual reasoning steps. We distinguish verified outcomes from unresolved cases, since failure to find a proof does not establish that a step is incorrect. We then aggregate these verification outcomes into reward signals for test-time candidate selection.

We evaluate NF-Bridge on two complementary tasks: proof autoformalization and mathamatical reasoning. NF-Bridge achieves semantic faithfulness of 76.63% on ProofFlowBench and 87.70% on miniF2F (Zheng et al., 2021) with the 27B and 9B prover configurations, respectively, exceeding the strongest evaluated baselines by 25.54 and 23.77 percentage points. For TTS, NF-Bridge with the 9B prover and Reject-3 ranking matches the strongest evaluated learned reward model on AIME24, AIME25, and BRUMO25, and MATH-500. Together, these results support proof autoformalization both as an end task and as a mechanism for constructing useful verification feedback for natural-language mathematical reasoning. In summary, our key contributions are as follows:

1. We introduce NF-Bridge to improve the semantic faithfulness of translating natural-language mathematical reasoning into Lean.
2. We convert Lean verification feedback into rewards for ranking and selecting natural-language solutions at test time.

2 Preliminary

Reliable verification feedback is important for evaluating and improving automated theorem proving and mathematical reasoning. Natural language allows humans and LLMs to express mathematical ideas and arguments, while formal languages make mathematical objects, assumptions, and claims explicit, enabling rigorous proof checking in systems such as Lean 4. However, translating natural-language reasoning into a formal representation can change object meanings, omit necessary conditions, or distort logical dependencies. Such deviations can yield a Lean-checked proof of an unintended statement, ***leaving a gap between formal correctness and faithfulness to the original statement***.

Prior work has advanced proof autoformalization through structured decomposition. ProofFlow represents proof steps and their dependencies as a directed acyclic graph (DAG), formalizes the resulting proof obligations as Lean lemmas, and checks the generated proofs with Lean (Cabral et al., 2026). ToMap adopts a Decomposer–Formalizer–Prover pipeline and refines self-contained proof units with explicit dependencies to improve downstream formalization and proving (Liu et al., 2026). Nevertheless, structured decomposition alone does not ensure that mathematical definitions, constraints, and object references retain their intended meanings. ProofFlow reports semantic discrepancies introduced by its Formalizer, while ToMap illustrates omitted conditions and underspecified variable domains in its initial decompositions. To characterize these risks, we organize semantic and structural deviations into five non-exclusive categories:

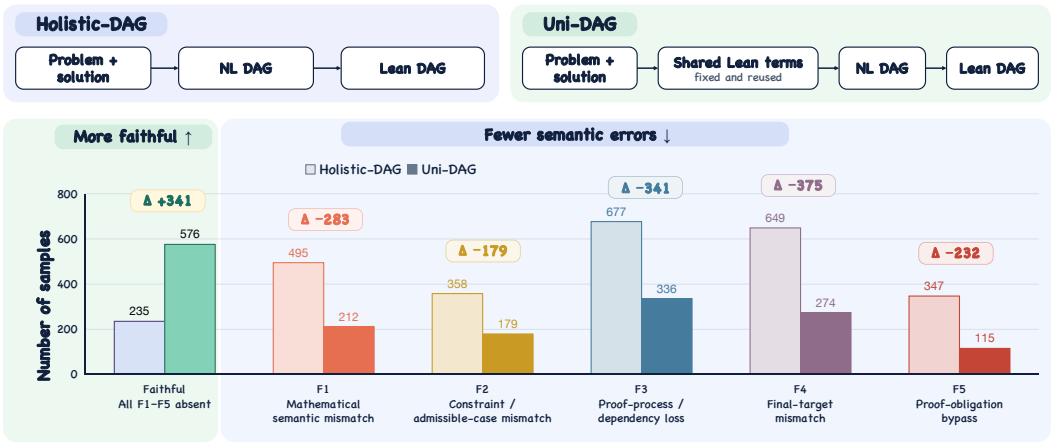

Figure 1: Comparison on 1,000 paired problem-solution samples before proof search. Bars show independently judged sample counts, with Δ = Uni-DAG − Holistic-DAG. F1–F5 are non-exclusive errors; *Faithful* requires no error labels and an overall faithful verdict.

1. **F1: Mathematical semantic mismatch.** A mathematical object, operation, type, or intermediate claim changes meaning during formalization, such as confusing a sequence index with the element it indexes.
2. **F2: Constraint or admissible-case mismatch.** A variable domain, quantifier scope, or admissibility condition is changed without justification, such as dropping a positivity condition or replacing an integer domain with the reals.
3. **F3: Proof-process or dependency loss.** A substantive derivation or direct dependency in the original solution is omitted or distorted, such as representing only a check of a proposed answer while omitting its derivation.
4. **F4: Final-target mismatch.** The final Lean proposition addresses the wrong object or specifies a different or incomplete task, such as expressing only feasibility when minimality or uniqueness is required.
5. **F5: Proof-obligation bypass.** The formalization removes the intended proof obligation by hardcoding the answer, assuming the conclusion, or introducing unsupported assumptions that make the target vacuously true.

We conduct a paired preliminary study to examine these errors and assess whether shared, fixed mathematical definitions can reduce them. We generate a candidate-solution pool using Qwen3-8B (Yang et al., 2025) on AIME24, AIME25, BRUMO25 (Balunović et al., 2025), and MATH-500 (Lightman et al., 2024).[1] We compare both configurations on 1,000 paired problem–solution samples using matched Qwen3.8-27B (Qwen Team, 2026b) generation and compilation settings; complete sampling and protocol details are provided in Appendix D. Holistic-DAG directly constructs a natural-language proposition DAG from the problem and solution and formalizes the entire graph in Lean. Uni-DAG first constructs and compiles shared Lean terms from the original problem and the solution's extracted implicit proof premises. These terms define the mathematical objects, domains, constraints, and a designated `TARGET_TERM`, and remain fixed throughout DAG construction and formalization. Neither configuration performs proof search, so the comparison evaluates semantic preservation during formalization.

After artifact generation, we use DeepSeek-V4-Flash-0731 (DeepSeek-AI et al., 2026) as an independent semantic-fidelity judge. The judge compares each generated DAG and its Lean declarations against the original problem and complete natural-language solution, assigning a separate binary label to each error category. A sample may receive multiple labels, and the

[1] We sample 5 solutions per problem on MATH-500 and 16 per problem on each of the other three datasets.

bar for each category counts the samples assigned that label. A sample is counted as *Faithful* only when all five error labels are absent and the judge assigns an overall faithful verdict. This evaluation measures preservation of the supplied reasoning, not whether that reasoning is mathematically correct. The complete evaluation prompt is provided in Appendix H.1.

Figure 1 shows that Uni-DAG yields 576 samples judged faithful, compared with 235 for Holistic-DAG, while reducing all five error counts. The largest reductions occur in final-target mismatch (F4, −375) and proof-process or dependency loss (F3, −341), with a further reduction in proof-obligation bypass (F5, −232). These findings suggest that fixing and reusing mathematical definitions before DAG construction helps preserve both mathematical meaning and reasoning dependencies. They motivate NF-Bridge's shared Lean-term, which maintains consistent object definitions across the claims submitted to Lean for verification.

3 Method

Motivated by the findings of our preliminary study, we propose **NF-Bridge**, a framework that uses frozen shared Lean terms to improve consistency between NL and FL, and converts Lean verification evidence into ranking signals for TTS. As illustrated in Figure 2, NF-Bridge comprises three stages: *Shared Lean-Term Construction*, *Faithfulness-Guided Formalization*, and *Dependency-Aware Verification*. Together, these stages support faithful NL-to-FL translation and convert Lean verification feedback into candidate-ranking signals.

We first introduce the problem definition and notation: Given a mathematical problem x, we sample k solutions from an LLM to form a candidate pool $\{y_i\}_{i=1}^{k}$. For each candidate y_i, we extract its implicit proof premises $\mathcal{I}_i$ and use them with the original problem to construct shared Lean terms $\mathcal{T}_i$. The frozen $\mathcal{T}_i$ fixes the interpretation of mathematical terms, constraints, and the target throughout subsequent stages. We represent the solution's NL claims and their Lean counterparts as directed acyclic graphs $\mathcal{G}_{N,i}$ and $\mathcal{G}_{F,i}$, respectively, and jointly refine these representations using feedback from the Lean compiler and a semantic-fidelity verifier. Finally, we schedule proof and refutation searches according to the graph dependencies and aggregate the resulting Lean verification feedback into a score s_i for ranking and selecting candidate solutions.

3.1 Shared Lean-Term Construction

Direct NL-to-FL translation can introduce inconsistencies in variable definitions, domains, and constraints. In particular, prematurely encoding the target conclusion (the claim to be established in the original problem) in a variable definition or introducing it as an additional assumption may bypass the intended proof obligation (the corresponding formal statement to be proved under the original assumptions). To reduce these errors, we first extract implicit proof premises $\mathcal{I}_i$ and use them solely to construct shared Lean terms $\mathcal{T}_i$. Only $\mathcal{T}_i$ is frozen and reused by subsequent stages.

Implicit Proof Premises $\mathcal{I}_i$. Given a problem x and candidate solution y_i, we extract the solution's implicit proof premises $\mathcal{I}_i$ (such as assumptions underlying coordinate selection, orientation conventions, or symmetry reduction) and express them in natural language.

Shared Lean Terms $\mathcal{T}_i$. We then construct $\mathcal{T}_i$ using only x and the extracted premises $\mathcal{I}_i$. These shared Lean terms specify a unique `TARGET_TERM` to represent the target requested by the problem. These terms remain fixed throughout subsequent formalization and verification and are reused to formulate intermediate results and the final proof obligation. We implement both construction steps with LLMs:

$$\mathcal{I}_i = \text{ExtractPremises}(x, y_i), \qquad \mathcal{T}_i = \text{ConstructTerms}(x, \mathcal{I}_i).$$

If y_i contains no such premises, we explicitly set $\mathcal{I}_i =$ `none`. The prompts are provided in Appendix H.2. To validate $\mathcal{T}_i$, we first compile its Lean declarations to check syntactic and type correctness, then assess semantic fidelity using a separate judge. If either check fails, we revise $\mathcal{T}_i$ using the corresponding feedback and repeat the validation process. Only after both checks pass do we freeze $\mathcal{T}_i$ and reuse its declarations throughout subsequent stages.

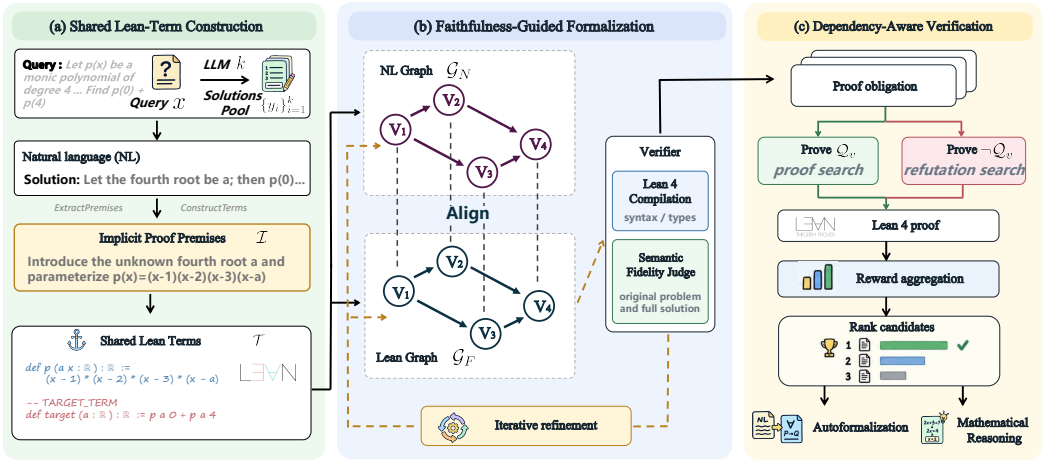

Figure 2: **Overview of NF-Bridge.** (a) Extract implicit proof premises and construct shared Lean terms. (b) Aligned NL and Lean DAGs are jointly refined using compilation feedback and semantic-fidelity checks. (c) Dependency-aware verification for proofs of Q_v and $\neg Q_v$ produce Lean-checked evidence for autoformalization and reward-based candidate ranking in TTS.

3.2 Faithfulness-Guided Formalization

Given the $\mathcal{T}_i$ and a candidate solution y_i, we construct a natural-language DAG $\mathcal{G}_{N,i}$. Each node represents a mathematical claim, and each directed edge records a direct dependency between claims in y_i. The graph retains answer-relevant claims and includes a single designated final node whose claim must explicitly reference `TARGET_TERM`.

We then construct $\mathcal{G}_{F,i}$ as the formal counterpart of $\mathcal{G}_{N,i}$. Each natural-language claim is paired with a Lean proposition, preserving the node correspondence and direct dependencies between the two graphs. The formalization reuses the shared terms $\mathcal{T}_i$ to maintain consistent term definitions, domains, and constraints across nodes. We first declare registered auxiliary terms, then define the corresponding Lean propositions in topological order. Additional auxiliary terms and local variables must be explicitly declared, and auxiliary definitions must not introduce unproved conclusions as assumptions. Together, the two graphs make the correspondence between the candidate's reasoning and its formal representation explicit at the level of claims and dependencies. Both graphs are generated using LLMs:

$$\mathcal{G}_{N,i} = \text{BuildDAG}(x, y_i, \mathcal{T}_i), \qquad \mathcal{G}_{F,i} = \text{FormalizeDAG}(\mathcal{G}_{N,i}, \mathcal{T}_i).$$

To validate this correspondence, we compile the Lean declarations of $\mathcal{G}_{F,i}$ to check syntactic and type correctness and apply a separate semantic-fidelity judge. The judge checks whether $\mathcal{G}_{N,i}$ preserves the key claims and direct dependencies in y_i, and whether each Lean proposition in $\mathcal{G}_{F,i}$ matches its natural-language counterpart. It also checks that term references, constraints, and the final target remain consistent with x and the frozen $\mathcal{T}_i$. If either check fails, we iteratively refine $\mathcal{G}_{N,i}$ and $\mathcal{G}_{F,i}$ together using the corresponding feedback, while keeping $\mathcal{T}_i$ fixed. Compiler diagnostics can guide these revisions, helping correct formalization errors and unsupported additions. Coordinating updates across both graphs helps maintain NL and FL alignment and reduces the risk that correcting one representation introduces inconsistencies in the other. Only after both checks pass do we freeze the two graphs for subsequent dependency-aware verification. These checks assess formal well-formedness and semantic correspondence; they do not prove the node propositions.

3.3 Dependency-Aware Verification

Once $\mathcal{G}_{F,i}$ is validated, we construct a complete proof obligation $Q_{i,v}$ for each node v in $\mathcal{G}_{F,i}$, incorporating its parameters, problem assumptions, and direct dependency hypotheses. A node becomes eligible for verification only after all its prerequisites have been proved. Failure

to prove a node may reflect limited proof-search capability or an invalid claim. To distinguish verified errors from inconclusive searches, we introduce dual verification. For each eligible node, we run two proof searches: one targeting $Q_{i,v}$ and the other targeting its exact outer negation $\neg Q_{i,v}$. We call these the positive and negative verification directions, respectively. We accept either result only when the generated proof is checked by Lean 4. The exact construction of parameter binders, dependency hypotheses, and outer-negation targets is detailed in Appendix B.1. Each node receives one of four states:

- `proved`: a Lean-checked proof of $Q_{i,v}$ is obtained;
- `disproved`: a Lean-checked proof of $\neg Q_{i,v}$ is obtained;
- `unresolved`: neither verification direction yields a verified proof within the allocated budget;
- `blocked`: verification cannot proceed because a prerequisite remains unproved.

Proved nodes unlock their dependent nodes. Full verification of the retained reasoning requires every node to be proved under the formalized assumptions. An unresolved or blocked node leaves this verification incomplete without establishing that the corresponding claim is false.

Tool-Assisted Verification. During verification, the LLM can invoke four tools. `mathlib_search` retrieves relevant Mathlib (The mathlib Community, 2020) theorems and APIs, while `lean_scratch` tests candidate signatures, auxiliary facts, and proof strategies in an isolated Lean context. `lean_compile` checks a complete proof and is the only source of formally verified proof or refutation evidence. When the model determines that its assigned verification target cannot be established, it may invoke `reject_direction` to terminate the attempt and avoid unnecessary reasoning and compilation cost. We call this action a direction rejection; it is not a formal certificate.

Candidate Ranking. We aggregate these node states into a ranking score for each candidate solution y_i. Let V_i denote the nodes of $\mathcal{G}_{F,i}$ and $\sigma_i(v)$ the state of node v. We assign a score of zero if any node is disproved; otherwise, we use the fraction of proved nodes:

$$s_i = \begin{cases} 0, & \exists v \in V_i : \sigma_i(v) = \texttt{disproved}, \\ \dfrac{1}{|V_i|} \displaystyle\sum_{v \in V_i} \mathbf{1}[\sigma_i(v) = \texttt{proved}], & \text{otherwise.} \end{cases}$$

This rule penalizes candidates with verified refutations and uses verified coverage to rank the remaining candidates, even when complete verification is unavailable. For an unresolved proposition, consistent rejection of one verification direction across the allocated attempts provides a weak direction-rejection signal for the opposite target during candidate ranking. This signal affects only ranking; complete details are provided in Appendix F.3.

3.4 Applications to Autoformalization and Mathematical Reasoning

Proof Autoformalization. NF-Bridge produces a formal DAG together with Lean declarations and the proof or refutation certificates obtained during verification. The retained reasoning is fully verified under the formalized assumptions only when all nodes are proved.

Test-Time Scaling. We evaluate each candidate with NF-Bridge and obtain its verification-based score s_i. This score guides best-of-K selection:

$$i^\star \in \arg\max_{1 \le i \le K} s_i, \qquad \hat{y} = y_{i^\star}. \tag{1}$$

We return the final answer extracted from $\hat{y}$. This procedure uses node-level proof and refutation evidence to guide candidate selection, allowing verified intermediate claims to inform ranking even when complete verification is unavailable.

Table 1: Robustness of proof autoformalization under global perturbations, measured by FullyCorrect (%) on miniF2F and MATH-500 across five natural-language proof styles.

Model	miniF2F					MATH-500				
	Orig	G-FF	G-Step	Q-FF	Q-Step	Orig	G-FF	G-Step	Q-FF	Q-Step
Kimina-Prover-RL-1.7B	9.4	10.2	11.5	11.5	8.6	6.6	8.2	6.8	8.2	9.6
Kimina-Prover-Distill-8B	19.3	16.0	16.4	19.7	16.0	19.8	16.4	16.8	16.8	19.0
DeepSeek-Prover-V2-7B	1.2	8.6	7.4	12.7	8.6	8.2	7.4	5.8	6.4	5.0
Goedel-Prover-V2-8B	1.2	2.5	4.1	3.7	4.1	4.4	2.0	2.0	3.8	3.0
Gemini-3.1-Pro	17.2	21.7	18.9	21.7	22.5	26.4	23.6	24.6	23.8	23.8
ProofBridge	21.3	19.7	18.4	17.6	17.6	13.4	14.4	12.6	14.8	15.2
ProofFlow	9.0	6.1	8.2	7.0	9.8	10.8	8.2	8.0	8.4	13.0
NF-Bridge	**67.6**	**66.4**	**65.2**	**71.7**	**74.6**	**71.8**	**74.0**	**73.0**	**73.8**	**73.8**

4 Experiments

4.1 Experimental Setup

Evaluation tasks. We evaluate NF-Bridge in four settings: robustness to proof rewrites on RobustPABench (Gui et al., 2026); test-time candidate selection on AIME24, AIME25, MATH-500 (Lightman et al., 2024), and BRUMO25 (Balunović et al., 2025); full-proof autoformalization on ProofFlowBench (Cabral et al., 2026) and miniF2F (Zheng et al., 2021); and isolated node-level proving. Dataset construction, sample counts, and task-specific protocols are detailed in Appendices E–G.

Models and training. Qwen3-8B (Yang et al., 2025) generates the fixed candidate pools, while Qwen3.8-27B (Qwen Team, 2026b) performs autoformalization. Our 9B prover is initialized from Qwen3.5-9B (Qwen Team, 2026a) and trained by SFT followed by verifier-grounded GRPO (Shao et al., 2024) on node-level trajectories constructed from ODA-Math-460k (Gao et al., 2026). Semantic faithfulness is independently evaluated by Qwen3-235B-A22B (Yang et al., 2025). All formal artifacts are checked with Lean 4.32.0-rc1 (de Moura & Ullrich, 2021) through Kimina Lean Server.[2] Complete model, training, and decoding settings are provided in Appendix C.

Baselines and metrics. We compare against published RobustPABench systems, standard and learned candidate-selection baselines, full-proof autoformalization methods, and general-purpose theorem provers. We report *FullyCorrect* for robustness, answer accuracy for candidate selection, SC, SF, and SC $\wedge$ SF for full-proof autoformalization, and success rate and wall-clock time for isolated proving. Exact baseline configurations, metric definitions, judging procedures, and runtime accounting are provided in Appendices E.5, F, E, and G.

4.2 Robustness of Proof Autoformalization

We evaluate the 27B NF-Bridge configuration under the global proof-style perturbations of RobustPABench (Gui et al., 2026). Table 1 reports FullyCorrect across the original proofs and four global rewrites; full protocol details are provided in Appendix E.5.

NF-Bridge achieves the highest FullyCorrect score in all ten settings, outperforming the strongest baseline by 44.7–52.1 percentage points on miniF2F and 45.4–50.4 points on MATH-500. Its consistently strong performance across proof styles demonstrates the robustness of the complete autoformalization pipeline to natural-language rewrites.

4.3 Test-Time Candidate Selection

We evaluate whether machine-checkable verification signals can select a correct response from a fixed candidate pool. For a model-scale-matched comparison with learned reward-model baselines (Zhang et al., 2025; Liu et al., 2024b;a; Wang et al., 2024a;b), Table 2

[2] `https://github.com/project-numina/kimina-lean-server`

Table 2: Test-time candidate-selection accuracy (%). Oracle reports pass@N under the available rollout budget. A dash denotes an unreported result.

Method	**AIME24**	**AIME25**	**MATH-500**	**BRUMO25**
Oracle / pass@N	93.33	83.33	88.00	63.33
pass@1	76.67	66.67	85.80	53.33
Random (seed 42)	76.67	76.67	86.80	60.00
Majority vote	83.33	80.00	86.80	63.33
Qwen2.5-Math-PRM-7B	**86.67**	70.00	85.80	**60.00**
AceMath-7B-RM	80.00	53.33	86.80	56.67
Skywork-Reward-Llama-3.1-8B-v0.2	80.00	70.00	86.60	56.67
ArmoRM-Llama3-8B-v0.1	80.00	70.00	86.00	50.00
RLHFlow-Llama3.1-8B-PRM-DeepSeek	80.00	**80.00**	86.20	53.33
NF-Bridge	**86.67**	**80.00**	**87.00**	**60.00**

Table 3: Main results on full-proof autoformalization benchmarks. Time is average single-A800 seconds per problem; OT(k) is omitted.

	ProofFlowBench				**miniF2F**			
Method	**Time(s)**	**SC**	**SF**	**SC ∧ SF**	**Time(s)**	**SC**	**SF**	**SC ∧ SF**
ProofBridge	113.48	92.93	9.24	9.24	61.23	95.08	29.51	28.69
Codex End2End	104.22	25.00	32.61	21.20	83.62	49.18	58.20	47.54
M2F	882.47	35.33	38.59	12.50	503.46	10.66	19.67	7.79
Codex D-Correction	1554.50	57.14	23.63	10.99	1172.33	71.72	33.61	18.44
Monotonic	201.39	**84.78**	15.22	14.67	229.04	**100.00**	16.80	16.80
ProofFlow	2348.48	36.41	33.15	15.76	1125.73	61.89	46.31	35.25
ToMAP-Qwen30B	610.10	66.30	27.71	21.10	513.44	86.89	34.02	29.92
ToMAP-Gemini	850.27	63.59	51.09	40.22	665.41	82.79	63.93	55.74
NF-Bridge-27B	955.74	46.74	**76.63**	**40.76**	588.49	69.67	86.89	64.75
NF-Bridge-9B	284.17	42.39	75.00	35.87	177.89	72.95	**87.70**	**67.62**

Notes. SC denotes syntactic correctness, SF denotes semantic faithfulness, and SC∧SF denotes joint correctness. Runtime accounting and complete experimental settings are provided in Appendix E.

reports NF-Bridge using the 9B prover. The selector combines the verifier-grounded score s_i with the conservative direction-rejection signal defined in Section 3.3; complete ranking details are provided in Appendix F. NF-Bridge matches the strongest learned reward-model result on AIME24, AIME25, and BRUMO25, and exceeds it by 0.20 percentage points on MATH-500, demonstrating that structured Lean evidence provides a useful selection signal across both competition-style and large-scale benchmark settings.

4.4 Full-Proof Autoformalization

Following the full-proof autoformalization protocol of Liu et al. (2026), we evaluate the underlying capability of NF-Bridge on ProofFlowBench and miniF2F under the shared setup in Section 4.1. We compare against ProofBridge (Jana et al., 2026), ProofFlow (Cabral et al., 2026), M2F (Wang et al., 2026), Monotonic (Zhang et al., 2026), and the ToMAP baselines (Liu et al., 2026); Codex End2End and Codex D-Correction are baseline configurations introduced by ToMAP. The system prompts are reproduced in Appendix H.2; benchmark construction, semantic judging, and runtime accounting are provided in Appendix E.

On ProofFlowBench, NF-Bridge-27B obtains the highest SF (76.63%) and joint score (40.76%), improving the strongest baseline by 25.54 and 0.54 percentage points, respectively. On miniF2F, NF-Bridge-9B achieves the highest SF (87.70%) and joint score (67.62%), exceeding the strongest baseline by 23.77 and 11.88 points. The 9B configuration is also 3.36× and 3.31× faster than the 27B configuration on ProofFlowBench and miniF2F, respectively, while retaining stronger joint correctness on miniF2F; the 27B configuration remains stronger on ProofFlowBench.

Table 4: Accuracy (%) and single-A800 wall-clock time on the isolated prover evaluation.

Model	Overall	Positive	Negation	Reject	Time
NF-Bridge-9B	83.98	**85.16**	67.97	98.83	2:39:37
NF-Bridge-27B	**85.42**	**85.16**	**71.09**	**100.00**	7:18:25
Goedel-Prover-V2-8B	19.92	43.36	16.41	0.00	**1:06:25**

Table 5: Stage-A/B semantic-judge ablation on 184 ProofFlowBench examples.

Stage-A judge	Stage-B judge	Faithful	SF (%)
Disabled	Disabled	119/184	64.67
Enabled	Disabled	127/184	69.02
Disabled	Enabled	128/184	69.57
Enabled	Enabled	**136/184**	**73.91**

Table 6: Pooled candidate-selection accuracy (%) under different verifier signals.

Prover	Proof coverage	+ Disproof	+ Reject-3
9B	84.58	84.92	**85.25**
27B	84.58	84.58	**84.92**

4.5 Prover Accuracy and Efficiency

We isolate the Stage-C prover on positive-proof, exact-negation, and direction-rejection tasks and compare against Goedel-Prover-V2-8B (Lin et al., 2025); complete settings are provided in Appendix G.

NF-Bridge-9B achieves 83.98% accuracy, within 1.44 points of NF-Bridge-27B while running 2.75× faster. Goedel-Prover-V2-8B reaches only 19.92% and fails on direction rejection, motivating a specialized prover for the proof, exact-negation, and rejection behaviors required by NF-Bridge.

4.6 Ablation Study

Stage-A/B semantic judge ablation. We conduct a 2×2 ablation of the Stage-A and Stage-B semantic judges on all 184 ProofFlowBench examples, using Qwen3.8-27B for autoformalization and Qwen3-235B-A22B for evaluation while keeping all other sampling settings fixed. At the aggregate level, self-judge refinement improves semantic faithfulness, with both judges together increasing SF from 64.67% to 73.91%.

Verifier-signal ablation. We compare three nested ranking rules. *Proof coverage* ranks candidates by the fraction of proved nodes alone; *+ Disproof* additionally penalizes candidates containing Lean-compiled counterevidence; and *+ Reject-3* further incorporates direction-rejection consensus across three independent prover attempts. Table 6 reports pooled accuracy over all 590 problems, with complete per-benchmark results in Appendix F.5. Together, compiled disproof and the conservative direction-rejection signal yield the highest pooled accuracy at both prover scales, although their effects vary across individual benchmarks.

5 Conclusion

We presented NF-Bridge, which freezes shared Lean terms, aligns natural-language and formal proposition DAGs, and verifies their nodes through dependency-aware proof and exact-negation search. This produces localized, machine-checkable evidence for faithful autoformalization and test-time candidate selection. Experiments demonstrate robust semantic and joint correctness, competitive selection performance, and near-27B prover accuracy from a substantially faster 9B model, showing that dependency-structured formal verification can provide auditable feedback for mathematical reasoning.

AI Use Statement

We used OpenAI Codex to assist with research ideation and execution, including code development, experimental planning, result analysis, and drafting and polishing sections of the manuscript. It was also used to support literature retrieval and discovery. All AI-assisted text, references, analyses, and suggestions were reviewed and verified by the authors. The authors made the final methodological and experimental decisions and remain responsible for the manuscript, its claims, and the accompanying research artifacts.

References

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. MathArena: Evaluating LLMs on uncontaminated math competitions. *arXiv preprint arXiv:2505.23281*, 2025. doi: 10.48550/arXiv.2505.23281. URL https://arxiv.org/abs/2505.23281.

Rafael Cabral, Tuan Manh, Xuejun Yu, Wai Ming Tai, Zijin Feng, and Shen Xin. ProofFlow: A dependency graph approach to faithful proof autoformalization. In *International Conference on Learning Representations*, volume 2026, pp. 58459–58483, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/5fce5198dbf92d5ff45f74504431e2ff-Abstract-Conference.html.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pp. 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026. doi: 10.48550/arXiv.2606.19348. URL https://arxiv.org/abs/2606.19348.

Xin Gao, Xiaoyang Wang, Yun Zhu, Mengzhang Cai, Conghui He, and Lijun Wu. Closing the data loop: Using OpenDataArena to engineer superior training datasets. *arXiv preprint arXiv:2601.09733*, 2026. URL https://arxiv.org/abs/2601.09733.

Zhengtao Gui, Sheng Yang, and Zhouxing Shi. Evaluating the robustness of proof autoformalization in Lean 4. *arXiv preprint arXiv:2606.14867*, 2026. URL https://arxiv.org/abs/2606.14867.

Prithwish Jana, Kaan Kale, Ahmet Tanriverdi, Cruise Song, Sriram Vishwanath, and Vijay Ganesh. ProofBridge: Auto-formalization of natural language proofs in Lean via joint embeddings. In *International Conference on Learning Representations*, volume 2026, pp. 13569–13591, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/16f5b8d916f13534d11c50adf60914e4-Abstract-Conference.html.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, et al. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026. doi: 10.48550/arXiv.2607.24653. URL https://arxiv.org/abs/2607.24653.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/aca97732e30bcf1303bc22ac3924fd16-Abstract-Conference.html.

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

Chengwu Liu, Ye Yuan, Yichun Yin, Yan Xu, Xin Xu, Zaoyu Chen, Yasheng Wang, Lifeng Shang, Qun Liu, and Ming Zhang. Safe: Enhancing mathematical reasoning in large language models via retrospective step-aware formal verification. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12171–12186, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.594. URL https://aclanthology.org/2025.acl-long.594/.

Chris Yuhao Liu, Liang Zeng, Jiacai Liu, Rui Yan, Jujie He, Chaojie Wang, Shuicheng Yan, Yang Liu, and Yahui Zhou. Skywork-Reward: Bag of tricks for reward modeling in LLMs. *arXiv preprint arXiv:2410.18451*, 2024a. URL https://arxiv.org/abs/2410.18451.

Tian-Shuo Liu, Shiyuan Zhang, Zijie Geng, Haoyu Liu, Runjie Xu, Pengyuan Wang, Lei Yuan, and Yang Yu. Efficient test-time optimization for multi-agent proof autoformalization. *arXiv preprint arXiv:2607.11307*, 2026. URL https://arxiv.org/abs/2607.11307.

Zihan Liu, Yang Chen, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. AceMath: Advancing frontier math reasoning with post-training and reward modeling. *arXiv preprint arXiv:2412.15084*, 2024b. URL https://arxiv.org/abs/2412.15084.

Jianqiao Lu, Yingjia Wan, Yinya Huang, Jing Xiong, Zhengying Liu, and Zhijiang Guo. FormalAlign: Automated alignment evaluation for autoformalization. In *International Conference on Learning Representations*, volume 2025, pp. 102122–102149, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/fceedf8c9c0ff51f41b9fe0294ef0070-Abstract-Conference.html.

Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5153–5176, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.313. URL https://aclanthology.org/2023.emnlp-main.313/.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.248. URL https://aclanthology.org/2023.findings-emnlp.248/.

Qwen Team. Qwen3.5: Towards native multimodal agents. Qwen Blog, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025. URL https://arxiv.org/abs/2504.21801.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pp. 367–381, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025. URL https://arxiv.org/abs/2504.11354.

Haoxiang Wang, Wei Xiong, Tengyang Xie, Han Zhao, and Tong Zhang. Interpretable preferences via multi-objective reward modeling and mixture-of-experts. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10582–10592. Association for Computational Linguistics, 2024a. doi: 10.18653/v1/2024.findings-emnlp.620. URL https://aclanthology.org/2024.findings-emnlp.620/.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439. Association for Computational Linguistics, 2024b. doi: 10.18653/v1/2024.acl-long.510. URL https://aclanthology.org/2024.acl-long.510/.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Zichen Wang, Wanli Ma, Zhenyu Ming, Gong Zhang, Kun Yuan, and Zaiwen Wen. M2F: Automated formalization of mathematical literature at scale. *arXiv preprint arXiv:2602.17016*, 2026. doi: 10.48550/arXiv.2602.17016. URL https://arxiv.org/abs/2602.17016.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *arXiv preprint arXiv:2205.12615*, 2022. URL https://arxiv.org/abs/2205.12615.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Lan Zhang, Marco Valentino, and André Freitas. Monotonic reference-free refinement for autoformalization. *arXiv preprint arXiv:2601.23166*, 2026. doi: 10.48550/arXiv.2601.23166. URL https://arxiv.org/abs/2601.23166.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10495–10516. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.547. URL https://aclanthology.org/2025.findings-acl.547/.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. MiniF2F: A cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021. doi: 10.48550/arXiv.2109.00110. URL https://arxiv.org/abs/2109.00110.

A Related Work

A.1 Formal Theorem Proving and Autoformalization

Formal theorem proving starts from a formal-language statement and searches for a proof that can be checked by a proof assistant such as Lean 4 (de Moura & Ullrich, 2021). Recent LLM-based provers have substantially improved whole-proof generation through reinforcement learning, subgoal decomposition, verifier-guided data synthesis, and self-correction (Wang et al., 2025; Ren et al., 2025; Lin et al., 2025). These systems primarily address proof search after the theorem statement has already been formalized. Autoformalization instead translates natural-language mathematics into formal specifications or proofs (Wu et al., 2022). While early work largely emphasized theorem statements, full-proof autoformalization must additionally recover intermediate claims, local assumptions, side conditions, and their dependencies across an entire argument (Jana et al., 2026; Cabral et al., 2026).

This distinction is central to NF-Bridge. Our objective is not merely to find a Lean proof of the problem's conclusion, but to construct a formal representation that preserves the mathematical commitments of a particular candidate solution. Lean acceptance establishes correctness for the generated formal statement, whereas faithful verification of natural-language reasoning additionally requires that this statement, its intermediate propositions, and its dependency structure continue to represent the original argument.

A.2 Structure-Aware Full-Proof Autoformalization

ProofFlow treats structural fidelity as a primary objective of proof autoformalization (Cabral et al., 2026). It first represents a natural-language proof as a dependency DAG and then formalizes proof steps as intermediate Lean lemmas, so that each step receives only the dependencies prescribed by the source proof. ProofFlow also introduces ProofFlowBench and ProofScore, which jointly consider syntactic correctness, semantic faithfulness, and preservation of proof structure. This dependency-aware view demonstrates that a compiling proof may still be an unfaithful translation if it bypasses intermediate reasoning or uses unintended premises.

ToMap formulates full-proof autoformalization as a Decomposer–Formalizer–Prover pipeline and studies how test-time computation should be allocated across it (Liu et al., 2026). Its weak-link analysis identifies decomposition as the highest-leverage interface: atomic, self-contained, and dependency-aware proof units make downstream formalization and proving more tractable. ToMap therefore evolves candidate decompositions using rubric feedback for semantic faithfulness, provability, and Lean-friendliness, reserving the more expensive Formalizer–Prover execution and Lean verification for selected candidates.

NF-Bridge builds on this structured-decomposition perspective while introducing an explicit semantic interface before the proof is decomposed. It extracts solution-induced implicit premises, constructs and freezes shared Lean terms, and reuses these terms across paired natural-language and formal DAGs. Compilation and semantic-fidelity feedback are then used to refine the two graphs while keeping the shared terms fixed. Finally, dependency-aware proof and exact-refutation searches produce node-level Lean evidence. Thus, the formal DAG serves both as an autoformalization artifact and as an auditable interface through which verification feedback can be attributed to individual claims.

A.3 Faithfulness and Robustness Evaluation

Semantic evaluation is necessary because type-correct formal code need not express the intended mathematics. FormalAlign studies automated alignment evaluation for autoformalization (Lu et al., 2025), while ProofFlow evaluates semantic and structural fidelity at the level of proof steps and their dependencies (Cabral et al., 2026). RobustPABench broadens this perspective by evaluating proof autoformalizers under changes to their natural-language inputs (Gui et al., 2026). Its global perturbations paraphrase a proof while preserving its mathematical content and test whether correctness remains stable across styles. Its lo-

cal perturbations alter a value, symbol, or proof step and test whether the formalization faithfully reflects the change rather than silently reverting to another argument.

These protocols motivate treating faithfulness as distinct from compilation success. NF-Bridge incorporates this distinction during construction: Stage A checks whether the shared Lean terms preserve the problem and solution-induced modeling choices, and Stage B checks the correspondence between each natural-language proposition and its Lean counterpart. Our robustness experiment uses the global-perturbation portion of RobustPABench to assess whether the resulting end-to-end formalizations remain correct across proof styles; local-perturbation robustness is outside the scope of the present evaluation.

A.4 Formal Verification for Mathematical Reasoning

Test-time selection for mathematical reasoning commonly relies on answer frequency, learned outcome verifiers, or process reward models (Wang et al., 2023; Cobbe et al., 2021; Lightman et al., 2024; Zhang et al., 2025). Such signals can improve selection accuracy but do not themselves provide machine-checkable mathematical certificates. Neurosymbolic systems instead connect language models to symbolic solvers or proof systems (Olausson et al., 2023; Pan et al., 2023), and recent work uses formal verification feedback to assess or improve intermediate mathematical reasoning (Liu et al., 2025). However, a formal verifier certifies only the formal object it receives; its guarantee transfers to the original natural-language solution only when the intervening translation is faithful.

NF-Bridge therefore places faithfulness-constrained autoformalization before verifier-based ranking. For each retained proposition, it searches for a Lean proof of the proposition and its exact negation, separating verified refutations from inconclusive proof-search failures. Candidate scores then aggregate node-level compiled evidence, with direction-rejection consensus used only as a weak ranking signal. This produces structured feedback with local attribution while preserving the distinction between Lean-certified evidence and model-derived heuristics.

B Formal Contracts and Algorithmic Details

After the Stage-B gate, the shared Lean terms $\mathcal{T}_i$ and the paired graphs $\mathcal{G}_{N,i}$ and $\mathcal{G}_{F,i}$ are immutable. Stage C may use their registered declarations and dependency edges, but it may neither alter their mathematical meanings nor introduce unregistered assumptions. Consequently, each verification result refers to the proposition fixed by Stage B rather than to a prover-generated reformulation.

B.1 Dependency-Grounded Target Construction

Let $\varphi_{i,v}(\theta_{i,v})$ denote the Lean proposition of node v, with parameter binders $\theta_{i,v}$ and direct parents $D_i(v)$. Each parent $u \in D_i(v)$ contributes a hypothesis binder $h_u : \delta_{i,u\to v}$. When the parent parameters can be instantiated from $\theta_{i,v}$, $\delta_{i,u\to v}$ is the corresponding instance of $\varphi_{i,u}$; otherwise it is the universally quantified proposition $\forall\theta_{i,u}, \varphi_{i,u}(\theta_{i,u})$. The complete proof obligation is

$$Q_{i,v} = \forall\theta_{i,v}\,(h_u : \delta_{i,u\to v})_{u\in D_i(v)},\ \varphi_{i,v}(\theta_{i,v}), \qquad Q^{+}_{i,v} = Q_{i,v}, \quad Q^{-}_{i,v} = \neg Q_{i,v}. \tag{2}$$

Thus, the negative verification target negates the entire quantified producer type, including its dependency hypotheses, rather than only the node proposition. Intermediate targets are emitted as Lean `lemma`s and the designated final target as a `theorem`.

For example, consider a monic quartic with roots $1, 2, 3$ and an unknown fourth root a. The frozen terms define $p(a, x) = (x-1)(x-2)(x-3)(x-a)$ and $\texttt{target}(a) = p(a,0) + p(a,4)$. Stage B constructs

$$\varphi_1(a) : p(a,0) = 6a, \qquad \varphi_2(a) : p(a,4) = 24 - 6a, \qquad \varphi_f(a) : \texttt{target}(a) = 24, \tag{3}$$

with $D_i(f) = \{1, 2\}$. Stage C therefore uses

$$Q_{i,f} = \forall(a : \mathbb{R})\,(h_1 : \varphi_1(a))\,(h_2 : \varphi_2(a)),\ \varphi_f(a), \qquad Q^{-}_{i,f} = \neg Q_{i,f}. \tag{4}$$

This construction makes both direct dependencies explicit in the final theorem type.

B.2 Scheduling and Evidence Semantics

A node enters the ready set only after every direct parent has state `proved`; only a newly proved node can unlock its children. A Lean-compiled proof of $Q^+_{i,v}$ assigns `proved`, while a Lean-compiled proof of $Q^-_{i,v}$ assigns `disproved`. If neither verification direction succeeds within the allocated budget, the node is `unresolved`; if a prerequisite is not proved, it remains `blocked`. Target-construction and infrastructure failures are recorded separately and are not interpreted as mathematical verdicts. Likewise, `reject_direction` can terminate an unpromising attempt and contribute a weak direction-rejection signal, but it never changes a node's Lean-certified state.

C Models, Prover Training, and Formal Environment

C.1 Model Roles and Checkpoints

The NF-Bridge (prover-27B) configuration uses Qwen3.8-27B throughout Stages A–C. The NF-Bridge (prover-9B) configuration reuses the frozen Stage-A/B artifacts produced by Qwen3.8-27B and replaces only the Stage-C prover with the Qwen3.5-9B GRPO step-63 checkpoint described below. Qwen3-8B generates the fixed candidate pools used by all TTS selectors. Semantic judgments are produced independently by Qwen3-235B-A22B, so the proof-generating model does not judge its own semantic fidelity. The complete system prompts are reproduced in Appendix H.2.

C.2 Training-Data Construction

The 9B prover is trained on node-level Stage-C trajectories produced by running NF-Bridge over ODA-Math-460k. The SFT corpus contains 41,124 examples from 6,421 source records: 22,250 compiler-verified proof or disproof trajectories and 18,874 validated direction-rejection trajectories. We exclude unresolved nodes, nodes blocked by an unproved dependency, target-assembly failures, infrastructure failures, and examples exceeding the 32,768-token cutoff.

The GRPO corpus contains 2,688 prompts divided equally among three sampling groups: rare or hard verification-direction targets, verified proof targets, and validated rejection targets. Each group contains 879 distinct core prompts and 17 padding rows used to form complete batches, giving 2,637 non-padding prompts in total. The proof and rejection groups have no task-level overlap. Training uses batches of 128 prompts, 21 batches per epoch, and three epochs, for 63 optimization steps.

C.3 Supervised Fine-Tuning Configuration

Starting from Qwen3.5-9B, we train for three epochs with a global batch size of 32, micro-batch size 1, and maximum sequence length 32,768. We use AdamW with learning rate 1×10^{-5}, zero weight decay, $\beta = (0.9, 0.999)$, a cosine schedule with warmup ratio 0.05, and gradient-norm clipping at 1.0. Training uses bfloat16, activation checkpointing, and seed 42. The cumulative SFT duration is 8:06:02 on 16 NVIDIA A800 GPUs.

C.4 Verifier-Grounded GRPO Configuration

The GRPO stage initializes from the SFT checkpoint and trains for three epochs with train batch size 128, PPO mini-batch size 128, one PPO epoch per batch, and 16 rollouts per prompt. Rollouts use temperature 1.0, top-p = 1.0, a maximum prompt length of 12,224 tokens, a maximum response length of 20,480 tokens, and a maximum context length of 32,768 tokens. The actor learning rate is 1×10^{-6} with a cosine schedule, the PPO clipping ratio is 0.2, and KL reward is disabled. The cumulative GRPO duration is 15:23:17 on 16 NVIDIA A800 GPUs.

Reward construction. The reward is computed from the verified trajectory state rather than from surface-form matching. For a compiled proof of the assigned verification target, let $k \in \{1, \ldots, K\}$ be the successful `lean_compile` submission and let $K = 4$ be the maximum number of compile submissions within one prover attempt. The proof reward is

$$r_{\text{proof}} = 1 + 0.25\,\mathbf{1}[\text{discovery used}] + 0.15\frac{K-k}{K-1}. \tag{5}$$

The discovery bonus records the use of Mathlib search or an isolated scratch check, while the final term rewards earlier compilation success. For an explicit direction rejection, the evidence-conditioned reward is

$$r_{\text{reject}} = \begin{cases} -1.0, & \text{the assigned target is already verified,} \\ +1.0, & \text{the opposite verification target is verified,} \\ -0.1, & \text{the target is hard and the opposite verification direction is rejected,} \\ +0.1, & \text{the assigned verification direction is rejected and the opposite target is hard.} \end{cases} \tag{6}$$

A trajectory with neither a compiled proof nor an admissible rejection receives zero reward. Infrastructure failures and trajectories that conflict with existing verified evidence are masked rather than trained as mathematical failures. The same compiled-success rule applies to positive propositions and their exact outer negations.

C.5 Decoding and Sampling Parameters

For candidate-solution generation, Qwen3-8B (Yang et al., 2025) runs in thinking mode with the recommended sampling parameters: temperature 0.6, top-p = 0.95, top-k = 20, and minimum probability 0. The maximum output length is 32,768 tokens. We sample 16 solutions per problem on AIME24, AIME25, and BRUMO25 and five per problem on MATH-500.

For Stage-A/B autoformalization and the trajectories used to construct the prover-training data, Qwen3.8-27B uses thinking with low reasoning effort, temperature 1.0, top-p = 0.95, top-k = 20, minimum probability 0, and a maximum completion length of 20,480 tokens. GRPO sampling instead uses 16 rollouts per prompt with temperature 1.0, top-p = 1.0, and the same 20,480-token response limit. During evaluation, Stage-A and Stage-B internal semantic judges use deterministic decoding with maximum completion lengths of 8,192 and 16,384 tokens, respectively. The independent full-proof semantic-faithfulness judge uses temperature 0.6, top-p = 0.95, and at most 16,384 output tokens. Each Stage-C prover attempt permits at most four `lean_compile` submissions.

C.6 Lean and Mathlib Environment

All formal artifacts are checked with Lean 4.32.0-rc1 in a pinned Mathlib environment. The Mathlib revision is `6cc0969466dc9a1043de33485f78471e3aaa1bd8`, and LeanArchitect is pinned at `d9013cc08bd2b5483e837368dfa4cc7ead92a5c2`. Compilation is executed through Kimina Lean Server. Mathlib search and isolated scratch compilation may assist proof construction, but only a complete declaration accepted by Lean contributes verified evidence; declarations containing `sorry` or `admit` do not count as proof success.

D Preliminary Semantic-Fidelity Study

D.1 Paired Sampling Protocol

Qwen3-8B generates 16 candidate solutions per problem on AIME24, AIME25, and BRUMO25 and five per problem on MATH-500. We then sample 1,000 problem–solution pairs, stratified equally across the four sources. Each sampled pair is processed under both semantic-interface configurations with the same Qwen3.8-27B generation settings. The generation parameters for both models are reported in Appendix C.5. Neither configuration performs proposition proving, so the comparison isolates semantic representation and formalization before proof search.

D.2 Compared Semantic Interfaces

Holistic-DAG directly decomposes the problem and candidate solution into a proposition DAG and formalizes the graph. Uni-DAG first constructs and compiles problem-level semantic objects, solution-induced modeling choices, and a distinguished target object; the resulting frozen identifiers are then shared by DAG construction and Lean formalization. Both configurations receive the same original problem and complete candidate solution.

D.3 Verdict Aggregation

DeepSeek-V4-Flash-0731 compares the original problem and complete natural-language solution with the generated DAG and Lean declarations. Decoding uses temperature 1.0, top-$p = 1.0$, seed 20260923, and a maximum completion length of 4,096 tokens. The judge assigns F1–F5 independently, so a single sample can contribute to multiple error counts. A sample contributes to the *Faithful* bar only when no category is assigned and the overall verdict is faithful. Figure 1 therefore reports sample counts rather than mutually exclusive percentages. The complete system prompt appears in Appendix H.1.

E Proof Autoformalization Evaluation

E.1 Benchmarks and Input Construction

We evaluate on all 184 ProofFlowBench examples and all 244 miniF2F examples used by the full-proof protocol of Liu et al. (2026). Each record supplies the original mathematical problem and its complete natural-language proof to NF-Bridge. The semantic-faithfulness evaluator receives these inputs together with the generated proposition DAG and Lean declarations.

E.2 Syntactic and Semantic Metrics

SC is positive exactly when the root `P_final` has label `proved`, verified direction `positive`, and a verified Lean declaration. SF is a binary judgment requiring semantic faithfulness of both the final theorem and every substantive proof step relative to the supplied natural-language proof. The joint metric is positive only when both SC and SF are positive. All four reported runs have fully resolved SC and SF judgments for every benchmark example.

E.3 Runtime Accounting

The 27B runs were executed on eight NVIDIA A800 GPUs; the measured end-to-end wall-clock times were 6:06:22 for ProofFlowBench and 4:59:09 for miniF2F. The 9B runs were executed on two NVIDIA A800 GPUs and took 7:15:44 and 6:01:42, respectively.

For a run using g GPUs, measured wall-clock duration T, and N benchmark examples, we report the single-A800-equivalent throughput time gT/N. This produces 955.74 and 588.49 seconds per problem for the 27B ProofFlowBench and miniF2F runs, and 284.17 and 177.89 seconds per problem for the corresponding 9B runs. These values normalize aggregate GPU usage and should not be interpreted as directly measured single-example latency.

E.4 ProofFlowBench and miniF2F Results

The exact counts underlying Table 3 are shown in Table 7. Percentages in the main table use the full benchmark size as denominator.

Table 7: Exact counts for the Full-Proof Autoformalization evaluation.

Benchmark	Method	SC	SF	SC ∧ SF
ProofFlowBench	NF-Bridge-27B	86/184	141/184	75/184
ProofFlowBench	NF-Bridge-9B	78/184	138/184	66/184
miniF2F	NF-Bridge-27B	170/244	212/244	158/244
miniF2F	NF-Bridge-9B	178/244	214/244	165/244

E.5 Robustness under Global Perturbations

Proof-style abbreviations. Orig denotes the original benchmark proof. G-FF and G-Step denote the Gemini-generated free-form and step-by-step rewrites, respectively; Q-FF and Q-Step denote the corresponding Qwen-generated rewrites. Each style contains the same 244 miniF2F problems and 500 MATH-500 problems.

Evaluation input construction. The released RobustPABench semantic-correctness evaluator extracts only the final `theorem`, `lemma`, or `example` declaration when constructing the inputs for StmtSC and ProofSC. In NF-Bridge, however, the substantive derivation is distributed across dependency-structured lemmas, and the final theorem may primarily compose previously verified results. We therefore collect all lemma and theorem declarations participating in the final derivation and order them according to the proposition DAG. For StmtSC, we retain their signatures while replacing every proof body with `by sorry`, yielding a statement-only artifact. For ProofSC, we retain the same declarations together with their complete verified proof bodies. The statement-only artifact is used exclusively as input to the StmtSC semantic judge and is never used to compute TC; TC is evaluated on the actual generated Lean artifact. The definition of FullyCorrect remains TC ∧ StmtSC ∧ ProofSC.

E.6 Additional RobustPABench Results

Table 8: Component metrics (%) underlying the RobustPABench global-perturbation results.

Metric	Model	miniF2F					MATH-500				
		Orig	G-FF	G-Step	Q-FF	Q-Step	Orig	G-FF	G-Step	Q-FF	Q-Step
TC	Kimina-Prover-RL-1.7B	38.5	32.4	39.3	40.2	45.9	39.8	38.8	38.0	39.2	37.4
	Kimina-Prover-Distill-8B	44.7	39.3	40.6	43.4	46.7	49.2	48.2	44.6	46.8	50.0
	DeepSeek-Prover-V2-7B	5.7	17.2	19.3	28.7	27.5	25.8	25.4	22.0	26.0	20.8
	Goedel-Prover-V2-8B	9.8	11.9	12.3	14.8	15.2	16.8	16.4	14.6	17.8	18.4
	Gemini-3.1-Pro	46.3	50.4	49.2	47.1	53.7	58.0	56.8	57.8	55.6	54.4
	ProofBridge	37.3	36.5	37.3	39.3	38.9	33.2	35.0	35.2	31.6	33.6
	ProofFlow	26.6	19.7	26.6	25.4	26.2	27.4	21.0	23.2	22.6	32.4
	NF-Bridge	69.7	68.9	70.1	75.4	78.3	73.8	75.6	74.4	74.6	75.0
StmtSC	Kimina-Prover-RL-1.7B	38.1	39.3	42.6	43.9	41.4	29.2	32.4	28.8	31.6	28.0
	Kimina-Prover-Distill-8B	74.2	63.1	65.2	66.8	69.7	58.4	56.8	59.2	60.0	58.8
	DeepSeek-Prover-V2-7B	57.4	63.1	66.0	64.3	58.2	50.8	53.0	49.6	50.6	49.6
	Goedel-Prover-V2-8B	50.8	46.7	51.2	52.9	51.2	42.2	38.6	39.8	40.8	40.4
	Gemini-3.1-Pro	82.0	79.5	79.9	78.3	81.1	81.4	81.8	82.0	82.8	81.2
	ProofBridge	71.3	69.7	69.7	71.3	71.3	50.8	51.6	48.8	50.4	49.0
	ProofFlow	42.2	56.6	39.3	42.2	40.6	47.6	58.4	60.6	58.2	45.8
	NF-Bridge	98.4	98.8	98.4	97.5	99.6	99.8	100.0	99.4	99.4	100.0
ProofSC	Kimina-Prover-RL-1.7B	20.9	23.0	20.1	23.0	17.6	16.8	15.8	18.0	19.2	19.2
	Kimina-Prover-Distill-8B	41.4	36.9	38.5	38.5	40.2	43.6	39.4	38.2	40.2	39.0
	DeepSeek-Prover-V2-7B	40.2	41.4	38.5	40.2	43.0	43.2	41.4	43.6	43.6	43.4
	Goedel-Prover-V2-8B	27.0	27.9	34.0	34.0	33.6	29.4	27.4	29.0	28.4	25.8
	Gemini-3.1-Pro	36.5	38.1	37.7	39.8	36.1	41.6	40.4	41.8	41.2	39.8
	ProofBridge	38.1	43.9	41.0	37.3	43.0	37.0	41.8	43.2	39.6	45.2
	ProofFlow	31.1	36.5	33.6	30.7	36.1	34.2	34.8	38.0	39.8	33.8
	NF-Bridge	68.0	67.2	66.4	72.5	75.0	72.4	74.2	74.2	74.4	74.2

Notes. TC denotes Type Correctness and records whether the generated Lean formalization compiles; StmtSC and ProofSC evaluate semantic consistency of the formal statement and formal derivation, respectively. All values are percentages. Each style contains 244 miniF2F examples and 500 MATH-500 examples. FullyCorrect in Table 1 is TC ∧ StmtSC ∧ ProofSC.

F Test-Time Candidate Selection

F.1 Candidate-Pool Construction

All selectors operate on the same stored candidate solutions generated by Qwen3-8B with a common generation template. The template is identical across datasets; only the problem text is substituted. AIME24, AIME25, and BRUMO25 each contain 30 problems with 16 rollouts per problem, while MATH-500 contains 500 problems with five rollouts per problem. The resulting fixed pool contains $30 \times 16 + 30 \times 16 + 500 \times 5 + 30 \times 16 = 3{,}940$ candidate solutions. No selector regenerates or modifies a candidate response.

F.2 Standard and Learned Reward-Model Baselines

We compare against pass@1, uniformly random selection with seed 42, majority voting, and an oracle reporting pass@N within the available pool. The learned baselines are Qwen2.5-Math-PRM-7B (Zhang et al., 2025), AceMath-7B-RM (Liu et al., 2024b), Skywork-Reward-Llama-3.1-8B-v0.2 (Liu et al., 2024a), ArmoRM-Llama3-8B-v0.1 (Wang et al., 2024a), and RLHFlow-Llama3.1-8B-PRM-DeepSeek (Wang et al., 2024b). Every baseline receives the same candidate identities and answer strings as NF-Bridge.

F.3 Verifier-Grounded Selection and Direction-Rejection Signals

Following Section 3.3, let V_i be the nodes of $\mathcal{G}_{F,i}$ and let $\sigma_i(v)$ denote the Lean-certified state of node v. A candidate containing a `disproved` node receives score zero; otherwise its score is the fraction of proved nodes:

$$s_i = \begin{cases} 0, & \exists v \in V_i : \sigma_i(v) = \texttt{disproved}, \\ \dfrac{1}{|V_i|} \displaystyle\sum_{v \in V_i} \mathbf{1}[\sigma_i(v) = \texttt{proved}], & \text{otherwise.} \end{cases}$$

Candidates without a verified refutation are ranked by s_i, while candidates containing compiled counterevidence are placed last; remaining ties follow the original rollout order.

For each unresolved node, the reject-augmented selector considers three independent prover attempts for each verification direction. Reject-3 produces a direction-rejection signal only when all three attempts reject the same verification direction. For example, unanimous rejection of the negative verification direction targeting $\neg Q_{i,v}$ provides weak positive evidence for $Q_{i,v}$ during ranking; unanimous rejection of the positive verification direction targeting $Q_{i,v}$ provides weak negative evidence. Incomplete or conflicting rejection produces no inference. These inferred outcomes are used only by the ranking procedure: $\sigma_i(v)$ remains `unresolved`, since only a compiled proof of $Q_{i,v}$ or $\neg Q_{i,v}$ can assign `proved` or `disproved`.

F.4 Per-Dataset Evaluation Protocol

Answer accuracy is computed over 30 AIME24 problems, 30 AIME25 problems, 500 MATH-500 problems, and 30 BRUMO25 problems. The denominator is therefore the number of problems rather than the number of rollouts. Oracle/pass@N is positive when at least one candidate in the fixed pool is correct; every other method selects exactly one candidate per problem.

F.5 Verifier-Signal Ablation

Table 9 reports the per-benchmark results underlying Table 6. "Proof coverage" corresponds to ranking by the proved-to-total ratio alone; "+ Disproof" adds compiled counterevidence; and "+ Reject-3" additionally incorporates the direction-rejection consensus across three prover attempts. Overall accuracy pools all 590 problems and is not the unweighted mean of the four benchmark percentages.

Table 9: Per-benchmark candidate-selection accuracy (%) for the verifier-signal ablation.

Prover	Ranking signal	AIME24	AIME25	MATH-500	BRUMO25	Overall
9B	Proof coverage	80.00	70.00	87.00	63.33	84.58
9B	+ Compiled disproof	86.67	70.00	87.00	63.33	84.92
9B	+ Reject-3	86.67	80.00	87.00	60.00	**85.25**
27B	Proof coverage	83.33	73.33	86.60	63.33	84.58
27B	+ Compiled disproof	86.67	73.33	86.60	60.00	84.58
27B	+ Reject-3	83.33	76.67	87.00	60.00	**84.92**

G Isolated Prover Evaluation

G.1 Task Construction

The isolated benchmark is constructed from frozen Stage-A/B artifacts over the four candidate-selection datasets. It contains 768 node-level verification tasks: 256 positive-proof targets, 256 exact-negation targets, and 256 direction-rejection targets. The set is source-balanced, with 64 proof and 64 rejection tasks sampled from each dataset; exact-negation tasks are allocated with a capacity-aware seeded procedure because the available disproved nodes are not uniform across sources. All 512 proof/disproof targets were replayed successfully in the pinned Lean environment before evaluation. The benchmark contains 442 records, 182 original problems, and 512 distinct proposition nodes.

G.2 Models and Inference Budget

We compare the Qwen3.5-9B GRPO step-63 prover, Qwen3.8-27B, and Goedel-Prover-V2-8B. Each model is evaluated on the identical 768 tasks using one NVIDIA A800 GPU and a maximum output length of 20,480 tokens. The Qwen provers use the Stage-C tool-using protocol, whereas Goedel-Prover-V2-8B uses the text-only prompt reproduced in Appendix H.2; the marker `REJECT_DIRECTION` denotes an explicit direction rejection. We exclude the separate 8,192-token Goedel run from the comparison.

G.3 Metrics and Runtime Accounting

A task succeeds only when the model produces the expected outcome class: a compiled proof for positive targets, a compiled proof of the exact outer negation for negative targets, or an explicit rejection for rejection targets. We report overall success and the three target-specific success rates. End-to-end wall-clock time covers the complete 768-task evaluation. The 9B prover succeeds on 645/768 tasks (83.98%) in 2:39:37; Qwen3.8-27B succeeds on 656/768 (85.42%) in 7:18:25; and Goedel-Prover-V2-8B succeeds on 153/768 (19.92%) in 1:06:25. The corresponding positive, exact-negation, and rejection counts are 218/256, 174/256, and 253/256 for the 9B prover; 218/256, 182/256, and 256/256 for Qwen3.8-27B; and 111/256, 42/256, and 0/256 for Goedel-Prover-V2-8B.

H Prompts and Semantic Rubrics

We reproduce the system-role prompts used by NF-Bridge and its semantic evaluators. Record-specific user messages are assembled mechanically from the problem, candidate solution, and frozen artifacts and are not reproduced.

H.1 Preliminary F1–F5 Evaluation Rubric

```
You are an expert in mathematical proof semantics and Lean 4 autoformalization.

Your task is to determine whether the supplied formalization faithfully preserves
the mathematical semantics and proof process of the original Problem and Complete
COT. Focus on proof-process semantic fidelity, not merely whether:
```

```
- the final answer is correct;
- the Lean code compiles; or
- the final proposition is true.

Judge only the supplied mathematical artifacts. Do not assume access to another
version and do not invent content that is absent. A missing stage is not
automatically an error. However, if a substantive mathematical commitment required
by the Problem or COT is not represented in the supplied objects, propositions,
dependency graph, or final target, treat that absence as evidence for the applicable
error category.

Artifact contract for this evaluation:

- The supplied Lean is a semantic declaration skeleton, not a completed executable
  proof.
- Intermediate proof nodes may be assembled as `lemma` declarations and the final
  node as a `theorem` declaration. Dependency hypotheses in those signatures encode
  graph edges.
- `by sorry` is an explicitly permitted structural placeholder. It carries no
  negative evidence and must never trigger F1, F2, F3, F4, F5, or an unfaithful
  overall verdict.
- Judge the mathematical content of object definitions, proposition and
  lemma/theorem signatures, dependency binders, DAG dependencies, and supplied DAG
  proof descriptions. Do not require tactic code, proof terms, theorem completion,
  or compilation without `sorry`.
- Never cite "only declared", "unproved", "no theorem implementation", "no proof",
  or the presence of `sorry` as an error. A substantive node or dependency may still
  be erroneous when its mathematical content is absent, mistranslated, or connected
  to the wrong premises.

A single record may contain errors from multiple categories. Judge the following
five categories independently.

F1: Mathematical semantic mismatch

The mathematical objects, operations, relations, types, or intermediate
propositions in the formalization do not match their actual meaning in the Problem
or COT. Relevant subcategories include:

- object_definition: a variable, function, set, sequence, geometric object, or
  other object is defined incorrectly;
- object_identity: two distinct mathematical objects are confused, such as an
  index and a value;
- operation_semantics: an operation changes meaning, such as a norm, division,
  exponent, square root, or similar operation;
- proposition_content: an intermediate proposition mistranslates a number,
  symbol, side of an equality, relation, or other mathematical content;
- type_semantics: the Lean type changes the original mathematical meaning.

Mathematically equivalent representations are not F1 errors.

F2: Constraint or admissible-case mismatch

The formalization changes the mathematical cases allowed by the Problem or COT by
omitting a condition or introducing an unsupported condition. Relevant
subcategories include:

- constraint_loss: a positivity, integrality, nonzero, distinctness, range, or
  domain condition is lost;
- constraint_addition: an assumption absent from the Problem and COT is added;
- domain_change: a variable's domain is incorrectly widened or narrowed;
- quantifier_change: a universal, existential, or fixed-instance quantifier is
  changed without justification;
- scope_change: the scope in which a condition applies is changed.

Do not decide F2 merely because a proposition appears stronger or weaker. The
question is whether the original admissible cases are preserved.

F3: Proof-process or dependency loss

A mathematically substantive derivation, proof route, or dependency from the COT is
missing or changed. Relevant subcategories include:

- key_step_loss: a key elimination, construction, case split, candidate filtering
  step, boundary analysis, uniqueness argument, or similar step is absent;
- route_substitution: the main COT proof route is replaced by a route that changes
  a substantive mathematical commitment, admissible case, dependency, or target
  justification. A different but mathematically sound route is not an error;
- dependency_distortion: a node uses the wrong premise, a dependency direction is
  wrong, or an essential dependency disappears;
- candidate_verification_only: the COT derives a candidate and then verifies it,
  but the formalization signatures and dependencies only verify an already known
  candidate and omit the substantive derivation, completeness, uniqueness, or
```

```
  optimality obligation required by the original task.

Do not automatically mark F3 for arithmetic simplification, standard definition
unfolding, simple algebraic manipulation, or merging mechanical steps. The
criterion is whether a non-mechanical mathematical commitment from the COT has
disappeared or changed. A proposition signature, lemma/theorem signature,
dependency binder, or specific DAG proof description counts as representation of a
step. Do not demand that the represented step also contain an implemented proof.

F4: Final target mismatch

The final formalized target does not correctly express the object or task that the
Problem and COT actually require. Relevant subcategories include:

- wrong_target_object: the formalization proves a statement about the wrong object;
- object_role_confusion: for example, a sequence index is treated as a sequence
  value;
- wrong_task_type: evaluation becomes existence, optimization becomes mere
  feasibility, or another task type changes;
- target_binding_loss: the final proposition is no longer connected to the
  original target object;
- incomplete_target: only part of the task is proved, such as feasibility without
  required minimality or uniqueness.

If an upstream object-definition error alone makes the target wrong, prefer F1.
Mark F4 as well only when there is an independent target-binding problem.

F5: Proof-obligation bypass

The formalization writes in the answer early, uses a circular definition, replaces
the goal with a tautology, or uses a vacuous construction so that the mathematical
relationship no longer has to be derived. Relevant subcategories include:

- answer_presupposition: the unknown or a key intermediate result is defined as its
  final value;
- hardcoded_target: the final answer is directly encoded;
- tautological_replacement: a substantive target is replaced by True, x = x,
  15 = 15, or another tautology;
- vacuous_context: a contradictory assumption, empty domain, or unreachable
  antecedent makes the proposition automatically true;
- circular_reasoning: the desired conclusion is used in advance as a definition or
  assumption.

The mere appearance of a constant, candidate, reflexive equality, empty list, or
solved expression is not by itself F5. Mark F5 only when it actually bypasses a
mathematical obligation required by the Problem or COT at the level of the
statements and dependencies. `by sorry` never constitutes such a bypass.

Defining an auxiliary object by a useful formula, recurrence, coordinate choice, or
candidate value is allowed when the lemma/theorem signatures and dependencies still
connect that object to the original hypotheses and required target. Mark F5 only
when the definition makes the target tautological or presupposes the answer while
removing the need for that connection. Merely checking a candidate is F5 only when
the original derivation, completeness, uniqueness, or optimality obligation is
structurally bypassed, not simply because a candidate appears in a definition.

Use the narrowest category supported by the evidence:

- Missing representation of a substantive derivation or dependency is F3.
- A wrong object, operation, proposition, or type is F1.
- An independently wrong or incomplete final task binding is F4.
- F5 additionally requires a clear tautological, circular, vacuous, or
  answer-presupposing bypass in the mathematical statements themselves.
- Do not assign F3, F4, and F5 together solely because the same candidate or answer
  constant appears in several artifacts.

Audit the record in this order:

1. Reconstruct the original target, unknowns, admissible cases, and key
   non-mechanical proof route from the Problem and Complete COT.
2. Map every supplied mathematical object, operation, relation, and type back to
   that reconstruction.
3. Compare constraints, domains, quantifiers, and scopes.
4. Compare the substantive propositions and their dependencies with the COT route.
   Treat lemma/theorem signatures, dependency hypotheses, and DAG proof descriptions
   as evidence that a step is represented.
5. Verify that the final target remains bound to the original target object and
   preserves the original task type.
6. Check for answer presupposition, hardcoding, tautology, vacuity, or circularity.
7. Decide the overall verdict, then assign the five category labels.

Apply these rules:
```

```
1. A record may belong to multiple F categories.
2. Do not count the same root cause repeatedly merely because it appears in
   multiple stages.
3. Lean compilation success does not imply semantic fidelity.
4. A correct final answer does not imply proof-process fidelity.
5. If the COT itself is wrong but the formalization faithfully preserves that
   error, do not mark it as a formalization error.
6. Equivalent mathematical rewrites, reasonable auxiliary objects, and merged
   mechanical steps are not errors.
7. Do not penalize a record merely because it exposes more objects, intermediate
   propositions, or dependencies than another possible formalization. Additional
   structure is an error only when its own mathematical content is wrong or it
   changes the represented task.
8. Do not require every sentence or computational detail in the COT to become a
   separate node. Preserve non-mechanical commitments, not textual granularity.
9. Use overall_verdict "unfaithful" if and only if at least one F category is 1.
10. Use overall_verdict "faithful" only when all F categories are 0 and the supplied
   evidence is sufficient to establish fidelity.
11. Use overall_verdict "uncertain" only when all F categories are 0 but the supplied
   evidence is genuinely insufficient to establish either fidelity or a concrete
   error. Do not use "uncertain" merely because the analysis is difficult.
12. Each reason must briefly identify the most important concrete evidence. The
    overall_reason must explain the overall verdict rather than repeat a generic
    statement.

Output only a valid JSON object in exactly this shape, with no Markdown fence and
no additional text:

{
  "overall_verdict": "faithful",
  "overall_reason": "Brief evidence-based justification for the overall verdict.",
  "categories": [
    {"F1": 0, "reason": ""},
    {"F2": 0, "reason": ""},
    {"F3": 0, "reason": ""},
    {"F4": 0, "reason": ""},
    {"F5": 0, "reason": ""}
  ]
}

Use 1 only when the category is confirmed and 0 when it is not confirmed in the
provided input. Replace the example verdict and all reason strings with the actual
judgment.
```

H.2 NF-Bridge Full-Proof Autoformalization Prompts

This subsection reproduces verbatim the system prompts used by the Full-Proof Autoformalization experiments. The prompt files are included directly in the manuscript source so that the displayed text and the archived experimental prompts remain byte-identical. Final-alignment post-processing was disabled and its unused prompt is therefore not included.

H.2.1 Stage A: solution-induced normalization

```
You extract only COT-introduced changes to the mathematical model.

The input contains an original Problem and a Complete COT. Answer exactly this
question: relative to the Problem, does the COT replace, specialize, normalize,
or explicitly reparameterize the mathematical model in order to reason about it?

Report only genuine model changes introduced by the COT, such as:
- a WLOG placement or orientation;
- a coordinate origin or axis alignment absent from the Problem;
- a scale normalization;
- a symmetry reduction used to replace the original model;
- an explicit parameterization that specializes the original objects.

When the COT defines a length or a length-ratio model using a numerical or
symbolic coefficient, the normalization paragraph may state that the
coefficient is strictly positive by default, unless the Problem or COT gives a
different domain. Include this only when that length or ratio definition is
part of the working model; do not add positivity assumptions to unrelated
numbers or parameters.

The test is: if the choice were removed, would the later Lean theorem describe a
different working model from the one actually used by the COT? If yes, report it.

Do not report:
- the final claimed answer;
```

```
- any derived equation, calculation, result, property, or proof step;
- an erroneous COT step recast as an assumption;
- standard mathematical background or implicit Problem semantics;
- base-10 notation, digit representations, or ordinary arithmetic conventions;
- Lean implementation choices such as matrices, finite types, indices, sets, or
  functions;
- direct restatements or interpretations of the Problem;
- enumeration reductions such as selecting representative cases after proving a
  symmetry or commutativity fact.

If the COT uses a normalized model while leaving equivalence with the original
model implicit, say that the later formalization may work directly with the
normalized model and that the equivalence is outside the current proof scope.
Do not claim that the equivalence has been formally proved.

Return plain-English prose with no heading, JSON, YAML, Markdown fence, or Lean
code. Multiple paragraphs are allowed when they make distinct model changes
clear. Do not include the final answer. If the COT introduces no additional
model assumption or normalization, output exactly:

The COT introduces no additional model assumptions.
```

H.2.2 Stage A: semantic-object construction

```
You are a faithful Lean 4 semantic object formalizer, not a problem solver or
truth judge.

The input contains only an original Problem and an optional natural-language
normalization handoff. You do not have access to the Complete COT or any claimed
answer.

Produce a complete Lean file that defines the mathematical objects needed to
state the Problem and identifies the one object requested by the Problem. Do not
state or prove the requested result.

Semantic rules:
- The Problem determines the original objects, quantification, constraints, and
  requested quantity.
- A nonempty normalization handoff may specify the working normalized model.
  Define that model directly; equivalence to the original model is outside the
  current proof scope.
- Do not invent additional normalization, assumptions, conclusions, or COT
  facts.
- Keep arbitrary Problem inputs as parameters of definitions. Preserve genuine
  conditions as named predicates or domains when needed.
- For every parameterized object, identify all well-formedness restrictions on
  its parameters, including restrictions implicit in the mathematical kind of
  object rather than literally written as an inequality. Dimensions, lengths,
  ratios of positive quantities, cardinalities, indices, probabilities, and
  similar parameters have restricted domains. Do not silently enlarge such a
  domain merely because a wider Lean type is convenient.
- Encode parameter restrictions in the object interface when practical, using
  a constrained subtype, a structure carrying validity fields, a named domain
  predicate, or an explicitly restricted set. If a construction is totalized
  over a wider Lean type, also define the intended admissibility predicate or
  domain so later propositions can retain the original restriction.
- Restrictions describe when an input object is well formed; they must not
  encode a COT-derived fact or the requested answer.
- Definitions must express object meaning, not encode a derived claim or a final
  answer.
- Definitions of exhaustive sets, outcome spaces, valid constructions, and
  requested quantities must be generated compositionally from the Problem's
  rules. Never replace such an object with the examples visible in the Problem,
  a guessed finite list, or a manually inserted numerator or denominator.
- Choose types by semantic behavior, not convenience. In particular, preserve
  multiplicity, order, and nonuniform outcome weights whenever the Problem does;
  a set of labels is not a bag of physical objects with possible duplicates.

Object/fact boundary:
- Stage 1 defines semantic objects only; it does not establish facts about them.
- Defining a predicate, relation, constrained set, subtype, or construction is
  allowed when it specifies what that object means.
- Do not assert or prove that a particular object has a property merely because
  that property will be useful later.
- Do not introduce auxiliary mathematical facts merely to make another
  definition type-check or to enable later computation.
- Use standard Lean/mathlib constructions that directly match the mathematical
  meaning of the Problem.
- Prefer a representation that is well-typed without requiring additional
  derived facts to be proved at Stage 1.
```

```
Proof-friendly representation guidance:
- When several faithful representations are available, prefer standard,
  reducible, `simp`-friendly mathlib data over hand-built encodings. For finite
  indexed data, prefer established `Fin`, `Fintype`, `Finset`, product, sum,
  subtype, and `Equiv` interfaces when they preserve the Problem semantics.
- Prefer deriving or exposing the ordinary computational structure that later
  proofs will need, such as `DecidableEq` and `Fintype`, instead of constructing
  bespoke enumeration machinery whose API is difficult to use.
- Represent genuinely finite collections in a form compatible with finite
  cardinality, sums, images, filters, and bijections. Do not use an unrestricted
  `Set` equality as the only interface when the Problem intrinsically supplies
  finite enumerated data and a faithful `Finset`/`Fintype` interface is
  available.
- Keep definitions transparent and compositional enough for `simp`, `rw`,
  `norm_num`, `fin_cases`, `ext`, and standard cardinality or sum lemmas to
  expose their meaning. Avoid unnecessary wrappers, custom instances, and
  opaque layers that add no semantic information.
- These preferences never authorize changing multiplicity, order, weights,
  admissible domains, geometric meaning, or any other mathematical content.

Declaration contract:
- Begin with `import Mathlib`.
- You may use top-level `def`, `noncomputable def`, `structure`, and `inductive`
  declarations.
- Do not use `abbrev`, `lemma`, `theorem`, `axiom`, `opaque`, `constant`,
  `class`, `instance`, `example`, or top-level `variable` declarations.
- Mark exactly one declared `def` using the exact comment
  `-- TARGET_OBJECT: identifier`, where `identifier` is the Lean ID of the
  quantity or object requested by the Problem.
- The target definition must capture the requested object's meaning but must not
  hard-code the COT's or dataset's answer.

For Euclidean geometry, remember that `ℝ × ℝ` has the sup metric `dist`; use
`EuclideanSpace ℝ (Fin 2)`, or keep `ℝ × ℝ` and define Euclidean distance
explicitly as `Real.sqrt ((p.1 - q.1)^2 + (p.2 - q.2)^2)`.

Compilation discipline:
- Each round is either exactly one `lean_compile` call or one discovery-only
  batch; never mix compile and discovery tools in the same round.
- On the first turn, either call `lean_compile` with the complete Lean file or
  make one discovery-only batch of `mathlib_search` and/or `lean_scratch`.
- After a failed compile, call `lean_compile`, `mathlib_search`, or
  `lean_scratch`.
- A discovery round may contain external `mathlib_search` calls and/or isolated
  `lean_scratch` compilations. Use scratch to test object representations,
  exact Mathlib signatures, types, coercions, instances, or local elaboration.
  Temporary scratch commands and declarations do not relax the Stage-1
  declaration contract above and never validate the Stage-1 file. The runner
  removes imports, and scratch declarations do not persist into another call
  or the later `lean_compile` submission.
- Every discovery round must be followed immediately by `lean_compile`.
- Compiler feedback may repair syntax, types, coercions, or representations, but
  must not change the mathematical model merely to compile.
- A compile-and-semantic-judge success ends the stage. Do not return an ordinary
  answer.
- The runner also applies an independent semantic judge to the exact submitted
  file. A submission succeeds only when both compilation and that judge pass.
  On a later turn, repair every concrete compiler error and semantic-judge defect
  reported for the previous candidate; do not argue around the feedback by
  relying on declaration names or comments.
```

H.2.3 Stage A: semantic-object validation

```
# Stage 1 Semantic Object Judge

You are a problem-grounded semantic judge, not a Lean compiler, prover, or
mathematical answer generator. Decide whether the submitted Stage-1 Lean file
faithfully defines the objects required by the original Problem and the supplied
normalization handoff.

Inspect the actual types and right-hand sides of every declaration. Identifiers,
comments, and docstrings are not evidence that a definition has its claimed
meaning. Reject a definition whose name is faithful but whose implementation is
not.

Check all of the following:

- every declared object and operation has the meaning required by the Problem;
- the representation preserves material structure such as multiplicity, order,
  probability weights and sample spaces, geometric metric, and admissible
  parameter domains;
```

```
- the numerator and denominator of a probability refer to the same weighted
  outcome space;
- the TARGET_OBJECT denotes the requested object compositionally rather than a
  guessed, partial, example-only, or hard-coded result;
- a finite enumeration claimed to be exhaustive is generated from the Problem's
  rules, not merely listed without justification;
- all basic objects needed to state the requested quantity are present, while
  proof facts, COT conclusions, and the requested answer are absent;
- the file respects the normalization when one is supplied, without silently
  adding a different model.

Compilation is checked independently and says nothing about semantic fidelity.
Do not accept a candidate merely because Lean can evaluate or compile it. Do not
solve the Problem or compare against a claimed COT answer.

Return exactly one bare JSON object:

{"accepted": 0, "reason": "specific actionable semantic defect"}

Use integer 0 or 1. The reason must be non-empty. When accepted, briefly state
why the representation and TARGET_OBJECT are adequate. Output no Markdown fence
or additional prose.
```

H.2.4 Stage B: semantic proposition DAG construction

```
# Stage 2 — Semantic DAG and Object Registry

You are a faithful semantic analyst. Read the original Problem, Complete COT,
and frozen compiled Stage-1 Lean object file. Return exactly one bare JSON
object, with no Markdown fence or prose outside it:

{
  "auxiliary_objects": [
    {
      "id": "aux_name",
      "description": "detailed natural-language definition and COT source",
      "mathematical_type": "natural-language mathematical type",
      "parameters": ["input_name"],
      "used_by": ["P1", "P2"]
    }
  ],
  "propositions_in_proof_order": [
    {
      "id": "P1",
      "prop": "complete natural-language proposition",
      "dependencies": [],
      "stage1_objects": ["exact_stage1_declaration_id"],
      "auxiliary_objects": ["aux_name"],
      "proof": "detailed Lean-oriented natural-language proof plan"
    },
    {
      "id": "P_final",
      "prop": "faithful final claim",
      "dependencies": ["P1"],
      "stage1_objects": ["exact_stage1_declaration_id"],
      "auxiliary_objects": [],
      "proof": "how P1 is used to derive the final claim in Lean"
    }
  ]
}

Use exactly these keys. IDs are unique. Dependencies reference only earlier
propositions. `P_final` occurs exactly once and last. Each auxiliary `used_by`
list must exactly equal, in proposition order, the nodes that list it under
`auxiliary_objects`.

## Reuse frozen and auxiliary objects

Reuse the exact identifiers already declared by Stage 1. Never create an
auxiliary object that duplicates, renames, specializes, or semantically overlaps
a Stage-1 object. List the Stage-1 and auxiliary objects semantically used by
each proposition. A proposition may refer through a wrapper object, so the
listed lower-level objects need not all occur textually in its eventual Lean
body. Object references are not proposition dependencies.

Prefer no auxiliary objects. Usually inline data used by only one proposition.
Create `aux_*` only for a stable data identity or construction introduced by
the COT and genuinely reused by multiple later proposition statements. An
auxiliary may be a number, point, set, function, sequence, or construction. It
must not be an assumption, fact, Prop, hidden witness, derived claim, right-hand
side of a pending equality, or claimed final answer. Its description must give
```

```
its exact construction, type, parameters, source in the COT, and meaning.

## Proposition fidelity

Do not judge truth. Represent the COT rather than solving or correcting it.
Preserve an incorrect, unsupported, incomplete, inconsistent, or contradictory
COT claim so later proof/disproof can detect it. Do not change a proposition
because you believe a different mathematical statement is true.

Internally account for every substantive COT claim in source order before
building the registry. A claim may be omitted only when it merely repeats an
original Problem assumption, accepted normalization, or object definition
already frozen in Stage 1 and contains no additional COT inference. Do not omit
a derived claim merely because it is difficult to formalize. Do not merge
distinct COT transitions when that would erase an intermediate assertion,
logical step, error, or dependency.

## Lean-like natural-language Props

Write every `prop` as a complete, self-contained, Lean-like natural-language
proposition that a later formalizer can translate into exactly one Lean `Prop`
without inventing mathematical content or choosing among interpretations.

- A `prop` denotes a fact, assertion, property, or relation. It never denotes a
  number, set, point, sequence, function, witness, domain, construction,
  configuration, or instruction to define an object.
- State every binder explicitly, including its intended mathematical type,
  quantifier, admissible domain, hypotheses, and scope. Make conjunction,
  implication, equivalence, negation, existence, uniqueness, membership,
  equality, and inequality logically explicit rather than relying on prose
  implication or surrounding context.
- Use exact frozen Stage-1 IDs and registered auxiliary IDs when they identify
  the intended objects. Make applications and relationships explicit enough to
  determine which parameters and arguments are used.
- Preserve every concrete value, coordinate, endpoint, index, bound, formula,
  membership condition, order relation, equality, inequality, uniqueness or
  extremality condition, and case condition asserted by the represented COT
  claim.
- If a one-use COT-local object is needed, identify it inside the same `prop`
  with an exact expanded expression, a claim-local let-style description, or a
  faithful explicit quantifier. Do not refer to “the object from P1”, “the
  previous witness”, “that configuration”, or other cross-proposition anaphora.
- Do not replace a specific COT object or construction with an arbitrary or
  weaker existential witness merely because the latter is easier to formalize.
- A dependency transfers only the truth of the dependency proposition. It does
  not expose existential witnesses, local bindings, or data objects occurring
  inside that proposition. Restate any required object identity locally or use
  a Stage-1/registered auxiliary object.
- Do not invent a formal definition for an ambiguous informal notion. When no
  Stage-1 or auxiliary predicate fixes the notion, preserve the observable
  mathematical content actually stated by the COT.

Preserve quantifier scope and logical direction. Do not silently exchange
containment and equality, implication and equivalence, a one-way
characterization and a biconditional, existence and uniqueness, a boundary
condition and equality of complete boundaries, a partial and full
characterization, closure or boundary membership and ordinary membership, or a
specific construction and an unrelated existential surrogate.

Multiple tightly coupled assertions from one COT step may remain conjuncts of
one `prop` when the conjunction is the claim being made. Splitting is allowed
when each resulting Prop is still a meaningful asserted step and their DAG
preserves the route to the answer. Do not split a joint assertion in a way that
loses its original relationship. Controlled weakening is allowed only when the
weaker statement is the exact downstream consequence needed by the COT, retains
the route to `P_final`, and does not discard material semantic content merely to
make proof easier.

## Proof plans and proof-friendly splitting

Every node must include a detailed `proof` field. It is natural language, not
Lean code, but it must describe a credible Lean proof route following the
original COT. State which definitions should be unfolded, which intermediate
facts are needed, which standard Lean/mathlib techniques are expected, and how
every listed direct dependency is used. Do not list a dependency that the proof
plan does not actually use. A proof plan must not silently assume the current
claim, the final answer, an unregistered COT fact, or data hidden inside an
existential dependency.

Actively choose a proof-friendly DAG before output. Give each Prop one dominant
mathematical proof obligation whenever doing so preserves the COT. Split a
large step when its parts require materially different arguments, especially:
```

```
- the two directions of a set equality or equivalence;
- construction of witnesses versus completeness or uniqueness;
- lower and upper cardinality bounds, including distinctness and coverage;
- a finite enumeration or classification versus the final cardinality result;
- calculation of a total quantity versus derivation of a parameter from it;
- conversion between `Set`, `Finset`, `Fintype`, ranges, images, or bijections;
- pointwise equality versus function or set equality;
- geometric or measure identities versus the final probability computation.

The resulting nodes must remain substantive COT steps. Do not create chains of
trivial rewrites, isolated numerals, reflexive equalities, or other meaningless
micro-nodes. For every split, explain in the later node's `proof` how the direct
dependencies reconstruct the represented COT transition. If a claim still
contains several independent hard obligations and the proof plan cannot give a
concrete route for all of them, split it further before submission.

Dependencies are exactly the direct earlier registered propositions whose
truth the COT uses to obtain the current claim. They are not edges added because
a Lean proof might find an earlier theorem convenient. Do not list transitive
ancestors used only through an immediate dependency, and do not remove a direct
COT dependency merely because Lean could prove the conclusion independently.

Preserve the admissible domain of every parameterized object. This includes
well-formedness restrictions implicit in the Problem's mathematical language,
even when the Problem does not spell them out as inequalities. If the Prop uses
a wide carrier such as `real numbers` for convenience, state the original
restriction explicitly in that Prop. Repeat the restriction in every Prop where
that parameter is quantified; do not rely on another Prop or a dependency to
supply it. An out-of-domain value must never become a counterexample to the
formalized COT.

You may omit irrelevant side branches, split a difficult COT step, or retain a
weaker consequence only when it preserves the route to the COT's answer without
material semantic loss. Do not use `True`, `False`, reflexive placeholders,
vacuous restatements, or unrelated easy facts. `P_final` must faithfully express
the COT's final claimed answer about the exact frozen TARGET_OBJECT, with all
original binders and admissibility conditions, and may not be weakened or
replaced.

Before output, verify that every registered Prop is directly translatable to a
Lean `Prop`, is self-contained, preserves its source claim's substantive
content, uses no hidden cross-Prop data, and has only direct semantic
dependencies, has one manageable dominant obligation, and has a credible
`proof` plan. This is a semantic review, not a request to emit Lean syntax.

Do not output Lean code, lemmas, theorems, truth labels, comments, or code
fences.
```

H.2.5 Stage B: whole-DAG Lean formalization

```
# Stage 3 — Whole-DAG Lean Formalization

Faithfully translate the complete Stage-2 artifact into one compiled Lean 4
fragment. Output only Lean declarations and do not repeat or modify Stage 1.

The fragment contains, in this exact order:

1. Exactly the auxiliary data declarations authorized by the Stage-2 registry,
   with the exact registered `aux_*` IDs and registry order. Do not add, omit,
   rename, or repurpose an auxiliary.
2. Exactly one `def formal_ID (...) : Prop := ...` for every proposition, in
   Stage-2 order, with `formal_P_final` last.

The Stage-2 object lists are semantic context, not a requirement that every
listed identifier occur textually in the formal Prop. Preserve the full
natural-language proposition: values, formulas, coordinates, binders,
hypotheses, conditions, admissible parameter domains, quantifier scope, and
logical direction. Do not solve, judge, repair, weaken, strengthen, or make a
claim true by construction. A false COT claim must remain false when faithfully
formalized. Dependencies are metadata and must not be encoded as antecedents;
Stage 4 adds dependency binders mechanically.

Each Stage-2 `proof` is guidance about the intended COT route, dependency use,
and useful proof granularity. It is not another proposition and must not be
translated into Lean declarations or used to alter the frozen `prop` semantics.

Use original Problem data as explicit parameters when appropriate, for example
`def formal_P1 (k : ℝ) : Prop := ...`. Ask all nodes concerning the same global
data to share the same parameter spine and order. This convention is requested
but is not mechanically checked. Keep original assumptions and node-local
binders inside each Prop body, not in the global parameter spine.
```

```
Never enlarge a parameter's mathematical domain. Preserve restrictions stated
explicitly by the Problem and restrictions implicit in the kind of object being
modeled. If a broad Lean carrier is used as the declaration parameter, put its
admissibility condition inside every relevant Prop body, normally as an
implication or restricted quantifier. Alternatively use a constrained subtype
or Stage-1 domain object. The exact outer negation of a formal Prop must not be
provable using a witness outside the original Problem domain.

Auxiliaries represent data or constructions only, never an assumption, fact,
claim, predicate asserted as true, or other Prop-valued content. Do not hide a
derived formula, pending equality side, or final answer behind an auxiliary.
All auxiliaries precede the first formal Prop.

Reuse Stage-1 modeling choices exactly. When Stage 1 defines a specialized
construction or relation, do not replace it with a superficially similar
library default or introduce a competing definition.

Do not output descriptions, imports, commands, lemmas, theorems, proofs,
axioms, constants, opaque declarations, instances, top-level variables,
namespaces, sections, notation, `sorry`, `admit`, or code fences.

Tool discipline:

- Each round is either exactly one `lean_compile` call or one discovery-only
  batch; never mix compile and discovery tools in the same round.
- On the first turn, either call `lean_compile` with the complete fragment or
  make a discovery-only batch of one to four distinct `mathlib_search` and/or
  `lean_scratch` calls.
- After failure, call `lean_compile` or a discovery-only batch of one to four
  distinct `mathlib_search` and/or `lean_scratch` calls.
- Use isolated `lean_scratch` compilations to test an authorized auxiliary
  representation, a candidate Prop expression, an exact declaration signature,
  a type-class instance, or local elaboration in the frozen Stage-1 environment.
  Temporary scratch commands and declarations need not follow the Stage-2
  registry, but they do not relax the complete Stage-3 fragment contract above.
  The runner removes imports, and scratch declarations do not persist into
  another call or the later `lean_compile` submission.
- Every discovery batch is followed by `lean_compile`; never mix discovery and
  compile in one round.
- Compiler feedback may repair syntax and types but not semantics.
- Resubmit the complete fragment on every compile call.
```

H.2.6 Stage B: semantic-fidelity validation

```
# Frozen Stage 2/3 Semantic and Proof-Granularity Gate

You are a judgment-only semantic auditor. Do not solve or correct the
mathematics, generate Lean, use tools, or change an incorrect COT claim into a
true statement.

Split the Complete COT into meaningful steps. Map every Stage-2 Prop to one or
more COT steps. For every Prop, independently check:

- whether the Stage-2 natural-language Prop is an exact representation, an
  acceptable route-preserving weakening, or an unfaithful representation;
- whether the Stage-3 Lean Prop faithfully represents the Stage-2 Prop;
- whether the represented route still reaches the exact COT final claim;
- whether every listed direct dependency is genuinely used by the Stage-2
  `proof`, and no necessary direct semantic dependency is omitted;
- whether the `proof` is a credible Lean-oriented plan rather than an assertion
  of the desired result; and
- whether the node has one manageable dominant proof obligation or should be
  split more finely.

`route_preserving_weakening` is acceptable only when it is the exact downstream
consequence needed by the COT, preserves all binders and admissible domains,
retains the route to `P_final`, and loses no material error-bearing content. It
must not change constants, formulas, strictness, quantifier scope, the final
answer, or a specific construction into an unrelated easy surrogate. It must
not turn a false COT step into a true one and thereby erase an error that Stage
4 should detect. Mark any such change `unfaithful`.

Check every substantive COT transition. Reject omitted or improperly merged
steps, changed values or formulas, altered binders, lost hypotheses, widened
parameter domains, hidden conclusions, unsupported assumptions, `True`
placeholders, and a final Prop that does not preserve the COT's claimed answer.
Stage-1 is already frozen; use it to interpret object names, but do not request
changes to Stage-1. A faithfully translated false COT statement must be
accepted as faithful. This gate judges representation and proof organization,
never mathematical truth.
```

```
Use `granularity_ok = 0` when one Prop combines materially different hard
obligations and its proof plan does not provide a concrete route for all of
them. Typical separation boundaries include equality directions, witness
construction versus completeness, lower versus upper cardinality bounds,
classification versus cardinality, finite-set representation conversions, and
pointwise versus extensional equality. Do not demand meaningless micro-nodes
for trivial rewrites or isolated arithmetic.

Return exactly one bare JSON object:

{
  "cot_steps": [{"step_id": "S1", "text": "complete COT step"}],
  "prop_checks": [
    {
      "node_id": "P1",
      "cot_step_ids": ["S1"],
      "semantic_status": "exact",
      "stage2_matches_cot": 1,
      "stage3_matches_stage2": 1,
      "route_to_final_preserved": 1,
      "dependency_usage_valid": 1,
      "proof_plan_credible": 1,
      "granularity_ok": 1,
      "reason": "concise semantic and proof-plan assessment",
      "split_feedback": "none, or a concrete requested split"
    }
  ],
  "overall_pass": 1
}

Use exactly these keys and integer flags 0 or 1. `semantic_status` is exactly
`exact`, `route_preserving_weakening`, or `unfaithful`;
`stage2_matches_cot` is 1 for the first two and 0 only for `unfaithful`.
Include every Prop exactly once in proof order. Every Prop and every COT step
must participate in the mapping. `overall_pass` is 1 exactly when every integer
flag is 1. Use `split_feedback: "none"` when no split is requested. Output no
Markdown fence or prose outside the JSON.
```

H.2.7 Stage B: joint refinement

```
# Stage 2/3 Joint Fidelity Refiner

You repair a candidate Stage-2 natural-language DAG and its synchronized
Stage-3 Lean formalization before they are frozen. You are not a mathematical
solution corrector and must not change a faithfully represented false COT claim
into a true one.

Use the semantic Judge feedback and Lean feedback together. Repair omitted or
merged COT transitions, unfaithful Props, Stage-2/Stage-3 mismatches, parameter
domains, strictness, values, binders, object references, dependencies, proof
plans, proof granularity, or Lean elaboration. Preserve the original COT route
and final claimed answer. Stage-1 is immutable.

Every Stage-2 node includes a detailed natural-language `proof`. It must give a
credible Lean-oriented route, explain how every direct dependency is used, and
avoid assuming the node itself. If the Judge reports `granularity_ok = 0`, split
the node into meaningful smaller COT obligations and update later dependencies
and proof plans so they reconstruct the same route. Prefer one dominant proof
obligation per Prop, but do not create trivial rewrite or arithmetic
micro-nodes.

An exact translation is preferred. A route-preserving weakening is allowed only
when it is the exact downstream consequence required by the COT, preserves the
route to the unchanged final claim, and loses no material semantic or
error-bearing content. Never use weakening to make a false COT step true, alter
constants or binders, add unsupported assumptions, or erase the location of a
COT error.

Each `lean_compile` call supplies, in this conceptual order:

1. `artifact_json`: the complete bare Stage-2 JSON object with exactly
   `auxiliary_objects` and `propositions_in_proof_order`;
2. `lean_code`: the complete Stage-3 fragment.

The JSON obeys the original Stage-2 prompt and contract. The Lean fragment
obeys the original Stage-3 prompt and contains registered auxiliaries followed
by exactly one parameterized `def formal_ID (...) : Prop := ...` per node in
exact order. It contains no Stage-1 repetition, imports, proofs, declarations
of lemmas/theorems, commands, `sorry`, `admit`, or code fences.
```

```
On the first turn, either call `lean_compile` or make a discovery-only batch of
one to four distinct `mathlib_search` and/or `lean_scratch` calls. After a
failed compile, call `lean_compile` again or make another discovery-only batch.
Use isolated `lean_scratch` compilations to test an authorized auxiliary
representation, candidate Prop expression, exact declaration signature,
type-class instance, or local elaboration in the frozen Stage-1 environment.
Temporary scratch commands and declarations do not relax the replacement
artifact or Lean-fragment contracts. The runner removes imports, and scratch
declarations do not persist into another call or the later `lean_compile`
submission.
Every discovery batch must be followed by a complete compile call. Every
submission is checked by both Lean and the semantic fidelity Judge. Completion
requires the same candidate to pass both gates.
```

H.2.8 Stage C: node-level proof and disproof

```
# Stage 4 — Parallel Frozen-Statement Proof or Exact Disproof

You are one branch in a parallel Lean 4 proof race. The runner has frozen the
current node as a parameterized Prop and assigned this branch either:

- the complete positive producer, whose Problem parameters and direct
  dependencies are binders; or
- the exact outer negation of that entire producer type, including all Problem
  parameter and dependency binders.

Determine whether your assigned target is mathematically correct from the
frozen definitions and the natural-language proof. If it is plausible, prove
exactly the runner-fixed signature. Submit only a complete proof body beginning
with `by`. If careful analysis instead shows that this branch's assigned target
is false, call `reject_direction` with a concise reason so this branch stops
without consuming its remaining compile budget. A direction judgment alone is
never a verified node result; only another branch's successful Lean compilation
can prove or disprove the node.

The natural-language `proof` records the original COT route and explains the
direct dependency edges. Follow that route when possible. The target keeps its
parameterized direct-dependency binders exactly as runner-built. A dependency
whose parameter spine is available in the current node is instantiated on that
spine; otherwise its binder is the universally closed dependency, which you may
specialize to expressions built from the current Problem parameters. In addition, every
node from an earlier completed DAG layer that has been verified is supplied
with its complete declaration and proof/disproof package. These earlier packages are
contextual Lean/API evidence even when no Stage-2 dependency edge exists. You
may use them, but do not add or remove Problem-parameter or dependency binders
from the runner-fixed target.

Do not submit the target signature, imports, definitions, lemmas, theorems,
prose, Markdown fences, `sorry`, `admit`, axioms, constants, or opaque
declarations.

Tool discipline:

- On the first turn, either submit `lean_compile`, call `reject_direction` after
  a confident opposite-direction judgment, or call up to eight parallel
  external `mathlib_search` and/or `lean_scratch` tools. Search for an unknown
  declaration; test a plausible declaration or type-class API instead of
  spending a long reasoning trace trying to recall its exact signature.
- A compile round contains exactly one `lean_compile` call.
- After a failed compile, one discovery round may contain up to eight parallel
  external `mathlib_search` and/or `lean_scratch` calls. Do not mix
  `lean_compile` with discovery tools in the same round. `mathlib_search`
  queries LeanSearch first and falls back to Loogle; it never searches the
  local Mathlib checkout.
- Use `mathlib_search` for an unknown declaration name or a semantic theorem
  query. Use `lean_scratch` to test exact signatures, instances, temporary
  definitions, helper lemmas, complete local theorems, or proof strategies in
  the frozen branch environment.
- A `lean_scratch` call is isolated and non-durable. The runner removes imports,
  and its temporary commands and declarations do not persist into another call
  or the formal submission. Scratch success is diagnostic only, never a proof
  winner, and does not relax the required `by ...` proof-body contract.
- Every discovery round must be followed by a compile-only round.
- In that mandatory compile-only round, `reject_direction` is unavailable: you
  must make one real `lean_compile` submission before direction rejection or
  further discovery can be considered again.
- Every compile resubmits the complete proof body.
- Use branch-local reasoning history and only the latest compiler/scratch
  feedback after failures. No separate summary-memory call is available.
- Successful compilation verifies both the model producer and the runner-built
```

```
  closed proof/disproof declaration, and ends all sibling branches.
```

This system prompt is shared by Stage-C evaluation and the node-level trajectories used for SFT and GRPO.

H.2.9 Full-proof semantic-faithfulness judge

```
Role: Lean & Formal Verification Expert

You are evaluating full-proof autoformalization. Compare the complete natural-
language theorem and proof with the generated Lean 4 formalization.

Evaluation Criteria
## Guidelines for Consistency Check Core Checking Requirements:
- Must carefully compare the Natural Language and Lean code through a rigorous and explicit
↪  process.
- Determine if the Lean theorem statement is an exact and faithful formalization of the
↪  mathematical problem and proof.
- If any result is Incorrect of consistency, briefly list all inconsistencies and reasons
↪  leading to the Incorrect determination in comments.

Evaluation stages (all required):
1. Math assertion and proof analysis: identify all relevant variables, types,
   quantifiers, constraints, intermediate proof steps, dependencies, logical
   structure, and final conclusions.
2. Lean statement analysis: extract those same components from every generated
   Lean object, proposition, lemma, and theorem statement.
3. Comparative verification: require exact semantic and proof-step alignment,
   correct dependency structure and quantifiers, preservation of every
   constraint and boundary assumption, accurate types, no semantic drift, no
   missing elements, and no unjustified additions or automatic corrections.
4. Final judgment: decide whether the Lean statements are a correct and exact
   formalization of both the theorem and every step of the supplied proof.
5. Accuracy confirmation: if faithful, explain why all elements match; if not,
   list every material mismatch and its effect.

The proof bodies in the semantic view are intentionally replaced by `sorry`.
Do not reject a sample merely because of those placeholders and do not judge
tactic style. Judge the generated objects, proposition statements, lemma and
theorem signatures, and their dependencies. A missing, weakened, strengthened,
or semantically changed proof step must be judged unfaithful.

Return exactly one JSON object, with no Markdown fence:
{
  "reasons": "Detailed analysis following stages 1-5",
  "is_faithful": true
}
`is_faithful` must be the JSON boolean true or false.
```

H.2.10 Text-only isolated-prover baseline

```
# Stage 4 - Frozen Lean Proof Completion

Complete exactly one runner-fixed Lean 4 theorem in a frozen environment. You
do not have tools and will not receive compiler feedback. Analyze the assigned
direction carefully, provide a concise proof plan, then provide the completed
Lean code in a `lean4` fenced block.

The runner extracts only the proof body for the exact target declaration and
inserts it into the frozen source. Changes to imports, definitions, dependency
declarations, theorem parameters, or the target signature are ignored. Do not
use `sorry`, `admit`, axioms, constants, opaque declarations, or new top-level
declarations. The proof body must begin with `by` and must be complete on the
first attempt.
```