FormalEvolve: Neuro-Symbolic Evolutionary Search for Diverse Autoformalization

Anonymous ACL submission

Abstract

Autoformalization aims to produce formal statements that compile and faithfully preserve the intended meaning of informal mathematics. Yet standard single-output evaluation protocols collapse a many-to-many problem into a single-output prediction task. For downstream proving, this granularity is too coarse: a formal statement is not merely a faithful translation endpoint, but also a prover-facing interface whose structure can alter proof search under a fixed budget. We therefore recast autoformalization as budgeted test-time search: FormalEvolve maintains a compilation-feasible archive for reuse, while reporting the deduplicated semantically accepted repertoire for evaluation and downstream proving. It expands the archive with LLM-driven mutation, crossover, bounded patch repair, and symbolic Abstract Syntax Tree (AST) rewrites for structural diversity. Under a generator-call budget of T=100 with a fixed LLM semantic judge, FormalEvolve reaches SH@100 of 58.0% on CombiBench and 84.9% on ProofNet, improving over all no-archive controls while reducing the cross-problem concentration of semantic successes. To assess downstream value, we evaluate the resulting repertoires under a fixed B=64 prover budget, where they improve theorem-complete proving over the matched no-archive control; additional stronger-base statement-generation experiments show that archive-search gains hold with stronger seed and repair models. Manual faithfulness audits calibrate these judge-positive outputs.

1 Introduction

Autoformalization translates informal mathematics into Lean statements that compile and preserve the intended mathematical claim. Yet the task is naturally many-to-many: the same theorem can often be encoded as multiple faithful Lean statements, while standard single-output protocols collapse this structure into one generated statement. For downstream proving, that granularity is too coarse. A formal statement is not merely a translation endpoint; it is the prover-facing interface through which proof search is attempted under a fixed prover and attempt budget. Compilation requires successful elaboration and type checking under the chosen context, while semantic faithfulness remains a separate requirement. As large language model provers become more capable (Polu and Sutskever, 2020; Han et al., 2022; Yang et al., 2023; Song et al., 2024), structural choices in the statement itself increasingly affect downstream proof search.

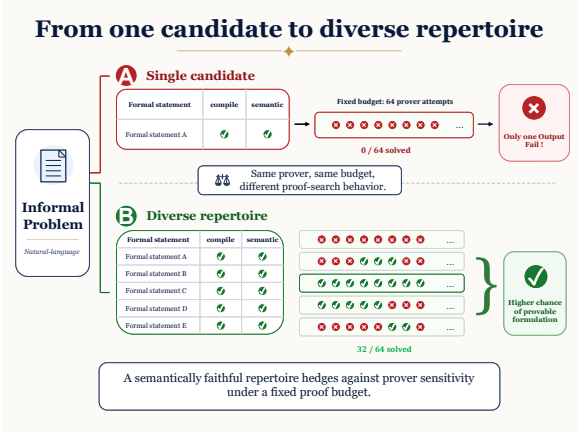

Figure 1: Motivating illustration under a fixed prover and attempt budget (N=64). Even when candidate statements compile and are judged semantically consistent, prover outcomes can vary substantially. Constructing a diverse repertoire hedges against this sensitivity and increases the chance that at least one provable statement is found within budget.

Semantic consistency therefore does not imply prover equivalence: by *prover equivalence* we informally mean comparable proof-search behavior under the same prover and attempt budget. A formal statement that compiles and preserves the intended claim is not necessarily a prover-friendly target; faithful variants can differ substantially in proof-search cost and success rate. Figure 1 illustrates this gap: candidates that pass compila-

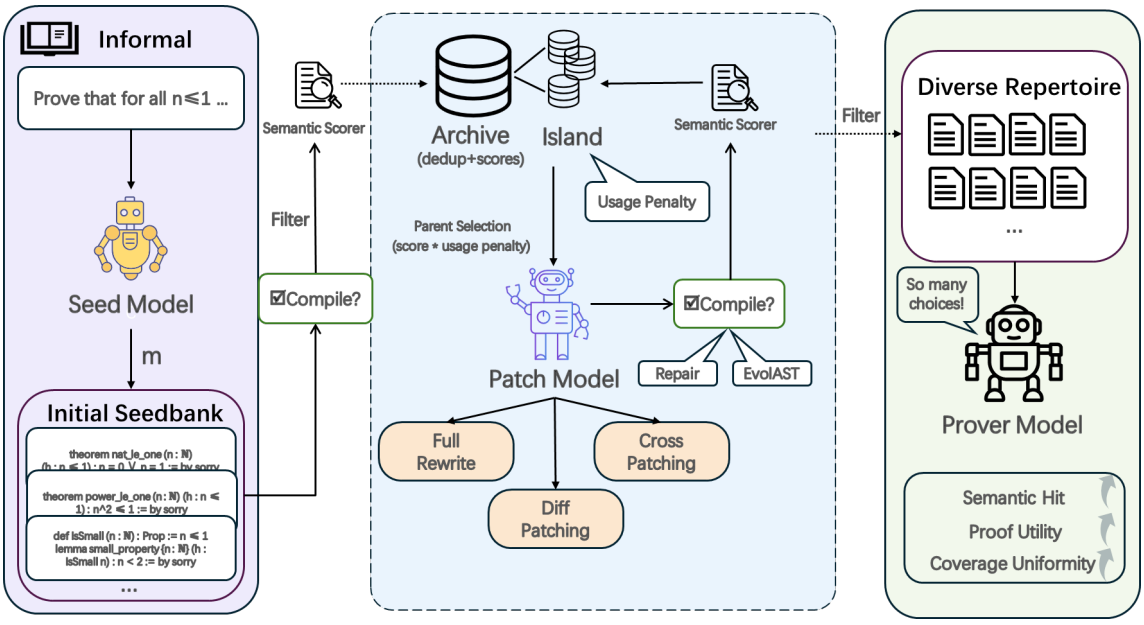

Figure 2: Framework overview of FormalEvolve. A seed model produces an initial seedbank (debited as generator calls) that initializes a compilation-feasible archive. Candidates are stored with semantic scores and selected with usage penalties; a patch model proposes edits (full rewrite, diff patching, and cross patching) and triggers bounded repair or EvolAST fallbacks on compilation failures. Semantic scoring filters the archive into a diverse repertoire for downstream proving and evaluation.

tion and semantic checks can still lead to different prover outcomes under the same prover protocol. A single generated statement therefore discards prover-relevant variation. This motivates a repertoire view: the search procedure should construct multiple faithful Lean statements that expose different proof-search behavior under the same prover budget.

We formulate autoformalization as a budgeted test-time search problem. Under a generator-call budget, the system must construct a repertoire of candidate formalizations that are compilable, semantically consistent, and useful for downstream proving. This requires preserving compilation feasibility, avoiding semantic drift, and introducing structural diversity, without burning budget on a handful of easy problems or near-duplicates.

To address these requirements, we propose FormalEvolve, a compilation-gated neuro-symbolic evolutionary framework built around an archive search protocol. FormalEvolve maintains a compilation-feasible archive of candidate Lean statements and expands it primarily through LLM-driven mutation, crossover, and bounded patch repair. In addition, conservative generator-call-free AST rewrites act as a structural diversification fallback when patching stalls, for example after repeated duplicates or compile failures. Usage penalties and duplicate-aware selection encourage the search to cover more problems and statement forms. Figure 2 gives a high-level overview.

Under a generator-call budget of T=100, FormalEvolve reaches SH@100 of 58.0% on CombiBench and 84.9% on ProofNet, improving over all no-archive controls on both benchmarks while reducing the cross-problem concentration of semantic successes. Under a fixed B=64 prover budget, the resulting repertoires improve theorem-complete downstream proving over the matched no-archive control on both benchmarks. To separate repair-model strength from archive search, we include a matched hybrid control that uses the same stronger repair model without an archive; FormalEvolve still gains at the statement stage. The pattern also holds in a stronger-base statement-generation setting. Because semantic labels come from an LLM judge, we also provide manual audits at both the statement and theorem-complete stages to calibrate the evaluation.

More broadly, FormalEvolve illustrates a verifier-gated diversity-search pattern: when many valid formal artifacts can express the same informal intent, test-time compute can construct a repertoire of useful alternatives under hard checker feedback.

We make three primary contributions:

1. **Problem Modeling and Protocol:** We formulate Lean 4 autoformalization as budgeted test-time repertoire search and introduce budget-

auditable metrics that connect statement-level coverage, cross-problem concentration, and downstream proving performance.

2. **Search Mechanism:** We propose FormalEvolve, a compilation-gated neuro-symbolic evolutionary framework that combines archive-based LLM mutation and crossover, bounded patch repair, and generator-call-free symbolic AST rewrites.

3. **Empirical Evidence:** We show that FormalEvolve improves semantic hit rates and cross-problem coverage uniformity under fixed generator-call budgets, improves theorem-complete downstream proving over a matched no-archive control by constructing diverse candidate repertoires, and preserves statement-stage archive-search gains under a stronger-base setting.

2 Related Work

Autoformalization and semantic evaluation. Large language models have enabled autoformalization at scale, with systematic evaluation on curated formal-mathematics benchmarks (Wu et al., 2022; Azerbayev et al., 2023; Zheng et al., 2022). Reliability remains a central bottleneck: compilation checks syntax and elaboration, but is only a weak proxy for semantic consistency, motivating dedicated semantic judges and alignment evaluators (Peng et al., 2025; Lu et al., 2024). Recent systems further improve semantic fidelity by interleaving iteration, tool feedback, and grounding (Chen et al., 2025; Guo et al., 2025; Wang et al., 2025; Yang et al., 2023). These lines of work primarily improve or assess individual formalizations within a single-output view. We study fixed-budget construction of semantically consistent and diverse candidate repertoires.

Statement variation and prover utility. For downstream proving, a formal statement serves both as a semantic target and as the object consumed by a prover. Even among semantically consistent candidates, minor, semantics-intended reformulations can substantially change proof-search difficulty (Zhao et al., 2025). At the same time, semantic evaluation itself is imperfect and can disagree with downstream prover behavior (Chen et al., 2025). These observations motivate treating autoformalization as a fixed-budget repertoire-construction problem, where multiple faithful candidates expose different prover utilities. Such a repertoire mitigates prover sensitivity by exposing several semantically aligned targets under the same fixed budget.

Test-time search and quality-diversity. Test-time search and evolutionary optimization with LLMs have been explored for sample-efficient program and algorithm improvement (Novikov et al., 2025; Lange et al., 2025; Liu et al., 2024). Quality-diversity methods construct repertoires of high-quality solutions (Lehman and Stanley, 2011; Mouret and Clune, 2015), and multi-criteria evolutionary optimization provides a complementary perspective on multiple objectives (Deb et al., 2002). EvolProver evolves formalized problems via symmetry and difficulty, including semantics-preserving AST rewrites, to augment training data for provers (Tian et al., 2025). For Lean statement search, elaboration and type checking define the feasible set: a candidate must compile before it can enter the archive or be reused as a parent. For each informal problem, FormalEvolve builds a test-time repertoire of candidate statements used for that problem's evaluation and downstream proving. EvolAST-style conservative rewrites are used only as a generator-call-free fallback for structural diversification when LLM patching stalls.

3 Method

FormalEvolve instantiates the shift from single-output prediction to many-to-many repertoire construction as per-problem test-time search over Lean 4 statement candidates under a generator-call budget T (Section 4). Given an informal statement x, the goal is to construct a deduplicated repertoire of candidates that compile and pass the semantic judge within budget. The search starts from a small compilation-feasible seedbank, stores unique compiled candidates in an archive, and expands it through LLM patching, bounded repair, and call-free EvolAST diversification. Because Lean compilation defines a narrow feasible region, archive entry is compilation-gated, near-feasible candidates are salvaged by bounded repair, and lightweight diversity operators reduce collapse. Our reported output is the deduplicated semantically consistent repertoire generated within budget, i.e., candidates satisfying $\mathsf{Comp}(c)=1$ and $\mathsf{Sem}(c)=1$, which we use for SH@T and downstream proving.

Algorithm 1 summarizes the same loop and makes the two domain gates explicit: Comp

checks whether a candidate passes Lean 4 compilation, while Sem is the semantic-faithfulness judge against the informal theorem x. The counter t tracks debited generator calls, $\mathcal{A}_I$ is an island archive, $\mathcal{Z}$ denotes sampled prompt context and inspirations, $\mathcal{B}$ is the candidate batch produced by proposal and bounded repair, and $\mathcal{G}$ is the persistent accepted repertoire returned at the end. Figure 2 gives the visual overview, and Appendix E.1 gives the detailed pseudocode.

3.1 Search Procedure and State

Basic objects and gates. We search over Lean 4 candidates of the form $c = (\hat{h}, y)$, where $\hat{h}$ is an import/header context and y is the primary Lean 4 statement or declaration target. Most candidates are theorem/lemma-style files with placeholder proofs, but benchmark contexts and model outputs can also contain `def`, `abbrev`, or `example` templates; for this reason, the prover-stage evaluation separately reports theorem-complete success for explicit `theorem`/`lemma` completions. We evaluate candidates using a compilation predicate $\mathsf{Comp}(c) \in \{0, 1\}$ and a semantic predicate $\mathsf{Sem}(c) \in \{0, 1\}$ (instantiated by an LLM judge in Section 4). Compilation is a hard feasibility gate: only candidates with $\mathsf{Comp}(c) = 1$ are eligible to enter the archive and be reused as parents/inspirations. We treat compilation/elaboration as verifier feedback: it both defines feasibility and provides the primary error feedback for bounded compilation repair. During search, the archive stores unique compilation-feasible candidates regardless of $\mathsf{Sem}(c)$ (i.e., it may include semantically inconsistent stepping stones) and is used only for parent/context sampling. For reporting SH@T and downstream proving, we define the *semantically consistent repertoire* as the deduplicated set of candidates generated within budget that satisfy both $\mathsf{Comp}(c) = 1$ and $\mathsf{Sem}(c) = 1$. FormalEvolve uses these two predicates during test-time search and bounded repairs; downstream proving is evaluated only when reporting proof utility. All generator-side LLM calls (seeds, patches, and repairs) are debited against the same per-problem generator-call budget T (Section 4).

Seedbank initialization. We initialize the search by sampling a small seedbank with a seed model M_{seed}, evaluating each seed with the compiler and semantic judge, and inserting the compilation-feasible subset into the archive. When a pre-

Algorithm 1 FormalEvolve for Lean statement search

Task: given informal theorem x, search Lean 4 statement candidates c
Gates: $\mathsf{Comp}(c)$ checks whether c passes Lean 4 compilation; $\mathsf{Sem}(c)$ checks faithfulness to x
Input: budget T, models $M_{\text{seed}}, M_{\text{patch}}$, islands K
Output: accepted repertoire $\mathcal{G}$
$t \leftarrow 0$; $\mathcal{A} \leftarrow$ empty K-island archive; $\mathcal{G} \leftarrow \emptyset$
$(\mathcal{A}, t) \leftarrow \textsc{SeedArchive}(x, M_{\text{seed}}, T, K)$ *(Seedbank initialization)*
$\mathcal{G} \leftarrow \{c \in \bigcup_I \mathcal{A}_I : \mathsf{Comp}(c) \wedge \mathsf{Sem}(c)\}$
while $t < T$ and $\bigcup_I \mathcal{A}_I \neq \emptyset$ **do**
  $I \leftarrow \textsc{SampleIsland}(\mathcal{A})$ *(Archive, islands, and duplicates)*
  $p \leftarrow \textsc{SampleParent}(\mathcal{A}_I)$; $n_p \leftarrow n_p + 1$ *(Scoring; Eq. 2)*
  $\mathcal{Z} \leftarrow \textsc{SampleContext}(\mathcal{A}_I, p)$
  $c \leftarrow \textsc{Propose}(p, \mathcal{Z}, x; M_{\text{patch}})$; $t \leftarrow t + 1$ *(Variation operators)*
  $(\mathcal{B}, t) \leftarrow \textsc{ExpandAndRepair}(c, p, x, t, T)$ *(Bounded repair + EvolAST fallback)*
  $\mathcal{A}_I \leftarrow \textsc{EvaluateAndUpdate}(\mathcal{A}_I, \mathcal{B})$
  $\mathcal{G} \leftarrow \mathcal{G} \cup \{b \in \mathcal{B} : \mathsf{Comp}(b) \wedge \mathsf{Sem}(b)\}$
end while
return $\mathrm{Dedup}(\mathcal{G})$

sampled seedbank is reused across methods, we still debit each reused seed as one generator call to preserve consistent accounting. We separate the seed and patch roles: M_{seed} provides initial candidates, while M_{patch} performs edit-conditioned proposals and bounded repairs. Concrete model choices are specified in Section 4.

Archive, islands, and duplicates. We maintain a per-problem archive $\mathcal{A}$ of unique compilation-feasible candidates that serves both as a parent pool and as prompt context. We use an island model with K semi-isolated archive populations (default $K = 2$). When $K = 1$, FormalEvolve uses a single archive population: it still retains archive-based parent reuse, archive-conditioned context sampling, usage-penalized selection, and bounded repair, but removes island partitioning and migration. Each iteration samples one island and restricts parent/context sampling to it. In the main configuration, migration occurs every 10 generations at rate 0.1 with the top candidate in each island protected. Detailed hyperparameters are provided in Appendix E.3. Exact duplicates under canonicalization (whitespace/name normalization) are rejected at archive insertion to reduce collapse. In Algorithm 1, SAMPLEPARENT uses the scoring rule defined below, and n_i tracks how often c_i has already been selected as a parent.

Scoring and usage-penalized selection. The selection rule separates archive storage from parent

sampling. Storage is purely feasibility-gated: only candidates with $\mathsf{Comp}(c) = 1$ can enter the archive, and the archive retains all unique compilation-feasible candidates. This matters because different faithful statements can expose different binder structures, imports, and theorem shapes to the downstream prover. Scores therefore bias reuse as future parents; archive contents remain available for later sampling, and proof utility is evaluated downstream under the fixed prover protocol.

For an archived candidate $c_i \in \mathcal{A}_I$, let n_i be how many times it has already been selected as a parent. We assign each candidate a sampling weight w_i and sample proportionally:

$$\Pr(p = c_i \mid \mathcal{A}_I) = \frac{w_i}{\sum_{c_j \in \mathcal{A}_I} w_j}. \tag{1}$$

The weight combines the gate score and a usage discount:

$$\begin{aligned} s_i &= \mathsf{Comp}(c_i)(1 + \mathsf{Sem}(c_i)), \\ \tilde{s}_I &= \mathrm{median}_{c_j \in \mathcal{A}_I}(s_j), \\ d_I &= \max(\mathrm{MAD}(\{s_j : c_j \in \mathcal{A}_I\}), \varepsilon), \\ z_i &= (s_i - \tilde{s}_I)/d_I, \\ u_i &= [1 + (1 + \beta)n_i]^{-1}, \\ w_i &= \sigma(\lambda z_i) u_i. \end{aligned} \tag{2}$$

Since archived candidates already compile, $s_i = 1$ for judge-negative archive members and $s_i = 2$ for judge-consistent members. u_i is a usage discount: it equals 1 for a never-used parent and decreases as the same parent is sampled repeatedly. We set $\varepsilon = 10^{-6}$ only to avoid division by zero when all scores are identical, and use $\lambda = 10$ and $\beta = 0.05$ in the main runs.

3.2 Proposal, Repair, and Diversification

Variation operators. Given a selected parent, the patch model proposes a new candidate using one of three prompt templates. *Full* patching rewrites the entire statement (and, when allowed, its surrounding imports) conditioned on the informal input and parent context. *Diff* patching performs localized edits that make minimal changes relative to the parent, guided by compiler/judge feedback. *Cross* patching additionally conditions on one or more *inspiration* candidates sampled from the current archive, and asks the model to combine useful elements (e.g., import context, binder structure, or type annotations) while preserving the informal input's meaning (see Appendix F for prompt details and Appendix D for audited examples). All three templates return a complete Lean 4 file (imports plus a single statement) and are budgeted identically as generator calls.

Bounded compilation and semantic repair. We compile the full Lean 4 file and treat successful compilation as a hard feasibility gate. When compilation fails, FormalEvolve invokes a repair prompt that proposes a minimal patch conditioned on compiler feedback. Repair is bounded to a fixed cap of two repair attempts per proposal path in the main configuration; each attempt is debited as a generator call, and repair stops immediately when the remaining budget is exhausted. Bounding repair makes the search budget-auditable under hard cutoffs and prevents pathological cases where a single hard candidate consumes most of the call budget. For a compilable candidate that fails the semantic judge ($\mathsf{Comp}(c) = 1$ and $\mathsf{Sem}(c) = 0$), we optionally apply a bounded *semantic repair* stage by prompting M_{patch} to revise the statement conditioned on the informal input and the judge's rationale (when available), and then re-evaluate; all repair and semantic-repair attempts are debited. We view compilation repair and semantic repair as two instantiations of the same edit-conditioned patch mechanism, differing only in the feedback source (compiler errors vs. judge rationale).

Call-free diversity fallback (EvolAST). Archive-based search can suffer from mode collapse where many candidates become near-duplicates or exploit superficial patterns. FormalEvolve mitigates this primarily via the usage penalty in parent sampling. We additionally apply a conservative, generator-call-free AST rewrite fallback inspired by EvolProver (Tian et al., 2025) when patching stalls (e.g., repeated duplicates or compile failures). In the reported runs, each triggering proposal contributes at most one EvolAST representative, and EvolAST outputs are not recursively rewritten or sent into LLM repair. EvolAST rewrites only within binder types and the goal type (imports/preamble unchanged) to produce symmetry/structure variants, which are then filtered by the same compilation gate and semantic judge. Thus EvolAST is free only with respect to generator calls; its compilation and judge overhead is reported separately in Appendix E.2.

4 Experiments

Benchmarks. We evaluate on ProofNet (Azerbayev et al., 2023) (Lean 4 port; test split, N=186) and CombiBench (Liu et al., 2025) (N=100, domain/style shift).

Environment. Candidates are compiled under a pinned Lean 4/Mathlib toolchain served by Kimina Lean Server (Dos Santos et al., 2025); full details are in Appendix E.3.

Budget. We use a generator-call budget $T = 100$ per problem, debiting each proposal or bounded repair LLM call as one call; repair/semantic-repair attempts are debited as separate calls. Compilation checks, semantic-judge calls, and EvolAST-triggered candidate evaluations are evaluator-side costs. Generator-call counts include seed, proposal, and repair LLM calls; Appendix E.2 audits judge-call overhead and EvolAST-attributed judge calls.

Baselines. Our sampling baselines instantiate a memoryless, no-archive setting: budget-matched variants with fresh sampling/repair and no archive state. They include **Sample** (no repair), **Compile Repair** (bounded compilation repair), and **Compile+Semantic Repair** (compilation repair plus one bounded semantic-repair call for compilable candidates failing the judge). To separate archive-search effects from patch-model strength, Table 1 also reports **Hybrid control**: the same Kimina-generation/Qwen3-repair stack under T=100, but with independently sampled trajectories and no archive state, parent reuse, archive context, usage-penalized selection, or island migration.

Models. In our main runs, the sampling baselines use Kimina-Autoformalizer-7B for sampling/repair, while FormalEvolve uses Kimina-Autoformalizer-7B for seeding and Qwen3-30B-A3B for patch/repair. For proof utility, we fix the prover to Goedel-Prover-V2-32B (Lin et al., 2025) with a fixed prompt template (Appendix F.5).

Search configuration. Unless stated otherwise, FormalEvolve uses $K = 2$ islands, archive capacity 40, parent-selection parameters $\lambda = 10$ and $\beta = 0.05$, an operator mix of full/diff/cross patching with probabilities 0.5/0.3/0.2, and migration every 10 generations at rate 0.1; Appendix E.3 gives the complete configuration.

Metrics. CH@T and SH@T denote the fraction of problems with at least one compilable candidate (resp. at least one compilable, judge-consistent candidate) within T debited generator calls. Coverage alone leaves open whether additional semantic successes are broadly distributed or concentrated on a small set of already-easy problems. Let $s_j(T)$ be the number of generated candidates with $\mathsf{Comp}(c) = 1$ and $\mathsf{Sem}(c) = 1$ for problem j within budget T; we therefore report cross-problem concentration via the Gini coefficient

$$\mathrm{Gini}(T) = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}|s_i(T) - s_j(T)|}{2N\sum_{j=1}^{N} s_j(T) + \varepsilon}, \quad (3)$$

with the same numerical stabilizer $\varepsilon = 10^{-6}$ used in parent-score normalization. We also report the Top-10% share $\sum_{j\in\mathrm{Top}_{10\%}} s_j(T)/\sum_{j=1}^{N} s_j(T)$ (where $\mathrm{Top}_{10\%}$ selects the problems with the largest $s_j(T)$). We evaluate downstream proof utility under a fixed $B = 64$ prover protocol (pass@64 / complete@64 / theorem-complete@64; Table 2); full definitions are in Appendix A.1.

Semantic judge and manual calibration. The semantic predicate Sem is implemented by CriticLean-Qwen3-14B with a fixed prompt and serving configuration; the judge prompt and judge-call accounting are reported in Appendix F.4 and Appendix E.2. Our semantic metrics rely on an LLM-as-judge implementation of Sem; we therefore include manual faithfulness audits for both a statement-stage sample from judge-positive ProofNet outputs and a theorem-complete prover-stage sample produced under the fixed $B = 64$ prover protocol. We summarize these audits in Section 5 and provide the protocol and tables in Appendix C.1.

5 Results

5.1 Coverage and Uniformity

Statement generation at $T = 100$. Table 1 reports statement generation under the same $T = 100$ generator-call budget, including coverage (CH/SH) and concentration (Gini/top-10% share). Against the hybrid no-archive control, which uses Kimina generation with the same Qwen3 repair model but no archive, FormalEvolve increases SH@100 from 0.530 to 0.580 on CombiBench and from 0.828 to 0.849 on ProofNet. We therefore interpret coverage together with cross-problem concentration and downstream proof utility.

Ablations and component effects. The strongest Kimina-only baseline is Compile+Semantic Repair.

Table 1: Statement generation at $T = 100$. CH/SH measure compile/semantic coverage; Gini and top-10% share measure concentration of semantic-success counts (lower is better).

Method	ProofNet (test, $N = 186$) CH@100	SH@100	Gini↓	Top-10%↓	CombiBench ($N = 100$) CH@100	SH@100	Gini↓	Top-10%↓
Baselines (Kimina; no archive)								
Sample	0.903	0.715	0.537	0.243	1.000	0.440	0.816	0.637
Compile Repair	0.909	0.720	0.566	0.263	0.990	0.400	0.825	0.641
Compile+Semantic Repair	0.909	0.780	0.555	0.264	0.990	0.460	0.813	0.609
Baselines (Qwen3; no archive)								
Qwen3 Sample	0.387	0.242	0.896	0.876	0.300	0.200	0.939	0.952
Qwen3 + Compile Repair	0.371	0.237	0.912	0.911	0.900	0.260	0.740	0.385
Qwen3 + Compile+Semantic Repair	0.688	0.548	0.802	0.656	0.940	0.390	0.827	0.655
Hybrid control (no archive)								
Hybrid control	0.973	0.828	0.505	0.241	1.000	0.530	0.790	0.588
FormalEvolve (ours)								
FormalEvolve (K=2)	0.973	0.849	0.443	0.229	1.000	**0.580**	0.759	0.531
FormalEvolve (K=1)	**0.984**	0.866	**0.362**	**0.198**	1.000	0.550	**0.726**	**0.442**
FormalEvolve (K=2; w/o EvolAST)	0.973	**0.871**	0.454	0.236	1.000	0.500	0.776	0.577
FormalEvolve (K=2; w/o patch repair)	0.903	0.780	0.449	0.211	1.000	0.470	0.772	0.494

Adding the stronger Qwen3 repair model without an archive already improves SH@100 from 0.460 to 0.530 on CombiBench and from 0.780 to 0.828 on ProofNet. FormalEvolve then adds archive-based search on top of this repair strength, so the hybrid row can be read as a matched no-archive ablation. Removing patch repair drops SH@100 from 0.580 to 0.470 on CombiBench and from 0.849 to 0.780 on ProofNet, identifying bounded repair as the largest direct driver of SH@100. The archive then makes repaired candidates reusable as parents and prompt context, allowing local repair gains to propagate through later search steps while reducing repeated edits of the same templates. The island and EvolAST ablations are smaller and dataset-dependent under the fixed budget: K=1 removes only island separation, remaining an archive-based single-population search, and it improves ProofNet coverage and concentration as well as CombiBench concentration. The default K=2 setting gives the strongest CombiBench SH@100 among FormalEvolve variants and is used for downstream proof evaluation. Accordingly, the island rows are reported as implementation sensitivity; the main controlled comparison remains archive-based search versus matched no-archive controls.

Cross-problem concentration. Coverage alone can hide a failure mode in which a method spends most of its successful calls on the same easy problems. We therefore report Gini and top-10% share over per-problem `semantic_ok` counts. The default FormalEvolve configuration lowers both concentration metrics relative to the hybrid control: on CombiBench, Gini decreases from 0.790 to 0.759 and top-10% share from 0.588 to 0.531; on ProofNet, the corresponding changes are 0.505 to 0.443 and 0.241 to 0.229. Bootstrap intervals over problems show that the Gini reductions are stable for the matched hybrid comparison (+0.031 [0.010, 0.055] on CombiBench; +0.062 [0.035, 0.090] on ProofNet), while top-10% reductions have the same sign but wider intervals (Appendix B.1). Figures 6 and 8 show the same pattern at the problem level and across budgets.

5.2 Downstream Proof Utility

Prover protocol. Goedel-Prover-V2-32B receives up to 64 deduplicated candidates per problem from the $T = 100$ semantically accepted repertoire; empty repertoires count as failures. The single-reference GT control uses the dataset reference statement. Table 2 reports solved-problem counts for pass@64, complete@64, and theorem-complete@64. The prover sees only candidates with `compile_ok`= 1 and `semantic_ok`= 1, deduplicated under canonicalization and capped at 64 statements per problem. The GT reference row runs the same prover on the dataset-provided formal statement and bypasses both generator and semantic judge.

Theorem-complete gains and boundaries. On CombiBench, FormalEvolve raises theorem-complete@64 from 9/100 for the matched hybrid no-archive control to 13/100, and from 8/100 for both Sample and Kimina Compile+Semantic Repair. On ProofNet, FormalEvolve also improves over the matched hybrid control on theorem-complete@64 (45/186 vs. 40/186), while remaining comparable to Kimina Compile+Semantic Repair (45/186 vs. 46/186). Paired bootstrap intervals

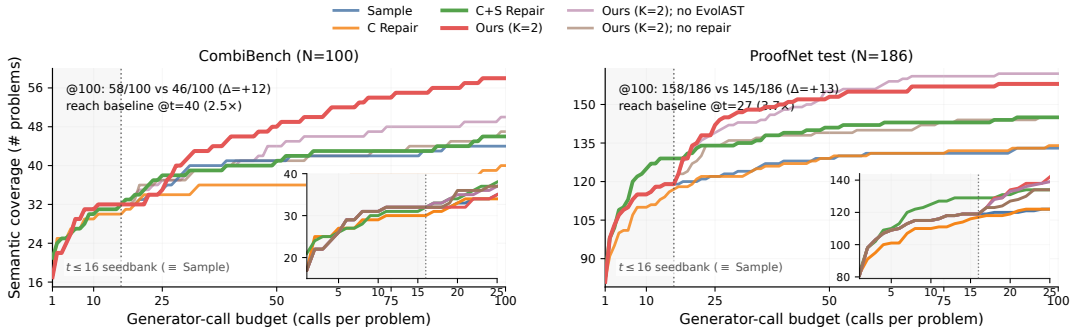

Figure 3: Semantic coverage vs debited generator calls t on both benchmarks. The dotted line marks the nominal seedbank boundary (t=16).

Table 2: Downstream proof utility at $B = 64$ prover attempts per problem. Theorem-complete counts complete proofs of explicit `theorem`/`lemma` declarations.

Method	CombiBench ($N = 100$) pass@64	complete@64	thm-complete@64	ProofNet (test, $N = 186$) pass@64	complete@64	thm-complete@64
Ours (K=2)	44/100	27/100	13/100	127/186	52/186	45/186
Hybrid control	40/100	22/100	9/100	110/186	49/186	40/186
Sample	41/100	23/100	8/100	106/186	46/186	41/186
C+S Repair	40/100	23/100	8/100	119/186	50/186	46/186
GT reference	96/100	68/100	18/100	146/186	46/186	34/186

for the matched-hybrid theorem-complete deltas are above zero (+0.040 [0.033, 0.110] on CombiBench; +0.027 [0.021, 0.043] on ProofNet; Appendix B.1), supporting the hedging motivation: diverse repertoires increase the chance of reaching a prover-friendly accepted statement under a fixed prover budget. Appendix A.2 and Appendix C.2 provide additional curves and audit details.

5.3 Stronger-Base and Manual Audits

Stronger-base statement-generation. As a supplementary CombiBench statement-generation check, we use Goedel-Formalizer-V2-8B seeding and GPT-5.4 repair with the same judge and T=100 accounting. FormalEvolve preserves the pattern over the matched hybrid control: SH@100 rises from 0.86 to 0.88, deduplicated $\text{SemOK}_{\text{total}}$ rises from 1886 to 3238 (1.7×), and concentration drops in both Gini (0.4832 to 0.4383) and top-10% share (0.3128 to 0.2202) (Appendix B.3).

Manual faithfulness calibration. Manual audits calibrate selected statement-stage and theorem-complete slices. On a matched 50-problem ProofNet statement-stage audit, FormalEvolve, Hybrid control, and Sample have similar Faithful/Partial/Severe totals: 42/5/3, 43/4/3, and 43/3/4. In this slice, archive search does not visibly amplify judge-positive semantic shift relative to the no-archive controls. At the theorem-complete end, a 161-artifact audit over the original three-method pool shows that FormalEvolve yields the largest absolute count of faithful artifacts (41), while severe counts are numerically similar across methods (11, 10, and 12). Proof success certifies the encoded formal statement, but not its faithfulness to the source statement; we therefore report proof success together with semantic faithfulness auditing (Appendix C.1).

6 Conclusion

FormalEvolve is a compile-gated test-time search method for Lean 4 autoformalization under a fixed generator-call budget. The central lesson is that autoformalization should be evaluated at the level of budgeted, semantically filtered repertoires: faithful variants of the same informal claim can expose different prover-facing interfaces. FormalEvolve operationalizes this view with a compilation-feasible archive that stores reusable candidates, expands them through repair and evolution, and filters the resulting repertoire for downstream proving. Across two benchmarks, this archive-based organization improves semantic coverage, reduces cross-problem concentration, and yields theorem-complete gains over matched no-archive multi-candidate controls under our fixed prover protocol. The same archive-and-filter pattern may also extend to Coq, Isabelle, and related proof assistants, where multiple faithful formalizations can likewise differ in downstream prover utility.

Limitations

Semantic consistency remains difficult to certify automatically, which is a central challenge for autoformalization. Our semantic metrics rely on an LLM-as-judge implementation of Sem, so results that use semantic labels–including semantic coverage, semantic-success counts, and concentration statistics–can depend on the judge model, prompt, and serving details. Compilation certifies Lean elaboration and type correctness, while semantic equivalence to the informal problem remains outside the compiler's guarantee. We therefore report manual faithfulness audits over outputs produced by our pipeline. On the matched ProofNet statement-stage slice, FormalEvolve, Hybrid control, and Sample have similar Faithful/Partial/Severe totals (42/5/3, 43/4/3, and 43/3/4), suggesting that archive search does not visibly amplify judge-positive semantic shift relative to no-archive controls.

Proof utility also depends on the chosen prover, proof budget, and prompting details. Our pass@64, complete@64, and theorem-complete@64 results use one fixed prover configuration with $B = 64$ attempts, so they measure utility under this protocol; broader gains across other provers and proof-search policies require separate evaluation.

References

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. 2023. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *Preprint*, arXiv:2302.12433.

Guoxin Chen, Jing Wu, Xinjie Chen, Wayne Xin Zhao, Ruihua Song, Chengxi Li, Kai Fan, Dayiheng Liu, and Minpeng Liao. 2025. ReForm: Reflective autoformalization with prospective bounded sequence optimization. *Preprint*, arXiv:2510.24592. ArXiv preprint.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

Marco Dos Santos, Hugues de Saxcé, Haiming Wang, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. 2025. Kimina lean server: A high-performance lean server for large-scale verification. *Preprint*, arXiv:2504.21230.

Qi Guo, Jianing Wang, Jianfei Zhang, Deyang Kong, Xiangzhou Huang, Xiangyu Xi, Wei Wang, Jingang Wang, Xunliang Cai, Shikun Zhang, and Wei Ye. 2025. Autoformalizer with tool feedback. *Preprint*, arXiv:2510.06857.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. 2022. Proof artifact co-training for theorem proving with language models. In *The Tenth International Conference on Learning Representations (ICLR 2022)*. OpenReview.net.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *Preprint*, arXiv:2509.19349. ArXiv preprint.

Joel Lehman and Kenneth O. Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223.

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. 2025. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *Preprint*, arXiv:2508.03613.

Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. LLM4AD: A platform for algorithm design with large language model. *Preprint*, arXiv:2412.17287.

Junqi Liu, Xiaohan Lin, Jonas Bayer, Yael Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, Zhengfeng Yang, Jujian Zhang, Lihong Zhi, Jia Li, and Zhengying Liu. 2025. CombiBench: Benchmarking LLM capability for combinatorial mathematics. *Preprint*, arXiv:2505.03171.

Jianqiao Lu, Yingjia Wan, Yinya Huang, Jing Xiong, Zhengying Liu, and Zhijiang Guo. 2024. FormalAlign: Automated alignment evaluation for autoformalization. *Preprint*, arXiv:2410.10135.

Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *Preprint*, arXiv:1504.04909.

Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *Preprint*, arXiv:2506.13131.

Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, Minghao Liu, Yihang Xia, Jiawei Shen, Yuchen Wu, Yixin Cao, Zhaoxiang Zhang, Wenhao Huang, Jiaheng Liu, and

Ge Zhang. 2025. CriticLean: Critic-guided reinforcement learning for mathematical formalization. *Preprint*, arXiv:2507.06181.

Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *Preprint*, arXiv:2009.03393.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2024. Lean copilot: Large language models as copilots for theorem proving in lean. *Preprint*, arXiv:2404.12534.

Yuchen Tian, Ruiyuan Huang, Xuanwu Wang, Jing Ma, Zengfeng Huang, Ziyang Luo, Hongzhan Lin, Da Zheng, and Lun Du. 2025. EvolProver: Advancing automated theorem proving by evolving formalized problems via symmetry and difficulty. *Preprint*, arXiv:2510.00732.

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

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. In *Advances in Neural Information Processing Systems*.

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

Haoyu Zhao, Yihan Geng, Shange Tang, Yong Lin, Bohan Lyu, Hongzhou Lin, Chi Jin, and Sanjeev Arora. 2025. Ineq-Comp: Benchmarking human-intuitive compositional reasoning in automated theorem proving on inequalities. *Preprint*, arXiv:2505.12680.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. miniF2F: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations (ICLR 2022)*. OpenReview.net.

A Additional Results

We give the metric definitions used throughout the experiments, followed by statement-level decompositions, proof-utility decomposition, and budget-sweep coverage gaps under the same generator-call accounting.

A.1 Metric definitions

We provide definitions of all reporting metrics under generator-call accounting. For a given problem, let E_t denote the set of candidates evaluated up to debited call t, and let $E_t^{\text{feas}} = \{c \in E_t : \mathsf{Comp}(c) = 1\}$. Let $G_t = \{c \in E_t : \mathsf{Comp}(c) = 1 \wedge \mathsf{Sem}(c) = 1\}$, and for dataset-level concentration let $s_j(t)$ be the number of semantic-success candidates for problem j at budget t. For proof utility, let $S_t^{(64)}$ denote the first 64 statements in $\tilde{G}_t = \mathrm{Dedup}(G_t)$ used by the fixed prover protocol. Let $P_{\text{pass}}(c) \in \{0, 1\}$ indicate whether the prover returns a Lean 4-accepted proof script for c (warnings allowed; `sorry` permitted), $P_{\text{complete}}(c) \in \{0, 1\}$ indicate a complete proof without `sorry`, and $P_{\text{theorem}}(c) \in \{0, 1\}$ indicate a complete proof of an explicit `theorem`/`lemma`.

Table 3: Metric definitions under generator-call accounting.

Metric	Definition and interpretation
$\mathrm{FY}(t)$	$\frac{1}{\lvert E_t \rvert} \sum_{c \in E_t} \mathsf{Comp}(c)$; fraction of debited calls yielding compilable candidates.
$\mathrm{CH}(t)$	$I[\exists c \in E_t : \mathsf{Comp}(c) = 1]$; at least one compilable candidate.
$\mathrm{SH}(t)$	$I[\exists c \in E_t : \mathsf{Comp}(c) = 1 \wedge \mathsf{Sem}(c) = 1]$; at least one compilable and judge-consistent candidate.
$\mathrm{SD}(t)$	$\frac{1}{\lvert E_t^{\text{feas}} \rvert} \sum_{c \in E_t^{\text{feas}}} \mathsf{Sem}(c)$, or 0 if E_t^{feas} is empty; semantic density among compilable candidates.
$\mathrm{SY}(t)$	$\frac{1}{\lvert E_t \rvert} \sum_{c \in E_t} \mathsf{Sem}(c)$; semantic yield per call, treating non-compilable candidates as $\mathsf{Sem} = 0$.
$\mathrm{Div}(t)$	$\lvert \mathrm{Dedup}(G_t) \rvert$; deduplicated semantic successes.
$\mathrm{Cov}(t)$	$\frac{1}{N} \sum_{j=1}^{N} I[s_j(t) \geq 1]$; dataset-level semantic coverage.
$\mathrm{Gini}(t)$	$\frac{\sum_{i,j} \lvert s_i(t) - s_j(t) \rvert}{2N \sum_j s_j(t) + \varepsilon}$; concentration of semantic successes (↓).
Top-10%	$\frac{\sum_{j \in \mathrm{Top}_{10\%}} s_j(t)}{\sum_j s_j(t)}$; success mass assigned to the easiest 10% problems (↓).
PU-pass@64	At least one Lean 4-accepted proof script among the first 64 deduplicated semantic candidates.
PU-complete@64	At least one proof without `sorry`.
PU-theorem@64	At least one complete proof of an explicit `theorem`/`lemma`.

Each debited generator call yields one evaluated representative candidate; repair and semantic-repair calls count as additional calls. Dedup uses conservative whitespace/name canonicalization.

A.2 Proof utility decomposition

Figure 4 decomposes theorem-complete@64 into (i) whether a method produces a non-empty statement repertoire to attempt under the fixed prover protocol, and (ii) whether at least one attempted statement is solved at theorem-complete level under the same attempt budget.

A.3 Statement-level decompositions at $T = 100$

Tables 4 and 5 decompose the fixed-budget statement-level results. FY is compilation yield, SD is semantic density conditional on compilation, and SY is semantic yield per generator call (treating compilation failures as semantic failures). Reporting FY/SH/SD/SY together separates gains from (i) producing more compilable candidates, (ii) improving semantic consistency among compilable candidates, and (iii) reallocating budget between proposal and repair.

Table 4: CombiBench statement-level results at $T = 100$.

Method	FY	SH	SD	SY
Sample	0.695	0.440	0.189	0.132
Compile Repair	0.555	0.400	0.199	0.110
Compile+Semantic Repair	0.324	0.460	0.371	0.120
FormalEvolve (K=2)	0.407	0.580	0.326	0.133
FormalEvolve (w/o EvolAST fallback)	0.346	0.500	0.313	0.108
FormalEvolve (w/o patch repair)	0.682	0.470	0.250	0.170

Table 5: ProofNet statement-level results at $T = 100$.

Method	FY	SH	SD	SY
Sample	0.702	0.715	0.599	0.421
Compile Repair	0.625	0.720	0.622	0.389
Compile+Semantic Repair	0.509	0.780	0.760	0.387
FormalEvolve (K=2)	0.562	0.849	0.775	0.435
FormalEvolve (w/o EvolAST fallback)	0.539	0.871	0.788	0.425
FormalEvolve (w/o patch repair)	0.694	0.780	0.684	0.475

A.4 Coverage gaps across the budget

Figure 5 reports the budget-sweep coverage gap relative to the strongest Kimina sampling baseline. The hybrid Kimina+Qwen3 baseline in Table 1 separates search effects from patch-model strength under the same generator-call accounting and repair protocol as the sampling baselines.

B Analysis

We provide additional measurements for the fixed-budget experiments: paired uncertainty, repair-call composition, prover-stage non-dominance, cross-problem concentration, stronger-base behavior, and component effects.

B.1 Paired uncertainty, call composition, and prover non-dominance

The measurements below are computed from the same fixed-budget artifacts used for the main figures and tables; no additional training or model runs are introduced. They cover paired uncertainty (Table 6), concentration uncertainty for the matched hybrid comparison (Table 7), per-problem repair-call composition (Table 8), and prover-stage non-dominance (Table 9).

Table 6: Paired uncertainty over problems for the matched Hybrid no-archive comparison. Hybrid denotes the matched Kimina-generation/Qwen3-repair no-archive control used in Table 1 and Table 2.

Dataset	Metric	mean(FE)	mean(base)	Δ [95% CI]
CombiBench	SH@100 vs Hybrid	0.580	0.530	+0.050 [0.003, 0.130]
ProofNet	SH@100 vs Hybrid	0.849	0.828	+0.022 [0.005, 0.048]
CombiBench	theorem-complete@64 vs Hybrid	0.130	0.090	+0.040 [0.033, 0.110]
ProofNet	theorem-complete@64 vs Hybrid	0.242	0.215	+0.027 [0.021, 0.043]

We use paired bootstrap 95% confidence intervals for the mean difference (FormalEvolve − base-

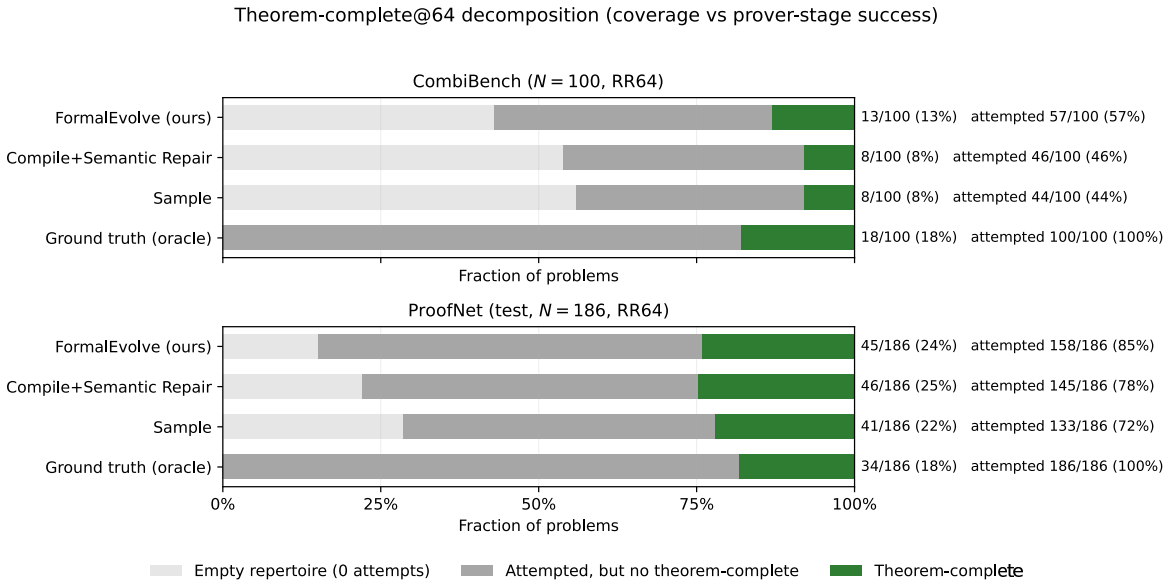

Figure 4: Decomposing theorem-complete@64 under a fixed prover and attempt budget (B=64). Each bar splits the benchmark denominator into problems with an empty statement repertoire for proving (zero prover attempts), problems with a non-empty repertoire but no theorem-complete success, and problems with at least one theorem-complete success.

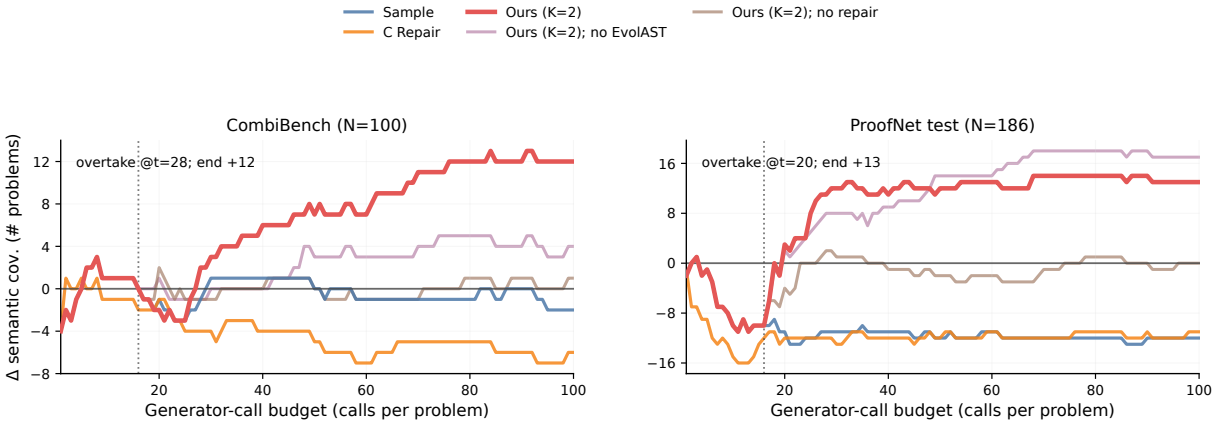

Figure 5: Coverage gap vs debited calls, plotted as Δ semantic coverage relative to the strongest Kimina sampling baseline (Compile+Semantic Repair). The zero line corresponds to the baseline; positive values indicate a method covers more problems at the same generator-call budget. The dotted vertical line marks the nominal seedbank boundary (t=16) used by evolution-based methods (pre-sampled seeds, debited calls); a small subset of problems can require additional debited seeds before evolution starts.

line).

Table 7: Bootstrap uncertainty for concentration reductions relative to the matched hybrid no-archive control. Positive Δ means the hybrid control is more concentrated than FormalEvolve, so lower concentration is better for FormalEvolve.

Dataset	Metric	FE	Hybrid	Δ (Hybrid − FE)	95% CI
CombiBench	Gini	0.759	0.790	+0.031	[0.010, 0.055]
CombiBench	Top-10% share	0.531	0.588	+0.057	[-0.009, 0.128]
ProofNet	Gini	0.443	0.505	+0.062	[0.035, 0.090]
ProofNet	Top-10% share	0.229	0.241	+0.011	[-0.005, 0.032]

We compute the intervals by resampling benchmark problems with replacement and recomputing each concentration metric on the resampled per-problem semantic-success counts.

Table 8: Debited repair-call composition under $T = 100$.

Dataset	Method	CRep (median [q25,q75])	SRep (median [q25,q75])
CombiBench	Sample	0.0 [0.0, 0.0]	0.0 [0.0, 0.0]
CombiBench	Compile Repair	29.0 [4.0, 52.0]	0.0 [0.0, 0.0]
CombiBench	Compile+Semantic Repair	14.0 [4.0, 34.5]	38.0 [26.0, 54.0]
CombiBench	Hybrid control	15.5 [2.8, 33.0]	43.5 [25.8, 57.0]
CombiBench	FormalEvolve (K=2)	30.5 [20.8, 39.0]	20.5 [14.0, 26.0]
ProofNet	Sample	0.0 [0.0, 0.0]	0.0 [0.0, 0.0]
ProofNet	Compile Repair	16.0 [0.0, 53.0]	0.0 [0.0, 0.0]
ProofNet	Compile+Semantic Repair	9.0 [0.0, 39.8]	11.5 [0.0, 37.8]
ProofNet	Hybrid control	6.0 [0.0, 29.0]	10.0 [0.0, 32.8]
ProofNet	FormalEvolve (K=2)	24.0 [8.0, 40.8]	4.0 [0.0, 16.0]

Table 9: Prover-stage non-dominance at theorem-complete@64.

Dataset	n	both fail	FE only	baseline only	both succeed
CombiBench	100	85	7	2	6
ProofNet	186	129	11	12	34

B.2 Cross-problem uniformity: coverage and concentration

We assess whether semantic successes concentrate on a small subset of easy problems under a fixed call budget. At $T = 100$, we summarize the per-problem `semantic_ok` count distribution using coverage (SH@100) and concentration (Gini and top-10% share; Table 1). Figure 6 provides a compact per-problem view. For readability, we filter to problems where at least one method attains a positive `semantic_ok` count (otherwise the column is identically zero). We order columns by the strongest Kimina sampling baseline (Compile+Semantic Repair) using its per-problem `semantic_ok` counts at $T = 100$, defining an x-axis independent of ours; Figure 7 uses the same ordering for the delta visualization. This subsection uses Compile+Semantic Repair only as a fixed ordering and visualization baseline; the matched Hybrid comparisons are reported separately in Table 6 and in the main results.

Aggregate pattern under baseline ordering. Under this baseline ordering, gains concentrate in the baseline-ranked tail: on the filtered set, $\Delta > 0$ dominates in the tail segment, while the head segment can favor the baseline (especially on ProofNet). We define head/mid/tail by splitting the baseline-ordered problems into three contiguous segments as evenly as possible within each benchmark. The table below counts how often $\Delta = \#\texttt{semantic_ok}(\text{FormalEvolve}) - \#\texttt{semantic_ok}(\text{Compile+Semantic Repair})$ is positive/negative/zero within each segment (reported as $+/-/0$).

Dataset	head ($+/-/0$)	mid ($+/-/0$)	tail ($+/-/0$)
CombiBench (filtered 64/100)	8/12/1	12/7/2	18/2/2
ProofNet (filtered 161/186)	6/41/6	35/19/0	49/5/0

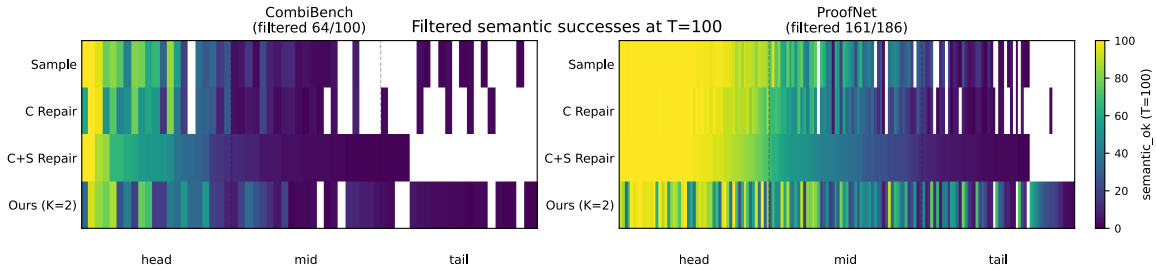

Figure 6: Filtered per-problem semantic-success counts at $T = 100$ (`semantic_ok`; 0 shown as white).

Table 10: Representative early semantic hits at a generator-call budget of $T = 100$. We include both directions (FormalEvolve earlier / baseline no-hit, and baseline-hit / FormalEvolve no-hit). "> 100" indicates no semantic hit within the call budget.

Benchmark	Problem id	Short description	Ours first hit	C+S Repair first hit
FormalEvolve hits earlier, or baseline has no hit within $T = 100$				
CombiBench	`0055_egmo_2022_p5`	Domino tilings: parity of $f(n, 2k)$ for all k	1	> 100
CombiBench	`0063_usamo_2000_p4`	3 colored squares form an axis-aligned right triangle	4	> 100
CombiBench	`0077_imo_2010_p5`	Box/coin operations reach $2010^{2010^{2010}}$?	4	> 100
CombiBench	`0035_brualdi_ch9_8`	Count SDRs for six 2-sets arranged in a cycle	21	> 100
CombiBench	`0011_brualdi_ch1_10`	No magic square of order 2	22	93
CombiBench	`0047_brualdi_ch13_10`	Tournament: a vertex reaches all others within 2 steps	27	95
CombiBench	`0057_imosl_2015_c6`	Infinitely many integers without a unique odd-sum representation	2	31
CombiBench	`0065_imo_2020_p3`	Split weighted colored pebbles into two equal piles	27	52
ProofNet	`0046_exercise_3_4_5b`	Quotients of solvable groups are solvable	19	> 100
ProofNet	`0124_exercise_26_12`	Perfect map: compact Y implies compact X	21	> 100
ProofNet	`0163_exercise_3_13`	Cauchy product of absolutely convergent series	25	> 100
ProofNet	`0119_exercise_23_3`	Connected union with shared intersection	18	86
ProofNet	`0095_exercise_5_6_14`	Distinct roots of $x^m - x$ in characteristic p	24	91
ProofNet	`0045_exercise_3_4_4`	Finite abelian groups have subgroups of each divisor order	12	62
ProofNet	`0005_exercise_6_4_12`	No simple group of order 224	6	45
Baseline hits, but FormalEvolve has no hit within $T = 100$				
CombiBench	`0037_brualdi_ch10_31`	Difference set $B = \{0, 3, 4, 9, 11\}$ in $\mathbb{Z}_{21}$	> 100	22
CombiBench	`0014_brualdi_ch2_36`	Counting combinations of multisets with bounded repetitions	> 100	39
CombiBench	`0068_imo_2000_p4`	Distribute cards into 3 boxes so sums are equal	> 100	50
CombiBench	`0021_brualdi_ch4_9`	Max inversions in a permutation of $\{1, \ldots, n\}$	> 100	85
ProofNet	`0185_exercise_5_1`	Blaschke condition for zeros of bounded holomorphic functions	> 100	3
ProofNet	`0009_exercise_11_2_13`	Divisibility in Gaussian integers implies divisibility in $\mathbb{Z}$	> 100	40
ProofNet	`0094_exercise_5_4_3`	A root of a polynomial with radicals is algebraic	> 100	49

B.2.1 Representative early semantic hits and baseline-miss instances

On each benchmark, there are 16 instances where Compile+Semantic Repair has zero `semantic_ok` at T=100 while FormalEvolve attains a positive `semantic_ok` count. Conversely, there are 4 such instances where the baseline hits but FormalEvolve misses on CombiBench, and 3 on ProofNet. To ground the aggregate patterns above and illustrate non-dominance, Table 10 lists representative instances in both directions: where FormalEvolve reaches a semantically consistent compilable statement substantially earlier than the baseline (or the baseline has no hit within $T = 100$ calls), and where the baseline hits but FormalEvolve misses within the same call budget. These examples are computed from the debited-call logs under the same budget.

Across both benchmarks, FormalEvolve improves coverage and reduces concentration (see Table 1 and Figures 6–8).

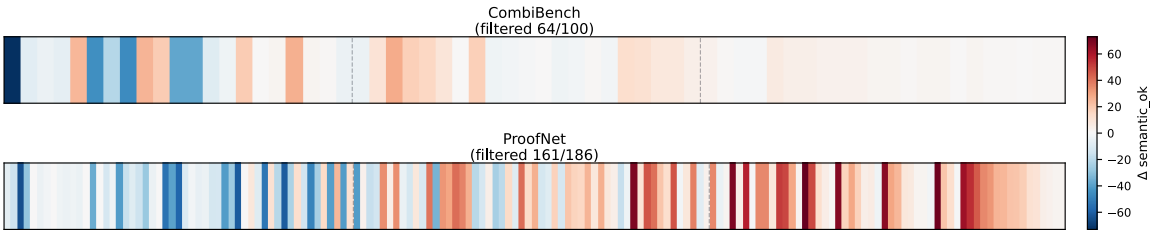

Figure 7: Delta strips at $T = 100$ under fixed-budget accounting on the same filtered set and in the same baseline ordering (sorted by Compile+Semantic Repair `semantic_ok`) as Figure 6, showing $\Delta = \#\texttt{semantic_ok}(\textsc{FormalEvolve}) - \#\texttt{semantic_ok}(\textsc{Compile+Semantic Repair})$ per problem.

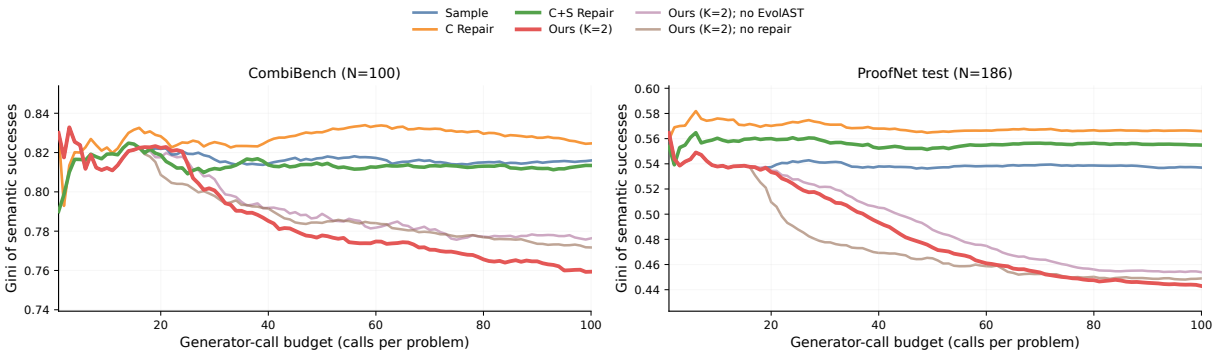

Figure 8: Cross-problem concentration of semantic successes vs generator-call budget, summarized by the Gini coefficient of per-problem semantic-success counts under fixed-budget accounting. Lower is more uniform (less concentrated on a small subset of problems). Table 1 reports the corresponding summary at $T = 100$.

B.3 Stronger-base statement-generation evaluation on CombiBench

Setup. The primary comparison uses a Kimina-based stack to keep the core comparison controlled under the same generator-call budget. To test whether the search layer still helps once the base stack is substantially stronger, we run a statement-generation evaluation on CombiBench under the same T=100 accounting with Goedel-Formalizer-V2-8B as the seeder and GPT-5.4 as the repair model. We keep the semantic judge and statement-stage reporting protocol unchanged.

Compared methods. SAMPLE uses Goedel-Formalizer-V2-8B alone, without repair or archive search. COMPILE+SEMANTIC REPAIR keeps the same stronger seeder and applies bounded compile / semantic repair without an archive. HYBRID CONTROL (NO ARCHIVE) keeps the same stronger seeder and stronger repair stack, with archive-based search disabled. **FORMALEVOLVE** then adds archive-based search on top of this no-archive control.

Table 11: Stronger-base statement-generation evaluation on CombiBench (N=100, T=100). $\text{SemOK}_{\text{total}}$ is the deduplicated aggregate count of judge-consistent compiled candidates across problems.

Method	CH@100	SH@100	$\text{SemOK}_{\text{total}}$	Gini↓	Top-10%↓
Sample	0.95	0.82	1478	0.5852	0.3410
Compile+Semantic Repair	0.93	0.85	1443	0.5693	0.3389
Hybrid control (no archive)	**0.98**	0.86	1886	0.4832	0.3128
FormalEvolve	0.95	**0.88**	**3238**	**0.4383**	**0.2202**

Interpretation. The stronger-base statement-generation setting preserves the main qualitative pattern. Relative to the matched no-archive hybrid control, archive-based search still improves SH@100 (0.86 → 0.88), increases deduplicated $\text{SemOK}_{\text{total}}$ by 1.7× (1886 → 3238), and further reduces concentration (Gini 0.4832 → 0.4383; Top-10% share 0.3128 → 0.2202). The search layer both expands the total pool of judge-accepted candidates and makes that pool less concentrated.

B.4 What drives gains: repair versus online diversity

Under fixed budgets, bounded patch/repair calls are the dominant contributor to semantic hit rate: removing patch repair reduces SH@100 by 11/100 (CombiBench) and 13/186 (ProofNet) (Table 1). EvolAST-style fallback has a smaller and non-monotonic effect: relative to the no-EvolAST ablation, the run with EvolAST gains semantic hits on 11 CombiBench problems and loses them on

3, while on ProofNet it gains 1 and loses 5. Because `semantic_ok` is assigned by the LLM-as-judge implementation of Sem, these differences should be read as changes in the distribution of judge-accepted candidates rather than guaranteed semantic improvements.

Illustrative instances (EvolAST can help or hurt). The examples below show one instance where EvolAST enables a semantic hit and one where it suppresses a hit under the same $T=100$ budget.

Problem	Dataset	first hit call (w/ EvolAST / w/o)	semantic_ok (w/ EvolAST / w/o)	Δ semantic_ok
`0011_brualdi_ch1_10`	CombiBench	22 / no hit	12 / 0	+12
`0014_brualdi_ch2_36`	CombiBench	no hit / 48	0 / 3	-3

B.5 Failure modes

We log (i) compilation failures, (ii) semantic drift (judge rejects), and (iii) judge/prover disagreements for qualitative inspection. The qualitative cases in Appendix D show representative instances of these disagreements.

Example (judge/prover mismatch). In Appendix D.1.3 (ProofNet `0004_exercise_6_4_2`), both baselines obtain theorem-complete prover outputs under the same RR64 budget, while our best judge-accepted statement is only pass-level. This illustrates non-dominance between semantic judging and prover utility: semantic acceptance and proof success provide different evidence under fixed-budget evaluation.

C Human and Prover-Stage Audits

C.1 Manual faithfulness audits

Audit overview. We report two complementary manual audits with the same *Faithful / Partial / Severe* label set. The first audit targets theorem-complete artifacts from the three original generator-based methods at the solved end of the pipeline. The second targets the main statement-stage setting with a matched ProofNet audit over FormalEvolve, the hybrid no-archive control, and Sample.

Theorem-complete audit: scope and audit unit. We audit the theorem-complete end of the original three-method comparison pool. The pool contains 161 theorem-complete artifacts across SAMPLE, COMPILE+SEMANTIC REPAIR, and FORMALEVOLVE: 29 from CombiBench and 132 from ProofNet. The audit unit is one solved *benchmark* × *method* × *problem* cell. If a solved cell contains multiple theorem-complete attempts, we take the first theorem-complete attempt in `per_problem.json` order and trace it back to the corresponding source statement in `rr64_dataset.jsonl`.

Labels. *Faithful* means that the proved formal statement matches the mathematical content of the informal target. *Partial* denotes a materially narrowed, incomplete, or otherwise only partially aligned statement. *Severe* denotes wrong-object, vacuous, or structurally broken cases that do not preserve the intended mathematical claim.

Table 12: Pool-level totals for the manual faithfulness audit over theorem-complete artifacts in the original three-method comparison pool.

Pool	Faithful	Partial	Severe
CombiBench (29)	18	4	7
ProofNet (132)	93	13	26
Total (161)	111	17	33

Table 13: Method-level totals for the same manual faithfulness audit.

Method	Faithful	Partial	Severe
FORMALEVOLVE	41	6	11
SAMPLE	34	5	10
COMPILE+SEMANTIC REPAIR	36	6	12

Multi-reader cross-check. On a multi-reader cross-check subset drawn from the manually audited portion, the initial agreement rate on the *Faithful / Partial / Severe* labels is 92% before adjudication. Disagreements were rechecked jointly, and uncertain cases were resolved conservatively.

Aggregate pattern. Within this audited theorem-complete pool, FORMALEVOLVE yields the largest absolute count of *Faithful* artifacts (41), while *Severe* counts are numerically similar across methods (11, 10, and 12). Proof success certifies the formal statement that was actually encoded and proved, but does not by itself establish faithfulness to the source statement; all three audited methods still contain severe mismatches at the solved end of the pipeline. The audit therefore complements the main fixed-budget results with a direct theorem-faithfulness view at the solved end of the pipeline.

Matched statement-stage audit. Because the statement-stage semantic metrics rely on judge-positive outputs, we also audit matched judge-positive statement-stage outputs on ProofNet. We sample 50 problems for which FormalEvolve, the hybrid no-archive control, and Sample each have a late-window compile+semantic judge-positive witness. For each method and sampled problem, the audit unit is the latest deduplicated judge-positive

candidate with insertion rank in $[20, 100]$. This design compares faithfulness profiles on a shared judge-positive slice; coverage is reported separately by SH@100.

Table 14: Matched manual statement-stage audit on late-window judge-positive ProofNet outputs. The same 50 problems are audited for all three methods.

Method	Faithful	Partial	Severe
FormalEvolve	42	5	3
Hybrid control	43	4	3
Sample	43	3	4

Interpretation. Under final adjudication, the matched statement-stage audit shows similar faithfulness profiles across the three methods: FormalEvolve has 42/50 *Faithful* candidates, Hybrid control has 43/50, and Sample has 43/50. Severe counts are also close (3/50, 3/50, and 4/50 respectively). In this slice, FormalEvolve's judge-positive outputs have a faithfulness profile similar to the two no-archive controls, with no obvious additional semantic shift from archive search.

C.2 Prover-stage audit: what proof success certifies

Scope and claim boundary. We audit the prover stage on the subset of judge-accepted statements produced within the generator-call budget (Section 4). A prover-complete Lean file certifies that the produced formal artifact is accepted by Lean and contains no remaining `sorry`. Faithfulness to the original informal statement is assessed by the LLM-as-judge implementation of Sem and checked with manual audits (Appendix C.1).

Prompt reference. The prover prompt template is fixed (Appendix F.5); the case studies report the proved formal statement and the prover response.

ASCII normalization. For compatibility with pdfLaTeX, raw transcripts in `PromptBox` are displayed after ASCII-only normalization of common Unicode math symbols (e.g., rendering quantifiers and connectives in ASCII form). This affects presentation only.

Theorem-complete reporting. In addition to standard pass@64 and complete@64, we report *theorem-complete@64*: a problem counts as theorem-complete@64 if at least one prover attempt yields a Lean-accepted, `sorry`-free file that contains an explicit `theorem` or `lemma`. This guards against Lean-complete outputs that only introduce auxiliary `def`/`abbrev`/`example` content without proving a named proposition.

Prompt categories. Table 15 also reports a coarse prompt-type audit for the audited generator-based methods *over prover attempts* (not over problems): the denominator is the number of prover prompts actually issued under a fixed per-problem attempt budget. We label an attempt as "Prompt has theorem" if the candidate statement sent to the prover contains an explicit `theorem`/`lemma` declaration. We label an attempt as "Prompt abbrev-only" if the candidate contains an `abbrev` declaration and no `theorem`/`lemma`. Remaining attempts typically correspond to `def` or `example` templates and are omitted from this summary.

Table 15: Prover-stage audit statistics under B=64 prover attempts per problem for the audited generator-based methods. Prompt-category counts are reported *per prover attempt*, not per problem. We report (i) whether the prover prompt contains a `theorem/lemma` declaration vs. abbrev-only prompts, and (ii) pass@64/complete@64 together with theorem-complete@64.

Benchmark	Method	Prompt has theorem	Prompt abbrev-only	pass@64	complete@64	theorem-complete@64
CombiBench	Ours	2299/3648 (0.630)	384/3648 (0.105)	44/100	27/100	13/100
CombiBench	Strong	1813/2944 (0.616)	384/2944 (0.130)	40/100	23/100	8/100
CombiBench	Sample	1792/2816 (0.636)	256/2816 (0.091)	41/100	23/100	8/100
ProofNet	Ours	9136/9536 (0.958)	192/9536 (0.020)	127/186	52/186	45/186
ProofNet	Strong	8938/9280 (0.963)	128/9280 (0.014)	119/186	50/186	46/186
ProofNet	Sample	8164/8512 (0.959)	128/8512 (0.015)	106/186	46/186	41/186

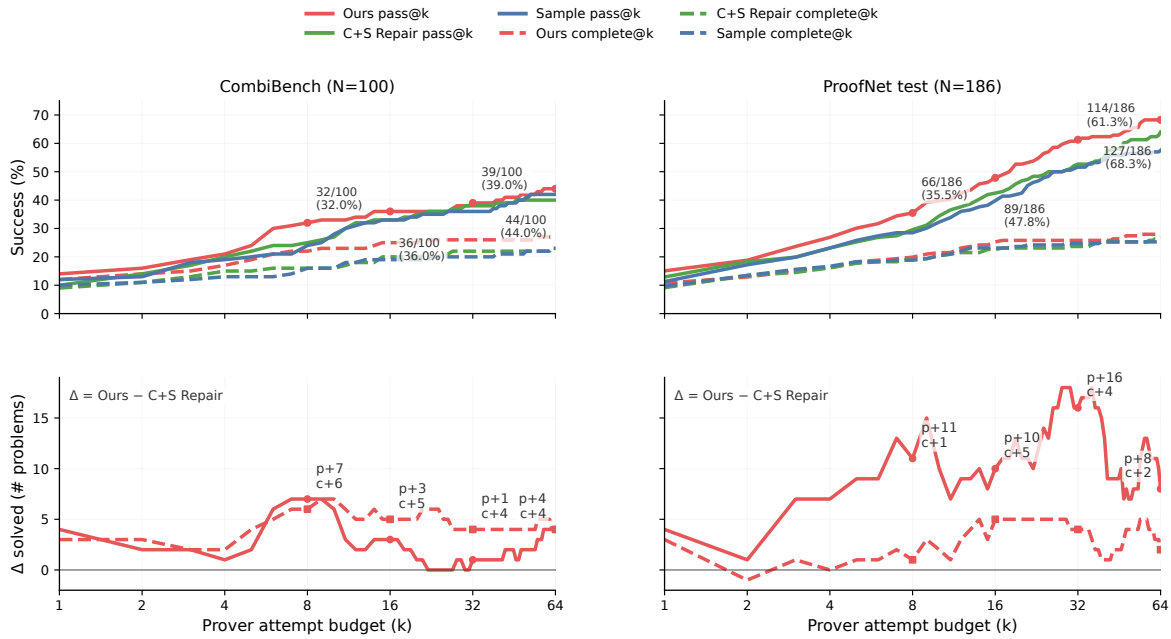

Figure 9: Proof utility as a function of prover attempts k on the judge-accepted repertoire produced within $T = 100$ calls. Solid: pass@k; dashed: complete@k; bottom row shows the gap relative to Kimina Compile+Semantic Repair.

Takeaway. On ProofNet, most prover prompts contain a `theorem/lemma` (around 0.96), so theorem-complete filtering is close to the standard metrics. On CombiBench, 9–13% of prompts are abbrev-only; theorem-complete@64 is therefore a stricter and more robust success signal.

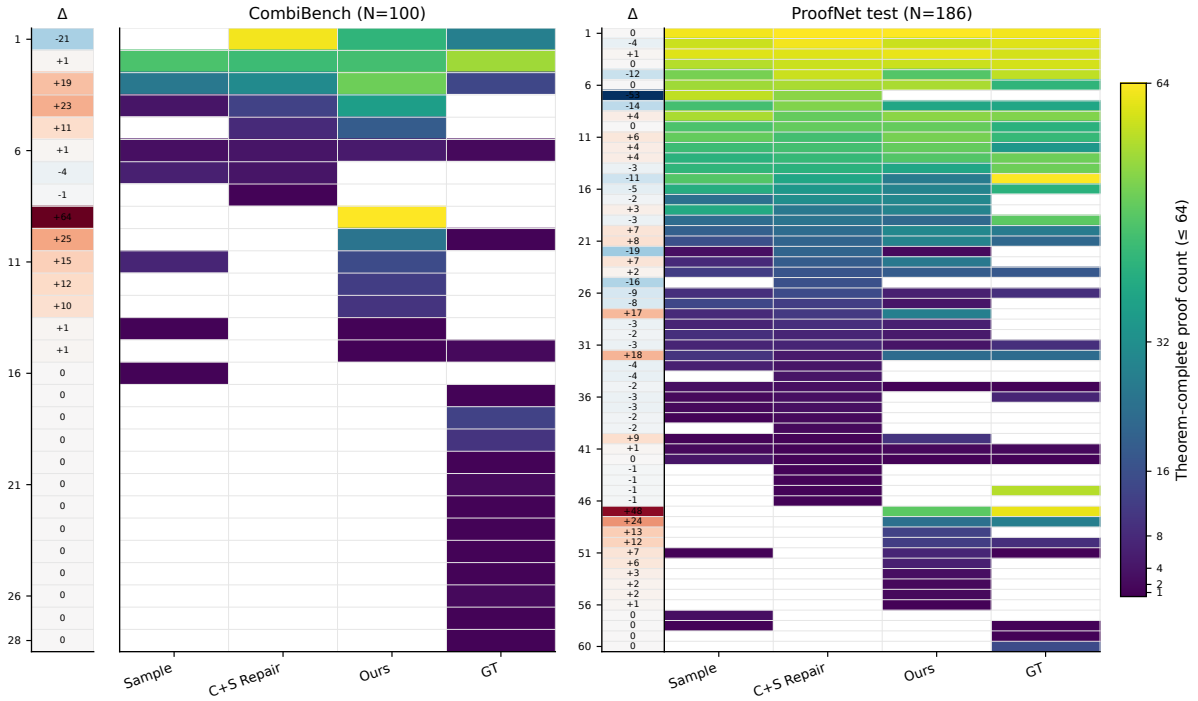

Figure 10: Filtered per-problem theorem-complete attempt counts under RR64 (cell = number of attempts, out of B=64, that yield a `theorem`/`lemma` without `sorry`). Rows show only problems where at least one method attains at least one theorem-complete attempt (so the number of displayed problems is smaller than N). Generator-based methods only attempt statements that pass the semantic judge (`semantic_ok`$=1$), whereas the Ground truth column runs the prover on the dataset-provided ground-truth formal statement (one per problem), bypassing the judge.

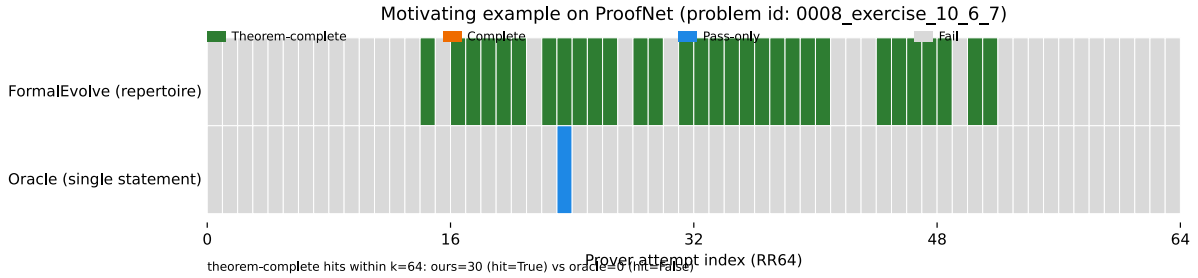

Figure 11: Data-driven motivating example under RR64 on ProofNet: even among judge-accepted candidates, prover outcomes can vary substantially under a fixed prover and attempt budget. The single-reference GT control uses one canonical statement (retried under RR64), while FormalEvolve provides a diverse judge-accepted repertoire, increasing the chance that at least one provable formulation is found within budget.

D Case Studies and EvolAST Ablation

We give representative statement/prover interactions and an EvolAST ablation to illustrate where archive search helps, where it is non-dominant, and how symbolic rewrites can change search trajectories under a fixed budget.

D.1 Case studies: two successes and one non-dominance example

Scope. We include two short success cases that illustrate prover-stage outcomes under the same RR64 budget (one where baselines fail to produce any Lean 4-accepted proof output, and one where a baseline reaches only pass-level artifacts), plus one counterexample that highlights non-dominance and judge/prover mismatch. To contextualize these examples, Table 9 quantifies prover-stage non-dominance frequency at theorem-complete@64 (FormalEvolve versus Compile+Semantic Repair) under the same RR64 protocol.

D.1.1 Theorem-complete proof: ours completes; baselines have no pass

Benchmark: `proofnet`.
Problem id: `0088_exercise_4_3_1`.
Informal statement:

> If R is a commutative ring and $a \in R$, let $L(a) = \{x \in R \mid xa = 0\}$. Prove that $L(a)$ is an ideal of R.

Outcome under the same fixed $B = 64$ prover budget. Our run attains a theorem-complete proof, while both sampling baselines fail to produce any Lean 4-accepted proof output within $B = 64$ attempts (no `pass`).

Takeaway. This instance illustrates a practical benefit of a repertoire: under a fixed prover attempt budget, the ability to try multiple judge-accepted statements can be the difference between reaching a complete proof and having no Lean-accepted proof output at all.

Notation. Lean 4 admits both Unicode and ASCII forms. For pdfLaTeX compatibility, we display prover transcripts in ASCII, where quantifiers/connectives may appear as `forall`/`->` and the constructor names `And`/`Or`, corresponding to $\forall$/ $\rightarrow$ and $\wedge$/$\vee$.

Abridged prover output excerpt (ours; theorem-complete).

```
import Mathlib

open Fintype Set Real Ideal Polynomial
open scoped BigOperators
noncomputable section

theorem my_theorem_123 {R : Type*} [CommRing R] (a : R) :
  forall x y : R, x * a = 0 -> y * a = 0 ->
    And ((x + y) * a = 0)
      (And ((-x) * a = 0)
        (forall r : R, And ((r * x) * a = 0) ((x * r) * a =
        0))) := by
intro x y hx hy
have h1 : (x + y) * a = 0 := by
  calc (x + y) * a = x * a + y * a := by ring
       _ = 0 := by simp [hx, hy]
have h2 : (-x) * a = 0 := by
  calc (-x) * a = - (x * a) := by ring
       _ = 0 := by simp [hx]
have h3 : forall r : R, And ((r * x) * a = 0) ((x * r) * a =
0) := by
  intro r
  have h4 : (r * x) * a = 0 := by calc (r * x) * a = r * (x *
  a) := by ring; _ = 0 := by simp [hx]
  have h5 : (x * r) * a = 0 := by calc (x * r) * a = (r * x) *
  a := by ring; _ = 0 := h4
  exact And.intro h4 h5
exact And.intro h1 (And.intro h2 h3)
```

```
Ours:    pass=True,  complete=True,  sorries=0
Strong:  pass=False
Sample:  pass=False
```

D.1.2 Complete proof: ours completes; baseline is pass-only

Benchmark: `proofnet`.
Problem id: `0000_exercise_2_3_2`.
Informal statement:

> Prove that the products ab and ba are conjugate elements in a group.

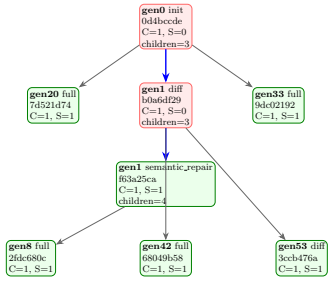

Figure 12: ProofNet `0000_exercise_2_3_2` statement evolution trace (Comp=compile-ok, Sem=semantic-ok). The repair step removes an extra conjunction and introduces the conjugation structure (the g^{-1} term).

Outcome under the same fixed $B = 64$ prover budget. Our run reaches a theorem-complete proof (`complete=True, sorries=0`), while Compile+Semantic Repair only reaches pass-level artifacts with remaining `sorry`.

Prover output excerpt (ours; theorem-complete).

```
import Mathlib

theorem my_theorem_xxx {G : Type*} [Group G] (a b : G) :
  exists g : G, a * b = g * (b * a) * g^(-1) := by
  refine Exists.intro a ?_
  simp [mul_assoc, mul_left_inv, mul_right_inv]
```

Statement evolution trace (mechanism note). This instance illustrates why we separate compilation from semantic checking: an early candidate may compile yet express a non-standard (and generally stronger) variant of the target claim, which can be judged inconsistent and/or harder for the prover to use under a fixed attempt budget. Here, the parent uses a conjunction of two equations, whereas the informal goal is a single conjugacy statement of the form $a*b = g*(b*a)*g^{-1}$. Bounded semantic repair removes the extra conjunct and rewrites the statement into this standard form while preserving compilation feasibility (Figure 12).

```
init (Comp=1, Sem=0): exists g, a*b = g*b*a /\ b*a = g*a*b
repair (Comp=1, Sem=1): exists g, a*b = g*(b*a)*g^(-1)
```

D.1.3 Non-dominance case: baseline theorem-complete; ours pass-only

Benchmark: `proofnet`.
Problem id: `0004_exercise_6_4_2`.
Informal statement:

> Prove that no group of order p q, where p and q are prime, is simple.

Outcomes under the same fixed $B = 64$ prover budget. In this instance, our best judge-accepted statement yields `pass=True` but not a complete proof, while both sampling baselines obtain a theorem-complete proof within the same prover attempt budget.

Takeaway. This example highlights non-dominance: even under the same prover budget, different judge-accepted statements can differ substantially in proof-search friendliness, so proof utility and semantic faithfulness need separate readouts.

Prover output excerpt (ours; pass-only). The excerpt below contains remaining `sorry`; the baseline theorem-complete outputs serve as the comparison point through the reported outcome flags. Ours (pass-only):

```
import Mathlib

theorem my_theorem_12345 {G : Type*} [Group G] [Fintype G] (p
q : Nat)
  (hp : p.Prime) (hq : q.Prime) (hcard : Fintype.card G = p *
  q) :
    Not (IsSimpleGroup G) := by
  have h_main : Not (IsSimpleGroup G) := by sorry
  sorry
```

```
Ours:    pass=True,  complete=False, sorries=1
Strong:  pass=True,  complete=True,  sorries=0
Sample:  pass=True,  complete=True,  sorries=0
```

D.2 Early proof successes under fixed prover budgets

Table 16 lists representative instances where FormalEvolve reaches a successful proof substantially earlier than Compile+Semantic Repair under the same prover attempt budget, or succeeds when the baseline fails within $B = 64$ attempts. We report the first prover attempt index k (under the round-robin schedule described in Appendix A.1) where the prover returns (i) any proof script accepted by Lean 4 (pass) or (ii) a complete proof without `sorry` (complete).

Table 16: Representative early proof successes under a fixed per-problem prover attempt budget $B = 64$, computed on the judge-accepted repertoire produced within $T = 100$ generator calls. We report the first prover attempt index k where the prover succeeds; "> 64" indicates no success within the attempt budget.

Benchmark	Criterion	Problem (id + short description)	Ours first k	C+S Repair first k
Both succeed, but FormalEvolve is much earlier				
ProofNet	pass	`0113_exercise_18_8a` – order topology: $\{x \mid f(x) \leq g(x)\}$ is closed	1	58
ProofNet	complete	`0113_exercise_18_8a` – order topology: $\{x \mid f(x) \leq g(x)\}$ is closed	12	58
ProofNet	pass	`0174_exercise_5_3` – $f(x) = x + \varepsilon g(x)$ is injective for small ε	2	64
ProofNet	complete	`0174_exercise_5_3` – $f(x) = x + \varepsilon g(x)$ is injective for small ε	13	64
CombiBench	pass	`0047_brualdi_ch13_10` – tournament radius-2 vertex exists	9	27
CombiBench	complete	`0047_brualdi_ch13_10` – tournament radius-2 vertex exists	21	27
CombiBench	pass	`0057_imosl_2015_c6` – infinitely many non-clean integers	1	5
CombiBench	complete	`0057_imosl_2015_c6` – infinitely many non-clean integers	3	15
FormalEvolve succeeds, baseline fails within $B = 64$				
ProofNet	complete	`0000_exercise_2_3_2` – ab and ba are conjugate in a group	2	> 64
ProofNet	complete	`0046_exercise_3_4_5b` – solvable quotients are solvable	1	> 64
CombiBench	complete	`0007_hackmath_8` – ferry crossing: women in first group	1	> 64
CombiBench	complete	`0063_usamo_2000_p4` – 3 colored squares make a right triangle	1	> 64

Table 17: ProofNet instances where the main run and the no-EvolAST ablation disagree on semantic hit@100 under fixed-budget accounting. We report the first call index where a semantic success appears (when it does), computed from the budget accounting logs.

Problem id	Short description	Main (repair + EvolAST)	No EvolAST (repair only)
`0066_exercise_8_3_6b`	$\mathbb{Z}[i]/(q)$ is a field with q^2 elements (for $q \equiv 3 \bmod 4$)	no hit	hit at call 42
`0070_exercise_9_4_9`	$x^2 - \sqrt{2}$ irreducible over $\mathbb{Z}[\sqrt{2}]$	no hit	hit at call 65
`0142_exercise_4_15a`	Uniform continuity iff having a modulus of continuity	no hit	hit at call 49
`0151_exercise_1_8`	No ordered-field structure on $\mathbb{C}$	no hit	hit at call 39
`0185_exercise_5_1`	Blaschke condition: $\sum_n (1 - \lvert z_n \rvert) < \infty$ for bounded holomorphic f	no hit	hit at call 66
`0123_exercise_25_9`	Identity component is a normal subgroup	hit at call 87	no hit

D.3 Ablation note: when EvolAST changes outcomes on ProofNet

On ProofNet, enabling EvolAST slightly decreases semantic hit@100 under fixed-budget accounting (158/186 vs 162/186). The gap is small (4 problems), but it illustrates an important point for budgeted search: a diversity operator can change the search trajectory and archive composition, and may not be universally beneficial across datasets.

Where the difference comes from. Under fixed-budget accounting, there are 5 problems that achieve at least one judge-accepted statement under the no-EvolAST configuration but not under the main configuration, and 1 problem where the reverse holds. Because the generator and patching process are stochastic, these disagreements reflect diverging search trajectories under nearly identical protocols; attributing causality to any single operator requires careful per-instance inspection. Empirically, these disagreements are consistent with a mixture of direct trigger effects (duplicates/compile-fail fallbacks), indirect trajectory effects, and dataset-dependent operator behavior.

Representative example (problem `0142_exercise_4_15a`). In this instance, the main configuration produced many compilable candidates but no semantic success, whereas the no-EvolAST configuration eventually reaches a judge-accepted statement after semantic repair. One characteristic we observe in the main run is that EvolAST fallback triggers repeatedly on duplicates and compile failures, producing compilable but judge-rejected variants; in this problem, many candidates are explicitly tagged as EvolAST fallbacks, and all have $\mathsf{Comp}(c) = 1$ but $\mathsf{Sem}(c) = 0$. This can bias the archive toward structurally perturbed but semantically misaligned candidates under a fixed budget, which in turn alters parent selection and patch contexts.

Concretely, the semantic judge flags a common failure mode here: *domain mismatch* (e.g., proving uniform continuity on $\mathbb{R}$ with the intended $[a, b]$ restriction missing). The best candidate found by the main run (compilable but judge-rejected) has the unconstrained target:

$$(\exists \mu, \cdots \wedge \forall s, t \in \mathbb{R},\ |f(s) - f(t)| \leq \mu(|s - t|)) \leftrightarrow \texttt{UniformContinuous}\ f$$

whereas the successful no-EvolAST semantic-repair output explicitly restricts the bound to $s, t \in [a, b]$ and targets uniform continuity on the interval:

$$(\exists \mu, \cdots \wedge \forall s, t,\ s \in \texttt{Icc}\ a\ b \to t \in \texttt{Icc}\ a\ b \to |f(s) - f(t)| \leq \mu(|s - t|)) \leftrightarrow \texttt{UniformContinuousOn}\ f\ (\texttt{Icc}\ a\ b)$$

This illustrates a fundamental limitation of a conservative type-rewrite operator: our EvolAST rule

set is limited to the existing binder and goal AST, so new problem-specific binders (e.g., explicit a, b interval parameters) and new domain-restriction hypotheses require LLM repair. Therefore, when the current parent trajectory has not yet reached a semantically correct "skeleton", applying EvolAST tends to preserve the same semantic mistake, producing many feasible variants that remain judge-rejected. This effect is amplified by our implementation choice that EvolAST candidates terminate the fallback path to keep accounting bounded, leaving later semantic correction to subsequent LLM proposals.

Takeaway. Online diversity operators can have dataset-dependent effects on semantic hit rates under a fixed budget. In practice, this motivates tunable EvolAST rule sets, tunable trigger conditions, and transparent ablation reporting.

EvolAST rewrite surface. In our experiments, EvolAST is used in its rule-based mode: it applies a bounded sequence of conservative, semantics-intended AST rewrites to the theorem type (binder types and goal type), keeping the proof body unchanged. The rewrite surface is intentionally controlled and includes conservative hypothesis reordering; commutativity/associativity/distributivity for conjunction/disjunction (Lean: $\wedge/\vee$); symmetry swaps (e.g., $a = b$ to $b = a$, $P \leftrightarrow Q$ to $Q \leftrightarrow P$); and dual relations (e.g., $a < b$ to $b > a$, $a \leq b$ to $b \geq a$).

E Algorithm, Budget Accounting, and Configuration

We give the detailed per-problem search procedure, generator-call accounting rules, semantic-judge overhead, and implementation configuration used in the fixed-budget experiments.

E.1 Detailed Pseudocode

Accounting conventions. Under fixed-budget accounting, we debit only generator-side LLM calls (proposals and bounded repairs). EvolAST is a symbolic fallback triggered on (i) exact duplicates after canonicalization and (ii) compilation failures; it does not consume generator-call budget and is evaluated as the representative candidate of the triggering call. Each compilation-repair or semantic-repair attempt is itself a debited generator call whose representative candidate is the repaired output; semantic repair preserves the original compilable candidate and appends a repaired variant, and both are evaluated. All metrics normalize by the fixed budget T per problem; any early termination or failed call is treated as a consumed call with $\mathsf{Comp}(c) = 0$ and $\mathsf{Sem}(c) = 0$.

Parent selection. Parent selection uses the usage-penalized robust weighting described in Section 3. For completeness, each archived candidate c_i has gate score $s_i = \mathsf{Comp}(c_i)(1+\mathsf{Sem}(c_i))$ and parent-usage count n_i. Within the selected island, we set $z_i = (s_i - m)/d$, where $m = \mathrm{median}_j(s_j)$ and $d = \max(\mathrm{MAD}(\{s_j\}_j), 10^{-6})$. The quality factor is $Q_i = \sigma(\lambda z_i)$, the reuse factor is $R_i = [1 + (1+\beta)n_i]^{-1}$, and the next parent is sampled with probability $Q_i R_i / \sum_j Q_j R_j$. Here $\mathrm{MAD}(A) = \mathrm{median}_{a \in A}|a - \mathrm{median}(A)|$, 10^{-6} is a numerical stabilizer, and R_i discourages repeatedly selecting the same parent.

Algorithm E.1 FormalEvolve (per problem; detailed)

Input: informal statement x, generator-call budget T, seed model M_{seed}, patch model M_{patch}, predicates $\mathsf{Comp}, \mathsf{Sem}$, islands K
Output: feasible archive $\cup_I \mathcal{A}_I$ partitioned into islands
$t \leftarrow 0$; $\mathcal{A}_{1:K} \leftarrow (\emptyset, \ldots, \emptyset)$
$(\mathcal{A}_{1:K}, t) \leftarrow$ INITSEEDS$(x, M_{\text{seed}}, T, K; \mathsf{Comp}, \mathsf{Sem})$ (debits budget; compiles/judges seeds; inserts feasible seeds into islands)
while $t < T$ and $\bigcup_I \mathcal{A}_I \neq \emptyset$ **do**
  $I \leftarrow$ SAMPLEISLAND$(\{I : \mathcal{A}_I \neq \emptyset\})$ (uniform over nonempty islands)
  $p \leftarrow$ SAMPLEPARENT$(\mathcal{A}_I)$; $n_p \leftarrow n_p + 1$
  $(\mathcal{I}_{\text{arch}}, \mathcal{I}_{\text{top}}) \leftarrow$ SAMPLECONTEXT$(\mathcal{A}_I, p)$
  $c \leftarrow$ PROPOSE$(p, \mathcal{I}_{\text{arch}}, \mathcal{I}_{\text{top}}, x; M_{\text{patch}})$
  $t \leftarrow t + 1$
  $c \leftarrow$ ENFORCE(c, p) (preamble/proof protocol)
  if ISEXACTDUPLICATE$(c, \mathcal{A}_I)$ **then**
    $c_{\text{rep}} \leftarrow$ EVOLAST(p) (generator-call-free representative)
    PROCESSCANDIDATE$(c_{\text{rep}}, x, p, \mathcal{A}_I, t, \text{false})$ (no semantic repair)
  else if COMPILES(c) **then**
    PROCESSCANDIDATE$(c, x, p, \mathcal{A}_I, t, \text{true})$
  else
    $c_{\text{rep}} \leftarrow$ EVOLAST(p) (compile-fail representative; no call)
    PROCESSCANDIDATE$(c_{\text{rep}}, x, p, \mathcal{A}_I, t, \text{false})$
    if $t < T$ **then**
      $c_{\text{fix}} \leftarrow$ CREPAIR$(c, x; M_{\text{patch}})$; $t \leftarrow t + 1$
      $c_{\text{fix}} \leftarrow$ ENFORCE(c_{fix}, p)
      PROCESSCANDIDATE$(c_{\text{fix}}, x, p, \mathcal{A}_I, t, \text{true})$
    end if
  end if
  MAYBEMIGRATE$(\mathcal{A}_1, \ldots, \mathcal{A}_K)$ (periodic island migration; see Appendix E.3)
end while

Table 18: Strict generator-call budget audit (call currency). Budget is fixed to $N \times T$ with $T = 100$ per problem; Gen/CRep/SRep partition the same budget.

Dataset	Method	Budget	Gen	CRep	SRep
CombiBench	sample	10000	10000	0	0
CombiBench	strong_compile	10000	7031	2969	0
CombiBench	strong_semantic	10000	4257	1991	3752
CombiBench	hybrid	10000	4081	1855	4064
CombiBench	ours	10000	4895	3042	2063
CombiBench	ours_no_evolast	10000	5139	3428	1433
CombiBench	ours_no_repair	10000	10000	0	0
ProofNet	sample	18600	18600	0	0
ProofNet	strong_compile	18600	13881	4719	0
ProofNet	strong_semantic	18600	10732	3987	3881
ProofNet	hybrid	18600	10887	3700	4013
ProofNet	ours	18600	11872	4903	1825
ProofNet	ours_no_evolast	18600	11984	4887	1729
ProofNet	ours_no_repair	18600	18600	0	0

Algorithm E.2 PROCESSCANDIDATE (compile gate, judge, and bounded semantic repair)

Input: candidate c, informal statement x, parent p, current island archive $\mathcal{A}_I$, call counter t, allow semantic repair flag $r \in \{\text{true}, \text{false}\}$
Effect: if c is feasible, evaluates $\mathsf{Sem}(c)$ and inserts into $\mathcal{A}_I$; if $\mathsf{Sem}(c) = 0$ and $r = \text{true}$, spends one additional generator call on bounded semantic repair and evaluates the repaired candidate

```
if COMPILES(c) then
  m ← EVALUATE(c, x)                    (semantic judge)
  A_I ← INSERT(A_I, (c, m))             (Comp(c) = 1 only)
  if r ∧ Sem(c) = 0 ∧ t < T then
    c' ← SREPAIR(c, x; M_patch); t ← t + 1
    c' ← ENFORCE(c', p)
    if COMPILES(c') then
      m' ← EVALUATE(c', x)
      A_I ← INSERT(A_I, (c', m'))
    end if
  end if
end if
```

E.2 Budget Audit and Evaluation Overhead

Generator-side. We audit generator-call usage under the budget currency ($T = 100$ calls per problem). We decompose the fixed budget $N \times T$ into generation calls (Gen), compilation-repair calls (CRep), and semantic-repair calls (SRep). Table 8 provides a complementary per-problem distributional view (median/IQR) of CRep/SRep under the same accounting.

Evaluator-side (semantic judge). We report evaluator overhead in terms of semantic LLM-as-judge calls (CriticLean-Qwen3-14B). In this pipeline, semantic judging is invoked only after successful compilation, and EvolAST outputs are judged under the same gate; EvolAST-Judge is the subset of judge calls attributed to EvolAST-labeled candidates (generator-call-free on the generator side).

Table 19: Semantic judge overhead (CriticLean-Qwen3-14B). JudgeCalls counts how many compilable candidates reach semantic checking; EvolAST-Judge is the EvolAST-attributed subset and is included in JudgeCalls.

Dataset	Method	Problems	JudgeCalls	EvolAST-Judge
CombiBench	sample	100	6953	0
CombiBench	strong_compile	100	5513	0
CombiBench	strong_semantic	100	6109	0
CombiBench	ours	100	4970	863
CombiBench	ours_no_evolast	100	3800	0
CombiBench	ours_no_repair	100	9025	3797
ProofNet	sample	186	13062	0
ProofNet	strong_compile	186	11624	0
ProofNet	strong_semantic	186	11904	0
ProofNet	ours	186	10536	331
ProofNet	ours_no_evolast	186	10103	0
ProofNet	ours_no_repair	186	13465	1008

Table 20: Key hyperparameters used in FormalEvolve.

Component	Setting
Models	Seed: Kimina-Autoformalizer-7B; patch/repair: Qwen3-30B-A3B.
Compilation	Lean 4.15.0 + Mathlib via Kimina Lean Server (Dos Santos et al., 2025).
Semantic judge	CriticLean-Qwen3-14B (LLM-based judge; proof ignored).
Islands	2
Island separation	enabled; parent selection and inspirations restricted to the sampled island
Archive invariant	compilation-feasible (compile_ok$=1$)
Archive size	global capacity 40 (compilation-feasible)
Migration policy	interval 10 gens, rate 0.1; moves feasible gen>0 candidates; elitism (top-1 protected)
Parent selection	weighted ($\lambda = 10$) with usage penalty $\beta = 0.05$
Operator mix (default)	`full`/`diff`/`cross` with probs 0.5/0.3/0.2
Patch attempts	max patch attempts 1 (per step)
Inspiration pool	per step: 4 archive inspirations + 2 top-k inspirations (when available)
Cross prompt inspirations	samples $k = 1$ inspirations uniformly from the pool
Repair budget	max attempts 2, temperature 0.7 (applies to compile + semantic repair)
EvolAST fallback	enabled as a fallback for (i) exact-duplicate edits, and (ii) compilation failures (evaluated as a generator-call-free fallback alongside bounded compile repair)

E.3 Configuration and hyperparameters

Table 20 summarizes the key hyperparameters used in the main experiments.

Lean 4 toolchain and library. All candidates are compiled under Lean 4.15.0 with Mathlib (`import Mathlib`). We use Mathlib4 `v4.15.0` (commit `9837ca9`) as bundled by our Kimina Lean Server installation. Compilation is served by Kimina Lean Server (Dos Santos et al., 2025), which provides a pinned Lean 4/Mathlib environment suitable for large-scale batch checking.

Implementation notes: archive and island mechanics.

- **Global feasible archive.** We maintain a global compilation-feasible archive (capacity 40); selection is applied only within this feasible set.
- **Island-local behavior.** Parent selection and cross-patch inspirations are restricted to the sampled island (via `island_idx`); children inherit the parent island by default.
- **Migration.** Every 10 generations, we migrate a small fraction of feasible candidates between islands (rate 0.1), while protecting the top-1 elite in each island.

F Prompting and Semantic-Judge Protocol

We report the prompt templates used for initial statement generation, evolution operators, bounded repair, semantic judging, and downstream proof completion. We show the full prompt skeletons because prompt details affect both reproducibility and the reliability of LLM-as-judge semantic labels.

Notation. Placeholders include {informal} (natural-language statement), {code_content} (current Lean file), {performance_metrics} (compile/semantic metrics), {text_feedback_section} (optional judge feedback), {original_code} (candidate file for repair), {compile_error_type}/{compile_error_msg} (compiler feedback), {critic_feedback} (semantic-judge feedback), {inspiration_code_1} (cross inspiration), and {lean_statement} (theorem statement for judging). For pdfLaTeX compatibility, we

Table 21: Prompt components used in our pipeline.

Component	Model	Purpose / output constraints
Initial generation	M_{seed}	Produce one Lean 4 file with a single theorem statement (`:= by sorry`).
Operators (full/diff/cross)	M_{patch}	Edit-conditioned rewrite of a parent; cross additionally conditions on one inspiration from the archive.
Compile repair (bounded)	M_{patch}	Minimal patch conditioned on compiler feedback; debited within the same call budget.
Semantic repair (bounded)	M_{patch}	Revision conditioned on the informal statement and judge feedback; debited within the budget.
Semantic judge	`CriticLean-Qwen3-14B`	Binary verdict on statement faithfulness (proof ignored), returned as JSON (`Correct/Incorrect` plus a short rationale).
Prover	`Goedel-Prover-V2-32B`	Complete a Lean file by replacing the placeholder proof under a fixed prover prompt.

render common Lean unicode symbols in ASCII in verbatim displays.

F.1 Statement generation (initial)

Prompt template.

```
You are an expert in Lean 4 theorem proving and Mathlib.

Given a problem statement in natural language, write a
COMPLETE Lean 4 file (imports + theorem)
that formalizes the mathematical content.

MANDATORY OUTPUT REQUIREMENTS:
- Output EXACTLY ONE Lean 4 code block: ```lean ... ```.
- The code MUST start with the following two lines (in this
order):
  import Mathlib
  import Aesop
- You MAY add additional `import ...` / `open ...` /
`set_option ...` lines after that if needed (unless a
mode-specific prefix locks the header, e.g., diff mode).
- Do NOT include any comments.
- Include EXACTLY ONE `theorem` declaration.
- The theorem MUST end with `:= by sorry` (do not provide a
proof).
- Use Lean 4 v4.15-compatible syntax and Mathlib definitions.

Natural language statement:
{informal}

Return ONLY the Lean 4 code block.
```

F.2 Evolution operators (full / diff / cross)

Shared prompt skeleton.

```
You are an expert in Lean 4 theorem proving and the Mathlib
library.
Your task is to improve a Lean 4 formalization for a given
natural language mathematical claim.

Hard requirements (must satisfy all):
- Output a COMPLETE Lean 4 file (imports + exactly one
theorem) in a single ```lean``` code block.
- The code MUST start with these two lines (in this order):
  import Mathlib
  import Aesop
- You MAY add additional `import ...` / `open ...` /
`set_option ...` lines after that if needed.
- Do NOT include any comments.
- Include EXACTLY ONE `theorem` declaration.
- The theorem MUST end with `:= by sorry` (do not provide a
proof).

Quality goals:
- compile_ok: The file compiles without errors.
- semantic_ok: The theorem statement matches the informal
mathematical meaning.

Natural language statement:
{informal}

Current Lean 4 program:
```lean
{code_content}
```

Evaluation results:
{performance_metrics}{text_feedback_section}

Return ONLY the Lean 4 code block.
```

Full-mode addenda (sampled variants; one per call).

```
(i) Default rewrite: revise types/hypotheses/structure as
needed.
(ii) Different interpretation: try a different reasonable
formalization of the same claim.
(iii) Type optimization: improve binder structure and
type-class constraints.
(iv) Mathlib alignment: prefer standard Mathlib names and
definitions.
(v) Simplification: if possible, state a simpler claim
without changing meaning.
```

Diff mode prepends a short local-edit instruction (still returning a full file), while cross mode additionally provides one sampled inspiration candidate from the archive/top-k pool as extra context.

Diff-mode prefix (local edit).

```
You are doing a LOCAL EDIT of a Lean 4 formalization file.
Header lock (diff mode only):
- Do NOT add/remove/reorder/modify any preamble lines
(imports/opens/options).
- Keep everything before the `theorem` declaration EXACTLY
unchanged.
Goal:
- Make the smallest possible change to improve compilation
and formalization quality.
- Keep changes minimal; do NOT change the intended
mathematical meaning.
```

Cross-mode prefix (inspiration-guided).

```
You are doing CROSS patching.
You are given one inspiration candidate from the
archive/top-k pool as extra context.
Your task is to improve the current formalization by
optionally borrowing useful structure from the inspiration.
```

Cross-mode extra context (one per call).

```
[Inspiration candidate]
```lean
{inspiration_code_1}
```
```

F.3 Bounded repair prompts

Repairs count toward the generator-call budget. Compilation repair triggers only when `compile_ok=0`; semantic repair triggers only when `compile_ok=1` and `semantic_ok=0`.

Compilation repair prompt.

```
You are an expert in Lean 4 theorem proving and Mathlib.
You are doing SYNTAX / COMPILATION REPAIR.

You will receive:
- A natural language statement (the semantic target)
- A Lean 4 file that fails to compile
- Compiler error feedback

Your job:
- Produce a corrected Lean 4 file that compiles.

IMPORTANT:
- Fix compilation only; do NOT change the intended
mathematical meaning.
- Keep changes minimal (types, identifiers, imports, binder
annotations, etc.).

Natural language statement:
{informal}

Current Lean 4 code (does NOT compile):
<CURRENT_CODE>
{original_code}
</CURRENT_CODE>

Compiler feedback:
- Error type: {compile_error_type}
- Error message:
```
{compile_error_msg}
```

Return exactly one Lean 4 code block, starting with `import
Mathlib` and `import Aesop`,
with exactly one theorem ending in `:= by sorry`.
```

Semantic repair prompt.

```
You are an expert in mathematics and Lean 4 (Mathlib).
You are doing SEMANTIC REPAIR.

You will receive:
- A natural language statement (the semantic target)
- A Lean 4 file that is intended to formalize it
- Critic feedback (Accuracy Confirmation) describing
mismatches

Your job:
- Modify the Lean 4 code so that the theorem statement
matches the natural language statement.

Rules:
- You MAY change hypotheses and the conclusion if needed to
match the semantics.
- You MAY adjust/add imports/opens/options as needed to keep
the file compiling.
- Do NOT "solve" the task by weakening it to `True` or a
tautology.
- Do NOT include any comments.
- Output a complete Lean 4 file starting with:
  import Mathlib
  import Aesop
- Include exactly one theorem, ending with `:= by sorry`.

Natural language statement:
{informal}

Current Lean 4 code:
<CURRENT_CODE>
{original_code}
</CURRENT_CODE>

Critic feedback (Accuracy Confirmation):
{critic_feedback}

Goal:
- Modify the Lean 4 theorem so that the semantic judge would
accept it as an exact formalization of the natural-language
statement.
- Address every mismatch mentioned in the Critic feedback.

Return exactly one Lean 4 code block, starting with `import
Mathlib` and `import Aesop`,
with exactly one theorem ending in `:= by sorry`.
```

F.4 Semantic judge prompt (CriticLean-Qwen3-14B)

We use CriticLean-Qwen3-14B as the LLM-as-judge implementation of Sem, judging semantic consistency between the natural-language statement and the Lean 4 theorem statement (proof ignored). The prompt template below is used for each semantic check.

```
Role: Lean 4 & Formal Verification Expert
Input:
- Mathematical_Text: A math problem and its answer (no
proof).
- Lean4Code: A Lean 4 theorem statement formalizing the
problem (proof intentionally omitted).
Goal:
Determine if the Lean 4 theorem statement is an exact and
faithful formalization of the mathematical problem.
Do not evaluate or consider the answer or the proof. Your
sole task is to verify the correctness of the formalization.

Evaluation stages:
1. Identify the mathematical claim: variables, types,
quantifiers, assumptions, constraints, logic structure, and
conclusion.
2. Extract the corresponding components from the Lean 4
statement, ignoring the proof.
3. Compare the two statements for semantic alignment,
quantifier correctness, preserved constraints/boundaries,
correct typing, syntactic validity, and absence of semantic
drift or unjustified additions.
4. Give the final judgment using only what is explicitly
expressed in the Lean statement, not the proof.
5. If correct, explain why the elements match; if incorrect,
list mismatches and their effect on correctness.

Output Format:
Return exactly one JSON object:
{
  "reasons": "<your detailed analysis as a single string>",
  "is_assistant_correct": "Correct or Incorrect"
}

Input Data:
-- Start of Mathematical_Text --
{informal}
-- End of Mathematical_Text --
-- Start of Lean4Code --
{lean_statement}
-- End of Lean4Code --
```

F.5 Prover prompt (Goedel-Prover-V2-32B)

We evaluate downstream proof utility using a fixed prover configuration (Goedel-Prover-V2-32B) and a fixed prompt template. The prompt asks the prover to complete a Lean file by replacing the placeholder proof.

```
Complete the following Lean 4 code.
Return ONLY a complete Lean 4 file inside a ```lean4``` code
fence (no explanations): {lean_file}
```