M2F: Automated Formalization of Mathematical Literature at Scale

Zichen Wang [1] **Wanli Ma** [2] **Zhenyu Ming** [3] **Gong Zhang** [3] **Kun Yuan** [4] **Zaiwen Wen** [2]

Abstract

Autoformalization is often evaluated one theorem at a time. Frontier mathematics exposes a different bottleneck. Before a hard theorem can be proved, a project needs well-modeled definitions, reusable APIs, canonical lemma statements, and stable dependencies. We present M2F (Math-to-Formal), a framework for compiling long mathematical sources into project-scale Lean infrastructure. M2F first turns source-aligned statement skeletons into a Lean project with proof placeholders. It then freezes statement signatures and repairs the remaining holes. Both stages use a verifier-gated accept/revert contract: an edit is committed only when Lean reports improvement under the pinned environment. We evaluate M2F as an artifact-construction system, not as a theorem-snippet prover. The released workspace spans seven source roots across six domains, with 591 generated Lean files and 10,637 declarations. It builds end to end, and a strengthened static scan finds no explicit bypass, unsafe, or search-disabling patterns. A two-expert adjudicated audit labels 2,023/2,089 matched statements as acceptable, and M2F closes all 875 proof obligations in the retained audited proof subset. Reuse, source-order, overlap, and `mathlib`-compatibility analyses indicate that the generated projects behave as local libraries rather than direct retrieval of existing facts. As a separate fixed-statement Stage 2 probe, M2F solves 96/100 FATE-H instances automatically and two more with human-written natural-language infrastructure hints that expose reusable declarations, not proofs. Together, these results support a statement-first view of large-scale autoformalization.

[1]School of Mathematical Sciences, Peking University [2]Beijing International Center for Mathematical Research, Peking University [3]Huawei Technologies [4]Center for Machine Learning Research, Peking University. Correspondence to: Zaiwen Wen <wenzw@pku.edu.cn>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

1. Introduction

Most mathematical knowledge lives in textbooks and research papers. The long-term goal of autoformalization is therefore not only to solve isolated theorems, but to grow reusable formal infrastructure: Lean-checked definitions, stable APIs, lemma libraries, and abstractions compatible with `mathlib` (The mathlib Community, 2020). Strong provers help only after this layer exists. Major theorems become tractable when the right definitions, intermediate lemmas, naming conventions, and domain abstractions are already available. Formalizing even one section in LEAN (de Moura et al., 2015; de Moura & Ullrich, 2021) remains labor-intensive because imports, definitions, dependencies, and statements must elaborate together under a pinned snapshot. This is a project-scale compilation problem, not only a theorem-proving problem.

Existing work provides important ingredients, often at a different layer. Neural theorem provers close goals *given* well-formed statements inside a buildable project. Autoformalization systems translate text into formal snippets. Long documents introduce failures that are global rather than local: unresolved names, incompatible definitions, import cycles, type-mismatch cascades, and statements that are locally provable but poor as reusable interfaces. We target the upstream task: compiling textbook- and paper-scale sources into buildable, provenance-linked Lean projects.

This library view changes the evaluation target. At theorem scale, a proof is the final product. At library scale, the statement is the durable interface, and the proof is a local consistency signal for that interface. A provable statement can still be a poor library artifact if it weakens the source meaning, uses the wrong definition or typeclass, sits at the wrong abstraction level, or creates downstream friction through coercions and glue lemmas. We do not claim to solve the expert problem of finding globally optimal `mathlib` abstractions. Instead, we assemble project-level evidence: generated statements elaborate together, support later results, follow source order, and connect back to canonical library concepts when such concepts already exist.

We address this setting with **M2F** (Math-to-Formal), a framework for automated formalization of mathematical literature at project scale. We formulate the task as *knowledge compilation* under a fixed Lean environment $\mathcal{E}$. Given

a textbook or paper, the goal is to produce a Lean project that elaborates end-to-end and records provenance from declarations back to project-local source items. The execution mechanism is *verifier-gated refinement* (VeriRefine): M2F proposes localized edits and commits them only when Lean reports improvement on compilation or, after compilation holds, on closing remaining proof holes. VeriRefine is a stability contract for long-horizon project construction, not a claim of globally optimal search.

M2F has two stages. Statement compilation builds projects with proof placeholders; proof repair freezes signatures and closes holes. The released Lean workspace contains *591 files*, *10,637 Lean declarations*, and *379,366 lines of Lean* across seven buildable source roots (Table 1). A strengthened comment-stripped scan finds no uncommented `sorry`/`admit`, source-level axiom-like declarations, unsafe or partial declarations, heartbeat-disabling options, meta/macro command surface, or imports of modules named `Unsafe`. Only about *22.6%* of declarations are directly source-matched; the remaining *77.4%* are generated definitions, helper lemmas, compatibility facts, wrappers, and proof-route infrastructure. On the provenance-audited subset with complete logs, Stage 2 closes all *875/875* retained audited proof obligations. FATE-H is used only as a fixed-statement Stage 2 probe. Our contributions are:

- **Project-scale formalization as knowledge compilation.** We formulate textbook-/paper-scale formalization as compiling a buildable Lean project under a pinned environment, with provenance and evaluation beyond per-snippet elaboration.
- **Buildable, auditable, cross-domain artifacts.** We produce long-form Lean projects spanning analysis, topology, category theory, algebra, optimization, and convex analysis, with no explicit bypasses under a strengthened static scan. Provenance-linked expert audit gates the retained matched-statement subset.
- **VeriRefine as a stability contract.** We instantiate feedback-driven refinement as an accept/revert protocol for long-horizon Lean project construction that preserves the best-known project state.
- **Proof repair as Stage 2 validation.** Under matched statements, M2F closes all audited long-form proof obligations and most fixed-statement FATE-H instances; natural-language infrastructure hints show that some remaining failures are missing declaration navigation rather than missing proof scripts.

Figure 1 shows the pipeline. An artifact repository with buildable generated Lean projects is available at `https://github.com/imathwy/M2F_repo`. The supplementary archive includes the generated Lean workspace, the document-to-item parser, and orchestration code with prompt templates and agent configurations.

2. Related Work

Neural theorem proving in Lean. Automated Lean provers typically combine retrieval, search, and verifier feedback for goals that already elaborate. Tool interfaces and benchmarks include MiniF2F and LeanDojo (Zheng et al., 2022; Yang et al., 2023), and recent systems improve performance with stronger models and agentic search (Xin et al., 2024; Ren et al., 2025; Chen et al., 2025; Jiang et al., 2025; Ospanov & Yousefzadeh, 2025). M2F targets the upstream regime where imports, name resolution, dependency ordering, definitions, and type-mismatch cascades must be repaired before proof search is meaningful.

Autoformalization with tool feedback and structural guidance. LLM-based autoformalization translates informal sources into formal statements and proofs (Wu et al., 2022); Lean-centric pipelines increasingly incorporate compiler/tool feedback (Lu et al., 2024b). Recent systems use structural cues and dependency information, including ATF (Guo et al., 2025), ProofFlow (Cabral et al., 2025), Aria (Wang et al., 2025), and SITA (Li et al., 2026). Related Isabelle/HOL work studies library-oriented consistency and real-world definition formalization (Zhang et al., 2024; 2025). M2F is complementary. We do not claim feedback-driven refinement is new in isolation; the change is the artifact unit and success criterion: a source-local Lean project that builds end-to-end and exposes declarations that can be audited and reused locally.

Statement quality, evaluation, and repair. Autoformalization evaluation includes aligned corpora and alignment metrics (Gao et al., 2025; Ying et al., 2024; Azerbayev et al., 2023; Gulati et al., 2024; Lu et al., 2024a; Poiroux et al., 2025). We add project-level evidence: buildability, provenance-supported audit beyond typechecking, matched-statement proof repair, and verifier-normalized compute. Methodologically, our accept/revert loop is related to diagnostic-conditioned program repair (Monperrus, 2018; Yasunaga & Liang, 2020; Wang et al., 2024; Bouzenia et al., 2025; Tang et al., 2024); proof assistants make the checker stricter through type-theoretic elaboration and goal-state feedback.

3. Problem Setup and Pipeline Overview

Figure 1 summarizes M2F. Given a long-form source $\mathcal{D}$, M2F produces a Lean project $\mathcal{P}$ that elaborates end-to-end under a pinned dependency revision. Statement compilation builds a project-local interface layer; proof repair stress-tests local consistency by closing holes under fixed signatures. Both stages use the same refinement protocol: Lean re-

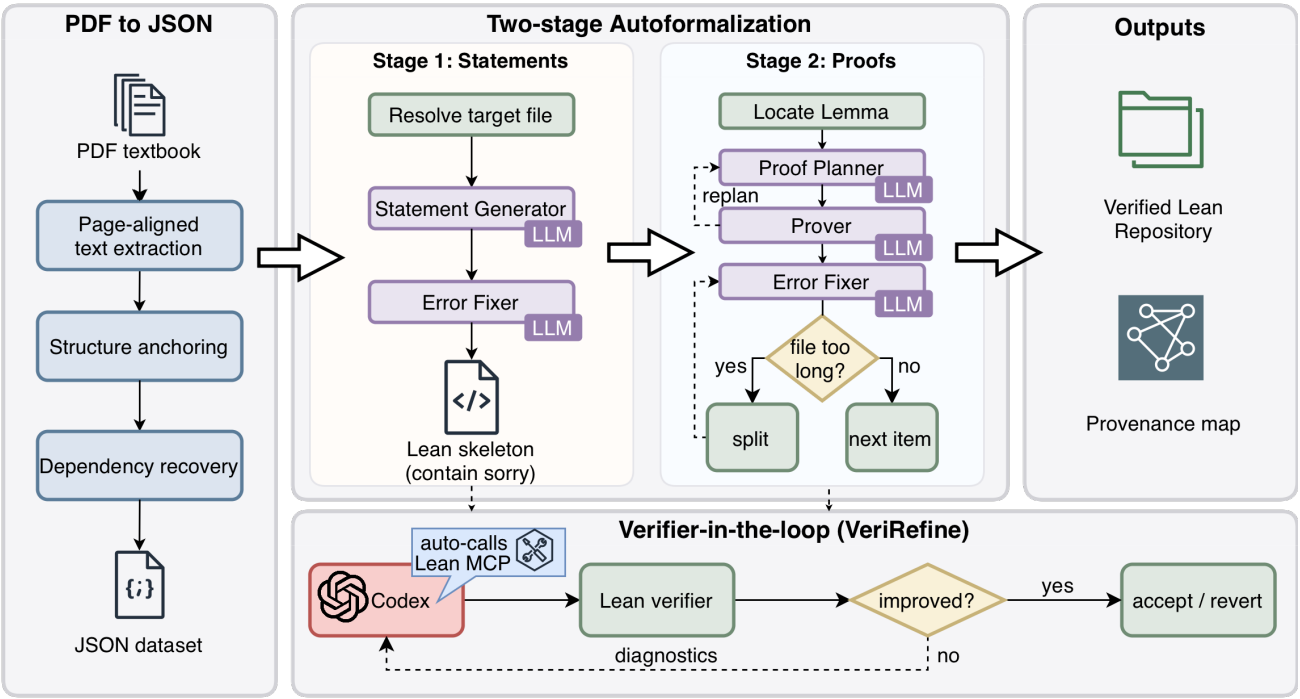


Figure 1. The M2F pipeline for project-scale automated formalization.

turns range-annotated feedback, and each bounded edit is accepted only when it improves the verifier-defined objective.

3.1. Input, Environment, and Output

The source $\mathcal{D}$ is normalized into ordered JSON items by the PDF/LaTeX-to-JSON preprocessing in Figure 1, which performs page-aligned extraction, structure anchoring, and dependency recovery. Let $\mathrm{Span}(\mathcal{D})$ be the project-local source spans carried by these items. All verification is performed under a fixed Lean environment $\mathcal{E}$, so build outcomes and diagnostics are comparable across runs.

We use the toolchain through project-level and file-level verification oracles:

$$\textsc{VerifyProj}_{\mathcal{E}}(\mathcal{P}) \to (ok, \Delta),$$
$$\textsc{VerifyFile}_{\mathcal{E}}(\mathcal{P}, f) \to (ok, \Delta_f).$$

Here Δ and Δ_f are finite multisets of range-annotated diagnostics. We write $\mathrm{err}(\cdot)$ for error-level diagnostics and treat ok as $\mathrm{err} = 0$. Warnings do not affect success. Provenance spans live in source coordinates, while diagnostic ranges live in Lean file coordinates.

The output is a Lean project $\mathcal{P}$ and provenance map $\pi : \mathrm{Decl}(\mathcal{P}) \to 2^{\mathrm{Span}(\mathcal{D})}$, where $\mathrm{Decl}(\mathcal{P})$ is the set of globally named declarations. Stage 1 permits `sorry` in proof bodies but requires all modules and imports to elaborate. Stage 2 targets no remaining `sorry`. The codomain is a set because one Lean declaration may depend on several project-local source spans, such as a statement, notation, hidden hypothesis, or proof paragraph. Conversely, one source item may generate several declarations.

This output view treats a declaration as a durable interface, not merely as a theorem to be solved once. A useful project artifact should remain usable after later statements depend on it, expose local APIs that connect back to canonical `mathlib` concepts when they overlap, and make provenance inspectable when a statement proves hard to use. Lean checks the formal object under $\mathcal{E}$; source faithfulness and global API quality are evaluated separately.

We use $\textsc{VerifyProj}$ for reporting project-level buildability. Refinement is checked at file granularity: each edit re-invokes $\textsc{VerifyFile}$ on the touched file and is accepted or reverted solely from toolchain output. The contract is environment-parametric only in this limited sense: it is not hard-coded to a specific Lean/`mathlib` release, but changing $\mathcal{E}$ changes names, elaboration behavior, and proof obligations and requires rerunning refinement.

3.2. Two-Stage Pipeline

M2F separates statement-layer construction from proof repair. At document scale, proof repair is meaningful only after imports, namespaces, types, and dependencies are project-stable.

In Stage 1, M2F maps $\mathcal{D}$ to a project $\mathcal{P}_1$ that elaborates end-

to-end under $\mathcal{E}$ while allowing proof placeholders. Each JSON item receives a target file; an LLM proposes a Lean declaration skeleton; compilation failures expose localized repairs for imports, namespaces, types, and missing prerequisites. After each patch, M2F re-runs VERIFYFILE and commits only verifier-improving edits. This is interface construction: definitions, hypotheses, and local lemmas must support later files without silently strengthening the source.

In Stage 2, M2F takes $\mathcal{P}_1$ and closes remaining proof holes to obtain $\mathcal{P}_2$ under matched statements. Existing signatures are fixed; edits are restricted to proofs and optional local helpers. The system locates each hole, queries goals when available, proposes a proof patch, and triggers bounded repair or replanning after failures. Compute is reported in VERIFYFILE calls. Thus proof repair is not used to silently change the mathematical interface. It is a stress test of whether the Stage 1 declarations expose enough local structure for later Lean code to use them.

4. Method: VeriRefine for M2F

VeriRefine is the accept/revert contract used by both stages. For project state $\mathcal{P}$, file f, and bounded scope S, the system snapshots f, proposes a patch, re-runs $\text{VERIFYFILE}_{\mathcal{E}}(\mathcal{P}, f)$, and commits only if the verifier-defined objective strictly improves. The primary objective is the number of Lean error-level diagnostics; Stage 1 breaks ties by localized diagnostics, and Stage 2 by remaining proof holes. A proof edit that closes a `sorry` but introduces an elaboration error is rejected. Full definitions, pseudocode, and schemas are in Appendix A–D. The proposal policy and the acceptance signal are deliberately separated: models may use source text, retrieval, LSP context, and previous failures, but success is defined only by the pinned verifier objective.

This contract is a stability mechanism, not a global optimizer. It prevents long-horizon runs from drifting into worse states and makes verifier-call budgets interpretable. Forward progress comes from candidate regeneration, replanning, rescoping, typed stubs for ordering constraints, and deterministic file splitting. The method can still get stuck within a finite budget.

The diagnostic value of such failures is part of the method. If repeated proof repair fails under a fixed signature, the trace often points to missing imports, absent helper interfaces, over-local definitions, or poor library navigation. M2F therefore uses proof repair not only to close holes, but also to expose where the statement layer lacks reusable infrastructure.

Stage 1: statement compilation. The input stream consists of ordered JSON items with labels, source spans, section context, and dependency hints. M2F preserves source order and hierarchy, augments items with theorem numbers, named definitions, and nearby context, and uses the dependency graph as an initial schedule. For each scheduled item, the model proposes a Lean declaration skeleton, usually with a `sorry`. Compilation failures expose missing imports, namespace mismatches, type errors, and missing prerequisites. Dependency recovery is heuristic; Lean-level well-formedness is checked by elaboration. Operationally, this is a dependency-aware Agent A/B/C loop (Appendix A.1, Algorithm 1): failed predecessors block dependents; Agent A proposes the initial statement/interface; source-level axiom-like outputs are rejected; Lean diagnostics, static quality-gate failures, and optional semantic-drift feedback are routed to Agent B for bounded repair. Agent C is only a review signal with configurable policies: `warn` records drift, `fix` routes it to repair, and `fail` rejects the item. A compile-improving patch may still be challenged by a source-facing check.

Stage 2: proof repair. Stage 2 starts from the buildable Stage 1 project and freezes statement signatures. It has an item-level proof mode and a file-level cleanup mode, both treated as proof repair rather than a third stage (Appendix A.2, Algorithms 2–3). In item-level mode, Agent C plans from the source proof/context, Agent A implements the proof, Lean checks it, and Agent B repairs compile failures. In cleanup mode, the system scans remaining `sorry` occurrences in a Lean file with no external natural-language hint and proves them from local Lean context and history. In both modes, guardrails reject source-level axiom-like outputs, protected-signature edits, and changes that merely move or increase `sorry`. Local helper lemmas are allowed only when existing signatures remain fixed.

5. Experiments

We evaluate M2F as a buildable library artifact, not as an isolated prover. The questions form an evidence chain: buildability, statement credibility, nontrivial reuse, source-local infrastructure, and finally proof repair under already elaborating statements. This separates two central risks: Lean code that builds but says the wrong mathematics, and faithful-looking statements that do not compose into a usable project.

- **Q1:** Can long source roots be compiled into buildable, proof-closed Lean projects?
- **Q2:** Are matched statements semantically acceptable?
- **Q3:** Is the artifact more than direct retrieval from existing libraries?
- **Q4:** Does the generated code behave like reusable source-local infrastructure?
- **Q5:** Under already elaborating statements, which proof-side failures remain?

This order keeps project buildability, statement-layer credibility, reusable infrastructure, and fixed-statement proof repair separate.

5.1. Setup

The source set covers Tao's *Analysis II* (Tao, 2022), Stacks topology and category-theory material (The Stacks Project Authors, 2026), Maassarani's algebra paper (Maassarani, 2025), Nesterov's optimization paper (Nesterov, 2005), Lebl's *Introduction to Real Analysis, Volume I* (Lebl, 2025)[1], and Rockafellar's *Convex Analysis* (Rockafellar, 1970). This mix covers mature and sparse-library areas, textbook exposition, and research-paper arguments. For the subset with complete logs and provenance audit, we report compilation, repair, verifier/model calls, and matched-statement proof closure. FATE-H[2] is reserved for the fixed-statement Stage 2 question.

All runs use Lean `v4.26.0` under the same pinned environment $\mathcal{E}$ and hardware (32 vCPU Intel Xeon 6982P-C, 123 GiB RAM, 1 TB NVMe SSD). *Buildable* means zero Lean error-level diagnostics; *proof-closed* means no remaining target proof holes. These terms do not imply source alignment, semantic equivalence, or `mathlib`-ready abstraction. Long-form runs took roughly one month wall-clock time. LLM interactions use Codex CLI with `model_reasoning_effort="high"`, and Lean tool queries use an LSP-based MCP server[3]. Thus our experiments evaluate this project-scale workflow under a strong base model; they are not a model-sweep or a model-agnostic performance claim. The supplementary archive contains generated Lean outputs, orchestration code, prompt templates, agent configurations, and the document-to-item parser; exact raw-source-to-Lean reruns also require source documents and model credentials.

Statement quality and source alignment are evaluated separately from build/proof success. The audit in Table 2 compares matched Lean statements against project-local source items, checking conclusions, quantifiers, domain constraints, and extra Lean-side assumptions. It uses two independent expert passes with no communication during labeling and final adjudication afterwards (Appendix C). Auditors could use project context and navigation tools for inspection, so we present this as expert adjudication rather than a blind inter-annotator-agreement study. This is evidence about the matched item layer, not a full PDF-to-Lean semantic-equivalence certificate. Appendix C separately reports source-to-JSON audits: 98/100 acceptable extractions in a combined Tao/Lebl/Local Rings sample, and 188/200 in targeted Rockafellar/Stacks/Nesterov samples. These checks inspect source artifacts or recovered excerpts $\rightarrow$ project-local JSON items. They strengthen preprocessing evidence, but neither JSON-item accounting nor matched-statement audit is a full original-document coverage denominator. The appendix also records the ledger schema, A/B disagreements, adjudication rule, and taxonomy of 66 modeling concerns; because this is expert adjudication rather than a strict blind IAA study, we do not report Cohen's κ.

5.2. Project Artifacts and Statement Compilation

For **Q1**, the primary output of M2F is a buildable Lean library artifact rather than isolated proofs. Table 1 reports seven source-aligned Lean developments that build under $\mathcal{E}$ with no explicit bypass constructs: 591 files, 10,637 declarations, and 379,366 physical lines. Here *source-aligned* means organized against project-local source items, not automatically source-preserving. The static pass finds zero uncommented `sorry`/`admit`, source-level axiom-like declarations, unsafe or partial declarations, heartbeat-disabling options, meta/macro commands, imports of modules named `Unsafe`, or simple explicit inconsistency patterns such as `False` binders. These checks do not rule out all vacuity or over-strong hypotheses; they remove the easiest mechanical bypasses. Whole-source-root build and proof closure are not global source-alignment guarantees, but they are strong local-coherence tests: definitions and statements must support later chapters without type mismatches, import cycles, or unusable APIs.

The released repository[4] contains the generated outputs under the pinned Lean/`mathlib` environment. It builds as a Lean workspace without post-hoc human cleanup. Reported status is based on Lean compilation; aggregate counts are computed from the released Lean sources.

Buildability is only the first gate. For **Q2**, Table 2 reports 2,023/2,089 acceptable matched statements; 66 raise modeling concerns and are not counted as successes. This denominator is the set of matched auditable Lean statements, not all mathematical statements in the original documents. For the two largest corpora, however, this denominator is close to the full set of extracted JSON items—the audit covers $523/526$ Tao and $840/875$ Rockafellar items (97.3% combined, and 65% of all audited statements)—so little filtering occurs at the JSON-to-Lean matching step, even though document-to-JSON recall remains a separate question (Appendix C). A Lean proof checks a fixed formal statement under $\mathcal{E}$, not whether that statement is the right mathematical interface for the source. Thus build/proof closure supplies local project-consistency evidence, while the statement layer needs a separate source-facing audit.

The audit labels are deliberately stricter than buildability. A

[1] https://github.com/jirilebl/real-analysis.
[2] https://github.com/frenzymath/FATE-H.git
[3] https://github.com/oOo0oOo/lean-lsp-mcp.git
[4] https://github.com/imathwy/M2F_repo.

Table 1. Released buildable artifact inventory. Decls are source-level Lean declarations counted after stripping comments. "Build+scan" means build success plus the strengthened static bypass scan described in the text.

Domain	Source	Size	Files	Decls	LoC	Build+scan
Analysis	Tao, *Analysis II*	195 pp	100	2,360	50,287	✓
Topology	Stacks Topology Ch. 5	30 sec	37	695	17,075	✓
Category theory	Stacks Categories Ch. 4 (20 sec)	20 sec	23	349	6,540	✓
Algebra	Maassarani, *On Some Local Rings*	6 pp	9	78	4,992	✓
Optimization	Nesterov 2005	27 pp	26	353	16,147	✓
Analysis	Lebl, *Intro. to Real Analysis I*	312 pp	57	1,201	37,179	✓
Convex analysis	Rockafellar, *Convex Analysis* (partial)	468 pp	332	5,601	247,057	✓
Workspace	Import aggregators	–	7	0	89	✓
Total	released workspace	–	591	10,637	379,366	✓

Table 2. Matched-statement audit. "Acceptable" combines source-preserving statements and minor non-semantic issues; "Concern" denotes a possible semantic/modeling change.

Corpus	Audited	Preserving	Minor	Acceptable	Concern
Tao Analysis II	523	241	274	515 (98.5%)	8 (1.5%)
Stacks Topology Ch. 5	142	40	91	131 (92.3%)	11 (7.7%)
Stacks Categories Ch. 4 (20 sec)	86	15	70	85 (98.8%)	1 (1.2%)
Local Rings	19	3	15	18 (94.7%)	1 (5.3%)
Nesterov 2005	65	35	21	56 (86.2%)	9 (13.8%)
Lebl Real Analysis I	414	330	74	404 (97.6%)	10 (2.4%)
Rockafellar Convex Analysis	840	461	353	814 (96.9%)	26 (3.1%)
Total	**2,089**	**1,125**	**898**	**2,023 (96.8%)**	**66 (3.2%)**

statement is not credited merely because Lean can prove it; auditors check whether the declaration exposes the mathematical interface that a later development would want to reuse. Several high-quality matches are substantial source-facing theorem interfaces, not local helper lemmas. Tao's inverse function theorem keeps the open neighborhoods, local bijection, inverse differentiability, and inverse-derivative formula visible in one declaration. Rockafellar's bipolar theorem combines polar-cone structure, the bipolar identity, and indicator-function conjugacy. Nesterov's accelerated-rate theorem exposes an algorithm package together with its explicit convergence bound. These cases are useful because their conclusions are not isolated end points: later files can refer to the named interface instead of reconstructing the source argument from scratch.

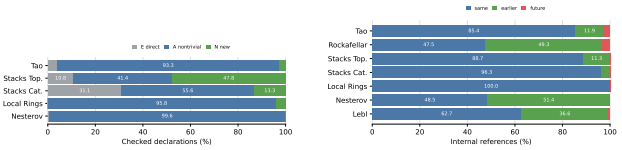
Conversely, the 66 modeling concerns show why proof closure alone is insufficient. Typical failures include definition-only matches for multi-component source constructions, partial coverage of multi-part propositions, fallback branches inserted to make a choice total, extra hypotheses that narrow a theorem, and source claims collapsed to easier special cases. Appendix Table 11 breaks this residual category into issue types; the largest groups are semantic-modeling risks and algorithmic/complexity abstraction mismatches. These examples are important for interpreting the numbers: M2F can build and prove a large local library, but statement quality still requires a separate gate. Appendix Tables 14 and 15 list representative positive and negative cases.

The declaration inventory also exposes the library-building burden. With a conservative docstring-marker rule, 2,407/10,637 declarations are directly source-matched and 8,230/10,637 are generated infrastructure. This is not a source-alignment metric; it shows that most generated Lean code consists of definitions, helper lemmas, compatibility facts, wrappers, and proof-route interfaces needed to make source-facing mathematics build and remain locally reusable. Appendix Table 35 gives the per-source split.

After buildability and matched-statement audit, **Q3** asks whether the artifact is more than direct retrieval. Figure 2 left classifies 943 declarations as direct reuse (E), nontrivial construction over related infrastructure (A), or new source-aligned construction (N). Only about 6% are E; about 94% require A or N work, indicating that the artifacts are not mostly one-step retrieval from `mathlib`, although this does not make every declaration mathematically novel.

Then **Q4** asks whether generated infrastructure accumulates in source order. Figure 2 right aligns Lean files with chapter/-section order and statically counts generated-to-generated references over seven JSON-aligned corpora. Most references are to the same or earlier section; future-section references are at most 3.2% in every corpus and 0.7% or below for Stacks, Local Rings, Nesterov, and Lebl. This is a source-level proxy, not a source-alignment proof or elaborator dependency graph. Still, it tests an important failure mode for long-form formalization: if the system produced isolated one-off theorems, later files would not consistently depend on earlier generated interfaces. The high same-or-earlier rate therefore suggests the repository behaves like a growing local library rather than a collection of disconnected theorem snippets. The proxy is also intentionally

conservative. It ignores uses of external `mathlib` APIs and counts only generated declarations calling other generated declarations. A high same-or-earlier rate therefore indicates that later generated files can find reusable project-local interfaces without circular lookahead. This does not prove that each interface is the best abstraction, but it is evidence that the artifact has usable internal structure. We use it as a library-coherence check, complementary to semantic audit and proof closure.



(a) Overlap with existing library facts.

(b) Source-order location of dependencies.

Figure 2. Visual summary of artifact-level infrastructure evidence. Left: most checked declarations require nontrivial construction or new source-aligned construction rather than direct reuse. Right: generated internal references mostly stay within the same source section or point backward. Exact counts are in Appendix Tables 36 and 37.

Table 3 reports downstream reuse and maintainability stress indicators. Across the released workspace, 8,314/10,637 declarations are referenced downstream and 2,176 cross file boundaries, supporting the narrow claim that later generated files use these interfaces as a local library. This is a stricter signal than raw declaration count: unused declarations may still compile, but they do not show that the artifact supports later mathematics. The stress columns expose helper/source ratio, local dependency surface, theorem-header hypothesis counts, and proof-size outliers. They are not quality scores: high reuse is necessary but not sufficient for good abstraction, while long proofs or high helper density show that buildable generated code is not automatically polished, human-maintainable, or upstream-ready `mathlib` code. We therefore claim reusable project-local infrastructure, not globally optimal public-library interfaces. Appendix Table 17 gives a non-human first-pass maintainability screen, and the theorem-header scan finds no explicit contradiction-looking `False`, `Not True`, or `False ->` patterns.

Representative high-reuse APIs are listed in Appendix Table 34. We also use compatibility certificates: when a local API overlaps with a canonical `mathlib` concept, the artifact should include an equality or iff theorem connecting the interfaces. These build-checked certificates guard against local API drift but do not solve the harder abstraction problem; for example, parts of the convex-analysis artifact are source-aligned over $\mathbb{R}^n$ but not generalized to normed or Hilbert spaces. Appendix Tables 21–22 report the aggregate scan and examples, and Appendix Table 39 gives source-to-Lean route certificates. The point is cumulative: a declaration that is source-matched, reused downstream, ordered before its consumers, and connected to canonical APIs is harder to dismiss as an isolated provable artifact. These proxies still fall short of mechanized semantic equivalence or global `mathlib`-quality, but they test whether later results can call stable intermediate APIs instead of rederiving local facts. They also make review more concrete: an auditor can inspect not only a final theorem, but the route of definitions and lemmas that later files actually use. The stress columns identify where expert refactoring would be needed before upstreaming.

For the instrumented subset, Stage 1 inserts each atomic block with `sorry` and repairs until the surrounding file has zero error-level diagnostics, under cap $K = 3$. Table 4 shows that all blocks compile. ARR remains far below 1.0 (0.42/0.08/0.20), giving conservative first-pass elaboration lower bounds of 58%/92%/80%; verifier/model-call totals are in Appendix E.

A separate statement-only ablation asks which parts of Stage 1 are needed before proof repair. We use a 180-item cross-domain prefix suite and a harder 74-item Convex Analysis stress subset. The metric is statement-stage compilability only: proofs may remain `sorry`, and statement quality/source alignment is not scored here. Table 5 shows that removing Agent B Lean-error repair is the dominant failure mode: pass rate drops from 97.8% to 71.1% cross-domain and from 82.4% to 18.9% on Convex Analysis. Thus the first proposal often gets close, but long-form projects need repeated local repair for imports, namespaces, typeclass assumptions, and predecessor dependencies. Semantic checking and static quality gates have smaller effects on raw compilability; their role is to prevent a compile-improving patch from silently changing the intended mathematical interface. On the Convex stress slice the semantic check even lowers raw compilability (full pipeline 82.4% versus 85.1% without it), which is expected under this metric rather than a regression: the check rejects statements that compile but drift from the source, and such rejections count as failures here even though they are the desired behavior. The gap is also small in absolute terms—61 versus 63 of 74 instances—so it is within two instances and not statistically meaningful on this slice.

5.3. Proof Repair under Matched Statements

After Q1–Q4 cover buildability, audit-gated statement evidence, and local reuse, **Q5** asks what remains hard once statements already elaborate. Stage 2 is evaluated under matched statements: each target proof is replaced by `sorry`, and methods may edit only proofs while preserving elaboration under $\mathcal{E}$. This removes statement drift as an escape route: a method cannot weaken the conclusion, add

Table 3. Within-artifact reuse and maintainability stress indicators. Reused and Cross-file are raw downstream-reference counts; API(%) is the static API-use proxy from the reuse scan; Helper/src counts generated infrastructure per source-matched declaration. Stress columns are static proxies.

Source	Decls	Reused	Cross-file	API(%)	Helper/src	Med deps	Avg/med hyps	Nonempty(%)	Med proof	Max proof / stmt
Tao, *Analysis II*	2,360	1,720	169	83.1	2.7	1.0	4.4/4.0	2.1	8.0	572 / 36
Stacks Topology Ch. 5	695	581	91	92.1	2.3	2.0	4.3/4.0	7.6	11.0	263 / 20
Stacks Categories Ch. 4 (20 sec)	349	240	9	72.1	1.8	1.0	4.1/3.0	22.4	6.5	224 / 17
Maassarani, *On Some Local Rings*	78	68	23	100.0	0.0	1.0	4.1/4.0	8.3	23.0	405 / 13
Nesterov 2005	353	289	99	80.0	0.1	2.0	7.8/7.0	10.6	25.0	493 / 29
Lebl, *Intro. to Real Analysis I*	1,201	715	83	88.9	0.1	1.0	3.0/3.0	2.4	13.0	711 / 46
Rockafellar, *Convex Analysis* (partial)	5,601	4,701	1,702	89.4	2.1	3.0	5.5/5.0	14.4	21.0	1,811 / 57
Total	10,637	8,314	2,176	86.3	1.7	2.0	5.0/4.0	10.1	15.0	1,811 / 57

Table 4. Stage 1 statement compilation on the instrumented audit subset; proofs may use `sorry`. ARR is mean bounded repair attempts per block.

Corpus	Blocks	Compiled	ARR	First-pass $\geq$	Build
Lebl Real Analysis I	416	416/416	0.42	58%	✓
Convex Analysis	560	560/560	0.08	92%	✓
Nesterov 2005	67	67/67	0.20	80%	✓

Table 5. Statement-pipeline ablation; the metric is Lean compilability of statement skeletons with proof placeholders. Percentages are shown in parentheses.

Ablation	Cross-domain	Convex stress	Main interpretation
Full pipeline	176/180 (97.8%)	61/74 (82.4%)	Repair + semantic warnings + quality gate
Semantic-check repair policy	174/180 (96.7%)	63/74 (85.1%)	Semantic warnings trigger repair
w/o semantic check	172/180 (95.6%)	63/74 (85.1%)	Drift/API-owner risk unchecked
w/o quality gate	174/180 (96.7%)	53/74 (71.6%)	Weaker structural filtering
w/o Agent B repair	128/180 (71.1%)	14/74 (18.9%)	Compile-guided repair is essential

Table 6. Proof-side controls under fixed, already elaborating statements. Long-form columns report proof success rates in percent. Hint JSON rows are lightly supervised infrastructure diagnostics: they name lemma roles but give no proof scripts, and are not used for compute-accounted comparison.

Method	Lebl	Convex	Nesterov
One-shot	45.45	63.94	46.15
Diagnostics-only	100	95.45	13.85
No replanning (C=1)	100	90.91	46.97
M2F (full Stage 2)	**100**	**100**	**100**

Method	Solved
Codex one-shot	31/100
Codex naive 5-loop	36/100
M2F (fully automatic)	**96/100**
M2F (+31 hint JSON)	**97/100**
M2F (+37 finer hint JSON)	**98/100**

stronger signature hypotheses, or change the surrounding API. Full Stage 2 closes all audited holes on the three instrumented corpora (875/875 retained obligations). Table 6 reports long-form proof-repair ablations and a fixed-statement FATE-H diagnostic probe; the latter is not used as an end-to-end source-to-Lean baseline. The Nesterov slice is most sensitive, with diagnostics-only collapsing to 13.85%.

FATE-H removes source parsing, statement generation, dependency recovery, and project compilation; it is a missing-infrastructure diagnosis, not a prover leaderboard. M2F reaches 96/100 fully automatically, while same-model one-shot and naive 5-loop controls solve 31/100 and 36/100. Human-written lemma-hint JSON exposes reusable declaration handles and roles, raising the result to 98/100 without supplying derivations; these supervised rows diagnose library navigation and are not cost-accounted in the same stream. The four automatic failures comprise two infrastructure-assisted cases, one benchmark statement inconsistent under $\mathcal{E}$, and one remaining open case.

This pattern matches the long-form artifacts: once a statement is fixed, many failures concern the right intermediate interface, imported theorem, or reusable lemma at the right abstraction level. A prover cannot use a definition that has not been introduced, a theorem that has not been stated properly, or an API whose connection to `mathlib` is missing. Stage 2 therefore does more than close holes: it stress-tests whether Stage 1 declarations are usable by later mathematics. In this sense, a failed proof is often a useful infrastructure signal: it points to a missing abstraction, a poorly exposed lemma, or a local API that should be connected to the surrounding library. Conversely, a short proof is meaningful only when the statement exposes a reusable interface rather than a one-off encoding. This is why our evaluation treats proof repair as evidence about the statement layer, not as a replacement for statement-quality audit.

6. Conclusion

M2F reframes long-form autoformalization as project-scale infrastructure construction. The central object is not an isolated proof, but a buildable Lean workspace with source-

indexed statements, reusable declarations, imports, and dependencies. Across analysis, topology, category theory, algebra, optimization, and convex analysis, the released artifacts show that evaluated source roots can yield proof-closed project-local developments under a pinned environment. We do not claim that proof search is solved, generated code is human-polished, or generated statements are globally `mathlib`-optimal. Our narrower conclusion is that current proof technology is strong enough to expose a practical bottleneck: building reusable formal infrastructure quickly and correctly.

This perspective also changes what should be measured next. Future systems should be judged not only by proof success on fixed statements, but by whether their generated interfaces survive downstream use, connect to canonical libraries, and remain inspectable by domain experts. M2F is a step toward that evaluation regime: proofs provide verifier feedback, while the lasting artifact is the formal infrastructure that later mathematics can build on.

References

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics, 2023. URL https://arxiv.org/abs/2302.12433.

Bouzenia, I., Devanbu, P., and Pradel, M. Repairagent: An autonomous, LLM-based agent for program repair, 2025.

Cabral, R., Do, T. M., Xuejun, Y., Tai, W. M., Feng, Z., and Shen, X. ProofFlow: A dependency graph approach to faithful proof autoformalization, 2025.

Chen, J., Chen, W., Du, J., Hu, J., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Shi, W., Wang, Z., Wang, M., Wei, C., Wei, S., Xin, H., Yang, F., Gao, W., Yuan, Z., Zhan, T., Zheng, Z., Zhou, T., and Zhu, T. H. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025. URL https://arxiv.org/abs/2512.17260.

de Moura, L. and Ullrich, S. The Lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G. (eds.), *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

de Moura, L. M., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. The Lean theorem prover (system description). In Felty, A. P. and Middeldorp, A. (eds.), *Automated Deduction – CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pp. 378–388. Springer, 2015. doi: 10.1007/978-3-319-21401-6_26. URL https://doi.org/10.1007/978-3-319-21401-6_26.

Gao, G., Wang, Y., Jiang, J., Gao, Q., Qin, Z., Xu, T., and Dong, B. Herald: A natural language annotated Lean 4 dataset. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. doi: 10.48550/arXiv.2410.10878. URL https://openreview.net/forum?id=Se6MgCtRhz.

Gulati, A., Ladsaria, D., Mishra, S., Sidhu, J., and Miranda, B. An evaluation benchmark for autoformalization in Lean4, 2024. URL https://arxiv.org/abs/2406.06555.

Guo, Q., Wang, J., Zhang, J., Kong, D., Huang, X., Xi, X., Wang, W., Wang, J., Cai, X., Zhang, S., and Ye, W. Autoformalizer with tool feedback, 2025.

Jiang, J., He, W., Wang, Y., Gao, G., Hu, Y., Wang, J., Guan, N., Wu, P., Dai, C., Xiao, L., and Dong, B. FATE: A formal benchmark series for frontier algebra of multiple difficulty levels, 2025. URL https://arxiv.org/abs/2511.02872.

Lebl, J. *Introduction to Real Analysis, Volume I*. Self-published online notes, 2025. Available online at https://github.com/jirilebl/real-analysis.

Li, C., Ma, W., Wang, Z., and Wen, Z. SITA: A framework for structure-to-instance theorem autoformalization, 2026. URL https://doi.org/10.1609/aaai.v40i23.38997.

Lu, J., Wan, Y., Huang, Y., Xiong, J., Liu, Z., and Guo, Z. FormalAlign: Automated alignment evaluation for autoformalization, 2024a. URL https://arxiv.org/abs/2410.10135.

Lu, J., Wan, Y., Liu, Z., Huang, Y., Xiong, J., Liu, C., Shen, J., Jin, H., Zhang, J., Wang, H., Yang, Z., Tang, J., and Guo, Z. Process-driven autoformalization in Lean 4, 2024b. URL https://arxiv.org/abs/2406.01940.

Maassarani, M. On some local rings. arXiv:2512.19197v1 [math.AC], 2025. URL https://arxiv.org/abs/2512.19197.

Monperrus, M. Automatic software repair: A bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24, 2018. doi: 10.1145/3105906. URL https://doi.org/10.1145/3105906.

Nesterov, Y. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1):127–152, 2005. doi: 10.1007/s10107-004-0552-5.

Ospanov, A. and Yousefzadeh, R. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning, 2025. URL https://arxiv.org/abs/2505.05758.

Poiroux, A., Weiss, G., Kunčak, V., and Bosselut, A. Reliable evaluation and benchmarks for statement autoformalization, 2025.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., and Ruan, C. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Rockafellar, R. T. *Convex Analysis*, volume 28 of *Princeton Mathematical Series*. Princeton University Press, Princeton, NJ, 1970.

Tang, H., Hu, K., Zhou, J. P., Zhong, S., Zheng, W.-L., Si, X., and Ellis, K. Code repair with LLMs gives an exploration-exploitation tradeoff, 2024.

Tao, T. *Analysis II*, volume 38 of *Texts and Readings in Mathematics*. Hindustan Book Agency, 4 edition, 2022. ISBN 9788195196128.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*. Association for Computing Machinery, 2020. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

The Stacks Project Authors. The stacks project. https://stacks.math.columbia.edu, 2026. Accessed: 2026-04-05.

Wang, H., Liu, Z., Wang, S., Cui, G., Ding, N., Liu, Z., and Yu, G. INTERVENOR: Prompting the coding ability of large language models with the interactive chain of repair. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024. doi: 10.18653/v1/2024.findings-acl.124. URL https://aclanthology.org/2024.findings-acl.124/.

Wang, H., Xie, R., Wang, Y., Gao, G., Yu, X., and Dong, B. Aria: An agent for retrieval and iterative autoformalization via dependency graph, 2025.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368, 2022.

Xin, H., Ren, Z. Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., Gao, W., Zhu, Q., Yang, D., Gou, Z., Wu, Z. F., Luo, F., and Ruan, C. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search, 2024. URL https://arxiv.org/abs/2408.08152.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Yasunaga, M. and Liang, P. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*. PMLR, 2020. URL https://proceedings.mlr.press/v119/yasunaga20a.html.

Ying, H., Wu, Z., Geng, Y., Wang, J., Lin, D., and Chen, K. Lean workbook: A large-scale Lean problem set formalized from natural language math problems, 2024.

Zhang, L., Quan, X., and Freitas, A. Consistent autoformalization for constructing mathematical libraries. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 4020–4033. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.233. URL https://aclanthology.org/2024.emnlp-main.233/.

Zhang, L., Valentino, M., and Freitas, A. Autoformalization in the wild: Assessing LLMs on real-world mathematical definitions. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 1720–1738. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.90. URL https://aclanthology.org/2025.emnlp-main.90/.

Zheng, K., Han, J. M., and Polu, S. MiniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations (ICLR)*, 2022. doi: 10.48550/arXiv.2109.00110. URL https://openreview.net/forum?id=9ZPegFuFTFv.

A. Detailed VeriRefine Specification

This appendix gives the formal details behind the compact description in §4.

Reader guide. Appendix A specifies the accept/revert contract and stage control flow. Appendix C gives the audit protocol, source-to-JSON boundary, row-level ledger schema, disagreement summary, and modeling-concern taxonomy. Appendix E defines verifier/model-call accounting. Appendix F records failure modes that can survive Lean buildability. Appendix G gives qualitative artifact examples, source-order/reuse statistics, and representative source-to-Lean route certificates. Appendix H and Appendix I state limitations and broader impacts at the end.

For a fixed Lean environment $\mathcal{E}$, file verification returns $\textsc{VerifyFile}_{\mathcal{E}}(\mathcal{P}, f) \to (ok, \Delta_f)$, where Δ_f is a multiset of range-annotated diagnostics and ok abbreviates $\mathrm{err}(\Delta_f) = 0$. A scope S is a finite union of Lean source ranges, normally the current declaration plus a bounded header region for imports, namespaces, and section variables. Diagnostic localization is $\textsc{Localize}(\Delta_f, S)$, and $\mathrm{err}_S(\Delta_f, S)$ counts error-level diagnostics whose range intersects S. When the file has no error-level diagnostics, M2F may query the elaborated goal state at a hole; this goal/context information conditions proposal only and is never an acceptance signal.

All edits are executed through a single TryPatch wrapper: snapshot the target file, apply the candidate patch within S, run VerifyFile, and accept only if the stage objective strictly improves. Stage 1 accepts iff $(\mathrm{err}(\Delta'_f), \mathrm{err}_S(\Delta'_f, S)) \prec (\mathrm{err}(\Delta_f), \mathrm{err}_S(\Delta_f, S))$, where $\prec$ is lexicographic order. Stage 2 accepts iff $(\mathrm{err}(\Delta'_f), |\mathcal{H}'(f)|) \prec (\mathrm{err}(\Delta_f), |\mathcal{H}(f)|)$, where $\mathcal{H}(f)$ counts remaining proof holes such as `sorry`. Rejected patches are rolled back, so accepted states are monotone under the verifier objective.

The proposal layer uses six operator families: GenSkeleton inserts a Lean declaration skeleton from a statement item; RepairPatch and FixCompileError target localized diagnostics; Plan/Replan produce proof plans from source proof text, goal type, and local context; ProposeProofPatch emits hole-closing edits; and SplitIfLargeAndResolve splits long files at declaration boundaries while preserving namespace context and target labels. These operators may use LLM calls, retrieval, and Lean IDE queries, but acceptance is always performed by TryPatch.

A.1. Statement-stage control flow

Stage 1 is an item-level generate–check–repair loop with dependency blocking. Each input item contains the source statement, label, chapter/section context, dependency hints, and target Lean file. The output is an item status such as `ok`, `ok_after_b`, `failed_after_b`, `blocked_by_dependency`, `agent_a_failed`, or `semantic_failed`. Theorem and lemma declarations may use `:= sorry` during Stage 1, but source-level `axiom`-like outputs are rejected or cleaned before the item can be accepted.

Agent A is the semantic translator: it decides whether to recall an existing interface or introduce a new one, chooses `def`/`abbrev`/`lemma`/`theorem`/`#check`-style declarations, names the main declaration, and may split one source item into several atomic declarations. Lean checking catches syntax errors, missing names, typeclass failures, binder errors, insufficient imports, and ill-typed statements; it does not catch missing hypotheses, wrong implication direction, over-specialization, over-generalization, or semantic drift that remains type-correct. The static quality gate filters interface-level failures that are cheap to detect: placeholder-like declaration names, missing source labels in docstrings/comments, source-level `axiom`, core definitions left as placeholders, large conjunction wrappers replacing natural APIs, and duplicated local aliases for canonical APIs. Agent C compares the source item, Lean snippet, context, dependency hints, and declaration metadata, and returns semantic feedback for missing assumptions, domain changes, wrong direction, surrogate objects, non-canonical APIs, or partial formalizations. Agent B is the Lean repair agent: it edits the current file using Lean diagnostics, aggregate build errors, quality-gate failures, and Agent C feedback. The repair budget is configurable; the instrumented ablation in Table 5 reports its own fixed setting, while the item-level runner supports a larger default budget.

A.2. Proof-stage control flow, including cleanup mode

Stage 2 eliminates proof placeholders after the statement layer is buildable. We describe both operational modes under the single name *proof repair*. The item-level mode is aligned to source items and can use the source proof/context. The file-level cleanup mode scans remaining `sorry` occurrences in Lean files after item-level repair; it sets the external natural-language hint to empty and relies on the Lean declaration, imports, local context, and proof history. This cleanup pass is therefore not a separate scientific stage in the paper, but a stricter proof-repair mode for residual holes.

Algorithm 1 Statement-stage item loop

Input: Source items; repair budget K; semantic policy in {`warn`, `fix`, `fail`}.
for item in dependency order **do**
  if some predecessor failed **then**
    record `blocked_by_dependency`; continue
  end if
  $y \leftarrow$ Agent A(item) *(choose APIs, declaration kind, names, imports, variables)*
  if Agent A fails **then**
    record `agent_a_failed`; continue
  end if
  reject or clean source-level `axiom`-like output
  for $t = 0, \ldots, K$ **do**
    run Lean on the target file
    run the static quality gate when Lean succeeds
    optionally run Agent C semantic review when Lean and quality checks succeed
    if Lean, quality, and semantic checks pass **then**
      record `ok` if $t = 0$, else `ok_after_b`; break
    end if
    if Agent C reports drift and policy is `warn` **then**
      record success with semantic warning; break
    end if
    if Agent C reports drift and policy is `fail` **then**
      record `semantic_failed`; break
    end if
    if $t = K$ **then**
      record `failed_after_b`; break
    end if
    Agent B repairs the current Lean file from Lean, quality-gate, and semantic feedback
  end for
end for

Common item-level statuses include `ok`, `ok_after_b`, `failed_after_b`, `failed_no_progress`, `failed_bad_statement`, `abandoned_in_planning`, `skipped_empty_proof`, `skipped_recall_only`, and `skipped_check_only`. The distinction between `failed_bad_statement` and ordinary proof failure is important: proof repair should not hide an incorrect or under-specified statement by endlessly searching for tactics.

Meaningful progress is stricter than textual change. It includes reducing the target declaration's `sorry` count, reducing the file-level `sorry` count, completing a newly introduced helper declaration, or detecting that the target was already solved. Pure formatting, comment edits, imports that do not help verification, or moving a placeholder are not counted as progress. Agent C is a proof planner in Stage 2, not the semantic reviewer used in Stage 1: it reads the statement, source proof/context when available, dependencies, current Lean snippet, proof history, and Agent A feedback, then proposes a route and possible helper lemmas. Agent A executes that route while preserving statement signatures. Agent B is reserved for compiler-level failures after a proof edit has been attempted.

The cleanup mode is intentionally conservative. It receives no external source proof sketch or lemma hint, so success means the remaining hole can be solved from the Lean context, existing imports, and accumulated project infrastructure. Its guardrails prevent the system from manufacturing apparent progress by changing protected theorem headers, adding unfinished helper lemmas, increasing the number of `sorry` occurrences, or moving a placeholder to another declaration. When a residual hole cannot be closed, structured outcomes such as missing theory or bad statement are recorded as diagnostics for the statement/infrastructure layer rather than treated as ordinary prover failures.

B. Prompting Philosophy and Instrumentation Details

This appendix documents two cross-cutting design choices that shape our system across all stages. First, we summarize the prompting philosophy that we use to make the model produce *canonical, `mathlib`-aligned* Lean statements and stable proofs. Second, we describe the instrumentation semantics that make the pipeline auditable and enable the verifier-normalized compute accounting in Appendix E. The supplementary code release includes the prompt templates used by the

Algorithm 2 Stage 2 item-level proof repair

```
Input: Source items, Stage 1 Lean files, planning budget C, no-progress budget R, repair budget B.
for item in source order do
    if item has no proof task, is recall-only, or is #check-only then
        record skipped status; continue
    end if
    locate the label-associated Lean declaration
    feedback ← ∅
    for c = 0, ..., C do
        plan ← Agent C(item, target declaration, proof history, feedback)
        for r = 0, ..., R do
            snapshot file and count target/file sorry occurrences
            Agent A edits proofs and optional local helpers under fixed statement signatures
            reject or clean source-level axiom-like output
            if Agent A reports failed_bad_statement then
                record failed_bad_statement; continue outer item loop
            end if
            if Agent A requests replanning then
                feedback ← Agent A feedback; continue next plan round
            end if
            if target sorry count decreases, file sorry count decreases, or a helper is fully proved then
                go to Lean check
            end if
            retry Agent A with stricter no-cosmetic-edit instructions
        end for
        record failed_no_progress; continue outer item loop
    end for
    Lean check: run the target-file verifier and, when required, the aggregate book verifier
    if verification succeeds then
        record ok; continue
    end if
    for b = 1, ..., B do
        Agent B repairs Lean errors using the verifier log and item context
        if target-file and aggregate checks succeed then
            record ok_after_b; continue outer item loop
        end if
    end for
    record failed_after_b
end for
```

Algorithm 3 Stage 2 file-level cleanup mode

Input: Lean files after item-level repair; `nl_hint=None`; planning and repair budgets.
for file in selected files **do**
  while file contains `sorry` and target budget remains **do**
    choose the first remaining `sorry` as target and extract its declaration snippet
    plan ← Agent C(file, target, history, `nl_hint=None`)
    snapshot file and count file/target `sorry` occurrences
    Agent A attempts to eliminate the target `sorry`
    if guardrail detects a new `sorry`, a moved placeholder, protected-signature mutation, or source-level `axiom`-like output **then**
      rollback and feed guardrail reason to replanning
    else if target `sorry` is removed or target/file `sorry` count decreases **then**
      verify the file and continue to the next target
    else if Agent A reports `failed_bad_statement`, `failed_missing_theory`, or `needs_replan` **then**
      feed the structured reason to Agent C
    else
      retry with stricter progress requirements
    end if
  end while
  run the target-file verifier and, when enabled, the aggregate book verifier
  if aggregate imports fail **then**
    Agent B repairs only the aggregate import file when possible
  end if
  record the file-level result
end for

orchestration pipeline; here we summarize the design principles that guided prompt construction and the semantics of the logged artifacts.

B.1. Prompting philosophy

Our prompting philosophy is organized around one principle: **statement-first formalization**. Across stages, we treat "writing the right statement" as the most important driver of downstream success. In practice, a statement that is minimal, canonical, and aligned with existing `mathlib` interfaces can dramatically reduce proof difficulty by avoiding accidental typeclass obligations, brittle goal shapes, and unnecessary rewriting overhead.

Choosing the right declaration form. Prompts are designed to make the model choose standard Lean and `mathlib` declaration forms with canonical binders. We encourage `def` for definitions of data/functions/predicates, `abbrev` when lightweight unfolding behavior is desirable, and `structure` for packaging fields when typeclass inference is not intended. We use `class` only when downstream code should consume the structure via typeclass inference; otherwise `structure` is preferred. For propositions, we use `theorem` for results explicitly presented as theorems in the source and default to `lemma` otherwise. Finally, we reserve `instance` for canonical typeclass instances that should be found by inference, and we avoid inventing instances solely to make automation or rewriting easier.

Avoid inline proofs inside terms. A stability rule in our prompts is to avoid embedding proofs inside larger terms. Concretely, prompts discourage (and in some stages forbid) writing proof snippets such as `(by ...)` or $\langle \ldots, by \ldots \rangle$ inside a term-level construction. When a construction requires proof obligations (for example, closure properties for subtypes or structure fields), we design prompts to first introduce a separate helper `lemma` with a clean, minimal statement and then reference that lemma by name in the larger term. This yields flatter goal structure, improves readability, and makes later repair and proof search more modular.

Additional constraints for definitions. Definitions are handled more strictly than propositions. Prompts are designed so that the model does not use tactic-mode proofs inside a `def` body; definitions should be expressed as short term constructions. If a definition appears to require substantial proof work, prompts steer the model toward splitting: define the object/predicate with a clean `def` or `abbrev`, and move properties or obligations into separate lemmas. When a definition must be left incomplete, prompts prefer `:= sorry` (not `by sorry`) to avoid introducing tactic blocks into definitions.

Placeholders in Stage 1. Stage 1 allows placeholders to obtain a compilable statement layer. Prompts enforce a consistent policy: proposition-valued declarations use `:= sorry` as a proof placeholder, and definitions use `:= sorry` if a placeholder is necessary. The final reported artifacts are then checked for explicit bypass constructs: after comment

stripping they contain no `sorry` or `admit` tokens, no source-level `axiom`, `constant`, `opaque`, `unsafe`, or `partial` declarations, no heartbeat-disabling options, no meta/macro command surface, and no imports of modules named `Unsafe`.

A checklist for "good statements". Prompts implicitly encode (and our human review follows) a checklist that encourages canonical statements. We reuse existing `mathlib` structures and predicates whenever possible instead of introducing ad-hoc ones, minimize binders and keep them canonical, separate object definitions from properties, avoid inline proofs inside terms by factoring helpers, and avoid non-canonical instances.

Docstrings are required. Prompts require a short natural-language docstring immediately above each new declaration (`/- ... -/` in Lean). Docstrings are intended to improve maintainability and future reuse: they should briefly describe what the declaration is, why it exists, and (when applicable) how it relates to the source material. We avoid copying long verbatim text from the source; concise descriptions are preferred.

Proof stability and timeout robustness. When writing proofs (especially in Stage 2), prompts prioritize stable proof scripts and discourage forms that frequently trigger deterministic slowdowns. We avoid long, deeply nested chains of `have` statements and prefer flatter structures using standard combinators such as `intro`, `refine`, `calc`, `rw`, and `simp`. When a proof repeatedly requires many intermediate facts, prompts treat this as a signal that the statement or helper-lemma shape may be suboptimal; the model is encouraged to revise the statement layer or factor additional helpers rather than continuing a brittle proof search.

Stage-specific intent. Stage 1 aims to produce a Lean project that elaborates under the pinned environment while allowing proof placeholders, so prompts emphasize canonical signatures, minimal assumptions, consistent placeholder policy, and diagnostics-driven repairs with small, local edits. Stage 2 aims to eliminate target proof holes while preserving the audited statement layer, so prompts emphasize respecting matched statements (no signature edits), goal-directed proof search when goal/context is available, stable proof structure, and factoring nontrivial obligations into helper lemmas rather than embedding proofs inside terms.

How we package context to the model. A central tradeoff is over-conditioning (too much context causing unnecessary refactors) versus under-conditioning (too little context causing missing assumptions or wrong types). We adopt a minimal sufficient context principle: each prompt includes only the information needed for the current local decision. Across stages, prompt inputs may include the focal content (an atomic block or a small neighborhood around a target hole), explicit edit-scope constraints, localized verifier diagnostics or goal/context information, and a compact shared header/import/notation context. Windowing and truncation rules are fixed to keep prompts within model limits and to ensure repeatability; the corresponding artifact schemas that store prompt inputs are described in Appendix D.

B.2. Instrumentation semantics

Instrumentation is designed so that reported metrics can be reconstructed from machine-readable logs rather than manual bookkeeping. We log at attempt granularity: each model invocation and each verifier check produces a structured record, and runs also emit an append-only metrics stream. This section describes *what* we log and *how* it is used; Appendix D gives the corresponding JSON/JSONL schemas.

B.2.1. Artifact types and their semantics

We distinguish three levels of artifacts. First, per-call logs provide a full audit trail for each model call (inputs/outputs/diffs/-tokens) and are primarily used for debugging and token backfill. Second, compact JSON/JSONL stores provide structured, queryable traces: progress checkpoints for resumability, history stores for "memory" and debugging, and a metrics event log for analytics. Third, optional rollups provide human-friendly summaries of persistent failures.

Table 7 summarizes the semantic role of each artifact category. We emphasize semantic intent rather than on-disk paths; Appendix D specifies the corresponding machine-readable formats.

B.2.2. Recoverability: checkpoints and resumable runs

Long runs can be interrupted by timeouts, infrastructure instability, or budget limits. To make experiments recoverable, each pipeline maintains a single-key JSON checkpoint that stores the next target to process. We treat the exact key name and schema as part of the JSON artifact specification and describe it in Appendix D.3; Appendix B.2.2 explains the operational semantics (when the cursor advances, when it stops, and how runs resume deterministically).

Table 7. Instrumentation artifacts: content and primary use.

Artifact	Contents	Primary use
Per-call model log	Full stdout/stderr for one model invocation, including effective prompt, tool-wrapper outputs, and emitted diffs; may also include token markers	Per-call auditing/debugging; token backfill from stable markers
Checkpoint JSON	A single resume cursor (e.g., next item or next file) plus minimal run identifiers	Crash-safe resumability without reprocessing completed targets
History JSONL	Compact per-target records across retries (plan summaries, key errors, selected fields)	Debugging and optional continuity context for later attempts
Metrics log JSONL	Append-only event stream with run/item/file boundaries, verifier checks, and oracle/model-call outcomes	Primary source for paper metrics and verifier/oracle call accounting
Metrics summary JSON	Run-end aggregates that mirror the final `run_end` event payload	Quick run-level accounting without scanning JSONL logs
Token backfill JSONL	Per-target token totals reconstructed from stable token markers in per-call logs	Robust token accounting when metrics events omit per-call token fields
Failure rollup (optional)	Append-only human-readable list of persistent item/file failures with pointers	Fast triage and rerun planning

Why checkpointing matters for accounting. Checkpointing is designed so that totals computed from logs can be reconstructed without ambiguity. When a run is resumed, the resumed segment emits a new run id in the metrics stream, so corpus-level totals are computed by summing over the run ids included in the reported experiment. This design prevents double counting and makes it clear which targets were attempted in each segment.

B.2.3. Per-call logs: auditability and token extraction

Per-call logs are produced for every model invocation. They are human-readable and capture the transcript needed to inspect what the model saw and did during an archived run, including diffs applied to the Lean project when those logs are retained.

Per-call log delimiters and token footer. Per-call logs begin with a literal `STDOUT:` section and then a literal `STDERR:` section. Token usage is recorded as a two-line footer containing the literal marker `tokens used` followed by a decimal integer (commas may appear). Our tooling parses this with a robust regex equivalent to `tokens used\s*([0-9][0-9,]*)`.

Audit scope. Because the per-call log contains the effective prompt (as printed by the Codex wrapper) and the corresponding stdout/stderr transcript, it provides a complete audit trail for debugging and for reconstructing token usage even when metrics events omit per-call token fields. Per-call logs are also used to diagnose failures that involve toolchain behavior (e.g., unexpected verifier output or wrapper exceptions) that may not be captured in compact JSON summaries.

B.2.4. Structured stores: history and metrics

History store (JSONL). The history store is an append-only JSONL stream of compact per-task records. Each record includes a timestamp, a run id, a task identifier, a kind tag (plan, replan request, fix attempt, warning cleanup, etc.), and a payload object. The payload is designed to be lightweight and may truncate long strings to keep the store compact. When history augmentation is enabled, a small window of recent records for the current file/task is loaded and included in later prompts to improve continuity across retries without requiring the full per-call logs.

Metrics event log (JSONL). The metrics event log is the primary machine-readable instrumentation for paper metrics and accounting. It is an append-only JSONL stream where each line is a JSON object with a timestamp, a run id, an event label, and an event payload. Event labels cover run start/end, item/file start/end, verifier checks, and model-call results. When present, event payloads may additionally include file snapshot summaries (size, modification time, line counts) that allow change-size analyses without reading full diffs.

MCP/LSP tool queries are logged but not counted. During statement editing, item-level proof repair, and Stage 2 cleanup-mode editing, the wrapper may issue Lean LSP/MCP queries (for example: diagnostics refreshes, goal inspection, hover/type information, local search, completions, and running small Lean snippets) to support interactive refinement. These tool queries are recorded only in the raw per-call transcripts and are not emitted as structured metrics events. As a result, the oracle-call totals in Appendix E—which are derived from model-invocation events—do not include the frequency of LSP/MCP queries, and token totals do not include any cost attributable to these local tool queries. In practice, LSP/MCP

usage can contribute to wall-clock time and local compute, but it is excluded from our verifier-normalized and token-based accounting.

B.2.5. Implementation notes: resumability and file-local verification

All pipelines iterate deterministically over their target inventory (dataset indices for item-level stages, or a fixed file list for file-level cleanup). To support interruption and restart, each pipeline maintains a single JSON checkpoint that stores the next target cursor; on restart, the pipeline reloads this checkpoint and resumes without re-processing completed targets. This design avoids manual bookkeeping and enables reliable aggregation across runs: if a corpus/stage setting is completed across multiple resumptions, the reported totals simply sum across the corresponding run ids.

Verification is performed *file-locally* using a pinned Lean environment: the pipeline repeatedly runs a single-file Lean compile check on the currently edited file rather than rebuilding the entire project after every local edit. This is both a performance choice and an accounting choice: each such single-file compile check is exactly one verifier call in our cost model (Appendix E.1), and the corresponding diagnostics provide the accept/reject signal for commit/rollback in all stages.

For very large generated sections, an optional splitter can refactor a file into multiple parts connected by a deterministic import chain (e.g., `sectionYY_partK.lean` files imported by a thin aggregate module). When splitting is enabled, proof repair targets only the containing part file, keeping both verification and patch scope local and reducing instability caused by overly large files.

B.2.6. Mapping metrics events to paper metrics

The paper reports (i) success rates, (ii) verifier-normalized compute, (iii) oracle-call frequency, and (iv) token usage (as a complementary cost proxy). All of these can be derived from the metrics event log and (for tokens) optionally corroborated via token backfill.

Success and completion. For item-level pipelines, completion status is defined by item-end events that record whether the target item was processed successfully under budget. For file-level cleanup mode, completion status is defined by file-end and task-end events that record whether the file check succeeded and how many `sorry` occurrences were eliminated.

Verifier calls. Verifier-call totals are computed by counting verifier-check events (labeled `lean_check` in our logs) and aggregating across the run ids included in a reported corpus/stage setting. For some Stage 1 runs produced with an older metrics schema, verifier-check events may not be emitted explicitly; in that case we reconstruct verifier-call totals from per-item retry counters as described in Appendix E.3. Appendix E defines verifier calls as our primary compute unit and describes the aggregation procedure.

Oracle calls and token usage. Oracle-call totals are computed by counting model-result events (agent result events) and summing across run ids, again matching the aggregation used for verifier calls. For some Stage 1 runs produced with an older metrics schema, explicit repair-call events may not be emitted; in that case we reconstruct oracle-call totals from the same per-item retry counters used for verifier-call reconstruction (Appendix E.7). When per-call token counts are present in metrics, token totals can be computed by summing those fields. To make accounting robust across logging variants, we also provide a token-backfill procedure that parses stable token markers from per-call logs and emits structured per-task token totals; Appendix D.7 specifies this artifact and Appendix E explains how it is used for reporting.

C. Statement-audit protocol and adjudication

Statement quality is evaluated separately from Lean buildability and proof closure. The audit used for Table 2 compares generated Lean declarations against matched project-local source items that were used as statement-generation inputs. It does *not* use Lean build status, proof closure, or absence of explicit bypass constructs as evidence of source alignment. Thus the audit is a matched source-item-to-Lean statement-quality check, not a full PDF-to-Lean semantic-equivalence certificate.

We distinguish three layers: (i) project-local source items extracted or segmented from the source, (ii) matched Lean declarations aligned to those items, and (iii) proof targets used for Stage 2 closure. The statement-quality audit reported here focuses on layer (ii): matched declarations whose source item can be audited. The 875/875 proof-closure result is a separate layer-(iii) proof-repair statistic under matched statements. This separation is important: proof closure is a local consistency signal, but it is not used as a substitute for source-alignment review.

Audit checklist. Each matched Lean declaration is compared with the corresponding project-local source item. The audit checks (i) quantifier structure and binder order, including implicit "for all" in prose; (ii) side conditions and domain restrictions; (iii) the logical strength of the conclusion; (iv) whether additional Lean-side assumptions or typeclass constraints change the intended meaning; and (v) whether the modeling abstraction preserves the main mathematical content. Rows are adjudicated into *source-preserving*, *minor style issue*, or *modeling concern*. We group source-preserving and minor-style rows as *acceptable* in Table 2: these rows preserve the main mathematical content, but the latter may still be weaker than ideal in presentation, helper splitting, abstraction choice, or auditability. A *modeling concern* means that the matched Lean statement appears to change hypotheses, conclusion strength, object type, quantifier structure, domain restriction, or algorithmic/asymptotic meaning.

Two-expert adjudicated audit. The final numbers are aggregated from an anonymized per-item audit ledger described in the supplementary material. The workflow consists of Phase 1 matching, two separate human-expert audit passes (Auditor A and Auditor B), and final adjudication over their outputs. AI tools may be used as auxiliary navigation or triage aids, but the labels and final adjudication are human expert judgments. Auditor A and Auditor B worked independently: neither auditor saw the other's labels or notes during labeling, and the auditors did not communicate about individual items before both passes were complete. Final adjudication was performed only after the two independent passes were collected. We do not present this as a strict blind inter-annotator-agreement study, because the auditors may know the project context/source corpus and may use AI tools for navigation or triage. For that reason, we do not report Cohen's κ or weighted κ as a blind-annotation agreement claim. Instead, we report adjudicated labels together with the raw A/B disagreement counts and a taxonomy of the final modeling concerns. The adjudicated ledger records 201 A/B category disagreements, including 31 source-preserving-vs.-modeling and 33 minor-vs.-modeling disagreements. As a consistency check, the final ledger contains 2,866 rows, has no duplicate canonical keys, and every row introduced in the final merge has an adjudicated category. For every corpus, the audited count equals the sum of source-preserving, minor-style, and modeling-concern rows.

Records and exclusions. Table 2 and Table 13 report the matched auditable statements. Items without a reliable matched Lean declaration or auditable formalization unit are not included in these statement-quality rates. For the two largest corpora the excluded fraction is small: every project-local JSON item was passed through the audit, and the matched auditable set covers $523/526$ Tao Analysis II items and $840/875$ Rockafellar Convex Analysis items ($1{,}363/1{,}401 \approx 97.3\%$, jointly 65% of all 2,089 audited statements; see Table 12). This limits selection effects at the JSON-to-Lean matching step for the corpora that dominate the audit, although it does not by itself bound how many original-document statements entered the JSON layer. We therefore report the adjudicated labels over matched auditable statements only, and do not treat this table as a source-coverage denominator.

Structured source-item accounting. Table 12 gives a separate accounting view over the structured JSON items available for the released corpora. These counts make the preprocessing denominator more transparent, but they are not statement-quality rates: a JSON item may generate zero, one, or multiple Lean declarations, and a generated source-facing declaration may require additional helper infrastructure.

Source-to-JSON preprocessing audit. To inspect the previous link in the pipeline, we also audit source-to-JSON fidelity from repository-local source artifacts or recovered source excerpts to project-local JSON items. This is separate from the JSON-to-Lean statement-quality audit above. The audit uses no Lean build, proof-closure, or bypass-scan signal. The v2 audit first identified rows with recoverable repository-local source excerpts and sampled Tao, Lebl, and Local Rings rows from that auditable pool. A targeted v4 follow-up then covered previously thinner source chains: Rockafellar through a DjVu embedded text layer, Stacks through cached official webpages, and Nesterov through a human-verified PDF-derived TeX transcription. Table 8 reports both audits. These are local preprocessing-alignment checks, not all-corpus source-to-JSON rates and not full PDF-to-Lean coverage denominators. In particular, the publicly derived Stacks corpora extract especially cleanly: both Stacks slices score $50/50$ acceptable in the source-to-JSON sample, consistent with their origin as structured, machine-readable official webpages, so the document-to-JSON link is least uncertain for these corpora even though the sample still bounds extraction fidelity rather than extraction recall.

Conditional source-to-Lean chain estimate. Combining the preprocessing audit with the matched-statement audit gives a useful but conditional estimate for the audited source-to-Lean chain. Across v2 and v4, the source-to-JSON acceptable extraction rate is

$$\frac{98 + 188}{100 + 200} = \frac{286}{300} \approx 95.3\%.$$

Table 8. **Source-to-JSON preprocessing audit.** Acceptable extraction is *strong extraction* plus *minor extraction issue*. Denominators are auditable samples, not full-document coverage rates.

Audit/corpus	Source basis	Sample	Strong	Minor	Concern	Acceptable
v2: Tao/Lebl/Local Rings	repository-local source excerpts	100	86	12	2	98/100
v4: Rockafellar	DjVu embedded text layer	50	44	6	0	50/50
v4: Stacks Topology Ch. 5	cached official webpage	50	50	0	0	50/50
v4: Stacks Categories Ch. 4	cached official webpage	50	50	0	0	50/50
v4: Nesterov 2005	human-verified PDF-to-TeX text	50	3	35	12	38/50
v4 total	targeted source chains	200	147	41	12	188/200

Notes. The audit checks repository-local source artifacts or recovered source excerpts against project-local JSON items. Candidate pools: v2 has 2,866 candidate rows and 509 auditable rows; v4 has 67 Nesterov, 1,313 Rockafellar, 265 Stacks Categories, and 212 Stacks Topology candidate rows. Most v4 minor issues and concerns come from transcription mismatches or DjVu text-layer noise.

The JSON-to-Lean matched-statement audit reports

$$\frac{2{,}023}{2{,}089} \approx 96.8\%$$

acceptable statements. Multiplying these two audited links gives a conditional chain estimate of about

$$95.3\% \times 96.8\% \approx 92.3\%.$$

This number should be read only as an *audited conditional source-to-Lean fidelity estimate* over auditable project-local source items. It is not a full-document PDF-to-Lean recall or coverage rate, because the denominator does not enumerate every mathematical statement, exposition fragment, diagram-dependent item, or context-dependent sentence in the original sources.

Audit provenance and identifiers. For the JSON-to-Lean statement-quality audit, each row starts from the project-local source-item payload used during statement compilation. Each audited statement is identified by an anonymized corpus/slice/item identifier and a human-readable label recorded in the execution logs. The source excerpt used for audit is exactly the item payload shown to the model during statement compilation: the mathematical content together with any accompanying section context and dependency notes when present. The audited Lean declaration is linked back to the same item through the label that also appears in the docstring/comment immediately preceding the generated declaration in the Lean file. In our current datasets and logs, the excerpt is stored as a LaTeX-style string.

Inspectable anonymized ledger. The supplementary audit description makes the adjudicated audit inspectable at item level without putting the full ledger in the main paper body. Each row is anonymized and traceable to its corpus, project-local JSON item, matched Lean declaration, Auditor A/B labels, and final category. The ledger uses project-local source items as the source basis; it does not claim to audit against original PDFs, webpages, or source prose outside those structured items. Table 9 summarizes the schema groups. Rows outside generated coverage or without a reliable Lean match are treated as coverage/matching/auditability records, not as semantic failures of matched Lean statements; the statement-quality denominators in Tables 2 and 13 include only matched auditable statements.

Modeling-concern taxonomy. To avoid treating the 66 adjudicated modeling concerns as an opaque residual class, we derive a normalized issue taxonomy from the existing audit ledger. This table is a derived summary over rows whose final category is *modeling concern*; it does not re-audit the data, change the original labels, or estimate a full artifact failure rate. Counts sum to 66, and the derived consistency report passes row-count, per-corpus, issue-frequency, and non-modeling-exclusion checks. The two largest groups are semantic-modeling risks and algorithmic/complexity abstraction mismatches. Per-corpus, Rockafellar contributes 26 concerns, mostly algorithmic/complexity abstraction and semantic-modeling risk; Stacks Topology contributes 11 concerns, all semantic-modeling risks; Nesterov contributes 9 concerns, mostly algorithmic/complexity abstraction.

C.1. Statement-pipeline ablation details

Table 18 gives corpus-level results for the Stage 1 statement-pipeline ablation summarized in Table 5. This cross-domain suite uses prefix samples from five corpora: 15 items from Local Rings, 15 from Nesterov 2005, and 50 each from Stacks, Tao Analysis II, and Lebl Real Analysis I. Each corpus is run under five variants, for 900 item-variant tasks. Table 19 then

Table 9. **Anonymized statement-audit ledger schema.** The anonymized row schema records enough information to trace each adjudicated label back to a project-local source item and matched Lean declaration, while avoiding absolute paths, full source text, full Lean proofs, run settings, credentials, and local environment details.

Field group	Representative fields and purpose
Anonymous row identity	`audit_id`; stable anonymous row id assigned in final-ledger order.
Project-local source item	`corpus`, anonymized slice/item identifiers, source label, source context; identifies the structured item used as the audit source basis without exposing concrete local filenames.
Matched Lean unit	`lean_file`, `primary_declaration`, `helper_declarations`; records the matched Lean declaration and shortened helper list when relevant.
Matching evidence	`matching_level`, compact source evidence, compact Lean evidence; provides traceability without full source/proof dumps or concrete local paths.
Audit labels	`auditor_a_category`, `auditor_b_category`, `adjudicated_category`; records two separate passes and final adjudication.
Adjudication and issue type	`adjudication_reason`, `issue_type`, `suggested_correction`; explains modeling concerns or presentation/auditability issues.
Coverage/auditability status	`excluded_reason`, `disagreement_marker`; separates coverage or matching limits from semantic labels on matched Lean statements.

Table 10. **A/B audit disagreement summary.** `Excluded` is treated as a matching/auditability status rather than a semantic-quality label. For Source-preserving/Minor/Modeling disagreements, adjudication uses the more conservative semantic category unless the evidence shows only a presentation-level difference.

Disagreement type	Count
Same category	2,665
Source-preserving vs. minor style issue	137
Source-preserving vs. modeling concern	31
Minor style issue vs. modeling concern	33
Excluded vs. non-excluded	0
Other	0
Total category disagreements	201

Table 11. **Derived taxonomy of adjudicated modeling concerns.** Percentages use the 66 modeling-concern rows as the denominator. These categories summarize final adjudicated concerns; they are not proof failures, build failures, or source-to-JSON extraction errors.

Issue type	Count	Share
Semantic-modeling risk	26	39.4%
Algorithm/complexity abstraction	17	25.8%
Statement-structure mismatch	5	7.6%
Weakened conclusion	5	7.6%
Hypothesis strengthening	4	6.1%
Definition-only for claim	2	3.0%
Helper/placeholder statement	2	3.0%
Choice fallback weakens definition	1	1.5%
Multi-component definition incomplete	1	1.5%
One-to-many split	1	1.5%
Partial theorem component only	1	1.5%
Type-modeling shift	1	1.5%

Table 12. **Structured source-item accounting for released corpora.** JSON items are project-local preprocessing units, not full-PDF coverage denominators. Source-matched and generated-infrastructure declaration counts are computed from the released Lean workspace.

Source	Item groups	Items	With proof	With deps.	Sections	Source decls	Infra decls
Tao, *Analysis II*	8	526	389	112	8	566	1,794
Stacks Topology Ch. 5	1	212	160	112	28	165	530
Stacks Categories Ch. 4 (20 sec)	1	109	46	30	20	103	246
Local Rings	1	19	15	10	2	23	55
Nesterov 2005	1	67	35	54	5	92	261
Lebl, *Intro. to Real Analysis I*	9	464	42	19	43	477	724
Rockafellar, *Convex Analysis*	26	875	693	633	29	981	4,620
Total	47	2,272	1,380	970	135	2,407	8,230

Table 13. **Detailed two-expert statement-quality audit on matched project-local source items.** This is the full version of Table 2. "Source-preserving" means the Lean statement preserves the source item's main mathematical content; "minor" means semantically aligned but weaker than ideal in style, abstraction, helper splitting, or auditability; "modeling" means a possible change in mathematical meaning.

Corpus	Audited	Source-preserving	Minor	Modeling	Main notes
Tao Analysis II	523	241	274	8	Modeling mainly semantic/modeling risk and statement-structure mismatch.
Stacks Topology Ch. 5	142	40	91	11	Modeling mainly semantic/modeling risk.
Stacks Categories Ch. 4 (20 sec)	86	15	70	1	One statement-structure mismatch.
Local Rings	19	3	15	1	One incomplete multi-part proposition coverage issue.
Nesterov 2005	65	35	21	9	Modeling mainly algorithm/complexity abstraction and partial theorem components.
Lebl Real Analysis I	414	330	74	10	Modeling mainly weak/helper or placeholder statements and missing clauses in a few real-number items.
Rockafellar Convex Analysis	840	461	353	26	Modeling mainly algorithm/complexity abstraction and semantic/modeling risk.
Total	2,089	1,125	898	66	2,023/2,089 audited statements are acceptable; 66/2,089 raise modeling concerns.

Table 14. **Representative high-quality matched statements.** Positive audited examples whose main mathematical content remains visible in Lean; these supplement, but do not replace, the aggregate audit.

Corpus / item	Matched Lean declaration	Statement content visible in Lean	Audit signal
Tao, Theorem 6.10	`inverse_function_theorem_several_variables_6_10`	Inverse function theorem with open neighborhoods, local bijection, a witness inverse, differentiability of the inverse, and the inverse-derivative formula.	Source-preserving; exposed as one source-facing theorem, not only helpers.
Rockafellar, Theorem 14.1	`polarCone_polar_polar_eq_and_indicator_conjugate`	Polar-cone properties, the bipolar identity, and Fenchel conjugacy of cone indicators under a continuous perfect-pairing abstraction.	Source-preserving; compact interface for cone geometry and conjugacy.
Rockafellar, Theorem 12.2	`fenchelConjugate_closedConvex_proper_iff_and_biconjugate`	Closed convexity of the conjugate, properness equivalence, closure invariance, and the biconjugation identity.	Source-preserving; central conclusions remain visible in one statement.
Nesterov, Theorem 1.3.1	`optimal_scheme_rate`	An algorithm package generates the relevant sequences and the theorem states both the weighted estimate and the explicit accelerated convergence bound.	Source-preserving; formalizes an algorithm theorem, not an isolated inequality.
Stacks Categories, Lemma 4.19.2	`filteredColimitCommutesWithFiniteLimitInTypes`	Canonical isomorphism expressing that filtered colimits commute with finite limits for type-valued diagrams.	Minor style issue; presentation/auditability concern only.
Tao, Lemma 8.8	`lemma_8_8_borel_cantelli`	Borel–Cantelli for Lebesgue measure on finite-dimensional Euclidean space, using limsup of measurable sets and summability of volumes.	Source-preserving; concise measure-theory statement using canonical encodings.

Table 15. **Representative adjudicated modeling concerns.** Negative audit examples showing statement-layer errors that can survive Lean buildability and proof closure.

Corpus / item	Matched Lean declaration	Audit concern	Why it matters
Tao, Definition 1.20	`FormalLimit`	The JSON item describes a completion construction with formal Cauchy limits, an equivalence relation, a quotient/completion space, an induced metric, and an embedding; the matched primary declaration exposes only the formal-limit carrier.	A buildable carrier is not the full construction; companion declarations are needed.
Local Rings, Proposition 2.2	`exists_submodule_range_eq_of_stabilizes`	The matched declaration covers only one component of a multi-part proposition.	Useful theorem, but not full source-item coverage unless all clauses are represented.
Nesterov, Definition 1.3.1	`BaseFieldWithT_Q`	The minimizer choice includes a fallback branch not present in the JSON definition.	Fallback totalization can weaken the intended minimizer interface.
Lebl, Example 2.6.4	`alternating_harmonic_rearrangement_any`	The source asserts arbitrary-target rearrangement of the alternating harmonic series, but the matched Lean unit weakens this to a same-sum identity-permutation case.	A provable special case is not an acceptable source-theorem replacement.
Lebl, Example 5.1.4	`dirichletFunction`	The matched primary declaration defines the Dirichlet function, while the source example also asserts lower and upper integral values.	Definition-only matches are infrastructure, not formalizations of asserted values.
Rockafellar, Text 5.4.1.2	`infimalConvolution_reflection_indicator`	The matched Lean unit adds a non-bottom hypothesis to an item stated for arbitrary functions.	Extra hypotheses can turn a general statement into an easier restricted one.
Stacks Categories, Definition 4.17.1	`IsCofinalFunctor`	The source item contains existential structure that is not visibly represented in the matched Lean fragment.	Hidden existence data is hard to audit when witnesses are abstracted away.

Table 16. **Static anti-vacuity scan over theorem headers.** The scan counts theorem/lemma/example headers after comment stripping. It is not a satisfiability proof for all hypotheses, but it checks whether proof closure is accompanied by explicit contradiction-looking assumptions or heavy use of empty-type assumptions.

Source	Proof decls	Median hyps	Avg. hyps	Max hyps	Nonempty/Inhabited	IsEmpty/Subsingleton/Unique	Contradiction-looking
Tao, *Analysis II*	2,058	4.0	4.4	32	33 (1.6%)	10 (0.5%)	0 (0.0%)
Stacks Topology Ch. 5	541	4.0	4.3	21	41 (7.6%)	0 (0.0%)	0 (0.0%)
Stacks Categories Ch. 4	218	3.0	4.1	13	43 (19.7%)	13 (6.0%)	0 (0.0%)
Local Rings	72	4.0	4.1	8	6 (8.3%)	0 (0.0%)	0 (0.0%)
Nesterov 2005	303	7.0	7.8	33	32 (10.6%)	0 (0.0%)	0 (0.0%)
Lebl, *Intro. to Real Analysis I*	951	3.0	3.0	15	23 (2.4%)	1 (0.1%)	0 (0.0%)
Rockafellar, *Convex Analysis*	5,074	5.0	5.5	40	717 (14.1%)	35 (0.7%)	0 (0.0%)
Total	9,217	4.0	5.0	40	895 (9.7%)	59 (0.6%)	0 (0.0%)

Table 17. **LLM-assisted maintainability proxy audit.** This is not human evaluation and is not used as a final `mathlib`-readiness claim. Three separated Codex proxy-assessment passes scored accepted declarations for readability/reviewability, mathematical usefulness, likely upstreamability after polishing, and abstraction-level fit. Counts are numbers of positive scores among 20 declarations per pass; uncertain scores are not counted.

Proxy pass	#Decls	Readable	Useful	Upstreamable	Abstraction OK	Notes
Agent A	20	20	19	2	15	Strong readability/usefulness scores; upstreamability mostly uncertain because many declarations are source-facing wrappers or project-specific APIs.
Agent B	20	19	20	6	15	More optimistic on reusable topology, category-theory, and convex-analysis statements; one readability concern for a long Rockafellar theorem.
Agent C	20	19	20	0	18	Most conservative on upstreamability, often marking source-aligned wrappers as locally useful but not direct upstream candidates.

Interpretation. The proxy screen supports only a limited claim: sampled declarations often appear readable and mathematically useful, but upstreamability and abstraction level remain disputed and require human `mathlib`-style review.

reports a separate 74-item Convex Analysis stress suite from the first two Rockafellar sections. The metric is statement-stage Lean compilability of generated skeletons, with proof placeholders allowed; downstream proof repair is excluded. The semantic-check and quality-gate rows should not be interpreted as semantic-quality scores, since compilability does not measure source alignment or API-owner drift.

Table 18. **Detailed Stage 1 statement-pipeline ablation across five corpora.** Entries are statement-stage pass counts; proof repair is not run.

Corpus	Items	Full	Semantic fix	w/o semantic	w/o quality	w/o Agent B
Local Rings	15	15/15	14/15	15/15	15/15	6/15
Nesterov 2005	15	14/15	15/15	12/15	12/15	4/15
Stacks Project	50	48/50	47/50	45/50	48/50	34/50
Tao Analysis II	50	50/50	49/50	50/50	50/50	44/50
Lebl Real Analysis I	50	49/50	49/50	50/50	49/50	40/50
Total	180	176/180	174/180	172/180	174/180	128/180
Rate	–	97.8%	96.7%	95.6%	96.7%	71.1%

Table 19. **Harder Convex Analysis statement-ablation subset.** The subset contains 74 items from the first two Rockafellar sections. It is harder and more dependency-sensitive than the multi-corpus prefix sample.

Ablation	Pass	Rate	Failed
Full pipeline	61/74	82.4%	13
Semantic-check repair policy	63/74	85.1%	11
w/o semantic check	63/74	85.1%	11
w/o quality gate	53/74	71.6%	21
w/o Agent B repair	14/74	18.9%	60

Table 20. **Failure propagation in the Convex Analysis stress suite.** The hardest ablation failures are amplified by dependency blocking: once an upstream generated statement fails, dependent items are not attempted.

Ablation	Failed	Terminal status breakdown
Full pipeline	13	Agent A failed: 1; blocked: 8; failed after repair: 4
Semantic-check repair policy	11	blocked: 5; failed after repair: 4; quality rerun: 2
w/o semantic check	11	Agent A failed: 1; blocked: 6; failed after repair: 4
w/o quality gate	21	blocked: 17; failed after repair: 4
w/o Agent B repair	60	blocked: 50; failed after repair: 9; quality rerun: 1

Combined ablation interpretation. The two ablation suites differ in difficulty but agree on the main mechanism. On the multi-corpus prefix benchmark, removing Agent B drops statement pass rate by 26.7 percentage points, from 97.8% to 71.1%. On the harder Convex Analysis subset, the same removal drops pass rate by 63.5 percentage points, from 82.4% to 18.9%. The harder subset also shows why dependency-aware scheduling matters: a single failed upstream statement can turn into many `blocked_by_dependency` terminal statuses. In contrast, semantic-check and quality-gate variants have smaller and less monotone effects on compile pass rate. This is expected: their main purpose is to reduce compile-success-but-wrong outputs, such as API-owner drift, hidden hypotheses, non-canonical wrappers, and statements whose domain or direction shifts while remaining type-correct.

Manual comparison of ablation outputs. We additionally inspected matched outputs for representative items across variants. The inspection supports the same two-level interpretation as the main text. In the Convex Analysis subset, convex-set and halfspace items show the same pattern: variants that recall canonical `mathlib`/project APIs are easier to reuse downstream, while wrapper-heavy local encodings can compile but increase later proof burden. The normal-cone item illustrates a semantic risk invisible to compile success: a total object over arbitrary points can typecheck while changing the source convention, which defines the normal cone at points of the set. For Nesterov's dual-norm definition, semantic-feedback variants tend to recall canonical norm/operator-norm APIs, while variants without semantic feedback may produce compilable but more fragile maximum-style formulations with hidden hypotheses. For Nesterov's adjoint-operator definition, the repair-enabled variants produce named adjoint interfaces, while the no-Agent-B variant finds only partial API hints and fails to produce a checked declaration. For the Local Rings polynomial expansion lemma, most variants produce essentially the same theorem, showing that simple algebraic statements can be handled by Agent A alone; later no-Agent-B failures nevertheless block dependent propositions. For Lebl's boundedness/supremum definition and Stacks' von Neumann hierarchy lemma, semantic checking and quality gates mainly affect API ownership, naming, and avoidance of redundant

local wrappers rather than raw pass rate. Thus Table 5 should be read as a compilability ablation: Agent B is the main coverage component, while semantic checking and quality gates are statement-quality controls whose value is only partly visible in pass-rate statistics.

C.2. Mechanized compatibility checks against `mathlib`

For source-facing APIs that intentionally rephrase concepts already present in `mathlib`, we add Lean theorems that connect the generated interface back to the canonical library interface. These certificates are a mechanically checked statement-quality signal: they do not prove that every generated statement is source-preserving, but they do show that selected generated APIs have the intended mathematical content relative to established `mathlib` notions. Table 21 gives an aggregate source-level scan. It counts generated lemma/theorem declarations whose names and signatures have bridge shape (equality, iff, transport, equivalence, or explicit comparison section) and whose statement mentions canonical library vocabulary such as `Set`, `BddAbove`, `IsLUB`, `Tendsto`, `Summable`, `Metric.ball`, `IsOpen`, `IsCompact`, `ContinuousAt`, `LipschitzWith`, `HasDerivAt`, `IrreducibleCloseds`, or `Convex`. This scan is conservative about proof-body use of `mathlib` but syntactic: it is an audit-facing inventory of build-checked bridge certificates, not a proof that every bridge is the best possible abstraction. Table 22 then lists thirty representative examples from the generated Lebl real-analysis artifact. They span book-facing APIs for sets, functions, ordered sets, sequences, series, continuity, differentiability, and metric topology. The first block includes the explicit generated section `ComparisonWithMathlib`; later rows are compatibility theorems distributed across the book artifact.

Table 21. **Aggregate scan of bridge-style `mathlib`-compatibility certificates.** Rows count generated Lean lemma/theorem declarations whose statement has equality/iff/transport/equivalence shape and explicitly mentions canonical library vocabulary. "Mixed" means that both iff and equality occur in the statement.

Source root	Cert.	Iff	Eq.	Mixed	Files	Explicit comparison section
Tao, *Analysis II*	286	96	151	39	58	0
Rockafellar, *Convex Analysis*	753	161	480	112	201	0
Lebl, *Intro. to Real Analysis I*	158	100	46	12	31	5
Stacks Project	102	38	45	19	31	0
Nesterov 2005	9	0	9	0	4	0
Local Rings	1	0	1	0	1	0
Total	1,309	395	732	182	326	5

Table 22. **Representative mechanized `mathlib`-compatibility certificates.** Each row is a Lean theorem in the generated artifact proving that a source-facing generated API agrees with a corresponding `mathlib` concept. These are selected build-checked certificates, not an exhaustive count.

Area	Generated certificate	Generated API	`mathlib` target	Relation
Sets	`elementCollection_eq_set`	`ElementCollection`	`Set`	equality
Sets	`subsetElementCollection_iff`	book subset	`Set.Subset` / `<=`	iff
Sets	`properSubsetElementCollection_iff`	book proper subset	strict subset `<`	iff
Functions	`injectiveFunction_iff`	book injective function	`Function.Injective`	iff
Functions	`surjectiveFunction_iff`	book surjective function	`Function.Surjective`	iff
Functions	`bijectiveFunction_iff`	book bijective function	`Function.Bijective`	iff
Ordered sets	`boundedAbove_iff_bddAbove`	`OrderedSet.BoundedAbove`	`BddAbove`	iff, assuming `le = <=`
Ordered sets	`boundedBelow_iff_bddBelow`	`OrderedSet.BoundedBelow`	`BddBelow`	iff, assuming `le = <=`
Ordered sets	`leastUpperBound_iff_isLUB`	`OrderedSet.IsLeastUpperBound`	`IsLUB`	iff
Ordered sets	`greatestLowerBound_iff_isGLB`	`OrderedSet.IsGreatestLowerBound`	`IsGLB`	iff
Ordered sets	`leastUpperBoundProperty_iff`	book LUB property	`BddAbove` + `IsLUB`	iff
Sequences	`boundedSequence_iff_isBounded_range`	book bounded sequence	`Bornology.IsBounded (Set.range x)`	iff
Sequences	`boundedAboveSequence_iff_bddAbove_range`	book bounded-above sequence	`BddAbove (Set.range x)`	iff
Sequences	`boundedBelowSequence_iff_bddBelow_range`	book bounded-below sequence	`BddBelow (Set.range x)`	iff
Sequences	`convergesTo_iff_tendsto`	epsilon–`M` convergence	`Filter.Tendsto x atTop (nhds l)`	iff
Sequences	`monotoneIncreasingSequence_iff_monotone`	book increasing sequence	`Monotone`	iff
Series	`seriesConvergesTo_iff_hasSum_shift`	book series convergence	shifted `HasSum`	iff
Series	`seriesConvergent_iff_summable`	book convergent series	shifted `Summable`	iff
Series	`series_limit_eq_tsum`	book series value	`Summable` + `tsum`	iff
Continuity	`book_uniformContinuous_iff_uniformContinuous`	epsilon–delta uniform continuity	`UniformContinuous` on subtype	iff
Continuity	`book_lipschitzOn_iff_lipschitzWith`	book Lipschitz on a set	`exists K : NNReal, LipschitzWith K`	iff
Differentiability	`differentiableAtOn_iff_hasDerivAt`	difference-quotient differentiability	`HasDerivAt`	iff
Metric topology	`openBall_eq_metric_ball`	book open ball	`Metric.ball`	equality
Metric topology	`closedBall_eq_metric_closedBall`	book closed ball	`Metric.closedBall`	equality
Metric topology	`isOpenMetric_iff_isOpen`	metric open set	`IsOpen`	iff
Metric topology	`isClosedMetric_iff_isClosed`	metric closed set	`IsClosed`	iff
Metric topology	`closureByClosed_eq_closure`	closure as closed supersets	topological `closure`	equality
Metric topology	`metricInterior_eq_interior`	metric interior	topological `interior`	equality
Metric topology	`metricBoundary_eq_frontier`	metric boundary	topological `frontier`	equality
Metric sequences	`metricConvergesTo_iff_tendsto`	metric sequence convergence	`Filter.Tendsto x atTop (nhds p)`	iff

C.3. Stage 2 ablation slices

For Table 6, each long-form corpus uses a fixed ablation slice. All ablation methods for that corpus are run on the same slice. We report the mathematical scope of each slice rather than concrete local dataset filenames.

- **Real Analysis (Lebl).** Observed scope: Chapter 1, Sections 1, 2, 3, and 5.
- **Convex Analysis (Rockafellar).** Observed scope: Chapter 4, Sections 1–8.
- **Nesterov 2005.** Observed scope: Chapter 1, Sections 1–5.

Nesterov micro-ablation. We additionally ran a Nesterov-only micro-ablation restricted to Chapter 1, Section 5.

D. Semantic-First JSON Schemas for Intermediate Artifacts

This appendix specifies the machine-readable intermediate artifacts used for preprocessing inputs, resumability, debugging, and metrics reconstruction. Appendix B.2 describes the semantic role of these artifacts; here we focus on stable schema structure and invariants. We intentionally avoid committing to any on-disk directory layout or filename convention. Instead, we define each artifact by its semantic category (dataset record, checkpoint, history record, metrics event record, metrics summary, token backfill).

D.1. Dataset record: a natural-language item

A dataset record is the canonical *task input* describing one extracted mathematical unit (definition/lemma/theorem/etc.) in natural language (LaTeX-like). The statement pipeline consumes the statement text to generate a Lean declaration skeleton. The proof pipeline consumes the proof text (when present) to attempt proof completion for items that are meaningful proof targets (e.g., skipping definition-only records).

Container format and ordering. A dataset is stored as a JSON array; each element is a record. Consumers iterate deterministically by sorting records on `index`, even if the input is already sorted. This makes partial runs resumable and ensures that logs are comparable across machines.

Record schema (semantic view). Each record combines (i) stable identity, (ii) source position metadata, and (iii) the extracted content used for prompting. Table 23 summarizes the keys and their meaning.

Table 23. Dataset record keys (semantic schema).

Key	Meaning / used for
`index`	Stable integer identifier; defines iteration order and is used as the primary resume cursor.
`label`	Human-readable label (e.g., "Lemma 1.1"); used in logs and target lookup.
`env`	Coarse type tag (e.g., definition vs. proposition); used for filtering which items are proof targets.
`number_components`	Numbering decomposition used as a fallback for deterministic placement.
`extracted_labels`	Source-level labels attached to the item; used for traceability.
`context`	Structured section metadata (chapter/section titles and indices) used for deterministic file/position inference.
`content`	Extracted statement text; primary input to statement compilation prompts.
`dependencies`	Lightweight dependency labels used as optional local context.
`proof`	Extracted proof text; may be empty. Proof runs that require proof content skip empty-proof items.

Section metadata (`context`). The `context` object records where the item appears in the source. In our runs it includes chapter/section titles together with numeric indices (chapter/section/subsection). Values may be empty strings when the source lacks a corresponding heading. Consumers use this metadata for deterministic target positioning and, in some configurations, for mapping items to a file/namespace layout.

Invariants. The dataset file is a JSON array, `index` is unique, and consumers treat `index` order as the canonical iteration order. If `proof` is empty, proof-stage consumers may skip the record depending on configuration (e.g., when proof text is required as reference context).

Example record (redacted).

```
{ "index": 1,
  "label": "Definition 1.1.1",
  "env": "def",
  "number_components": [1, 1, 1],
  "extracted_labels": ["def:1.1", "eq:1.1"],
  "context": {
    "chapter_number": 1,
    "chapter": "___",
    "section_number": "1",
    "section": "___",
    "subsection_number": "",
    "subsection": ""
  },
  "content": "\\begin{definition} ... \\end{definition}",
  "dependencies": [],
  "proof": "" }
```

D.2. Lemma-hint JSON supervision artifact

In the lightly supervised FATE-H setting, we use optional *lemma-hint JSON* artifacts to provide minimal expert guidance for library navigation. The +31 condition uses a human-written JSON map with 31 entries and solves one additional instance (97/100); the finer +37 condition solves two additional instances relative to the fully automatic run (98/100). A lemma-hint JSON consists only of declaration-level hints (fully-qualified names and short natural-language role descriptions) and explicitly excludes any formal proof scripts, intermediate proof steps, or tactic traces. It conditions planning/retrieval, but it does not affect acceptance: every edit is still accepted or rejected solely by verifier feedback under $\mathcal{E}$. We therefore treat the lemma-hint setting as an infrastructure diagnostic rather than as a fully automatic proving result: it measures whether a human expert can expose the right reusable formal declarations and natural-language roles in the same JSON style used for textbook formalization, without handing the system a proof.

Schema (semantic view). A lemma-hint artifact can be represented as a JSON object keyed by problem id (or dataset index). Each entry stores a list of relevant Lean declarations and an optional short note explaining why they are relevant. The exact key names are not important; the semantics are.

```
{
  "problem_id": "FATEH_XX",
  "decl_hints": [
    "Mathlib.Analysis....",
    "Mathlib.Topology...."
  ],
  "notes": "optional natural-language rationale"
}
```

Consumption invariant. Downstream prompts are instructed to treat these hints as *navigation cues* (which existing theorems/definitions to look at), not as proof steps. In particular, the system may import, unfold, or apply the hinted declarations, but it must still produce a verifier-checked proof without relying on any unchecked intermediate claims.

D.3. Progress checkpoint: resumable cursor

A progress checkpoint stores the resume cursor for a pipeline. It is a single JSON object with exactly one integer key. Item-level pipelines store the next dataset index to process; file-level pipelines store the next file index to process. Checkpoints are overwritten in-place and are intended to support resuming after interruption.

Schema. Item-indexed pipelines use a key of the form `next_index`. File-indexed pipelines use a key of the form `next_file_index`. The value is always the next target to process.

D.4. History record: compact per-task trace

A history store is an append-only JSONL stream of compact records used for debugging and (optionally) prompt augmentation. Each line is a single JSON object. As described in Appendix B.2, the purpose of history is to provide lightweight

"memory" without reading full per-call logs; here we specify the stable top-level schema and the most common record kinds.

Top-level schema. Each history record includes a timestamp, a pipeline tag, a run id (for cross-reference into the metrics log), a Lean file identifier, a task identifier, a kind tag, an optional short summary, an optional log pointer, and a kind-specific payload object. Table 24 summarizes the keys and their meaning.

Table 24. History record fields (top-level schema).

Field	Meaning
`ts`	UTC timestamp (ISO8601).
`pipeline`	Pipeline tag (e.g., statement, proof, or cleanup; some legacy logs use `final` for cleanup mode).
`run_id`	Run identifier linking to the metrics event log.
`lean_file`	Lean file identifier (relative path or logical name).
`task_id`	Task identifier (item index, file-index+line target, compile/warnings task, etc.).
`kind`	Kind tag indicating payload interpretation (plan, fix, replan request, warning cleanup, etc.).
`summary`	Optional short summary (may be truncated).
`log_path`	Optional pointer to a per-call log for deep inspection.
`payload`	Kind-specific payload object (may truncate large strings).

Common kinds. In our runs, common kinds include planner outputs, executor replan requests, repair attempts, and (in Stage 2 cleanup mode) compilation-fix and warning-cleanup records. Planner records typically store both a structured plan object (when parsable) and a raw plan text block; fix records typically store the motivating Lean error output (as a string) together with basic metadata such as attempt number and token usage.

Truncation invariants. To keep history compact, the history writer may truncate long string fields (for example, summaries and large error logs). Truncation is applied to top-level strings (including selected payload strings) and is intended as a space-saving mechanism; full raw text remains available in per-call logs when needed.

Example history record (redacted).

```
{ "ts": "2026-01-19T09:55:23.567889+00:00",
  "pipeline": "proof",
  "run_id": "proof_stage2_..._e70e21ee",
  "lean_file": "<redacted Lean module>",
  "task_id": "34",
  "kind": "agent_c_plan",
  "summary": "status=ok | main=smoothedObjective_lipschitz_gradient",
  "log_path": ".../proof_agent_c_...log",
  "payload": {
    "round": 1,
    "code": 0,
    "tokens_used": 12345,
    "model": "___",
    "reasoning_effort": "___",
    "plan": { "...": "..." },
    "plan_raw": "{...}"
  } }
```

D.5. Metrics event log: append-only run instrumentation

The metrics event log is the primary machine-readable instrumentation used for paper metrics and compute accounting. It is an append-only JSONL stream where each line is a JSON object with four top-level fields: timestamp, run id, event label, and event payload.

Top-level schema. Every event record has the same top-level structure: `ts` (UTC timestamp), `run_id` (identifier), `event` (event label), and `data` (payload object). This invariant makes logs easy to parse and supports schema evolution by adding fields to `data` while preserving the top-level form.

Event taxonomy. The precise set of `event` labels is implementation-defined but stable within a logging schema version. For parsing and analysis, it is sufficient to group events into a small number of families: run boundaries, item/file boundaries, verifier checks, per-agent oracle-call results, and (for plan-driven runs) planning-loop bookkeeping events. Appendix B.2.6

explains how these families are used to derive paper metrics; the key schema invariant here is that every line shares the same top-level structure and the `data` object contains enough identifiers to attribute the event to a target.

Optional change-size snapshots. Some oracle-result events may include file snapshot summaries before and after applying an edit. These snapshots record whether a file exists and, if it does, its size and a line-count summary. They enable change-size analyses without parsing full diffs.

Example metrics record.

```
{ "ts": "2026-01-14T17:23:57.451963+00:00",
  "run_id": "proof_stage2_...",
  "event": "item_start",
  "data": { "index": 1, "label": "Lemma 1.1", "chapter": 1, "section": 1,
    "local_index": 1 } }
```

D.6. Metrics summary: end-of-run totals

For convenience, each run writes a metrics summary as a single JSON object. The summary mirrors the `run_end` event payload and provides run-level totals without scanning the full JSONL stream.

Common fields. All summaries include a pipeline identifier and a small set of run-level totals such as processed targets, resume cursor, total elapsed seconds, and aggregated attempt counters. The exact set of counters depends on the pipeline, but the intent is consistent: the summary provides a concise accounting view for that run segment.

Example summary (redacted).

```
{ "pipeline": "cleanup",
  "processed_files": 2,
  "next_file_index": 2,
  "total_seconds": 951.1927,
  "total_b_attempts": 1,
  "total_c_plans": 1,
  "total_a_attempts": 1,
  "total_sorries_eliminated": 1,
  "total_tokens_used": 123456 }
```

D.7. Token backfill: per-task token totals from per-call logs

Token usage can be reconstructed either from metrics events (when per-call token fields are present) or by parsing stable token markers from per-call logs. To make accounting robust to logging variants, we provide a token-backfill procedure that parses per-call logs, aggregates token totals per task, and emits these totals as a dedicated metrics run.

Backfill event schema. Backfill emits metrics records with event label `task_tokens`. Each record includes a stage tag, a task identifier, total tokens for that task, and a per-agent breakdown when multiple agents contribute to the same task. The record may also include optional convenience fields such as the human label and Lean file identifier when available.

Example backfill record (redacted).

```
{ "event": "task_tokens",
  "data": {
    "stage": "cleanup",
    "task": "0_L119",
    "tokens_used_total": 65831,
    "tokens_used_by_agent": { "a": 34170, "c": 31661 },
    "log_file_count": 2,
    "lean_file": "<redacted Lean module>" } }
```

E. Verifier-Normalized Compute and Cost Accounting

This appendix defines the compute and cost quantities reported in our experiments and explains how they are reconstructed from the structured artifacts described in Appendix D. Our primary compute unit is the number of Lean single-file compilation checks (*verifier calls*), because all stages repeatedly apply candidate edits and then re-check the affected Lean file via a uniform pinned environment. In addition, we report the frequency and token usage of model invocations ("oracle calls") and include a conservative robustness check that charges each oracle call as a fixed fraction of a verifier call.

E.1. Compute unit: verifier calls

We measure compute in terms of *verifier calls*. A verifier call is one invocation of Lean elaboration/typechecking for a single file under the pinned environment $\mathcal{E}$. Operationally, this corresponds to running a single-file Lean compile check and collecting the resulting diagnostics.

What is counted. We count *every* verifier invocation performed by the method, including verifier calls inside repair/search loops and those triggered by any optional stabilization procedures. Concretely, whenever a stage applies a candidate patch and then re-checks the file to decide accept/reject (or to advance to the next target), that re-check contributes one verifier call.

Success criterion for accounting. A file check succeeds iff Lean reports zero *error-level* diagnostics under $\mathcal{E}$. Warnings are ignored for build/check accounting unless they are explicitly promoted to errors by the toolchain or by a pipeline configuration.

Why this unit matches the pipeline design. All stages rely on repeated single-file compilation checks as the ground-truth accept/reject signal. In the statement stage, the system appends a skeleton and compiles the target file; if compilation fails, it applies a small repair and compiles again. In item-level proof repair and cleanup mode, the system alternates between proposing proof edits and compiling to confirm that the file still elaborates (and that the targeted `sorry` is removed when required). Because these checks are performed in a uniform pinned environment, verifier-call counts are comparable across corpora and stages.

E.2. Budgets and per-target caps

Our pipelines are budgeted by placing explicit caps on the number of propose→verify loops and on the number of fallback repairs, so that total compute is predictable and runs are resumable. This subsection restates the caps used in our experiments and fixes notation for the accounting formulas below.

Stage 1 (statement compilation). For each dataset item processed in Stage 1, the pipeline first generates a Lean statement skeleton (allowing proof placeholders) and then runs a verifier call on the affected file. If the check fails, a local repair routine is invoked and the file is checked again. We cap the number of local repair rounds per item by a small constant K (a fixed hyperparameter in the experiment configuration). As a result, each Stage 1 item triggers at least one verifier call and at most $1 + K$ verifier calls, with early termination when the file check succeeds.

Stage 2 (proof completion). Stage 2 is organized as a bounded planning-and-execution loop. A planning step proposes a structured proof strategy, and an execution step applies proof edits for the target hole; the file is then checked by Lean. If execution fails in a way that suggests a flawed plan, the system may request replanning; if the Lean check fails due to local compilation issues, the system may attempt bounded local repairs. To make accounting explicit, we use two caps: C is the maximum number of planning rounds (including the initial plan), and R is the maximum number of executor attempts per plan. This yields the conservative upper bound $B = R \cdot C$ on the number of executor-driven candidate proof patches for a single target. This is an upper limit; in practice, targets often terminate early once solved or once a stopping condition is reached.

Optional file splitting overhead. In some proof runs, a file may be split into multiple parts when it exceeds a line-count threshold, and additional verification checks may be triggered as part of that process. When splitting is enabled, we count the splitter's model calls as oracle calls (see §E.7) and we count any compilation checks performed during or after splitting as verifier calls.

E.3. Reporting total verifier calls

Per-run totals from metrics events. Verifier calls are reconstructed from the metrics event log described in Appendix D.5. Each verifier invocation emits exactly one verifier-check event (in our logs this event is labeled `lean_check`). For a single run r, let $\mathsf{Events}(r)$ be the multiset of metrics records for that run, and define

$$V_r \;=\; \#\{\, e \in \mathsf{Events}(r) \;:\; e.\texttt{event} = \texttt{lean_check} \,\}.$$

When a run is interrupted and resumed, each resumed segment produces its own run id; the corpus-level total is computed by summing V_r across all run ids included in the reported experiment.

Stage 1 metrics variant. Some Stage 1 runs in our archived artifacts were produced with a metrics schema that does not emit explicit `lean_check` events. For these runs, verifier-call totals can still be reconstructed because each processed item triggers one initial file verification, and each bounded repair attempt triggers one additional verification. Let $\mathcal{I}(r)$ be the set of Stage 1 items completed in run r (equivalently, the set of `item_end` events), and let $\texttt{b_attempts}(i)$ be the recorded number of bounded repair attempts for item i. We then compute

$$V_r \;=\; |\mathcal{I}(r)| \;+\; \sum_{i \in \mathcal{I}(r)} \texttt{b_attempts}(i).$$

When `lean_check` events are present (as in Stage 2 runs), we use the direct event-count definition above.

Aggregating by corpus and stage. For each instrumented corpus/stage pair reported in the main paper, we aggregate (i) the number of targets processed, (ii) the number solved, and (iii) the total verifier calls. For Stage 1, "targets" refers to the number of dataset items processed in the aggregated runs; "solved" may either be the number of items whose target file check succeeded at item end, or simply equal to targets if the pipeline configuration is set to return a checked output per item. For Stage 2, "targets" refers to the number of proof targets attempted under the reported setting, and "solved" refers to the number of targets whose designated proof hole was successfully closed under budget.

Summary table. Table 25 reports the aggregated target counts, solve counts, and total verifier calls for the instrumented audit subset and for FATE-H.

Table 25. Total verifier calls and verifier-normalized cost for the instrumented audit subset and FATE-H. These counts are reconstructed from complete run logs and should be read separately from the released-workspace inventory in Table 1; compute cells are dashed where the lemma-hint JSON runs were not included in the cost logs.

Setting	Stage	Targets	Solved	Total verifier calls	Calls/solved	Notes
Lebl Real Analysis I	1	416	416	592	1.42	SCC = 100% `sorry` allowed
	2	339	339	628	1.85	Matched statements closure = 100%
Convex Analysis	1	560	560	605	1.08	SCC = 100% `sorry` allowed
	2	499	499	1065	2.13	Matched statements closure = 100%
Nesterov 2005	1	67	67	81	1.21	SCC = 100% `sorry` allowed
	2	37	37	77	2.08	Matched statements closure = 100%
Benchmark: FATE-H (auto)	2	100	96	283	2.95	One instance is incorrect see §5.3
Benchmark: FATE-H (+31 hint JSON)	2	100	97	–	–	31 decl hints see Table 6
Benchmark: FATE-H (+37 finer JSON)	2	100	98	–	–	37 decl hints see Table 6

E.4. Paper metrics: definitions and log reconstruction

This subsection defines the primary paper metrics and explains how they are reconstructed from the structured logs in Appendix D. Throughout, we treat the Lean toolchain under the pinned environment $\mathcal{E}$ as the only acceptance signal: a

proposed change "counts" only if it is committed by the pipeline and the subsequent verifier check reports zero error-level diagnostics.

PB (project buildability). PB is true iff the final generated project builds under $\mathcal{E}$ (allowing `sorry` in Stage 1). Operationally, PB is assessed on the end-to-end project state produced by the pipeline, not on an individual item or file. In our instrumentation, PB is recorded as a run-level outcome and can also be reproduced by running a project build in the pinned environment.

SCC (statement compile completion). Let $\mathcal{B}$ be the set of Stage 1 atomic blocks processed in a corpus run. A block is counted as compiled iff its enclosing Lean file elaborates under $\mathcal{E}$ after the block insertion/repair loop terminates. We report $\mathrm{SCC} = |\mathcal{B}_{ok}|/|\mathcal{B}|$. In logs, $\mathcal{B}$ is reconstructed from the processed item inventory; success is reconstructed from item-end status markers and the final verifier outcome for the affected file.

ARR (average repair rounds per block). For each block $b \in \mathcal{B}$, let $a(b)$ be the number of candidate repair patches attempted after the initial insertion (blocks that elaborate immediately contribute $a(b) = 0$). We report $\mathrm{ARR} = \frac{1}{|\mathcal{B}_{ok}|} \sum_{b \in \mathcal{B}_{ok}} a(b)$, i.e., the mean repair-attempt count over successfully processed blocks. In logs, $a(b)$ can be reconstructed either by counting repair-result events attributable to that block or by counting the verifier checks that occur inside the bounded verify→repair loop for the block (minus the initial post-insertion check).

Proof-closure rate under matched statements. Let $\mathcal{H}$ be the evaluated set of proof obligations under matched statements (the statement layer is frozen and not edited during proof repair). A hole counts as closed iff (i) the designated `sorry` is eliminated in the target file and (ii) the verifier still reports zero error-level diagnostics for that file under $\mathcal{E}$. We define the proof-closure rate as $|\mathcal{H}_{closed}|/|\mathcal{H}|$. In logs, $\mathcal{H}$ is reconstructed from the evaluated task inventory together with per-target identifiers, and closure is reconstructed from the first verifier check that both verifies and decreases the hole count for that target.

Auxiliary reporting on FATE-H. One FATE-H instance is a benchmark issue whose Lean statement is incorrect under $\mathcal{E}$ and thus not provable as stated. For completeness, excluding this instance yields $96/99 \approx 97.0\%$ solved automatically and $98/99 \approx 99.0\%$ solved with lemma-hint supervision; the main table reports the stricter denominator of all 100 instances.

E.5. External FATE-H Reference Numbers

Table 26 records external FATE-H numbers only as auxiliary context. They are not model-matched or compute-matched with M2F, and we do not use them as evidence for a prover-leaderboard claim. The main-text FATE-H table instead compares same-model controls and M2F variants, because the role of FATE-H in this paper is to diagnose whether residual failures under fixed formal statements are proof-search failures or missing-infrastructure failures.

Table 26. External FATE-H reference numbers. These are auxiliary context only, not model-matched or compute-matched comparisons.

System	Protocol	Solved
Seed-Prover 1.5 (Chen et al., 2025)	strongest reported variant	80/100
Kimina / Goedel prover	prover	$<$ 10/100

E.6. FATE-H per-problem length and solved status

To contextualize aggregate success rates on FATE-H, Figure 3 reports, for each of the 100 problems, a simple code-length proxy together with the outcome category under our Stage 2 runs. We define the length of a problem as the number of *non-empty lines of Lean code* in its source file, measured after preprocessing and proof blanking but before any successful proof patch is applied. The 100 problem files contain 2,612 non-empty lines in total. Because FATE-H contains many problems, we plot a bar chart: each bar corresponds to one problem (x-axis), and bar length is the line count (y-axis).

Outcome categories. Bars are colored by outcome: *solved fully automatically* (green), *solved with light expert help via lemma-hint JSON* (yellow), and *unsolved* (red). One additional category marks the single *wrong-statement* instance ("Error"), discussed in §5.3; this instance is not solvable under $\mathcal{E}$ as stated and is visualized with a distinct styling (e.g., a separate color or hatch pattern). In our reported runs, this yields 96 solved automatically, 1 additional instance solved by the +31 JSON, 1 further instance solved by the finer +37 JSON, 1 remaining open, and 1 wrong instance (Table 6).

Table 27 expands the same outcome categories as a missing-infrastructure diagnosis. The two JSON-assisted successes

are not counted as automatic proof successes because the human expert supplied reusable declaration handles and natural-language role descriptions. They do not contain proof scripts, tactic sequences, or derivations. This supports the interpretation used in the main text: at this level, the residual bottleneck is often exposing the right formal infrastructure rather than executing the final proof once that infrastructure is available.

Table 27. FATE-H automatic-failure diagnosis. Lemma-hint JSON entries are infrastructure hints: declaration names plus natural-language roles, not proof scripts.

Outcome category	Count	Diagnosis
Solved fully automatically	96	existing infrastructure sufficient
Solved with +31 hint JSON	1	missing declaration navigation
Solved with +37 finer hint JSON	1	finer reusable lemma map needed
Still open	1	missing specialized infrastructure
Wrong statement	1	benchmark statement issue under $\mathcal{E}$

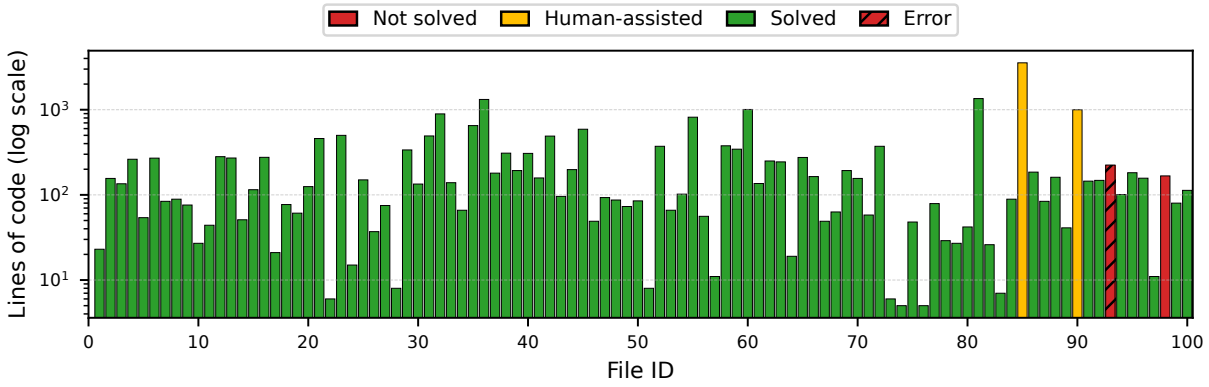


Figure 3. FATE-H per-problem code length (non-empty lines; 2,612 total) and outcome category. The legend is drawn inside the plot; the single wrong-statement instance is shown with a distinct style (see §5.3).

E.7. Non-verifier oracle calls and their frequency

In addition to verifier calls, the system makes non-verifier *oracle calls* to the model through Codex CLI. These calls produce candidate Lean edits (statement skeletons, proof patches, repairs) or structured plans that guide subsequent edits. Oracle calls are not counted as verifier calls, but they can dominate wall-clock time and token usage, so we report their frequency and use them in robustness checks.

What counts as an oracle call. An oracle call is any model invocation whose output can directly change the Lean project or the plan used to change it. In Stage 1, oracle calls include the initial statement-synthesis call and any bounded repair calls used to make the file compile. In Stage 2, oracle calls include planning calls, executor calls that apply proof edits, repair calls that address compilation failures after edits, and (when enabled) splitter calls that refactor large files into multiple parts. In Stage 2 cleanup mode (when used), oracle calls include compilation-fix and warning-cleanup calls as well as plan/execution calls that eliminate remaining `sorry` tokens.

How oracle calls are counted in logs. Oracle calls are reconstructed from the metrics event log (Appendix D.5). Each model invocation emits one oracle-result event whose label identifies the agent role (planner/executor/repair/splitter). We count oracle calls by counting these oracle-result events within each run. If a reported corpus/stage aggregates multiple run ids (e.g., due to resumption), we sum oracle-call counts across those run ids, matching the aggregation used for verifier calls.

Stage 1 metrics variant. Some Stage 1 runs in our archived artifacts were produced with a metrics schema that does not emit explicit repair-call events for the bounded repair agent. However, the Stage 1 control flow makes oracle-call totals reconstructible from the same per-item repair counter: each processed item always triggers one statement-synthesis model call, and each bounded repair attempt corresponds to one additional repair model call. Let $\mathcal{I}(r)$ and `b_attempts` be as

above; we compute

$$Q_r \;=\; |\mathcal{I}(r)| \;+\; \sum_{i \in \mathcal{I}(r)} \texttt{b_attempts}(i),$$

which matches the verifier-call reconstruction for Stage 1.

Note on Lean LSP/MCP tool queries. Some runs issue lightweight Lean IDE queries during editing (goal inspection, hover/type information, local search, completions, and small snippet execution). These are not model invocations and are not emitted as metrics events, so they are excluded from the oracle-call totals above and from the token-based accounting (see Appendix B.2.4).

Summary table. Table 28 reports oracle-call frequency alongside verifier calls, and normalizes by the number of targets to make stages and settings comparable.

Table 28. Oracle-call frequency for the instrumented audit subset and FATE-H. Dashes indicate that the lemma-hint JSON runs were not included in the oracle-call accounting logs.

Setting	Stage	Verifier calls	Oracle calls	Calls/target	Notes
Lebl Real Analysis I	1	592	592	1.42	Statement synthesis + bounded repairs
	2	628	1263	3.73	Planner/executor/repair calls
Convex Analysis	1	605	605	1.08	Statement synthesis + bounded repairs
	2	1065	2001	4.01	Planner/executor/repair (+ optional split)
Nesterov 2005	1	81	81	1.21	Statement synthesis + bounded repairs
	2	77	148	4.00	Planner/executor/repair calls
Benchmark: FATE-H (auto)	2	283	339	3.39	Proof-only; statements already elaborate
Benchmark: FATE-H (+31 hint JSON)	2	–	–	–	Proof-only; conditioned on 31 decl hints
Benchmark: FATE-H (+37 finer JSON)	2	–	–	–	Proof-only; conditioned on 37 decl hints

E.8. Token-based cost for oracle calls

Oracle-call frequency measures how many model invocations are made, but it does not capture their size. We therefore additionally report token usage as a proxy for model-side cost.

Token totals. When per-call token counts are recorded directly in metrics events, total tokens can be computed by summing those per-call fields over oracle-result events in the run. To make accounting robust to logging variants, we treat the token-backfill artifact described in Appendix D.7 as the canonical source of token totals when available, because it reconstructs per-task token usage directly from stable token markers in the per-call agent logs.

Summary table. Table 29 reports total token usage for oracle calls as a complementary proxy for model-side cost, together with a per-solved normalization for the Stage 2 settings in the instrumented subset and FATE-H.

E.9. Fractional accounting robustness check

Verifier calls are our primary compute unit, but model invocations can also contribute nontrivial cost. To check that our conclusions are not sensitive to how oracle calls are priced, we report a conservative *fractional accounting* view that charges each oracle call as a fixed fraction of a verifier call.

Definition. For a run (or an aggregation of runs), let V be the total number of verifier calls and let Q be the total number of oracle calls (as defined above). For a chosen fraction $\alpha \in [0, 1]$, we define a verifier-equivalent compute total

$$\mathrm{Cost}_\alpha \;=\; V \;+\; \alpha \cdot Q.$$

We report Cost_α (and optionally Cost_α/solved) for a small set of α values that cover "oracle is cheap" to "oracle is comparable" regimes.

Reporting. Unless otherwise stated, we report the two values used in Table 30, $\alpha \in \{0.10,\ 0.25\}$. The corresponding totals are computed from the same reconstructed V and Q used in Tables 25 and 28.

Table 29. Token usage for oracle calls in the instrumented audit subset and FATE-H. Dashes indicate that the lemma-hint JSON runs were not included in the token-accounting logs.

Setting	Stage	Oracle calls	Total tokens	Tokens/solved
Lebl Real Analysis I	1	592	20,429,061	49,108
	2	1263	140,040,564	413,099
Convex Analysis	1	605	28,421,469	50,753
	2	2001	307,966,089	617,167
Nesterov 2005	1	81	2,875,173	42,913
	2	148	16,362,141	442,220
Benchmark: FATE-H (auto)	2	339	44,177,658	460,184
Benchmark: FATE-H (+31 hint JSON)	2	–	–	–
Benchmark: FATE-H (+37 finer JSON)	2	–	–	–

Summary table. Table 30 reports verifier-equivalent compute totals under fractional accounting for representative choices of α, using the same V and Q aggregates reported above.

Table 30. Fractional accounting on the instrumented audit subset and FATE-H: verifier-equivalent compute $\mathrm{Cost}_\alpha = V + \alpha Q$. Dashes indicate that the lemma-hint JSON runs were not included in the compute-accounting logs.

Setting	V	Q	$\mathrm{Cost}_{0.10}$	$\mathrm{Cost}_{0.25}$
Lebl Real Analysis I (Stage 2)	628	1263	754.30	943.75
Convex Analysis (Stage 2)	1065	2001	1265.10	1565.25
Nesterov 2005 (Stage 2)	77	148	91.80	114.00
FATE-H (auto, Stage 2)	283	339	316.90	367.75
FATE-H (+31 hint JSON, Stage 2)	–	–	–	–
FATE-H (+37 finer JSON, Stage 2)	–	–	–	–

F. Failure Taxonomy and Audit Implications

Table 31 summarizes the main failure modes we observe or expect in project-scale autoformalization. We include this taxonomy because successful Lean compilation is not the same as high-quality mathematical infrastructure. Some failures are caught directly by Lean, but the most important statement-layer failures require provenance audit, route checks, compatibility theorems, or human review of abstraction quality.

Example: overgrown generated infrastructure in later Stacks formalization. A representative failure mode appears in later Stacks-style experiments beyond the main Chapter 5 topology and Chapter 4 category-theory subsets. The Stacks Project places *Stacks* in Chapter 8 of the preliminary material; Section 8.12, "Functoriality for stacks," studies how stacks change when the base site changes.[5] Before Lemma 8.12.12, the section introduces several layers of structure: the base-change category $u^p\mathcal{S}$, the pushforward $f_*\mathcal{S}$, auxiliary categories such as $u_{pp}\mathcal{S}$, localization-type constructions, and stackification. Lemma 8.12.12 then states, under the hypotheses of Sites, Lemma 7.21.8, that the canonical functor $\mathcal{S} \to f_* f^{-1}\mathcal{S}$ is an equivalence of stacks and that f^{-1} induces an equivalence on morphism categories. Its proof reduces the morphism-category statement to the stack equivalence, then checks fully faithfulness and local essential image using earlier stack criteria and the explicit construction of the canonical functor.

This structure is exactly where project-scale autoformalization becomes difficult. The source proof can refer informally to "the canonical functor", "the corresponding morphism of sites", "stackification", and "the construction f^{-1}". In Lean, these names must be materialized as definitions, functors, morphisms of fibred categories, instances for pullbacks/equalizers, preservation-of-limit assumptions, and comparison maps between the generated stack-level and fibred-category-level APIs. If the surrounding library has not already stabilized these abstractions, the model can still produce buildable code, but the code may become locally convenient in a way that is hard for a human reviewer to inspect. The following presentation-normalized

[5] Official tags: Section 8.12 is https://stacks.math.columbia.edu/tag/04WA; Lemma 8.12.12 is https://stacks.math.columbia.edu/tag/04WS.

Table 31. **Failure taxonomy for project-scale autoformalization.** The taxonomy separates verifier-detectable failures from statement-quality failures that can survive compilation. The last column records how the current paper mitigates or exposes each risk.

Failure mode	Typical symptom	Detection signal	Current mitigation / remaining limitation
Compile-level inconsistency	Unresolved names, import cycles, type errors, or broken namespaces prevent elaboration.	Lean error diagnostics under the pinned environment.	Handled by VeriRefine accept/revert and project-level build checks. This is the easiest class to detect mechanically.
Weakened or shifted statement	The Lean statement is provable but no longer matches the source conclusion, quantifiers, or hypotheses.	Provenance-linked expert audit; source-to-Lean route checks for selected results.	Audited for retained targets, but not mechanically guaranteed for the full corpus. The anonymized ledger schema and aggregate audit results are inspectable; release of full source excerpts remains constrained by anonymization and copyright.
Over-strong or vacuous assumptions	A theorem is true because assumptions are stronger than the source, inconsistent, or too narrow.	Audit of hypotheses/domain constraints; nonempty-context checks when available; downstream reuse patterns.	Absence of explicit bypass constructs does not by itself rule this out. We therefore treat anti-vacuity checks as an important next evaluation layer.
Ad-hoc local API drift	The system introduces a local definition even though a canonical `mathlib` notion exists, causing semantic drift or unnecessary glue lemmas.	`mathlib`-compatibility theorems, expert inspection, and downstream glue-lemma burden.	For selected generated APIs, we prove equality or iff links to `mathlib`; global optimality of abstraction level is not claimed.
Suboptimal abstraction level	The statement preserves the source meaning locally, but is specialized below the level expected of a reusable library API.	Expert review of mathematical generality; downstream reuse and glue-lemma burden; comparison with mature `mathlib` interfaces.	This is visible in parts of *Convex Analysis*: many statements over $\mathbb{R}^n$ should eventually be generalized to normed or Hilbert spaces, and topological notions such as closure should often be phrased relative to an ambient set/subspace. We treat these as acceptable local formalizations when semantically aligned with the source, but not as `mathlib`-ready abstractions.
Overgrown generated infrastructure	Definitions are buildable but too verbose or too dependent on many generated declarations for humans to audit easily.	Large local dependency surface, many generated prerequisites, long typeclass/local-instance preambles, and poor human-review readability.	The artifact can still be useful as machine-checked infrastructure, but human maintainability becomes the bottleneck. This is especially visible in higher-categorical or stack-theoretic material.
Missing library navigation	The statement may be source-appropriate, but proof repair fails because the system cannot expose the right existing declarations or intermediate lemmas.	FATE-H hint-JSON runs and unresolved proof failures after repeated replanning.	Lemma-hint JSON provides declaration names and natural-language roles, not proof scripts. This diagnoses missing infrastructure rather than pure proof-search failure.
Diagram-dependent or context-fragment source	The source statement depends on a diagram, earlier informal convention, or local context that is not standalone.	Preprocessing and provenance audit.	Excluded from the retained target layer unless enough context can be formalized. This is a scope limitation for the retained proof targets.

excerpt illustrates this risk. It is a comparison morphism used in such a route, not a formalization of the full Stacks lemma. The issue is not a failed build; it is a maintainability and auditability warning.

```
/-- The canonical comparison morphism `S ⟶ Sinv.pullback u`
associated to the chosen inverse-image stackification
`iSinv : u ₚ S ⟶ Sinv`. -/
noncomputable def pushforwardInverseImageComparison
    [Functor.IsContinuous u J K]
    (S : StackOver J)
    (Sinv : StackOver K)
    (iSinv : u ₚ S ⟶ Sinv) :
    S ⟶ Sinv.pullback u :=
  let SinvPull : StackOver J := Sinv.pullback u
  let _ : HasPullbacks C := inferInstance
  let _ : HasEqualizers C := inferInstance
  let _ : PreservesLimitsOfShape WalkingCospan u := inferInstance
  let _ : PreservesLimitsOfShape WalkingParallelPair u := inferInstance
  let F :=
    pushforwardPullbackFibredMorphismFunctor
      u (S : FibredCategoryOver C) (Sinv : FibredCategoryOver D)
  show S ⟶ SinvPull from
    ofFibredCategoryMor (F.obj iSinv)
```

Table 32 unpacks why this example is more serious than ordinary verbosity. The generated declaration is short enough to compile and use, but each local line hides a large mathematical interface decision. For example, the local instances for pullbacks, equalizers, and preservation of limits are not decoration: they determine which categorical infrastructure the comparison morphism can use. Similarly, the coercions between `StackOver` and `FibredCategoryOver` decide whether the formal route really follows the stack-theoretic construction or merely proves a locally convenient surrogate statement.

Table 32. **Why the Stacks example is hard to audit.** The declaration builds, but each Lean signal below corresponds to a nontrivial mathematical-interface choice.

Audit point	Concrete Lean signal	What must be checked
Change of base site	`pullback`, inverse-image stackification, and comparison morphisms.	The generated maps should model the intended f^{-1} and f_* constructions, not a weaker source-local surrogate.
Canonical comparison	`pushforwardInverse ImageComparison`.	The source word "canonical" hides a construction route; the Lean morphism must follow that route, not merely have the right type.
Fibred-category layer	Coercions between `StackOver` and `FibredCategoryOver`; `ofFibredCategoryMor`.	The bridge from stack-level APIs to fibred-category APIs must preserve the intended mathematical object.
Limit assumptions	`HasPullbacks`, `HasEqualizers`, `PreservesLimitsOfShape`.	These hypotheses may be necessary, but they can also be over-strong if inserted only to make the construction typecheck.
Generated helper API	`pushforwardPullback FibredMorphismFunctor`.	The helper may be useful, but a nonstandard abstraction can force later files to accumulate glue lemmas.

This example motivates a distinction used throughout the paper. Buildability and proof closure are necessary artifact checks, but they are not sufficient for library quality. A declaration intended to grow from project-local infrastructure into a reusable mathematical library must also expose the right abstraction boundary, reuse canonical APIs when possible, and keep the local dependency surface reviewable. For this reason, our evaluation combines build status with overlap analysis, downstream reuse, source-order coherence, `mathlib`-compatibility certificates, and source-to-Lean route checks; none of these individually proves full semantic equivalence or source alignment, but together they expose more failure modes than proof closure alone.

G. Qualitative Showcase: Navigable Artifacts with Provenance and Dependency Governance

This appendix provides a qualitative view of what the end-to-end system produces *as a library artifact*, complementing the quantitative results in the main text. While Table 1 already reports the primary scale statistics (files/decls/LoC and end-to-end buildability), we focus here on three properties that are hard to convey with a single success rate: (i) a closed-loop workflow with explicit governance points, (ii) ToC-aligned *navigability* from a source section to a small, checkable edit scope in Lean, and (iii) provenance- and dependency-centric indicators that support auditing and maintenance.

G.1. Workflow overview: a closed loop, not one-shot generation

The system treats project-scale formalization as a *closed loop* driven by the Lean toolchain: edits are accepted only when the verifier reports a strict improvement in a build- and hole-based objective. This design gives two practical properties that matter at scale. First, failures are localized (file/section/part granularity), enabling fast revert-and-retry without corrupting the rest of the project. Second, outputs are not merely a pile of proofs, but a structured repository with explicit provenance anchors and a controlled import footprint, so that readers can inspect the artifact and future users can decide which parts are worth maintaining. Figure 4 provides a compact, governance-oriented view of this loop, complementing the end-to-end pipeline diagram in Figure 1.

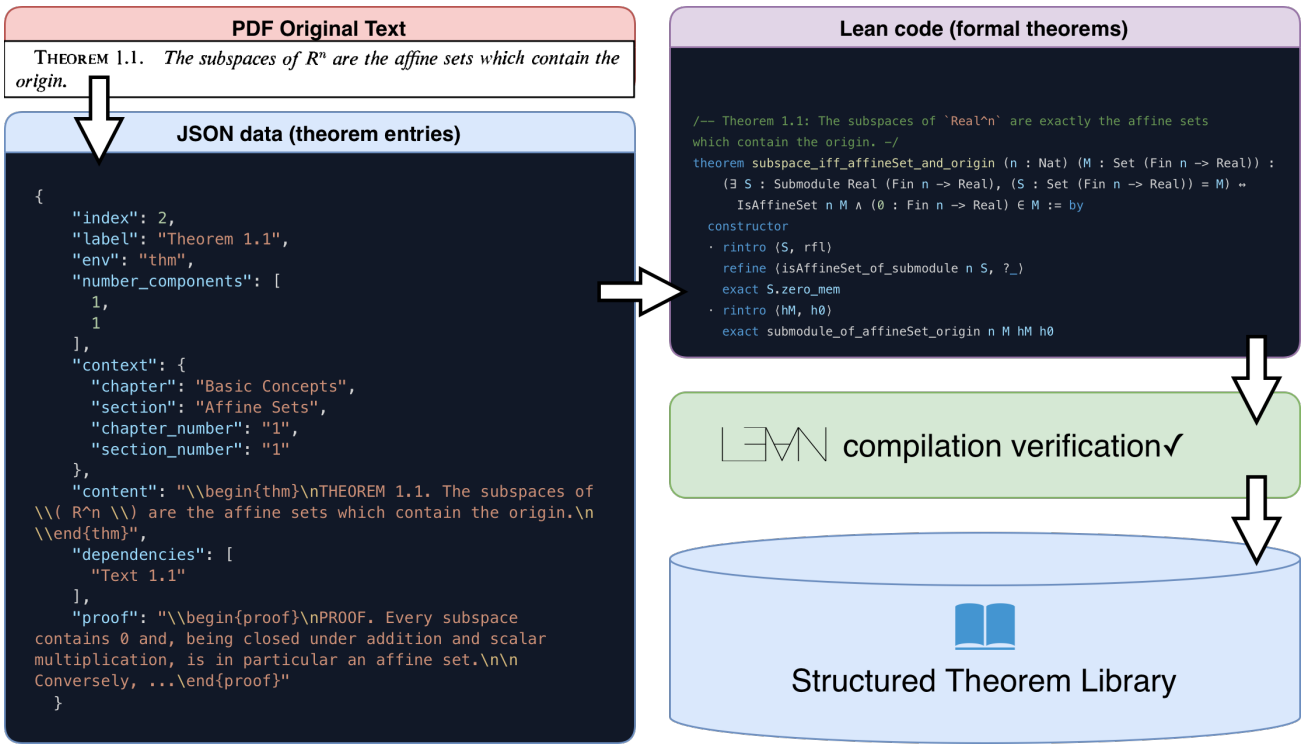


Figure 4. **Workflow overview.** A verifier-in-the-loop pipeline that turns PDF-derived structure into a buildable Lean project with (i) provenance anchoring for statement auditability, (ii) human gatekeeping at governance points, and (iii) accept/revert refinement driven solely by Lean diagnostics, yielding a queryable repository of Lean-checked declarations rather than isolated one-off proofs.

G.2. Source-to-project alignment at section granularity

A central "project-scale" requirement is *navigability*: readers should be able to move from a source section to the corresponding Lean files with minimal cognitive overhead. For the partial Rockafellar *Convex Analysis* artifact, we mirror the source section numbering (§1–§15) at the module level and display the generated Lean file tree with indentation (not full paths). When a section is too large to edit and verify reliably, the workflow performs a structure-preserving split into `_partK` modules, while keeping a stable section-level entry module as the public import target.

G.3. Workflow capability indicators beyond raw scale

The main text already reports corpus scale and buildability (Table 1). Here we focus on *governance* indicators that are more directly tied to whether the workflow remains controllable as projects grow: how aggressively the system can split a single section without breaking the public module interface, how large the import surface becomes, and whether the artifact carries enough source-facing documentation to support auditing. The counts below are computed by the same source-level statistics pass used for Table 1; they are intended as artifact-governance indicators, not as semantic-equivalence certificates.

Highlights. In the *Convex Analysis* artifact, the workflow executes a **23-way** structure-preserving split of a single section, emits **5,528** docstrings, and manages **335** unique imports within that source root. Across the released workspace, the artifact contains **10,155** docstrings and **588** distinct imported modules, while retaining zero explicit bypass constructs after comment

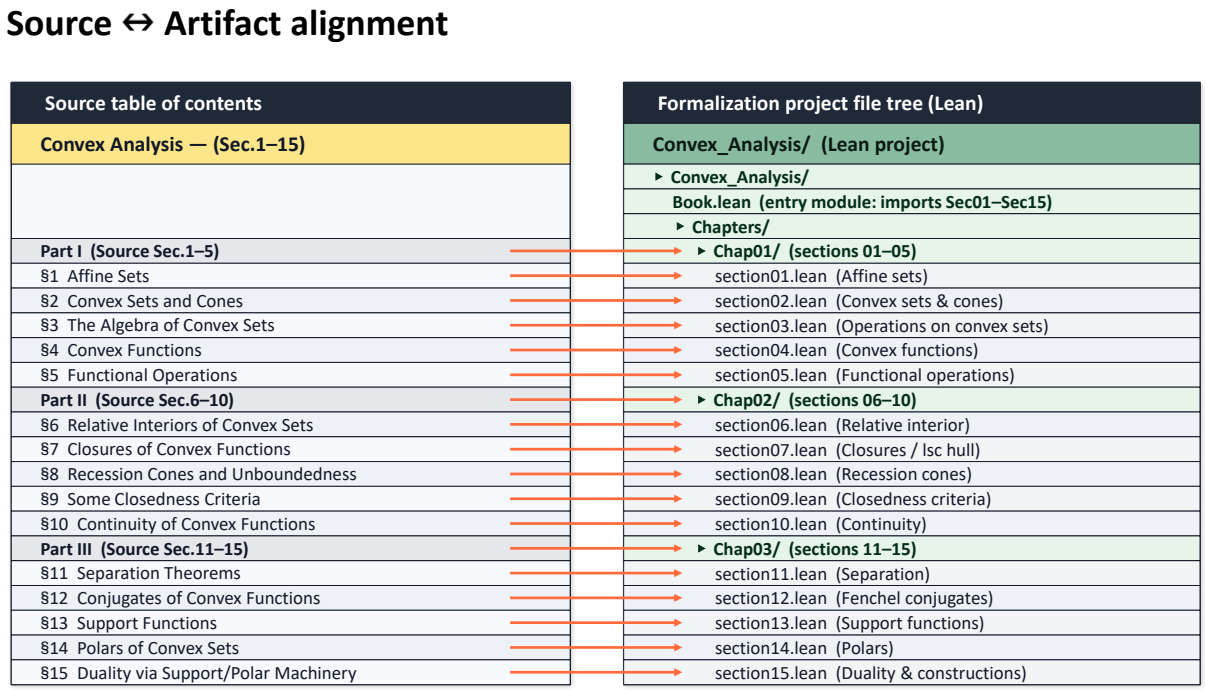


Figure 5. **Navigability and locality.** The partial Rockafellar *Convex Analysis* artifact is organized into a ToC-aligned Lean project under the source section numbering (§1–§15). The right panel is an indented file tree (not full paths) illustrating structure-preserving splitting (e.g., a single source section expanded into many `_partK` modules) while keeping a stable section module for downstream imports and verifier-local edits.

Table 33. **Artifact governance indicators from the released workspace.** Unique imports count distinct uncommented imported modules under each source root; max split is the largest number of sibling `_partK` files sharing a section base name; docstrings count Lean `/- ... -/` docstring openers. Bypass counts explicit bypass constructs after comment stripping: `sorry`/`admit` tokens, source-level `axiom`/`constant`/`opaque`/`unsafe`/`partial` declarations, and heartbeat-disabling options. The same scan separately checks for meta/macro command surface and imports of modules named `Unsafe`; both are zero in the released workspace. Together these indicators summarize edit locality, dependency surface, and audit-facing documentation beyond raw scale.

Domain	Source	Files	Decls	Unique imports	Max split	Docstrings	Bypass
Analysis	Tao, *Analysis II*	100	2,360	101	4	2,352	0
Topology	Stacks Topology Ch. 5	37	695	30	3	652	0
Category theory	Stacks Categories Ch. 4 (20 sec)	23	349	7	3	336	0
Algebra	Maassarani, *On Some Local Rings*	9	78	10	4	71	0
Optimization	Nesterov 2005	26	353	26	10	350	0
Analysis	Lebl, *Intro. to Real Analysis I*	57	1,201	50	6	866	0
Convex analysis	Rockafellar, *Convex Analysis* (partial)	332	5,601	335	23	5,528	0
Released workspace	All sources + import aggregators	591	10,637	588	23	10,155	0

stripping: no `sorry`/`admit` tokens, no source-level `axiom`/`constant`/`opaque`/`unsafe`/`partial` declarations, and no heartbeat-disabling options. The same pass also finds zero meta/macro command lines, zero imports of modules named `Unsafe`, and zero simple explicit inconsistency patterns such as `False` binders or `False ->` assumptions. These indicators reflect the system's ability to keep edit scopes small and dependencies inspectable while still producing end-to-end buildable projects.

Table 34. **Representative reused generated APIs.** We select math-facing declarations from the static reuse analysis, excluding helper-only names. These examples show the kinds of reusable interfaces that later generated code builds on; downstream references are source-level references within the same artifact.

Source	Declaration	Kind	Refs	Mathematical role
Rockafellar	`fenchelConjugate`	def	3,676	Core convex-analysis transform used throughout conjugacy and duality developments.
Rockafellar	`effectiveDomain`	def	2,089	Domain API for extended-real convex functions and properness statements.
Rockafellar	`epigraph`	def	1,749	Epigraph-based modeling interface for closedness, convexity, and separation results.
Nesterov 2005	`DualPairing`	def	514	Optimization geometry interface connecting primal and dual spaces.
Lebl Real Analysis I	`IntervalPartition`	structure	269	Darboux-integration infrastructure reused by sums, refinements, and integrals.
Stacks Topology Ch. 5	`IsConstructibleIn`	abbrev	145	Constructible-set API reused across local closedness and stratification lemmas.
Tao, *Analysis II*	`OpenBox.volume`	def	1,180	Measure-theoretic box volume interface used by outer-measure constructions.

G.4. Detailed source-infrastructure, reuse-cost, and source-order statistics

Table 35 reports a conservative source-level split between directly source-matched declarations and generated infrastructure. Tables 36 and 37 give the exact counts behind Figure 2. They are source-level artifact statistics. The source-infrastructure split evaluates how much Lean code is needed to turn source-facing statements into a reusable local library. The overlap table evaluates whether generated declarations collapse to direct reuse from existing library facts. The source-order table evaluates whether generated internal references follow the source exposition order. None of these tables is a semantic-equivalence certificate; they are objective checks on the generated repository as reusable source-local infrastructure.

Table 35. **Directly source-matched declarations versus generated infrastructure.** A declaration is counted as directly source-matched only when its nearest docstring has an explicit source marker such as a numbered theorem, lemma, definition, text item, or equation. All other declarations are counted as generated infrastructure, including explicit helper lemmas, local APIs, compatibility facts, wrappers, and proof-route support declarations. The explicit-helper column is a conservative named/docstring subset of generated infrastructure, not the full infrastructure count.

Domain	Source	Decls	Direct source-matched	Generated infrastructure	Explicit helpers
Analysis	Tao, *Analysis II*	2,360	566 (24.0%)	1,794 (76.0%)	1,545
Topology	Stacks Topology Ch. 5	695	165 (23.7%)	530 (76.3%)	382
Category theory	Stacks Categories Ch. 4 (20 sec)	349	103 (29.5%)	246 (70.5%)	182
Algebra	Maassarani, *On Some Local Rings*	78	23 (29.5%)	55 (70.5%)	1
Optimization	Nesterov 2005	353	92 (26.1%)	261 (73.9%)	13
Analysis	Lebl, *Intro. to Real Analysis I*	1,201	477 (39.7%)	724 (60.3%)	26
Convex analysis	Rockafellar, *Convex Analysis* (partial)	5,601	981 (17.5%)	4,620 (82.5%)	2,024
Total	released artifacts	10,637	2,407 (22.6%)	8,230 (77.4%)	4,173

Table 36. Overlap analysis on five cross-domain artifact subsets. Decls are source-level declaration counts from Table 1; E/A/N classify checked declarations as direct reuse, nontrivial construction over related infrastructure, or new source-aligned construction; non-direct-reuse is A+N.

Domain	Source	Decls	Checked	E(%)	A(%)	N(%)	Non-direct(%)
Analysis	Tao, *Analysis II*	2,360	300	4.0	93.3	2.7	96.0
Topology	Stacks Topology Ch. 5	695	278	10.8	41.4	47.8	89.2
Category theory	Stacks Categories Ch. 4 (20 sec)	349	45	31.1	55.6	13.3	68.9
Algebra	Maassarani, *On Some Local Rings*	78	71	0.0	95.8	4.2	100.0
Optimization	Nesterov 2005	353	249	0.4	99.6	0.0	99.6
Weighted subtotal	five domains	3,835	943	≈6.0	≈78.1	≈15.9	≈94.0

G.5. Dependency and import structure

Table 38 gives a static import-graph view of the released artifacts. Each uncommented Lean import contributes one edge. When the imported module resolves to another generated file inside the same source root, we count it as a generated-to-

Table 37. Objective source-order coherence analysis. We align generated Lean declarations with chapter/section order from the source JSON and count static references between generated declarations. "Order-compatible" means the referenced declaration appears no later than the referencing declaration in generated source order. Same/earlier/future section columns classify the referenced declaration relative to the referencing declaration's source section. This is a source-level proxy, not an elaborator dependency graph or a semantic-equivalence proof.

Source	Internal refs	Order-compatible	Same section	Earlier section	Future section
Tao, *Analysis II*	9,001	97.3%	85.4%	11.9%	2.7%
Rockafellar, *Convex Analysis*	52,694	89.1%	47.5%	49.3%	3.2%
Stacks Topology Ch. 5	2,757	100.0%	88.7%	11.3%	0.0%
Stacks Categories Ch. 4	678	100.0%	96.3%	3.7%	0.0%
Maassarani, *On Some Local Rings*	216	100.0%	100.0%	0.0%	0.0%
Nesterov 2005	2,421	99.5%	48.5%	51.4%	0.1%
Lebl, *Intro. to Real Analysis I*	5,894	99.3%	62.7%	36.6%	0.7%

generated import; otherwise it is counted as a library import. The import graph is acyclic in all seven source roots. The total file count here is 584 because the table restricts attention to the seven source roots and excludes the seven top-level workspace import aggregators counted in Table 1.

Table 38. **Dependency and import structure of the released artifacts.** Generated-to-generated imports are resolved imports whose target file lies under the same source root. Library imports are unresolved inside the released repository, including `Mathlib`. Max import depth is the longest resolved within-source import chain. This is a static source-level import summary, not a semantic dependency graph.

Domain	Source	Files	Import edges	Gen.-to-gen.	Library	Max depth	Cycles
Analysis	Tao, *Analysis II*	100	210	128	82	7	0
Topology	Stacks Topology Ch. 5	37	75	43	32	14	0
Category theory	Stacks Categories Ch. 4 (20 sec)	23	30	10	20	3	0
Algebra	Maassarani, *On Some Local Rings*	9	19	12	7	5	0
Optimization	Nesterov 2005	26	71	50	21	25	0
Analysis	Lebl, *Intro. to Real Analysis I*	57	160	113	47	11	0
Convex analysis	Rockafellar, *Convex Analysis* (partial)	332	1,248	940	308	127	0
Total	Seven source roots	584	1,813	1,296	517	127	0

G.6. Source-to-Lean proof-route certificates

Table 39 gives a small set of source-to-Lean proof-route certificates for recognizable results. Each certificate records (i) the source proof skeleton, (ii) the named generated declarations/interfaces used by the compiled Lean proof route, and (iii) the local statement-quality property being checked. The certificate is build-checked because the referenced Lean route occurs inside the released proof-closed artifact. It is not a mechanized semantic-equivalence proof for the source sentence. Its role is narrower: it checks that the generated statement layer exposes the same intermediate mathematical infrastructure that the source proof uses, rather than proving a weakened final statement through an opaque shortcut.

G.7. Representative code excerpts

We include three presentation-normalized excerpts from Lean-checked files that illustrate complementary sources of complexity in project-scale formalization. First, we show a textbook-level interface definition from *Convex Analysis* that is imported across multiple chapters, illustrating how the artifact exposes reusable notions behind a small, stable API surface (here, the definition of a supporting hyperplane). Second, we include a classical result from *Real Analysis* (the mean value theorem) as a representative example of a long proof whose elaboration must remain stable under localized edits and refactoring; for space, we elide internal steps with ellipses. Third, we present a paper-style theorem with heavy parameterization and layered hypotheses (Algorithm 3.11 / Theorem 1.4.1), which stresses scoping, abbreviation management, and long-range dependencies across the library. For PDF rendering, the listings are lightly normalized: we use TeX escapes to render Lean symbols and omit nonessential proof bodies with ellipses. They are evidence of the shape of the generated declarations, not copy-paste buildable snippets; the underlying released files are the buildable artifacts.

```
/- Convex Analysis (textbook): multi-file dependency surface + interface lemma. -/
import Mathlib
import FormalBook.Chapters.Chap01.section01_part1
import FormalBook.Chapters.Chap02.section09_part3
import FormalBook.Chapters.Chap03.section11_part4
```

Table 39. **Representative source-to-Lean proof-route certificates.** Each row records a source proof skeleton, the named generated interfaces used by the compiled Lean route, and the statement-quality property checked by that route. Names are presentation-normalized; the released files contain the buildable versions. This is a build-checked route certificate, not a mechanized semantic-equivalence certificate for the full informal statement.

Source result	Source proof skeleton	Build-checked Lean route	Statement-quality signal
Tao, Cauchy–Schwarz inequality	The textbook proof decomposes the target into upper/lower quadratic bounds and then packages the two inequalities as an absolute-value bound.	The theorem `cauchySchwarz_fin_sum_real` routes through upper/lower helper lemmas, square/sign facts, and an absolute-value closing lemma.	The formal proof preserves the inequality-decomposition route rather than proving the final statement as an opaque one-off.
Stacks Topology, closed graph lemma	The source identifies the graph as an equalizer/preimage of the diagonal and uses that the diagonal is closed in a Hausdorff space.	The theorem `isClosed_graph_of_continuous` introduces a graph map, proves continuity of the two coordinate projections, and closes the goal through `isClosed_eq`.	The generated statement exposes the diagonal/equalizer API expected by a topology library.
Stacks Topology, image of a section is closed	The proof rewrites the image of a section as the fixed-point set $\{x \mid s(f(x)) = x\}$ and again uses Hausdorff equalizer closedness.	The theorem `isClosed_range_of_continuous_section` first proves the range/fixed-point equivalence, then applies continuity facts and `isClosed_eq`.	The formal route reuses the same equalizer infrastructure, showing local library accumulation across adjacent lemmas.
Rockafellar, biconjugate/closure facts	The source develops conjugacy through closed convex closures and proper closed-convex hypotheses before applying the biconjugate theorem.	Biconjugate lemmas route through `fenchelConjugate`, `convexFunctionClosure`, `ProperConvexFunctionOn`, and `ClosedConvexFunction`.	Core convex-analysis notions become named interfaces reused across later formalizations.

```
/-- Text 11.3.2: A supporting hyperplane to `C` is a hyperplane which is the boundary
    of a
supporting half-space to `C`. Equivalently, a supporting hyperplane has the form
`H = {x | x dot_v b = β}` with `b ≠ 0`, such that `x dot_v b ≤ β`
for every `x ∈ C` and `x dot_v b = β` for at least one `x ∈ C`. -/
def IsSupportingHyperplane (n : Nat) (C H : Set (Fin n -> Real)) : Prop :=
  ∃ (b : Fin n -> Real) (β : Real),
    b ≠ 0 ∧
      H = {x : Fin n -> Real | x dot_v b = β} ∧
        (∀ x, x ∈ C → x dot_v b ≤ β) ∧
          ∃ x, x ∈ C ∧ x dot_v b = β
```

```
/-- Theorem 4.2.4 (Mean value theorem): If `f : [a, b] → Real` is continuous on
the closed interval and differentiable on the open interval, then some
`c ∈ (a, b)` satisfies `f b - f a = deriv f c * (b - a)`. -/
theorem mean_value_theorem
    {f : Real → Real} {a b : Real} (h1 : a < b)
    (hcont : ContinuousOn f (Set.Icc a b))
    (hdiff : DifferentiableOn Real f (Set.Ioo a b)) :
    ∃ c ∈ Set.Ioo a b, f b - f a = deriv f c * (b - a) := by
  ...
```

```
/-- Theorem 1.4.1.
Assume the structural model (2.2) and that `fhat` is differentiable with M-Lipschitz
    gradient on
`Q1`. Let `d1` be a prox-function of `Q1` with parameter `sigma1 > 0` and prox-diameter `
    D1` as in
(4.3_D1). Let `d2` be a prox-function of `Q2` with strong convexity parameter `sigma2 >
    0` and define
`D2 := max_{u ∈ Q2} d2 u` as in Proposition 2.7. Apply Algorithm 3.11 to the smoothed
    problem
(4.1) with `mu = mu(N) = (2normAnorm_{1,2}/(N+1)) * sqrt(D1/(sigma1 sigma2 D2))`
    (equation (thm3_muN)). After `N`
iterations define `\hat x := y_N` and
`\hat u := sum_{i=0}^N 2(i+1)/((N+1)(N+2)) u_mu(x_i)` (4.2). Then `0 ≤ f(\hat x) -
    phi(\hat u)` and the
```

```
duality-gap bound (4.3) holds, and consequently the epsilon-solution bound (4.4)
    holds. -/
theorem algorithm311_duality_gap_bound {E1 E2 : Type*} [NormedAddCommGroup E1]
    [NormedSpace Real E1] [FiniteDimensional Real E1] [NormedAddCommGroup E2]
    [NormedSpace Real E2]
    [FiniteDimensional Real E2] (Q1 : Set E1) (Q2 : Set E2)
    (A : E1 →L[Real] (E2 →L[Real] Real)) (fhat : E1 → Real) (phihat : E2 → Real)
    (d1 : E1 → Real) (d2 : E2 → Real) (sigma1 sigma2 M D1 : Real) (xSeq ySeq : Nat →
    Q1) (umu : E1 → E2)
    (N : Nat) (hatu : Q2) : let A' : E1 →ℓ[Real] Module.Dual Real E2 :=
      { toFun := fun x => (A x).toLinearMap
        map_add' := by
          intro x y;ext u;simp
        map_smul' := by
          intro c x;ext u;simp }
    let D2 : Real := sSup (d2 '' Q2)
    let mu : Real := (2 * OperatorNormDef A' / ((N : Real) + 1)) *
        Real.sqrt (D1 / (sigma1 * sigma2 * D2))
    let fbar : E1 → Real := SmoothedObjective Q2 A phihat d2 mu fhat
    let f : E1 → Real := fun x => fhat x + sSup ((fun u => A x u - phihat u) '' Q2)
    let phi : Q2 → Real := AdjointFormPotential Q1 Q2 A fhat phihat
    LipschitzOnWith (Real.toNNReal M) (fun x => fderiv Real fhat x) Q1 →
    0 < sigma1 → 0 < sigma2 → StrongConvexOn Q1 sigma1 d1 → StrongConvexOn Q2
    sigma2 d2 →
    IsProxDiameterBound Q1 d1 D1 → OptimalSchemeAlgorithm Q1 fbar d1
      (M + (1 / (mu * sigma2)) * (OperatorNormDef A') ^ 2) sigma1 xSeq ySeq →
    (∀ x, IsSmoothedMaximizer Q2 A phihat d2 mu x (umu x)) →
    (hatu : E2) = Finset.sum (Finset.range (N + 1)) (fun i =>
        (2 * ((i : Real) + 1) / (((N : Real) + 1) * ((N : Real) + 2))) smul
          umu (xSeq i : E1)) →
    ConvexOn Real Q1 fhat → DifferentiableOn Real fhat Q1 →
    ConvexOn Real Q2 phihat →
    (∀ x,
      fderiv Real (SmoothedMaxFunction Q2 A phihat d2 mu) x =
        (AdjointOperator A' (umu x)).toContinuousLinearMap) →
    0 ≤ M → 0 ≤ D1 →
    0 < M + (1 / (mu * sigma2)) * (OperatorNormDef A') ^ 2 →
    IsClosed Q1 → IsOpen Q1 → ConvexOn Real Q1 fbar → DifferentiableOn Real fbar Q1
    →
    0 ≤ mu → (∀ u ∈ Q2, 0 ≤ d2 u) →
    (∀ x, BddAbove ((fun u => A x u - phihat u) '' Q2)) →
    BddAbove (d2 '' Q2) → Q2.Nonempty →
    (∀ u, BddBelow ((fun x => A x u + fhat x) '' Q1)) →
    0 ≤ f (ySeq N : E1) - phi hatu ∧
      f (ySeq N : E1) - phi hatu ≤
        (4 * OperatorNormDef A' / ((N : Real) + 1)) * Real.sqrt (D1 * D2 / (sigma1 *
    sigma2)) +
          (4 * M * D1) / (sigma1 * ((N : Real) + 1) ^ (2 : Nat)) ∧
      ∀ epsilon > 0, (N : Real) ≥
          (4 * OperatorNormDef A' / epsilon) * Real.sqrt (D1 * D2 / (sigma1 * sigma2)) +
            2 * Real.sqrt (M * D1 / (sigma1 * epsilon)) → f (ySeq N : E1) - phi hatu ≤
    epsilon := by
    ...
```

H. Limitations

Compilation and proof closure check internal Lean consistency, not semantic equivalence to the source. Our statement-quality audit checks matched Lean statements against project-local source items rather than proving semantic equivalence to the original PDFs or all source prose. The audit uses two human expert passes with final adjudication; because AI assistance may be used for navigation or triage, we report adjudicated labels rather than treating the result as a formal inter-annotator-agreement study. Whole-project closure shows local compatibility and reuse within the generated book/project, not global statement quality: semantic modeling, definition/typeclass reuse, abstraction level, domain style, polymorphism, and downstream glue-lemma burden remain expert judgments. M2F is not model-agnostic: we use a strong closed model and do not report a broad model sweep. FATE-H is a fixed-statement Stage 2 stress test and missing-infrastructure diagnosis rather than a compute-matched prover benchmark, and the lemma-hint JSON results are infrastructure-assisted rather than fully

automatic. We also do not report a head-to-head textbook-scale comparison with other agentic autoformalization pipelines under the same source slices and budgets. The monotone accept/revert rule prevents regression but can miss solutions requiring temporary error increases. Generated code is buildable but not necessarily `mathlib`-ready, dependency recovery is heuristic, and diagram-heavy sources remain out of scope. The artifact release provides buildable generated artifacts for inspection at `github.com/imathwy/M2F_repo`. Table 40 summarizes the supplementary archive contents and reader-facing use. The archive contains four inspectable components: the generated Lean workspace, the document-to-item parser, the M2F orchestration code with prompt templates and agent configurations, and release notes with build-status metadata. It does not include original source PDFs/LaTeX files, closed-model credentials, full raw interaction logs, repository metadata, or local build caches; exact raw-source-to-Lean reruns therefore still require those external inputs and services. Appendix C and Appendix G give further details on the audit protocol and artifact structure.

Table 40. **Supplementary artifact release checklist.** The archive separates generated Lean artifacts from the parser and orchestration code used to produce them. It supports inspection of reported outputs and system components, but is not a self-contained rerun package for original-source-to-Lean generation.

Component	Included material	Reader use
Generated Lean workspace	`M2F_results`: generated Lean projects, pinned Lean/`mathlib` metadata, and project modules.	Inspect buildable artifacts, module layout, proof closure, source-local APIs, and representative statements.
PDF/LaTeX-to-JSON parser	`mathdoc-parser-main`: parser code for converting mathematical source documents into structured JSON items.	Inspect the document-to-item interface used before statement compilation.
M2F orchestration code	`codex_agent_bookformalization`: orchestration source, prompt templates, agent configuration, and repair workflow code.	Inspect the system-level execution protocol, prompt templates, and verifier-in-the-loop control logic.
Pinned environment metadata	Lean toolchain and dependency metadata included with the generated workspace.	Check the verifier environment used for the reported build status.
Release notes	Release README and build report in the supplementary folder.	Check included components, build-status notes, and the stated exclusion of source PDFs/LaTeX files, credentials, full raw interaction logs, repository metadata, local build products, caches, and author-identifying headers.

I. Broader Impacts

Project-scale autoformalization can make mathematical knowledge more searchable, checkable, and reusable. If reliable, systems like M2F could help build infrastructure for formalizing frontier papers, checking long dependency chains, teaching formal methods, and creating machine-readable interfaces for mathematical discovery. The main positive impact is therefore not replacing mathematicians, but reducing the cost of turning existing literature into auditable formal libraries that humans and machines can build on.

The same capability also carries risks. Generated formal artifacts may be over-trusted if buildability is confused with semantic faithfulness, and poor abstractions can create large libraries that compile but are hard to maintain or review. Because the current workflow uses strong closed models and copyrighted or licensed mathematical sources may be involved, deployment should respect source-document rights, disclose model/tool dependence, and keep provenance information attached to generated declarations. We view human expert review, provenance-aware audit summaries, buildable artifacts, and clear limitation statements as necessary safeguards for responsible use.