The Future Is the Test: Library Abstractions for Mathematics Not Yet Written

Anonymous authors
Paper under double-blind review

Abstract

The value of a mathematical abstraction lies in the mathematics it supports next. On major Mathlib refactors, adopted designs complete 60/60 use attempts against 29/60 for their predecessors, with no losses. We formulate future-use generalization (FUG) as evaluating abstractions on withheld mathematical uses in Lean. FUSE turns failures and recurring proof work on observed uses into revisions of shared statements. FutureUseBench connects historical designs, authored use graphs, and real-library replay to test these components separately. Across eight authored development targets, exposing use descriptions increases unseen declaration completion $2.28\times$. With accepted interfaces fixed, feedback on observed uses removes 81% of remaining declaration failures. Among jointly completed, source-screened uses, exposure reduces downstream source tokens by 16–27%, a proxy for adaptation burden. Proof dependence changes method rankings, and shared revisions can both enable reuse and lose interface acceptance. Reliable transfer to unseen proofs after revision remains open. These findings establish later mathematics as an executable test and a concrete design signal for abstraction.

1 Introduction

Mathematical abstractions determine what can be built next. Across five major Mathlib refactors, adopted designs complete all 60 downstream use attempts, while their predecessors complete only 29, with no use lost after refactoring. Yet on eleven design pairs, shortening source is negatively correlated with gains in later use ($\rho = -0.59$; Section 5.1). We study abstraction quality as *generalization across uses*: observed work guides design; withheld requirements test it (Figure 1). Held-out examples test predictions; held-out uses test abstractions.

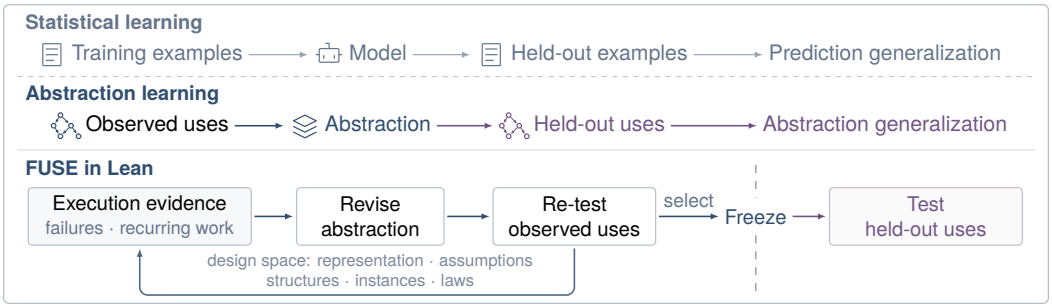


Figure 1: Observed uses guide revision; held-out uses test generalization. FUSE turns failures and recurring proof work into proposed abstraction changes, retests them, and selects a design. Held-out uses provide no feedback.

Formal mathematics makes design consequences executable. Autoformalization evaluates correspondence to a target (Wu et al., 2022; Azerbayev et al., 2023; Liu et al., 2025a; Poiroux et al., 2025); later constructions test its representation. Mathlib's convexity refactor illustrates this pressure: distributions and affine points need averaging without ambient addition; nested barycenters need a shared mixing law (Figure 2). Generated designs likewise differ in whether scalars are fixed or parameterized (Appendix F.4). Such choices shape a growing library (Baanen et al., 2026). Library learning evaluates new synthesis tasks, held-out compression, and search with learned li-

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

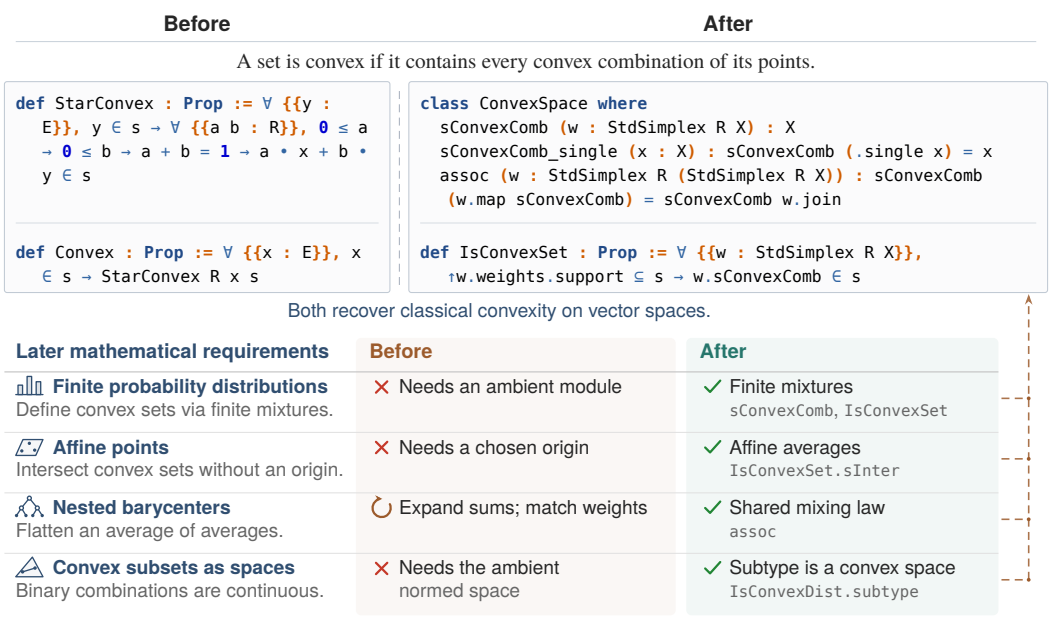


Figure 2: Two faithful convexity designs support later mathematics differently. ✓: direct support; ×: extra structure or representation needed; ⟳: repeated proof work. Dashed links mark design pressure from the four requirements. Ambient parameters are omitted; sources and full DAG: Appendix H.1.

braries (Ellis et al., 2023; Bowers et al., 2023; Grand et al., 2024). We focus on mathematical design: placing assumptions, organizing data and laws, and sharing constructions across proofs. DNA refactors corpus concepts for composing statements (Min et al., 2026a); Theory-Level Autoformalization advocates interdependent libraries and abstraction through refactoring (Min et al., 2026b). FUSE also relates to iterative self-refinement and execution-based debugging (Madaan et al., 2023; Shinn et al., 2023; Chen et al., 2023): those methods improve an answer or trajectory from feedback, whereas FUSE uses downstream mathematical uses to diagnose and revise a shared Lean statement, then tests the revision on replayed and held-out uses.

An *autoformalizer* produces a *statement*: a Lean module containing the main declaration, definitions, instances, and laws. Future-use generalization (FUG) measures support for withheld uses across targets and runs. FUSE (Future-Use-Guided Synthesis with Execution) searches for designs using downstream execution. Across eight authored development targets, exposure alone improves unseen declaration completion from 112/624 to 255/624; exposure and feedback on the scored uses raise observed declaration completion from 19.38% to 53.88%. Our contributions are:

1. We formulate mathematical abstraction design as future-use generalization, defining statement compatibility and generator-level FUG across representations, assumptions, structures, instances, and laws. The framework evaluates an abstraction through unseen mathematical requirements and records the additional work needed to reuse it.
2. We introduce FUSE, which traces downstream failures and recurring proof work to shared design choices, then proposes revisions with explicit expected effects. We test its components separately through use exposure, downstream repair, shared revision, and held-out evaluation, examining how execution can guide abstraction search.
3. We develop FutureUseBench, connecting major Mathlib refactor pairs, authored use graphs, and real-library checks. Historical pairs anchor design comparisons, authored graphs expose shared dependencies, and library replay checks compatibility with existing developments; each resource retains its own information and proof requirements.
4. We disentangle interface formation, downstream repair, and shared revision, measuring source tokens on jointly completed uses as a proxy for adaptation burden. Together with checks for dependence on unproved laws, this analysis distinguishes completing more uses from reducing the work needed to reuse an abstraction.

2 Problem Statement: Future-Use Generalization

We study abstraction design as search for faithful representations that support later mathematics. Given a natural-language target x and a fixed Lean library, an *autoformalizer* produces a statement s: the main declaration with supporting definitions, instances, and laws. It must compile and preserve the target's meaning; any generalization must recover the target through an instance or specialization. The design space includes representations, assumptions, and the organization of data, operations, and laws into structures. Properties may remain separate assumptions, and instances connect concrete types to shared structures. Later requirements are written independently of the candidate and may ask for new objects as well as proofs. Their completion reveals what can be reused and what requires extra conversions or bridge proofs. Figure 2 compares these costs; Appendix H.1 gives the Mathlib refactor evidence.

Later uses form a directed acyclic graph (DAG). Its edges specify prerequisites required by the task: shared objects or results that must be established before a dependent use is attempted. A reference proof's incidental helper steps do not by themselves impose such requirements. A downstream autoformalizer D tries each use once its prerequisites succeed; if a prerequisite fails, the uses depending on it are blocked, so fixing one shared law can reopen several branches. Successful proofs are informative too: they reveal conversions and bridge lemmas that the statement forced them to build, which we record as *adaptation burden*. Each target has two nonempty groups of uses (Figure 3). *Revision uses* Q_x supply the permitted design information while the autoformalizer improves the statement; their execution can provide future-use supervision. *Held-out uses* H_x are never shown to it and only evaluate the final statement, just as a test set evaluates a trained model. The protocol can withhold whole mathematical use families; the studies reported here use fixed use-ID and description splits within development targets (Section 5.2). Related uses may therefore occur on both sides. Whole-family holdout on untouched targets remains the intended generator-level test. The two groups may share general prerequisites, but held-out requirements and outcomes never return to revision. Evaluating either group also requires developing those prerequisites, but only the group's uses contribute to its score. Each held-out use that D cannot complete, whether it fails directly or is blocked, counts as unfinished.

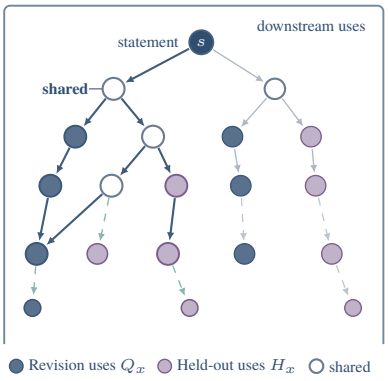


Figure 3: Uses of statement s.

Definition 1 (Future-use compatibility)**.** Let D formalize the held-out uses H_x on top of a statement s for target x. Under the specified dependency workflow, the compatibility of s is the fraction of these uses that D completes:

$$C_D(s) := \frac{\#\,\text{held-out uses completed}}{\#\,\text{held-out uses}} = \frac{\left|\{\, u \in H_x : D \text{ completes } u \text{ on top of } s \,\}\right|}{|H_x|}. \tag{1}$$

Definition 2 (Future-use generalization (FUG))**.** Run an autoformalizer M $n_x \geq 1$ times on each target in a nonempty test set $\mathcal{X}_{\text{test}}$, and let $s_{x,r}$ be its statement from run r. The reported FUG is the mean compatibility across these runs, with equal weight for each target and zero for runs that return no valid statement (Appendix A.3):

$$\text{FUG}(M) := \frac{1}{|\mathcal{X}_{\text{test}}|} \sum_{x \in \mathcal{X}_{\text{test}}} \frac{1}{n_x} \sum_{r=1}^{n_x} C_D(s_{x,r}). \tag{2}$$

Within each study, all compared methods use the same downstream autoformalizer and resources; the downstream model differs across studies (Appendix B.5). A target with many uses does not outweigh a target with few uses: completing half the held-out uses contributes 0.5 to that target's average in either case. FUG depends on D, the completion level (Table 2), and the declared held-out panel, which must be fixed before methods are compared; we write FUG_{decl} and $\text{FUG}_{\text{proof}}$ for declaration- and proof-level completion. Definition 2 assumes test targets untouched by method development; our panels split uses within development targets, so Table 7 gives development-split estimates of

FUG. Adaptation burden separates designs with similar completion but different downstream work. Appendix A gives proof requirements; Appendix G.4 measures downstream source size as a burden proxy.

3 FUSE: Designing Abstractions through Downstream Use

FUSE uses downstream consequences as a design signal for upstream abstraction formation (Figure 4). Within each target, an upstream autoformalizer A writes and revises the statement; a fixed downstream autoformalizer D attempts observed revision uses. The models' parameters remain unchanged. Each target starts a new candidate search using only its target, library context, and permitted revision information. Cross-target evaluation asks whether this fixed procedure repeatedly produces designs that generalize. A fixed generalization prompt guides the initial design, and subsequent execution supplies evidence for candidate updates. Algorithm 1 gives the search; Appendix B supplies its detailed rules.

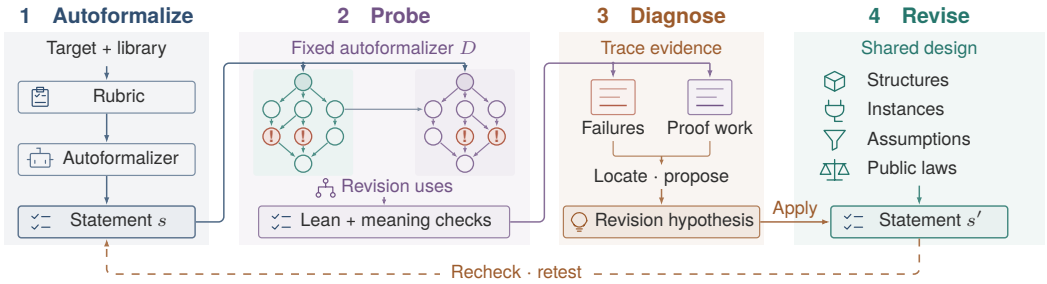


Figure 4: Future-use supervision in FUSE: execute a downstream DAG, revise the shared statement from its trace, and replay the same uses.

3.1 From Downstream Work to Design Evidence

D follows the task DAG, generating each use from the statement and accepted prerequisites. A failure with completed prerequisites identifies an attempted construction, the declaration it needed, and an unmet requirement. Blocked successors are traced to that same cause, so the DAG connects a local difficulty to its downstream consequences. Each trace identifies a concrete requirement and a possible shared cause, giving revision more specific guidance than the final completion count. A supported revision can interrupt execution immediately; otherwise D continues along unblocked branches. After a complete attempt, successful proofs provide a second signal: recurring conversions, instance constructions, structure packaging, and bridge proofs. A examines the concrete work behind these patterns and proposes a shared declaration that could replace it. Similar-looking steps are only clues; the proposal must explain their common mathematical role and how the uses would invoke it. Thus, even an all-successful run can expose a design choice that each proof has been compensating for locally. Appendix F.5 examines how successful proofs reuse an existing normal-closure/kernel bridge.

3.2 Revising and Testing the Shared Abstraction

A can change representations, move assumptions to results that need them, regroup data and laws into structures, revise instances, or introduce shared operations and laws. Before editing, it records the evidence, affected declaration, expected gain, and existing successes to recheck. D then formalizes the same uses under unchanged capabilities and requirements (Figure 5). Recovery means the failed use completes; reduced work means it completes without the named adaptation. Replay records regressions and marks expectations unresolved when execution cannot settle them. Further failures or recurring successful work can trigger another revision. Figure 5 illustrates shared proof completion and interface extension in a paired Ruzsa-distance example; Section 5.3 examines its gains and regressions.

Lean checks types and supplied proofs; an independent semantic checker compares the candidate with the target. FUSE saves accepted versions and evaluates each unchanged on all revision uses and prerequisites. Only complete evaluations enter selection, which chooses the most completed

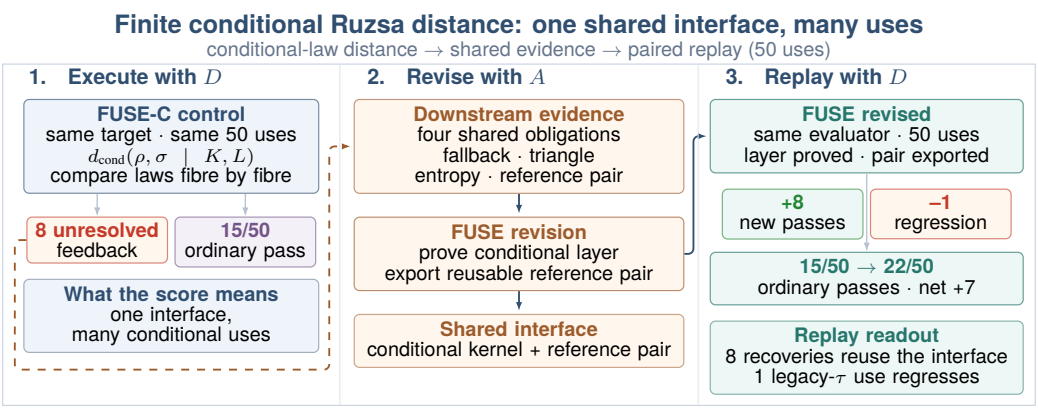


Figure 5: Paired finite conditional Ruzsa-distance replay on the same 50 observed uses. The distance compares conditional laws fibre by fibre. FUSE-C repairs the uses from the same feedback while keeping the upstream statement fixed; it obtains 15/50 ordinary passes, leaving eight upstream declarations unproved; FUSE adds a checked conditional layer and reusable reference pair, reaching 22/50 (eight gains, one regression). Source-screened counts approximate proof completeness.

revision uses and retains earlier designs if later edits regress. Held-out uses assess the final statement and never return feedback.

Algorithm 1: FUSE: revise a statement from its downstream uses

Input: Target x, library, revision uses Q_x; upstream autoformalizer A, downstream autoformalizer D; budget including final candidate evaluation
Output: Final statement s^*, or failure

Start with no saved statements and no pending update
for *each upstream round within budget* **do**
  A writes the initial statement s or applies the recorded update; check s and save it if accepted
  if *s fails Lean or proof-placeholder checks* **then** return errors to A; begin the next round
  Start a trace for s and clear the next update
  foreach *revision use or prerequisite, prerequisites first* **do**
    if *prerequisites did not pass* **then** mark the use blocked and continue
    D formalizes the use on top of s and its prerequisites; check the result
    if *the use fails* **then**
      Ask A for an edit, its evidence, expected gains, and successes to recheck
      if *the proposal is supported and another round is affordable* **then** record the update; stop trying uses
  if *a hypothesis produced s* **then** check its predicted gains and lost successes
  if *no next update is recorded* **then**
    Ask A to fix failed checks or, after a complete DAG attempt, propose a supported edit from repeated proof work
  if *no update is available or no further round is affordable* **then** end revision
Hold each saved statement unchanged; rerun all revision uses and prerequisites, score only Q_x, and keep candidates with complete evaluations
if *no candidate has a complete evaluation* **then return** failure
return the candidate s^* that completes the most revision uses, using the declared tie-breaking rule

Appendix H.2 specifies how to test whether search forms an abstraction absent from its context, through proved realizations, recovery of the target, and reuse in unseen families.

4 FutureUseBench: An Executable Test of Abstraction

4.1 Construction and Release

FutureUseBench connects a mathematical target to the later uses that test its design. Major Mathlib refactors provide alternative human designs; authored use graphs expose specific mathematical

requirements; real-library replay checks connections to existing developments. In the 378-pair inventory, eligible downstream declarations have a median count of 573 and a maximum of 187,402 per pair. These static footprints motivate use-based evaluation; a task DAG specifies which prerequisites each authored use requires. The construction process assembles targets, use requirements, dependencies, and a library environment for each measured study (Figure 6).

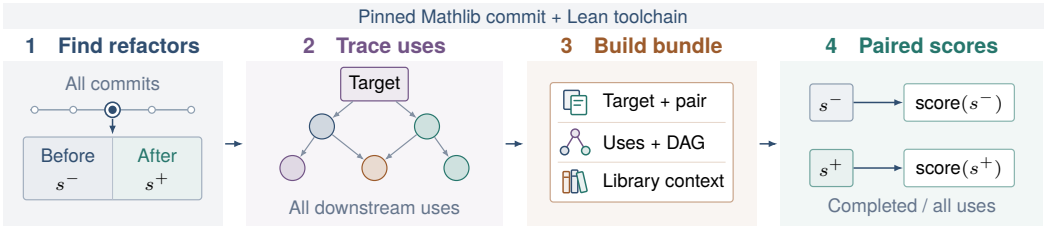


Figure 6: From provenance and reviewed task authoring to an executable task bundle: paired designs face the same downstream uses.

Table 1: Materials and measurement coverage. Draft uses and authored tasks describe the resource; the measured studies retain their own evaluation settings.

Material	Size	Description / Setting
Mathlib refactor inventory	13,905 source bundles	Extracted design-change candidates
Semantic pairs / draft uses	378 / 4,172	Screened pairs and drafted use requirements
Major refactors	5 pairs; 11-pair judge study	Human designs compared through use; judge rankings checked separately
Use-information study	8 targets, 50 uses each	Exposure and repair; two generation trials per target
Real-library replay	100 targets	Compile existing downstream dependencies
Extended authored tasks	53 targets, 2,650 uses	Development specifications; full evaluation outstanding

Table 1 distinguishes the resource from completed measurements. The 53 authored specifications provide 159 visible and 2,491 held-out uses; their complete evaluation remains outstanding. For generalization evaluation, uses withheld from revision assess the final statement without returning feedback. Related targets and equivalent use families require separation when constructing new tests; Appendix C.2 gives the grouping rules. The observed-use studies and the development-target held-out-use panel are identified separately in Section 5; neither is presented as a completed generator-level family-holdout benchmark.

Table 2: What completion requires. Ordinary counts follow the task's declaration or use-proof requirement; source-screened counts add an approximate dependency check.

Measure	Description
Declaration	Accepted declaration; designated theorem proofs may remain open. Table 5 uses this requirement.
Use proof	Complete use proof; shared upstream laws may remain unproved.
Full proof closure	Complete proofs throughout the dependency chain, including shared statement and instance laws; required for proof-closure FUG.
Source-screened	Excludes uses with source-detected dependencies on unproved declarations; approximates full proof closure without certifying it.

4.2 Executable Evaluation and Human Reference

The downstream autoformalizer D executes the held-out use DAG on top of the final statement, attempting prerequisites first (Figure 7). Completion requires satisfying the mathematical task, passing Lean and semantic checks, and actually using the statement's objects or laws. Table 2 states the proof

requirements for reading the results. Task construction, acceptance, and execution rules appear in Appendices A and C.2.

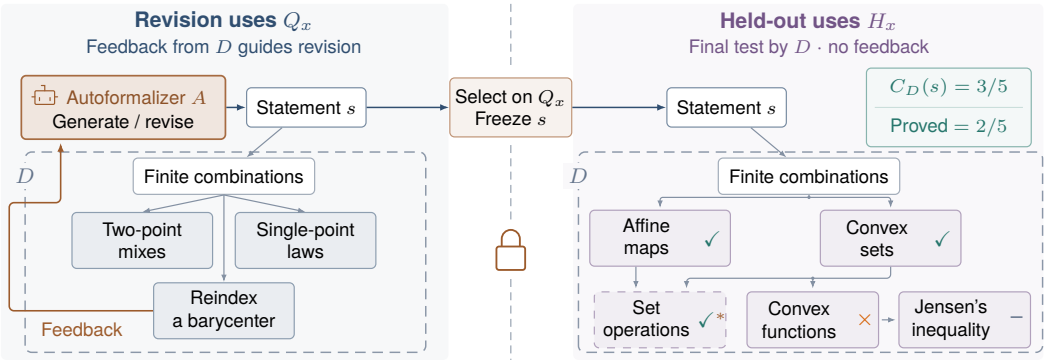


Figure 7: Example future-use DAG and its split. Dashed frames mark uses generated by D; white nodes are prerequisites and colored nodes are scored uses. ✓, ×, –, and ✓* denote completed, failed, blocked, and completed-with-unproved-theorem outcomes.

Mathlib's adopted statements provide human design comparisons. Removing completeness from positive operators unlocks four uses requiring that generality (Table 3). These comparisons test sensitivity to consequential design changes; Appendix D.2 gives their settings.

5 Experiments

We ask whether future-use evaluation informs design (RQ1), whether FUSE's components improve completion and generalization (RQ2), and why shared revisions help (RQ3). The studies separately test exposure and downstream repair (Table 5), joint revision (Table 6), and held-out evaluation (Table 7); they do not constitute a completed end-to-end evaluation of Algorithm 1.

5.1 Is Future-Use Evaluation Informative? (RQ1)

Across five major Mathlib refactors, adopted designs complete **60/60** use attempts against **29/60** for their predecessors, with no loss (Table 3). Averaging Definition 1's compatibility equally over the five targets gives **48.33%** before and **100%** after. Four designs improve and one ties. In the eleven-pair control study, same-intent refactors are nearly flat (91.75% to 92.79%), while capability and behavior controls rise from 59.52% to 85.71% (Appendix D.3). Completion responds to the mathematical work a design enables. At equal completion, adaptation work provides complementary evidence (Figure 2).

Table 3: Mean completion (%): four uses, three trials per refactor; Δ: after − before.

Refactor	Before	After	Δ (pp)
Proper cone	75	100	+25
Fréchet packaging	100	100	0
Positive operators	0	100	+100
Quadratic maps	0	100	+100
Bilinear forms	66.67	100	+33.33
Equal-target average	48.33	100	+51.67

Table 4: Four evaluators on eleven design pairs. Absolute τ_b ranks design coverage; Gain τ_b ranks after-minus-before gains.

Evaluator	What it sees	Absolute τ_b	Gain τ_b
Source-size proxy	Source only; prefers fewer bytes	0.142	−0.432
Target-only judge	Target and designs; no use information	0.283	−0.021
+ use requirements	Also sees mathematical use requirements	0.335	0.208
+ execution evidence	Also sees generated use code	0.414	0.391

We compare a source-size proxy and three judges with increasing use evidence (Table 4). The judge's agreement with measured gain rises from −0.021 to 0.391. All three judges already choose the

adopted design on the six non-tie pairs; execution improves estimates of how much it helps. Source shortening correlates negatively with reuse gain (Spearman $\rho = -0.589$). This source-byte proxy differs from learned-library compression (Appendix D.3).

5.2 Does FUSE Improve Completion and Generalization? (RQ2)

Repair in Tables 5–6 receives feedback on all 50 scored uses per target, measuring supervised observed-use completion, not FUG; FUSE-versus-FUSE-C FUG is therefore not measured. Table 7 withholds evaluated uses from upstream generation and revision.

Table 5: Declaration completion: eight authored development targets, two trials, 50 uses each. Accepted: the statement compiles and passes a model check of the task's interface requirements. Conditional: completions under accepted statements; Total: all opportunities. Repair receives feedback on the scored uses (observed-use completion, not FUG).

Condition	Setting	Accepted	Conditional	Total	Rate (%)
Direct	Generate from target and library	4/16	155/200	155/800	19.38
Visible	Add observed-use descriptions	9/16	349/450	349/800	43.63
+ binary repair	Fix Visible statement; repair uses with pass/fail feedback	9/16	418/450	418/800	52.25
+ detailed repair	Fix same statement; repair uses with structured feedback	9/16	431/450	431/800	**53.88**

Exposure raises upstream interface acceptance from **4/16 to 9/16**; conditional completion barely changes (77.50% versus 77.56%), locating most gains in interface acceptance, including conventions revealed by descriptions. On the same nine Visible statements that pass this gate, detailed repair reduces failures from **101 to 19** (81.2%). All five targets with passing Visible interfaces improve under both repairs; none decreases. The combined gain is 34.50 points (95% target-bootstrap interval [10.13, 62.50]), or $2.78\times$ Direct. Four repair configurations reduce failures from 346 to 145 (binary) and 71 (detailed); costs, target counts, and uncertainty are in Appendix G.2–G.3.

Table 6: Revision with feedback on scored, observed uses. Screened also removes uses that depend, directly or through named references, on unproved candidate or helper declarations, approximating proof dependence. Runs A–C share targets.

		Run C (each /1600)		Runs A–C (each /2800)	
Condition	**Setting**	**Ordinary**	**Screened**	**Ordinary**	**Screened**
Direct	Target and library only	163	163	238	238
Visible	Add observed-use descriptions	256	171	447	332
FUSE-C (fixed statement)	Repair uses; statement fixed	356	240	595	435
FUSE	Revise statement and uses	301	245	538	471

In Runs A–C, FUSE-C and FUSE complete 435 and 471 screened observed uses versus Direct's 238/2800 ($1.83\times$ and $1.98\times$). On jointly completed, screened uses, Visible reduces total source tokens by 16–27% (322 files shorter, 122 longer, 10 equal); description-withheld uses save 5–15% (127 shorter, 75 longer, 5 equal). However, full screened proof completion on the withheld subset falls from Direct's 429 to Visible's 395/2128. This separate proof-required analysis measures source tokens as a burden proxy, without upstream-gate filtering (Appendix G.4).

Held-out uses. Before repair, exposure raises unseen declaration completion from **112/624 to 255/624** ($2.28\times$; +22.92 points, 95% target-bootstrap interval [4.81, 45.67]). Conditional rates are 72.7% and 72.9%, locating most gains in interface acceptance. Two other configurations yield 111 to 146/608 and 110 to 163/608; they reuse the main Direct outputs, and one also draws on earlier failure analysis. These are dependent development comparisons. Completion by authored difficulty rises from 19% to 49% (core), 18% to 47% (derived), 31% to 46% (advanced), and 8% to 13% (research; Appendix G.2). After revision, binary feedback is numerically higher on held-out proofs (26.51% to 30.00%), but the +3.49-point difference has a 95% paired-target interval of $[-2.38, 9.84]$, crossing

Table 7: Held-out evaluation. Panel A withholds descriptions by use ID, so related uses can appear on both sides. Panel B gives feedback on IDs 1–20 and tests IDs 21–50 on seven development targets with three repeats, after one revision round rather than a full run of Algorithm 1. A detailed-feedback run is excluded because a workspace-check error affected its final selection (Appendix G.3).

Condition	Setting	Completed	Rate (%)
A. Declaration-level FUG, development split: pooled completion, eight targets			
Direct	Target and library only	112/624	17.95
Visible	Other descriptions; scored uses withheld	255/624	40.87
B. Proof-level FUG after one revision round: seven development targets			
Initial Direct	Initial statement without use exposure	167/630	26.51
No-diagnostic retry	Revises statement without diagnostic feedback	150/630	23.81
Binary feedback	Revises statement using pass/fail feedback	189/630	30.00

zero. Identical-statement re-evaluation differences have a standard deviation of 8.55 points across target–condition pairs (Appendix G.3).

5.3 Why Does FUSE Work When It Works? (RQ3)

Shared revisions help when several branches need the same missing proof or structural connection. In the finite conditional Ruzsa-distance case (Figure 5), FUSE-C leaves eight upstream declarations unproved; FUSE proves them and adds a reference-pair functional. Across 50 observed uses, ordinary completion rises from **15 to 22**, with eight gains and one regression; screened completion rises from **8 to 22**. Multiple branches reuse these shared layers. Across Runs A–C, FUSE also shortens downstream code on jointly completed, screened observed uses: 65 files are shorter than FUSE-C, 8 longer, and 270 equal, with consistent direction in all three runs. This measures joint revision of statements and use proofs (Appendix G.4).

In the simplicial-complex study, both runs receive identical prompts and pass local checks. Feedback identifies a missing parent projection; revision through inheritance passes the target, interface probe, and module of 38 direct uses. The control stops at the probe (Appendix F.5). Use evidence guides revision when it identifies a shared obligation or structural connection that later proofs consume.

Proof dependencies distinguish an available interface from a supported construction. Across Runs A–C, screening removes 55% of Visible's apparent gain over Direct: 209 additional completions become 94 (Table 6). The FUSE-C/FUSE ranking changes from 595 versus 538 to 435 versus 471. Shared revision also risks losing interface acceptance. Across all five configurations, screened observed-use totals are nearly tied: **899 versus 901 /6000** for FUSE-C and FUSE. A post-hoc breakdown locates the offset: in 49 trials where both final interfaces pass, completion rises from **823 to 901 /2450**; the 78 additional completions are almost entirely offset by 76 fewer completions in the remaining trials. This subset is selected after revision and describes where gains occur without isolating the effect of statement edits. The two repeated repair runs favor FUSE-C overall (Appendix G.3).

Limits. Exposure can also hurt: on ten Mathlib targets, Visible completes 47/112 declarations versus Direct's 53/112 (Table 28). Held-out proof gains remain unresolved amid evaluation variability (Table 7). Real-library replay improves pooled counts across 100 targets, but both target-equal intervals cross zero (Appendix F.2).

6 Conclusion

Mathematical work supplies an executable test and a design signal for abstraction: adopted Mathlib designs complete 60/60 use attempts against 29/60 for their predecessors, losing none. Exposure improves unseen declaration completion $2.28\times$; observed-use feedback supports repair and shared reuse, although joint revision can lose interface acceptance. Proof dependencies change design rankings, and source tokens reveal reuse costs. Reliable proof generalization after joint revision remains open. FUG and FUSE make this a concrete research objective: design abstractions by the work they enable, then test them on mathematics their designers have not seen.

AI USE STATEMENT

Large language models play two distinct roles in this work. As objects of study, they serve as the upstream and downstream autoformalizers, semantic checkers, and design judges whose behavior the experiments measure; Appendices B.5, D.2, and G record the models and settings of each study. Separately, we used generative AI tools as research assistants. LLM-based coding agents helped implement experiment runners, Lean workspace tooling, and analysis scripts, and recomputed reported statistics from archived raw records; the supplementary analysis scripts check the reported values against those records. LLM assistants also helped draft and revise text, keep the English manuscript aligned with a Chinese review version, organize related literature, and prepare LaTeX and TikZ figures. Generative AI was not used to create or alter experimental outcomes. The authors reviewed all AI-assisted code, analyses, and text, checked the mathematical statements and references, and take responsibility for the final content, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work uses public Mathlib sources under the Apache 2.0 license and cites public design discussions by link; it involves no human-subject data. Because generated statements can compile while misrepresenting mathematics, our evaluation reports compilation, semantic checks, and proof dependence separately.

REPRODUCIBILITY STATEMENT

Section 2 and Appendix A define compatibility, FUG, and the S, P, and P^* completion requirements. Appendix B gives the complete FUSE procedure, information access, and feedback rules. Appendix C describes task construction and the splits used, and Appendices D–G report each measured study with its models, denominators, and uncertainty. The anonymized supplementary material contains the FUSE harness, prompts, and run configurations; the FutureUseBench specifications and Mathlib refactor inventory; the archived per-use outcomes and final Lean sources behind each table; and scripts that recompute the reported numbers from these archives without model calls or Lean runs (Appendix I).

REFERENCES

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

Anne Baanen, Matthew Robert Ballard, Johan Commelin, Bryan Gin-ge Chen, Michael Rothgang, and Damiano Testa. Growing mathlib: Maintenance of a large scale mathematical library. In *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, pp. 51–70. Springer Nature Switzerland, 2026. doi: 10.1007/978-3-032-07021-0_4. URL https://doi.org/10.1007/978-3-032-07021-0_4.

Ian Berlot-Attwell, Frank Rudzicz, and Xujie Si. LLM library learning fails: A LEGO-Prover case study, 2025. URL https://arxiv.org/abs/2504.03048.

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, 2023. doi: 10.1145/3571234. URL https://doi.org/10.1145/3571234.

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. babble: Learning better abstractions with e-graphs and anti-unification, 2022. URL https://arxiv.org/abs/2212.04596.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sablé-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 381(2251), 2023. doi: 10.1098/rsta.2022.0050. URL https://doi.org/10.1098/rsta.2022.0050.

Yuming Feng, Frederick Pu, One An, Osbert Bastani, Li Zhang, Jiani Huang, Xujie Si, and Ziyang Li. Theory-scale auto-formalization of logics for computer science, 2026. URL https://arxiv.org/abs/2606.26525.

Gabriel Grand, Lio Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. LILO: Learning interpretable libraries by compressing and documenting code. In *International Conference on Learning Representations*, pp. 30399–30446, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/819cebb05f993840e8a52d7564c5c282-Abstract-Conference.html.

Jiewen Hu, Thomas Zhu, and Sean Welleck. miniCTX: Neural theorem proving with (long-)contexts. In *International Conference on Learning Representations*, pp. 10106–10130, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/1b5ef7bcc702a0232b4f1aea2523d0d2-Abstract-Conference.html.

Bart Jacobs. Convexity, duality and effects. In *Theoretical Computer Science*, volume 323 of *IFIP Advances in Information and Communication Technology*, pp. 1–19. Springer, 2010. doi: 10.1007/978-3-642-15240-5_1. URL https://dl.ifip.org/db/conf/ifipTCS/ifipTCS2010/Jacobs10.pdf.

Jongyoon Kim, Hojae Han, and Seung-Won Hwang. Benchmarking testing in automated theorem proving. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 2241–2260. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-industry.150. URL https://aclanthology.org/2026.acl-industry.150/.

Adarsh Kumarappan, Mo Tiwari, Peiyang Song, Robert Joseph George, Chaowei Xiao, and Anima Anandkumar. LeanAgent: Lifelong learning for formal theorem proving, 2024. URL https://arxiv.org/abs/2410.06209.

Zhening Li, Gabriel Poesia, Omar Costilla-Reyes, Noah Goodman, and Armando Solar-Lezama. LEMMA: Bootstrapping high-level mathematical reasoning with learned symbolic abstractions, 2022. URL https://arxiv.org/abs/2211.08671.

Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-STaR: Learning to interleave thinking and proving. In *International Conference on Learning Representations*, pp. 66041–66062, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a5357781c204d4412e44ed9cbcdb08d5-Abstract-Conference.html.

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

Qi Liu, Xinhao Zheng, Renqiu Xia, Xingzhi Qi, Qinxiang Cao, and Junchi Yan. Beyond theorem proving: Formulation, framework and benchmark for formal problem-solving, 2025b. URL https://arxiv.org/abs/2505.04528.

Haijian Lu, Wei Wang, and Jing Liu. FormalEvolve: Neuro-symbolic evolutionary search for diverse autoformalization, 2026. URL https://arxiv.org/abs/2603.19828.

Jianqiao Lu, Yingjia Wan, Yinya Huang, Jing Xiong, Zhengying Liu, and Zhijiang Guo. FormalAlign: Automated alignment evaluation for autoformalization, 2024. URL https://arxiv.org/abs/2410.10135.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Marcus Min, Yeqi Gao, Wilson Sy, Zhaoyu Li, Xujie Si, and Osbert Bastani. Divide and abstract: Autoformalization via decomposition and abstraction learning. In *International Conference on Learning Representations*, pp. 98205–98255, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/9f94298bac4668db4dc77ddb0a244301-Abstract-Conference.html.

Marcus Min, Deyuan Mike He, Zhaoyu Li, Zixuan Yi, Sharad Malik, Aarti Gupta, Xujie Si, and Osbert Bastani. Position: The case for theory-level autoformalization. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026b. URL https://icml.cc/virtual/2026/poster/67212.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17947–17969, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.907. URL https://aclanthology.org/2025.emnlp-main.907/.

Lushi Pu, Weiming Zhang, Xinheng Xie, Zixuan Fu, Bingxiang He, Hengyu Zhao, Hongya Lyu, Xin Li, Jie Zhou, and Yudong Wang. MathForm: Scaling mathematical autoformalization with knowledge retrieval and verification-guided refinement, 2026. URL https://arxiv.org/abs/2608.14221.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Jasivan Alex Sivakumar, Philipp Borchert, Ronald Cardenas, and Gerasimos Lampouras. Conjecturing: An overlooked step in formal mathematical reasoning, 2025. URL https://arxiv.org/abs/2510.11986.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, Heng Liao, and Xiaodan Liang. LEGO-Prover: Neural theorem proving with growing libraries, 2023. URL https://arxiv.org/abs/2310.00656.

Haocheng Wang, Baiyu Huang, Yingjia Wan, Xiao Zhu, Xiaoyang Liu, Yinya Huang, and Zhijiang Guo. FormalRx: Rectify and eXamine semantic failures in autoformalization, 2026a. URL https://arxiv.org/abs/2607.04655.

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

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems 35*, pp. 32353–32368, 2022. doi: 10.52202/068431-2344. URL https://doi.org/10.52202/068431-2344.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Haowei Zhang, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. In *International Conference on Learning Representations*, pp. 72274–72303, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/b3b55c366d641c07180c40e4f978f311-Abstract-Conference.html.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models, 2023. URL https://arxiv.org/abs/2306.15626.

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

Thomas Zhu, Joshua Clune, Jeremy Avigad, Qiaochu Jiang, and Sean Welleck. Premise selection for a Lean hammer. In *International Conference on Learning Representations*, pp. 127594–127607, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/cf6501108fced72ee5c47e2151c4e153-Abstract-Conference.html.

Sections A–D define scoring, the FUSE procedure, benchmark construction, and evaluation checks, adding the corresponding measurements where available. Section E specifies the complete evaluation that has not been run. Sections F–H report archived measurements and cases under their original conditions, and Section I describes reproduction materials and scope.

A Definitions and Scoring Rules

A.1 Statement Acceptance and Proof Status

An autoformalizer outputs a statement containing the main declaration and supporting definitions, instances, and law declarations. We revise, select, and freeze this Lean module as a whole. Acceptance has three checks: Lean compilation and the proof-placeholder policy, semantic agreement with the current natural-language requirement, and the public-declaration rule. An independent language-model checker reviews objects, quantifiers, assumptions, and conclusions; Lean checks types and proofs. Public declarations may introduce a shared structure absent from the requirement when a concrete instance or specialization recovers the original target. They must serve that shared mathematical role and cannot contain constants, fields, or answers specific to one use. Local helper lemmas remain in the use files. Designated theorem-proof positions may contain `sorry`, recorded as unproved propositions. Placeholders cannot replace definitions, data, or data-carrying instances; additional axioms are disallowed. Each candidate record lists open proofs and their dependencies. Statement acceptance, completion of a use proof, and a fully proved dependency closure are distinct statuses. Compilation does not turn an open proposition into a proved theorem.

A.2 Completed Uses and Adaptation Burden

Before execution, each node is assigned a declaration-only or proof-required task. A declaration-only output must compile, pass a semantic check against its requirement, and use the candidate's declarations. Designated theorem-proof positions may remain open. A proof-required output must also complete its own and its helpers' proofs without new admissions. If it depends on unproved propositions in the candidate or prerequisite nodes, these are listed as premises: the use proof is checked relative to those declarations. Only a dependency closure with no open proofs supports a claim that the whole development is proved. *Proved compatibility* counts a scored use only when its whole dependency closure, including theorem positions left open in the candidate and law fields of every instance it uses, has no open proofs; *proved FUG* averages it exactly as in Equation (3). Ordinary FUG measures what a statement makes expressible and provable relative to its stated laws, while proved FUG removes any credit that rests on unproved laws.

Use may be indirect through accepted prerequisites, but must trace to the candidate's objects, operations, or laws. An import, a name mention, or an irrelevant dependency does not suffice. Downstream tasks may construct instances, convert representations, and prove bridge lemmas; they may not reimplement the candidate to bypass it. The original use context must support any extra assumptions required by the candidate. Simply strengthening the use's assumptions does not count as completion. Adaptation burden records extra assumptions and how they are met, representation or type conversions, and bridge proofs separately. Node records also retain work on packing and unpacking structures and constructing instances. Public-declaration size, cross-branch reuse, and compute consumption are separate measurements, not an undefined combined score.

A.3 Traversal Outcomes and Score Denominators

All execution and scoring below refer to the declared task DAG, not the unreviewed source dependency graph. For a scoring set U, its execution graph $\overline{U}$ contains U and all prerequisites. The revision and held-out scoring sets Q_x, H_x are nonempty and disjoint; their execution graphs may share prerequisites. A full traversal keeps the statement fixed, attempts each node whose prerequisites completed, and records nodes behind failed prerequisites as blocked. This produces the completed set $\mathcal{K}_D(s;\overline{U})$. The completion rate on U is $C_D(s;U) \coloneqq |\mathcal{K}_D(s;\overline{U}) \cap U|/|U|$; Definition 1 is the held-out case $C_D(s) = C_D(s;H_x)$, and selection uses $C_D(s;Q_x)$. The failure frontier records attempted failures with completed prerequisites. Tracing a blocked node backwards identifies the frontier failures responsible for it, allowing diagnosis to group several blocked uses under a shared cause.

Failed, blocked, and timed-out scoring nodes remain in the denominator. Prerequisites outside U count toward execution cost and supply declarations to later nodes. A global interruption may leave nodes unvisited; that traversal is incomplete and cannot serve as a candidate-selection score.

With the held-out set written explicitly, Definition 2 is

$$\mathrm{FUG}(M) = \frac{1}{|\mathcal{X}_{\text{test}}|} \sum_{x \in \mathcal{X}_{\text{test}}} \frac{1}{n_x} \sum_{r=1}^{n_x} C_D(s_{x,r}; H_x), \tag{3}$$

where $\mathcal{X}_{\text{test}}$ is the set of test targets, n_x the number of trials on target x, and $s_{x,r}$ the statement returned in trial r: compatibility is averaged over trials within each target and then equally across targets. These targets are held out from method development. Each trial fixes the model, prompts, access rules, revision procedure, selector, and stopping rule; a trial returning no accepted statement contributes zero. Revision and held-out evaluation share the downstream model, prompt, tools, decoding, per-node budget, semantic checker, and acceptance rules. The held-out phase rebuilds prerequisites for the frozen statement and scores H_x. See Appendix C.2.

B FUSE Implementation Details

Algorithm 1 in Section 3 summarizes the revision loop and final selection. This appendix defines its evidence and update records; Algorithm 2 gives the complete procedure with validation, access, and budget branches.

B.1 Roles and Information Access

The upstream autoformalizer generates and revises the statement. The executor calls a fixed downstream autoformalizer to complete use nodes with the current candidate. An independent semantic checker compares upstream and downstream outputs with their respective requirements. Lean compilation and proof checking are separate. Revision scoring nodes Q_x and their execution closure $\overline{Q}_x$ provide design evidence; held-out scoring nodes H_x evaluate the final statement. Descriptions in Table 8 are natural-language requirements, not reference Lean implementations or proofs.

Table 8: Inputs to the upstream autoformalizer.

Mode	**Initial generation**	**Revision**
Full FUSE	Target, library, descriptions	Descriptions and feedback
Feedback-only FUSE	Target and library	Feedback only

Both receive the fixed generalization prompt and the same setting of basic validation feedback. Feedback-only hides use descriptions from the upstream autoformalizer during both generation and revision. The downstream autoformalizer reads its own node requirements in both modes. Held-out uses and outcomes never enter upstream inputs, diagnosis, or selection. The fixed generalization prompt asks for a formulation that preserves the target and can support later applications. It poses five concrete questions: which objects to use, where assumptions belong, how to represent data, how operations compose, and which laws to expose. These cover grouping or separating structure fields, keeping parameters and application-specific properties local, and supplying structures through instances. Appendix B.5 reserves the exact text.

B.2 Execution Traces and Revision Hypotheses

Let s_t be the statement at revision round t, A the upstream autoformalizer, and D the fixed downstream autoformalizer. A run on the revision DAG $\overline{Q}_x = (V, E)$ creates a version-specific trace τ_t. Each node record distinguishes passed, failed, blocked, timed out, and unvisited states, with accepted prerequisites, diagnostics, proof status, actual use of the candidate, adaptation work, and cost. Starting a revision from an early failure leaves a partial trace. Nodes blocked by that failure remain incomplete for scoring but do not supply separate failure evidence.

Let $V_t \subseteq V$ contain the nodes actually attempted, and let $o_t(v) \in \{\mathsf{pass}, \mathsf{fail}\}$ coarsen their outcomes, counting an exhausted node-level budget as a failure. Define the passed set and failure frontier:

$$\mathsf{Passed}_t \coloneqq \{v \in V_t : o_t(v) = \mathsf{pass}\}, \qquad F_t \coloneqq \{v \in V_t : o_t(v) = \mathsf{fail},\ \mathrm{Pred}(v) \subseteq \mathsf{Passed}_t\}, \tag{4}$$

where $\mathrm{Pred}(v)$ contains the direct prerequisites of v. Blocked and unvisited nodes do not enter V_t. For a passed node, $\alpha_t(v) \subseteq \mathcal{A}$ records its adaptation categories. The fixed vocabulary $\mathcal{A}$ covers representation conversion, instance construction, structure packing or unpacking, and bridge proofs. Write $S_t(c) \coloneqq \{v \in \mathsf{Passed}_t : c \in \alpha_t(v)\}$. After a complete traversal, define

$$W_t \coloneqq \{(c, S_t(c)) : c \in \mathcal{A},\ |S_t(c)| \geq 2\}, \qquad z_t \coloneqq \Phi_m(\tau_t \restriction_{F_t}, W_t)\,. \tag{5}$$

During an interrupted traversal, the repeated-work view is empty and failure feedback remains available. The map Φ_m retains only records allowed by information mode m. Full mode permits revision-use descriptions; Feedback-only mode hides them and filters private code and context from execution records. The category groups in W_t are search cues: two conversions or bridge proofs may concern different constructions. A revision hypothesis must identify the concrete work a shared edit is expected to remove; the group alone does not establish such a common abstraction. Cross-branch recurrence is recorded but is not a requirement.

Before changing Lean code, the upstream autoformalizer records a revision hypothesis:

$$h_t \coloneqq (B_t,\ e_t,\ \Gamma_t,\ \mathcal{P}_t). \tag{6}$$

Here B_t cites records in z_t, $e_t \in \mathcal{E}(s_t)$ specifies an edit from the allowed set, Γ_t is a nonempty set of tests for expected gains, and $\mathcal{P}_t \subseteq \mathsf{Passed}_t$ lists successful uses to monitor. The edit e_t records both a site in s_t and the change made there. A site identifies a declaration, argument, assumption, structure field, instance, operation, or law, or names a proposed new declaration. Edit families cover changing representations, relocating assumptions, grouping or separating structure fields, revising instances, and changing public operations or laws. Each gain names a recorded failure or a concrete work item and a corresponding replay condition. Removing a bridge proof counts as a gain only if the use still succeeds.

A hypothesis is well formed when its cited evidence resolves in the permitted trace view, its site resolves or names a new declaration, its edit belongs to a declared family, its gain tests are evaluable under the fixed record schema, and its protected uses were observed to pass. Feedback-only proposals use allowed anonymous identifiers, not private task text or code. The formal calls are

$$h_t = \mathrm{Hyp}_{A,m}(x, s_t, I_m(\overline{Q}_x), z_t), \qquad \widetilde{s}_{t+1} \coloneqq \mathrm{Revise}_A(x, s_t, h_t), \tag{7}$$

where $I_m(\overline{Q}_x)$ contains exactly the permitted use descriptions. The before/after statement diff and hypothesis are stored together. With the same downstream model, tasks, topological order, tools, checks, and node budgets, replay produces

$$\widetilde{\tau}_{t+1} \coloneqq \mathrm{Exec}_D(\overline{Q}_x; \widetilde{s}_{t+1}). \tag{8}$$

Randomness and any paired seeds belong to the declared evaluation configuration; fixing the executor does not make sampled model outputs deterministic.

Each gain test $g \in \Gamma_t$ maps a replay trace to 1, 0, or $\perp$: confirmed, contradicted, or unresolved. Use 0 when the available outcomes settle the prediction negatively. In particular, a node-recovery test is contradicted by failure, node-level timeout, or known blockage by a failed prerequisite. Use $\perp$ only when the recorded execution is insufficient to decide the test, as can happen after an early interruption. An unvisited node is not automatically unresolved if a known prerequisite failure already settles the condition. A protected use likewise records preservation, regression, or an unresolved outcome; failed, timed-out, or known-blocked protected uses are regressions.

Let $\widetilde{\mathsf{Passed}}_{t+1}$ be the passed nodes in replay and $\mathcal{S}_x$ the accepted statement class. The named predictions receive full observational support precisely when

$$(\widetilde{s}_{t+1}, \widetilde{\tau}_{t+1}) \models_x h_t \quad \Longleftrightarrow \quad \widetilde{s}_{t+1} \in \mathcal{S}_x \ \wedge\ (\forall g \in \Gamma_t,\ g(\widetilde{\tau}_{t+1}) = 1) \ \wedge\ \mathcal{P}_t \subseteq \widetilde{\mathsf{Passed}}_{t+1}. \tag{9}$$

This relation concerns the committed tests and protected uses. It implies neither a higher whole-DAG completion rate nor preservation of successes outside $\mathcal{P}_t$. All observed regressions are retained, including those outside $\mathcal{P}_t$; complete candidate evaluations determine overall benefit. A contradicted

or unresolved hypothesis remains in the audit trail and does not erase an otherwise acceptable statement snapshot. Matched replay supplies controlled observational evidence, not a formal causal proof. Basic Lean or semantic repair uses a separate record grounded in candidate validation. Downstream-driven revision uses the versioned trace, permitted evidence, edit hypothesis, and matched replay. The language model proposes the diagnosis and Lean code; the recorded predictions make that proposal testable.

B.3 Online Revision, Archiving, and Selection

Algorithm 2 allows at most T upstream-formalization rounds: the initial draft and every revised statement consume one round, including versions that fail compilation. A frontier failure or timeout returns the permitted design evidence immediately. A grounded, well-formed hypothesis can start the next round without waiting for the rest of the DAG. If no such edit is available, the executor continues along unblocked nodes. After a complete traversal, the upstream autoformalizer may also form a hypothesis from repeated-work patterns. Finishing every node therefore need not end revision. Review, editing, and replay all consume the same online budget.

Each new statement is checked in Lean and against the proof-placeholder policy. A failure returns permitted validation errors and skips downstream execution. Compilable statements undergo meaning and public-declaration checks. Semantic judgments and reasons remain private except for fields allowed by the binary or detailed setting. A compilable but semantically unsuitable statement may still produce a diagnostic trace, but it cannot enter selection. Every statement that passes all upstream checks is saved as an immutable snapshot. When a revision is attempted, the next run rebuilds and rechecks downstream outputs against the new statement in dependency order. An earlier accepted downstream file is evidence about the old version, not acceptance under the new one. The trace records which expected gains were realized and which monitored or unexpected regressions occurred. A completed DAG with an unacceptable upstream statement can still inform a subsequent semantic repair. Online revision stops when no new hypothesis is used or the round or resource limit is reached.

For selection, let Saved_x be the finite set of snapshots that passed all upstream checks. Each is held fixed and evaluated afresh on $\overline{Q}_x$, without upstream revision, using common downstream and scoring settings. The selectable archive is

$$\mathsf{Pool}_x := \{s \in \mathsf{Saved}_x : \text{selection evaluation of } s \text{ on } \overline{Q}_x \text{ is complete}\}. \tag{10}$$

A complete evaluation assigns every node a passed, failed, timed-out, or blocked outcome. Failed, blocked, and timed-out scoring nodes remain in the denominator. An interrupted evaluation with unsettled nodes is not complete and cannot enter Pool_x; its partial score is not used for selection. If $\mathsf{Pool}_x \neq \varnothing$, choose

$$s_x^* \in \operatorname*{arg\,max}_{s \in \mathsf{Pool}_x} C_D(s; Q_x), \tag{11}$$

where $C_D(s; Q_x)$ is the completion rate on the scoring nodes Q_x after executing $\overline{Q}_x$. A predeclared rule over separately reported burden components and compute resolves ties. An empty selectable archive returns failure even if saved snapshots exist. Earlier versions remain eligible when later edits regress. The selected statement is frozen for held-out evaluation, which never returns feedback.

The archive retains selectable statements and evaluation records, without carrying revision-generated downstream proofs or helpers into held-out execution. Full code and partial online traces may be kept separately for private audit. Binary semantic-feedback runs must hide checker reasons in all upstream-visible messages and archive views. Selection evaluation and repeated online work both count toward the total budget.

Algorithm 2: FUSE: trace-grounded abstraction revision

Input: Target x; Lean library; disjoint nonempty scoring sets Q_x, H_x and their predecessor-closed execution DAGs $\overline{Q}_x, \overline{H}_x$. Every node specifies a natural-language task and whether it requires a declaration or a completed proof; edges point from prerequisites to dependents. The evaluator keeps H_x hidden until freezing.

Roles: An upstream autoformalizer writes statements; a fixed downstream autoformalizer writes use nodes; an independent semantic checker reviews meaning; Lean checks types and proofs.

Settings: Fixed generalization prompt, information-access mode, semantic-feedback mode, total budget with a selection reserve, and at most T upstream-formalization rounds including the first draft.

Output: Frozen statement s_x^* with proof status, or failure

Initialize an empty snapshot set and no current update record
for *at most T upstream-formalization rounds* **do**
  The upstream autoformalizer writes the first s, or applies the current update record to the previous s
  if *s fails Lean compilation or the placeholder policy* **then**
    Return permitted validation errors upstream; save a validation-repair record; start the next round
  Check s against x and the public-declaration rule; store the private verdict and reasons
  if *s passes all upstream checks* **then** save an immutable snapshot of s
  Initialize the version-specific trace τ_s and set the next update record to none
  foreach *v in a fixed topological order of $\overline{Q}_x$* **do**
    if *the online budget is exhausted* **then** end the current traversal
    if *a prerequisite failed, timed out, or is blocked* **then** record v as blocked
    else
      The downstream autoformalizer generates the declaration and required proof from s, the task at v, and accepted prerequisite declarations
      Check compilation, meaning, proof completion, and actual dependence on s; append the outcome and adaptation work to τ_s
      if *v passes* **then** make its declaration available to dependents
      else
        Extract the frontier failure; immediately return its permitted evidence upstream
        Ask for $h = (B, e, \Gamma, \mathcal{P})$: cited evidence, a typed edit at a named site in s, expected gains, and successful nodes to protect
        if *h is grounded, well formed, and another round is affordable* **then** record h as the next update; end the current traversal
  If a hypothesis produced s, test Γ and monitor $\mathcal{P}$; known failure or blockage contradicts recovery, while insufficient execution remains unresolved
  if *no next update has been recorded* **then**
    if *s failed an upstream semantic or public-declaration check and another round is affordable* **then**
      Create a validation-repair record from the permitted checker feedback; record it as the next update
    else if *the traversal completed and another round is affordable* **then**
      Group recurring adaptations among passed nodes as design cues; return the permitted evidence
      Ask for a revision hypothesis with the same four fields
      if *the result is grounded and well formed* **then** record it as the next update
  if *a next update exists* **then**
    Set its record as current; start the next upstream-formalization round
  else
    End online revision
For every accepted snapshot, hold it fixed and rerun all of $\overline{Q}_x$ without upstream revision; score only Q_x, and admit only complete evaluations to the candidate archive
if *the archive is empty* **then return** *failure*
Choose s_x^* by completed-use fraction; resolve ties by the predeclared rule over separate burden components and compute
Freeze and **return** s_x^*; execute $\overline{H}_x$ once and score only H_x, with no feedback

B.4 Semantic Feedback and Downstream Feedback

For upstream semantic feedback, the checker reads the original target, candidate, and fixed library context, without downstream uses. Judgment and reasons are stored privately before reporting. Binary feedback returns only pass/fail; detailed feedback also returns the reasons. Both use the same check and acceptance rule. Binary messages, logs, and archive views exclude checker reasons, error categories, locations, confidence, and suggested edits. Existing Lean diagnostics may be returned, but cannot carry these private fields. These are independent feedback factors. The four main conditions share one upstream semantic-feedback setting. Its binary/detailed ablation changes only the reasons returned upstream. Node-level semantic judgments remain part of downstream-use feedback, whose format stays fixed during that ablation.

For downstream execution feedback, the comparisons use binary outcomes, filtered compiler messages, and structured errors with dependency checks. The main FUSE protocol also returns permitted summaries of successful adaptation after complete traversals. A successful-work ablation removes that extra review, keeping private logging and final selection unchanged. Feedback-only retains anonymized node or branch labels to compare rounds, while removing private use identifiers, use text, local context, and reference content. Filtering applies to failed and successful nodes alike, including effort summaries: code, types, and local contexts that disclose the task remain private. Aggregate categories or counts may be returned when allowed by the fixed schema. A binary-only downstream variant does not silently receive structured effort fields. The upstream autoformalizer's own statement and compilation errors are not hidden use descriptions. The feedback fields used in the reported runs are defined by the supplementary harness, prompts, and configurations (Appendix B.5); the Feedback-only filtering described above belongs to the full protocol, which the reported runs do not execute (Appendix E).

B.5 Prompt and configuration record

The reported studies use a shared generalization prompt, an upstream autoformalizer, a downstream autoformalizer, and an independent semantic checker. The upstream input contains the target and permitted library context; use descriptions and execution feedback are added only in the conditions named in the main tables. The downstream input contains one use requirement and accepted prerequisites, and the checker sees the candidate and its own requirement. Binary feedback returns a verdict, while detailed feedback additionally returns the recorded diagnostic fields. The exposure and repair study uses GPT-5.4/high for upstream generation, GPT-5.6-sol/high for semantic checks, two generation trials per target, and up to three downstream repair rounds; the five-pair history calibration uses GPT-5.6-luna/low downstream and GPT-5.6-sol/high for checking. Other archived studies retain their own model and snapshot records, stated with their results below. The exact prompt and configuration files, with input hashes, are included in the supplementary material; proposed variants without an archived run are not treated as measurements in this paper.

C FutureUseBench Construction

C.1 Cases, Dependency Graphs, and the Four Tests

Each case contains the current mathematical requirement, fixed library context, allowed imports, a task DAG, and declaration or proof requirements for each node. History-derived cases originate in Mathlib refactors; authored-use and real-library studies retain their own provenance and evaluation settings. For a history-derived case, downstream tasks are derived from uses of the affected declarations in a fixed snapshot. The complete source graph remains a provenance record. Before evaluation, reviewers construct the task DAG by classifying source edges: a mandatory edge supplies an object or result explicitly required by the dependent task; an implementation edge records a helper, conversion, or proof route used by the reference. Only mandatory edges block execution. Optional implementation edges remain in the source record, and implementation-only helper nodes can be omitted from the task graph with the reason recorded. An edge is mandatory relative to the stated task contract, not a claim that every proof of the underlying theorem must use it. Any proof or representation satisfying that contract is accepted, without requiring the reference's helper names or proof steps.

Every mandatory edge needs a task-level justification and reference feasibility check, recorded before evaluating any candidate or viewing its score. If reviewers cannot separate an implementation step from a required object, the case is deferred from the primary test. It may be reported separately under its original workflow, with that restriction explicit. The executor still schedules a fixed DAG; it does not choose a new dependency graph per candidate. Consequently, even reviewed scores measure compatibility with the prescribed mathematical workflow, not proof-path-independent abstraction quality. Reference code remains private and is never scored as a model output.

Use counts in Table 1 sum target–use memberships, not repeated executions. Family counts sum target–family memberships; refactor-group counts measure the split units. Node counts and depth describe the reviewed task DAG, with depth counted in edges; source-graph sizes and removed implementation edges are reported separately. The four tests differ in the work they require (Table 9).

Table 9: Four tests of future-use generalization. Every outcome is evaluated on held-out uses.

Test	**Required outcome**
New-object reuse	Apply the same definitions or laws to an unseen object
Cross-theory composition	Join results from two theories in one downstream use
Guided construction	Build a hinted abstraction and reuse it in new branches
Abstraction discovery	Infer an unhinted abstraction and reuse it in new branches

The hint contains no Lean signature, code, or proof and remains visible during generation and revision. It is independent of access to use descriptions, so guided and discovery trials separate following a supplied design from proposing one. All four tests use the completion rules in Appendix A: the output compiles, passes semantic review, uses the statement's declarations, and supplies the required proofs; similar names or irrelevant references do not establish reuse.

Resources measured here. The completed studies do not instantiate the four tests. They use five major Mathlib refactor pairs with four fixed uses each, an eleven-pair judge study with 91 primary uses, and eight authored development targets with 50 uses each. The released authored specifications contain 53 targets and 2,650 uses (159 visible, 2,491 held out); each task graph has 51 nodes, a median of 124 edges (interquartile range 113–147; range 63–197), and a median depth of 9 edges (interquartile range 8–12; range 4–23).

C.2 Use Splits and Held-Out Execution

The whole-family and untouched-target construction below specifies the intended generator-level evaluation. The completed studies in Table 7 instead use fixed use-ID/description splits within authored development targets; related uses may occur on both sides. No whole-family generator-level result is reported here.

Before prompt tuning, method development, or test execution, form an undirected relation on targets. Link targets from the same refactor commit or documented refactor PR, targets sharing an edited declaration after accounting for renames, and targets sharing an identical or equivalent downstream task or a narrowly defined mathematical use family. Its connected components are the refactor groups. A family identifies the same construction or tightly related specializations; a broad field label or a common background Mathlib import does not suffice. Reviewers fix family identities and equivalence decisions without method scores, conservatively merging or withholding uncertain links. Allocate whole groups to development or test. Targets already used for calibration or method choices, and all targets linked to them, stay outside the untouched test set. Record target-to-group membership, cross-split overlap checks, exclusions, and counts. The same frozen groups are used for resampling in Appendix E; groupwise inference does not substitute for this split.

For each target, group uses into mathematical families before splitting. The primary test chooses nonempty held-out families and assigns all their scored uses to H_x; Q_x contains scored uses from other families. Both sets are nonempty. No task from a held-out family, nor an equivalent task under another name, may occur in the revision execution closure $\overline{Q}_x$. If dependencies would expose one, enlarge the held-out family assignment or withhold the target from the primary split before evaluation. General prerequisites outside those families may be shared, and held-out uses may depend

on revision-family prerequisites rebuilt during testing. Thus $\overline{Q}_x$ and $\overline{H}_x$ may overlap while the held-out families remain unseen. Family labels describe mathematical constructions, not disjoint connected components of the DAG. Report any individual-use holdout within an already seen family separately; it supports within-family transfer only. The proposed generator-level FUG comparison would use family holdout, with the same assignments across methods and hint conditions.

During benchmark construction, a sandbox is prepared with the fixed prerequisite library, excluding the original target and downstream reference implementations. The candidate supplies the target statement. The downstream autoformalizer generates nodes in dependency order, receiving the current requirement and accepted prerequisites. Lean and the semantic checker inspect outputs separately. A prerequisite that fails its assigned requirements blocks dependent nodes; reference solutions cannot fill the gap. Revision executes $\overline{Q}_x$ and scores only Q_x. Held-out evaluation starts again from the frozen statement and executes $\overline{H}_x$, rebuilding all prerequisites, including any nodes also in $\overline{Q}_x$. It does not reuse declarations, proofs, or helpers generated during revision. Only H_x enters the compatibility numerator and denominator; prerequisite work counts toward evaluation cost. The candidate remains unchanged, and held-out outcomes never return to revision or selection.

Splits used in this paper. Panel A of Table 7 withholds descriptions by use ID: the generator sees ten (targets 1–4) or twelve (targets 5–8) fixed use descriptions per target, and the remaining 624 uses across sixteen generation trials are scored without their descriptions. Panel B returns feedback on uses 1–20 and scores uses 21–50 on seven targets with three evaluation repeats (630 attempts). Both splits lie within authored development targets, so related uses can occur on both sides.

C.3 Mathlib Refactor Pairs from a Fixed Snapshot

The construction in Figure 6 combines major Mathlib refactors with their downstream uses. The measured studies retain their own pinned environments: Experiment 016 uses Mathlib commit `81a5d257c8e410db227a6665ed08f64fea08e997` with Lean 4.32.0, while the later authored-use and source-rebuild studies use Mathlib commit `e3a0ab1c9aa1473c9fd343e77f00716bc6c23be9` with Lean 4.34.0-rc1. These snapshots are recorded separately; no single version is claimed for every study.

1. **Mine the full history.** Index every commit reachable from the fixed revision. Screen declaration changes for refactors, retaining before/after statement pairs that reorganize the same mathematical intent. Record the source commits, changed declarations, and selection reasons. Cleanup-only and feature-only changes do not supply design pairs; screening exclusions are logged without consulting downstream scores.

2. **Collect all consumers.** Map each refactored target to its counterpart in the reference snapshot, recording renames and replacements. Trace references in types, definitions, and proofs to collect all reachable direct and indirect consumers, without depth or node-count truncation. Preserve all source dependency edges, including merges across branches; review their role separately when constructing the task DAG. Public, source-linked declarations become tasks; generated or internal declarations remain traceable dependencies where needed. Record mapping and extraction failures separately.

3. **Build one bundle per target.** Store the common target, design pair, consumer tasks, dependency DAG, prerequisite context, and version identifiers. Review each task's mathematical requirement without prescribing the after version's Lean encoding. Classify edges as task-required or implementation-specific, record justifications, fix declaration/proof requirements, and check the reviewed task graph's feasibility before admission. Original code remains private reference material. Freeze whole refactor groups for development/test and whole use families for revision/held-out evaluation as specified in Appendix C.2.

4. **Evaluate on the common bundle.** Insert either before/after design or a generated candidate into the same isolated context. The fixed downstream autoformalizer, checker, and per-node budget rebuild the reviewed tasks in the declared dependency order. The full-bundle score counts completed tasks over all tasks admitted to that bundle; the target and context-only prerequisites are not scored. Failures and blocked tasks stay in the denominator. Report targeted and untargeted scores separately; only held-out scores enter FUG.

Before/after comparison uses the common toolchain, not each design's original environment. Port names or syntax without changing assumptions, operations, or laws; record portability failures separately. Imports and retrieval exclude original target and consumer implementations, so evaluation must rebuild the uses rather than import their solutions.

The extraction code distinguishes the full reverse-reachability pool from an optional smaller task selection. Its shortest-path selection is neither the full consumer DAG nor a complete prerequisite closure. Any reduced evaluation must retain the full source bundle and report its selection rule and coverage. Selection alone does not establish that the retained edges are mathematically required; C-DATA and C-SPLIT must include the edge review and group/family split records. Candidate catalogs and machine visibility estimates do not replace reviewed manifests and frozen splits.

Inventory scale. History mining yields 13,905 source bundles; screening retains 378 semantic refactor pairs from 2022–2026 with 4,172 drafted use requirements. Across these pairs, reachable downstream declarations total 3,416,467, with a median of 573 per pair (interquartile range 105–4,612); 43% of pairs reach at least 1,000 declarations, and the largest reaches 187,402. The longest dependency chain has median depth 14 (90th percentile 31, maximum 46), and 194 of the 378 pairs link to a public design discussion. These are static footprints; the pairs have not been evaluated as test targets.

D Does the Evaluation Measure Design Value?

D.1 Semantic Checks and Independent Calibration

Upstream and downstream statements receive independent semantic checks against their own requirements. The model, prompt, and decision rule are fixed before evaluation. Checks cover changed objects, quantifiers, assumptions, conclusions, and vacuous replacements. A semantic repair is an initially rejected draft whose final statement passes the same check without changing the target. Use feedback may motivate revision, but use success is not an equivalence certificate. Post-freeze BEq+, stronger provers, executable contracts where applicable, and blind expert review provide independent checks on model judgments. An equivalence certificate must have complete proof dependencies and cannot rely on unproved candidate laws. No certificate means unresolved, not semantically incorrect. Disagreements are recorded separately. Calibration outcomes do not return upstream or justify replacing the checker after seeing results.

Archived calibration. In the FormalTCS archive, fixed BEq+ certifies equivalence for 3/175 Direct and 7/175 predicted-use statements; bidirectional proof search then certifies 12 of 30 unresolved candidates across six source targets (Appendix F.2). A missing certificate therefore marks unresolved equivalence rather than a semantic failure.

D.2 Dependency and Major-Refactor Controls

Dependency checks trace completed node types and proofs back to the candidate; node review checks that the reference has a mathematical role. Compilable vacuous statements such as `True` serve as negative controls for counting bypass solutions as reuse. A strong-law control adds, as an open theorem or a structure field, a law strong enough to discharge the held-out uses directly, while leaving the target recoverable; it should raise ordinary FUG but not proved FUG, and a gap between the two on generated statements is reported per method. Expert-solved uses measure how many known-solvable tasks the fixed downstream autoformalizer completes. Controlled design pairs satisfy the same current requirement but differ in later completion or adaptation work. Both outcomes and burden are recorded, rather than using code length or proof-search difficulty alone to represent design value. Import, retrieval, and feedback audits check access to original target code, downstream reference solutions, and unauthorized use information.

Measured control. On ten FLT3 targets from RLM25, reference declarations support 26/26 direct and 222/222 transitive library uses, whereas compilable `True` replacements support 0/26 and 0/222 (Appendix F.2). The strong-law control has not been run.

Measured comparison on five major Mathlib refactors. Table 3 reports Experiment 016, completed on 20 September 2026. It includes all five manually assembled records in dataset release

v0.1.0; no pair was deferred after the portability checks. Each pair has four manually specified uses with source commits and declarations recorded before scoring. The ten roots are ports of refactor designs into Mathlib v4.32.0 at commit `81a5d257c8e410db227a6665ed08f64fea08e997`, with Lean v4.32.0. These selected uses calibrate the score; they are not the complete downstream DAGs specified for the broader benchmark in Appendix C.3.

Each fixed design is evaluated in three trials. The downstream autoformalizer is `gpt-5.6-luna` with low reasoning and a 150,000-token budget per four-use run; the independent semantic checker is `gpt-5.6-sol` with high reasoning. Before/after labels are replaced by candidate identifiers, and the same uses, prompts, and settings apply to both designs. Prompts retain original declaration names, so this is the original-description condition. Consumers must compile without admissions, satisfy import and workspace checks, substantively depend on the supplied root, and preserve the use's objects, assumptions, and conclusion. The solver and checker run without network access. We evaluate downstream formalization on fixed refactor roots.

The 30 completed runs contain 120 downstream use evaluations: 29 of 60 pass before and 60 of 60 pass after. In matched use–trial comparisons, 31 pass only after and 29 pass under both designs; none pass only before. Target averages give 48.33% before and 100% after, with a paired gain of 51.67 percentage points. Resampling the five target pairs 10,000 times gives a paired-target bootstrap 95% interval of $[16.67, 86.67]$ percentage points. One budget-exhausted before run on bilinear forms was excluded and resampled under the same configuration: its three partial passes were not scored, and the replacement completed two uses. The reported mean is over completed runs under this resampling rule. An earlier sandbox initialization failure started no model calls. This comparison covers these five manually assembled pairs and does not estimate performance across the full structural-pair catalog or supply the expert-reference score for the separate generated-statement comparison.

Extended refactor comparison separates representation refactors from changes that expand capabilities or alter behavior. Across three completed batches, adopted designs win, tie, or lose depending on the target and execution setting; the score is not simply a label for the later revision. Appendix G.5 reports all targets in these batches, including controls and reversals.

D.3 Alternative Design Criteria

Use-specific evidence connects a design judgment to the mathematics it predicts. We examine this connection in an eleven-pair major-refactor comparison with controls: a judge first sees the shared design context, then receives use requirements, and finally sees code produced while executing those uses. Each request returns a general abstraction-quality score on a 0–100 scale and a predicted use-coverage fraction. The comparison measures how additional use evidence changes alignment with downstream compatibility, while keeping design choice and benefit estimation separate.

Evaluator comparison on major refactors and controls. The eleven pairs comprise four same-intent refactors and seven capability, abstraction, or behavior-change controls (Table 10). Two downstream trials per design give 44 jobs; 91 primary uses contribute 182 attempts per arm, with three construction-only uses scored separately. Compatibility averages dependency-closed completion over trials within each target, then targets equally. The mixed-group mean is 71.24% before and 88.29% after. The four same-intent pairs move from 91.75% to 92.79%; the seven controls move from 59.52% to 85.71%. This grouping separates near-preserving representation changes from expanded capabilities. The near-tie on same-intent pairs makes adaptation work a complementary design signal: two statements can reach the same uses while one forces fewer conversions, structure packages, or bridge proofs. The complete evaluator release adds the last pair's three judgments to the earlier ten-pair release, reusing its 30 requests. The resulting 33 requests each score both candidate designs.

The run configuration uses `gpt-6-luna`/high for downstream generation, a 1,000,000-token cap per job and a 120-minute timeout, with `gpt-6-sol`/high for semantic checking. The proxy judge also uses `gpt-6-sol`/high. Mathlib and Lean are v4.32.0 at Mathlib commit `81a5d257`. Each judge mode balances candidate labels across targets (six after designs labeled A, five labeled B), with one request per pair and mode; execution scores and check outcomes remain hidden. The three modes progressively add use requirements and execution traces, while the formal checker uses fixed mathematical-meaning records and accepts appropriate changes of representation. Thus the comparison isolates

Table 10: The eleven-pair evaluator comparison. Uses counts primary tasks; completion is in percent, averaged over two trials per design.

Refactor or design change	Uses	Before	After
Same-intent refactors			
Restricted power-series representation	10	100	100
Complete-lattice axiom packaging	12	83.33	87.50
Uniform-integrability definition	25	92	92
Cartan subalgebra as a parameter	24	91.67	91.67
Capability, abstraction, and behavior controls			
Semilinear submodule-quotient equivalence	3	66.67	100
Jordan–Hölder relation abstraction	6	75	100
Generalized completion inclusion	3	50	100
Stopped σ-algebra correction	2	75	100
Bilinear forms as nested linear maps	2	50	50
Separation-space abstraction	2	100	100
Semilinear integral composition	2	0	50

the information supplied by concrete use evidence; the final compatibility score remains an execution result.

Rank agreement and selection. Table 11 reports pooled correlations over 22 designs and correlations between eleven after-minus-before changes. Kendall's τ_b is recomputed from the unrounded raw judgments, with pairs tied in both variables excluded from the single-variable tie terms; Spearman's ρ is unchanged. Source-size signals prefer fewer raw bytes, lexical tokens, or declarations counted by a regular expression. They include source scaffolding and are surface-size proxies. Selection loss is the target-equal compatibility difference between the empirically best design and the design chosen by the signal, averaging tied choices.

Table 11: Design judgments versus measured compatibility. Absolute correlations pool 22 designs; gain correlations use eleven paired changes. Loss is in percentage points.

	Absolute		Gain		Selection
Signal	τ_b	ρ	τ_b	ρ	**loss**
Shorter source (bytes)	0.142	0.113	−0.432	−0.589	14.39
Fewer lexical tokens	0.132	0.178	−0.381	−0.464	14.77
Fewer declarations	0.117	0.146	−0.431	−0.471	11.93
Target: quality	0.228	0.275	−0.208	−0.281	0
Target: coverage	0.283	0.330	−0.021	0.029	0
Descriptions: quality	0.172	0.210	−0.126	−0.195	0
Descriptions: coverage	0.335	0.386	0.208	0.252	0
Implementations: quality	0.206	0.281	0.145	0.245	0
Implementations: coverage	0.414	0.502	0.391	0.517	0

Predicted coverage tracks compatibility more closely than general quality in all three conditions. Its gain-rank agreement rises from -0.021 to 0.208 and 0.391 as use information is added. Every judge setting selects the after design on the six non-tie pairs, giving zero observed selection loss in this cohort. Choice of direction is therefore easier than estimating the size of a design's future benefit: the gain ranking changes sharply as execution evidence is added, while selection remains unchanged. Absolute coverage errors also distinguish rank agreement from numerical calibration: mean absolute error is 21.70, 25.98, and 21.08 percentage points across the three conditions. On the original eight targets, excluding the three added provenance controls, gain τ_b is 0.231, 0.386, and 0.386. In a paired-target bootstrap with 5,000 samples, the implementation-minus-description difference has a 95% interval of $[-0.059, 0.249]$ for absolute τ_b and $[-0.203, 0.616]$ for gain τ_b. These intervals describe uncertainty in the observed ordering; they retain the two arms and all judge conditions together within a target.

The power-series pair makes the information effect concrete. One design states coefficient decay directly; the other specializes a multivariate definition. Both complete all ten uses in both trials. Predicted coverage of the direct definition rises from 58% to 82% with requirements and 96% with implementations; the corresponding predictions for the multivariate design are 91%, 99%, and 99%. Seeing what the direct definition actually supports corrects a large initial underestimate. The Cartan pair illustrates a different issue: the judge predicts 8% versus 99% after seeing implementations, while both designs complete 22 of 24 uses in each trial. Its explanation favors direct agreement with the reference's parameterized syntax; the execution gate explicitly permits the same subalgebra to be stored or parameterized. This contrast identifies the importance of a shared mathematical acceptance contract when comparing inspection with execution.

Selection among generated designs. The complementary multi-candidate experiment asks whether inexpensive criteria select generated abstractions without sacrificing held-out compatibility. Compare accepted final statements from the prescribed methods and runs within each target, using the same library, held-out uses, downstream autoformalizer, and allowance. Fix inclusion from acceptance and record completeness before inspecting scores. Merge byte-identical modules and average comparable repeated evaluations. Report target, refactor-group, distinct-design, and evaluation-repeat counts. The response is the statement's held-out compatibility $C_D(s)$; complete, comparable records from the primary evaluation supply these scores.

The proxies are source length excluding comments and whitespace, explicit declaration count, and a blinded LLM ranking. Shorter code and fewer declarations are the preferred directions for the two size measures. The judge sees only the target, common library context, and candidate code; method names, refactor direction, revision traces, held-out requirements, and execution scores are hidden. Fix the judge, prompt, repetitions, and order counterbalancing before measurement, allow ties, and archive judgments before joining them to compatibility. Record judging cost alongside the cost of downstream evaluation.

For targets with at least three distinct designs, measure within-target Spearman rank agreement and the compatibility lost by choosing the proxy's top-ranked candidate rather than the candidate with the highest observed mean compatibility. Average over tied proxy choices. Fix a practically meaningful loss threshold on development targets before test scoring. Report each proxy's rank agreement and selection loss with target-equal summaries and refactor-group intervals (Appendix E). Constant predictor or response values have undefined correlations; report their counts. Compilation and semantic acceptance are constant in this subset and cannot rank its designs. Low rank agreement together with material selection loss supports the need for execution-based selection: the proxy misses consequential differences among faithful abstractions. High agreement and negligible loss would instead challenge that need by providing an effective cheaper selector. Intervals spanning the loss threshold leave that comparison unresolved.

Measured on generated designs. Run C evaluates each of its 64 Direct and Visible statements (eight per target) on the same 50 uses regardless of the upstream gate, which permits a within-target comparison. Against ordinary completion, fewer declarations, fewer lexical tokens, and shorter source have mean Spearman correlations of -0.51, -0.43, and -0.35, negative in all eight targets. After source screening removes dependence on unproved declarations, these become -0.17, -0.09, and -0.16, and choosing each proxy's top design loses 10.3–11.9 of 50 uses, against 9.1 for a random choice. The model's upstream semantic score has a mean correlation of $+0.11$ with screened completion, and statements that fail the gate complete 14.2 screened uses on average, against 11.5 for statements that pass. Size proxies therefore give no useful ranking among these generated designs, and the binary gate does not predict downstream support.

E Protocol for the Complete Evaluation (Not Run)

Section 5 reports component studies, each with its own settings. The exposure and repair study uses GPT-5.4/high for generation with GPT-5.6-sol/high checks, two generation trials per target, and up to three repair rounds; its upstream statement stays fixed during downstream repair. Runs A–C, the repeated repair runs, the held-out proof study, and real-library replay retain the configurations recorded in Appendix G. None of these runs executes Algorithm 1 end to end.

The complete generator-level evaluation, which we specify but have not run, crosses use descriptions with execution feedback. Direct, Descriptions-only, Feedback-only FUSE, and Full FUSE share the upstream model, generalization prompt, semantic-feedback setting, and budget ceilings, and conditions with the same description access share their initial drafts. Matched self-refinement receives the same permitted records, revision scope, selector, and budget as Full FUSE but omits the structured evidence-to-revision step. A Selection-only control draws candidates without execution feedback, seals the pool, and applies the same selector, separating selection from revision; a successful-work ablation removes the review of completed traversals. Final statements, together with Mathlib's adopted design as an expert reference, are evaluated on whole held-out use families of untouched targets with the same downstream autoformalizer, and again with a second downstream model D'. A budget-sensitivity diagnostic re-evaluates fixed final statements at 0.5 to 4 times the per-use allowance. FUG and proved FUG average trials within targets and weight targets equally, with at least three seeds per target and 95% intervals from resampling groups of related refactor targets (Appendix C.2). Historical controls would add API-neutral rewrites of the targeted uses and the untargeted uses of each refactor bundle.

F Design and Evaluation Diagnostics

The development studies below examine three links in the argument: local checks versus actual use, representation choices versus mathematical reach, and execution evidence versus upstream revision. Each retains its original evaluation setting; the unified FUSE comparison uses the protocol in Appendix E.

F.1 Use Visibility and Downstream Repair with a Frozen Interface

Experiment 014 has eight cases, two trials per condition, and 50 uses per trial. In its original terminology, Direct hides the use descriptions and Visible supplies them. Binary and Detailed start from the Visible interface and repair only downstream files; the interface stays frozen. They are repair conditions, not full FUSE.

The original pool mixes uses available to the pipeline with uses held out under that protocol. These groups do not match the new Q_x/H_x split. The first numeric column reports the upstream interface gate: the statement must compile in Lean and pass a model check against the task's interface requirements, including reference conventions such as argument order. The last column reports the original all-use end-to-end (E2E) metric; a trial that fails the gate contributes zero.

Table 12: Archived partial measurements under their original protocol.

	Trials	**Completed uses**	
Condition	Passed original gate	In passed trials	Across all trials
Direct	4/16	155/200	155/800
Visible	9/16	349/450	349/800
Binary repair	9/16	418/450	418/800
Detailed repair	9/16	431/450	431/800

For the 400 uses in Cases 1–4 (10 descriptions), the completed counts are 42 for Direct, 171 for Visible, 199 for Binary, and 200 for Detailed. For Cases 5–8 (12 descriptions), the counts are 113, 178, 219, and 231 in the same order. See Appendix I for reproduction materials.

How far does information about a use transfer? A separate development comparison uses eight author-designed targets in Mathlib, with 50 uses per target and two trials. Each design prompt exposes 12 uses: one configuration spans shared intermediate constructions, while the other concentrates on a local chain; the two match the annotated difficulty distribution. The remaining 38 uses are classified as same-chain or structurally distant before the run. Table 13 compares one-shot completion with and without descriptions, before downstream repair. Both configurations reuse the same no-description candidates, scored on their respective hidden-use sets. Every planned use remains in the denominator, including uses blocked by an unacceptable upstream design. The solver

is `gpt-5.4`/high, the checker `gpt-5.6-sol`/high, and Mathlib/Lean are v4.32.0. Some spanning selections were informed by earlier failure analysis; these are development diagnostics, separate from the history-derived FutureUseBench split.

Table 13: Use-description transfer: completed hidden uses; Δ in percentage points.

Visible uses	Hidden uses	No descriptions	Descriptions	Δ
Spanning constructions	Same chain	95/490	129/490	+6.94
Spanning constructions	Structurally distant	16/118	17/118	+0.85
Local chain	Same chain	44/194	74/194	+15.46
Local chain	Structurally distant	66/414	89/414	+5.56

Summing the two hidden strata gives 111/608 versus 146/608 for spanning constructions, and 110/608 versus 163/608 for the local-chain configuration. In both configurations, the gain is larger within the same chain, by 6.09 and 9.90 percentage points respectively. These are declaration results before consumer repair, and the two configurations reuse development targets and the main comparison's Direct candidates. The bridge panel was informed by earlier failure analysis, so these are dependent development comparisons.

An earlier dependency analysis conditions on five trials where both Direct and Visible yield accepted interfaces. Hidden declaration completion increases from 144/200 to 165/200 (+10.50 percentage points). The case totals are 66/80 to 60/80 for group generators, 25/40 to 37/40 for spectral norms, and 53/80 to 68/80 for pushforward measures. In the spectral case, a shared spectral-radius-limit construction supports seven hidden uses that all change from failure to success. This post-hoc common-accepted subset exposes a mechanism beyond opening the interface gate, while the group case retains a regression; it does not replace the full-cohort comparison. The archived dependency analysis follows theorem types and definitions, excluding proof bodies. Together, these observations motivate testing across use families: support for nearby consequences and support for a different mathematical construction are different generalization demands.

F.2 Semantic and Use Evaluation

Semantic equivalence and downstream utility answer complementary questions. FormalTCS compares the original Direct prompt with a Predicted-use prompt that adds predicted downstream uses. Across 175 targets, fixed BEq+ finds 3 equivalence certificates for Direct and 7 for Predicted-use. There are 167 targets with parseable outputs in both conditions: four are certified only for Predicted-use, three for both, and 160 for neither. Each condition has four parse errors, retained as uncertified in the full-target rate. The SDK pipeline-completion counts include infrastructure retries; the BEq+ jobs refer to the candidates frozen for that evaluation. Beyond Compilation checks meaning on ordinary mathematics prompts without actual downstream uses.

A separate four-domain archive contains 568 primary declaration attempts: 539 elaborate and 305 pass model-assisted semantic checks including prerequisites. Its 33-record interrupted feedback run remains excluded under the archive's primary-analysis rule. Domain environments and exposure settings differ; these are archive counts rather than one benchmark rate. In the vector-bundle subset, all 80 declarations elaborate but only five pass the semantic dependency check. The archived judgments used GPT-5.4/high and were marked for later review; they diagnose a gap between compilation and mathematical acceptance without certifying the labels.

RLM25 is the real-world Lean benchmark introduced with BEq+ (Poiroux et al., 2025). A development check contrasts reference declarations, which complete $26/26$ direct uses and all 222 downstream uses, with a compilable `True` replacement, which completes none. This checks whether the evaluator distinguishes a usable declaration from a vacuous one. In the last two table rows, the middle columns pool examples; summaries weight prompt clusters or targets equally and give 95% confidence intervals in percentage points (pp).

Table 14: Prompt-level development comparisons, preserving each study's outcome measure.

Study / measure	Sample	Direct	Predicted-use	Summary
FormalTCS / pipeline	175 targets	165/175	170/175	$+5$ completions
FormalTCS / BEq+	175 targets	3/175	7/175	Four additional certificates
FormalTCS / ill-typed	175 targets	4	2	Two fewer outputs
Beyond Compilation	400 rows / 362 clusters	389/400	393/400	Cluster-weighted $+0.87$ pp; CI $[-0.95, 2.76]$
RLM25 / direct-use edges	354 uses / 100 targets	177/354	200/354	Target-weighted $+0.34$ pp; CI $[-5.50, 6.20]$

Table 15: Real-library replay on 100 targets. Pooled counts and target-equal rates retain distinct aggregation units; intervals are in percentage points.

Measure	Description	Direct	Use planning	Change [95% CI]
Direct uses	Pooled immediate dependencies	177/354	200/354	+23
All descendants	Pooled transitive dependencies	1570/3724	1710/3724	+140
Direct uses (%)	Equal target weights	58.38	58.72	+0.34 $[-5.50, 6.20]$
All descendants (%)	Equal target weights	56.10	57.04	+0.95 $[-4.93, 6.91]$

The more revealing evaluator result is what stronger search recovers. From 30 candidates left unresolved by fixed BEq+ (15 per condition, 16 source targets), bidirectional proof search certifies 12, with six in each condition. These 12 candidates span six source targets. For each implication, the prover receives the other statement as a premise and gets eight attempts; proof checks enforce the frozen statement, complete proofs, and the permitted axiom dependencies. The 480 premise-assisted attempts produce 202 checked proofs. A separate one-attempt-per-direction control without the premise produces four proofs in 60 attempts. The candidate-level result is $12/30$ newly certified and $18/30$ unresolved: a concrete demonstration that proof-search failure is not a semantic verdict. This motivates recording unresolved equivalence separately from rejected meaning when evaluating abstraction design.

F.3 Local Compilation versus Source Rebuilds

The source-rebuild diagnostic compares generation with no use descriptions (None), with the target's own use descriptions (Same target), and with descriptions reassigned from other targets in the same abstraction-decision category (Other targets). Reassignment orders targets by description length and shifts them cyclically so no target retains its own descriptions. Four paired seeds on 14 targets yield 56 runs per condition. For each target and seed, the first two prompt messages are identical across conditions; a third message supplies the condition-specific use descriptions. Generation and rubric scoring use Kimi-K3; semantic natural-language inference (NLI) uses Opus-5. The environment is Mathlib commit `e3a0ab1c9aa1473c9fd343e77f00716bc6c23be9` with Lean 4.34.0-rc1. The local checker compiles a candidate in a clipped task environment. The source-rebuild check reinserts it into the original target module and builds every eligible direct-use module supported by the reference, including declarations omitted by the local checker. A run passes only if the target and all these modules compile. This is an all-or-nothing check of existing source, whereas C_D in Definition 1 measures the fraction of held-out uses freshly formalized by D. No downstream autoformalizer generates uses in this diagnostic, and the rebuild counts are not FUG.

Same-target descriptions yield more local compilations, more NLI acceptances, and higher rubric scores than no descriptions, yet fewer successful source rebuilds. The paired rebuild-success difference is -5.36 pp (target-cluster bootstrap 95% CI $[-17.86, 3.57]$), with three wins, 47 ties, and six losses. At four samples per target, both conditions have at least one successful rebuild on $4/14$

Table 16: Local compilation, semantic review, and source rebuilds for the same candidates. Rubric means use valid scores.

Outcome	None	Same target	Other targets
Local target compiles	25/56	28/56	29/56
Full target module compiles	12/56	10/56	12/56
Target and all direct uses compile	10/56	7/56	10/56
Completed direct-use checks	309/1672	285/1672	260/1672
All target NLI checks pass	5/56	7/56	5/56
Mean rubric score	89.90	92.46	91.59

targets, versus $3/14$ with other-target descriptions. The useful distinction is between predicted usefulness and compatibility with existing source: a richer description or more favorable judgment does not ensure that the resulting design fits its mathematical environment. Rebuilding that environment exposes the missing obligations.

F.4 Generated Designs and Mathematical Reach

The convex-combination study isolates an interpretable design choice: whether the scalar type is fixed or remains a parameter. Both generated roots compile without proof placeholders and pass the study's root semantic review. Direct represents weights using nonnegative reals and connects averaging to real modules. The use-informed candidate parameterizes weights, the averaging operation, and its laws by a scalar type R, with a module realization under the recorded ordered-semiring assumptions. The following excerpts show the weight representations; ambient namespaces are omitted.

Direct: fixed scalar

```
structure ConvexWeights (ι : Type*) where
  toFinsupp : ι →₀ NNReal
  sum_eq_one :
    toFinsupp.sum (fun _ w => w) = 1
```

Use-informed: scalar parameter

```
structure ConvexWeights (R : Type*)
    (ι : Type*) [Semiring R] [PartialOrder R]
    [IsOrderedRing R] [Fintype ι] where
  toFun : ι → R
  nonneg : ∀ i, 0 ≤ toFun i
  sum_eq_one : ∑ i, toFun i = 1
```

The distinction becomes visible at MD01, which asks for convex combinations in a module over a general ordered semiring. The Direct candidate's `ConvexWeights` fixes the wrong level of generality for this task. Its failure blocks four successors: MD02, MD03, SM01, and FN03. The use-informed design supports all five; both designs support the other 20 visible uses. Thus a single upstream choice accounts for the $20/25$ versus $25/25$ statement-stage outcome. The comparison uses one candidate from each recorded pipeline: Direct from the newer semantic loop and the use-informed candidate from an earlier run followed by semantic repair. Downstream theorem statements permit proof placeholders; their mathematical acceptance is the outcome here. A stricter surface-dependency check scores Direct $18/25$, flagging SX01 and FN02 for how they use predecessors. These are recorded separately from the five-use scalar branch.

Table 17: Where generated design choices matter. Convexity reports semantic statement acceptance on visible uses; bilinear forms report elaboration on newly evaluated, non-visible uses.

Uses / outcome	Direct	Use-informed
Convexity: general-scalar branch	0/5	5/5
Convexity: remaining visible uses	20/20	20/20
Bilinear forms: all non-visible uses	46/50	48/50
Bilinear forms: contract-corrected subset	46/47	47/47

A bilinear-form pilot extends the question to uses absent from the design prompt. Each candidate is tested on 50 visible and 50 non-visible uses; the latter receive fresh downstream runs with the same model, prompt, environment, and one attempt per use. Elaboration succeeds on $46/50$ non-visible uses for Direct and $48/50$ for the use-informed candidate. Removing three recorded task-contract defects gives $46/47$ and $47/47$, respectively. This is a statement-stage evaluation with proof place-

holders allowed. Together, the two cases make design reach tangible: parameter choices govern an entire branch of mathematics, and supporting more observed uses can also extend the set of expressible unobserved statements.

F.5 Execution Feedback Changes Upstream Structure

The simplicial-complex case (item115, paired v6) shows how expanding the executed context changes the design itself. Both candidates compile in the target environment without added axioms or proof placeholders. The local-check condition stores the faces and their closure property directly. The compatibility-feedback condition instead inherits the existing pre-abstract complex structure, retaining affine independence and the intersection condition as additional fields (omitted below). The decisive signal is explicit: at turn 14, the target probe reports the missing parent projection; at turn 16, the revised declaration supplies it through inheritance.

Local-check candidate

```
structure SimplicialComplex where
  faces : Set (Finset E)
  isRelLowerSet_faces : IsRelLowerSet faces fun s ↦ s.Nonempty
```

Compatibility-feedback candidate

```
structure SimplicialComplex extends PreAbstractSimplicialComplex E where
```

The feedback candidate makes six check calls (five failures, then a pass); the control makes one successful local call. The prompt-invariance audit confirms the paired initial task. In fixed-source replay, the control passes the complete target modules but stops at the API probe, before downstream compilation. The feedback candidate passes the complete compatibility check covering 38 direct uses. The probe names the reference projection `toPreAbstractSimplicialComplex`, so this is a test of recovering a required library connection through API feedback. Its mechanism is broader than proof repair: execution identifies which shared structural relationship the upstream representation must expose.

A complementary case turns a downstream obligation into a shared mathematical object. The task is to transport a group presentation along an isomorphism: generators build elements, while relations identify equal words. Both conditions start from the same indexed-generator statement, downstream files, and binary feedback. One repairs only downstream files and helpers; the other may also revise the statement. The fixed-interface answer compiles but transports only generators. Joint revision retains that interface and adds `IndexedPresentation`, packaging generators, relators, and equality between their normal closure and the free-group map's kernel. This condition makes the relations complete, not just valid. The excerpts omit namespaces and comments.

Shared presentation structure

```
structure IndexedPresentation (G : Type*) [Group G] (ι : Type*) where
  generators : IndexedGroupGenerators G ι
  relators : Set (FreeGroup ι)
  normalClosure_relators : Subgroup.normalClosure relators = generators.freeGroupLift.ker
```

Downstream transport

```
theorem consumer19 {G H ι : Type*} [Group G] [Group H]
    (P : IndexedPresentation G ι) (e : G ≃* H) :
    Nonempty (PresentedGroup P.relators ≃* H) :=
  ⟨P.presentedGroupEquiv.trans e⟩
```

The new `presentedGroupEquiv` constructs the isomorphism from the quotient by the declared relations to the original group, using the kernel equality and the generator map's surjectivity. It does not take that isomorphism as a field:

```
noncomputable def presentedGroupEquiv (P : IndexedPresentation G ι) :
    PresentedGroup P.relators ≃* G :=
  (QuotientGroup.quotientMulEquivOfEq P.normalClosure_relators).trans
    (QuotientGroup.quotientKerEquivOfSurjective P.freeGroupLift P.generators.freeGroupLift_surjective)
```

The downstream proof composes this equivalence with the supplied group isomorphism, showing that the same relations present the target group. Both final answers compile and pass the recorded proof-admission and substantive-use checks; only the joint revision completes the presentation task. The archived roots also compile and contain no proof placeholders. This case repairs an observed

use by changing both the statement and its downstream proof; evaluation replays those source files. Its contribution is the concrete chain from a missing relation obligation to a shared structure and an actual call, complementing the simplicial case's restoration of an existing library connection.

Success feedback as a reusable proof bridge. Experiment 030 fixes the Case01/Trial03 Visible source and targets Consumer15 with four independent trajectories: ordinary `fuse` or `fuse_success`, each under binary and detailed feedback. The baseline is 29/50 and every trajectory finishes at 30/50; the binary–detailed difference is zero in both arms. The mechanism is visible in the archived Lean sources. `Consumer15_fuse.lean` reaches the quotient equivalence through `P.presents`, while `Consumer15_fuse_success.lean` explicitly consumes the successful predecessor theorem `P.normalClosure_relators_eq_ker` before composing the same right-inverse equivalence. The named bridge already exists in the initial Root, proved by symmetry of the same presentation equation. Both conditions have access to the Root and predecessor sources; the success condition additionally receives an explicit feedback block. The final code therefore records alternative uses of an existing bridge, with zero observed treatment gain; it does not establish that the extra feedback caused the choice. The run has one target Consumer and `dependency_gate_audit=false`, so it is a mechanism case rather than a population-level estimate of success-feedback gains; neither source is counted in the operational ladder.

G Unified Experimental Results

G.1 Per-Target Results and Breakdowns

The measured studies answer complementary questions. Table 5 tests use exposure and declaration repair; Table 6 records shared-statement revision; Table 15 checks existing dependencies in real libraries. Major Mathlib refactor comparisons use fixed human designs. These studies have different models, edit permissions, and proof requirements, and are not pooled as one method ranking. Concept-discovery counts and the all-successful ablation remain extensions specified in Appendix E.

Source screening marks a Root or helper declaration as unproved when its body contains `sorry` or names another marked declaration, and removes each completed use whose source names a marked declaration; short generic names are excluded from matching. In Run C, ordinary counts for Direct, Visible, FUSE-C, and FUSE are 163, 256, 356, and 301 out of 1600, while screened counts are 163, 171, 240, and 245. Implicit instance dependencies may be missed. Among the twelve trials with accepted statements in both revision arms, screened completion is 206 versus 245, with eight wins, three losses, and one tie; this conditional subset does not replace the complete comparison.

G.2 Operational Ladder: Per-Case Results

Table 18 expands the operational ladder in Section 5.2. Each entry is the statement-level end-to-end completion count out of the fixed 100 consumer opportunities for that case (two trials with 50 consumers); the parenthesized value is the number of API-passing trials out of two. API failures remain zero in the E2E denominator. The feedback rows are continuations of the frozen Visible candidates, so this table is a workflow decomposition rather than four independent random arms.

Table 18: Per-case operational results. Entries are strict E2E completions per 100 opportunities, followed by API-passing trials out of two.

Case	Direct	Visible	Binary	Detailed
Indexed group generators	0 (0/2)	88 (2/2)	99 (2/2)	100 (2/2)
Row-echelon certificates	0 (0/2)	0 (0/2)	0 (0/2)	0 (0/2)
Canonical spectral norm	0 (0/2)	0 (0/2)	0 (0/2)	0 (0/2)
Total pushforward measures	42 (1/2)	83 (2/2)	100 (2/2)	100 (2/2)
Continuous linear representations	71 (2/2)	71 (2/2)	89 (2/2)	94 (2/2)
Abstract phase-space tiles	0 (0/2)	0 (0/2)	0 (0/2)	0 (0/2)
Finite conditional Ruzsa distance	0 (0/2)	65 (2/2)	82 (2/2)	87 (2/2)
Structured Bohr data	42 (1/2)	42 (1/2)	48 (1/2)	50 (1/2)
All cases	**155 (4/16)**	**349 (9/16)**	**418 (9/16)**	**431 (9/16)**

The visible-use jump is distributed across the API-viable cases rather than concentrated in one case. Binary feedback adds 11, 17, 18, 17, and 6 completions in Cases 1, 4, 5, 7, and 8, respectively; detailed feedback adds 1, 5, 5, and 2 further completions in Cases 1, 5, 7, and 8. Cases 2, 3, and 6 remain shared API bottlenecks in this run. These per-case values preserve the fixed denominator and make the scope of the operational claim explicit; they do not replace the held-out, cross-target FUG analysis.

Visible generation uses 6,593,702 tokens; binary use repair adds 2,878,113 and detailed use repair adds 2,375,515, giving effective totals of 9,471,815 and 8,969,217. Detailed repair uses 502,598 fewer incremental tokens (17.46%) and 28:31 less cumulative call time; including the shared seed, token savings are 5.31%. Relative to Visible, binary adds 69 completions and detailed adds 82. Among the 450 uses under accepted Visible statements, failures fall from $450 - 349 = 101$ to $450 - 431 = 19$, a net reduction of $82/101 = 81.19\%$. The $431/155 = 2.78$ ratio compares workflow completions under different total costs.

The aggregate joins two source groups: Cases 1–4 use canonical-10 exposure, and Cases 5–8 coherent-12. Models, checkers, repeat counts, blocking rules, and library revision agree; description counts and composition differ. The aggregate is descriptive, not a single-exposure benchmark score.

Table 19: Source groups within the declaration study. All entries retain 400 opportunities per group.

Source group	Direct	Visible	Binary repair	Detailed repair
Cases 1–4, ten descriptions	42	171	199	200
Cases 5–8, twelve descriptions	113	178	219	231

Uses withheld from initial generation. For the initial Direct/Visible comparison only, we apply the Visible exposure split to both conditions: ten described uses per target in Cases 1–4 and twelve in Cases 5–8. Over two trials, the remaining uses provide 624 opportunities. Direct completes 112/624 and Visible 255/624; the common visible-ID set gives 43/176 and 94/176. Conditional on interface acceptance, hidden completion is 112/154 (72.7%) versus 255/350 (72.9%). Counts are reconstructed from per-use repeat records, counting twice-passed uses twice and once-passed uses once. These are declarations before consumer repair, not the final feedback branches on an untouched proof test. The interface gate also checks reference conventions; some exposed descriptions reveal those conventions, so the result includes improved alignment with the required interface.

The broader comparison with matched self-refinement, an expert reference, and a second evaluator remains unmeasured; the declaration counts above are completed archived measurements.

Paired uncertainty and unseen-declaration difficulty. We recompute percentile bootstrap intervals from archived target outcomes using 50,000 resamples and seed 20260926. Each target retains both generation trials, all dependent uses, and all paired methods. The primary analysis resamples four targets within each original exposure stratum (canonical-10 and coherent-12), preserving the study's four-plus-four composition and recomputing pooled completion rates. Targets blocked by an upstream gate failure remain included. These post-hoc intervals describe resampling sensitivity on eight authored development targets, not a randomly sampled population of new targets; neither the 800 total opportunities nor the 624 unseen opportunities are treated as independent samples.

Table 20: Paired changes in declaration completion (percentage points). Intervals resample targets within the original exposure strata.

Setting	Comparison	Change	95% interval
All uses	Visible − Direct	+24.25	[5.13, 48.50]
Observed-use repair	Binary − Visible	+8.63	[3.63, 13.75]
Observed-use repair	Detailed − Visible	+10.25	[4.13, 16.25]
Combined workflow	Detailed − Direct	+34.50	[10.13, 62.50]
Observed-use repair	Detailed − Binary	+1.63	[0.50, 2.63]
Unseen before repair	Visible − Direct	+22.92	[4.81, 45.67]

Without exposure stratification, the target-bootstrap intervals are [9.25, 62.50] for Detailed minus Direct on all uses and [4.72, 46.50] for Visible minus Direct on unseen uses. The primary stratified intervals for individual all-use rates are Direct [5.25, 37.13], Visible [18.63, 67.75], Binary [22.25, 81.25], and Detailed [23.38, 83.63] (%). Before repair, the unseen-use intervals are [4.97, 33.65] for Direct and [16.99, 63.94] for Visible. The paired differences preserve covariance across conditions.

Table 21: Unseen declarations before repair, by target and authored difficulty. Counts include both generation trials.

Target / difficulty	**Direct**	**Visible**
Indexed group generators	0/80	68/80
Row-echelon certificates	0/80	0/80
Canonical spectral norm	0/80	0/80
Total pushforward measures	33/80	63/80
Continuous linear representations	48/76	49/76
Abstract phase-space tiles	0/76	0/76
Finite conditional Ruzsa distance	0/76	44/76
Structured Bohr data	31/76	31/76
Total	**112/624**	**255/624**
Core	29/152 (19.1%)	74/152 (48.7%)
Derived	49/272 (18.0%)	129/272 (47.4%)
Advanced	24/78 (30.8%)	36/78 (46.2%)
Research	10/122 (8.2%)	16/122 (13.1%)

Unseen IDs are the complement of the archived Visible IDs: a use passed in both trials counts twice, and a use passed once counts once. These rules recover 112/624 and 255/624 exactly. Later repair receives feedback on these same uses and measures observed-use completion. The script `Experiments/recompute_target_intervals.py` and its companion JSON retain per-target counts, resampling settings, and input hashes.

G.3 Operational controls, untouched holdout, and mechanism slices

Runs A, B, and C in the main text correspond to archived v3 (eight targets, two trials), the four-target two-trial pilot, and v4 (eight targets, four trials). The two repeated repair runs are v6 binary and detailed, reusing Run C initial statements. These overlapping development configurations are not independent replications or complete runs of Algorithm 1.

The primary ladder is accompanied by controls that keep its scope explicit. These panels use their archived protocols and are not merged with the 800-opportunity headline table. The first control gives every revision condition binary downstream feedback; the second records earlier protocol versions; the third evaluates frozen statements on an untouched hidden-use set. The development cases are mechanism traces rather than estimates of generator-level generalization.

Table 22: All-binary-source four-arm control. Every condition receives binary feedback; FUSE-C keeps the Root fixed, while joint FUSE may revise the Root and consumers. The denominator is the archived 8-case, 2-trial cohort.

Condition	**Strict E2E**	**Rate (%)**
Direct	40/800	5.00
Visible	70/800	8.75
FUSE-C (Root fixed)	88/800	11.00
FUSE (joint revision)	93/800	11.625

The joint condition adds only five completions over FUSE-C in this control, with 15 paired gains and 10 paired losses. It therefore supports the possibility of joint revision while showing that its stability depends on the protocol; it is not a replacement for the detailed-feedback continuation in the primary ladder.

Table 23: Earlier protocol controls, reported without pooling them with the primary panel. All entries are strict E2E completions with the displayed fixed denominator.

Protocol	Direct	Visible	Binary/FUSE-C	FUSE
Minimal 4-case, 2 trials	35/400	121/400	151/400	144/400
8-case, 4 trials, recovery fix	163/1600	256/1600	356/1600	301/1600

The earlier controls show why the main claim is protocol-qualified: revision scope, evaluation noise, and failure localization can change the final ranking. In particular, the 8-case recovery control has FUSE-C above joint FUSE, so the primary detailed-feedback gain should be read as a measured continuation under its frozen-seed contract, not as a universal monotonicity law.

Table 24: Held-out proof transfer within seven development targets. One revision round receives feedback on IDs 1–20; IDs 21–50 supply three evaluation repeats each. These 630 opportunities test within-target transfer, not whole-family or untouched-target generalization.

Frozen statement	Completed	Rate (%)
Initial Direct statement (S_0)	167/630	26.51
No-diagnostic retry	150/630	23.81
Binary-feedback final	189/630	30.00
Detailed, old-runner selection[†]	138/630	21.90

Binary feedback completes 22 more uses than the initial statement (+3.49 percentage points), with a paired-target interval crossing zero as detailed below. The initial statements are generated without consumer exposure. The panel contains development Cases 1–6 and 8, with thirty held-out uses and three downstream repeats per target; each revised statement comes from one generation run. Thus 630 counts evaluation opportunities on 210 distinct target–use pairs, not 630 independent tasks or unseen targets. Case 7 lacks three revised statements and its nine downstream evaluations were not scheduled.

Uncertainty of held-out proof transfer. A paired percentile bootstrap over the seven targets (50,000 resamples, seed 20260926) gives a 95% interval of $[-2.38, 9.84]$ percentage points for binary feedback minus the initial statement. Each sampled target retains all thirty held-out uses and three evaluation repeats in both arms. The per-target initial/binary counts are 36/31, 13/13, 55/49, 22/39, 19/20, 14/22, and 8/15 for Cases 1–6 and 8, each out of 90. These are development-target resampling intervals, not a population estimate on untouched targets.

Candidate hashes identify eight target–condition contrasts that reevaluate the identical initial statement: four Retry, two Binary, and two Detailed. Their after-minus-initial counts are 0, -10, -6, -5, -12, -1, 9, and 8 (each out of 90). The sample standard deviation is 7.70 completions, or 8.55 percentage points. Contrasts share initial evaluations, so this is a descriptive measure of evaluation variability, not eight independent noise samples. The script `Experiments/recompute_proof_intervals.py` and companion JSON retain all counts, candidate hashes, and input hashes; only evaluation repeats 01–03 are used.

[†] The detailed-feedback endpoint is provisional and excluded from the main comparison in Table 7. Case 4's revised Root passed compilation, admission with no proof placeholders, and semantic review, but a workspace check rejected its unimported scratch file. The ninety recorded held-out outcomes therefore evaluate the initial Root, with ten passes. The recovered candidate has not produced a scored correction; if it completes t of those ninety uses, the corrected total will be $(128 + t)/630$. The old 138/630 result is retained with its candidate identity. Repeated evaluations of the same Root can also differ: the initial Case 4 Root scores 22/90 in the initial arm and 10/90 after fallback. This study tests held-out use proofs, separately from the 431/800 observed-use declaration result.

Table 25: Development mechanism slices. Each row is a paired trace from the development cohort and is reported to explain shared-interface formation, not to estimate FUG.

Trace	Before	After	Net change
Structured Bohr data (C8/T4)	17/50	31/50	+14
Finite conditional Ruzsa distance (C7/T4)	15/50	22/50	+7
Continuous linear representations (C5/T1)	26/50	28/50	+2
Indexed group generators (C1/T2)	38/50	41/50	+3

The Bohr trace exposes a shared layer for `dilate`, width, chord sets, and extremal data; the Ruzsa trace organizes a conditional kernel, fallback, `ReferencePair`, and `tauRef`. The other two traces provide smaller examples of the same pattern: word transport and presentation structure in the group case, and product representation with transport coherence in the continuous-representation case. These cases were selected from the development cohort to make the edit and its reuse inspectable; they are marked development and are not included in the untouched generalization estimate.

Exploratory and statement-stage diagnostics. Group-presentation development pilots provide a small proof-transfer check. A base pilot completes 3/6 held-out uses with both its initial and revised statements. A separate pilot with human-supplied design guidance improves from 0/6 to 4/6; the hint specifies complete relation data, the kernel equality, and quotient operations. Each has one upstream trajectory. A subsequent fresh-use study on another frozen pair completes 18/18 versus 17/18 over six uses and three downstream repeats. These distinct studies retain their own candidates and tasks; the hinted gain illustrates an achievable design improvement without establishing an unaided method advantage.

Additional archived runs are kept as diagnostics with their original labels. An exposure-panel adjustment (Visible-12) changes 214/400 to 261/400; an API-only binary-feedback run changes 97/400 to 193/400; a four-case consumer-feedback diagnostic changes 251/400 to 282/400; and a small faithful-prompt recovery test changes three preselected failed Root gates from 0/3 to 3/3. These records show where prompt constraints and API feedback remove avoidable interface errors, but they use exploratory or statement-stage protocols and do not contribute to the primary E2E denominator.

Six-condition execution pilot. Three authored targets (FUB013, FUB023, and FUB032) each schedule 47 held-out declaration tasks per condition. Table 26 reports recorded completions against that planned denominator. Some FUB013 checks remain unresolved: three under Direct, eleven under Feedback-only, and one under Full FUSE. They are not counted as formalization failures. Every source summary marks this pilot as unsuitable for paper-level method comparison.

Table 26: Pilot diagnostic: recorded completions out of 47 scheduled uses per target (141 total). Incomplete checks and shared blocking prevent interpreting these counts as a settled ranking.

Condition	FUB013	FUB023	FUB032	Total
Direct	0	5	3	8
Descriptions-only	0	7	4	11
Feedback-only	0	4	2	6
Full FUSE	3	3	2	8
Matched self-refinement	2	3	4	9
Local repair	2	5	4	11

Only 17 of the 141 scheduled tasks have different recorded pass indicators across conditions (4, 10, and 3 by target). This descriptive indicator groups failed, blocked, and unresolved tasks as not recorded passed; it does not equate their causes. Shared prerequisite failures and budget limits leave little signal for comparing design quality. This pilot motivates checking execution viability before a larger comparison and contributes no measured FUG claim.

Repeated declaration repair. Four archived configurations keep the upstream statement fixed and allow at most three downstream repair rounds with either binary or detailed feedback (Table 27).

Experiments 014 and 015 share eight targets; the two 015 conditions vary which descriptions are exposed. This is a repeated development pattern, not four independent benchmark replications. The feedback provider also supplies the semantic acceptance judgments.

Table 27: Remaining failures under accepted interfaces. Tokens count incremental repair work, excluding generation of the shared seed.

	Uses	Remaining failures			Repair tokens (M)	
Configuration		**Seed**	**Binary**	**Detailed**	**Binary**	**Detailed**
011	300	84	49	18	2.237	2.259
014	450	101	32	19	2.878	2.376
015, bridge	250	49	13	5	1.476	1.259
015, distant uses	350	112	51	29	3.087	2.642
Total	1350	346	145	71	9.678	8.536

Detailed feedback leaves fewer failures in every configuration. It removes $275/346 = 79.5\%$ of initial failures; $1279/1350 = 94.7\%$ of conditional opportunities complete. Compared with binary feedback, it leaves 49.0% as many failures and uses 11.8% fewer aggregate repair tokens, although tokens increase by 1.0% in 011. Equal round caps do not imply identical computation, and these data do not isolate feedback from its teacher's acceptance preferences.

Exposure comparison inventory. The ten comparisons catalogued in the evidence review have higher recorded Visible counts in nine entries (Table 28). They share targets and sometimes Direct candidates; the six-condition pilot also contains unresolved checks. This directional tally is descriptive and does not estimate an independent replication rate. The Mathlib-target comparison reverses, showing that exposure benefits are not uniform across sources.

Table 28: The review's exposure-comparison inventory, at declaration level. Entries retain each source's opportunities; the pilot is an incomplete diagnostic.

Source	**Opportunities**	**Direct**	**Visible**
003	400	214	248
005/006	400	283	287
007/008	400	97	193
014	800	155	349
015, bridge	800	155	201
015, distant uses	800	155	238
Convex combinations	99	29	45
Bilinear forms	100	80	88
Six-condition pilot	141	8	11
Mathlib targets, ten pairs	112	53	47

The Mathlib-target study has ten candidate pairs and 112 uses per condition, with all forty semantic-review jobs completed. Candidate approval rises from 3/10 to 4/10 while dependency-closed use approval falls from 53 to 47; one pair accounts for most of the difference (11 versus 3). Source conventions, reference interfaces, and target difficulty can therefore mediate the effect of descriptions.

Shared revision across protocol versions. Table 29 puts all five run–feedback configurations on their full planned opportunities. The three four-condition runs use 800, 400, and 1600 opportunities; their screened Direct/Visible/FUSE-C/FUSE counts are respectively 40/58/75/85, 35/103/120/141, and 163/171/240/245. The two v6 conditions reuse v4's initial candidates, then run new repair trajectories.

Table 29: Revision comparison across versions. Each row retains all planned opportunities, including initial interface failures. Screened counts remove detected source dependencies on unproved declarations.

	Uses	Ordinary		Screened	
Configuration		**FUSE-C**	**FUSE**	**FUSE-C**	**FUSE**
v3	800	88	93	75	85
Four-target pilot	400	151	144	120	141
v4	1600	356	301	240	245
v6, binary	1600	319	268	208	196
v6, detailed	1600	394	313	256	234
Total	6000	1308	1119	899	901

The source procedure segments Root and helper declarations, seeds an unproved set with bodies containing proof placeholders, and propagates explicit name references. Passed uses referencing that set are removed. Ambiguous short names are excluded; implicit instances, notation, and name resolution can be missed. These are source estimates, not Lean-checked proof closure, and no general one-sided error bound is established.

A separate conditional comparison keeps only trials where both revision arms end with an accepted interface. In order across the five rows, it contains 5/6/12/13/13 trials and screened FUSE-C/FUSE counts 75/85, 120/141, 206/245, 190/196, and 232/234. The 49-trial total is 823 versus 901 out of 2450 opportunities, with 25 FUSE wins, 15 losses, and 9 ties. This excludes acceptance regressions and must not replace Table 29. In v4, three of fifteen initially accepted statements lose acceptance and none of seventeen rejected statements recover. In each v6 condition, two initially accepted statements are lost. The pooled counts describe overlapping development runs and establish neither statistical equivalence nor a stable advantage of joint revision.

Outcome-selected consumer-deletion sensitivity. We also compute a deliberately post-hoc sensitivity analysis on the archived all-binary v4 records. The rule selects records for which FUSE failed while FUSE-C succeeded; because the rule is defined after observing outcomes, it is not used for the main estimate. On the full 1,600-record cohort, FUSE-C completes 356 (22.25%) and joint FUSE completes 301 (18.81%), a gap of -3.438 percentage points. Removing the 108 selected records changes the denominator to 1,492, leaves FUSE at 301 (20.17%), and changes FUSE-C to 248 (16.62%), flipping the displayed gap to $+3.552$ points. No Lean artifact or successful proof is added; the change comes entirely from the selected denominator. This panel is therefore a sensitivity diagnostic and evidence for retaining the full fixed cohort, not evidence of method improvement or future-use generalization.

Table 30: Post-hoc consumer-deletion sensitivity on the archived v4 control. The deletion rule is outcome-selected and cannot replace the full-cohort estimate.

Cohort	**Removed**	**Denominator**	**FUSE-C**	**Joint FUSE**
Full cohort	0	1600	356/1600 (22.25%)	301/1600 (18.81%)
Outcome-selected subset	108	1492	248/1492 (16.62%)	301/1492 (20.17%)

G.4 Ablations, Cost, and Robustness

Downstream source size on jointly completed uses. We measure a coarse proxy for adaptation burden in the three four-condition runs and both v6 repair conditions. We pair the same case, trial, and use, retaining a pair only when both use results pass and neither source names a Root/helper declaration marked by the source screen described above. This analysis initially ignores the upstream interface gate; it conditions on common use success, not equal overall reach or accepted upstream designs. After removing comments by the archived regular-expression rule, we count each word or non-whitespace symbol as one lexical token. For each paired set, the reported change is $100(\sum L_{\text{after}}/\sum L_{\text{before}} - 1)$, where L is the whole consumer file's token count. This includes the-

orem text, imports, and boilerplate as well as proof code; it does not separately count conversions or bridge lemmas. Equal-size files remain in the totals.

Table 31: Downstream source under Direct versus Visible on jointly completed, source-screened uses. Change compares total lexical tokens. Last column counts shorter / longer / equal-size Visible files. Withheld descriptions refer to initial generation, before downstream repair.

Run	Descriptions	Pairs	Change (%)	Shorter / longer / equal
v3	All	178	−15.7	125 / 49 / 4
	Shown	90	−25.6	73 / 16 / 1
	Withheld	88	−5.4	52 / 33 / 3
Pilot	All	73	−26.6	49 / 18 / 6
	Shown	45	−35.4	34 / 7 / 4
	Withheld	28	−13.6	15 / 11 / 2
v4	All	203	−19.9	148 / 55 / 0
	Shown	112	−24.9	88 / 24 / 0
	Withheld	91	−14.7	60 / 31 / 0
Combined	All	454	−19.5	322 / 122 / 10
	Shown	247	−27.3	195 / 47 / 5
	Withheld	207	−10.9	127 / 75 / 5

Table 31 gives total source reductions of 16–27% across runs. Restricting further to accepted upstream statements in both arms leaves 34, 31, and 64 pairs, with reductions of 11.6%, 21.8%, and 24.7%; jointly, 99 files are shorter, 27 longer, and three equal. For the description-withheld uses, the full source-screened completion counts without the upstream gate are 129 versus 134 /608 in v3, 36 versus 72 /304 in the pilot, and 264 versus 189 /1216 in v4: 429 versus 395 /2128 combined. Lower source size on shared successes therefore coexists with mixed completion changes. These proof-required runs differ from the declaration-only transfer study in Table 5.

Table 32: Downstream source under FUSE-C versus FUSE on jointly completed, source-screened uses. Every compared use belongs to the observed repair set.

Configuration	Pairs	Change (%)	Shorter / longer / equal	Identical
v3	65	−2.3	7 / 2 / 56	56
Pilot	103	−9.8	16 / 1 / 86	86
v4	175	−19.6	42 / 5 / 128	127
v6, binary	177	−5.1	14 / 11 / 152	152
v6, detailed	204	−11.5	43 / 20 / 141	140
First three	343	−13.6	65 / 8 / 270	269
All five	724	−11.1	122 / 39 / 563	561

Across all five revision configurations, total lexical tokens fall from 86,503 to 76,913 (Table 32). Most pairs are unchanged, and every configuration's median paired change is zero. All pairs in this revision comparison also have accepted upstream statements. The effect concerns part of the downstream code produced by joint revision, which changes both Root and use proofs; it does not isolate the Root edit or successful-work review. Runs share targets and statements, and uses share dependencies, so pooled sign-test probabilities treating uses as independent are not used as population-level evidence. The source screen can both miss dependencies and misidentify names; these counts are not certified Proved FUG.

Boundary checks find no corresponding shortening signal on r8's untouched proof evaluation (27 shorter versus 30 longer files for revised Binary statements), or on the five-refactor study's 29 common successes (10 shorter, 18 longer, one equal; mean tokens approximately 116 versus 110). These diagnostics preserve the distinction between completion, source size conditional on completion, and generalization after revision. The separate work-component and successful-work ablations below remain unmeasured. Per-use sizes, ties, source hashes, and recomputation rules are recorded in `Experiments/adaptation_burden_20260926.json`.

G.5 Major Mathlib Refactors and Invalid Runs

Table 33 reports three completed refactor-comparison batches under their original settings. The pilot tests four post-snapshot refactors, with 20 uses per target and three trials per design. Reviewed-L and Reviewed-S each test four same-intent designs and four separately labeled controls, with four trials per design. All use Lean/Mathlib v4.32.0 at `81a5d257` and an offline `gpt-5.6-sol`/high semantic checker. The pilot uses `gpt-5.6-luna`/low with a 200,000-token cap per run; Reviewed-L uses that solver with a 1,000,000-token cap, and Reviewed-S uses `gpt-5.6-sol`/high at the same cap. Reviewed-S also changes contracts and dependency extraction, so both the evaluator and task specification vary between the reviewed batches. Counts are primary uses per target, not independent refactor groups. Summaries average trials within targets, then weight targets equally.

Table 33: Extended Mathlib refactor comparison. Completion in percent; Δ in percentage points. Controls remain separate from same-intent designs.

Batch	Design change	Uses	Before	After	Δ
Pilot	Lie-algebra basis parameter	20	3.33	1.67	-1.67
	Linear-equivalence construction	20	5	15	+10
	Torsion rename (control)	20	28.33	10	-18.33
	Solvability namespace (control)	20	6.67	3.33	-3.33
Reviewed-L	Restricted power series	10	87.50	87.50	0
	Distributivity axiom bundling	4	0	12.50	+12.50
	Uniform integrability	34	1.47	0	-1.47
	Lie-algebra basis parameter	2	25	12.50	-12.50
Reviewed-S	Restricted power series	10	100	100	0
	Distributivity axiom bundling	12	33.33	91.67	+58.33
	Uniform integrability	25	100	100	0
	Lie-algebra basis parameter	24	91.67	91.67	0
Additional controls: capability, abstraction, or behavior changes					
Reviewed-L	Semilinear quotient equivalence	3	0	100	+100
	Jordan–Hölder relation	6	66.67	91.67	+25
	Completion-map generalization	3	33.33	100	+66.67
	Stopped sigma-algebra	2	0	100	+100
Reviewed-S	Semilinear quotient equivalence	3	50	100	+50
	Jordan–Hölder relation	6	66.67	100	+33.33
	Completion-map generalization	3	33.33	100	+66.67
	Stopped sigma-algebra	2	25	100	+75

The pilot's four-target average is 10.83% before and 7.50% after, with one win and three reversals; the paired-target bootstrap interval for the difference is $[-14.17, 6.67]$ percentage points. Its two substantive refactors average 4.17% and 8.33%. Reviewed-L averages 28.49% and 28.13% on the four same-intent targets, with one win, one tie, and two reversals (difference interval $[-9.38, 9.01]$). Reviewed-S averages 81.25% and 95.83%, with one win and three ties (difference interval $[0, 43.75]$); its entire net gain comes from distributivity axiom bundling. These results make the evaluation's role concrete: later adoption alone does not determine utility, and an aggregate gain can come from one design choice rather than every refactor.

All 24 pilot jobs and all 64 jobs in each reviewed batch completed. The pilot retains five jobs rejected by its workspace rule for forbidden temporary Lean probes. Reviewed-L corrects an earlier theorem-only audit that wrongly rejected 76 compiled definitions or abbreviations: the corrected evaluation reuses the same 648 generated files, rechecks compilation and dependencies, and reruns semantic review without new solver calls. Reviewed-S is a separate run with 704 generated files. Source hashes, original scores, and the correction are retained in the reproduction records.

Per-pair records retain the refactor sources, original and API-neutral wording, review decisions, repeated-run scores, and outcome classifications. The full results report both the paired comparison and its coverage of the preselected pool.

H Case Studies and Concept Discovery

H.1 Known Mathematics and Evidence for Discovery

The convexity example in Figure 2 selects four uses. It preserves the four declaration excerpts, including the operation and both laws; the rows below explain the work needed for each use. Figure 9 retains the complete 33-task graph, the four declaration excerpts, and the refactor design pressures. The original refactor sources include indexed convex combinations, convex sets over general scalars, and the linked review discussion.

The earlier three-stage convex-combination update illustration is retained here as a method-history reference. It is the compact execution–revision–replay view that preceded the C7/T4 pilot in the main text; its full mathematical context and the 33-task dependency graph follow below.

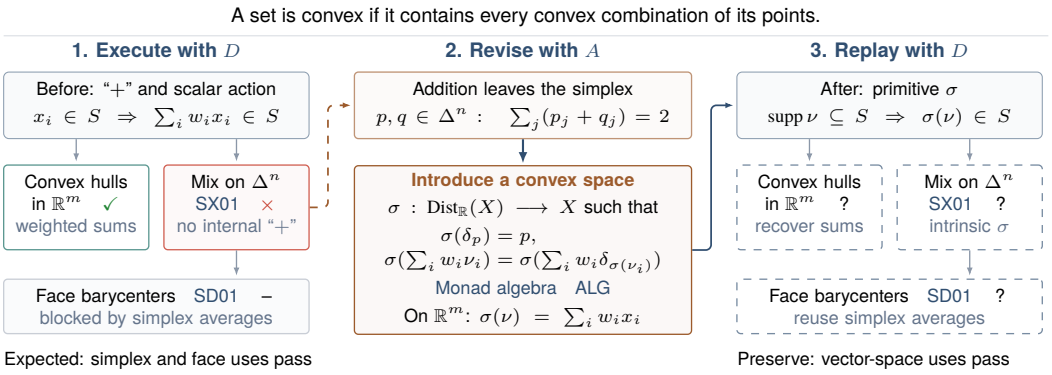


Figure 8: Earlier convex-combination update illustration. A downstream failure returns a design requirement, the upstream edit introduces an intrinsic convex space operation, and the same uses are replayed. This schematic is retained as historical context; the main-text pilot is Figure 5.

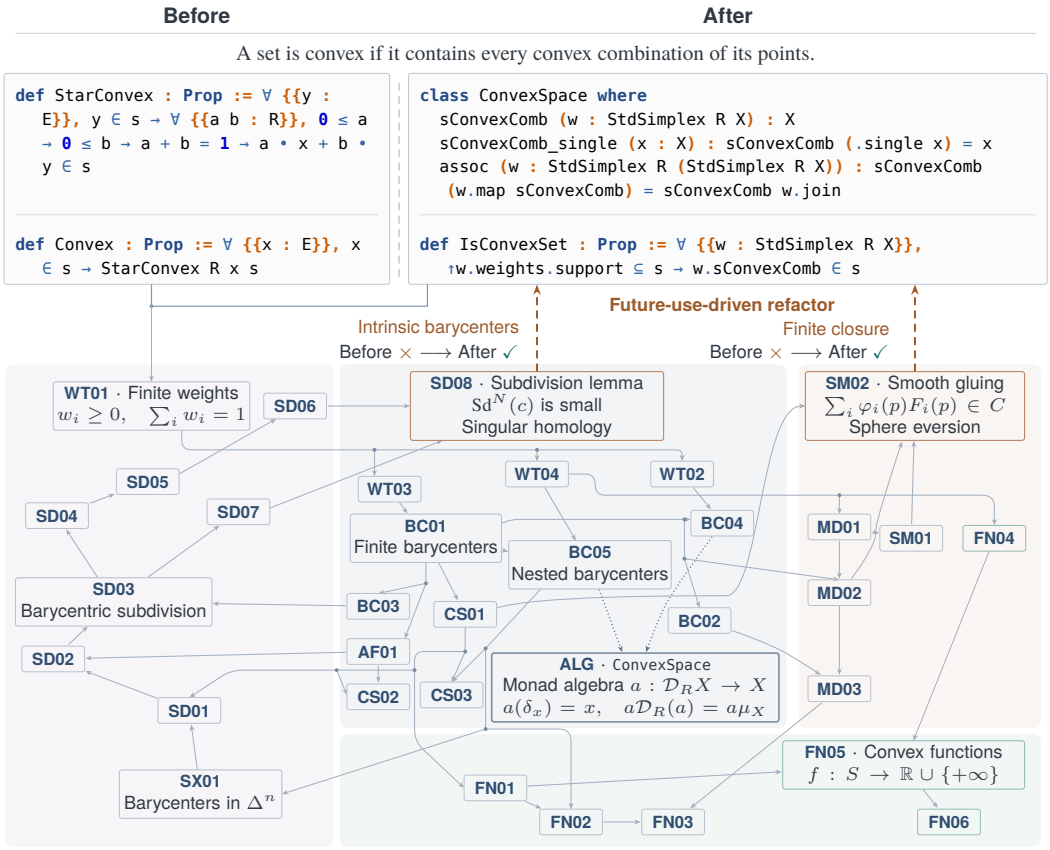


Figure 9: The full convexity design case: 33 downstream tasks and an algebra interpretation. Solid paths preserve prerequisite reachability; dashed feedback arrows identify the subdivision and gluing demands. Tags link to Table 34; ×/✓ compare the primitives exposed by each design. The extended-valued function tasks illustrate prospective applications.

Figure 9 makes the algebra explicit. A point of $\mathcal{D}_R X$ is a finite formal combination $\sum_i w_i \delta_{x_i}$, with nonnegative weights in R summing to one. The unit puts weight one on a single point; multiplication μ_X flattens a distribution of distributions by multiplying outer and inner weights. An algebra $a : \mathcal{D}_R X \to X$ evaluates these combinations and satisfies

$$a(\delta_x) = x, \qquad a \circ \mathcal{D}_R(a) = a \circ \mu_X.$$

The second equation says that evaluating each inner combination and then the outer one gives the same point as flattening all weights before evaluation. Mathlib's design notes identify `StdSimplex R` as the monad and `sConvexComb` as its algebra operation. Its scalar assumptions are `PartialOrder R`, `Semiring R`, and `IsStrictOrderedRing R`. Over the reals, the correspondence between convex structures and Eilenberg–Moore algebras is classical (Jacobs, 2010, Theorem 4). This classical connection explains the common laws in the displayed design.

Figure 9 contrasts two designs, each with two declarations. Before shows `StarConvex` and `Convex` from `227e3bac`. They require `Semiring R`, `PartialOrder R`, `AddCommMonoid E`, and `SMul R E`, with ambient addition and scalar multiplication. `StarConvex` asks for closure along segments from a fixed point; `Convex` requires this at every point in the set. After shows the foundational `ConvexSpace` alongside `IsConvexSet`, introduced by `63ad6315` (21 May 2026, PR #38905). Both use `Semiring R`, `PartialOrder R`, and `IsStrictOrderedRing R`; the predicate additionally assumes `ConvexSpace R X`. `IsConvexSet.of_convexCombPair_mem` derives finite closure from binary closure under its field assumptions. The declarations coexist: `ConvexSpace` predates this PR, and the old predicates remain available.

The snippets retain the predicate bodies and all three `ConvexSpace` fields: `sConvexComb`, `sConvexComb_single`, and `assoc`. Scalar, point, and distribution variables are named `R`, `X`, and `w`;

strict-implicit binders use Lean's equivalent ASCII notation `{{...}}`. Parameters, instance binders, and the custom constructor name are omitted. ALG gives a mathematical reading of the two law fields. The linked sources supply the surrounding assumptions for these excerpts.

Both feedback arrows reach the shared After frame; their labels distinguish the intrinsic operation from subset closure. SD08 motivates intrinsic barycenters on simplex points. Over the reals, their ambient zero and sum do not stay in the simplex. The source supplies `ConvexSpace R (StdSimplex R I)` directly by `join`, independently of `IsConvexSet`. SM02 uses both layers: `ConvexSpace` provides combinations, and `IsConvexSet` keeps them in the target set. Its feedback arrow singles out the finite-closure requirement. Smoothness is supplied by separate results. The bundle records CS01 (convex subsets) as an ancestor of SD08 through SD01 and as a direct prerequisite of SM02; this is task ancestry, while the intrinsic operation is supplied by the space structure.

The figure's $\times/\checkmark$ labels indicate which displayed design exposes the required primitive. The bundle instantiates SM02 pointwise over the reals; SD08 highlights intrinsic averaging. Appendix I.2 separates these design comparisons from claims about whole-application provability.

Figure 9 separates mathematical demands from the reach of the resulting design. In the account supplied with this case, Yaël Dillies identifies two original goals of the overhaul: non-field scalars for Patrick Massot's sphere-eversion work, and non-module spaces for the subdivision lemma in singular homology with Andrew Yang. He also anticipates extended-valued convex functions, naming `IsConvexFunOn`. The July 2023 sphere-eversion discussion independently documents the first need: partitions of unity provide function-valued coefficients for gluing, so the scalar ring need not be a field. The figure traces feedback from smooth convex gluing and the subdivision lemma to two changes: make finite combinations primitive, and separate their evaluation from ambient module operations. Amber arrows mark the motivating demands; the teal branch shows prospective uses.

Our convex-combination case contains 33 natural-language downstream tasks, all retained in Figure 9. This diagram records the original bundle dependencies; Appendix C.1 reviews which become mandatory in the primary task DAG. Solid arrow paths display the transitive reduction: 47 of 57 direct dependencies, preserving the complete prerequisite reachability relation. Paths from the same source may share a trunk; dots mark branches, and unmarked crossings do not connect. Branch dots identify shared paths. Every task carries its original tag; eight pivotal tasks also show mathematical content. The connection from both code frames to WT01 identifies shared weight constraints, as an annotation. ALG explains the core `ConvexSpace` design through its single-point and mixing laws. Dotted links connect this annotation to the two law fields. SD08 and SM02 both feed back to the After frame, which stacks `ConvexSpace` above `IsConvexSet`. Arrow direction, rather than vertical position, specifies prerequisite order. Extended-valued convexity (FN04–FN06) remains a prospective use: the pinned snapshot contains no declaration under the name `IsConvexFunOn`, and its TODO list still calls for convex functions on convex spaces. Table 34 records the mathematical requirements for these tasks.

Table 34 explains every tag and lists its original direct prerequisites. Throughout the table, weights are nonnegative and sum to one; bar_X denotes the chosen averaging operation, without assuming addition on X. Division and topological hypotheses remain local to tasks that need them. Homology-class conclusions in SD06 and SD08 apply to cycles.

Table 34: Mathematical meaning of every tag in Figure 9. The last column lists the original direct prerequisites, including edges omitted by the figure's transitive reduction. ALG is an interpretation of two laws, not an extra benchmark task.

Tag	Mathematical requirement	Prerequisites
WT01	Finite convex weights: on a finite set I, require $w_i \geq 0$ and $\sum_{i \in I} w_i = 1$.	—
WT02	Single-point weights: assign weight 1 to a chosen index and 0 to every other index. Verify the weight conditions.	WT01
WT03	Merge weights under a relabeling $f : I \to J$: $v_j = \sum_{i:f(i)=j} w_i$ is again a system of convex weights.	WT01
WT04	Flatten nested weights: outer weights w_i and inner weights $(v_{ij})_{j \in J_i}$ give convex weights $w_i v_{ij}$ on pairs (i, j). Each finite J_i may depend on i.	WT01

Continued on the next page

Table 34 continued: convex-combination tasks

Tag	Mathematical requirement	Prerequisites
BC01	Form the finite barycenter $\mathrm{bar}_X(w, x)$ without assuming addition or scalar multiplication on X. Merging equal points and adding their weights leaves the result unchanged.	WT03
BC02	Combine two points with $a, b \geq 0$, $a + b = 1$. Weights $(1, 0)$ and $(0, 1)$ return the respective endpoints; equal endpoints return that point. Over $\mathbb{R}$, write $a = t$, $b = 1 - t$.	WT01, BC01
BC03	Reordering points and their weights by the same bijection leaves the barycenter unchanged.	WT03, BC01
BC04	Unit and constant-family laws: a point of weight 1 evaluates to itself; a family consisting only of x has barycenter x.	WT02, BC01
BC05	A barycenter of barycenters equals the flattened one: $\mathrm{bar}_X(w, i \mapsto \mathrm{bar}_X(v_i, x_i)) = \mathrm{bar}_X((w_i v_{ij}), (x_{ij}))$, where $x_i = (x_{ij})_{j \in J_i}$ and each finite J_i may depend on i.	WT04, BC01
AF01	A map $f : X \to Y$ preserves combinations if $f(\mathrm{bar}_X(w, x)) = \mathrm{bar}_Y(w, f \circ x)$ for every finite weighted family. Verify that identities and compositions preserve combinations.	WT03, BC01
MD01	Module realization: on a module over an ordered semiring, define $\mathrm{bar}_M(w, x) = \sum_i w_i \bullet x_i$ and establish the unit and flattening laws.	WT01, WT04
MD02	Show that the abstract barycenter on this module realization agrees with the ordinary weighted sum $\sum_i w_i \bullet x_i$.	MD01, BC01
MD03	Over an ordered field, prove equivalence between closure under two-point combinations and closure under all finite convex combinations in a vector space.	MD02, BC02
SM01	Let P be a smooth manifold and V a finite-dimensional real vector space. For finitely many smooth maps $F_i : P \to V$ and smooth nonnegative weights $\varphi_i : P \to \mathbb{R}$ summing pointwise to 1, prove $p \mapsto \sum_i \varphi_i(p) F_i(p)$ smooth.	MD01
SM02	Under SM01's assumptions, if $C \subseteq V$ is convex and $F_i(p) \in C$ for all i, p, prove the average is smooth and takes values in C. This pointwise real-scalar task is motivated by smooth gluing in sphere eversion.	SM01, MD02, CS01
SX01	Barycenters in a simplex: for finite I and $\Delta_I = \{p : I \to R \mid p(i) \geq 0, \sum_i p(i) = 1\}$, show that $p(i) = \sum_k w_k p_k(i)$ remains in Δ_I whenever $p_k \in \Delta_I$.	WT04, BC05
CS01	Convex subsets: $S \subseteq X$ is closed under every finite barycenter of its points.	BC01
CS02	Convex-set constructions: empty, whole, and singleton sets; arbitrary intersections; images and inverse images under combination-preserving maps.	CS01, AF01
CS03	Internal barycenters: for convex $S \subseteq X$, restrict the ambient barycenter to points of S and inherit the combination laws.	CS01, BC05
SD01	Face barycenters in the real standard simplex Δ^n: for a nonempty vertex set J, set $b_J(i) = 1/\lvert J \rvert$ on J and 0 elsewhere. Prove membership in the face spanned by J; division is local to this construction.	SX01, CS01
SD02	A strict flag $J_0 \subsetneq \cdots \subsetneq J_k$ of nonempty faces of Δ^n determines an affine map $\Delta^k \to \Delta^n$ sending vertex r to b_{J_r}. Prove its image lies in the largest face, spanned by J_k.	SD01, AF01
SD03	Barycentric subdivision: define $\mathrm{Sd}(\sigma)$ as the signed sum of simplices obtained from maximal flags of nonempty faces, for an oriented simplex $\sigma : \Delta^n \to X$.	SD02, BC03
SD04	Face compatibility: restricting subdivision to a face agrees with subdividing that face, with the corresponding orientation.	SD03, BC03
SD05	Chain-map law: for every singular chain c, prove $\partial \mathrm{Sd}(c) = \mathrm{Sd}(\partial c)$.	SD04
SD06	Chain homotopy: construct h with $\partial h + h\partial = \mathrm{Sd} - \mathrm{id}$. In particular, a cycle and its subdivision represent the same homology class.	SD05
SD07	For a singular simplex $\sigma : \Delta^n \to X$ and an open cover of its image, find N such that every resulting simplex after N subdivisions has image in some cover member. Use a Lebesgue number for the pulled-back cover of Δ^n and decreasing subdivision mesh.	SD03
SD08	For a finite singular chain c and an open cover of the union of images of its nonzero terms, find one N such that every nonzero simplex term in $\mathrm{Sd}^N(c)$ has image in some cover member. If c is a cycle, its homology class is unchanged.	SD05, SD06, SD07

Continued on the next page

Table 34 continued: convex-combination tasks

Tag	Mathematical requirement	Prerequisites
FN01	For convex $S \subseteq X$ and an ordered combination space Y, a map $f : X \to Y$ is convex on S if $f(\mathrm{bar}_X(w, x)) \leq \mathrm{bar}_Y(w, f \circ x)$ for every finite weighted family with $x_i \in S$.	CS01, BC01
FN02	Apply FN01's definition to obtain the finite Jensen inequality for a specified weighted family, including zero coefficients. Verify that zero-weight terms have no effect on either side.	FN01, BC05
FN03	Classical agreement: for a real-valued function on a convex subset of a real vector space, finite-family convexity is equivalent to $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$ for $0 \leq t \leq 1$.	FN02, MD03
FN04	For real convex weights and values in $\mathbb{R} \cup \{+\infty\}$, a positive-weight infinite term gives $+\infty$. Otherwise discard zero-weight infinite terms and take the weighted sum of finite values. Verify that nested averages flatten correctly.	WT01, WT04
FN05	For convex $S \subseteq X$ and $f : S \to \mathbb{R} \cup \{+\infty\}$, define convexity by the finite Jensen inequality with real convex weights and FN04's infinity convention. When all values are finite, recover the real-valued inequality.	FN01, FN04
FN06	Convex-set indicators: define $\iota_S(x) = 0$ on a convex set S and $+\infty$ outside it, and prove that ι_S is convex.	FN05, CS01
ALG	Core design of `ConvexSpace` as a monad algebra: view finite weighted families as distributions $\mathcal{D}_R X$. Evaluation $a : \mathcal{D}_R X \to X$ obeys $a(\delta_x) = x$ and $a \circ \mathcal{D}_R(a) = a \circ \mu_X$, where μ_X flattens nested distributions.	BC04, BC05 (interpretation)

The following code-level constructions provide complementary evidence of reuse. They are pinned to the later snapshot `e3a0ab1` (18 August 2026), which includes additions after #38905. They are distinct from the application branches in Figure 9. In this snapshot, `Analysis/Convex/MetricSpace.lean` is the only file outside `Geometry/Convex` that mentions `IsConvexSet`. These constructions document available support, not widespread adoption.

Construction	Declaration or source
Modules	`ConvexSpace.ofModule`
Affine spaces	`AddTorsor.toConvexSpace`
Affine transport	`IsConvexSet.image`, `preimage`
Product sets	`IsConvexSet.prod`, `pi`
Convex hulls	`Convexity.convexHull`
Convex subtypes	`ConvexSpace.subtype`
Submodule instances	`ConvexSpace R S` instance
Convex-distance laws	`IsConvexDist.subtype`
Continuous binary combinations	`continuous_convexCombPair`
Affine maps between simplices	`StdSimplex.isAffineMap_map`
Possibly infinite vertex sets	`StdSimplex.topologicalSpace`

For a convex subset S of a real normed space, the subtype and distance instances let the joint-continuity theorem apply to $[0, 1] \times S \times S \to S$ directly. The old predicate formulates the same operation in the ambient space; the new construction exposes it on S itself. Vertex maps between simplices preserve convex combinations and can be bundled as `ConvexSpace.AffineMap`, with `comp` and `assoc` supplying composition. The older `AffineMap` instead requires affine torsors over modules. Finally, the older `stdSimplex` assumes a finite vertex type, whereas `StdSimplex` uses finitely supported weights on any vertex type. Under the source's ordered topological ring assumptions, its topology is determined by maps from finite simplices.

The module and affine-space constructions are explicitly installed through named definitions. Convex hulls depend on `IsConvexSet`, while the submodule and convex-distance constructions use `ConvexSpace.subtype`. The earlier `ConvexSpace.ofConvex` already assumed a `ConvexSpace` compatible with a module over an ordered field; it constructs a convex subtype. Its associativity proof expanded weighted sums, whereas the new subtype construction invokes `iConvexComb_assoc`. This historical contrast illustrates how a shared law can replace repeated sum expansions; it motivates the proof-work review in Section 3 through a concrete example. The excerpts credit Alexander Bentkamp, Yury Kudryashov, and Yaël Dillies for `Convex` (copyright 2019 Alexander Bentkamp); Yaël

Dillies for `StarConvex` (2021) and `IsConvexSet` (2026); and Kim Morrison, Andrew Yang, and Yaël Dillies for `ConvexSpace` (copyright 2025 Lean FRO, LLC), under the Apache 2.0 license.

The historical example in Figure 10 shows how a reusable object can separate index constraints from the construction of a complex. It complements the convexity example and motivates a future concept-formation study; no generated concept-formation trace is included in this paper.

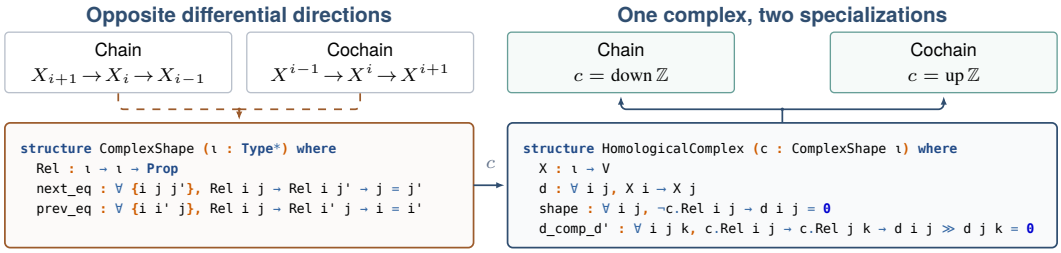


Figure 10: From index constraints to a shared construction: `ComplexShape` lets `HomologicalComplex` represent both chain and cochain complexes. Each Lean code block links to its full Mathlib declaration; ambient parameters and default proof tactics are omitted.

H.2 Concept-Formation Criteria

We assess *context-relative concept formation* with the same three product requirements for every method: the abstraction is absent from its supplied input context; two distinct realizations have complete proofs of the shared laws, with one recovering the target; and complete proofs in at least two held-out mathematical use families actually depend on its objects or laws. These criteria do not require execution feedback, a revision hypothesis, or any particular generation process. Audit each method's actual inputs before its first proposal, including the target, prompts, library access, retrieval, corpus-provided abstractions, tool responses, and feedback, checking equivalent formulations as well as names. A retrieved or supplied abstraction counts as reuse rather than context-relative formation; unavailable evidence remains unresolved. The audit includes corpus and extraction inputs when those stages belong to the evaluated method.

The proposed discovery evaluation distinguishes three nested counts on discovery trials. Proposed counts distinct abstraction proposals within each target–trial, including proposals later rejected or left unresolved. Verified requires the context-absence audit and complete Lean proofs of the shared laws and both realizations. Reused is the subset with complete proofs in at least two held-out mathematical use families that actually depend on the abstraction. Totals sum target–trial counts, with trial counts alongside them; they do not count globally unique mathematical concepts. Concrete data or witnesses instantiate the original task, and each realization proves the laws declared by its shared structure. A completed concept case would collect the objects, realizations, certificates, and downstream dependencies. Post-freeze ablations compare the candidate with valid earlier or reduced designs to test each change's effect on completion and proof work. Invalid counterfactuals and incomplete certificates receive separate status entries; the evidence boundaries are collected in Appendix I.2.

Direct, Descriptions-only, and DNA-style methods could satisfy these criteria without a downstream trace. This paper contains no completed concept-formation run, so the criteria are retained as a specification for future evaluation rather than as a measured result.

I Reproducibility, Scope, and Limitations

I.1 Reproduction Materials

Reproduction records have four parts. Data manifests retain targets, graphs, splits, sources, and licenses. Run configurations retain role-specific models, prompt hashes, tool versions, seeds, and budgets. Round traces retain candidates, checks, upstream-visible feedback, four-field revision hypotheses, statement diffs, replay outcomes, and selection evidence. Evaluation records retain node outputs, dependencies, proof status, component costs, and final summaries. Messages actually received upstream are stored separately from private checker records to audit feedback access. Devel-

opment data, major Mathlib refactor comparisons, and untouched test targets are marked separately. Incidents, retries, failures, and protocol changes remain recorded; public artifacts exclude credentials and personal information. The manuscript archive contains paper sources; the reproduction manifest supplies the locations of full experimental artifacts.

For the major Mathlib refactor evaluator calibration, the archived record contains all 33 judge inputs and outputs, each design's two trial scores, and 633 hash-verified source/project files covering 35 of the 44 reported workspaces. The missing nine workspaces and per-use compilation, dependency, and semantic verdicts remain to be archived. Correlations and selection losses are recomputed from raw judgments and trial summaries; this reanalysis does not rerun downstream checks. The three added provenance controls each use two short requirements rather than the original cohort's full refactor declaration contracts. Artifact provenance and the statistical correction are retained with the analysis.

I.2 Evaluation Scope and Mathematical Novelty

Compatibility is conditional on the task bundle, its declared dependency workflow, the downstream autoformalizer, and budget. Reviewing source edges reduces dependence on one implementation but does not establish invariance to all alternative proof paths. Cases retaining unresolved implementation constraints support only workflow-specific claims and are separated from the primary reviewed test. Failure can arise from design, search, or resource limits; it does not establish mathematical impossibility. Expert solutions and solver-sensitivity comparisons assess these effects. Semantic judgments require independent certificates or expert checks. Designated open theorem proofs are tracked explicitly: declaration acceptance, a use proof relative to its premises, and a fully proved dependency closure support different claims. Verified concept cases require complete proof dependencies for their exhibited realizations and applications. A law assumed in a field or a compilation-breaking removal cannot certify the proposed abstraction's value.

The primary split withholds use families from candidate construction. A forward-in-time extension additionally fixes a historical design cutoff and evaluates mathematics first introduced afterward by contributors other than the design authors. Its provenance records identify the design commit, use-introduction commits, authorship, and equivalent earlier formulations. Design information and revision evidence must predate the cutoff; later additions to a ported root are excluded. This adds a chronological separation to family holdout. The dated subset and its results are reported separately from the five-pair refactor comparison and the primary family split.

Context-relative formation concerns the material available before a run's proposal, including equivalent abstractions. Supplied abstractions count as reuse, and uncertain availability remains unresolved. Held-out evaluation controls design-time access; it does not establish absence from pretraining or priority in mathematical history. API-neutral descriptions measure sensitivity to explicit code cues, while the tasks and DAG still come from the later library and candidate code remains visible. This control reduces one source of after-design advantage but cannot erase memorized code or remove the selection of consumers from the after snapshot. Multiple proved realizations and held-out-family reuse are product standards shared across methods. Downstream evidence and prediction fulfillment support a separate account of the FUSE mechanism. Major Mathlib refactors motivate the method, while claims about generated concepts require actual outputs and provenance records. The expert reference is a point of comparison rather than an upper bound, and agreement between two downstream autoformalizers does not exclude preferences they share. Scores cover traceable consumers of a pinned snapshot; targeted refactor gains do not imply improvement on every use or coverage of all future mathematics.

Figure 2 contrasts available design primitives, not the provability of sphere eversion or subdivision in Lean. Ambient-coordinate proofs remain possible. The historical demand for non-field scalars does not certify instances for every function ring under the pinned assumptions; the bundle's smooth-gluing task is pointwise over the reals. The figure combines an existing space structure, a later set predicate, and subsequent uses. The tagged requirements are distinct from completed formalizations and from an experimental FUSE trajectory.

J Related Work

This section places FUSE among four lines of work that are often conflated in discussions of feedback-driven formalization: iterative self-refinement, statement autoformalization, proof-assistant-guided theorem proving, and learning reusable libraries. The distinction that matters here is the object being revised and the object being evaluated. Existing systems commonly improve an answer, a proof trajectory, or a library under a fixed collection of tasks. FUSE revises a shared Lean statement from the consequences of its downstream uses and evaluates that statement on uses that did not provide the revision feedback.

J.1 Iterative self-refinement and execution feedback

Madaan et al. (2023) introduced *Self-Refine* as a general generate–critique–revise loop: one language model proposes an output, produces textual feedback, and uses that feedback to improve the output. Shinn et al. (2023) similarly stores verbal accounts of failures and uses them as reinforcement for later attempts, while Chen et al. (2023) studies self-debugging from execution errors in programming tasks. These methods establish an important baseline for any claim about feedback-driven generation. They show that a model can turn a failure signal or a critique into a better next attempt, but they do not by themselves specify which part of a shared mathematical abstraction should change or whether the change transfers to a new family of uses.

Our Binary-FUSE condition (called FUSE-C, for fixed statement, in the main tables) is deliberately close to this literature. It keeps the upstream statement fixed, gives the downstream repair process a binary success/failure signal, and measures how many observed use files can be completed after repair. Calling this condition a self-refinement baseline is more informative than presenting it as a wholly new algorithm: it tests the value of execution feedback when the representation itself is held constant. Full FUSE adds a different level of intervention. It attributes repeated failures and repeated adaptation work to a shared representation, assumption, structure, instance, operation, or law; records a proposed statement edit and its expected benefit; replays the affected uses; and retains the edit only when the replay and protected-use checks support it. The final statement is then evaluated on held-out uses. Thus the difference from Self-Refine or Reflexion is not the presence of another model call. It is the feedback target (a shared mathematical statement), the evidence used for credit assignment (multiple downstream uses), and the separation between uses that supervise a revision and uses that test generalization.

Execution feedback has also been used to evolve formalization candidates, for example in neuro-symbolic search for diverse statements (Lu et al., 2026) and verification-guided refinement for mathematical autoformalization (Pu et al., 2026). These systems reinforce the value of checking generated artifacts. FUSE complements them by making the unit of revision explicit and by treating downstream mathematical reuse as the signal for abstraction design rather than only as a pass/fail filter for one translation.

J.2 Autoformalization and faithful evaluation

Early work on large-language-model autoformalization translated natural language mathematics into Lean or related proof assistants (Wu et al., 2022). ProofNet paired autoformalization with formal proof of undergraduate mathematics (Azerbayev et al., 2023). More recent work has focused on whether a statement faithfully preserves the source meaning: FormalAlign proposes an automated alignment evaluation (Lu et al., 2024), Rethinking and Improving Autoformalization studies dependency-aware retrieval and faithful metrics (Liu et al., 2025a), and Reliable Evaluation and Benchmarks for Statement Autoformalization analyzes failure modes in statement-level evaluation (Poiroux et al., 2025). FormalRx makes semantic failure rectification an explicit stage, MathForm combines retrieval and verification-guided refinement, and M2F studies large-scale formalization of mathematical literature (Wang et al., 2026a; Pu et al., 2026; Wang et al., 2026b).

This literature gives the first half of our evaluation: a statement should compile, use its declarations, and agree with the mathematical requirement. FUSE adds a second question that is orthogonal to local fidelity: after the statement is accepted, can later mathematics use the representation without repeated conversions, missing instances, or re-proving the same bridge? FUG measures that future-use compatibility on a fixed use graph. In this view, semantic checking answers whether the current state-

ment is about the right mathematics, whereas future-use execution tests whether its design exposes the right reusable interface. A statement can pass the former and fail the latter, which is precisely why downstream evidence is useful for revision.

The distinction also clarifies the role of our benchmark. FutureUseBench does not replace faithful statement evaluation or proof checking. It composes them with dependency-aware downstream uses, an observed-use revision split, and a held-out-use split. The benchmark therefore evaluates an executable property of an accepted abstraction rather than treating compilation or semantic agreement as a complete measure of abstraction quality. Recent work on testing benchmarks for automated theorem proving makes a related case for testing the behavior of formal systems beyond a single proof outcome (Kim et al., 2026).

J.3 Proof-assistant feedback and theorem proving

Proof-assistant feedback is central to modern theorem-proving systems. LeanDojo connects language models to Lean environments and retrieved premises (Yang et al., 2023); miniCTX studies repository-scale context (Hu et al., 2025); Lean-STaR interleaves natural-language reasoning with tactic generation (Lin et al., 2025); and DeepSeek-Prover-V1.5 combines proof-assistant feedback with reinforcement learning and Monte-Carlo tree search (Xin et al., 2025). Lean Hammer uses learned premise selection and external automation to improve proof search (Zhu et al., 2026), while LeanAgent and LeanMarathon study longer-lived or long-horizon formal proving (Kumarappan et al., 2024; Zhang et al., 2026).

These systems primarily use feedback to select premises, choose tactics, search a proof tree, or improve a proof-producing policy. FUSE uses the same executable environment for a different inference: a cluster of downstream failures can be evidence that the shared statement has the wrong representation or exposes an incomplete law. The proposed edit is therefore made above the individual tactic level and is checked by replaying the dependent uses. The distinction is one of feedback scope: proof-search feedback changes a proof trajectory, downstream repair changes a use file, and FUSE can change the shared abstraction that many use files import. Conjecturing and formal problem-solving work broaden the formal task to include choosing or forming a problem itself (Sivakumar et al., 2025; Liu et al., 2025b); FUSE keeps that broader motivation in view while focusing on the executable design of a supplied target statement.

J.4 Library learning and reusable abstractions

Library-learning systems provide the closest conceptual precedent for treating abstractions as first-class objects. DreamCoder learns reusable programs and uses a learned library to solve held-out synthesis problems (Ellis et al., 2023). Stitch performs top-down library learning and evaluates compression on held-out programs (Bowers et al., 2023); LILO combines library compression with interpretable documentation and tests a frozen library on new tasks (Grand et al., 2024). LEMMA learns symbolic abstractions for mathematical reasoning (Li et al., 2022), while LEGO-Prover grows a library during theorem proving (Wang et al., 2023). The accompanying analysis of library-learning failures also shows why reuse should be checked directly rather than inferred from a single successful proof (Berlot-Attwell et al., 2025).

Program-synthesis work such as babble learns abstractions with e-graphs and anti-unification (Cao et al., 2022). In formal mathematics, Divide and Abstract decomposes statements and learns abstractions (Min et al., 2026a); Theory-Level Autoformalization argues for interdependent formal theories whose abstractions support later results (Min et al., 2026b); and theory-scale formalization explores this direction for logics for computer science (Feng et al., 2026). DNA-style systems similarly extract concepts, organize them in dependency graphs, and compose them in later formalization (Min et al., 2026a).

FUSE shares the goal of finding reusable structure, and it adopts held-out-use evaluation rather than claiming that such evaluation is new. Its contribution is the source of the design signal and the unit of credit assignment: observed mathematical uses expose missing structure or repeated adaptation, a statement revision names the shared design choice expected to remove that work, and FutureUseBench tests the resulting statement on withheld use families. The benchmark also retains major Mathlib refactors as concrete design comparisons, authored use graphs for controlled feed-

back, and real-library replay for compatibility with existing developments. These resources let us distinguish library growth or compression from the narrower claim that a shared statement generalizes across executable mathematical uses.

J.5 Held-out uses and future-use generalization

Held-out evaluation is established in program synthesis and library learning: DreamCoder and Stitch evaluate learned libraries on tasks or programs withheld from library construction, and LILO evaluates a frozen library on new tasks (Ellis et al., 2023; Bowers et al., 2023; Grand et al., 2024). The formalization literature likewise contains benchmarks for translation, proof completion, and long-horizon development. We therefore do not claim that FUG is the first held-out metric, or that a use graph is novel merely because it is represented as a DAG.

The relevant separation is between two kinds of holdout. In FUSE, the revision-use descriptions and execution outcomes can guide a candidate's statement, while the held-out uses provide no such signal and are evaluated after the candidate is selected. The held-out object is a mathematical use family that imports and relies on the shared statement, not merely another natural-language prompt sampled from the same distribution. This makes it possible to ask whether a representation choice survives a change in the construction, proof obligations, or dependency path. FUG summarizes this property across targets, while the appendix records compatibility, adaptation burden, proof status, and the major Mathlib-refactor comparisons separately.

This positioning also explains why our baselines matter. Direct and descriptions-only generation measure what can be obtained without downstream feedback; Binary-FUSE measures the familiar generate–feedback–repair loop with a fixed statement; and FUSE measures whether downstream evidence can justify a shared statement revision whose benefits survive replay and held-out evaluation. The baselines are consequently comparisons of information access and revision scope, rather than claims that prior self-refinement or theorem proving systems use an identical implementation.