
What Has Actually Been Proved? Auditable Paper-to-Lean Alignment with LeanTrail

Fabricio Venturim

Fundação Getulio Vargas
Rio de Janeiro, Brazil
fabricio.venturim@fgv.br

Leighton Estrada

Fundação Getulio Vargas
Rio de Janeiro, Brazil
leighton.rayme@fgv.br

Claudio Silva

New York University
New York, USA
csilva@nyu.edu

Jorge Poco

Fundação Getulio Vargas
Rio de Janeiro, Brazil
jorge.poco@fgv.br

Abstract

Compiler feedback certifies that Lean accepts a declaration, not that the declaration captures the intended result: a repository can compile while a central theorem is weakened, merely auxiliary, or transitively dependent on `sorry`. Formalization progress should instead be measured against the source paper, in terms of its *paper items*: definitions, lemmas, and theorems. *LeanTrail* implements this view as a post-hoc, tool-agnostic audit: deterministic retrieval proposes candidate counterparts for each paper item in the Lean repository and, when available, the Blueprint; a language model judges statement-level correspondence, only among those candidates; and proof completeness comes from a deterministic transitive dependency audit never exposed to the model. Fusion assigns each item one of five coverage states, from *Formalized* to *Missing*, each backed by inspectable evidence. On consensus annotations of 98 items from three Lean projects, with three runs of each of two judge models, *LeanTrail* assigns the correct coverage state to 91.8% (Gemini 3.8 Flash) and 92.5% (GPT-6 Sol) of items. 96–97 of the 98 states are identical across runs, and GPT-6 Sol never reports a *Partial* item as *Formalized*. Uncovered items form a natural task queue for formalization agents, anchored to the specification rather than the compiler.

1 Introduction

Formalization projects in proof assistants such as Lean [5], supported by libraries such as *mathlib* [17], have become a practical medium for verifying research mathematics. Large language models increasingly translate informal statements into formal ones [20], guide proof search [8], and act as agents that iterate against the compiler [22]. As producing compilable Lean code becomes easier, a second bottleneck emerges: determining whether the intended mathematics has actually been formalized.

Compiler feedback certifies that Lean accepts a declaration, not that the declaration is faithful to the source paper: a repository can compile while a central theorem depends transitively on `sorry`, or while a declaration proves something weaker or merely auxiliary. Checking progress therefore requires reasoning across artifacts: Is Theorem 2 actually proved? Does its Lean statement match the paper’s? These judgments matter in human–AI workflows, where mathematicians delegate mechanical work but keep mathematical control [3].

Existing systems address parts of this workflow. *Lea* [19] supports mathematician-led formalization and audits generated Lean artifacts; Blueprints [10, 16] and *LeanArchitect* [24] connect a formal-

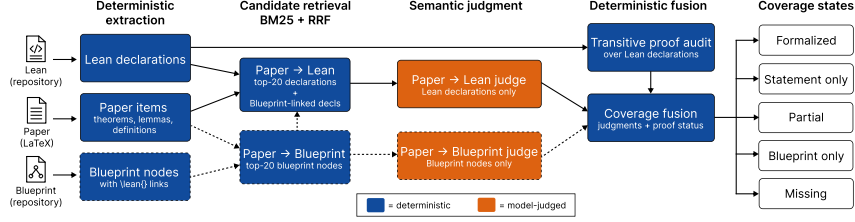


Figure 1: The *LeanTrail* pipeline. Blue stages are deterministic: extraction, candidate retrieval, the transitive proof audit, and the final fusion. The only model-judged stages (orange) label the statement-level semantic relation of retrieved candidates; dashed elements require a Blueprint.

ization plan to formal declarations; *Lean Atlas* [21] scopes human semantic review over the Lean dependency graph, and *KnowTeX* [18] exposes dependencies within the $\text{\LaTeX}$ source. All are organized around declarations or plan nodes, and none answers the question a project targeting an existing paper raises: which of the paper’s results are formalized? Declarations do not say which paper items they cover, and a Blueprint is a re-authored plan that can drift from the paper it describes. Autoformalization metrics instead judge a given pair: BEq+ [14] compares a generated formal statement with a reference formal one, and BPF [11] scores how faithfully a formal candidate encodes a natural-language statement by probing its consequences. *LeanTrail* must first find the candidate among a repository’s declarations and combine its judgment with the proof audit; a certifier such as BPF could replace or check that judgment. We argue that formalization progress should be measured against the paper itself, in terms of its paper items: definitions, lemmas, propositions, and theorems.

LeanTrail answers that question with a post-hoc audit of any paper and repository (Figure 1), built to keep apart two questions that progress reports usually conflate. Whether a Lean statement says what the paper says is a semantic question; a language model answers it, but only over a short list of deterministically retrieved candidates. Whether that statement is proved is a mechanical one; we answer it by walking the elaborated dependency graph for `sorryAx`, and the model never sees the result. Fixed rules fuse the two answers into one of five coverage states per item, each shown with the candidates it rejected. The split has a concrete consequence: a model error can mislabel how faithful a statement is, but cannot turn an unproved result into a proved one. Five states rather than a percentage then tell a contributor what to do next.

Our contributions are: (i) a paper-level formulation of formalization progress that is auditable by construction, since the model proposes and classifies but never certifies a proof; (ii) a pipeline that keeps deterministic retrieval, model-judged correspondence, and deterministic proof auditing separate; (iii) an interface for inspecting and contesting paper-to-Lean coverage; and (iv) an evaluation on 98 consensus-annotated items from three projects, with two judge models and three runs each: *LeanTrail* recovers the correct state for 91.8–92.5% of items and repeated runs agree on 96–97 of 98 states, with the annotations released.

2 Alignment and Coverage Classification

LeanTrail converts a formalization project into a precomputed paper-level representation: for each item in the paper, it records candidate Lean declarations and, when available, Blueprint nodes, with one coverage state. Figure 1 summarizes the pipeline: deterministic stages (blue) extract entities, retrieve candidates, audit proof completeness, and fuse the results, while the only model-judged stages (orange) classify semantic correspondence and can never certify that a proof is complete.

Extraction. From the paper, *LeanTrail* extracts theorem-like $\text{\LaTeX}$ environments (theorems, lemmas, propositions, definitions, including paper-defined variants), recording each item’s kind, label, statement, and source location. From the Lean repository, it indexes declarations, signatures, source ranges, and the constants referenced by each elaborated declaration, which form the declaration-dependency graph used by the proof audit. From the optional Blueprint, it extracts nodes, statements, dependencies, and explicit links to Lean declarations (details in Appendix B).

Candidate retrieval. Rather than asking the model to examine an entire repository, *LeanTrail* narrows each search space deterministically, as in retrieval-augmented generation [9] and premise selection [22]. Alignment follows two routes, Paper→Lean over declarations and Paper→Blueprint over plan nodes; each fuses BM25 textual rankings [15] with identifier similarity via reciprocal rank

Paper item	Blueprint	Lean candidate	Proof audit	Coverage state	Next action
Theorem 1	equivalent	int • partial	contains sorry	Partial	fix statement
Lemma 7	partial	unbalancing' • equivalent	reaches sorryAx	Statement only	finish proof
Proposition 19	equivalent	no reliable match	—	Blueprint only	implement

■ model-judged ■ deterministic

Figure 2: Three APAP paper items and the evidence behind their coverage states in a Gemini 3.8 Flash run, as read from the interface (Figure 3, Appendix A): semantic relations are model-judged (orange), the proof audit deterministic (blue).

fusion [4] and retains the top 20 candidates, the smallest cutoff we tested at which every annotated counterpart is retrieved on both routes (Table 1). This is an empirical choice for these projects, not a guarantee for larger repositories, where the cutoff can be raised. For each retrieved Blueprint node with an explicit Lean link, the linked declaration is added to the Lean candidate pool; these links improve recall, but their provenance is hidden from the Lean judge.

Semantic judgment. Two independent model calls, one per route, classify the retained candidates as *Equivalent*, *Partial*, or *None*. *Equivalent* denotes the same substantive result, allowing representation changes or direct instantiation. *Partial* denotes the same intended result with a material mismatch in hypotheses, domain, parameters, or conclusion. *None* covers proof ingredients, related results, and unrelated results. The model receives statement-level information only: declaration bodies, proof-completeness data, and cross-route evidence are withheld, and it neither searches the repository nor decides whether a proof is complete. The prompt policy, response schema, and exact templates are given in Appendix C.

Proof audit. In parallel with semantic judgment, *LeanTrail* propagates `sorryAx` deterministically through the elaborated declaration-dependency graph, so a declaration counts as incomplete even when a `sorry` enters only through its dependencies. If a declaration’s status cannot be resolved from the graph alone, a targeted `#print axioms` certificate settles it before fusion. Besides `sorry`, a proof can rest on an *axiom*, a statement that a project asserts without proof and that Lean accepts just as silently. The certificate names every axiom a declaration depends on; in the evaluated projects, no project declares an axiom of its own, and every certificate lists only the three axioms of Lean’s standard foundations (`propext`, `Classical.choice`, `Quot.sound`), which *mathlib* itself assumes. The model never sees the outcome of this audit, so whether a proof is complete cannot sway its judgment of whether a statement matches the paper.

Fusion into coverage states. The final state combines semantic correspondence with proof status; a *local* match is a declaration in the project repository itself. *Formalized* requires an equivalent local declaration whose proof (or implementation, for definition-like items) is complete; *Statement only*, an equivalent Lean proposition whose audit is incomplete or unknown; *Partial*, a partial relation on either route with no equivalent local match; *Blueprint only*, an equivalent Blueprint node without a validated local match; and *Missing*, no equivalent or partial match on either route. Applied in this order, the rules give each item exactly one state.

3 Inspecting the Alignment

Coverage states are only useful if a user can check them, so the interface makes each decision inspectable and open to challenge. From a paper item, a user reaches its Lean counterpart and proof status, the retained candidates with their judged relations and rationales, and a coverage table over all items, so a questionable state can be traced to the judgment or audit that produced it, including the rejected alternatives. A *Formalized* state therefore still needs checking: the proof is verified mechanically, but whether the Lean statement matches the paper is the model’s judgment. Figure 3 (Appendix A) shows these views on the example below.

Figure 2 shows why the five states are worth separating, on APAP (Arithmetic Progressions and Almost Periodicity) [2, 6], a formalization of Kelley–Meka-type bounds for sets free of three-term arithmetic progressions. All three items have a Blueprint counterpart (equivalent for Theorem 1 and Proposition 19, partial for Lemma 7), yet end in different states and call for different next actions.

Table 1: Performance across the three evaluated stages of *LeanTrail* (%). N_P and N_L denote paper items and indexed Lean declarations, and N_+ the positive queries per route; $@k$ is query-level recall at cutoff k . Judgment and coverage accuracies are means over three runs of each judge model, with the standard deviation in the overall row.

(a) Candidate retrieval													
Project	N_P	N_L	N_+	Paper→Lean					Paper→Blueprint				
				@1	@5	@10	@15	@20	N_+	@1	@5	@10	@15 @20
FinEqs	7	11	6	50.0	100.0	100.0	100.0	100.0	7	100.0	100.0	100.0	100.0
APAP	29	929	12	50.0	75.0	83.3	91.7	100.0	24	83.3	95.8	100.0	100.0
Ringel	62	396	17	41.2	70.6	82.4	94.1	100.0	6	50.0	83.3	83.3	100.0
Overall	98	1,336	35	45.7	77.1	85.7	94.3	100.0	37	81.1	94.6	97.3	100.0

(b) Semantic judgment and coverage						
Project	Gemini 3.8 Flash			GPT-6 Sol		
	Judgment		Coverage	Judgment		Coverage
	Paper→Lean	Paper→Blueprint	5-state	Paper→Lean	Paper→Blueprint	5-state
FinEqs	100.0	100.0	100.0	85.7	100.0	85.7
APAP	87.4	86.2	87.4	82.8	79.3	82.8
Ringel	93.0	96.8	93.0	97.8	93.5	97.8
Overall	91.8±1.0	93.9±0.0	91.8±1.0	92.5±0.6	89.8±0.0	92.5±0.6

The differences are substantive: `int` quantifies $c > 0$ after A and N , allowing the constant to depend on them, whereas the paper requires an absolute constant; `unbalancing` states *Lemma 7* faithfully, but its transitive dependency closure reaches `sorryAx`. No single progress number distinguishes these cases, and §4 measures whether *LeanTrail* recovers the distinction reliably.

4 How Reliable Is the Alignment?

Evaluation setup. To measure how often assigned states are wrong, and in which direction, we evaluate all three stages, retrieval, semantic judgment, and coverage classification, against an item-level ground truth at fixed commits over three paper–repository pairs: FinEqs [1, 13], on reducing the number of equations defining a subset of n -space over a finite field (7 items, 11 indexed Lean declarations); APAP [2, 6] (29 items, 929 declarations); and Ringel [12, 7], on decomposing complete graphs into copies of a fixed tree (62 items, 396 declarations). All 98 items come from explicit statement environments, checked against the source papers before annotation.

Two annotators familiar with Lean jointly reviewed every item against the repository snapshot and, when available, the Blueprint and project mappings, recording by consensus the corresponding Lean declaration, if any, its relation to the paper statement, and the coverage state. Project links were treated as evidence rather than labels, and proof completeness came from the transitive audit of §2, not from manual judgment. The ground truth comprises 7 *Formalized*, 5 *Statement only*, 23 *Partial*, 13 *Blueprint only*, and 50 *Missing* items (protocol and rationales in Appendix D, released with the supplementary material).

Table 1 reports each stage: query-level retrieval recall@ k , where a query is positive when the ground truth names a counterpart on that route; semantic-judgment accuracy over *Equivalent*, *Partial*, and *None*; and five-state coverage accuracy. We run the semantic judgment with Gemini 3.8 Flash and GPT-6 Sol, three times each on identical inputs, and report means over the runs (configuration in Appendix C). Appendix E defines the metrics.

Results. At $k = 20$, retrieval recovered an annotated counterpart for every positive query on both routes, although recall at $k = 1$ is low (45.7% and 81.1%), so candidates reach the judge as a pool. Paper→Lean judgment reached 91.8% accuracy (84.1% macro-F1) with Gemini 3.8 Flash and 92.5% (80.7%) with GPT-6 Sol, and Paper→Blueprint 93.9% (84.7%) and 89.8% (77.7%). After fusion with the proof audit, *LeanTrail* assigned the correct coverage state to 91.8% of items (five-state macro-F1 83.9%) with Gemini and 92.5% (77.6%) with GPT-6 Sol, and separating any coverage from *Missing* was correct for at least 97 of 98 items in every run. Repeated runs agree on 96 (Gemini) and 97 (GPT-6 Sol) of the 98 states, so the remaining errors are systematic rather than sampling noise.

Failure modes. We separate errors by direction. An error *over-claims* when it reports more coverage than annotated, such as a *Partial* item as *Formalized* or *Statement only*, and *under-claims* when it reports less. Over-claiming is the harmful direction for an audit, since a user may then trust an unfaithful statement. Gemini 3.8 Flash over-claims on 4.4% of items per run and under-claims on 3.7%. Of its *Formalized* reports, 17.9% are wrong. GPT-6 Sol never over-claims: every *Formalized* report is correct and every *Partial* item detected. All its errors, 7.5% of items per run, under-claim faithful items as *Partial* for differences the annotators judged immaterial, such as a rounded threshold. As a result, it recognizes 52.4% of *Formalized* items, against 85.7% for Gemini. The models thus differ in the direction of their errors rather than in accuracy: GPT-6 Sol leads on Ringel, whose partial matches hide subtle mismatches, and trails on APAP and FinEqs, whose faithful matches carry harmless restrictions.

5 Toward an Agentic Formalization Loop

LeanTrail turns every non-*Formalized* paper item into a structured, agent-ready task; the required action follows from the coverage state: *Missing* calls for a new formalization, *Blueprint only* for implementing a planned result, *Statement only* for completing a proof, and *Partial* for repairing a mismatch or accepting an intentional deviation. Each task bundles the evidence behind its state with a completion criterion and is exported as JSON for downstream agents (Appendix F).

Rerunning the audit after repository changes, such as an agent’s edits, recomputes each task against the same specification. Judgments are reused while the paper statement and the chosen counterparts are unchanged, while candidates and proof status are recomputed on every rerun. Discharging the last transitive `sorryAx` dependency of an equivalent counterpart moves an item from *Statement only* to *Formalized*; unrelated declarations move nothing. Coverage states therefore provide a progress signal that compilable code alone cannot satisfy. Agent loops such as the propose–judge–update cycle of AlphaApollo [23] verify intermediate solutions with tools, which checks the agent’s own work but not whether the result is the one the paper states. Coverage states add that check, so agents act on paper-grounded objectives, and mathematicians see why each task was generated and whether it was completed.

6 Limitations and Conclusion

Limitations. Our evaluation covers three projects and 98 items, with few examples in some states, and the ground truth was built by joint consensus, so we cannot report inter-annotator agreement. Semantic correspondence remains model-judged, and its residual errors depend on the model: Gemini 3.8 Flash still reports some *Partial* items as more complete states, while GPT-6 Sol reports some faithful ones as *Partial*. A *Formalized* state should therefore be confirmed by a person before it is relied on. The proof audit tracks only `sorry`: a proof resting on an axiom declared by the project would still count as complete, although its certificate would name that axiom. The ground truth takes proof status from the same audit, so this stage is not validated independently. Extraction assumes structured $\text{\LaTeX}$ sources. *LeanTrail* also treats the paper as a fixed specification; when formalization uncovers new results or requirements, as in software verification, declarations that no paper item covers could instead propose revisions to the paper or Blueprint.

Conclusion. *LeanTrail* treats the source paper as the specification a formalization must meet, keeps proof completeness out of the model’s hands, and recovers the correct coverage state for 91.8–92.5% of items across two judge models and repeated runs. Re-audited as the repository changes, uncovered results become paper-grounded tasks for formalization agents: progress means the mathematics is complete, not that Lean code compiles.

Acknowledgments and Disclosure of Funding

This work was financed in part by the Brazilian National Council for Scientific and Technological Development (CNPq), grant # 313454/2026-4; the Coordination for the Improvement of Higher Education Personnel (CAPES), Finance Code 001; and the Getulio Vargas Foundation (FGV). Silva was supported by NSF grants OAC-2411221 and DRL-2604563, and by the DARPA expMath program (Agreement No. HR0011262E029). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA, or the authors’ affiliated institutions. The authors declare no competing interests.

References

- [1] Stefan Barańczuk. Reducing the number of equations defining a subset of the n -space over a finite field. *Annales de la Faculté des Sciences de Toulouse: Mathématiques*, 33(1):177–182, 2024. doi: 10.5802/afst.1766.
- [2] Thomas F. Bloom and Olof Sisask. The Kelley–Meka bounds for sets free of three-term arithmetic progressions. *Essential Number Theory*, 2(1):15–44, 2023. doi: 10.2140/ent.2023.2.15.
- [3] Katherine M. Collins, Simon Frieder, Jonas Bayer, Jacob Loader, Jeck Lim, Peiyang Song, Fabian Zaiser, Lexin Zhou, Shanda Li, Sam Looi, Joshua B. Tenenbaum, Umang Bhatt, Adrian Weller, Jose Hernandez-Orallo, Cameron E. Freer, Valerie Chen, and Ilia Sucholutsky. Characterizing initial human–AI proof formalization workflows. arXiv preprint arXiv:2606.04273, 2026.
- [4] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Büttcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR ’09, pages 758–759, 2009. doi: 10.1145/1571941.1572114.
- [5] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635, 2021. doi: 10.1007/978-3-030-79876-5_37.
- [6] Yaël Dillies and contributors. APAP: Arithmetic progressions and almost periodicity. Lean 4 repository, <https://github.com/YaelDillies/APAP>, 2026. Evaluated at commit 2daecd9.
- [7] Walid Elkersh and contributors. Ringel’s conjecture in Lean 4. Lean 4 repository, <https://github.com/walidelkersh/Ringel>, 2026. Evaluated at commit 7fc71bb.
- [8] Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023.
- [9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.
- [10] Patrick Massot. Lean blueprints. GitHub repository, <https://github.com/PatrickMassot/leanblueprint>, 2020. Accessed 2026-09-01.
- [11] Noor Islam S. Mohammad and Tamim Sheikh. The faithfulness gap: Certifying semantic equivalence between natural-language and formal mathematical statements. arXiv preprint arXiv:2606.16541, 2026.
- [12] Richard Montgomery, Alexey Pokrovskiy, and Benny Sudakov. A proof of Ringel’s conjecture. *Geometric and Functional Analysis*, 31(3):663–720, 2021. doi: 10.1007/s00039-021-00576-2.
- [13] Bartosz Naskręcki, Pietro Monticone, and contributors. FinEqs: A Lean formalization of reducing the number of equations over a finite field. Lean 4 repository, <https://github.com/nasqret/fineqs>, 2025. Evaluated at commit c177541.
- [14] Auguste Poiroux, Gail Weiss, Viktor Kunčák, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17947–17969. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.907.

- [15] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009. doi: 10.1561/15000000019.
- [16] Terence Tao. Formalizing the proof of PFR in Lean 4 using Blueprint: A short tour. Blog post, <https://terrytao.wordpress.com/2023/11/18/formalizing-the-proof-of-pfr-in-lean4-using-blueprint-a-short-tour/>, November 2023. Accessed 2026-09-01.
- [17] The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, 2020. doi: 10.1145/3372885.3373824.
- [18] Elif Uskuplu, Lawrence S. Moss, and Valeria de Paiva. KnowTeX: Visualizing mathematical dependencies. arXiv preprint arXiv:2601.15294, 2026.
- [19] VIDA-NYU. Lea: An agent backbone for mathematician-led formalization. Project website, <https://vida-nyu.github.io/Lea/>, 2026. Accessed 2026-09-01.
- [20] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 32353–32368, 2022.
- [21] Banri Yanahama and Akiyoshi Sannai. Lean Atlas: An integrated proof environment for scalable human-AI collaborative formalization. arXiv preprint arXiv:2604.16347, 2026. 1st Workshop on AI, Proof and Verification (AIPV 2026), co-located with FM 2026.
- [22] 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. In *Advances in Neural Information Processing Systems*, volume 36, pages 21573–21612, 2023. Datasets and Benchmarks Track.
- [23] Zhanke Zhou, Chentao Cao, Xiao Feng, Xuan Li, Zongze Li, Xiangyu Lu, Jiangchao Yao, Weikai Huang, Tian Cheng, Jianghangfan Zhang, Tangyu Jiang, Linrui Xu, Yiming Zheng, Brando Miranda, Tongliang Liu, Sanmi Koyejo, Masashi Sugiyama, and Bo Han. AlphaApollo: A system for deep agentic reasoning. arXiv preprint arXiv:2510.06261, 2025.
- [24] Thomas Zhu, Pietro Monticone, Sean Welleck, and Jeremy Avigad. LeanArchitect: Automating blueprint generation for humans and AI. In Ekaterina Komendantskaya and Tobias Nipkow, editors, *17th International Conference on Interactive Theorem Proving (ITP 2026)*, volume 382 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, 2026. doi: 10.4230/LIPIcs.ITP.2026.25.

A Interface: Paper View and Additional Views

The paper view described in §3 is shown in Figure 3. It is one of three coordinated views; the other two are repository-centered and support orientation rather than coverage inspection.

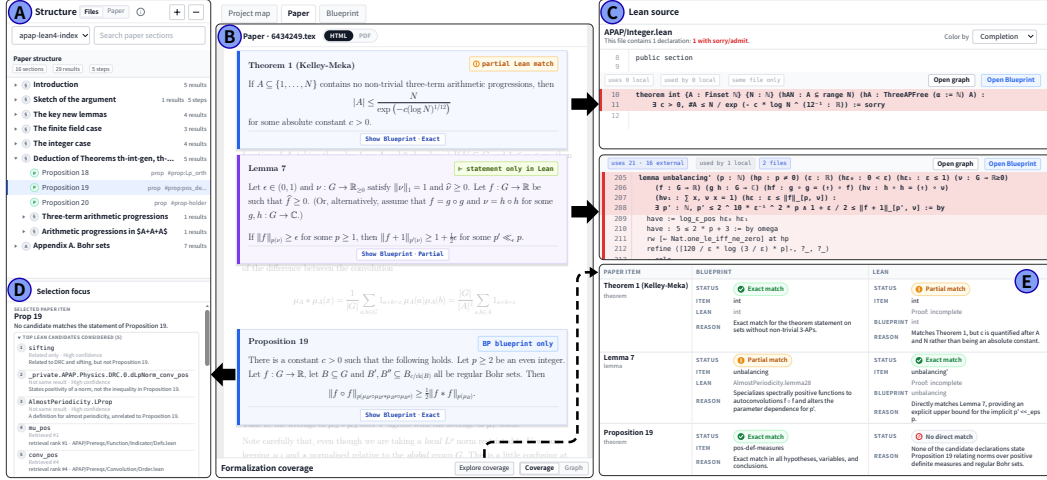


Figure 3: The paper view on the APAP formalization. Extracted paper items are listed in the navigator (A) and inspected in the context of the paper (B). The Lean panels show the selected declarations and their proof status (C); the focus panel exposes candidate judgments with relations and rationales (D); and the coverage table summarizes paper items against Blueprint nodes and Lean declarations (E). Overlaid arrows link each paper item to its Lean counterpart and to its row in the coverage table.

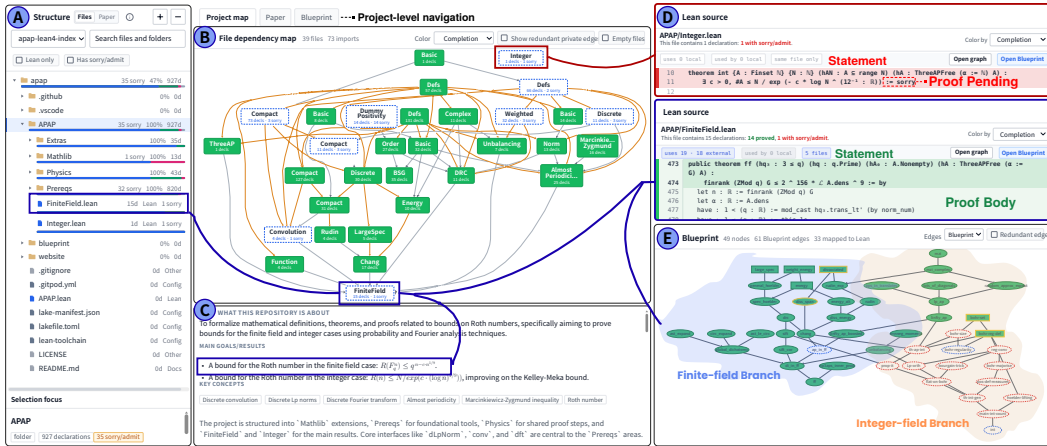


Figure 4: The *LeanTrail* interface on the APAP project, focusing on the project-map and Blueprint tabs. The structural navigator (A), project map (B), repository summary (C), Lean source panel (D), and Blueprint view (E) operate as coordinated views over repository structure, formal code, and the formalization plan. In the project map (B), orange edges denote public import (re-exported dependencies) and gray edges ordinary import; arrows point from the imported dependency to the importing file. Node colors encode proof completion.

Structural navigator. The navigator (Figure 4A) operates in two modes. In *Files* mode, users browse the repository’s directories and files. In *Paper* mode, it lists items extracted from the paper, so exploration can begin from the mathematical text without knowing where each result lives in the code.

Search locates files, declarations, or paper items by name or text fragment, and selections propagate to the other views.

Project map. The project map (Figure 4B) combines the file-import graph with a repository summary (Figure 4C). The graph exposes central modules, peripheral regions, and files that concentrate incomplete work. The summary complements this structure with the repository’s main concepts, thematic areas, and possible entry points for reading. Selecting a node reveals its dependencies and opens the corresponding source.

Blueprint view. The Blueprint view (Figure 4E) presents the formalization plan, when available, as a navigable artifact. Users can inspect plan nodes, statements, dependencies, and their relations to paper items and Lean declarations. Because the Blueprint lies between paper and code, this view distinguishes nodes already linked to code from those that remain planned or whose correspondence is uncertain.

Code panel and coordinated selection. The code panel anchors exploration in Lean source: when a selected element has a Lean match, the panel opens its file and highlights the declaration together with its proof-audit status. All views share a common selection mechanism, so a user can start from the map, a paper item, or a Blueprint node and arrive at the corresponding source range.

B Pipeline and Data Model Details

The interface consumes precomputed JSON artifacts rather than inferring alignments interactively, so the coverage table, candidate evidence, proof status, and source panels all read from the same representation. In addition to the entities described in §2, extraction computes the declaration-reference and file-import graphs, per-file proof-audit summaries, and source-navigation indices.

Cards and declaration roles. Paper items, Blueprint nodes, and Lean declarations are normalized into *cards* containing prose, notation, concepts, identifiers, kinds, files, and source locations. Proposition-valued declarations receive an explicit role: a *statement definition* such as `def Foo : Prop := P` represents the proposition *P* but does not by itself certify a proof, whereas a *proof-producing* declaration or another term whose type is a proposition may certify one when its dependency closure is complete.

Paper-item extraction. LeanTrail recursively expands `\input` and `\include` files and recognizes both standard and paper-defined mathematical-statement environments declared with `\newtheorem`. It retains each item’s kind, label, statement, section, and source location. The extractor can also conservatively promote independently labeled equations or explicit textual consequences, although none entered this evaluation: all 98 evaluated items came from explicit statement environments. Re-running extraction on the evaluated sources reproduced the same 7, 29, and 62 unique identifiers for FinEqs, APAP, and Ringel, respectively; each occurs exactly once in the corresponding ground truth. The inventories were additionally checked against the source papers before annotation.

C Semantic Judgment Configuration

We evaluate the semantic judgment with two models and the same prompt: `gemini-3.8-flash` with medium thinking, and `gpt-6-sol` with medium reasoning effort. Both return schema-constrained JSON with no explicit temperature override. Each paper item produces one Paper→Lean call and one Paper→Blueprint call. The Lean route receives the paper item and Lean statement cards; the Blueprint route receives the paper item and Blueprint statement cards. Declaration bodies, proof-completeness fields, Blueprint provenance in the Lean route, and Lean-link evidence in the Blueprint route are rejected by a recursive whitelist check before prompting.

Diagnostic labels. Candidate judgments use *exact statement*, *representation-equivalent*, *stronger direct instance*, *partial statement*, *supporting only*, *related only*, *not the same result*, or *no reliable match*; the route-level recommendation may additionally be *ambiguous*. For evaluation and fusion,

exact, representation-equivalent, and stronger direct instances collapse to *Equivalent*; partial statements collapse to *Partial*; and supporting, related, rejected, absent, or ambiguous matches collapse to *None*. Up to five candidate judgments per route are persisted with confidence labels and rationales.

Decision policy supplied to the model. The prompt instructs the model to judge statement-level correspondence only and not to force a match. A faithful-family label requires agreement of substantive hypotheses, quantifiers, domains, parameter restrictions, and conclusions. A candidate known to target the same result is *partial statement* if any such agreement fails or remains materially uncertain; in particular, proving $A \rightarrow B$ when the paper states B is partial when A is substantive. Before a faithful-family label, the model audits the candidate: it names the paper counterpart of every Lean hypothesis and every parameter restriction, and the Lean counterpart of every paper hypothesis and every conclusion; unfolds local predicates whose definitions appear among the candidates and compares their thresholds with the paper; and checks quantifier order, what each constant may depend on, and whether the hypotheses can hold together. A candidate that proves only part of the paper result, such as one direction, one side of a bound, or one case, is *partial statement*. A different ingredient or sublemma is *supporting only*, not partial. Differences of representation, for example sets versus predicates or bundled versus unbundled parameters, may be representation-equivalent when the mathematical content agrees. The rationale must identify the concrete representation mapping and why it preserves the substantive statement conditions. For a stronger direct instance, it must specify the instantiation that recovers the paper statement without an additional substantive hypothesis. A proposition definition may be semantically equivalent to the paper statement, but whether it has been proved is deliberately deferred to deterministic fusion.

Prompt template. The template below summarizes the *LeanTrail Alignment Prompt* used in the reported evaluation, including its route-specific headers and shared decision instructions. Exact fixed templates, including the output-format instructions, are supplied as `paper_to_lean_prompt.txt` and `paper_to_blueprint_prompt.txt`; `response_schema.json` contains the API schema. Each template’s final placeholder is replaced by the route-specific JSON packet described below.

LeanTrail alignment prompt (summary; verbatim templates in the supplementary material)

Paper->Lean header:

You are judging ONLY the Paper->Lean route for one mathematical paper item.

Question: Does the paper item directly match any local Lean declaration candidate?

- Compare the paper statement to Lean names/signatures only.
- Ignore Blueprint planning evidence in this route.
- A declaration may have entered through a deterministic link, but link provenance is hidden and is not semantic evidence.

Paper->Blueprint header:

You are judging ONLY the Paper->Blueprint route for one mathematical paper item.

Question: Does the paper item match any Blueprint node candidate?

- Compare the paper statement to Blueprint titles/statements only.
- Do not judge the linked Lean code in this route.
- A Blueprint->Lean link is metadata for later fusion, not evidence that the Paper->Blueprint alignment is correct.

Common instructions:

Classify the statement-level relationship between the paper item and each candidate. Judge mathematical statements only. Proof completeness is unavailable and handled by a deterministic stage.

Do not force a match: `no_reliable_match` and `ambiguous` are valid.

Safety policy:

- Assign `exact_statement`, `representation_equivalent`, or `stronger_direct_instance` only when all substantive hypotheses, quantifiers, domains, parameter restrictions, and conclusions agree.
- If a candidate clearly targets the same result but any such agreement fails or remains materially uncertain, use `partial_statement`.
- Use `ambiguous` when the identity of the target is unresolved.
- Use `supporting_only` only for a genuinely different ingredient, sublemma, or result.

Audit before a faithful-family label:

1. Lean -> paper: name the paper condition matching every Lean hypothesis and parameter restriction; an unmatched hypothesis is additional unless it is an instance argument or follows from the setting.
 2. Paper -> Lean: name the Lean counterpart of every paper hypothesis and part of the conclusion.
 3. Local definitions: unfold project predicates defined among the candidates and compare every threshold with the paper; a matching name is not enough.
 4. Quantifiers and constants: check quantifier order and what each constant may depend on; a statement whose hypotheses cannot hold together, or that is trivially true, does not state the result.
 5. The candidate states the paper result itself rather than only a proof ingredient.
- If any step fails or remains materially uncertain for a same-target candidate, use `partial_statement`.

Definitions:

- `exact_statement`: same result and representation.
- `representation_equivalent`: same mathematical content in a different formal representation.
It never changes which objects are allowed, a threshold, or the conditions defining a case.
- `stronger_direct_instance`: genuinely stronger or more general; the paper item follows by direct instantiation without a new substantive argument.
Use it only when the instantiation can be stated; if it changes what a constant may depend on, use `partial_statement`.
- `partial_statement`: same intended target, but weakened, narrowed, specialized, incomplete, or subject to an additional substantive hypothesis.
Proving only one direction, one side of a bound, or one case is `partial_statement`, not `no_reliable_match`.
- `supporting_only`: useful for the proof, but not the result itself.
- `related_only`: shared topic or vocabulary only.
- `not_same_result`: unrelated or incorrect candidate.
- `no_reliable_match`: no candidate is reliable enough.

Additional rules:

- Type or container differences are not mismatches by themselves.
- Do not call a proof ingredient `partial_statement`.
- If the paper states B and the candidate proves A -> B for a substantive A absent from the paper, use `partial_statement`.
- A definition ``def Foo : Prop := P`` may match P semantically; whether P is proved is decided after semantic judgment.
- Retrieval scores are weak hints only.

Return only schema-valid JSON. Keep at most five candidate judgments and each rationale under 20 words.

For `representation_equivalent`, use the rationale to identify the concrete representation mapping and why it preserves the substantive statement conditions.

For `stronger_direct_instance`, use the rationale to specify the instantiation that recovers the paper statement without an additional substantive hypothesis.

Input:

<route-specific JSON packet>

Input packet schema. Each judgment call receives a single JSON packet containing the paper item and its retained candidates. The packet below shows the Paper→Lean form.

Input packet (Paper→Lean route)

```
{
  "paper_item_id": "...",
  "display_name": "...",
  "route": "paper_to_lean | paper_to_blueprint",
  "paper_item": {
    "kind": "...",
    "label": "...",
    "title_text": "...",
    "section_path": "...",
    "statement_latex": "..."
  },
  "candidate_bundles": [
    {
      "candidate_id": "...",
      "candidate_source": "lean | blueprint",
      "candidate_type": "...",
      "lean": {
        "decl_id": "...",
        "full_name": "...",
        "kind": "...",
        "declaration_role": "..."
      }
    }
  ]
}
```

```

    "file_path": "...",
    "signature": "...",
    "defined_proposition": "..."
  }
}
]
}

```

For the Blueprint route, each candidate bundle replaces the lean object with a blueprint object containing `blueprint_id`, `label`, `title`, `kind`, `retrieval rank` and `score`, and `statement_latex`. Route-inappropriate fields are absent rather than masked with placeholder values.

Response schema. The model must return JSON conforming to the schema below, which the API enforces; responses that violate it are rejected and retried.

Schema-constrained model response

```

{
  "paper_item_id": "string",
  "final_recommendation": {
    "best_judgment":
      "exact_statement | representation_equivalent |
      stronger_direct_instance | partial_statement |
      supporting_only | no_reliable_match | ambiguous",
    "best_candidate_ids": ["candidate_id"],
    "rationale": "short reason"
  },
  "candidate_judgments": [
    {
      "candidate_source": "lean | blueprint | structural",
      "candidate_id": "candidate_id",
      "judgment":
        "exact_statement | representation_equivalent |
        stronger_direct_instance | partial_statement |
        supporting_only | related_only | not_same_result |
        no_reliable_match",
      "confidence": "high | medium | low",
      "rationale": "short reason"
    }
  ]
}

```

D Annotation Protocol

Two annotators familiar with Lean jointly reviewed all 98 items from FinEqs, APAP, and Ringel against the corresponding papers and repository snapshots and, when available, the project Blueprints and author-provided mappings. The evaluated commits are c177541 for FinEqs, 2daecd9 for APAP, and 7fc71bb for Ringel. Project-provided links were treated as evidence rather than labels: each proposed match was checked against the paper statement and Lean signature. For each route, the annotators recorded the corresponding candidate identifier or none and the relation *Equivalent*, *Partial*, or *None*; ambiguous and partial decisions include short rationales. All decisions were reached jointly by consensus.

The resulting ground truth contains 7 *Formalized*, 5 *Statement only*, 23 *Partial*, 13 *Blueprint only*, and 50 *Missing* items.

Faithfulness criterion. *Equivalent* means that the candidate states the same mathematical result, allowing definitional unfolding, notation changes, and mathematically interchangeable representations. A genuinely stronger or more general theorem also counts as equivalent when the paper item follows by direct instantiation without a substantive new argument. *Partial* means that the candidate targets the same result but adds a substantive hypothesis, omits a hypothesis or conclusion, strengthens a parameter restriction, specializes the domain, or otherwise covers only part of the paper statement. A result that is merely useful in a proof, or only shares vocabulary, is *None*.

Coverage-state derivation. For theorem-like paper items, *Formalized* requires an equivalent local match and a complete proof-producing declaration; an equivalent proposition definition without a proof is *Statement only*. Definition-like paper items may be *Formalized* by a complete implementing declaration. An equivalent selected proof whose audit is incomplete or unknown is *Statement only*. In the absence of an equivalent local Lean match, a partial match on either route yields *Partial*; otherwise an equivalent Blueprint node without a validated local Lean match yields *Blueprint only*. If none of these conditions holds, the item is *Missing*. Proof completeness is derived from the transitive audit and targeted Lean certificates rather than from semantic annotation.

E Evaluation Details

Run provenance. Semantic and coverage results come from six evaluation runs of this prompt on all 98 items, three with `gemini-3.8-flash` and three with `gpt-6-sol`, each with 196 completed route judgments and no final route errors. The Gemini runs used 198, 199, and 196 provider requests, retrying five invalid structured responses in total; the GPT-6 Sol runs used 196 requests each. Recorded usage was 5.86M prompt and 0.75M output-plus-thinking tokens for Gemini, and 5.42M and 0.36M for GPT-6 Sol. The same extracted inventories, retrieval candidates, gold annotations, proof evidence, and fusion rules were used for all projects and runs. The run and artifact hashes are recorded in the supplementary `run_manifest.json`.

Metrics. Candidate retrieval is evaluated independently on the Paper→Lean and Paper→Blueprint routes. A positive query is recovered when at least one annotated counterpart appears among its first k candidates; items whose reference relation is *None* are excluded from this denominator. We report query-level $\text{recall}@k$. Semantic judgment is evaluated as three-class classification over *Equivalent*, *Partial*, and *None*, using accuracy and macro-F1. We additionally report whether an annotated target was selected among positive items. Final coverage classification is evaluated using accuracy, macro-F1, and per-state precision, recall, and F1 over the five coverage states. Semantic and coverage metrics are computed per run and averaged over the three runs of each model.

Full retrieval results. Table 1(a) reports the complete retrieval curves. At $k = 20$, every positive query is recovered on both routes; at $k = 15$, two Paper→Lean queries are still missed. FinEqs saturates trivially, since its whole Lean inventory has 11 declarations.

Semantic judgment diagnostics. Across all paper items, Paper→Lean judgment reaches $91.8 \pm 1.0\%$ accuracy and $84.1 \pm 1.5\%$ macro-F1 with Gemini 3.8 Flash and $92.5 \pm 0.6\%$ and $80.7 \pm 2.0\%$ with GPT-6 Sol. Paper→Blueprint reaches 93.9% accuracy and 84.7% macro-F1 with Gemini and 89.8% and 77.7% with GPT-6 Sol, with the same accuracy in every run. Among positive items, the models select an annotated target in 30–31/35 (Gemini) and 31–32/35 (GPT-6 Sol) Paper→Lean cases, and in 35–36/37 (Gemini) and 35/37 in every run (GPT-6 Sol) Paper→Blueprint cases. On the isolated Paper→Blueprint route, the *Partial* relation has 55.6% (Gemini) and 38.5% (GPT-6 Sol) precision and 71.4% recall for both.

Stability across runs. The same final state is assigned in all three runs to 96/98 items with Gemini and 97/98 with GPT-6 Sol. The unstable items are APAP’s Corollary 9 (*Partial* once, *Missing* twice) and Ringel’s Lemma 15 (*Partial* twice, *Formalized* once) under Gemini, and Ringel’s Theorem 5 (*Formalized* twice, *Partial* once) under GPT-6 Sol. Seven items are misclassified in every run of each model, so a majority vote over runs does not improve accuracy (90/98 and 91/98).

Five-state breakdown. Table 2 reports per-state performance for the final coverage classification, averaged over the three runs of each model. Gemini 3.8 Flash errs in both directions, while GPT-6 Sol recovers every *Partial* item and never reports a more complete state than annotated, at the cost of *Formalized* and *Statement only* recall.

Table 2: Per-state performance (%) of the final coverage classification, mean over three runs of each model.

State	<i>N</i>	Gemini 3.8 Flash			GPT-6 Sol		
		Precision	Recall	F1	Precision	Recall	F1
Formalized	7	82.1	85.7	83.8	100.0	52.4	68.5
Statement only	5	50.0	60.0	54.5	100.0	20.0	33.3
Partial	23	85.7	78.3	81.8	75.8	100.0	86.3
Blueprint only	13	100.0	100.0	100.0	100.0	100.0	100.0
Missing	50	98.7	100.0	99.3	100.0	100.0	100.0
Macro	–	83.3	84.8	83.9	95.2	74.5	77.6

Confusion matrix. Table 3 reports the full five-state confusion matrix, summed over the three runs of each model (294 predictions). Under Gemini, 4 predictions report a *Partial* item as *Formalized* (Ringel’s Theorem 10 in every run and Lemma 15 once) and 9 as *Statement only*; the other errors report faithful items as *Partial* or, twice, APAP’s Corollary 9 as *Missing*. Under GPT-6 Sol, all 22 errors report a *Formalized* or *Statement only* item as *Partial*. Coarse final coverage detection (any coverage versus *Missing*) is correct for 292 of 294 Gemini predictions and all 294 GPT-6 Sol predictions.

Table 3: Confusion matrix for final five-state coverage classification, summed over three runs of each model. Rows are ground-truth states and columns are predicted states.

Gold \ Pred.	Gemini 3.8 Flash					GPT-6 Sol				
	Form.	Stmt.	Partial	BP only	Missing	Form.	Stmt.	Partial	BP only	Missing
Formalized	18	0	3	0	0	11	0	10	0	0
Statement only	0	9	6	0	0	0	3	12	0	0
Partial	4	9	54	0	2	0	0	69	0	0
Blueprint only	0	0	0	39	0	0	0	0	39	0
Missing	0	0	0	0	150	0	0	0	0	150

F Agent-Ready Task Export

Every non-*Formalized* paper item can be exported as a structured task for a downstream formalization agent. The task type and completion criterion are derived deterministically from the current coverage state, as summarized in Table 4.

Table 4: Agent actions and completion criteria derived from coverage states.

Coverage state	Suggested action	Completion criterion
Missing	formalize_missing_result	Validated correspondence is established
Blueprint only	implement_planned_result	Validated local Lean match exists
Statement only	complete_proof	Equivalent Lean match with complete proof
Partial	repair_semantic_alignment	Local Lean match becomes Equivalent

Each exported task contains the paper item and source location, current coverage state, the retained Lean and Blueprint candidates with their semantic judgments, proof-audit information, the reason the item remains uncovered, and its state-specific completion criterion.

Task schema. Each task is serialized as the following JSON object.

```

Agent-ready task (JSON export)
{
  "task_id": "...",
  "paper_item_id": "...",
  "coverage_state": "missing | blueprint_only | statement_only | partial",

```

```

"suggested_action": "formalize_missing_result | implement_planned_result | complete_proof |
repair_semantic_alignment",

"paper_item": {
  "display_name": "...",
  "statement_latex": "...",
  "source_file": "...",
  "source_range": "..."
},

"lean_candidates": [
  {
    "declaration": "...",
    "relation": "equivalent | partial | none",
    "rationale": "...",
    "proof_status": "complete | incomplete | unknown",
    "source_file": "...",
    "source_range": "..."
  }
],

"blueprint_candidates": [
  {
    "blueprint_id": "...",
    "relation": "equivalent | partial | none",
    "rationale": "..."
  }
],

"reason_uncovered": "...",
"completion_criterion": "..."
}

```

Re-auditing tasks. After a new repository snapshot is processed, *LeanTrail* recomputes semantic alignment and proof status against the same paper item and derives its coverage state again. A task is resolved when its completion criterion is satisfied; otherwise, its state and evidence are updated. For example, a *Statement only* task becomes resolved when its equivalent Lean match no longer depends transitively on `sorryAx`, whereas unrelated additions to the repository leave the task open.