From Lean Proofs to Public Claims

Release checks and trust boundaries in one formal mathematics repository

Will Cook

July 2026

Abstract

Lean is a proof assistant, software for writing and checking formal proofs. Its proof-checking core is called the kernel. Lean verifies that a proof establishes the formal statement written in the source. It does not verify that a repository's public front page (the README) or a paper describes that statement faithfully. That leaves a decision Lean cannot make: whether public wording has the formal theorem's meaning, assumptions, and limits. In the repository studied here, a maintainer-reviewed record pairs each selected public claim with named Lean theorems, the finite cases they cover, and the stronger conclusion that remains open. Before release, Lean rechecks the proofs; a second program tests whether the recorded names, wording, and limits still agree after edits. That program can check only relationships a person selected and reviewed.

The worked example concerns a finite result around Erdős Problem 249. Lean checks a finite calculation, called a certificate, at each of 28 named parameter values ending at $t = 64$. The open statement requires certificates beyond every fixed cutoff, and the development proves that this unbounded supply exists exactly when the totient constant in Problem 249 is irrational, that is, not a ratio of integers. Claiming that these certificates continue beyond every cutoff would announce a solution while changing no formal proof. An earlier evaluation applied ten false edits. One changed the finite result into this stronger open claim and passed the release checker because the finite/open distinction was not yet recorded. After the claim record and checker were repaired to include that distinction, the checker rejected a test copy with the false wording. This covers one known failure; it gives no detection rate or general measure of reliability. The repository also studies Erdős Problem 257. Both mathematical problems remain open.

1 The publication gap

The repository studied here is a public development in the Lean 4 proof assistant [1] around two unsolved problems in number theory, Erdős Problems 249 and 257. Both problems remain open. The project proves intermediate results, exact reformulations, and finite certificates around them; it does not claim a solution to either problem.

One theorem illustrates the problem this paper is about. Lean has checked a finite certificate at each of 28 listed scales, the largest labelled $t = 64$. Here a scale is a listed value of t. A certificate is a finite check proving that a particular value is not an integer, although rationality would require it to be one. The corresponding open requirement asks for certificates beyond every fixed cutoff. Whatever the largest checked scale is, a

cutoff lies beyond it, and the finite list says nothing there. The finite list therefore does not settle the open problem.

A later README edit can nevertheless overstate the result. By an equivalence proved in the development (Section 3), certificates beyond every fixed cutoff are not merely better evidence. Their existence is equivalent to answering Problem 249 affirmatively: the totient constant is irrational. Three objects recur throughout this paper: the *finite theorem*, certificates at the 28 listed scales; the *open requirement*, certificates beyond every cutoff; and the *equivalence* between the open requirement and the irrationality. Lean has checked the first and the third; no Lean theorem carries the first to the second, and proving that implication would settle the problem. An edit that presents the finite cases as completing the open requirement therefore asserts, in English, exactly the bridge the development does not contain; in substance it announces a solution to an open problem. Every Lean proof remains valid under that edit, because the README is not part of any proof. No program in the repository decides whether a line of English has the same meaning as a formal statement. Lean acceptance therefore does not verify the public description.

The design has five parts: Lean source, the maintainer-reviewed claim record, authored public documents, generated indexes and summaries, and the release program and continuous-integration workflow (CI) that runs both checks on each proposed change. The repository keeps a *maintainer-reviewed claim record* in `docs/claims.json`. For each selected result the record states the public wording, its status, the named Lean theorems or definitions that support it (Lean calls such named items *declarations*), the bounded domain it covers, and the stronger conclusion the record marks as open. A release checker, `scripts/check_release.py`, then verifies that the recorded relationships still hold across the Lean source, the record itself, the authored public pages, and the generated files. The record covers only *registered claims*, meaning claims entered into it, not every line of prose in the repository; software cannot preserve a relationship it was never pointed at, and Section 5 shows this limit operating in practice. The shortest accurate summary of the division of labour is:

> Lean checks the formal proofs. A maintainer reviews what those proofs mean and how they may be described. The release machinery checks that the recorded relationships remain intact before a release.

Readers focused on the release design may skip Sections 3.1–3.2; Section 3.3 resumes with the reviewed record. Section 2 shows the workflow; Sections 4–6 give its guarantees, observed failure, and limits.

2 The release workflow

Read the upper half of Figure 1 from left to right: Lean source, claim record, and public documents, with generated views built from the record. The dashed arrows show human review. The lower half is automated: each job runs on its own fresh copy. The decision is: do both jobs pass?

Someone must compare a formal theorem with its public wording and judge whether the wording is faithful within stated limits. Once that judgement is recorded,

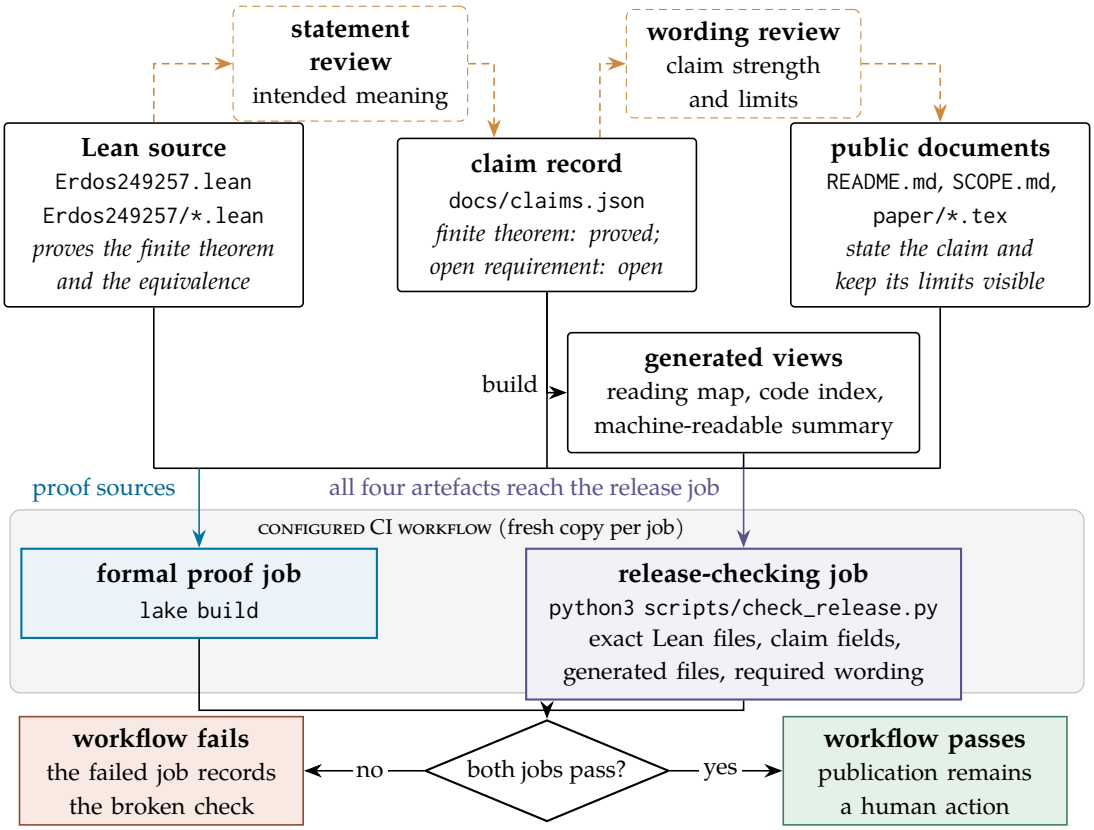

Figure 1: The configured checks for one saved change. Dashed elements are human: one maintainer performed both review steps here. Solid elements are files, programs, and automated checks. Lean checks the formal proofs. Human review compares their intended meaning with the recorded public wording and limits. The release job checks that recorded names, wording, Lean files, and generated views still agree. Continuous integration reruns the Lean and release jobs; neither interprets unrestricted prose. In the worked example of Section 3, the release job protects the recorded distinction between the finite theorem and the open requirement. Blue marks the Lean job, amber review, and violet the release job.

a program can test whether later changes satisfy the recorded rules about names, files, and wording. It does not decide whether the original judgement was correct.

The workflow does not technically force a second independent mathematician. A single maintainer can edit a Lean statement, its claim record, and the public wording together, and every check can pass while the shared interpretation is wrong; Section 6 applies this limit to the worked claim. The checks detect departures from reviewed decisions. They do not replace review or add authority of their own.

The two jobs answer different questions. `lake build` asks whether Lean accepts the imported formal statements and proofs. It knows nothing about the README, and it would still accept a change whose README falsely states that the open requirement has been met, because no public document is among its inputs. The second job, `python3 scripts/check_release.py`, asks whether the recorded relationships

among proofs, record, public pages, and generated files still hold; it checks that the Lean source matches the recorded version but does not run Lean. The workflow in `.github/workflows/lean.yml` runs them as two separate jobs whenever a saved change is sent to GitHub, either directly or for review.

Each job runs on a new GitHub-hosted virtual machine (a *runner*) and downloads the selected repository version (a *revision*) [2]. A checker error stops the program; the workflow treats that exit as a failing job. Repository settings decide whether that blocks release. Failure can stop the configured release process; a passing release job proves no mathematics.

The generated views select and reorganise recorded information (a reading map, a code index, a machine-readable summary) so that a reader or a program can enter the repository without opening every file. They create no new mathematics. Each view records the program that rebuilds it from source, and stale output is regenerated rather than edited by hand. The complete file map, ownership table, and maintenance commands live in `ARCHITECTURE.md`; this paper gives only what this argument needs.

3 One claim from Lean theorem to public page

We now follow one result from Lean source to public page. We begin with its public meaning, not its internal name:

> Lean has checked a finite certificate at each of 28 listed scales through $t = 64$. The registered open requirement asks for certificates beyond every fixed cutoff. The finite list does not settle it.

3.1 What the certificates are for

Erdős Problem 249 asks whether the totient constant

$$S = \sum_{n \geq 1} \frac{\varphi(n)}{2^n}$$

is *irrational* [3], meaning that it is not a/b for any integers a and b with $b \neq 0$. Here φ is Euler's totient function: $\varphi(n)$ counts the integers from 1 to n sharing no factor greater than 1 with n. The following exact equivalence explains why the certificates matter. Multiplying S by 2^N makes its first N terms integral; write R_N for the remaining rescaled tail. Thus R_N differs from $2^N S$ by an integer. Put $D_{N,h} = R_{N+h} - R_N$. It differs from $2^N(2^h - 1)S$ by an integer. For $h > 0$, the factor $2^N(2^h - 1)$ is a nonzero integer, so an integral $D_{N,h}$ would force S to be rational. Conversely, every rational number has an eventually repeating binary expansion, the base-2 analogue of a repeating decimal. Thus some $h > 0$ makes $D_{N,h}$ integral for every sufficiently large N. Lean checks the exact equivalence:

$$S \text{ is irrational} \quad \Longleftrightarrow \quad D_{N,h} \notin \mathbb{Z} \text{ for every } h > 0 \text{ and every } N.$$

For fixed h and N, Lean proves $D_{N,h} \notin \mathbb{Z}$ exactly when a certificate exists at a finite depth L. Its integer calculation approximates $2^L D_{N,h}$ with error at most $r = N + h + L + 2$. The

certificate checks that the residue modulo 2^L lies strictly between r and $2^L - r$, outside both bands compatible with an integral value. The omitted infinite tail is thus accounted for by an explicit bound. The claim record calls this a *certified kill*. We use the shorter word *certificate*: it proves only that this particular tail difference is not an integer.

The scales of the finite theorem compress this two-parameter requirement to one. The diagonal $H_t = \mathrm{lcm}(1, \dots, t)$ is the least common multiple of $1, \dots, t$: the smallest positive integer divisible by every integer in that list. It is therefore divisible by every candidate period up to t. If S were rational, its binary digits would have a period h_0 after a position N_0. For $t \geq \max(h_0, N_0)$, one has $h_0 \mid H_t$ and $H_t \geq N_0$, so rationality would make D_{H_t,H_t} integral; a diagonal certificate rules this out. The development proves the reduction exact: certificates along the diagonal beyond every fixed cutoff are equivalent to the full two-parameter requirement, and therefore, through the chain above, to the irrationality itself. Certificates on a finite list, however long, are not.

Writing $\mathrm{Cert}(t)$ for "a certificate exists at scale t", the two statements then read:

$$\begin{aligned} \text{checked: } & \mathrm{Cert}(t) \text{ for every } t \text{ in a fixed list of 28 scales;} \\ \text{open: } & \text{for every cutoff } T,\ \mathrm{Cert}(t) \text{ for some } t > T. \end{aligned}$$

The boundary between them is a matter of logical form, not of scale. A list of 28 scales, of 280, or of 28,000 stands in the same relation to the open requirement: each is a fixed finite list with a cutoff beyond it. No finite list, however long or expensive to verify, implies the second statement without a further theorem that produces new witnesses. Thus $t = 64$ is the endpoint of this verified computation, not a frontier that Lean has proved will continue. This is the boundary an edit can silently delete.

3.2 The formal evidence

The Lean source file (or *module*) `Erdos249257/DiagonalPincerCertificatesT64.lean` lists the 28 scales through $t = 64$: 1 and the prime powers at which H_t changes. Intermediate scales add no new case. Imported modules prove each listed case. The theorem `certifiedKill_diagonal_all_imported_through_t64` combines them: for every listed t, it gives a finite checking depth d_t and proves a certificate whose period and position are both H_t. Lean evaluates the remaining finite arithmetic at depth d_t. Lean's *kernel* is the small trusted program that checks every proof Lean accepts. The certificate proofs use `decide`: it evaluates the relevant finite yes-or-no proposition and, when the answer is yes, supplies a proof for the kernel to check [4]. The equivalence is also a proved Lean theorem, not an assumption of this paper. The module `Erdos249257/LcmConeFlatness.lean` proves the chain of Section 3.1: the tail-difference and pointwise certificate equivalences, followed by the diagonal characterisation `irrational_totient_series_iff_lcm_diagonal_certificate_supply`. The root `Erdos249257.lean` imports both modules, so `lake build` checks them.

3.3 The reviewed record

The quoted record entry contains four names specific to this repository. A *certified kill* is the certificate defined in Section 3.1. The entry abbreviates least common multiple as "lcm", so its *lcm-diagonal scales* are the values H_t. The *small periods* are the periods 1

through 8, each handled by one explicit certificate. A *certificate shard* is the same calculation at a fixed period, position, and depth, isolated in another Lean file and checked by `decide`.

Table 1 shows the substance of the corresponding entry in `docs/claims.json`, ordered as a reader meets it: what is claimed, where it stops, what remains open, then the supporting machinery. The entry also records an exact source line for each declaration; those coordinates are omitted from the table.

Field	Content
Public statement	"Kernel-checked certified kills for small periods and at 28 explicit lcm-diagonal scales through $t = 64$, plus finite certificate shards, each discharged by `decide`."
Bounded domain	The listed small periods, the 28 scales above, and the fixed period, position, and depth of each separate certificate.
Remaining open	A link to the registered open requirement *unbounded certificate supply*: produce a certified witness beyond every fixed cutoff.
Status	*verified finite instance*: Lean checked the stated finite inputs.
Supporting declarations	Five named Lean theorems with recorded modules and source lines, ending in `certifiedKill_diagonal_all_imported_through_t64`.
Claim identifier	`certified_kill_instances`

Table 1: The reviewed claim entry for the worked example, exact source coordinates omitted. Together, the positive statement, bounded domain, and remaining-open link distinguish the finite result from the open problem. In the failure of Section 5, overstatement came from deleting this boundary, not from inventing a theorem.

The table records the claim rather than explaining it. The remaining-open field points to the open requirement of Section 3.1, while the status *verified finite instance* classifies the finite theorem. Under the equivalence the remaining-open field is part of the claim: without it, the entry no longer separates the finite theorem from an answer to Problem 249. The record gives explanatory prose, the release checker, and generated views one stable reference. Prose must still bridge the gap between an exact claim and a reader's understanding.

The fields differ in kind, and the difference matters for what "reviewed" means. The public statement, bounded domain, remaining-open link, and status are fields about meaning. They record a human judgement, and only a person can meaningfully change them. The record fixes the allowed status terms, and the checker rejects any other status. The claim identifier lets software find the claim; it appears last, not in place of the wording, and the per-declaration source lines are location data that a program may refresh when code moves; updating them does not repeat the judgement about meaning. No field can by itself establish that the English is faithful to the Lean: the maintainer judged that the public statement above describes the five declarations correctly, and the record stores the outcome of that judgement, not a justification of it. Review and authorisation are distinct: review asks whether the wording is faithful; authorisation decides which

wording the project will publish. One maintainer performed both here. The review itself is concrete: read the final theorem, check that the wording claims the listed scales and no more, confirm that the open requirement stays named as open. It is also an event, not a permanent property: the review covered particular declarations and wording. A later change to either reopens it; later checks can only preserve what was recorded. That decision must survive ordinary rewrites months later.

3.4 What the checker verifies here

For this claim, `scripts/check_release.py` verifies a finite, listed set of relationships. Each declaration name must appear in the Lean source within three lines of its recorded coordinate. This checks that the name remains near its recorded position, not that the declarations entail the public wording. The checked Lean source itself must match the saved Git revision named in the claim record. The checker compares the top-level Lean file and the entire library directory byte for byte with that recorded version. It rejects any difference or extra Lean file absent from that version, so the verified names cannot refer to an older source than the one being shipped. The required limitation wording must remain present on the registered public pages; for this claim that wording is the visible trace of the boundary of Section 3.1, the clause keeping the open requirement open. The generated views must equal a fresh rebuild from their sources. Presence, identity, and proximity are all the checker can test. Lean checks the theorem; the checker checks the recorded description around it; a person remains responsible for the claim that the two say the same thing.

4 What the checks establish

The word "verification" is easy to overread here. Four separate questions are involved. Did Lean accept the formal statements and proofs? Did a maintainer record a judgement that selected public wording describes them within stated limits? Does the release checker find the recorded relationships intact at this revision? Did continuous integration run the configured jobs on that revision and report success? Each question needs different evidence. A yes to one does not settle the others.

When both jobs pass, Lean has accepted the proofs and the release checker has found every recorded relationship intact. This does not repeat the maintainer's review, establish that the review was correct, or show that every consequential claim was registered. Table 2 states what each layer can and cannot establish. Keeping the layers separate prevents the single word "verified" from making their combined guarantee sound stronger than it is.

Instantiated on the worked example, the development contains the finite theorem and the equivalence, but no theorem deriving the open requirement from the finite list. A maintainer recorded the judgement that the public wording asserts the finite theorem, confines it to the listed scales, and keeps the open requirement named as open. The release checker finds the recorded traces of that judgement intact: declaration names near their coordinates, agreement between the Lean files and saved revision, the limitation clause on the public pages, and freshly rebuilt views. Continuous integration records that each job ran on its own fresh copy of the revision and passed.

Layer	What it can establish	What it cannot establish
Lean build and kernel	The written formal statements are proved from their imported definitions and assumptions, using the recorded Lean version and dependencies.	That a formal statement is the one the author intended, or that any English description of it is faithful.
Maintainer review	A recorded judgement that a named formal statement has the intended meaning and selected public wording describes it within stated limits.	That Lean accepts the proofs; that every important claim was registered; or that the review was independent or correct.
Release checker	That recorded names, source lines, fields, links, required wording, Lean files, and generated files satisfy the declared rules; that prohibited proof shortcuts are absent; and that deliberately wrong examples still fail.	What unregistered prose means; whether the record is complete; or whether the human judgements it preserves are correct.
Continuous integration	That each job ran on its own fresh copy of the uploaded revision and exited successfully.	Independent mathematical approval; anything beyond what the two jobs themselves establish.

Table 2: What each layer establishes and leaves open.

The limits follow the same lines. Lean cannot notice English that quietly implies the open requirement met. The recorded review does not show that every page repeating the claim was found. The checker cannot reject a paraphrase it was never given. Passing both jobs cannot make a mistaken reading of the equivalence correct.

One part of the release checker scans the repository's Lean source itself. Among other patterns, it rejects proof placeholders (`sorry` and `admit`), project-defined `axiom` declarations, and native evaluation in each form covered by the pinned Lean version: `native_decide`, `decide +native`, or `decide` with its native configuration enabled. An axiom assumes a statement without a proof. The ordinary `decide` used for the certificates reduces a finite yes-or-no computation to a proof term that the kernel checks. Native evaluation instead compiles and runs that computation, then adds a dedicated axiom recording the result. The kernel can track the axiom but cannot replay the compiled computation [4, 5]. Such a theorem therefore also trusts the compiler and the code it ran. The scan covers project source only; imported library axioms remain part of the separate trust in the pinned dependencies. The checker also compares all public Lean files byte for byte against the saved formal source revision and rejects extra Lean files. It requires generated views to match a fresh rebuild, verifies the source and PDF file hashes of the shipped papers, and runs a set of deliberately wrong examples that must keep failing. For each covered example, this detects if a later checker edit stops rejecting that known fault. It does not show that every rule has such an example or that an unseen

fault will be rejected. None of these comparisons determines meaning. A file hash is a short fingerprint of a file's contents. Different hashes prove files differ; matching hashes strongly suggest identity. No hash says which file expresses the right mathematics.

When a mathematical file changes, local review begins with `lake build`. A person then asks whether any statement, assumption, or intended meaning has changed and updates the record and public pages when it has, following `docs/methodology.json`. The named builders regenerate the generated views, and `python3 scripts/check_release.py` compares the resulting files. Uploading any revision runs both jobs on a fresh copy. During local review, a prose-only edit need not repeat the proof build because no Lean file changed, but the release checker and human review still matter. If the words make a stronger or different mathematical claim, the record and the review must change with them. The escaped edit of Section 5 changed the claim without either change. The repository guide carries the full command list.

5 A boundary the checklist missed

The evidence comes from three different times. The historical exercise, the present post-repair test, and the later executable reconstruction answer different questions and must not be merged. Appendix A identifies their files and commands.

Historical exercise. The structured report in `docs/publication_evidence.json` identifies a saved Git revision. It says that ten deliberate false edits were applied to a separate copy one at a time, restoring it between runs. The release checker ran after each edit; the Lean proof job did not. The edits covered seven kinds of recorded relationship: allowed proof commands, record structure, source coordinates, required paper links, boundary wording, generated-file freshness, and size budgets. Examples included upgrading a conditional status to "proved here", deleting an open-boundary clause, moving a recorded source line, introducing `native_decide` into a proof, and changing a generated count. The original run logs were not retained, so the file is a report of the runs rather than their original output.

According to that record, nine of the ten edits were rejected. One escaped: the README clause saying that the finite cases *do not supply* the open requirement was changed to say that they *complete* it. Under the equivalence of Section 3.1, the altered page claimed in effect that Problem 249 was settled. It supplied in prose exactly the missing bridge from the finite theorem to the open requirement. The Lean proofs were untouched, and the release checker passed because this clause was absent from its checklist. The escape therefore shows that a passing release checker does not certify all public prose. It is not a statistical result and gives no detection rate.

Repair and present witness. The repair recorded the clause as an *anchor*, meaning wording that must remain on the public page. It also derived an opposing copy in which that wording is weakened or removed. The current README passes and the false copy fails. The repository calls this failing copy a *boundary witness*. The test shows that the repaired checker distinguishes the approved wording from one known false neighbour; it measures no broader effectiveness. The other nine edits were not rerun against the extended checklist, so there is no post-repair aggregate result.

Later reconstruction. The reconstruction file in Appendix A specifies ten edits that can be run against the saved evaluation version. Three scripted edits preserve their exact original targets; the other seven use fixed replacement targets because the original targets were not registered in the retained record. This reconstruction is a new experiment, not the missing runs. It is evidence about its documented replacement edits, not about the original outputs or timings.

The evidence marks a coverage boundary, not a reliability score. The edits were authored by the checker's author, so the human review gate of Figure 1 was deliberately omitted and the exercise tests the checklist alone. Five of the seven relationship kinds were probed by one edit. The exercise covers one repository, one maintainer, and one working style, with no comparison against ordinary continuous integration or careful manual review. Reader error and transfer to other repositories were not measured. The nine reported rejections rest on the historical report alone.
The retained evidence supports one design lesson:

> The program can preserve a relationship only after a person has identified and recorded that relationship.

Requiring a clause to be present does not detect a contradictory stronger claim elsewhere on the page. Coverage grows only when a person notices a relationship, records it, and supplies an example that must fail. Coverage can also shrink: deleting a registered relationship leaves no rule requiring it, so every remaining check passes. Retiring a commitment therefore deserves the same review as creating one.

The shape of this limit is familiar from Section 3.1. The recorded checklist stands to unrestricted prose roughly as the finite certificate list stands to the open requirement: every registered relationship is real and checked, and no finite list of relationships exhausts an open-ended domain. The comparison is structural, not exact: prose has no equivalence theorem and no diagonal compresses it. It shows why the checker's silence outside its records is a continuing limit, not a defect one repair can remove.

6 Scope, reuse, and limits

The failures the design addresses. The checks address accidental disagreement after a sound review. A declaration may move while its recorded line number stays fixed, and a generated view may be stale or hand-edited. A rewrite may remove a required limitation, a status may be upgraded, or the shipped Lean files may cease to be the reviewed ones. A check may also decay until it can no longer fail. In each case, one recorded item disagrees with the others, which a mechanical comparison can detect.

The failures it does not address. Three failure modes remain outside the design. *Unregistered wording:* a paraphrase can strengthen a claim without touching a registered anchor, and a contradiction, misleading emphasis, or new public document can escape for the same reason. Requiring one clause to be present does not require the page to agree with it. *Coordinated but wrong change:* a maintainer can alter source, record, and prose together, so every comparison agrees. If the certificate definition were weakened while the record and prose were updated to match, all checks could pass although the

public reading of the equivalence was wrong. *Mistaken review:* if the original judgement was wrong, the machinery preserves the mistake. It checks persistence, not the quality of the judgement.

Limits of the design and this implementation. Some limits follow from the design. Claims remain in unrestricted prose beside an external record, so human interpretation cannot be eliminated. Only listed relationships are checked. The repository calls this set its *assurance perimeter*: recorded commitments inside it trigger automatic checks; outside it, the checker is silent. Choosing what to record remains a human judgement. This repository gives priority to headline results and to wording whose accidental strengthening would change the project's public status, as the deleted boundary clause did under the equivalence. If the same claim appears on several pages, the checker sees only the appearances named in the record. Other limits are implementation choices: one maintainer performs both review and authorisation; the checker accepts a theorem name within three lines of its recorded location; anchors require exact wording; the exercise used one edit per relationship type; and the original logs were not retained.

Reuse. Another formal project could reuse the pattern without copying any file format. It applies when bounded formal evidence sits beside a stronger public target. Examples are finite cases beside a universal statement; a conditional theorem beside its unresolved hypothesis; one direction of an implication beside a claimed equivalence; or a result under named assumptions beside an unconditional headline. What transfers is the reviewed boundary between what is established and the adjacent stronger statement that is not. The minimum obligations are concrete. Each public claim needs its own record entry, named formal evidence, and the exact source version containing it. Its scope, assumptions, and the stronger conclusions it does not establish must be explicit. Human review is distinct from automated checking. Each generated public view needs a named builder. Proof checking and publication checking must be separate and able to fail. Every important recorded relationship needs an example that deliberately violates it, and the record must present its coverage as selective rather than complete.

Stronger variants impose more structure. A project could generate public wording from the record instead of anchoring authored prose; extract formal statements and assumptions instead of recording source lines; require an independent reviewer for designated claims; constrain critical claims to a controlled vocabulary; or use tools that interpret prose to propose discrepancies for human review. This repository instead keeps unrestricted prose and an external record.

What is not established. The architecture does not establish a solution to either Erdős problem; that a formal statement is the statement the author intended; that software understood any unregistered prose; that the record is complete; that one maintainer's review is independent or adequate; or that the design transfers to other projects with its behaviour intact. A large number of passing comparisons measure none of those things. What a passing workflow establishes is narrower: the configured jobs ran on the named revision, Lean accepted the formal proofs, and the release checker found every recorded relationship intact. This paper is an architecture note with one bounded case study, not an empirical evaluation of the design. A credible evaluation would need several stages,

each with its own limit. Records naming the reviewer, revision, and time could establish who checked which revision, when, and how often, but not effectiveness. Deliberately wrong edits by people other than the checker's author could measure the response to unseen faults, but not behaviour in ordinary use. Only after those stages would comparison with ordinary review on the same changes become meaningful. None of those stages exists yet, and this paper claims none of them.

7 Related systems

A *proof blueprint* pairs an informal outline with Lean declarations and records which results depend on which earlier results. `leanblueprint` stores that outline in TEX and checks that every named declaration exists [6]. LeanArchitect stores blueprint data in Lean, infers those dependencies, tracks formalisation progress, and exports synchronised TEX [7]. Both support a formalisation in progress. The claim record instead asks which public wording was reviewed after Lean accepted a proof, and which stronger conclusion remains open.

Lean Atlas and EconCSLib ask whether a formalisation expresses its intended source mathematics. Lean Atlas leaves semantic verification to people. Given chosen theorem statements, its Lean Compass selects the project declarations whose meaning can affect them, narrowing what a person must inspect; it assumes Lean's standard library and the mathematical library Mathlib are semantically correct rather than inspecting them [8]. EconCSLib uses a different division: an AI language model writes Lean; Lean checks the formal statements and proofs; a dashboard and separate language-model reviewers help people assess whether a source paper was translated correctly. Its paper reports that human review remained incomplete [9]. This paper addresses a later boundary: it starts from accepted Lean proofs and asks how subsequent public wording can remain within a reviewed interpretation of them.

Isabelle/DOF, a document system built on the Isabelle proof assistant, places formal and informal material in one checked document. Authors define an *ontology*: document classes with typed fields and rules. They label passages with those classes, and Isabelle's editor reports rule violations as they edit [10]. The bounded domain and open conclusion in Table 1 could be typed fields in such an ontology. This repository keeps prose unrestricted and checks a separate record; its checker cannot inspect prose that the record does not name.

CASCADE derives tests and a second implementation from the same documentation using language models. It reports a likely inconsistency only when a generated test fails on existing code but passes on code generated from that documentation. A person must confirm it [11]. Unlike this checker, it can inspect unrecorded documentation.

Mutation testing asks whether tests detect deliberately introduced faults [12, 13]. The deliberately wrong copies in Section 5 serve that narrower purpose: a known wrong input must continue to fail. The ten-edit exercise is a small, hand-authored mutation run. Since nine edits were not rerun after repair, it gives no post-repair detection rate.

Requirements traceability follows a requirement from its origin through development, deployment, use, and revision [14]. An assurance case sets out a system claim, the argument for it, and supporting evidence. The Goal Structuring Notation (GSN) is

a graphical notation for that argument structure [15]. The claim record resembles both, but it does not contain an argument that its Lean declarations justify the public wording. It records the approved wording, supporting declarations, scope, and limits; the justification remains a human judgement.

8 Conclusion

The mathematical boundary is that Lean has checked certificates at 28 listed scales through $t = 64$, while the registered open requirement asks for certificates beyond every fixed cutoff. The development proves that this unbounded supply is equivalent to answering Problem 249 affirmatively; it does not prove the supply. Both Erdős Problems 249 and 257 remain open.

The evidence boundary is different. The historical report says that one of ten deliberate edits escaped the then-current checker. That escape shows that a passing release checker does not certify all public prose; it gives no detection rate and no comparison with ordinary review. The present post-repair witness accepts the current README and rejects a test copy containing the false clause.

Whether these checks reduce inaccurate public claims at acceptable cost remains untested. Until the evaluation described in Section 6 exists, a passing workflow means only that the configured jobs ran on the named revision, Lean accepted the proofs, and the release checker found every recorded relationship intact.

A Reproducibility

The reviewed claim record, `docs/claims.json`, names the saved Git revision of the Lean source, which the release checker requires to match exactly. This paper omits the changing commit identifier so that the claim record is the only place that states it.

Use of AI assistance. Large-language-model assistants were used during development for proof drafting, refactoring, and prose editing. The author selected and reviewed the claims in this paper and remains responsible for them. AI output is not proof authority: Lean checks each formal proof against the fixed library version, while the author remains responsible for connecting those proofs to public wording.

The paper inventory, `docs/publication_contract.json`, records source and PDF cryptographic hashes and validation commands. The evidence file for Section 5, `docs/publication_evidence.json`, records the protocol, outcomes, and limitations. The reconstruction file, `experiments/publication_mutations.json`, specifies the ten edits. The script `scripts/run_publication_mutations.py` checks that each edit applies once (`--verify-operators`) and, by default, runs all ten in a copy of the saved evaluation version (`--all`). For an edit whose exact original target was not preserved, the file names a fixed replacement. The original raw outputs were not retained; the reconstruction does not claim to be them.

The papers build with Tectonic or standard LaTeX via `make -C paper`. The tracked root PDF is the shipped copy, and the inventory checks its file hash. These identities name artefacts; they do not interpret them.

References

[1] L. de Moura and S. Ullrich, *The Lean 4 Theorem Prover and Programming Language*, in *Automated Deduction—CADE 28*, Lecture Notes in Computer Science 12699, 2021, pp. 625–635, DOI.

[2] GitHub, *GitHub-hosted runners*, documentation, accessed 18 July 2026.

[3] P. Erdős and R. L. Graham, *Old and New Problems and Results in Combinatorial Number Theory*, Monographies de L'Enseignement Mathématique 28, L'Enseignement Mathématique, Université de Genève, 1980, p. 61, scan.

[4] Lean project, *The Lean Language Reference: Validating a Lean Proof*, documentation, accessed 18 July 2026.

[5] Lean project, *The Lean Language Reference: Axioms*, documentation, accessed 18 July 2026.

[6] P. Massot, *leanblueprint*, plasTeX plugin for Lean formalisation blueprints, 2020, software repository, accessed 18 July 2026.

[7] T. Zhu, P. Monticone, S. Welleck, and J. Avigad, *LeanArchitect: Automating Blueprint Generation for Humans and AI*, in *17th International Conference on Interactive Theorem Proving*, LIPIcs 382, 2026, pp. 25:1–25:16, DOI.

[8] B. Yanahama and A. Sannai, *Lean Atlas: An Integrated Proof Environment for Scalable Human–AI Collaborative Formalization*, 2026, arXiv:2604.16347.

[9] N. Garg, *EconCSLib: AI-Assisted Lean Formalization for Economics & Computation research*, 2026, arXiv:2606.13306.

[10] A. D. Brucker and B. Wolff, *Isabelle/DOF: Design and Implementation*, in *Software Engineering and Formal Methods*, Lecture Notes in Computer Science 11724, 2019, pp. 275–292, DOI.

[11] T. Kiecker, J. A. Sparka, M. Reuter, A. Ziegler, and L. Grunske, *CASCADE: Detecting Inconsistencies between Code and Documentation with Automatic Test Generation*, *Proceedings of the ACM on Software Engineering* 3 (FSE), Article FSE168, July 2026, pp. 3816–3838, DOI.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, *Hints on test data selection: Help for the practicing programmer*, IEEE Computer 11(4), 1978, pp. 34–41, DOI.

[13] Y. Jia and M. Harman, *An analysis and survey of the development of mutation testing*, IEEE Transactions on Software Engineering 37(5), 2011, pp. 649–678, DOI.

[14] O. C. Z. Gotel and A. C. W. Finkelstein, *An analysis of the requirements traceability problem*, in *Proc. First IEEE International Conference on Requirements Engineering*, 1994, pp. 94–101, DOI.

[15] Assurance Case Working Group, *Goal Structuring Notation Community Standard (Version 3)*, Safety-Critical Systems Club, 4 May 2021, standard.