WHAT DOES A PASSING CHECK PROTECT? CONTRACT-BASED AUDITING OF RESEARCH CLAIMS

Anonymous authors
Paper under double-blind review

ABSTRACT

A research workflow can pass its checks and still break the requirements behind a paper's claims. We introduce GateCheck, a contract-based framework for auditing this gap in research compendia, the repositories released with papers. It describes each check by what it protects, its reference, what it observes, when, and for which artifact. A witness that never reads the verdict labels each accepted fault, per artifact, as restored, escaped or unresolved. Five compendia by other authors give three findings. First, a pass and a protected claim are different things: the reference can move with the edit, the check may not observe the violation, or a build fixes the output but not the commit. Two checks that reject equally many faults left 2 versus 0 violations in the outputs. Second, complementary requirements need complementary checks. Re-deriving labels from their evidence and matching records against an approved list caught disjoint faults on one compendium; together they rejected every fault we made there and, given declared approvals, allowed every permitted change, but also an incomplete removal. Third, a correct diagnosis need not add protection. On a compendium with a different verifier, the contract reading predicted the native check's verdicts, but the reused repairs caught nothing it missed. Verification should be judged by the requirements and artifacts it protects, not by how many mutations it rejects.

1 INTRODUCTION

A research compendium is the repository released with a paper: its data, its code and the numbers it reports. Many compendia ship a check, such as a test suite or a command that rebuilds the paper's numbers. When the check passes, a reader tends to trust the claims. This paper asks one question: *when does a passing check show that a declared requirement holds in the artifact being released?*

Here is an example, on work by authors unrelated to us. CSV_Wrangling (van den Burg et al., 2019) is the compendium of a data-mining paper on detecting the dialect of a CSV file. We call it the CSV compendium. The command declared in its pinned CI configuration is `make output && git diff`. We changed one line of its analysis code, so that two dialects count as equal when their delimiters alone match. A computed accuracy behind the paper's abstract moves from 97% to 98%, and 118 of the 269 claim values the build writes change. The command rebuilds those numbers, prints the change, and exits 0, because `git diff` without `--exit-code` exits zero whether or not it prints a difference. The check passes, and the wrong number stays in the output (Fig. 1).

Whether a pass protects a claim depends on three things. *Where the reference lives*: a check that compares with expected values kept in the same commit passes when a result and its expected value are edited together. *What the check observes*: an unchanged accuracy can hide an evaluation that used the wrong rows. *Which artifact and stage it assesses*: a build can repair the output while the committed claim stays wrong. With the requirement, they form a check's *verification contract*.

We first define the contract and the outcomes of an audit (§2). Then we audit checks on three compendia by other authors, two of them in machine learning (§3–4), and test repairs built from the diagnoses on one of them and two more (§4.5). Our contributions are three.

1. **Verification contracts.** We describe a check by five parts: the requirement, the reference, the observation, the stage and the artifact. They separate the rule a check implements from the requirement it is meant to protect.

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

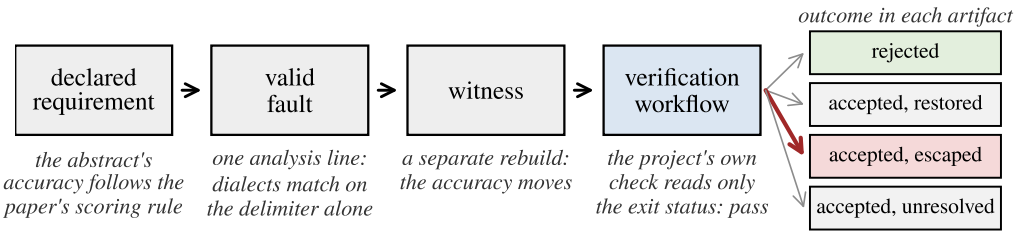

Figure 1: **One audit measurement**, on the CSV compendium's analysis defect. The witness sees the accuracy move from 97% to 98%; the project's own check reads only the exit status, so the wrong value stays in the generated output (heavy arrow).

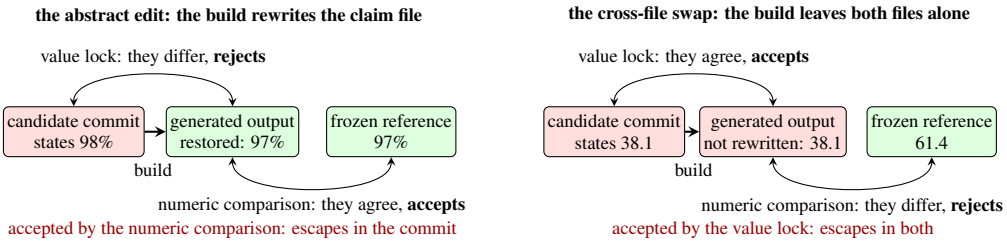

Figure 2: **Two checks, two contracts.** Two rows of Table 5. Red boxes hold wrong values, green boxes the declared ones. Each check rejects the fault the other accepts.

2. **Audit outcomes per artifact.** For each fault a workflow accepts, a witness that never reads the verdict tells whether the fault is restored, escaped or unresolved in each artifact. Checks with equal rejection counts can leave different violations in what is released.
3. **Complementary checks and the limits of repair.** The contract finds distinct gaps and checks that cover them together. It also shows when checks add nothing: any two of three output comparisons gave the same rejections, and reused repairs on another verifier caught nothing new.

Testing a project's own checks with mutations is not new (Shulepov, 2026); we ask what a pass shows about a declared requirement in a named artifact (§5).

2 Verification Contracts and Audit Outcomes

An example. The accuracy in the CSV compendium's abstract is computed from the committed detection records: a detected dialect counts as correct only when its delimiter, quote character and escape character all match the annotated one. So the *requirement* is: every stated accuracy equals the value this rule computes over the declared records. Two checks could guard it. A *value lock* runs the build and asks whether the 27 claim files still equal the candidate commit. A *numeric comparison* runs the build and fails if any of 285 registered numbers appears in its prose file fewer times than in the *frozen reference*, a copy made before the edit; which claim a number belongs to is not checked. Neither pass says that the requirement holds everywhere. Fig. 2 shows two faults they disagree on.

The verification contract. The *verification contract* of a check has five parts, $C = (r, b, o, t, p)$: the requirement r the claim needs, the reference b the check compares with, the observation o it reads, the stage t at which it reads, and the assessed artifact p its verdict speaks for. The artifact is the *candidate commit* (the edited revision as committed) or the *generated output* (the work tree the build leaves, or the terminal output if nothing is written). A pass says only that at stage t the observation o agrees with b. Table 1 gives each check's rule; injected faults test whether it is enough.

check	passes when	reference b	edit reaches b	observation o	stage t	artifact p
The CSV compendium						
the project's own check	the build exits with status zero	none	–	the build's exit status	after build	none
one-flag fix	no tracked file differs	candidate commit	yes	every tracked file	after build	candidate commit
value lock	no claim file differs	candidate commit	yes	27 claim files	after build	candidate commit
numeric comparison	no registered count drops	frozen reference	no	285 (file, number) pairs	after build	generated output
claim registry	no claim moves, drifts or goes missing, and no evidence file changes	frozen reference	no	the registered claims and the evidence files	before and after build	candidate commit, then generated output
The graph-network compendium (no check of its own)						
README comparison (reconstructed)	each printed accuracy equals the README's at the README's precision	README of the candidate commit	yes	3 printed accuracies	after the run	generated output
full recomputation	the accuracy recomputed over the declared test ids equals the printed one	the declared test ids	no	the model's outputs and the printout	after the run	generated output
loader-id check	the loader's test id list equals the declared ids, with counts	the declared test ids	no	the loader's test id list	after the run	generated output
The agent-failure compendium (its authors' check)						
the authors' check	its three CI steps exit zero	expected values in the candidate commit	yes	22 computed values	during the command	candidate commit
frozen report comparison	each value equals its frozen one	frozen reference	no	33 report values	after the command	generated output

Table 1: **The contract of each check, as implemented.** "Passes when" is the implemented rule, not the claim's requirement. "Edit reaches b": an edit to the candidate commit can change the reference. The one-flag fix is `git diff --exit-code` after the build. Codes: App. B.

Two trust settings. A check's reference can lie inside what an edit may change, or outside it (column "edit reaches b" of Table 1). In the *fixed-verifier* setting, an edit may change code, data, evidence or a stated result, but not the check or its expected values. In the *co-edit* setting, it may also change the check's comparison or its expected values. A reference frozen outside the candidate commit resists co-edits, but it must be rebuilt after every approved update.

Where requirements come from. Table 2 lists the 7 requirements behind the findings on the three compendia: 5 are inherited from the paper or its compendium's own computation, 1 is an evaluation convention we declared, and 1 is imposed by us. Column "kind" says what a violation changes: a stated result (changed quantity or classification), how a number was computed even if it is unchanged (evaluation procedure), or whether a restated number agrees with its source (reporting consistency). For the last two we do not claim that the conclusion is wrong.

One measurement. Each measurement links a requirement, a valid fault that violates it, a witness, the workflow, and each artifact's state afterwards (Fig. 1). **The project's own check** G_0 (the **native check**) is a documented verifier or the command its CI declares; if a project ships none, we build a **reconstructed** one and claim less from it. An **added check** K runs beside it. A **workflow** is G_0 together with a set $\mathcal{K}$ of added checks, declared for each arm of a study: on a candidate x, $G_{\mathcal{K}}(x) = G_0(x) \wedge \bigwedge_{K \in \mathcal{K}} K(x)$. A workflow accepts only if every check in it passes, and an error is never a pass. A **valid fault** changes at least one byte at the place its rule chose, and the file still parses. A **witness** re-runs the computation a fault touches and compares a declared quantity with a reference named before any run. It never reads the tested check's verdict. It says that the quantity moved, not which value is right.

Outcomes, and the artifact they concern. Take a fault that breaks its requirement: a witness saw the violation, or the edit itself is the violation. Assume the workflow accepts the same run without the edit (its *clean pair*). The workflow either rejects the fault or accepts it. An accepted fault is *restored* in an artifact when a witness shows that the requirement holds there again; rewritten bytes alone

requirement	source and justification	protected claim	permitted transformations	witness	kind
stated value (CSV)	inherited: the abstract reads these files	stated accuracies, e.g. 97%	reformat, move, rename, refactor	separate rebuild	changed quantity
scoring rule (CSV)	inherited: the compendium's scoring code	every computed accuracy	refactor	rebuild with the commit's code	evaluation procedure
population (graph)	convention: declared test ids, once each	test accuracies 81.0, 71.9, 78.9	any order (declared)	rows at the evaluation call	evaluation procedure
metric (graph)	inherited: the README's test accuracy	test accuracies 81.0, 71.9, 78.9	reformat the README	recompute from logits	changed quantity
label = rule(evidence) (agent)	inherited: the paper's decision rule	regime counts, e.g. 13/25	refactor; re-derived update (declared)	the rule in our code	changed classification
stated value (agent)	inherited; rates as numbers: ours	printed counts, e.g. 1 survivor	rename, refactor; 1.000 as 1.0	counting in our code	changed quantity
README consistency (all three)	imposed by us: no shipped check compares it	README numbers	reformat; round to its precision	README row vs reference	reporting consistency

Table 2: **Where each requirement comes from.** Convention: declared by us in a protocol, before the runs that test it. Permitted transformations leave the requirement intact.

are not enough, because a build could replace one wrong value with another. It *escapes* when the requirement is still violated, and it is *unresolved* when its final status cannot be established. A build can restore the generated output while the candidate commit still holds the edit, so every outcome names its artifact. For a fault i with requirement r_i and an artifact p, $\mathrm{EscapedViolation}_i(p) = \mathbf{1}\{\text{the workflow accepts } i\} \cdot \mathbf{1}\{r_i \text{ is violated in } p\}$.

Permitted maintenance. Rewriting 0.30 as 0.3 changes bytes but no claim. An edit is permitted maintenance when the requirements still hold; an approved update moves numbers on purpose and is judged against its new requirement. A check that fires on permitted maintenance raises a *nuisance alarm*: it enforces its own rule, but blocks an edit that the claims allow.

Using GateCheck. The researcher specifies the requirement, the assessed artifact, the trusted reference, the permitted changes, a witness and the candidate checks. GateCheck makes the edits, runs each workflow with clean pairs and assesses each artifact. It tells whether a violation escapes, whether an added check prevents it, whether a simpler check already does, and which permitted edits each check blocks. An audit may find that proposed repairs miss the remaining gap (§4.5).

3 Experimental Design

We study three compendia by other authors (§4.1–4.4), reread our injection records (§4.4), and test repairs on one of these and two more (§4.5).

Three compendia. On the CSV compendium, we test each added check at one pinned commit on the analysis defect, a witness-sensitivity control (every dialect equals every other), a crash control and four *claim drifts*, edits to a stated value itself (§4.3). The graph-network compendium is SGC (Wu et al., 2019), the code of an ICML paper on a simplified graph convolutional network. It ships no test, CI or committed output, so its check is reconstructed from the README table of three printed accuracies. The agent-failure compendium was chosen by a rule committed before the search; its CI runs the authors' claim verifier, audited as shipped. Its claim faults are split into the two trust settings by a rule reading only the edit sites (App. O). Every edit runs with a clean pair, and checks are compared only on the same work tree at the same stage.

Controls and protocols. A passing run is evidence only if the check can fail, so the studies include clean, witness-sensitivity and positive controls. Each controlled study followed a protocol committed before its first run, which fixed the edits, checks, scoring rule, predictions and the claim each outcome allows (App. E). Outcome labels, trust settings and requirement sources were added post hoc to stored records and change no count a protocol fixed (App. U). No result is an independent reproduction. The re-derivation check of §4.5 was validated against a separately written reading of the paper's decision rule and 32 hand-worked boundary cases; the check and that second reading share authors but not code (App. P).

trust setting	claim faults	the authors' check alone	+ value lock	+ frozen report comparison	+ numeric comparison
fixed verifier	5	3 rejected, **2 escaped**	n/a	3 rejected, **2 escaped**	4 rejected, **1 escaped**
co-edit	2	0 rejected, **2 escaped**	n/a	2 rejected, 0 escaped	0 rejected, **2 escaped**

Table 3: **Claim faults on the agent-failure compendium, by trust setting** (§2; each fault and the split rule: App. O). Escaped (bold): accepted, with the requirement violated in both artifacts. n/a: nothing to compare. All 16 clean runs pass.

What the studies can show. In all but the last controlled study we chose each fault after reading the code it targets, so each fault is a counterexample, not a sample. One counterexample refutes an implication such as "a matching accuracy shows that the declared procedure was followed"; how often checks fail would need a sample of compendia. The transfer study tests reuse within a shared verifier template. The last study drew its faults at random, within families we fixed. On its compendium, a witness that applies the paper's stated exclusion rule does not reproduce 48 of the 433 printed values. Post hoc, we traced all of them to one step of the released code that the paper's text does not state: it drops fits with a large standard error. We left those values out before the draw, so that study measures the remaining audited scope (App. T).

4 Findings

4.1 Where does the reference live?

A reference kept in the candidate commit can move with the edit it should catch. The agent-failure compendium is the code of a workshop paper on tool-use agent failures (Li, 2026a), by an author who is not one of ours. Its CI runs a claim verifier that counts committed trace labels, like nontrivial (failing from a later replayed prefix) or prefix-0 (failing from the start), and checks 22 values against expected values in its source. We also ran the value lock, the numeric comparison and a new frozen report comparison: the verifier's report against the frozen reference. The authors' command recounts committed results and re-runs no model.

On an independent ML compendium whose authors ship a claim verifier, that verifier rejected an analysis-code defect, an input-data edit and an edit to a committed result label. Each made a value the verifier computes differ from the paper: the survivor count (the paper prints 1), the tasks per difficulty tier (40, implied by 'ten tasks per family-tier cell'), and the natural-failure regime counts (13/25 and 7/25).

The reference moves with the edit. Table 3 splits the claim faults by trust setting. The verifier's reference lives in the candidate commit. It accepted two edits while the printed count disagreed with the paper: one weakened its comparison, the other changed a result label and its expected count together. A comparison against a frozen pristine reference rejected both. It is the verifier's own comparison with its reference frozen outside the candidate commit, so this outcome follows from where the reference lives; the run confirms the mechanism. In the fixed-verifier setting, the frozen report comparison lets through the same faults as the authors' check. We chose the seven faults after reading the verifier, one per layer of the claim path; which ones escape follows from that choice, not from a sample. Two of the four that the authors' check accepted edit the verifier or its expected values. Of the two fixed-verifier faults it accepted, one changes the evidence behind a label (§4.2); the other is a README edit. Neither the authors' check, the value lock nor the frozen report comparison reads the README's restated counts, so an edit there passed those three workflows. The numeric comparison, which counts numbers in prose files, rejected it. Because the command writes nothing, no accepted fault was restored. Each escaped violation was present in the candidate commit and in the generated output alike.

Two further limits of a reference. On this compendium the value lock we used on the CSV compendium had nothing to compare. This project's command writes no tracked file, so the lock compared the candidate commit with itself and accepted every edit. Its passes here say nothing

edit	requirements broken	README comparison	recomputation	loader-id check	combined check
drop one test node	population, metric	**accepts**	rejects	rejects	rejects
evaluate every test node twice	population	**accepts**	**accepts**	rejects	rejects
use validation rows at the evaluation call	population, metric, reported value	rejects	rejects	**accepts**	rejects
max values instead of argmax	metric, reported value	rejects	rejects	**accepts**	rejects
reformat the README number	none	accepts	accepts	accepts	accepts

Table 4: **Which check accepts which edit, on the graph-network compendium.** Each word holds on all 3 claims. Bold: accepts while a requirement is broken. Combined: recomputation and loader-id check together. Every check accepts all 3 clean runs. Not a ranking.

about any requirement. When we appended one byte to a committed result file, the lock exited 1. The reference also fixes the form of a value. The verifier compares repair rates as strings, as the authors' writer formats them. Rewriting 1.000 as 1.0 in the committed summary, a hand edit their pipeline would not make, failed it and blocked the edit in every workflow. Under our requirement, which reads rates as numbers, this is a nuisance alarm; under the authors' string declaration it is not. The frozen comparison accepted the edit. This is one ML compendium: a case, not a rate.

4.2 What does the check observe?

A matching aggregate does not show that the record-level steps behind it were right. On the graph-network compendium, the *population* requirement, our convention (Table 2), says that each declared test node is evaluated exactly once, in the rows passed to the evaluation function. The *metric* requirement says that each printed accuracy is the model's accuracy over the declared test set, and the *reported-value* requirement that the README value equals the printed value at the README's precision. An accepted violation escapes in the terminal output.

A matching accuracy does not show that each example was evaluated once. Table 4 is a controlled diagnostic on the same compendium, not a new repository: 5 edits we chose and 3 clean runs (App. N.5). Dropping one test node moves the printed accuracy so little that it still rounds to the README's number, so the README comparison accepts it and the recomputation rejects it. Evaluating every test node twice doubles both the correct predictions and the evaluated nodes, so every printed accuracy stays the same. The README comparison and the full recomputation both accept it, although it breaks the population requirement. A check that compares the loader's test id list with the declared ids, keeping counts, rejects it. It reads the loader's list, not the rows passed to the evaluation, so it accepts an edit made at the evaluation call: using validation rows there leaves the checked list unchanged. So neither a matching metric nor a correct list upstream shows that the evaluation followed the procedure we declared for it. The combined check rejects the four breaking edits. It reads no README, and rejects the validation-row edit only because the accuracy moves.

A matching count does not show that each label follows from its evidence. On the agent-failure compendium, each committed label must equal the paper's decision rule applied to its replay evidence (Table 2); our witness applies that rule. The authors' verifier has the same limit as the checks above. It accepted an edit to the committed replay evidence that, under the paper's own decision rule, moves one natural failure from nontrivial to prefix-0. The verifier counts the committed labels and never re-derives them from the evidence. The frozen report comparison accepted the edit too, because the reported counts did not change; the numeric comparison reads no record.

4.3 Which artifact and stage does a check assess?

A check speaks only for the artifact it reads, at the stage it reads it.

The analysis defect. Both added checks reject the defect of §1; the project's own check passes it after printing a 23-file diff. A fresh replay reproduces the accuracy and the changed values (App. X).

Equal rejection counts, different escapes. Table 5 adds the four claim drifts: a *table swap* of two values in a table the build rewrites, a *cross-file swap* of two values across two claim files it does

		value of the claim			workflow: the project's own check ...		
claim fault	requirement type	declared	committed	after build	alone	+ value lock	+ numeric comparison
analysis defect	stated value	97	97	98	escapes in output	rejects	rejects
table swap	stated value	93.75	94.92	93.75	escapes in commit	rejects	escapes in commit
cross-file swap	stated value	61.4	38.1	38.1	escapes in both	escapes in both	rejects
abstract edit	stated value	97	98	97	escapes in commit	rejects	escapes in commit
README edit	README consistency	11	12	12	escapes in both	escapes in both	rejects

Table 5: **Five claim faults on the CSV compendium, and where each accepted one stays wrong.** "Committed": in the candidate commit; "after build": in the generated output. "Escapes in" names the artifact in which an accepted fault violates the requirement. Every clean pair is accepted.

not rewrite, an *abstract edit* of the accuracy the abstract reads, and a *README edit* of an integer nothing rewrites. The project's own check rejects 0 of the 5 faults. The value lock and the numeric comparison each reject 3, but not the same ones (Fig. 2). The value lock accepts the cross-file swap, because the build leaves those files alone, and the README edit, because the README is not a claim file. The numeric comparison accepts the table swap and the abstract edit, because the build restored both values before it read the output; the commit still holds each edit.

Two contracts, two artifacts. If a project releases the commit without a build, the value lock lets 2 violations escape into it and the numeric comparison 2, and the faults differ. If it releases the generated output, the value lock lets 2 escape and the numeric comparison 0. So a rejection count does not say what a check protects. These 5 faults on one compendium do not rank the checks.

4.4 Coverage, diagnosis and maintenance cost

A richer representation adds diagnosis, not detection. The claim registry stores each claim with its type, location and value, and a fingerprint of the evidence files; we call this typed claim identity. It reads before and after the build. At the reading point the other arms use, typed claim identity detected nothing the numeric comparison did not; what it added is the kind of each alarm, and a swap it saw only before the build. A reading before the build cannot see what the build changes, such as the analysis defect. So a richer representation is not a better detector when the decisive information is missing or rewritten.

Protection costs permitted changes. Stricter checks can block edits that the requirement permits (Table 2). On the CSV compendium we made 5 maintenance edits that the witness showed change no claim value. The value lock fired on 1 of them, because it reads bytes, not values (App. N). The frozen reference was rebuilt 4 times, after the alarms and one approved update. The one-flag fix failed all 5 clean runs on our toolchain, because a rebuilt figure differs in bytes. On the agent-failure compendium, the authors' check blocked 1 of 3 maintenance edits (§4.1); on the graph-network compendium, no check blocked the reformatted README.

A surviving mutation is not always a missed error. A mutation that passes the check *survives*; it is a missed error only if a witness shows its requirement still violated in the released artifact. Pooled over our earlier injection studies (App. D), of 68 injections with a witnessed violation and a passing clean pair, the checks rejected 50, 2 escaped, and 16 are unresolved (App. C). Both escapes are on the CSV compendium: the analysis defect of §1 and its witness-sensitivity control. So a count of surviving mutations is not a count of missed errors.

4.5 From diagnosis to repair

Complementary requirements. The authors' verifier counts labels; it does not check that a label follows from its evidence (§4.2). The *re-derivation check* adds that relation: it applies the paper's decision rule to all 239 stored labels. On the first repair study's 7 claim faults, all accepted by the authors' verifier, it rejected 6 and accepted all 3 permitted changes (App. P). Each verdict follows from the check's definition. Removing a record and lowering the expected count to match passed the re-derivation check, because every label left still follows from its evidence. The frozen report

workflow	faults escaped (of 9)	permitted blocked (of 5)	after approvals
the authors' verifier alone	9	0	0
+ label re-derivation	3	0	0
+ approved record list	6	1	0
+ both (re-derivation and record list)	0	1	0
+ re-derivation, row counts and unique ids	1	1	0
+ frozen evidence bytes	0	5	3

Table 6: **What each added check protects, on the agent-failure compendium.** Escaped: accepted, with the requirement violated. Every clean run passes. All workflows: App. R.

comparison and the frozen evidence lock rejected it. The re-derivation check tests whether each label present is right, not whether a label is missing. *Consistency* (each present label follows from its evidence) and *approved membership* (the records present are the approved ones) are different requirements. A check that enforces one need not enforce the other.

A follow-up study also required the trace ids present to match a list kept outside the repository. Its 9 faults are the first study's 7 and two new ones that keep every count. Both checks together rejected all 9 claim faults we made and accepted all 5 permitted changes, one after we approved a new list. Each verdict follows from the checks' definitions. They caught disjoint faults, 6 and 3 (Table 6). Row counts with unique ids matched the record list on every edit but a record relabelled as a trace the release does not hold; so the full id list adds protection only against that count-preserving substitution. Frozen evidence bytes rejected every fault too, but blocked every permitted change. This holds under the pinned record-level contract: after the approved withdrawal, both checks passed an incomplete removal that left its raw trace; the README's old counts went unchecked.

Complementary and redundant comparisons. On a prompt-injection benchmark (Li, 2026b) by the same author, a script derives the committed aggregates from per-run results. Each derived value has three versions: committed (y_c), and computed from the candidate's inputs by the approved code (y_a) or by the candidate's code (y_k). The *aggregate re-derivation* tests $y_c = y_a$, a *candidate-code comparison* $y_k = y_a$, and a plain *rebuild-then-diff* $y_c = y_k$. Comparing the candidate's aggregation code with the pinned code on the same inputs caught a sign error, as did the rebuild-then-diff, but the aggregate re-derivation missed it; the comparison also flagged a rounding-only refactor (App. S). The aggregate re-derivation missed the sign error while the committed aggregates were stale; once they were rebuilt by the wrong code, the rebuild-then-diff missed it instead. On the evaluated runs, any two comparisons gave the same rejection coverage (by transitivity of exact equality, not a new result); which two fail adds diagnosis: whether the code or the outputs changed. The rounding-only refactor moved no number the paper defines, yet was flagged on 38 cells.

Transfer of diagnosis versus transfer of protection. We reused frozen repair specifications on the prompt-injection benchmark, which has the same verifier template. They needed 36 lines of configuration and 302 lines of new generic code (App. Q). In the fixed-verifier setting, both repairs let 2 of 5 faults through instead of 4; the aggregate re-derivation made the whole difference. We froze the repair code, then chose a compendium by other authors whose check hash-locks its inputs and byte-compares rebuilt tables. *Verdict prediction.* The contract reading, committed before the sites were drawn, predicted the native check's verdict on 10 of 10 randomly drawn faults (0 unclassified); always predicting the native check's most common verdict gets 6 of 10. The two drawn faults of each family got the same native verdict, so this baseline is weak. A simpler rule, written post hoc after the verdicts were known, asks only whether each edited file is hash-locked, rebuilt and compared, or never read by the check; it also gets 10 of 10 (App. T). So the prediction shows that the contract describes what the check sees, not that it reasons better than such a rule. *Incremental protection.* Of the 4 fixed-verifier and 2 co-edit faults that broke a requirement, the check let 1 through: a README number, as the reading said. It blocked 1 of 3 permitted edits. The two repairs the reading picked added no catch or block, nor did the third (App. T). No frozen repair reads the README: a reference outside the commit helps only if the compared observation captures the requirement. The other 4 drawn faults broke no requirement the witnesses check, and we do not count them as detections. The diagnosis transferred; the protection did not (Table 7).

requirement	covered by	same coverage from	status
result and expected value edited together	frozen report comparison	–	observed (Table 3)
each label follows from its evidence	label re-derivation	evidence bytes	observed; bytes block all edits (Table 6)
the records present are the approved ones	approved record list	row counts	observed; counts miss one relabel (Table 6)
derived values follow the approved code	two of three comparisons	any other two	observed redundancy (App. S)
a README number equals its source	numeric comparison	–	observed; no frozen repair reads it (App. T)
restatements after an approved update	none	–	untested (App. R)
stated exclusion rule matches the code	none	–	out of witness scope; post hoc: App. T.1

Table 7: **Which check covered which requirement**, from completed studies (not validated prospectively). A repair helps only where it observes the requirement and the native workflow has a gap.

5 Related Work

Mutation testing measures a test suite by the seeded faults it detects. For 73% of real faults, the test added to reveal the fault also detects a mutant from standard operators (Just et al., 2014). MLReproMutate (Shulepov, 2026), concurrent with this work, mutates 4 configuration choices, such as the seed, in ML repositories and asks whether each repository's own validation workflow notices a mutation confirmed to change the program's behaviour: the workflows detected 2 of 23 such mutations (8.7%). Executable papers rebuild an article from its data and code (Luger & showyourwork contributors, 2021), and claim checkers recompute printed statistics (Nuijten & Wicherts, 2024); Table 47 in App. V compares these lines of work. GateCheck separates rejecting a fault from the requirement holding afterwards in each artifact, and uses the difference to choose repairs (§4.5).

6 Limitations

Scope. The contract findings rest on three compendia by other authors, and the transfers on a fourth by one of those authors and a fifth by others: five cases, not rates. Except for the last study's draws, we chose every edit and the claim registry's claims after reading each compendium (App. E). GateCheck takes requirements, witnesses and contracts from the researcher; a wrong one misleads it. We have not shown that its diagnoses change a maintainer's decisions on another compendium, or that the repairs add protection elsewhere: on the fifth they added none. A witness shares the project's recipes and is not fully independent; no repository group we studied is a sample (App. W).

7 Conclusion

GateCheck separates two questions: does a workflow accept a change, and does a declared requirement still hold in a named artifact? On other authors' compendia, a reference that moves with the edit, an observation that misses the violation, or a build that fixes only the output makes the answers differ. Re-deriving labels protects consistency, and a record list protects approved membership. A diagnosis can transfer without adding protection. Judge verification by the requirements and artifacts it protects and the changes it permits, not by its rejection count.

Reproducibility Statement

Everything this paper pins is re-derived by one command, `paper2/verify_paper2.sh`: it regenerates every derived file we ship and fails if a byte comes back different, rebuilds the PDF, checks the build for errors, for a box overflowing its page by more than 5pt and for the page limit in our TeX Live 2018 build, recomputes each pinned quantity from its artifact, re-reads 42 released run logs and their verdicts, and runs an anonymity scan. It does not check the numbers nobody pinned; App. H says how many those are. The supplement ships the injection harness, the operator catalogue, every target description, every protocol with its amendments, the claim ledger of Table 12 with one row per injection, the post hoc outcome overlay, the current outcome layer and the matched studies of App. U, and every raw result file. Of the 17 repositories with a check of their own among the chosen targets and the JOSS draws of the injection studies, all but our two are public, and so are the five contract compendia (CSV_Wrangling, SGC, delta-mfp-local-agents, RepoGuardBench and UNIT); each public description records the clone URL, the exact commit and the environment. Regenerating a number from shipped records is not the same as re-running a target, so we did both for the analysis defect. A fresh replay of the defect and its clean runs from the pinned target, into a new output directory on the same machine and environment, reproduces both accuracies, the changed claim values, the exit status, the verdicts of the project's own check, the one-flag fix, the value lock, the mention pin and the directory lock (the numeric comparison was not part of that study and was not replayed), the claim files byte for byte and the paths the diff names, on a checkout verified against the pin. It is a same-environment replay, not an independent reproduction. App. X says what replays offline at each of three levels and what does not.

Ethics Statement

This work injects faults into software repositories in disposable copies. Faults went into 17 repositories with a check of their own (four chosen targets, two of them our own, and 13 drawn from JOSS's published-paper index), the five contract compendia (CSV_Wrangling, SGC, delta-mfp-local-agents, RepoGuardBench and UNIT, all with MIT-licensed code), one withdrawn target (`lifelines`), and the 15 compendia of App. M; all but our two are public, permissively licensed scientific software. We modified no upstream repository, opened no issue, and contacted no maintainer; every mutation happened in a local scratch copy that was deleted afterwards, and each copy was taken at a pinned commit that the supplement records. Every public repository's licence permits use, modification and redistribution of source for research. We report each project's catch counts as a measurement of what a suite does against defects it was not written for, never as a criticism of the project: most of what a low count holds is unresolved or out of scope (§4.4), and ours are measured the same way. Nothing here involves human subjects or personal data.

AI Use Statement

We used a large language model as a coding and writing assistant throughout: drafting and refactoring the injection harness and analysis scripts, drafting prose, and searching for related work. Every experimental result was produced by scripts in the supplement and re-run by the authors. The pinned quantities are checked against a released data file by an automated audit; the audit also reports which prose numbers are *not* pinned, and that list is in App. H.

References

Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. DeepCrime: mutation testing of deep learning systems based on real faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 67–78, 2021. doi: 10.1145/3460319.3464825.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 654–665, 2014. doi: 10.1145/2635868.2635929.

Daoyuan Li. Before the fall: Delta minimal failing prefixes for local tool-use agent failures. ICML 2026 Workshop on Failure Modes in Agentic AI (FAGEN), non-archival. `https://`

openreview.net/forum?id=KAA8FR6fEq, 2026a. Code: https://github.com/DaoyuanLi2816/delta-mfp-local-agents, commit c01fbafc, MIT licence.

Daoyuan Li. RepoGuardBench: Repository-borne prompt injection attacks and lightweight defenses for local coding agents. 5th Deep Learning for Code (DL4C) Workshop at ICML 2026, non-archival, 2026b. Code: https://github.com/DaoyuanLi2816/RepoGuardBench, commit f1c1d28d, MIT licence.

Rodrigo Luger and showyourwork contributors. showyourwork: a workflow management tool for open source scientific articles. https://github.com/showyourwork/showyourwork, 2021. Software; the article PDF is rebuilt from the figure scripts and data on every push. Accessed 19 September 2026.

Michèle B. Nuijten and Jelte M. Wicherts. The effectiveness of implementing statcheck in the peer review process to avoid statistical reporting errors. *Advances in Methods and Practices in Psychological Science*, 7(2), 2024. doi: 10.1177/25152459241258945.

Annibale Panichella and Cynthia C. S. Liem. What are we really testing in mutation testing for machine learning? a critical reflection. *arXiv preprint arXiv:2103.01341*, 2021.

Ilya Shulepov. Mutation testing for reproducibility safeguards in machine learning research software: An empirical study, 2026. URL https://arxiv.org/abs/2608.27100. Preprint, 27 August 2026. Implementation and archive released by the authors: https://github.com/ilyuka/MLReproMutate.

Gerrit J. J. van den Burg, Alfredo Nazábal, and Charles Sutton. Wrangling messy CSV files by detecting row and type patterns. *Data Mining and Knowledge Discovery*, 33(6):1799–1820, 2019. doi: 10.1007/s10618-019-00646-y. Research compendium: https://github.com/alan-turing-institute/CSV_Wrangling, commit 3e073b8e, MIT licence.

Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6861–6871, 2019. Code: https://github.com/Tiiiger/SGC, commit 2c7a2727, MIT licence.

A ROADMAP: WHERE EACH MAIN-TEXT CLAIM IS SUPPORTED

Table 8 maps each claim of the main text to the table that holds its evidence, the appendix that describes the study, the code that ran it, and the checker that re-derives its counts from the shipped records. One command, `bash paper2/verify_paper2.sh`, runs every checker and rebuilds the paper; App. X says what replays offline and what needs the pinned target checkout. The appendices after this one keep each study's full record, in the order the studies were run; the main text does not depend on that order.

main-text claim	evidence	study	code that ran it	re-derived by
a reference kept in the candidate commit moves with the edit (§4.1)	Table 3	App. O	`experiments/128_ml_transfer/run_e128.py`	`paper2/check_e128.py`
a matching aggregate does not show the record-level steps (§4.2)	Table 4	App. N.5, O	`experiments/126_population_contract/run_e126.py`	`paper2/check_e126.py`
the analysis defect passes the project's own check (§1)	Fig. 1	App. N, X	`experiments/121_consequential_protection/run_e121.py`	`paper2/check_paper2.py`
equal rejection counts, different escapes; 2 versus 0 in the outputs (§4.3)	Table 5, Fig. 2	App. N.1	`experiments/122_lock_pin_diagnostics/run_e122.py`	`paper2/check_fivefault.py`
a richer representation adds diagnosis, not detection (§4.4)	–	App. N.2	`experiments/124_typed_claim_identity/run_e124.py`	`paper2/check_paper2.py`
protection costs permitted changes (§4.4)	–	App. N	`experiments/121_consequential_protection/run_e121.py`	`paper2/check_paper2.py`
a surviving mutation is not always a missed error (§4.4)	–	App. C, U	`gatecheck/runner.py`	`paper2/check_current.py`
complementary requirements: re-derivation and an approved record list (§4.5)	Table 6	App. P, R	`experiments/130_label_rederivation/run_e130.py`, `experiments/132_membership/run_e132.py`	`paper2/check_e130.py`, `paper2/check_e132.py`
complementary and redundant comparisons (§4.5)	–	App. S	`experiments/133_candidate_computation/run_e133.py`	`paper2/check_e133.py`
transfer of diagnosis versus transfer of protection (§4.5)	–	App. Q, T	`experiments/131_repair_transfer/run_e131.py`, `experiments/134_different_template/e134_run.py`; post hoc: `experiments/review_20260926_gc/scope_reading.py`	`paper2/check_e131.py`, `paper2/check_e134.py`, `paper2/check_posthoc0926.py`
which check covered which requirement (§4.5)	Table 7	assembled post hoc from the rows above	–	–

Table 8: **Roadmap.** Each main-text claim, its evidence, the study that holds it, the code that ran the study and the checker that re-derives its counts from the shipped records. Table 7 is a summary assembled after the studies, not a separate measurement.

B NAMES IN THE MAIN TEXT, AND THEIR CODES

The main text uses descriptive names. The appendices, the protocols and the released records use short codes. Tables 9 and 10 map one to the other.

C THE CLAIM LEDGER, IN FULL

This appendix holds the historical ledger that §4.4 summarises. Table 11 sorts its surviving mutations by their evidence.

Eight labels. The historical studies gave every injection one label. *Out of scope*: the gate never opens the mutated file. *Inactive*: the mutated line never ran, or the mutation did not apply. *Unresolved*: no witness record exists ("no witness record" in Table 11), so nothing says whether anything moved. *No observed effect*: a witness ran and saw the declared quantity unchanged. *Witnessed and*

name in the main text	code	what it is
Studies		
the historical injection studies	E75–E121	the claim ledger, Table 12
the CSV compendium	CSV_Wrangling	the compendium of van den Burg et al. (2019), at one pinned commit
the analysis defect and its controls	E121	one defect, its positive control, a crash control, 5 maintenance edits and one authorized update (App. N)
the five claim faults	E122	the defect and four claim drifts, one work tree, six arms (App. N.1)
the claim-registry study	E124	E122's edits, plus a relocated claim and a deleted registry (App. N.2)
the coexisting-conditions study	E125	One compendium plus one synthetic fixture, all edits chosen by us; counts, never rates. The fixture is synthetic, written by us, and not an independent repository (App. N.3)
the graph-network compendium	SGC, E123	three README commands, three printed accuracies (App. N.4)
the population-contract study	E126	3 clean runs and 5 edits on the graph-network compendium; a controlled diagnostic, not a new repository (App. N.5)
the agent-failure compendium	delta-mfp-local-agents, E128	the compendium of Li (2026a); 16 clean runs and 12 edit runs (App. O)
the repair study	E130	the re-derivation check and the frozen evidence lock beside the agent-failure compendium's checks; 15 clean runs and 11 edit runs (App. P)
the prompt-injection compendium; the transfer study	RepoGuard-Bench, E131	the compendium of Li (2026b), by the author of the agent-failure compendium; the two repairs frozen as text, then set up with 36 lines of configuration and 302 lines of new generic code (App. Q)
the follow-up study	E132	a frozen record list beside the re-derivation check (App. R)
the candidate's own computation	E133	a third comparison on the prompt-injection compendium (App. S)
a compendium by other authors	UNIT, E134	`PsychometricsMZ/UNIT`, a rebuild-and-compare verifier (App. T)
Checks		
the project's own check; the authors' check, also called the authors' verifier on the agent-failure compendium	G0; K0 (E128, E130–E134)	the native command: `make output && git diff` on the CSV compendium; the three CI steps (tests, claim verifier, release scan) on the agent-failure compendium; the five CI steps on the prompt-injection compendium, which App. Q calls the native check
the README comparison	G0 (E123); K1 (E126)	reconstructed: each printed accuracy equals the README's at one decimal
the full recomputation	K2 (E126)	the accuracy recomputed over the declared test population equals the printed one
the loader-id check	K3 (E126)	the loader's test id list equals the declared ids, with counts
the combined check	K4 (E126)	the full recomputation and the loader-id check together
the registered-claim check	G2 (E123)	each printed accuracy equals the value registered in the E123 protocol; it never reads the README
the one-flag fix	G1a	`git diff --exit-code` after the build
the value lock	G1; K1 (E128)	the claim files after the build must equal the candidate commit
the directory lock; the mention pin	R_lock; R_pin	our two frozen repairs; run as one arm, R, in E121 and E123
the numeric comparison	G2 (E122, E124); K3 (E128, E130, E131)	each registered (file, number) count, against the frozen reference
the frozen report comparison	K2 (E128, E130); R1 (E131)	each value of the claim verifier's report, against a frozen pristine report
the re-derivation check	K4 (E130)	each stored label against the paper's decision rule applied to its stored evidence
the frozen evidence lock	K5 (E130)	the five evidence files must keep their pristine hashes
the aggregate re-derivation	R2 (E131, E133, E134)	every committed aggregate file against a rebuild from the committed inputs with the pinned derivation code
the record list; both checks	K_member; K_composed	trace ids present against a frozen list; K_member and the re-derivation check
row counts	K_count (E132)	a frozen row count per file and no trace id twice
the candidate's code against the pinned code	R3 (E133)	both run by the approved command on the same inputs, cell by cell
a plain rebuild-and-diff	RD (E133)	a rebuild with the candidate's code against the committed files
the numeric comparison cut to the registry's pairs	G2r	a retrospective reading of G2's alarm lists (App. U)
the claim registry	G3	a registry of claims with location and value, and an evidence digest; its protocol calls it typed claim identity
the variant that reports every flag	G3m	the claim registry with simultaneous flags; the variant changes the diagnosis, not the detection

Table 9: **Names and codes: studies and checks.** A study is named by what it measured; the codes are the ones its protocol and records use. The code G2 names two different checks: the numeric comparison on the CSV compendium and the registered-claim check on the graph-network compendium, K1–K3 name different checks in E126 and E128, and K0 is each compendium's own check in E128, E130 and E131.

rejected; *witnessed and surviving*. *Repaired during execution* ("regenerated; final validity unverified" in Table 11): the build regenerated the mutated value before the gate read it, and no witness compared the regenerated value with a clean run. We keep the archived label here; Table 11 does not call these rows restored, because §2 requires a witness for that. *Not a claim*: the site holds no quantity the paper states. The unresolved count is printed beside every detection count.

name in the main text	code	what it is
Edits		
the analysis defect; witness-sensitivity control; crash control	D; PC; C1	one line of `scripts/common/dialect.py`; the protocol calls PC the positive control
the table swap; the cross-file swap	S1; S2	two values swapped inside a rebuilt table; across two claim files the build does not rewrite
the abstract edit; the README edit	I1; I2	the abstract's accuracy in its claim file; the README's compute-time estimate
the reformatting edit; the moved citation block	B1; B2	evidence-preserving edits
the authorized baseline update	B4	one file retracted from the corpus, as the README allows
dropping one test node	D1b	one node dropped from the graph-network compendium's declared test population
evaluating every test node twice	DUP	the graph-network compendium's loader returns its test id list twice
using validation rows at the evaluation call	D1	the evaluation call receives the validation rows; the loader's list is untouched
max values instead of argmax; reformatting the README number	D2; B1 (E126)	a metric edit; a README edit that keeps the value
the seven claim faults; the weakened comparison alone (a control); maintenance	F1, F2, F3a, F3b, F4, F6, WF1; F5; M1–M3 (E128)	one per layer of the agent-failure compendium's claim path; WF1 is F1 with the verifier's comparison weakened, the edit §4.1 says "weakened its comparison" (App. O)
the four evidence edits; the label swap; the relabelling; the removed record; the rate change; the rewriting; the approved update	B1–B4; L2; L1; D1; I1; M1; A1 (E130)	B1 is E128's F3b; L1, D1 and A1 are co-edits (App. P)
the five gaps; the faults built for them; the control; the approved update; maintenance	G1–G5; F1–F7; C1; A1; P1–P5 (E131)	F4, F6, F7, A1, P4 and P5 are co-edits (App. Q)
a removed, a copied and a relabelled record; a reordering; an approved withdrawal	D1; X1; X2; R1; P1 (E132)	D1 is E130's; X1, X2, R1 and P1 are new (App. R)
a sign error both repairs missed, stale and rebuilt; a rounding-only refactor; a rule change	F2; F8, F9; P7; A2, A2N, KN (E133)	F9 is a co-edit (App. S)
a README number; the drawn faults; the permitted edits	FP#1; FA, FI-cell, FE, FP, FV (#1, #2); P1#1–P3#1 (E134)	FV sites are co-edits (App. T)

Table 10: **Names and codes: edits.** Each edit's code is the one its protocol and records use.

	witnessed	no witness record	out of scope	inactive	no observed effect	regenerated; final validity unverified	not a claim
surviving mutations (240)	13	109	70	28	15	4	1

Table 11: **The evidence behind the mutations the checks passed**, in the injection studies. "No witness record": no witness ran. "Out of scope": no check opens the mutated file. "Inactive": the mutated line never ran, or the mutation did not apply. "Regenerated; final validity unverified": the build rewrote the mutated file before the check read it, and no witness compared the new value with a clean run; these are accepted and unresolved. The full ledger (Table 12) has 15 witnessed-and-surviving rows, a control and an authorized update included.

The table. Table 12 is the claim ledger of the historical injection studies, E75–E121: 397 injections in 9 lines, each carrying one of the eight labels. It holds none of E122–E124, which are scored per run (Tables 5 and 43). A further 38 rows are reported in App. H instead: 25 from `lifelines`, whose gate is not deterministic, 12 from a superseded wrong-constant draw, and E121's liveness control.

The survivors. The gates passed 240 of the table's mutations. A witness saw a declared quantity move for 13 of them. Of the rest, 109 are unresolved, 70 out of scope, 28 inactive, 15 without observed effect, 4 repaired during execution and 1 not a claim. Summed down its columns, the largest ledger column is also the unresolved one, 158 of 397 rows, of which the gates rejected 49.

From 15 to 13: the bridge. The table's witnessed-and-surviving column sums to 15. The count in §4.4 is 13: the table's mutation rows that the gate passed (240 rows) and that carry this label. The difference is 2 rows, both on the E121 line, and both kept out by their ledger role, not by their label:

population	defect class or control	n	out of scope	inact.	unres.	no eff.	rej.	surv.	rep.	no claim
4 chosen targets (E76, E77)	seven portable classes	98	23	3	49	8	8	7	0	0
System A (E75)	twelve own classes and one variant	43	0	0	31	0	5	2	4	1
13 JOSS repositories (E94, E98)	prose number, weakened assertion, wrong constant	115	47	17	51	0	0	0	0	0
5 of the JOSS 10 (E96, E116)	swapped call	18	0	5	0	7	4	2	0	0
10 JOSS repositories (E96)	deleted step	32	0	5	27	0	0	0	0	0
15 public compendia (E114)	deleted directory, prose drift	22	0	0	0	0	16	0	0	6
15 public compendia (E114)	controls: blank line, comment, re-save	43	0	0	0	43	0	0	0	0
System A, held out (E107, E110)	deleted directory, prose drift	18	0	0	0	0	17	1	0	0
CSV_Wrangling (E121)	one code defect, its positive control, 5 maintenance edits and one authorized update	8	0	0	0	5	0	3	0	0

Table 12: **The claim ledger of the historical injection studies** (E75–E121). Columns after n: out of scope (no gate opens the file), inactive (the line never ran, or the mutation did not apply), unresolved (no witness record), no observed effect, witnessed and rejected, witnessed and surviving, repaired during execution, not a claim. There is no total row. E121's three surviving rows are the defect, its positive control and the intended baseline update, which the post hoc overlay scores as an authorized update, not a surviving error (App. U). The row rules are in `data/review_20260919/claim_ledger.json`.

E121's positive control (1 row), which is a control and not a mutation, and E121's baseline update B4 (1 row), which the overlay scores as an authorized update with its requirement satisfied. On every other line of the table the column and the headline count the same rows. No row with this label is kept out of the table, and no historical label was changed. The inclusion record, one line per row with its role, label and reason, is `data/review_20260924_gc/survivor_bridge.json`.

A wrong analysis is mostly not a miss. A swapped call for a confusable partner gives 18 rows: 5 inactive (2 of them void mutants), 7 no observed effect, 4 witnessed and rejected, 2 surviving. Of the gates' 12 scored misses on it, 2 changed a returned value on a call the gate made, 7 never differed on an observed call and 3 never ran (App. K).

Executed is not witnessed. We rebuilt three targets under coverage and traced twenty-three of their twenty-four weakened-assertion and wrong-constant sites. $19/23$ **sit on lines the gate runs, and none of the** 19 **was caught** (Table 22). Running is not seeing. Sixteen constants, four per repository, were perturbed by 10% across the 4 targets, and ten were called substantive by a hand label applied after the outcomes (App. I). A gate-independent probe could be built at 7 of the 16 misses and saw a change at 0 under the locked rule; only 1 of those probes can see even a tenfold change.

The outcomes, where the records allow them. Post hoc, we read each historical row under the four outcomes of §2, beside its label (App. U). A witnessed violation with a passing clean pair is 15 of 98 rows for the chosen targets, 11 of 43 for System A, and 0 of 115 for the JOSS prose, assertion and constant draws. The final state is mostly unknown, because the historical runner reset the tree after the gate: edit persistence is recorded for 12 of the 397 rows of Table 12. So the chosen targets' 7 accepted violations are accepted and unrecorded, not known to persist.

D THE OPERATORS, IN FULL

Each operator enumerates its whole site population by rule; the driver samples uniformly with seed 20260828 (single repository) or 20260829 (the cross-repository runs). Draw sizes are fixed in the locked protocol.

What we break, and why these twelve. The catalogue (Table 13) comes mostly from a record of defects shipped in one research repository over about a year, each with its cause and the published numbers it invalidated. One entry is the reason this paper exists. A tool-call parser matched balanced braces with a regular expression, so when a model wrote a JSON document *into* a string argument the call was discarded: of the stored failures whose whole reply we had kept, 43–50% were valid calls thrown away. The refutation it produced stood for months, and every gate was green.

id	where it lives	the defect	from a real one?
D1/G1	in the prose	a number drifts away from the data it came from	✓
D3	in the prose	a retired measurement is quoted as current	✓
D4/G4	in the prose	a cross-reference points at nothing	
D12	in the prose	the main text runs past the page limit	
D2/G2	in the data	a released value moves and the prose does not follow	✓
D5/G3	in the data	a released result has its sign flipped	
D6	in the data	a locked verdict is relabelled	✓
D8	in the code	a parser silently discards valid input	✓
D9	in the code	a bias correction is dropped from an estimator	✓
D11	in the code	a script default drifts from the config that made the artifact	✓
D10/G5	in the tests	an assertion is weakened to `assert True`	
D7/G6	in the release	a directory of released evidence is simply gone	✓

Table 13: The catalogue, organised by where the defect lives. The id is the operator's code: D-numbers for the twelve of the single-repository experiment, G-numbers for the portable ones Table 12 and Fig. 4 use. 8 of twelve are re-injections of defects that shipped in the repository under test, each recorded in that project's correction chronology before this study. The next table defines each one.

id	defect	site population	mutation
D1/G1	prose number drifts	decimal literals in prose files	scale by 1.05, same precision
D2/G2	released value moves	numeric leaves in released `.json`/`.csv`	scale by 1.10
D3	retired value quoted as current	the registry of retired values	write it into a current section
D4/G4	reference points at nothing	`\ref{}` occurrences	retarget to a missing label
D5/G3	released result sign flipped	numeric leaves in released data	negate
D6	locked verdict relabelled	verdict fields in result files	flip REFUTED to CONFIRMED
D7/G6	released evidence gone	data directories	delete one
D8	truncating parser	the call parser	restore the pre-fix regular expression
D9	bias correction dropped	the estimator	return the uncorrected ratio
D10/G5	test stops constraining	`assert` statements in tests	replace with `assert True`
D11	config drifts from artifact	a script default	change it away from the recorded config
D12	main text over the limit	the main text	insert filler
D–/G7	wrong constant in code	**tokenised** `NUMBER` literals in source	scale by 1.10

Table 14: D-numbers are the twelve operators of the single-repository experiment; G-numbers are the seven portable ones. G7 tokenises the source; an earlier text-matching version perturbed docstrings and is discussed in App. E. Two further portable operators were added after this table and are defined in App. G: G8, a call swapped for a confusable partner, and G9, a deleted step.

Void draws. A mutation that changes no bytes is void and redrawn up to three times. This is the analogue of the equivalent-mutant problem: a mutation that cannot change behaviour is not evidence about the checker.

E LOCKED PREDICTIONS, AND THE FIVE AMENDMENTS

Both experiments locked their operators, draw sizes, seed, scoring rule and predictions before any injection ran, and locked the scoring script before results were read. This appendix reports what we got wrong, because a pre-registration record that only contains successes is a record written afterwards.

E.1 SINGLE REPOSITORY: 3 CONFIRMED, 3 REFUTED

	locked prediction	verdict	observed
P1	overall catch rate ≥ 0.70	**refuted**	0.359
P2	a deleted test assertion is never caught*	confirmed	0.00
P3	prose-number and data-value drift each ≥ 0.75	**refuted**	0.00 and 0.33
P4	no false alarms in five clean re-runs	confirmed	$0/5$
P5	at least one operator is caught 0.00 of the time	confirmed	five of them
P6	a deleted evidence directory is always caught	**refuted**	0.00

*P2's wording is as locked. The operator replaces the predicate with `assert True`, so what it makes is a *weakened* assertion, and that is the operator's name everywhere else in this paper. We do not rewrite a locked prediction to match a later, better name for the thing it predicted about; a reviewer was right that the two names were inconsistent. This row and the sentence in App. H that records the rename are the only two places the old name appears at all, and a check in `paper2/check_paper2.py` fails if a third one shows up.

P2 and P5 predicted that our own machinery would *fail*, and were locked so that finding a blind spot could not afterwards be presented as something we set out to find. P6 is the one discussed in App. H (Fig. 3).

The ordering is checked, not asserted. Every claim in this paper about a protocol coming first rests on git history, and until now no gate read it. One does: `experiments/50_consistency/check_prereg_order.py` maps each artifact to the protocol version whose hash it records, and fails if that protocol commit does not precede the artifact's. It covers the 99 artifacts this archive ships and runs in the verification chain. (Our working tree holds one more, the concurrent submission's frozen record, which this archive does not carry; counting it here would print a number the archive cannot reproduce, and for one version it did.) In a tree with no history — the release archive — it says so and skips, rather than passing silently.

E.2 THE REVIEWER-DRIVEN EXPERIMENTS, AND THEIR VERDICTS

Each was locked before running, in the same way and for the same reason. The ledger is split across Tables 15 and 16 only so that each fits a page; every row and verdict is as scored.

Two of these refutations are worth more than the confirmations around them. E87's P1 failed because our rule said `pytest` reads test and source files, and Pingouin ships its datasets *inside* the package, so `pytest` reads those too; Amendment 1 is labelled as an amendment and both verdicts stand in the record. E89's P2 failed on the plotting line we said before running we were least sure about — two of four perturbed constants sit on lines the suite never runs, which is exactly the gap between "the gate reads the file" and "the gate runs the line".

E.3 ACROSS REPOSITORIES: EVERY LOCKED PREDICTION AND ITS VERDICT

Fifteen predictions were locked for the two cross-repository experiments and scored in their result files. Until now this subsection named none of them, which is not what its heading promises.

The two void entries are the useful ones.

Amendment 3: an operator that measured the wrong thing. The wrong-constant operator matched decimal literals as text. All four sites drawn on Pingouin landed inside docstrings, so it

Table 15: Locked predictions and verdicts, part one: the reachability, coverage and witness experiments on the four targets (E87–E95), and the two frame-sampled runs (E94, E96).

	locked prediction	verdict	observed
E87 P1	every catch is on a file-reachable cell	**refuted**	3 Pingouin cells
E87 P1′	the same, under Amendment 1	confirmed	0 violations
E87 P2	G1 is unreachable on the two `pytest`-only targets	confirmed	both
E87 P3	G5 and G7 are reachable on all four	confirmed	$8/8$
E87 P4	the rule flags the withdrawn lifelines catch	confirmed	it does
E89 P1	the weakened-assertion sites are on executed lines	confirmed	$4/4$
E89 P2	at least 3 of 4 constant sites are executed	**refuted**	$2/4$
E89 P3	no executed constant site was caught	confirmed	0 of 2
E90 P1	both executed constant sites change a computed value	**refuted**	$1/2$
E90 P2	the unexecuted plotting site changes nothing	confirmed	$0/1$
E91 P1	$\geq 60\%$ of traced sites are on executed lines	confirmed	$19/23$
E91 P2	every traced weakened-assertion site executes	confirmed	$12/12$
E91 P3	no executed site was caught	confirmed	0 of 19
E91 P4	another target hides sites behind skips	**refuted**	only Ruptures
E93 P1	no covering test asserts on the changed value	confirmed	0 of 3
E93 P2	a covering test compares the output to itself	confirmed	`test_seed`
E93 P3	weakened-assertion sites are assertion-reachable	confirmed	$4/4$
E95 Q1	under half the executed constant sites reach a value	confirmed	$1/7$
E95 Q2	no gate measures test strength	confirmed	$0/3$ gates
E95 Q3	some covering test classifies *self*	**refuted**	our rule cannot see it
E95 Q4	the unclassifiable bucket is not empty	confirmed	$2/7$
E94 N1	some new repository catches something	confirmed	3 of 10
E94 N2	a prose number is never caught	confirmed	$0/38$
E94 N3	G7 caught more often than G5	confirmed	0.29 vs 0.00
E94 N4	fewer than half of candidates are eligible	confirmed	$14/120$
E94 N5	ten eligible within sixty examined	confirmed	11
E94 N6	under half of drawn sites assertion-reachable	confirmed	$3/90$
E94 N7	at least one drawn site is an equivalent mutant	VOID	built later, control failed
E96 W1	G8 caught more often than E94's G7	**refuted**	$4/16$ vs $8/28$
E96 W2	silent G9 caught less often than G8	**refuted**	$2/3$, three sites
E96 W3	executed unbound-name deletions all caught	confirmed	$22/29$, 0 reachable misses
E96 W4	at least half of drawn sites are dynamically reachable	confirmed	$27/50$
E96 W5	under half of reachable G8 sites assertion-reachable	confirmed	$0/10$
E96 W6	some repository catches neither	confirmed	6

Table 16: Locked predictions and verdicts, part two: the stratified prose draw, the two repairs, the baseline tool, and the three semantic procedures (E100–E116). †: see the note under the table.

	locked prediction	verdict	observed
E100 Y1–Y3	a covered prose number is caught more often	confirmed	1 vs 0
E105	*post hoc*: a reclassification, no predictions	—	—
E107 R1	≥ 7 of 8 held-out deletions caught	confirmed	$8/8$
E107 R2	it catches every site the un-repaired gate passed	confirmed	$5/5$
E107 R3	no declared benign edit trips the gate	confirmed	$0/6$
E107 R4	clean passes; runtime within 25%	confirmed	57 vs 68 s
E110 M1	≥ 9 of 10 held-out drifts caught	confirmed	$9/10$
E110 M2	at most 2 are caught without it	confirmed	$0/10$
E110 M3	no declared benign edit trips it	confirmed	$0/5$
E110 M4	it adds under 2 s	confirmed	$+0.1$ s
E112 M1	the baseline tool behaves as its own README says	confirmed	both fixtures
E112 M2	no `(file, line)` is a candidate for both tools	confirmed	0 shared
E112 M3	it finds nothing on ≥ 3 of the five targets	**refuted**	2 of 5
E112 M4	everything it finds survives the target's gate	**refuted**	9 of 88 killed
E112 M5	every target has an operator it cannot express	confirmed	all 5
E113 W1	a probe can be built for ≥ 12 of 20 sites	**refuted**	11
E113 W2	at least half the probed sites are witnessed	**refuted**	$0/11$
E113 W3	at least one scored miss is witnessed	**refuted**	0; 3 post hoc
E113 W4	the rule reproduces E90's hand-written labels	**refuted**	2 of 3 disagree
E113 W5	no site is unstable across two clean runs	confirmed	0
E114 R1	at least half the compendia check nothing	confirmed	$11/15$
E114 R2	the directory lock catches $\geq 80\%$ of deletions	confirmed	$13/13$
E114 R3	the mention pin catches $\geq 60\%$ of drifts	confirmed†	$9/9$
E114 R4	no benign edit raises a false alarm (three trivial control types)	confirmed	$0/43$
E114 R5	both checks run in under 5 s per repository	confirmed	0.62 s
E115 S1	≥ 15 of 20 sites get a usable pairing	**refuted**	11
E115 S2	at least half the decided sites are necessary	**refuted**	$0/11$
E115 S3	at least one site is redundant	confirmed	11
E115 S4	no target's clean gate disagrees with itself	confirmed	0 unstable
E116 N1	≥ 6 of 8 `nansum` swaps execute	**refuted**	5
E116 N2	at most 2 executed swaps ever see a NaN	confirmed	0
E116 N3	≥ 6 of 8 swaps are equivalent	**refuted**	5; 3 never ran
E116 N4	at least one scored site changes a value	confirmed	6

†E114 R3 is confirmed on the locked count, 9 of 9. Post hoc, only 3 of the 9 seeded draws were numeric statements, and those 3 were caught; the other 6 were identifiers, versions, references or a figure width (App. M).

Table 17: The cross-repository predictions, as locked and as scored. VOID marks a prediction whose wording did not say which of two rates it meant; NOT RUN marks one that could not be executed, because lifelines' gate is non-deterministic and App. H computes that it would need about 62 runs per arm to decide. "Partial" means the prediction was locked over two new targets and only one proved measurable — it passed there, and we do not round that up to a pass.

	locked prediction	verdict
E76 P1	System B catches less than System A	confirmed
E76 P2	Pingouin catches more than System A	VOID
E76 P3	a prose number is caught nowhere	confirmed
E76 P4	a moved released value is caught $\geq 3/4$ in Pingouin	**refuted**
E76 P5	a weakened assertion is caught nowhere	confirmed
E76 P6	no clean re-run raises a false alarm	confirmed
E76 P7	a deleted directory: caught in Pingouin, missed in System B	confirmed
E76 P8	a wrong constant is caught $\geq 3/4$ in Pingouin	VOID
E77 P1	a prose number is caught in neither new target	partial
E77 P2	a weakened assertion is caught in neither	partial
E77 P3	a wrong constant is caught in neither	partial
E77 P4	a moved released value is caught $< 1/2$ in lifelines	NOT RUN
E77 P5	no clean re-run raises a false alarm	partial
E77 P6	each new target has at least one blind spot	partial
E77 P7	the new targets' surface rate is under 0.40	partial

perturbed example output rather than code. The prediction that depended on it is reported VOID — as run it could not test what it was locked to test — and the corrected, tokenising version is reported separately as exploratory, taking no pre-registration credit.

Amendment 5: a prediction that named no estimand. "Pingouin's catch rate exceeds System A's" never said *which* rate, and the two readings disagree (App. H.1). Reported VOID rather than scored on the reading that suited us.

Amendments 1, 2 and 4: the baseline rule earning its place. A target that does not pass its own gate before injection is refused outright, because every injection would then read CAUGHT for the wrong reason. It fired three times in one afternoon: on a `PATH` that resolved the interpreter to one without the test runner; on an exclusion rule of ours that stripped the paper's figure files so the build failed on a missing figure; and on System A's own completeness guard correctly refusing a results directory we had just created. Without it System B would have scored $30/30$ and looked better verified than System A.

Amendment 4 records a harness defect we audited *before* fixing. Leaf paths inside released data were dotted strings, which break on a key that itself contains a dot. The question that matters is not whether it errored but whether it could ever have written to a *different* leaf than it read from, silently. Round-tripping all $18{,}274$ numeric leaves: zero silent mis-targets, always a raised error, always excluded from the denominator. No reported rate is affected. The residue is a sampling bias — the affected keys are model names, so per-model results are under-represented in what those two operators tested — and it is disclosed rather than absorbed.

F TEN MORE REPOSITORIES, DRAWN FROM SOMEONE ELSE'S LIST (E94)

Four repositories was the honest ceiling of this paper, and two of them were ours. Reviewers kept naming the same two problems, and both are fair: the count is small, and *we picked the four*. A reader cannot tell from four hand-chosen targets whether we chose ones with weak gates.

This experiment fixes the second problem first, because the first is worthless without it.

We did not choose the repositories. The population is peer-reviewed research software written in Python. The frame is the Journal of Open Source Software's own list of published papers, snapshotted to `data/e94_frame/joss_frame_snapshot.json`: **2000 papers**, of which **1357**

target	what it is	its gate	whose gate
System B	a compendium: paper, data, tests, and **no checking machinery**	tests + a clean build	reconstructed
System A	a compendium with $\sim 6{,}592$ lines of bespoke gates	its documented verifier	its own
Pingouin	a peer-reviewed statistics library, unrelated authors	`pytest`	its own CI
Ruptures	a change-point detection library, unrelated authors, 652 tests	`pytest`	its own CI

Table 18: The four measured targets. Pingouin and Ruptures are public, with clone URL, commit and environment recorded. System B is a private repository by the same authors as System A; we mark it so rather than presenting it as something a reader can fetch. A fifth target, `lifelines`, was attempted and withdrawn (App. H).

declare Python in the paper's own metadata. JOSS reviews every submission and each record carries the authors' repository URL. We did not decide who is on that list or in what order.

We sort by DOI so the order cannot depend on when we fetched, shuffle with one fixed seed, and then walk the list. That is the whole selection rule. Every candidate we look at is reported, in order, with the first test it failed.

What the frame is not. It is not all scientific software. It leans toward projects whose authors wanted a software paper, and toward recent years. Our claim is about what this frame covers and we do not stretch it further.

Seven tests, all applied before any defect exists. A candidate has to pass all seven:

- it clones from `github.com`;
- it has a `pyproject.toml` or `setup.py` at the root;
- it has a CI workflow that runs a test runner — **the gate has to be the project's own**, because a gate we wrote would be our artifact and not theirs;
- it has at least one test file;
- it installs into a fresh virtual environment exactly as its own CI does;
- it **passes its own suite three times in a row**;
- each of those runs finishes inside fifteen minutes.

None of these tests can see a catch.

We wrote down the two biases they create *before* running anything, including the one that does not flatter us:

1. Keeping only repositories whose gates *work* keeps the ones more likely to catch something. That pushes against our hypothesis. Conservative.
2. The cap, five minutes when we wrote this, keeps only fast suites, and a fast suite is usually a smaller one, which is likely to catch *less*. That pushes for our hypothesis. **Not** conservative. So we report how many candidates were dropped by the time cap alone, and a reader can size the hole instead of taking our word that it is small.

What we changed before any defect existed, and why we are allowed to. The protocol was amended four times between locking it and injecting the first defect, and every amendment commit precedes the first result file — which is the property that matters and the one `git log` shows. Three of the amendment commits also carry the protocol's hash in their message; the others do not, and we do not claim otherwise. Each amendment was triggered by something that cannot see a catch, and each makes our claim *harder* to hold, not easier. The first is the one that produced our 120 candidates: if fewer than ten eligible targets appear within sixty examined, the cap extends in blocks of sixty to at most 240, with the walk order, the seed, the eligibility tests and the primary estimand unchanged. The final ledger shows eleven eligible inside the first sixty, which looks like the trigger

never firing. It fired under the harness as it then was, before Amendments 2 and 3 fixed the install extractor and the gate command; every install-or-suite rejection was re-screened afterwards, and eligibility rose. We extended the cap on what we knew at the time and report the screen as it ended.

1. **The suite budget went from five minutes to fifteen.** At five, four of the first seven candidates that reached the suite test were killed by the clock and none was eligible. The cap had become the selection rule, and it selected the smallest suites — the weakest gates. A longer budget admits better-tested packages, which are more likely to catch.
2. **The gate became the project's own, literally.** Our harness had installed only declared extras and run bare `pytest`. Projects' CIs install `pytest-cov pyflakes` on the next line, or `-r requirements.txt`, or `.[test]` with the extra declared in `setup.cfg`, and run `pytest tests/`. We had rejected repositories for `ModuleNotFoundError`s of our own making. The harness now reads the install lines and the test path out of the workflow file the eligibility test already matched, and records both per target so a reader can compare what we ran against what we read it from.
3. **Our extractor had read YAML comments as package names.** "Install the package in editable mode" became sixteen packages; when that install failed we fell back to `pytest` alone. Fixed, and every install-or-suite rejection was re-screened once under the final harness so no repository is excluded for an omission of ours.

Two failure shapes no single environment can reproduce were named before injection and are reported as *unmeasurable here*, not as rejections: a CI whose matrix jobs install mutually exclusive backends (`jax` in one, `tensorflow` in another), and a CI that provisions a headless browser. They are limits of the measurement and carry no weight in either direction.

Three operators, the ones that travel. A number in the prose drifting from its data (G1); a test that still passes but no longer constrains anything (G5); a wrong numeric constant in the code that computes results (G7). These are the only three common to all four original targets, and the only three a general Python package is guaranteed to have surfaces for. We do *not* run the operators that need bundled data or a release manifest: including them would make the denominators incomparable across targets, which is the mistake the original catalogue made.

Four sites per operator per target, drawn by rule with a fixed seed. Nothing hand-picks a site.

The measurement is written down before the gate runs. This is what the existing corpus cannot do, because its outcomes are already known. For every drawn site, before its gate is executed even once, we record the exact byte change, what a reader would have to look at to see the change, and all three reachability layers — does a gate command read the file, does the clean suite execute the line, and can a covering test's assertion see the change. The prospective record goes to disk first; the gate runs second.

The primary number is a repository, not an injection. For each target: did its own gate catch *at least one* semantically active injection? We report how many repositories caught something, out of the total, with an exact upper bound, pooled with the original four on the same three operators. The unit is the repository because that is the unit a reader generalises over — more injections inside one repository do not add a repository.

The words "semantically active" are the estimand's weak point, and N7 is why. The protocol promised a mechanical check that every drawn site changes something a caller could observe, and we did not build it, so the qualifier was never evaluated. A catch proves activity, so the numerator is safe: each of the 3 catching repositories caught a live defect. The denominator is not. A repository counted as non-catching may have been handed an inert mutation, which would make the bound optimistic in our favour. That is the cost of not having the check when N7 had to be scored, and it is why N7 is VOID rather than approximated.

We built the check afterwards. It failed its own control, twice. E97 asks the question directly. Replay each recorded wrong-constant diff, run the project's own gate, and compare what the gate prints. Before any of that, run the gate twice on the untouched tree. If the same tree prints two different things, the instrument cannot see what a mutation did, and the site is unmeasured. That

control failed on all 7 of the 7 repositories. So all 28 sites are unmeasured and all five locked predictions fail, including the positive control that a recorded catch must change what the gate prints.

The cause is not our comparison. In one project we recorded what moved: `257 passed` both times, and a coverage table in which a block of the project's own lines runs on one run and not on the other (`data/e97_semantic_witness/diagnosis.json` in the supplement). Dropping that table, which is what Amendment 2 does, made 5 of the 7 suites stable. So the coverage table was the whole instability in most of them, and not the whole story in the other 2.

So we narrowed the instrument, and we locked the narrower version before running it (Amendment 2 of `experiments/97_semantic_witness/protocol.md`). It compares only what the suite says about its tests: the progress line, the pass/fail counts, and any failure text. It drops the coverage table. That makes the witness less sensitive, and less sensitive is the direction that hurts us — more of our own denominator will look inert.

The narrowed run covered all 7 repositories. 5 are output-stable and 2 are still not, so T0 fails again. T1, the positive control, holds: all 8 sites E94 recorded as caught changed what the suite prints. T2 fails — 0 of the 12 *measurable* misses changed anything the suite reports. The other misses sit in the two repositories that failed T0, and they carry no label at all.

Narrowing cost the witness its independence, and we should say so plainly. The narrowed comparison reads the progress line, the pass/fail counts and any failure text. A pristine suite passes, so almost the only way to change any of those is to make a test fail, and a failing test is what the gate scores as a catch. The labels say exactly that: the witness and the gate disagree on 0 of the 20 sites that carry a label at all — every catch is witnessed and every measurable miss is not. So T1 and T2 are close to restatements of the catch outcome, not independent evidence that a mutation is semantically active. That is the price of Amendment 2: the locked comparison could have answered the activity question and failed its control, and the narrowed one survives the control by asking a question whose answer we already had. Amendment 2 says what to do when T0 fails, which is stop, so we stop. No label from the narrowed run moves a number in either paper. The scorer enforces that: with T0 failed it refuses to compute an estimate restricted to the stable repositories, which is the rescue we would otherwise be tempted by.

Two things are worth keeping. N7 stays VOID and the primary is unchanged. And the reason nobody has an outcome-blind activity check for real repositories is now measured rather than asserted: real suites do not print the same thing twice. 7 of 7 fail that test on their whole output, and 2 of 7 still fail it on their test report alone. T2's zero says the missed constants changed nothing the suite reports, which given the collinearity above is another way of saying they were missed. One number from this experiment does earn its place in the method. All 7 pairs of pristine runs agreed on *exit status*, including the 2 whose text moved. That is why an injection method scored on exit status survives on these projects where one scored on output would not, and it is the only thing the papers take from E97.

The starting point, for scale: on these three operators the original 4 caught $\mathbf{0}$ of 56 injections and $\mathbf{0}$ of 4 at the repository level, an exact upper bound of $\mathbf{0.53}$. Ten more all-zero repositories would take that to 0.19; it would take about **twenty-six** before an upper bound near 0.11 could support a strong prevalence claim. We are buying one step, not the answer.

F.1 WHAT THE TEN FOUND

Screening. 120 candidates examined in protocol order. 14 were eligible and 106 were rejected. The rejections, in the order of the seven tests: 7 did not clone from `github.com` (five were not GitHub URLs, two failed to clone), 16 had no installable package at the root, 19 no workflow that invokes a test runner, 1 no test file, 31 would not install here, and 32 did not pass their own suite three times here. None lacked an operator site. 7 of that last 32 were killed by the fifteen-minute cap alone. Those 7 are the non-conservative bias, sized: a slower suite might have passed with more time. Every rejection is in `data/e94_frame/screening.json` with the first test it failed; that file is the ledger written before any injection, and the re-screen the injection driver ran as it rebuilt each target ships beside it as `screening_at_injection.json`. The locked prediction N5 — ten eligible within sixty examined — held. One substitution, by rule but after outcomes existed:

PySLSQP passed screening at 655 s and hit the 900 s cap when rebuilt for injection, after eight targets had run; the eleventh eligible in protocol order, rfbzero, was injected, and it caught four. We did not choose it, the order did, but the decision post-dates eight outcomes, so the primary is reported with and without it: 3 of 10 and 2 of 9; pooled, 0.47 and 0.41.

Which repositories could have caught it. A chance statement for this frame uses the 11 repositories that drew a wrong-constant site. The other 3 drew none, so no gate of theirs could have caught one, and counting them as non-catchers would credit our own empty cells to the projects: that urn gives the larger P=0.33 (0.46 without the substitute). Both values are in the result file. We report the smaller one, which is the one less favourable to us. The restricted urn was computed after the outcomes, like the chance check itself.

Where the prose result actually comes from. Of the 62 prose-number injections across all fourteen repositories, 12 sit in a file some gate opens: the two targets that build a paper. The other 50 are in files no gate reads, because every other gate is pytest. Both groups were missed, but they say different things. The first is a gate that reads the file and does not check the number. The second is a gate that never looks.

What the prose-number operator actually perturbed. G1 is named for a number in the prose drifting from its data. The operator implements that as: perturb any decimal literal in the prose files. What that draws depends on what the prose holds, and in this frame most decimal literals are identifiers. Reading all 38 drawn literals at their pinned commits: 19 are DOI prefixes, 4 are arXiv identifiers, 9 are version or licence numbers, and 6 are numbers that could drift from data — three of those six are cells in one project's results table. So the class this frame supports is narrower than its name: *no gate in any of these repositories compares a number in prose against anything*, identifier or measurement. The claim we are entitled to is that one; the sharper claim, that a drifting *measurement* is never caught, rests on 6 injections here and on the original four's prose, where the literals are mostly measured values. The classification is in data/e94_frame/g1_literal_kinds_posthoc.json. The operator now skips a literal that reads as a DOI, an arXiv identifier, a version or a licence, with a test that fails if the rule is removed. That changes later runs, not this one: the 38 injections reported here were drawn before the rule existed, and we report them as drawn.

The weakened-assertion operator edited the wrong line 5 times. Its site closure asked the substitution helper for the *first* match of the drawn line's text, not for the drawn line. Where a test file repeats an assertion verbatim, the mutation therefore landed on the earlier copy. Re-reading the six repositories at their pinned commits: 5 of the 24 injections edited a line other than the one their site names, and 1 pair produced a byte-identical mutant, so the 24 injections are 23 distinct ones. Every one was missed either way, so no catch count moves; what moves is the independence of the draw and the line numbers in the record. The operator is fixed, which changes later runs and not this one, and the measurement is in data/e94_frame/g5_site_collapse_posthoc.json.

One stale label, left as written. Every *suite* timeout rejection in both ledgers carries the text suite TIMEOUT >300s; the two install timeouts print their own budget correctly. The cap was 900 s. That string was a literal in the first screener, and Amendment 2 raised the constant without updating it; the shipped screener prints the constant, so a re-run writes 900. The field to read is the recorded return code, 124: PySLSQP was killed at 900.1 s. The durations in those rows were taken on the wall clock, and three of them read far above the cap because the machine slept mid-run — the screener records a monotonic duration now. We left the text alone. Both ledgers were written before the outcomes they precede, and editing a record after its outcomes exist is the thing this paper is about.

Per repository. Each gate is the command the project's own workflow runs, on the path it names. A dash means *our rule drew no site there*, which is not the same as the project having none. 7 cells are dashes, and only 1 of them is the project's doing: pysymmpol's source was searched and holds no constant our rule accepts. The other 6 are ours. 2 of the 4 repositories with no weakened-assertion site had no test file *that we found*: our discovery looked under tests/, and PYroMat keeps its tests in src/test/, which is the same gap the coverage re-measurement later fixed — its one catch

below fails in `src/test/test_units.py`. In the other 2 we read the test files and found no single-line `assert` statement to blank; both write their checks another way. 2 of the 3 repositories with no wrong-constant site had no source file searched at all, because no package resolved from their metadata. We report the counts as scored, and we say which cells our own instrument emptied.

The four the stopping rule left, run under predictions locked first. The screen found 14 eligible repositories and the stopping rule took the first ten, so four eligible repositories were screened and never injected. Extending to them costs no new screening and offers no choice: the order was fixed by the seed when E94's screen was run, before E98 existed. We locked six predictions and a fallback for each before running any of them (E98, `experiments/98_frame_extension/protocol.md`).

	prediction	outcome
X1	no prose number is caught	**confirmed**: 0 of 9
X2	no weakened assertion is caught	**confirmed**: 0 of 4
X3	at least one catches a wrong constant	**confirmed**: 2 of 3, 2 of 12 injections
X4	all four still pass their own suite three times	**refuted**: 1 timed out at 900 s on the re-screen, after passing the screen
X5	our rule finds an assertion to weaken in at least two	**refuted**: 1 of 3
X6	fewer than half the prose sites are file-reachable	**confirmed**: 0 of 9

The fallbacks were written before the outcomes and we ran them as written. X4's failure removes `PySLSQP` from every denominator here: it installed and passed three times at the screen and its suite exceeded the budget at the re-screen. The elapsed time between the two does not change that outcome; we did not diagnose the cause and did not repair it — a gate we fixed would be our artifact. X5's failure is a limit of our own rule, so the weakened-assertion denominator shrinks rather than the missing sites counting as misses.

What the extension adds is worth stating plainly, because it cuts both ways. The two universal misses stayed universal on repositories we had never touched: 0 of 47 prose numbers caught across the frame-sampled repositories, and no weakened assertion caught anywhere in them. That count is the frame's, not every prose injection this paper reports — the four repositories we chose add $0/6$, $0/6$, $0/6$, $0/6$ more, and E100's stratified draw on this paper caught one (App. H). The wrong-constant rate went the other way. 2 of the 3 caught one, which is a higher rate than the first ten, and pooled over every public target the count is 5 of 13 repositories and 10 of 40 injections. E94's locked primary is unchanged and is reported first; this is the extension, and it says our earlier estimate of what a suite catches was if anything pessimistic.

One cell our instrument could have filled wrongly. Python's `glob` descends into directories whose name begins with a dot, so the weakened-assertion rule's pattern `**/tests/*.py` also reaches `.venv/lib/.../site-packages`: it could draw a site inside an installed dependency and score the project on somebody else's test suite. In the repositories reported here it never did. Every site in every *scored* result file is a file in that project's own tree, across 165 recorded runs, and `check_paper2.py` checks it on every one rather than asking to be believed. One shipped file is exempt and says so in its own directory: the superseded first pass, kept precisely because its four sites were inside `numpy`, `matplotlib` and `h5py`. The gate skips any directory named for its supersession, which is why that file can be shipped as evidence of the defect without being counted as a measurement. We did not notice this by reading the code. We noticed it when the rule was pointed at four more repositories and drew four sites inside `numpy`, `matplotlib` and `h5py`, two of which were scored as catches; those runs are void, they are kept in the supplement with the reason, and the fix is one exclusion in the one function every operator's population passes through. A defect in the instrument that measures defects is the ordinary case, not the ironic one, which is why the check now runs beside the others.

repository	its own gate	prose number	weakened assertion	wrong constant
`kooplearn`	`pytest -q tests/`	$0/4$	$0/4$	$0/4$
`phylox`	`pytest -q`	$0/3$	$0/4$	$0/4$
`PYroMat`	`pytest -q`	$0/3$	–	$\mathbf{1/4}$
`rfbzero`	`pytest -q`	$0/4$	$0/4$	$\mathbf{4/4}$
`scikit-topt`	`pytest -q`	$0/4$	–	–
`w2w`	`pytest -q`	$0/4$	$0/4$	$\mathbf{3/4}$
`nilt`	`pytest -q`	$0/4$	$0/4$	–
`solar-data-tools`	`pytest -q tests/`	$0/4$	–	$0/4$
`pysymmpol`	`pytest -q`	$0/4$	–	–
`pref_voting`	`pytest -q tests/`	$0/4$	$0/4$	$0/4$

The catches, one by one. Each bullet names the tests we read at the pinned commit, not every test the run failed; the result files hold each run's full failure list.

- **PYroMat**, `units.py`: the constant 1.8 became 2.0. That is the Rankine factor. `test_temperature_scale[R]` compares the conversion both ways against known temperature pairs and failed.
- **rfbzero**, `redox_flow_cell.py`: four default arguments of the cell model — transfer coefficient 0.5, mass-transfer coefficient 0.8, roughness 26.0, temperature 298.0 — each perturbed by 10%. `test_cc`, `test_cv1` and `test_cccv_symmetric_cell` build a cell from those defaults and assert on its simulated curves; all three failed, four times out of four.
- **w2w**, `w2w.py`: 100.0 became 110.0 in a routine whose output columns must sum to 100; `test_scale_hi` asserts exactly that, with `approx`. Two resampling constants ($2.0 \to 2.2$) failed the resampler's output-value tests.

Every one is an assertion on an output *value* that the constant feeds. None of the sixteen constants in our original four had one.

Predictions. Six of seven were scored, and all six held. N1 — at least one new repository catches something — was the one we half expected to lose; three did. N2, a prose number is never caught: 38 of 38. N3, a wrong constant is caught more often than a weakened assertion: $8/28$ against $0/24$. N4, fewer than half of candidates are eligible: $14/120$. N5, ten eligible within sixty: eleven. N6, under half of drawn sites covered by a test that asserts a number, by the automatic rule: 3 of 90. **N7 could not be scored.** It said at least one drawn site would be an equivalent mutant — a change overwritten before any observable moves — and promised a mechanical check on every site. That check did not exist when N7 was scored; we built it later and it failed its own control. An earlier draft scored a different sentence under N7's name; we say so, and the primary denominator may therefore include equivalent mutants, which can only lower the catch rates reported.

The reachability record, before and after. For every site we wrote down three yes-or-no facts before its gate ran. Does a gate command read the file? Does the clean suite execute the changed line, as measured by coverage? Does a test that executes the line assert a number the change can move, as classed by an automatic rule that reads each covering test's assertions? We call these the file, dynamic and assertion layers.

The record was wrong where it mattered most. 7 of the 8 sites that were later caught had been marked "not executed" or "not measurable". A suite cannot fail on a line it never runs, so those marks were measurement gaps, and we found four:

- coverage records a multi-line signature or statement against its first line, so a default argument or a closing paren on a continuation line read "not executed" while the code ran (9 sites);
- one project's coverage run died on a dependency's Cython source and every site went unmeasured;
- one project keeps its tests in `src/test/`, where our discovery had not looked;
- one project's own `--cov` options started a second coverage inside ours, and ours saw nothing.

corpus	prose number	weakened assertion	wrong constant
the four we chose (4)	0/24	0/16	0/16
drawn by seed from JOSS (10)	0/38	0/24	8/28
the extension that still builds (3)	0/9	0/4	2/12

Table 19: **Every repository injected, and what its own gate caught**, as caught/scored, repository count in brackets. The extension is three of four; the fourth no longer passes the build screen. Counts are never added across corpora.

The automatic assertion rule did no better. It marked three sites as covered by a number-asserting test; all three were missed. It marked none of the eight that were caught. On this data the label has no predictive value, and the paper does not lean on it.

Measured again, with the gaps fixed. Gate outcomes were not touched; only the measurement was re-run. 41 of the 90 sites are now executed by the suite (18 newly so), every target has a coverage record, and **all 8 catches sit on executed lines**. 8 of the executed sites carry a covering assertion the rule classes as number-asserting. The prospective record ships unchanged beside the re-measured one, so a reader can see exactly what the four fixes changed. N6 was scored on the prospective record and stands as scored.

G A WRONG ANALYSIS AND A DELETED STEP (E96)

A reviewer put it in one sentence: the catalogue cannot express a wrong analysis or a deleted analysis step, two important scientific failure classes. Every operator we had perturbed a number, a document, a test or a manifest. None changed *what is computed*. A researcher who calls the wrong test, or drops a normalisation step, has produced a wrong result with every number in place.

Locked before the outcomes it sits beside. The protocol (`experiments/96_wrong_analysis/protocol.md`, hashed in its commit) was written while E94's injection run was on its first target. Two of E94's outcome lines had scrolled past — both documentation edits on `pref_voting`, both missed, a class a `pytest` gate cannot read by construction. No other outcome and no result file had been read. Nothing below depends on those two lines and nothing below could have been tuned to them.

Two operators, by rule. Both enumerate their sites with `ast` over the package's source, never its tests; the driver draws four per operator per target with a fixed seed.

G8, a wrong analysis. A call whose callee resolves by name to the left column of a closed table is replaced by its partner: `mean`→`median`, `std`→`var`, `nanmean`→`mean`, `sum`→`nansum`, `ttest_ind`→`ttest_rel`, `pearsonr`→`spearmanr`, `linalg.norm`→`abs`, `floor`→`ceil`, `argmax`→`argmin`, and `sorted(x)`→`sorted(x, reverse=True)`. Each pair is one a reader could miss: the same signature, a plausible answer, the wrong one.

G9, a deleted step. One assignment inside a function body, whose target is used later in that body, is removed. If the name was already bound — an argument, an earlier assignment — the function still runs and returns something plausible and wrong; we call these *silent*, and they are the class the reviewer meant. If the name was not bound, Python raises `NameError` when the line runs; these are the *positive control*: a gate that executes the line *must* catch them, or the harness is broken.

Same ten repositories, same commits, same gates. E94's targets, rebuilt from their pinned commits with the install recipe and gate command E94 recorded. Every rebuilt tree's clean gate passed before a defect was injected — which also serves as the fresh clean control for the three repositories that caught E94 injections. Every site's reachability record was written before its gate ran.

repository	swappable calls	wrong analysis	deleted step, silent	deleted step, raises
`kooplearn`†	17	0/2	–	**1/4**
`phylox`	2	**2/2**	–	**4/4**
`PYroMat`	12	0/4	0/1	**2/3**
`rfbzero`	0	–	**1/1**	**3/3**
`scikit-topt`	0	–	–	–
`w2w`	0	–	**1/1**	**3/3**
`nilt`	0	–	–	–
`solar-data-tools`	80	0/4	–	**1/4**
`pysymmpol`	0	–	–	**4/4**
`pref_voting`	386	**2/4**	–	**4/4**

† 2 wrong-analysis sites are void (a malformed mutant, `np.linalg.abs`) and are excluded from both counts; see the text above.

What it found. The positive control held: 22 of 29 unbound-name deletions were caught, and every one of the 7 misses sits on a line the ladder did not show executed: 5 on lines coverage saw not run, 2 in files the suite never imports, so nothing was measured there — zero reachable misses (W3). That is what makes the rest interpretable.

2 injections are void, and both are our fault. The table maps `np.linalg.norm` to `np.abs`, and the first operator rewrote only the last token, producing `np.linalg.abs` — which does not exist. Two kooplearn sites got that mutant. One "caught" it with an `AttributeError` crash, not a wrong analysis a test saw; the other was recorded as a miss. Neither is evidence about a gate. We found the defect by reading the recorded output of *catches*, so the miss at the same site shape survived a further review round inside the denominator. An audit that reads only what fired finds only the faults that fire.

The rule that scores them VOID was written after the run and is recorded in the protocol's post-run note; the locked protocol says only that a harness fault is ours when the positive control fails, and it forbids skipping a site for being unfair. So we state the rule and its date rather than claim it was locked. The rule names the site shape and nothing else. It first carried "and the run was caught", which read the outcome before deciding and left the second malformed run in the denominator as a genuine miss; removing that clause voids both and *raises* the rate. The operator now rewrites the whole callee when the partner changes module. That change alone does not make either site well-formed: both calls are `np.linalg.norm(..., axis=0)`, and `np.abs` is a ufunc that takes no `axis`, so the repaired rewrite would raise `TypeError` where the old one raised `AttributeError`. So a call whose keywords the partner rejects is no longer a site at all. Three other rows carry the same hazard — `np.median` rejects `dtype`, `ttest_rel` rejects `equal_var`, `spearmanr` rejects `method` — and no site drawn in this run passed such a keyword, so no count moves. A self-test applies three of the table's ten swaps — `linalg.norm`, `mean` and `ttest_ind` — to a synthetic file, and fails if a mutant does not parse, names a function its module does not define, or is drawn where the partner rejects the call. It runs in the verification chain. The seven rows it does not exercise include the reversed sort that produced every catch here. The raw result files are not edited. Without the voids the count is 5 of 18; with them, 4 of 16. W1 is refuted either way.

What the denominator is made of. A pooled rate hides two facts, so here it is per swap.

swap	caught	scored
`np.sum` → `np.nansum`	0	8
`sorted` → `sorted(reverse=True)`	**4**	6
`np.mean` → `np.median`	0	1
`np.nanmean` → `np.mean`	0	1

Every catch is one swap, a sort reversed. And 8 of the 16 scored injections are another single swap, `np.sum` for `np.nansum`, which computes the same number unless the array holds a `NaN`. As first written, this appendix said we had not checked whether those arrays do, so the 8 might be equivalent mutants. That was the original E96 reading. E116 has since checked it on the gate's own calls (App. K): of the 16 sites, 6 differed on an observed call, 7 never did, and 3 never ran; of the 12 misses the split is 2, 7 and 3. That rests on 2,085 completed comparisons, one per observed call,

with floating-point equality taken as `allclose` at 10^{-12}; a site the gate never ran is not shown equivalent. A denominator of 16 sites describes what our table happened to draw. It is not an estimate.

A wrong analysis was caught 4 of 16 scored injections (2 void), in 2 of the 5 repositories whose *searched* source makes any call in the table. Five of the ten have none, for two different reasons. In 3 we searched the source and it does not call `numpy` or `scipy` statistics by these names. In 2 no source file was searched at all, because no package resolved from their metadata: that record is in the E94 result files, whose discovery this experiment reuses. Those two zeros are our instrument's, not their code's. The locked primary is the repository indicator over all ten: 2 of 10 caught a wrong analysis, exact upper bound 0.51. Read against the repositories whose searched source makes such a call it is 2 of 5, and that denominator is a lower bound, because 2 of the ten were never searched. We give the locked number first and the narrower one beside it. The catches: a reversed sort in a phylogenetic-network representation, caught twice, and each time by a *doctest* and unit tests together, because that project's gate runs `--doctest-modules`; two reversed sorts in voting methods, one caught by an expected-winner assertion and one by `assert mg1 == mg1` — a margin graph no longer equal to itself, because the swap sat inside its `__eq__`.

A silent deleted step was caught 2 of 3. Three is the whole sample: under this rule, an assignment whose name was already bound and is used later is rare — most assignments in these codebases bind a fresh name. The two catches were a degradation term in a flow-cell experiment, and a text label in a data-cleaning routine. For those two the recorded gate output is the list of failing test names, capped at 600 characters by the driver as it then was, so we name the tests and not the assertions inside them; the miss was a gas constant reassigned deep in a 3,300-line registry that no test reaches. We report the ratio and decline to read it as a rate.

Predictions. Six were locked. Two failed.

- **W1**, a wrong analysis is caught more often than E94's wrong constant: **refuted**. 4 of 16 against 8 of 28, so 0.25 against 0.29. The original E96 reading was that a swapped function and a perturbed constant are about equally visible to the projects' own tests. E116 supersedes that reading: the wrong-analysis misses now carry an observed-effect partition (2 differed, 7 did not, 3 never ran) and no witness was run on E94's wrong-constant misses at all (the witness of App. I covers the four targets only), so the two all-mutant denominators do not support a comparison of meaningful-defect detection. The verdict on W1 stands as locked.
- **W2**, a silent deleted step is caught less often than a wrong analysis: **refuted**, on 3 sites, which is not a result.
- **W3**, every unbound-name deletion on a line the suite executes is caught: confirmed. Of the 7 misses, 5 sit on lines coverage saw not run and 2 in files the suite never imports.
- **W4**, at least half of the drawn sites are executed by the suite: confirmed.
- **W5**, under half of the executed wrong-analysis sites are covered by a test that asserts a number: confirmed, 0 of 10 — but the rule could not classify 8 of those 10, so the verdict rests on two sites it read. One of the ten was caught anyway, by a comparison of expected winners that the rule does not count as a number. The other three catches sat outside this set, on sites the prospective record had marked not executed or not measurable; re-measured, all four sit on executed lines.
- **W6**, some repository catches neither a wrong analysis nor a silent deletion: confirmed, 6 of ten. Three of the six drew no wrong-analysis site and five drew no silent-deletion site, so for most of them the verdict says what our operators could reach, not what their gates missed. The void moved this count from five to six: `kooplearn`'s only wrong-analysis catch was the malformed mutant.

What this changes. The two failure classes the catalogue could not express are now in it, and the answer is not "caught by nothing". On the locked all-mutant denominator a wrong analysis was caught about as often as a wrong constant; that is the original E96 reading, and App. K narrows it, because 10 of the 12 wrong-analysis misses never differed on an observed call or never ran. The catches came from tests of more than one kind — an expected-winner assertion, a self-equality test the swap broke, a doctest; a deleted step that raises is caught wherever the suite was shown to run

the line; a deleted step that stays silent is the one we still cannot measure at scale, because the rule finds too few of them. The blinded human baseline remains the comparison this table cannot make.

H FULL RESULTS

	defect class	System B	System A	Pingouin	Ruptures
G1	a number in the prose drifts from its data	$0/6$	$0/6$	$0/6$	$0/6$
G2	a released value moves, the prose does not	$0/5$	$0/5$	$1/6$	—
G3	a released result has its sign flipped	$0/4$	$0/4$	$2/4$	—
G4	a cross-reference points at nothing	$0/4$	$\mathbf{4/4}$	—	—
G5	a test assertion is weakened	$0/4$	$0/4$	$0/4$	$0/4$
G6	a directory of released evidence is gone	$0/2$	$\mathbf{3/3}$	$\mathbf{1/1}$	—
G7	a wrong constant in the analysis code	$0/4$	$0/4$	$0/4$	$0/4$
	repository-surface rate	0.000	0.233	0.160	0.000
	common-operator rate (G1, G5, G7 only)	0.000	0.000	0.000	0.000
	false alarms	$0/5$	$0/5$	$0/5$	$0/5$

Table 20: Seven portable operators, 4 repositories. "—" is an empty site population, not a miss. The common-operator set is G1, G5, G7 — the operators every target has; G2, G3, G4, G6 are excluded because at least one target has nothing for them to mutate. A fifth target, lifelines, was withdrawn (§6).

comparison	repository-surface	common-operator
System B − System A	-0.233 $[-0.409, -0.069]$ **sep.**	$+0.000$ $[-0.215, +0.215]$
System B − Pingouin	-0.160 $[-0.346, -0.009]$ **sep.**	$+0.000$ $[-0.215, +0.215]$
System B − Ruptures	$+0.000$ $[-0.215, +0.117]$	$+0.000$ $[-0.215, +0.215]$
System A − Pingouin	$+0.073$ $[-0.146, +0.274]$	$+0.000$ $[-0.215, +0.215]$
System A − Ruptures	$+0.233$ $[-0.011, +0.409]$	$+0.000$ $[-0.215, +0.215]$
Pingouin − Ruptures	$+0.160$ $[-0.076, +0.346]$	$+0.000$ $[-0.215, +0.215]$

Table 21: Newcombe 95% intervals on the *difference*. 2 of 6 surface comparisons clear zero, both of them System B against a target that has verification machinery. No common-operator comparison clears zero: on the operators every target shares, every target catches nothing. These are all six pairwise contrasts on four repositories, computed after the fact and not corrected for multiplicity, so they describe this finite set and are not a test.

	caught	95% upper bound	
each repository, on its own	$0/14$	0.215	
pooled, assuming every draw independent	$0/56$	0.064	(optimistic)
repository as the unit, 4 of them	$0/4$	0.490	(pessimistic)

target	operators scored	the ones with nothing to mutate
System B	7 of 7	—
System A	7 of 7	—
Pingouin	6 of 7	no LaTeX (cross-references)
Ruptures	3 of 7	no data files, no LaTeX, no data directory

Table 22: How many of the seven portable operators each target could be scored on. The three classes missed *everywhere* are exactly the three that apply to every target, which is why they carry the paper's main claim.

What the catch-rate denominator contains (E105, post hoc). A single rate over a pooled total does not say how many injections a gate could have seen at all. So we split every injection into the **external** targets into classes — 165 of them across 13 repositories — and we assigned the class from facts recorded *beside* the outcome — did the mutation apply, does the

defect class	caught
a retired value quoted as current	$4/4$
a cross-reference pointing at nothing	$4/4$
a truncating parser, a dropped bias correction, a drifting script default, the main text over the page limit	$1/1$ each
a released value moves, the prose does not	$2/6$
a number in the prose drifts from its data	$\mathbf{0/6}$
a released result has its sign flipped	$\mathbf{0/4}$
a locked verdict relabelled in the generated ledger	$\mathbf{0/4}$
a directory of released evidence is gone	$\mathbf{0/3}$
a test assertion is weakened	$\mathbf{0/4}$
overall (locked catalogue)	$\mathbf{14/39 = 0.359}$
false alarms	$0/5$

Table 23: System A against its own eleven-step verifier: perfect on everything a guard was written for, zero on five classes. Ten of the fourteen catches come from one step, the cross-file consistency checker; seven of the eleven steps stopped nothing at all. The per-class rows are what we defend.

gate open that file, did coverage see the line run — with the classifier never given the outcome. Then we cross-tabulated. System A's own 43 injections are *not* in this table and cannot be: their records predate the reachability fields, so no outcome-blind class can be assigned to them without re-running them. Their denominators are the ones in the claim ledger (App. C). The analysis is `experiments/105_outcome_taxonomy/analyze_taxonomy.py` and `data/e105_outcome_taxonomy.json`. It is post hoc: no predictions, and every locked rate in this paper keeps its own denominator. This table is reported beside them, not instead of them.

class, decided without looking at the outcome	n	caught	rate
the mutation did not apply	0	0	—
the gate never opens that file	47	0	0.00
the gate opens it; no coverage, so activity unknown	29	13	0.45
the gate opens it; coverage says the line did not run	33	6	0.18
the gate opens it; coverage says the line ran	56	20	**0.36**
pooled	165	39	0.24

Read it as two different findings that the pooled 0.24 was hiding. Where no gate opens the file, the catch rate is 0.00 over 47 injections — that is a scope gap, not a checker failure, and it is the honest reading of the operator's name. Where the gate opens the file and coverage saw the line run, it is 0.36 over 56. That is an exposure stratum, not a catchability bound: a line that runs can leave every value and every predicate unchanged, which is what Apps. I and J then measure. The table is the raw record, with the 2 later-voided draws still in it, and its rows are not combined with those corrected labels. It is a historical measurement, not corrected exposure: its line attribution carries the continuation-line errors described next, and for both reasons it is no longer quoted in the main text.

Two things this table would have hidden if we had let it. First, 0 catches landed in a class we called unexposed; had any landed there, the class label would be wrong, not the catch. Second, 6 catches landed on lines coverage called *not executed*, and all 6 of them are bracketed continuation lines — an argument inside a multi-line signature, an element of a multi-line list, a closing parenthesis on its own line. `coverage.py` records the first line of a multi-line statement and no other, so such a line is never marked executed even when the statement runs every time. The catches are genuine; the label is an artifact of line-level attribution. That makes the fourth row a *conservative* floor on exposure rather than a clean negative, and we say so rather than moving the sites into a friendlier class. The sixth class the review asked for, *self-restored*, has no instance in this external frame, because no gate here regenerates a file we mutate; where it does occur is our own repository under operator D6, and it is counted below, not here.

The repair, tested on deletions it was not built on (E107). Finding a gap and writing a fix is not the same as fixing it. The manifest was developed on 3 deletions, and until now the only evidence for it was those same 3. So we drew 8 more directories by seed from the manifest, excluding the 3 it was built on, and ran the project's real verifier on each one in two arms: with the manifest, and with the manifest present but naming no directory, which makes the check inert without editing the gate under test. The protocol is `experiments/107_repair_holdout/protocol.md`, locked with no data in the tree.

held-out deletions caught, with the repair	$8/8$
sites where the *un-repaired* gate passed	5
of those, caught by the manifest check itself	$\mathbf{5/5}$
declared benign edits that tripped the gate	$0/6$
median trial runtime against a clean run	57 s vs 68 s

The middle row is the one that answers the question. At 5 of the 8 sites the un-repaired verifier **passed** — the same silence that hid the original three — and the manifest caught every one of them, at step 11, with its own message. The other three sites are caught in both arms, because an earlier step reads those cells, so the repair gets no credit for them; the paired definition was locked in advance for exactly that reason, because crediting them would have made the repair look better than it is.

The benign edits are declared in the protocol by name: a new file in a shipped directory, a comment appended to a source module, an mtime update, a new empty subdirectory, a blank line in the README, and — the one that matters — retiring a directory the documented way, by deleting it *and* removing its line from the manifest. None tripped the gate. That last case is the operation the repair's own error message tells a maintainer to perform, and a repair that failed there would be unusable. 0 of 6 is not a low false-alarm rate: the exact one-sided 95% upper bound is 0.39, and six edits chosen by us are not a sample of what maintainers do. The runtime is the easy part — the check is a set difference over the manifest's directory names, and the median trial ran in 57 s against a 68 s clean run.

The second repair, and the same test (E110). The review named two repairs worth evaluating and gave the recipe once: develop on known failures, freeze, draw held-out mutations, include benign edits that should pass, measure detection gain and runtime, and keep the added gate separate from the original workflow. The first repair is above. The second is for numerical claims: *validate that the expected occurrences are still found, as well as validating their values*.

The gap it fills is one this project's own correction record contains. System A's consistency checker pins a value, finds every mention with a pattern, and compares each mention's number to the pinned value. It already fails when a pinned quantity is mentioned *nowhere*. But a value that drifts *out of its own pattern* while a second occurrence still matches leaves the count above zero, and the drifted text is no longer looked at, so nothing fires. The repair is a committed baseline of mention counts, failing when a count falls *below* it. A count that rises is a new sentence; a count that falls is the drift signature. The asymmetry was declared in `experiments/110_mention_repair/protocol.md` before anything was tested, because a gate that flags every added paragraph gets switched off.

held-out drifts caught, with the repair	$\mathbf{9/10}$
the same drifts caught *without* it	$0/10$
detection gain	$\mathbf{9}$
declared benign edits that tripped the gate	$0/5$
added runtime	$+0.1$ s

The ten pins were drawn by seed from those with at least two mentions, so removing one occurrence could not simply trip the existing mentioned-nowhere check and measure that instead. The benign edits are named in the protocol: emphasise a pinned number in place, add a *second* mention of one, reword the sentence around one, move a mention from one appendix file to another, and add a paragraph with no numbers in it. None tripped the gate — including the two that exist to test the rule's asymmetry, the one that raises a count and the one that moves a mention without changing the total. 0 of 5 is not a low false-alarm rate: with zero of five the exact one-sided 95% upper bound is 0.45.

operator	gate never opens the file	activity unknown	opened, line not executed	**opened and line executed**	all
G1	0/47	—	—	—	0/47
G5	—	0/8	0/1	0/19	0/28
G7	—	3/9	5/21	2/10	10/40
G8	—	2/2	1/6	2/10	5/18
G9	—	8/10	0/5	16/17	24/32

Table 24: Every injection, per operator, as caught/scored inside each outcome-blind class. The bold column is the one that carries a claim about checking: the gate opened the file *and* coverage saw the mutated line run. **These are the raw counts**, before the two malformed mutants were voided, because the classification is of the record as it was run. So G8 reads 5/18 here and 4/16 in App. G, where the voided pair is removed. Both are correct; they count different things, and nothing else in the table is affected.

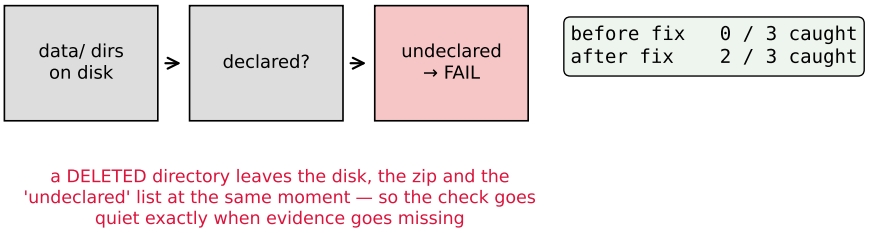

Figure 3: A guard that could not fail. It refuses a directory that *exists* but was never declared, and cannot see one that has vanished. Nobody would find this by reading the code; it took deleting a directory and watching the suite stay green.

The one miss is ours, and we report it as a miss. At one site the pin's pattern is `0\.01\b` and our runner replaced the first raw substring `0.01` in the file, which sits inside `-0.017` — text the pin never matched. Nothing the checker looks at changed, so nothing fired, and this says nothing about the repair. A reader who prefers to score that site as *not implemented* gets 9 of 9. We keep 9 of 10 because we noticed only by looking at the miss, and the prediction holds either way. The runner ships exactly as it ran, so the artifact regenerates from the released code; locating the occurrence with the pin's own pattern belongs in a follow-up, not in a quiet edit to a script whose output is already released.

Per operator: how many were exposed, how many were active, how many were scored. A reviewer asked exactly this, and asked what changes when unknown activity is kept apart from a meaningful miss. Table 24 is the answer, generated from the same artifact.

Three things change, and they change in different directions. **G1** — a prose number drifting from its data — is *entirely* scope: all 47 of its injections are in files no gate opens, so its zero says the gate never looks, not that it looks and fails. **G5** — an assertion weakened to `assert True` — is the opposite: 0 of 19 on lines the gate opened *and* coverage saw run. That is a zero on demonstrably exercised code. Execution alone does not make it a hole: a weakened assertion runs and passes by construction, and App. J found that for the 7 assertions it could pair on the four targets, the gate still failed without them; no JOSS assertion was paired. Separating the classes says where the zero sits, not what it costs. **G9** — a deleted step that leaves a name unbound — is caught 16 of 17 in the same column, which is what a working test suite looks like. So the split does not flatter us in one direction: it weakens the prose result, places the assertion zero on executed lines, and leaves the constant result where it was.

Uncertainty on the three universally-missed classes. Draws inside one repository share one gate, so pooling them as independent flatters the bound. Four separate replications is the honest description.

The same blind spot, one level up. The manifest that fixed the deleted-directory gap was itself not in the distributed archive. An outside reviewer extracted the release and found it missing: the check printed a note, skipped itself, and the release still reported success — the identical failure shape, now protecting the fix. The manifest is shipped and required today, and a negative packaging test deletes it on purpose so the gap cannot reopen quietly.

The exploratory operator, and why the headline rate excludes it. Locked operator D6 flips a REFUTED verdict in the generated prediction ledger. That file is rebuilt from the per-experiment result files at step 2 of the verifier, so the flip is overwritten before any later gate can read it: the operator measures something other than what it was written to measure. Amendment 1, committed before any injection ran, added **D6b**, the same flip applied to the result files the ledger is built from — the place a verdict would actually have to be laundered. D6b is exploratory and takes no pre-registration credit, so the overall rate is the locked 39. It is the best-caught operator in the experiment, 3 of 4, which is why we give both numbers: excluding it moves the headline from 17/43 down to 14/39, in the direction our thesis prefers.

The same reasoning cuts the other way and we should follow it there too. If D6's flip never reaches a gate, its 1×4 runs are not a test of the gate either, and the honest denominator over the classes the verifier could actually see is 14 of 35 rather than 14 of 39. That is a *higher* catch rate, so it works against this paper's thesis, which is why it belongs here rather than in a footnote. We report the locked rate first because it is what the protocol locked, and this one beside it.

Positive controls on the reconstructed gate. System B's gate ships with the project only in the sense that its tests and its paper build exist; we assembled the gate from them, so it needs evidence that it can fail at all. A test asserting `False` stops it; a syntax error in a source module stops it; an undefined LaTeX command inside the document stops it. A dangling `\ref` does not — which is the finding, not the control.

A changed byte is not automatically a changed result. A mutation can be real and mean nothing: a redundant assertion, a constant on a path that never runs. We classify the perturbed constants and report the substantive ones separately (App. C), but *after* seeing whether the gate fired — weaker than Shulepov (2026), who dropped one before any outcome was known.

Clean replicates bound the wrong side, and less tightly than they look. 20 clean runs with no failure do not measure a low false-alarm rate: with zero of 20, the exact one-sided 95% upper bound is 0.14, and those runs re-execute the *same unchanged tree*, so they sample no benign edit at all. That 0.14 pools every repository's clean runs. System A alone contributes five, and five of five with no failure bounds it only at $\mathbf{0.45}$ — a reviewer read "five clean re-runs" in the results beside the pooled bound in the discussion and computed that number, correctly, so both are printed here. Neither number is a false-alarm rate over benign edits, because we made no benign edits. What they bound is flakiness, not the assumption a measured zero needs — that a gate which *can* catch a defect catches it every time. Nuijten & Wicherts (2024) is what a real evaluation of a checker looks like: preregistered, matched controls, thousands of articles. We are not at that standard.

Our own checks have gaps, and we list them. We caught one operator measuring documentation instead of code only by looking at the sites it drew, which Panichella & Liem (2021) predicted. Errors kept moving to whatever surface had no check, so we enumerated them: 20 of 22 are gated, 16 with a test that deliberately breaks them, and 100 prose numbers are right today with nothing checking them (App. H).

What a weakened assertion can and cannot show. The operator replaces `assert X` with `assert True`. The test still runs and still passes, so an ordinary test run has no reason to fail, and a zero catch rate here is close to expected rather than surprising. We had named the operator "deleted assertion", which oversold it, and it is now named for what it does. What the zero measures is whether the surrounding verification process notices that a test lost its strength — not whether a test fails. Nothing in these suites looks for that, and nothing we injected would have told them, so the honest reading is narrow: no gate here monitors test strength, and we did not establish that any weakened assertion was protecting a live defect. Establishing that needs a production defect the

original assertion catches and the weakened one lets through, which is the next experiment rather than a claim we make now.

Coverage. A decimal literal in the paper source counts as covered if some check's pattern matches at that position. We used to call that the condition under which a drift is caught: either the comparison fails, or the pattern stops matching and the check reports itself stale. The second half of that sentence is wrong, and the next paragraph is the experiment that showed it. On System A that was only 473 of $2{,}083$ occurrences (22.7%) across the whole document, and 47.8% in the main text. We reached that figure after two wrong ones — string-matching canonical values gave 31%, and counting any literal within tolerance of any pinned value gave 99.4% because at 151 pinned values the tolerance bands tile the interval. Three criteria, three answers between 22.7 and 99.4; only the first describes what would actually be caught.

Coverage is not catchability, and we measured the gap. A reviewer can object that the prose-number zero says where our uniform draw landed, not what the checker can see. So we drew again on purpose: 6 literals from the positions a check's pattern matches and 6 from the rest, predictions locked before any run, the gate this repository's own unmodified verifier (E100). The covered stratum was caught 1 of 6 times and the uncovered stratum 0 of 6. That is the predicted direction and it is one catch: Fisher's exact test on this table gives $P{=}1.0$, so the difference is not separable from chance. The objection is answered in a different way than we expected.

Replaying the six covered sites against the checker explains it. A check requires at least one match of its pattern and that every match equal the canonical value. It does not pin how many matches there are. A drifted literal stops matching, so if its check still matches anywhere else, the check passes on what it can still see: the five misses moved their check's match count from 28 to 27, 22 to 21, 6 to 5, 4 to 3 and 3 to 2, and the one catch moved it from 1 to 0, which emptied the check and made it report itself stale. So a drift is caught only when its check has exactly one match in the whole paper. That is 29 of our 129 checks, covering 25 of the 1944 literals in today's source — not the 429 that the coverage criterion counts. Coverage overstates catchability by more than a factor of ten, and the uniform zero is exactly what a 1% catchable fraction predicts.

The fix is one line of discipline this repository already applies to facts quoted from other papers and never applied to its own: pin the *number* of mentions, not only their values. That gate works — adding a second mention of a cited count to this paper failed the build until the expected count was updated — and applied to our own canon checks it would have caught five of the six.

Which surfaces are gated, and which are not. Errors in this project kept moving to whatever surface had no check: a false claim, then checks that compared nothing, then a table reference printed as two question marks that shipped because LaTeX only warns about an undefined reference, then a derived record that nothing rebuilt. Each round closed one surface and the next round found the next one, so we stopped waiting and enumerated them.

`paper2/surface_audit.py` lists 22 places a claim can live in this paper: table cells, table captions with counts, generated macro values, figure data (bars, points), figure axis and legend labels, figure captions, prose numbers with a canon check, prose numbers via generated macros, other prose numbers, the canonical record itself, raw result artifacts, derived artifacts vs their generators, released run logs, cross-references and citations, page limit, LaTeX build errors and overfull boxes, supplement contents, anonymity, target descriptions, protocol files, numbers quoted from cited papers, and the checkers themselves. For each it records what gates it and whether a negative test proves that gate fires. 20 are gated; 16 have a negative test.

2 are not gated. Prose numbers with no pinned check: of 580 numeric literals in the prose, 33 sit inside a check or a generated macro and 18 are *parameters* rather than results — seeds, the perturbation factor, the confidence level, locked thresholds, which are fixed in the protocol and have no measurement to drift from. That leaves 100 that match a value in the released artifacts and are genuinely unchecked. And the protocol files, which are prose we do not check. The audit runs as a verifier step, so this list prints on every run.

An earlier version of this paragraph reported those counts as typed numbers, and they went stale within a round — in the paragraph about numbers going stale. They are macros now. The same pass

found the paper claiming 2,900 lines of checking machinery when the measured figure had grown to 6,592.

One more of these survived until a reviewer's round: §4 used to say System A implements "fourteen distinct kinds of refusal". Nothing could check it — there was no list, no script, and no definition of "kind". Counting mechanically, as one place where a checker appends a problem that makes it exit non-zero, gives 54. The typed number was either stale or measuring something we can no longer reconstruct, and we could not tell which, which is the whole argument for not typing it. The definition and the regex are in `paper2/derive_prose_facts.py`, so a reader can disagree with the definition and recount.

Are the misses stable? We re-ran every catch and required it to reproduce, but never re-ran a miss — and the misses are the half this paper's claim rests on. A miss that would have been caught on a second run makes every zero optimistic. Ruptures is the right target to check on: its gate takes about 8 seconds and fails 0 of 30 clean runs, so repetition measures mutation-side variability and nothing else. Every one of its 14 injections was run 3 times. 0 of 14 gave a different verdict across 42 runs. On a target whose gate is deterministic, a miss is a property of the gate and not of the run. This says nothing about a target whose gate is stochastic, which is why lifelines was withdrawn rather than repeated.

We release the log of every one of those 42 runs — one file per run, each with its own SHA-256, each stripped of machine paths. A reader does not have to take "missed" on trust. Open a log and look for a failure line: our harness writes `GATE FAILED at step` n whenever a gate command exits non-zero, so a log without that line is a gate that ran everything and passed, which is a miss. There is no such line in any of the 42 logs, and the checker does not ask you to believe that either: on every run it re-hashes all 42 files and reads the verdict back out of the bytes. Each file holds the last 40,000 characters of the gate's output, which is the end, where the verdict is.

The previous version of this paragraph said the release carried a hash of each run's log. Neither half was true. The script kept one log per injection, so each repeat silently overwrote the one before it — 14 files cannot be a hash of 42 runs — and the logs were not shipped at all, so the hash certified a file nobody could open. Nothing checked either half, which is why it survived. That is the same failure this paper is about, in the paper's own release: a witness that was never a gate.

How many repositories would it take? A reviewer asked the question a future study needs answered, so here it is. Treating the repository as the unit — the pessimistic bound, and the one that has to shrink — and assuming every repository catches none of these classes:

repositories	4	8	12	16	20	30	50
95% upper bound	0.53	0.31	0.22	0.17	0.14	0.10	0.06

Ours is the first column. Every entry is the exact one-sided Clopper–Pearson bound, the same family the frame-sampled primary uses; the Wilson two-sided bound quoted elsewhere for the same four repositories is 0.49 against this 0.53. A bound a practitioner could act on — "fewer than one project in ten catches this" — needs on the order of thirty repositories, every one of them catching nothing. The part worth saying out loud is that **adding injections inside the four we have does not move this number at all**. The repository is the unit, so more mutants per repository buys precision on the wrong axis. That is why we describe this study as a pilot and why the next one should spend its budget on targets rather than on operators.

Our seven operators are not one kind of defect. A reviewer asked what makes each mutant non-equivalent. The honest answer is that it depends on the operator, so we say which is which instead of one blanket rule.

kind	operators	what a missed one costs
a wrong number reaches a reader	G1, G2, G3	the paper states a value its own data does not support. The disagreement between prose and artifact is by construction. Whether the number is a claim a reader relies on is not: on the JOSS repositories 32 of 38 drawn literals were identifiers or versions (App. F).
	G7	only when the constant is on a live path. A hand label applied after the outcomes called 10 of 16 substantive; that count is historical, and one of the ten is an overwritten placeholder (E90 below). The measured witness (App. I) could build a probe at 7 of the 16, saw a changed value at 0 under the locked rule and 3 post hoc, and only 1 of the no-witness probes saw a tenfold change.
the document or release breaks	G4, G6	a reference resolves to nothing, or the evidence a claim rests on is not in the archive. Nothing a reader *reads* is wrong; what breaks is the ability to check it.
a guard stops guarding	G5	no number changes at all. A weakened assertion can only cost future protection, and App. J did not observe that cost: for the 7 assertions it could pair with one perturbation, the gate still failed without them; the other 9 are unresolved.

Two consequences we should state rather than leave implicit. First, the overall catch rate mixes all three kinds, which is a second reason not to lead with $14/39$ — the first is catalogue weighting (App. W). Second, none of the three classes missed by all four repositories is shown result-changing by this table. G1 changes a prose byte by construction; whether a reader relies on that number is a site-level fact. G7 changes a result only where a witness sees it, and the locked witness saw a change at none of the 7 it could probe, only 1 of which could see a tenfold change (App. I). G5 weakens a guard, and the paired test found no lost protection in the 7 cases it could decide (App. J). The strongest form of our claim is about what the gates looked at, not about proven consequences.

The withdrawn fifth target, in full. We tried a fifth repository — lifelines, a survival-analysis library with 537 passing tests — and had to withdraw it. The reason is worth more than the datapoint.

Its sweep returned five catches of twenty-five. One made no sense: the mutation edited a file under `docs/`, and the gate is `pytest lifelines/tests/`, which never reads `docs/`. A whole-tree diff confirmed exactly one file had changed. So we tested it instead of reporting it, alternating mutated and pristine trees back to back. The mutated tree failed 0 of 3 times. **The pristine tree failed** 1 **of** 3, on a different test. The cause is the project's own `conftest.py`: it reseeds the random number generator randomly before every test, and several tests then draw tens of thousands of samples and assert a fitted coefficient lands within a fixed tolerance. The suite fails sometimes on an untouched checkout.

The precondition. Scoring by exit status assumes the gate is **deterministic**. We had not written that down, and it is false for one of the four public repositories we tried. lifelines is reported as *unmeasurable by this method*; its five catches are withdrawn.

Excluding it is a protocol choice with a price, not a law. Dropping a target because its gate is stochastic selects for unusually tidy repositories, and plenty of real research pipelines are not tidy. A stochastic gate can still be scored — just not by exit status. The estimand becomes

$$\text{excess} \;=\; P(\text{gate fails} \mid \text{mutant}) \;-\; P(\text{gate fails} \mid \text{clean}),$$

measured by running both arms many times. Our lifelines data can already be read that way, and the reading is that we have almost nothing: the controlled alternation gives a point estimate of $\mathbf{-0.333}$ — the *mutated* arm failed *less* often than the pristine one — with a Newcombe interval of $[-0.79, 0.50]$, nearly the whole admissible range. Three pairs cannot measure this.

What would? At lifelines' measured clean failure rate of 0.0625 (1 of 16 clean runs), detecting an excess of 0.20 at 95% and 80% power needs about **62** runs per arm; an excess of 0.10 needs 175. (Our first version of this calculation left out the continuity correction and said 53 and 156 — optimistic by about a sixth, which is the wrong direction to be wrong in when the number's only job is to say how expensive the experiment would be.) Its suite takes minutes, so one injection is hours and a catalogue is weeks. That is the real reason this paper stops here, and it is a cost, not a principle. We write the number down so the exclusion reads as a scoped choice rather than a claim that stochastic gates cannot be studied (`data/e77_independent/e84_stochastic_gates.json`). We also looked for a cheaper one. E94 (App. F) screened 120 public repositories, of which 46 got as far as running their own suite three times; none of those passed once and then genuinely failed. The one candidate that looked stochastic was a slow gate whose second run the clock killed. So the inexpensive demonstration a reviewer asked for still has no target, and we say so rather than manufacture one.

What this does to the main result. An earlier draft said a stochastic gate can only *add* spurious catches, so a measured zero is safe. That was wrong. Flakiness runs both ways: the one test that would have caught the defect can pass by luck, and we score a miss.

So a zero is an upper bound, not a proof. And the right way to state the bound is not one big sample. Draws inside one repository share one gate, so pooling all 56 as independent flatters us. What we actually have is **4 separate gates**, each of which caught none of these three defect classes. Two of the 4 are ours and share an author with the catalogue, so they are not independent of us or of each other; Pingouin and Ruptures are not ours. We say "separate gates", not "independent samples", because we have no sampling model and the four were not drawn from anything.

Each repository is $0/14$ on those three operators, an upper bound of 21.5% on its own. Pooling all 56 draws as independent would say 6.4%; treating each repository as one observation would say 49%. We give both rather than the flattering one (App. H). Neither is a statement about a fifth repository: with four cases and no sampling model, these bounds describe the four we ran. Ruptures also fails 0 of 30 clean runs, so its zeros are not hidden by noise in the other direction.

Two changes follow for anyone using the harness. Make a target pass its gate *repeatedly* before injecting. Re-run every catch, and count it only if it reproduces.

We re-ran the misses too, since those are the half this paper's claim rests on. Every Ruptures injection was run 3 times: 0 of 14 gave a different verdict across 42 runs. On a target whose gate is deterministic, a miss is a property of the gate and not of the run (App. H).

This is also an instance of our own thesis. Our measuring instrument produced a number that no mechanism could explain, and every automated check we had was green. We caught it by asking how a docs file could break a test that never reads one.

Can the guard even see the file? (E87) Our central sentence is that a gate catches what somebody pointed it at. A reviewer pointed out that this is unfalsifiable if "pointed at" is decided after seeing which guard fired: it would then explain every outcome and forbid none. Fair. So we fixed a rule first, from two things that contain no outcome — the gate commands and the operator site populations — and tested the observed catches against it.

The rule. A (target, operator) cell is *file-level reachable* if some gate command reads a file the operator can mutate. The name matters, and a reviewer was right to insist on it: reading a file is the weakest of three things one might mean. *Syntactic*: the command opens or imports the file. *Dynamic*: the mutated line is actually executed. *Assertion*: executing it could change a predicate that decides pass or fail. This rule measures the first. The next paragraph measures the second on the one target we can rebuild; nothing here measures the third. Commands are classified by what the tool reads by definition: `pytest` reads the test files and the package it imports; `pdflatex` reads the LaTeX sources; a project's own verifier script is a suite and reads everything it declares; a one-liner grepping a build log reads none of the site populations.

gate reads →	G1 prose	G2 data	G3 data	G4 refs	G5 test	G6 dir	G7 const
System A (`verify.sh`)	yes	yes	yes	yes	yes	yes	yes
System B (pytest + LaTeX)	yes	no	no	yes	yes	no	yes
Pingouin (pytest)	**no**	yes	yes	no	yes	yes	yes
Ruptures (pytest)	**no**	—	—	no	yes	—	yes

This splits the headline into two different findings, and only one of them is surprising. On the two public libraries the gate is `pytest` and nothing else, so it never opens the files G1 mutates. "Nothing caught the prose drift" there is a statement about what the suite is aimed at, not about a guard failing. G5 and G7 are the opposite: `pytest` reads the test files G5 edits and imports the source files G7 edits, so on *every* target the gate could see those defects. Counting only cells the gate can actually read, 44 **injections were visible to the machinery and** 0 **were caught** — 32 from G5 and G7 across all four targets, plus 12 G1 injections on the two targets whose gate does build the paper. The remaining 12 G1 injections are on gates that never look, and we no longer count them toward the claim.

Does the suite actually run the line? (E89) Ruptures rebuilt from its recorded URL and commit, its own gate — `pytest tests/` — run on the *clean* tree under `coverage.py`. No injection is involved, so no outcome can leak into the measurement. Then: is each drawn site on a line the suite executed?

operator	site	line executed?	outcome
G5	`tests/test_costs.py`:196	yes	missed
G5	`tests/test_costs.py`:239	yes	missed
G5	`tests/test_datasets.py`:78	yes	missed
G5	`tests/test_l1potts.py`:488	yes	missed
G7	`costs/costrbf.py`:34 (kernel bandwidth)	yes	missed
G7	`datasets/pw_normal.py`:29 (correlation)	yes	missed
G7	`show/display.py`:87 (plot span)	**no**	missed
G7	`show/display.py`:87 (plot span)	**no**	missed

The positive control passes: all four weakened assertions are on executed lines, so the coverage measurement is doing its job. Our locked prediction that at least three of the four constants would be executed was **refuted**: two of four are.

Why those two sites are never run is the interesting part. They are in `show/display.py`, and the suite prints `652 passed, 4 skipped`. The four skips are the plotting tests, skipped for a missing optional dependency — and the project's own workflow runs `pip install .[test]`, which does not bring that dependency either. So this is not our rebuild being thin: **the target's own CI never executes its plotting code**, and everybody reads the 652. A silent skip is a third kind of blind spot, next to the guard nobody wrote and the rule that cannot fail.

What survives. Six sites executed, zero caught — on this target the suite *runs* the line and still does not notice a 10% change to the number on it. And this bounds what the 44 is worth: it is a file-level count, half the constant sites here were never run, and we do not restate it as if it were dynamic. Layer three is untouched by all of this.

How much of the corpus does the gate actually run? (E91) E89 traced eight sites on one target. A reviewer asked the obvious next question — what about the rest? — so we rebuilt every target we can and ran each one's own gate on its clean tree under coverage. No injection is involved anywhere in this, so nothing measured here can depend on an outcome.

target	gate	sites traced	on executed lines
Ruptures	`pytest tests/`	7	6
Pingouin	`pytest tests/`	8	**8**
System A	`./verify.sh` (its Python steps)	8	5
System B	private, not rebuildable	—	not measured

19/23 **sites are on lines the gate demonstrably runs, and** 0 **of the** 19 **were caught.** That is a smaller number than the 44 file-reachable injections and a much harder one to argue with: these are

not defects the machinery never looked at. All twelve weakened-assertion sites execute, which is the positive control and it passes on every target. The four that do not execute are Ruptures' plotting line (its tests skip, above) and three of System A's four constants — live-collection code its unit tests do not exercise. System A's figure is a *lower* bound: its gate has shell steps we cannot line-trace, so a site those steps touch would be missed here.

Two things this took two attempts to get right, both ours. Pingouin's four test sites first came back unexecuted; they had never been *measured*, because Pingouin's own `pyproject.toml` tells coverage to omit `*/tests/*` and we had inherited that config. And our first pass counted an unmeasured site as an unexecuted one, which turned a gap in the instrument into a refutation. Rates here are over measured sites, and unmeasured ones are named rather than folded in.

Does the change reach a number anyone computes? (E90) Layer three, on the two executed sites, without ever consulting the gate: apply the harness's own perturbation to a copy of the source, call one deterministic function that reaches the constant with a fixed seed, and compare the value returned.

site	probe	changes the value?
`datasets/pw_normal.py`:29 (correlation)	generate a signal, fixed seed	**yes**, $318.14 \rightarrow 317.29$
`costs/costrbf.py`:34 ("bandwidth")	fit the cost, read the error	**no**

The second row is a correction to our own catalogue. `self.gamma = 1.0` is a placeholder that the next four lines overwrite — `self.gamma = 1 / K_median` runs whenever the median squared distance is non-zero, which it is for any real signal. Perturbing it changes nothing. **It is an equivalent mutant, and our hand labelling — done after the gate outcome was known — called it substantive.** That is exactly the failure mode an outcome-blind eligibility rule prevents, arriving in our own data, and it is why we now report the substantive count as a labelling we did late rather than as a property of the catalogue.

And can the change reach an assertion? (E93) The third layer, on the one site that survives the first two — read by the gate, executed by the suite, and observed to change a number on the gate's own calls. Coverage with per-test contexts names the tests that execute it, mechanically. Then we read what those tests assert, and quote every line the reading rests on.

test that executes the line	what it asserts	class
`test_empty_arg`	`func()` — nothing follows	none
`test_normal`	`assert signal.shape == (n_samples, 2)`	shape
`test_seed`	`assert np.allclose(signal1, signal2)`	**self**

Three tests run the line and not one of them looks at a value it affects. The third is the sharpest: it calls the generator twice with the same seed and asserts the two results match *each other*. Change the correlation and both sides change identically, so that assertion cannot fail for any value of the constant. It is a determinism test, and it is doing its job — it is simply not a test of the number.

So the miss has a mechanism, not just a zero. The gate reads the file, runs the line, the value changes, and nothing asserts on it. That is a better answer than "nothing caught it", and it is four lines of the target's own source that any reader can check. The positive control passes: all four weakened-assertion sites classify as reachable, because the deleted line *is* the predicate — a method that called those unreachable would be broken.

One site, one target, and the second step was our reading rather than a mechanical rule. E95 turns that reading into a rule and runs it over everything.

The same question, asked of every executed site (E95). A reviewer asked it plainly: of the executed sites, how many can actually reach a pass/fail predicate? E93 answered for one. E95 answers for all 19, and it has to ask two different questions because the corpus holds two kinds of site.

A wrong constant (7 sites). The change moves a number. It can reach a predicate only if some test that runs the line asserts on a number the change can move. We take the tests coverage names, parse

each one, and classify its assertions with E93's vocabulary. Result: **1 of 7**, and even that is an *upper* bound — our rule counts a site whenever *any* covering test asserts *any* number, which need not be a number this site can move. 2 sites our rule could not classify, and we report them as their own bucket rather than putting them where they would suit us.

The sharpest way to say it is per assertion. Those covering tests contain 41 assertions between them. **1 of 41 asserts a number at all.** 20 assert a shape, a length, a type, or that an exception was raised. 4 compare two things the target itself computed, so a wrong constant moves both sides and the assertion cannot fail for any value of it. Those tests are not bad tests — a determinism check is doing its job. They are simply not tests of the number.

A weakened assertion (12 sites). Here the change *is* the assertion, so running the line proves nothing — a weakened test runs and passes. It can only reach a predicate if some gate step measures how strongly a test constrains: a coverage threshold, a mutation tool, an assertion count. We grepped each target's own gate, CI workflow and config for those, and found such a step in **0 of 3 targets.** So every weakened assertion in this corpus is unreachable at this layer *by construction* — not because we could not find a path, but because nothing in these gates is a function of test strength.

Two senses of "reachable", and why they come apart. E93 called a weakened assertion reachable; E95 calls it unreachable. Both are right and they mean different things, so we name them:

- **it sits on a predicate** — the changed line *is* a pass/fail check. An assertion weakened to `assert True` always satisfies this. That was E93's sense, and it is the right one for a positive control: a method that called a weakened assertion unreachable would be broken.
- **it can flip a predicate** — the change can turn some check from pass to fail. A weakened assertion cannot. That is E95's sense.

For a weakened assertion the two come apart, and the gap between them *is* the reason the gate passes. The defect sits exactly on a check and still cannot make one fail. Our headline uses the stricter sense, because it is the only one that bears on whether a gate could have caught anything.

We used E93 as a control on the rule, and the rule is coarser. E93 read one site's three covering tests by hand. The automatic rule must at least agree on the decision, and it does (yes): no covering test asserts a value that site can move. On the finer labels it disagrees. It takes the strongest assertion anywhere in a covering test, and it cannot tell which assertion relates to the mutated value, so where the human wrote *self* it writes *unclassified*. The disagreement does not move the decision — neither label is *value* — but it does show the automatic rule is the weaker instrument of the two.

And it refuted one of our own predictions, for that exact reason. We had predicted (Q3) that at least one covering test would classify *self*. None did. The reason is not that the pattern is absent — counted per assertion it occurs 4 times, and E93 quoted one of them. The reason is that our rule labels a test by its *strongest* assertion, and E93's `assert np.allclose(signal1, signal2)` sits in a function that also contains `assert bkps1 == bkps2`, which the rule ranks higher. So the automatic rule cannot see the very thing the hand audit found. We keep the refutation, because a rule that quietly agreed with us here would have been the more worrying outcome.

One site is executed by no test at all. System A's perturbed constant sits on a line the gate demonstrably runs — but the gate runs several pipeline scripts before it runs `pytest`, and it is one of those scripts that executes the line. Coverage names *zero* covering tests. So the change is dynamically reachable and cannot reach a test assertion, because there is no test on the path. It is the cleanest case in the corpus of the two middle layers coming apart.

What E95 is not. Every one of these sites had already been run through its gate and every outcome was already a miss. E95 explains why those misses are structural; it adds no independent trial. The prospective version of the same measurement is E94, on targets whose outcomes did not exist when the measurement was written.

The rule was wrong once, and we say which part. As locked, it said `pytest` reads test and source files. That refuted our first prediction on three Pingouin cells with catches: its datasets live *inside* the installed package at `src/pingouin/datasets`, and its tests load them through the package, so `pytest` does read them. Amendment 1 adds "data shipped under a source package

root", which is still a path test with no outcome in it. We report the verdict both ways — refuted as locked, confirmed as amended — because a rule repaired after seeing a result is weaker evidence than one that was not.

The rule reproduces the anomaly we found by hand. Applied to the withdrawn lifelines run, it flags that run's G1 catch: a mutation of `docs/Survival Regression.rst` scored CAUGHT by a gate that is `pytest` over `lifelines/tests/`. We originally found that by reading the drawn sites one at a time. A rule that finds it without being told is a rule with teeth.

H.1 TWO ESTIMANDS, AND THE RANKING WE WITHDREW

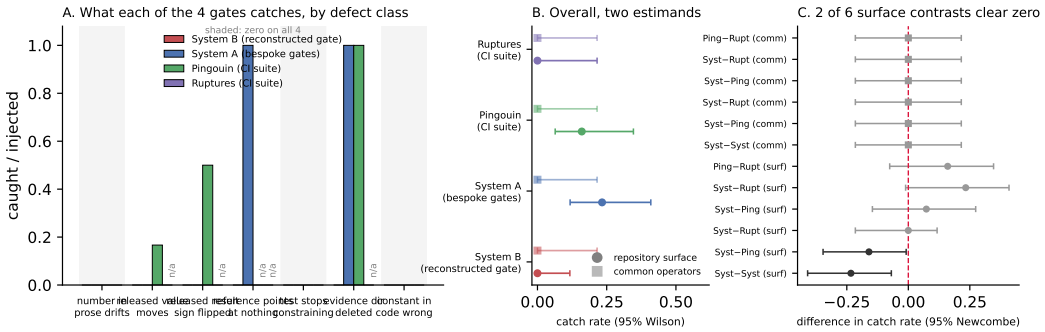

Figure 4: **A**: what each of the four gates we chose catches, by defect class (G1–G7, App. D); System A and System B are the targets we wrote. **B**: overall catch rate under two estimands, with 95% Wilson intervals. **C**: intervals on the *difference*; two of six surface comparisons clear zero, both System B against a target that has gates. Computed after the fact and not corrected for multiplicity.

This was a main-text subsection. It is here because its own conclusion is that it settles nothing, and Table 21 shows why: of the six pairwise contrasts, four span zero and the two that clear it both involve the reconstructed gate; we predeclared no contrast and apply no multiplicity correction. It is a description of four repositories, not a test.

Two rates answer different questions. The **repository-surface** rate averages over every operator with a site population in *that* repository: of defects drawn from its own surface, what fraction does its gate catch? The **common-operator** rate restricts to operators applicable to all four: given the same kinds of defect, which gate catches more? The second is the comparative one: under the surface rate System A banks $4/4$ on an operator Pingouin, with no LaTeX, cannot be scored on.

A ranking needs an interval on the *difference* (Figure 4C). Under the surface rate System B is below System A and Pingouin, and both differences clear zero (-0.233, 95% $[-0.409, -0.069]$; -0.160, $[-0.346, -0.009]$). **Every other comparison spans zero under both estimands.** We predeclared no contrast and apply no multiplicity correction: these are a description of four repositories, not a test.

A claim we withdrew. Our first draft led with "a peer-reviewed library catches less than a bespoke verifier". One inapplicable operator was the entire difference, the comparison reverses on operators applicable to both, and neither direction clears zero; the prediction never said which rate it meant, so it is VOID. The paper ranks nothing except System B against System A and Pingouin.

I A VERDICT-INDEPENDENT WITNESS FOR EVERY WRONG-CONSTANT INJECTION

This is the appendix that argues against the paper, so it is worth saying the result before the method. **Of the 19 wrong-constant injections we scored as missed, the rule we locked in advance witnesses 0 of them.** A rule we wrote afterwards, which reads a project's documentation as well as its artifacts and is not a superset of the first (one site with an artifact probe under the locked rule has no documentation probe under it), witnesses 3. For most of the rest we cannot tell. These counts

are over all 20 sites the protocol locked, including the four in `lifelines`; Table 25 reports the 16 on the four targets (0 witnessed, 3 post hoc), because `lifelines`' gate is not deterministic. Its recorded catch enters no scored count of ours; it stays in the historical 20-site record here, and App. V keeps a separate single-run E112 failure on the same target with its cause unresolved.

Why we needed this at all. A reviewer put the problem in one sentence: *reading the tested gate's own pass/fail report cannot serve as an independent semantic oracle.* If we mutate a constant, run the project's tests, and the tests pass, we have learned nothing about whether the mutation would have changed a result — we have only learned that the tests did not notice. Counting that as a miss assumes the mutation mattered.

We had tried twice.

- **E97** compared what the project's own suite *prints*, before and after. That is the gate's output, so it is not independent, and it did not work either: the output was unstable on all seven repositories and all 28 sites came back unmeasured.
- **E90** did the right thing — call a function directly, fixed seed, compare the number it returns, never look at the gate — on *two* constants of *one* target, with probes written by hand.

E113 keeps E90's method and removes both of its limits: every scored site, and probes built by a rule instead of by us.

The rule, fixed before any outcome was read. For each site, find the innermost function holding the mutated line, then take the first worked example that exists: a `>>>` block in that function's docstring, then its class's, then the module's. If there is none, run a script in the repository that reaches that module and compare the numbers it writes to a JSON file. If neither exists, the site is **no probe** — reported on its own, never folded into "no witness". Each probe runs twice on the untouched tree and once on the mutated one. Nothing reads a gate status, a return code or a log.

Two checks that stop a probe from lying. Both were added because the first version of this experiment returned no witnesses at all, and a probe that cannot fail proves nothing.

1. **Did the mutated line actually run?** A line tracer records which lines of the mutated file the example executes. A "no witness" whose line never ran is not evidence that the constant is inert.
2. **Could the probe see *any* change?** Every no-witness site is re-run with the constant multiplied by ten. If the probe still sees nothing, the probe is blind at that site.

Those two checks found four defects in our own probe before they found anything about the sites. A bare expression in an example was discarded instead of evaluated, so the value the example computes was never captured. An array of matplotlib axes crashed the canonicaliser, so a plot compared equal to itself. A `src/` layout never matched an importing script, so every site in such a project read as no probe. And an example with no seed made every ruptures site unstable.

The result.

	locked rule	with documentation (post hoc)
sites	20	20
a probe could be built	11	13
witnessed	0	3
no witness	11	10
no probe	9	7

W1, W2, W3 and W4 are all refuted. We predicted a probe for at least 12 of 20 and got 11; that at least half of the probed sites would be witnessed and got none; that at least one miss would be witnessed and, under the locked rule, none was; and that the rule would reproduce E90's hand-written labels, which it does not. Only W5 holds: no site was unstable.

defect class	injected	caught	probed	observed effect	unpaired/unknown	measured on
a prose number drifts from its data	47	0	0	—	47	the 13 JOSS
a test assertion is weakened	16	0	7	0	9	the 4 targets
a numeric constant is wrong (locked rule)	16	0	7	0	9	the 4 targets
a numeric constant is wrong (post-hoc rule)	16	0	9	3	7	the 4 targets
a call is swapped for a confusable one	16	4	9	2	3	5 of the JOSS 10

Table 25: **What each procedure could see.** "Probed": a probe was built (constants), the gate called the site (swaps), or a pairing was found (assertions), not that it could see the defect; "observed effect": misses where it saw one; "unpaired/unknown": no probe, never called, or no pairing. Constants and swaps measure a changed returned value, assertions lost gate protection for one perturbation. A 0 beside a non-zero unknown column is not a resolved zero; 6 of the 7 constant probes cannot see a tenfold change, so their 0 is inconclusive. Read down, not across.

What the no-witness column actually means. It does not mean the mutations are harmless. Of the 10 no-witness sites, the mutated line never executed in 3 of them, and in 9 of them (the 3 unexecuted among them) the probe cannot see a *tenfold* change either. So for almost all of them the honest label is **we cannot tell**, not *nothing happened*. Exactly one site — a constant in System B's conformal script — has a probe that demonstrably sees a tenfold change and does not see the 10% one, and that is the only place we can say the recorded perturbation was too small to matter. On the four targets the locked rule built 7 probes; 6 of them executed the mutated line, 1 saw the tenfold control, and 6 did not. Post hoc it built 9, 8 executed the line, and 5 of its no-witness probes were blind to the control. So "probe built" in Table 25 means built, not sensitive: a no-witness result from a probe that is blind to a tenfold change is an inconclusive negative, and we keep those 6 apart from the 9 sites with no probe at all.

Where the rule disagrees with E90, and who is right. E90 published three probes on ruptures and called `display.py:87` a control, on the grounds that its line never executes. Driven by ruptures' own README example, the mechanical probe executes that line and witnesses a change. E90's probe had not found the site inert; it had failed to run. We report both labels and name which came from a hand-written probe and which from the rule. E90's published numbers are not overwritten.

What this does to the paper's headline. The wrong-constant miss rate stands as measured — these are the same injections, scored the same way — but its *interpretation* narrows. Without reading the gate's verdict, we can show that a missed constant would have changed something the project itself computes in 3 of 19 cases. For the rest, the gate passed the mutation, and whether that is a missed error is unresolved. A reader who wants the strong reading — "the gate missed a defect that would have changed a result" — is entitled to it for three sites and no more, and a reader who wants the opposite reading — "the missed constants were harmless" — is entitled to it for one probe's output (System B's conformal script) and no more.

J WAS THE WEAKENED ASSERTION DOING ANYTHING?

The result first. **Of the 20 weakened-assertion injections we scored as missed, we could pair 11 with a perturbation, and all 11 were redundant for that one perturbation.** Not one of them was the thing catching that defect; the gate still went red without it. This test reads the gate, so it measures lost protection for one perturbation, not consequence. These counts include the four sites in `lifelines`, each read from a single run of a gate that is stochastic, so they are kept as recorded and not read; we report the 16 on the four targets (7 decided, all redundant, 9 with no pairing), for the reason given in App. I.

Why this had to be asked. A reviewer put it plainly: ordinary test execution has no reason to fail when an already-passing assertion is weakened, so a zero catch rate is unsurprising unless the experiment shows that the removed assertion was necessary. That is right, and it is a hole in what we report. A suite has no reason to go red just because a check that was passing got weaker. The miss only means something if the assertion was *load-bearing* — if a real defect would now get through that would not have before.

How to find out, in three runs. Pair each weakened assertion with a defect its own test exercises: a numeric constant on a source line the test executes, found with line coverage and perturbed by the same $\times 1.10$ our wrong-constant operator uses. Then run the project's own gate three times.

1. **Perturb the constant, leave the assertion alone.** The gate should fail. If it does not, this pairing is no good and the site is reported as having none.
2. **Weaken the assertion, leave the constant alone.** The gate passes. This is the miss the paper already reports, re-run here as a control.
3. **Do both.** If the gate *still* fails, something else caught it — the assertion was **redundant**. If the gate now passes, the assertion was the thing doing the work — it was **necessary**.

The edits are our own operators, imported rather than rewritten, so each site is mutated exactly as it was when the paper scored it. Both edits are confirmed to have landed before a site is labelled: an edit that silently fails to apply looks exactly like redundancy, because the gate fails for the same reason it failed without the weakening. The merge re-derives that check independently.

The result.

weakened-assertion sites	20
redundant — something else caught the paired defect	**11**
the assertion's own test still failed	6
only other tests failed	5
necessary — the assertion was the only thing catching it	**0**
no pairing: no covered constant the gate constrains	6
no pairing: no constant sits on a line the test runs	3

S1 and S2 are refuted. We predicted a pairing for at least 15 of the 20 and got 11. We predicted that at least half the decided sites would be necessary and got none. S3 and S4 hold: at least one site is redundant — all of them are — and no target's clean gate disagreed with itself across two runs.

The two lines under "redundant" are read from the failing test ids, not from tracebacks, which we did not store. A failing id does not say what failed it. In one `pingouin` case the paired constant is the default of `ttest`'s `confidence` argument, and the library's own range check rejects the perturbed value before any test assertion runs. So "redundant" here means only that the gate still went red (`data/review_20260912/semantic_summary.json` lists each paired site with what was stored).

The two ways a site had no pairing. Both are worth naming, because neither is a shrug. In 3 cases the test executes no perturbable constant at all: ruptures has only fourteen decimal constants in its whole package, so a single test covers none. In 6 cases the test executes constants but the gate does not constrain any of them — we perturbed every covered constant against the assertion's own test, and up to eight of them against the whole gate, and the gate stayed green. In three of these six sites (two in System A, one in System B) a ninth covered constant was never tried against the whole gate, so the search was bounded, not exhaustive. That second group is unresolved, not settled: within the search budget we found no perturbation the gate rejects, so the assertion could not be tested for redundancy, and a constant outside the budget or a defect of another class might still be one it guards.

What this does to the weakened-assertion result. The miss count stands — these are the same injections, scored the same way, and the gates really did stay green. What changes is what the count supports. We cannot say that a missed weakened assertion means a defect would now slip through: in every one of the 11 cases we could decide (7 on the four targets), the suite caught the paired perturbation anyway. The supported statement is narrower: *7 selected pairings on the four targets retained rejection of their particular perturbation after the weakening; 9 assertions were unpaired and are unresolved; other defects were not assessed.* That is not a general no-hole result, and we do not state one.

What this cannot show. Redundancy is measured against *this* pairing. An assertion that is redundant for a wrong constant may still be the only thing standing between the project and a different

defect — a wrong branch, a dropped call, a changed data file. We measured one defect class per assertion, because that is the class our operators can inject on a line the test runs. A wider claim would need a wider pairing, and we do not make one.

K DOES THE WRONG-ANALYSIS SWAP CHANGE ANYTHING?

The result first, because it is the largest correction in this paper. **Of the 16 wrong-analysis injections we scored, 7 returned the same value under the swap on every call their gate made, and 3 more never ran at all.** All 10 of them were recorded as misses. Only 2 of our recorded misses are misses of a mutation that changed a returned value on the gate's own inputs. This says nothing about inputs the gate never used.

The reviewer saw this coming. Half of the 16 scored mutations are `sum`/`nansum` substitutions, and a reviewer asked us to establish whether a relevant input ever contains a NaN and to show the returned difference. The arithmetic is not subtle. `np.sum` and `np.nansum` return the same number for any array with no NaN in it. If the array a project sums never holds a NaN, swapping one for the other changes nothing, the gate had nothing to catch, and calling that a miss credits the defect with a consequence it does not have. We had scored 8 of these swaps without checking the condition.

How we checked it. A probe is loaded before the project imports anything. It wraps the functions our operator swaps. On every call from one of our sites it records whether the argument holds a NaN, what the original returns, and what the swap *would* return — on the same argument, in the same process. Then it returns the original, so the project computes exactly what it always computes and the gate's own outcome is untouched. Nothing here reads a gate result.

Three details decide whether such a probe is honest.

1. **A call is a span, not a line.** Our operator edits the line where the text sits, which is not always the line the interpreter attributes the call to. One site is the closing parenthesis of a `sorted(` that opens two lines earlier. Matching on the exact line reports it as never executed — which is how our own earlier measurement came to label that site *not dynamically reachable* while still scoring it as a catch. The probe matches when the drawn line falls anywhere inside the call's span, and that site turns out to run six times and to change its result.
2. **One line can hold two calls.** PYroMat's line 873 is `np.sum(mw * self._q, axis=0) / np.sum(self._q, axis=0)`, and the operator drew it twice. Our first probe recorded the column of each call but keyed sites by file and line, so the two draws received one merged record set. The corrected probe keys a site by the full span of the call expression, recovered from the column of the edit in E96's stored diff, and the two draws are two sites: 500 calls each, none differing.
3. **Every call must be compared, not the first forty.** Our first probe counted every call but stopped comparing after forty stored records per site. On a fixture with forty equal calls and a NaN on the forty-first it reported *equivalent*. 7 of the executed sites were called more than forty times, so their labels rested on a prefix; a reviewer found both defects. The corrected probe compares every call and caps only the stored examples. The numbers below are from a rerun of all five targets with it, in environments rebuilt at the recorded commits (`data/review_20260912/e116/`); the locked labels are kept as recorded, and on every one of the 13 executed sites the fresh label agrees with the locked one, now with every call compared.

Before it was pointed at anything real, the probe was run on a fixture built to produce both answers: a sum over an array with a NaN (it reports a difference), a sum over one without (it reports none), and a call split across lines (it matches). A probe that can only say one thing proves nothing.

The result.

	all 16 sites	the 8 `sum`→`nansum`
changes a value on an observed call	6	**0**
no difference on any observed call	7	5
never executed	3	5 of 8 did run
ever saw an argument with a NaN	—	**0**

Not one of the 8 `sum`→`nansum` swaps ever saw a NaN. The 5 executed sites were called 520, 56, 500, 500, 224 times; every call was compared and none differed. All 8 were recorded as misses. Not one of them could have been caught on the calls the gate made.

What that does to the numbers. The locked counts stand: they are what we ran and what the gates said. Beside them we report the same counts with the sites that never differed on an observed call removed.

	caught	missed
as locked and reported	4	12
counting only sites that demonstrably change a value	4	**2**

Every catch survives: all 4 were on mutations that really do change a value, so the gates caught real things. What shrinks is the miss column, from 12 to 2. **We are not re-scoring anything** — both rows are printed, and the artifact lists every site removed and why.

Predictions. N2 and N4 hold: at most two executed swaps would see a NaN (none did), and at least one scored site does change a value (6 do). N1 and N3 are refuted, both by one site: we predicted at least six of the eight swaps would execute and six would be equivalent, and got 5 and 5. N3's fallback says to conclude that these swaps really do change the answer. **We do not execute it, because its premise is false** — not one executed swap was non-equivalent. N3 fails only because three sites never ran, and a site that never runs is not evidence either way.

What this cannot show. Equivalence is measured on the data the gate runs. A swap that changes nothing on a test suite's inputs may change the answer on a user's real data. That is a statement about what these suites exercise, which is what this paper measures. It is not a claim that the code is correct.

L THE EVIDENCE IN TWO TABLES

A reviewer asked for one corpus-accounting table and one mechanism-and-repair table, with no inference from a pooled total of mixed applicability. These are those two tables. Both are generated from the artifacts, both are counts, and **neither has a total row**. That absence is the point. Adding two corpora whose operators differ produces a number no reader should divide, which is the thing the request warns against.

Table 26: **Corpus accounting.** Every injection in this paper, by corpus. The two rows are not addable: the corpora were injected with different operator sets on repositories with different surfaces, so a combined rate would mix applicability. The last column says where the breakdown lives. The second row's counts are the raw record, before two malformed mutants were voided, matching Table 24.

corpus	repos	operators	injections	caught	detail
the 4 repositories we chose	4	7	98	11	Tab. 20
repositories we did not choose	13	5	165	39	Tab. 24

How to read the first table. The 4 repositories we chose took 98 injections across all seven portable operators and their own gates caught 11. The 13 repositories we did not choose — drawn from a public index by seed — took 165 injections across five operators and caught 39. Those two

Table 27: **Mechanisms and repairs.** What each added check catches, beside the population it was measured on. The population column is not decoration: the four rows count different things on different repositories, and the only safe reading is down a row, never across. The last row carries the corrected count and the locked one beside it: of the nine seeded drift draws scored as drifts, three are numeric statements. The first two rows are App. H; the last two are App. M.

repair	the defect it looks for	caught	measured on
expected-set validation	a released directory is gone	$8/8$	held-out deletions, System A
mention pinning	a prose number drifts from its data	$9/10$	held-out drifts, System A
both, in repositories we did not write	a released directory is gone	$13/13$	15 public compendia
	a numeric statement drifts from its data	$3/3$, locked $9/9$	15 public compendia

numbers describe different experiments. The first is four gates we know well; the second is a sample we did not pick, with a narrower operator set. A reader who divides either one gets a rate for that corpus and nothing more.

How to read the second table. Each row is a repair developed on failures we had already seen, frozen, and then tested on defects it had not. The third row is the one that matters most for whether any of this generalises: the same two rules, ported to 15 research compendia by other authors, catching every held-out deletion, and every drift at a numeric statement (3 of 9 seeded draws; the rest were identifiers, versions, references or a figure width), with 0 false alarms across 43 trivial controls. What the table cannot show is a rate across its rows, and it does not offer one.

What these two tables deliberately do not replace. Table 20 keeps the per-repository, per-operator cells; Table 24 keeps the outcome-blind classes; Table 22 keeps which operators each target could be scored on at all. Consolidating means giving a reader one place to see the shape of the evidence, not deleting the evidence. Every number in both tables above is also somewhere below, and nothing was dropped to make these fit.

M THE TWO REPAIRS, RUN ON COMPENDIA WE DID NOT WRITE

Both repairs in this paper were developed on our own repository and tested on defects they had not seen — but still on our own repository. A reviewer asked the obvious next question: do they work anywhere else? This appendix runs them on research compendia belonging to other people.

Who picked the repositories. The frame is GitHub's own topic `research-compendium`, restricted to `language:Python`. Other people applied the label to their own work; we chose the topic and the language filter. It is a convenience frame of 15 repositories with 8 owners, 7 of them from one owner, not a sample of research compendia, and choosing an outside label does not make it one. Every repository the search returns enters: 16 of them. One leaves, with its reason recorded — it ships no data directory and states no number in prose, so neither repair has anything to hold. That leaves 15.

What ported, and what did not. Our own repairs are wired into our own layout: one is a list in `paper/shipped_data_dirs.txt` that our supplement script enforces, the other a baseline that our consistency checker enforces. Neither script would run anywhere else. What ports is the *rule*:

1. **A directory lock.** Write down the data directories a repository ships. Fail when one is gone.
2. **A mention pin.** Write down how many times each number appears in the prose. Fail when a count drops.

Both are built from the target repository's own state, and both are frozen before any defect is drawn. The drift draw was meant to skip a DOI, an arXiv identifier or a version string, the same intention as our G1 operator's rule, because changing one of those tests whether a checker reads identifiers, not whether it reads measurements. The filter as run did not achieve that: 6 of the 9 seeded draws were an encoded DOI, an SQLite version, an Ubuntu version, two section references and a LaTeX figure width. They are excluded post hoc below, and the locked verdicts are kept as scored.

The first result needs no injection at all. 11 of the 15 lack the prespecified markers: no CI workflow, no Makefile, no `noxfile`, no `tox.ini`, no `run_all.sh` or `reproduce.sh`. That is the count R1 was scored on and it stays as scored. It is a statement about eight filenames, not about verification.

Reading all fifteen (post hoc, review of 12 September). The marker count says nothing about the eleven repositories it did not read, so we fetched all fifteen at the commits E114 recorded, by commit hash and never by branch, and read every marker file, every README, and every script either one names, one level deep: 153 files, stored with their SHA-256 (`data/review_20260912/e114_scope_sources/`). The 4 marker files are builds: a Sphinx documentation build, two Makefiles that produce figures and a manuscript, and one CI workflow whose only step is `make BUILD_DIR=_build`; the earlier inspection of these four (E114b) fetched them from the default branch, and their hashes match the pinned commits, so it read the right bytes but only those five files. 7 of the 11 marker-negative repositories, all by one owner, document a `compendium.sh` that the marker rule does not name; each clones a solver, runs it, and draws the paper's figures. Two repositories name no script at all in their README. Across the 153 files, none runs a test runner and none compares an output to a stored reference. One (`fickettmodel-reproducibility`) executes 18 `assert` statements in the scripts its build calls; they check array lengths, types and parameter ranges, not the paper's numbers. We never ran any repository's build, so there are no original-gate outcomes to report. What this bounds: in what these compendia document running, we found no check of a released number against the paper. What it leaves open: 129 script files that nothing documented names, and the external solvers the run scripts clone, were not read. Detection is lexical (Python by the tokenizer, shell and Make by pattern), so a check written in an unlisted form would be missed; the stored sources let a reader re-check by hand.

Held-out defects and controls. For each repository, with both baselines built and frozen first, we remove one shipped data directory drawn with seed 114, change one prose number drawn with the same seed, and make three controls: a blank line appended to a prose file, a comment line appended to a prose file, and a data file re-saved with identical bytes.

	caught	tested
a shipped data directory disappears	13	13
a prose number drifts (locked draw, as scored)	9	9
a prose number drifts (numeric statements only, post hoc)	3	3
of which a stated result of the repository	1	1
draws excluded as identifier, version, reference or layout	—	6 of 9
false alarms on controls (15 blank-line, 15 comment, 13 unchanged re-save)	0	43

All five locked predictions hold on the locked counts, and the worst repository costs 0.62 seconds to build both baselines and run both checks. The locked run reported 9 of 9 drifts caught. A reviewer read the nine draws against their pinned sources: five were not measurements (an encoded DOI, an SQLite version, an Ubuntu version, two section references), one was a LaTeX figure width, and on that repository the run changed the wrong number, because first-substring replacement altered an earlier `0.89 m` instead of the drawn `0.8`. The historical verdicts are kept as recorded. The corrected rows are post hoc: each site was annotated from the pinned source (`data/review_20260912/e114_sites.json`), then mutated at its exact span with the file hash checked before and after (`e114_corrected_scored.json`). Three drift sites, one of them a stated result, cannot support a rate. The controls include no numeric reformatting, claim relocation or rename; the pin alarms on an equivalent reformatting such as $0.50 \to 0.500$ and stays silent when

two claims' values are swapped, so the zero false alarms says nothing about realistic maintenance edits.

Two honest deductions from that table. It is cleaner than it looks.

First, **the deletion row is close to true by construction**. The lock lists the directories and the test removes one of them; of course the lock notices. What is not automatic is the rest: that the lock can be built at all from a repository we did not write, that it passes on the untouched tree, and that 43 controls leave it quiet. Those are the parts that could have failed.

Second, **the mention pin had nothing to hold in 6 of 15 repositories**. Several of these compendia state almost no numbers in their prose — the paper is elsewhere, and the repository is the code and the data. So the drift row is 9 of 9, not of 15, and we report it that way. Our own repository, where the paper and its numbers live in the same tree, is the easy case for this repair rather than the typical one.

What this does not show. The 15 eligible repositories have 8 owners: 7 belong to one person, two to another, and six are single. Other people applied the topic label, but we chose the topic and the Python filter. This is a convenience frame of 15 repositories, not an estimate of how common checks are across research compendia. The topic `research-compendium` spans astronomy, geography and archaeology; almost none of these are machine-learning compendia. So this answers *do the repairs generalise to research compendia we did not write*. It does not answer *do they generalise to ML compendia*, and we are not going to let the first stand in for the second.

N ONE DEFECT, ONE PUBLISHED ABSTRACT, ONE CI COMMAND

The result first. On one published compendium by unrelated authors, a one-line code defect changed the computed overall accuracy that supports the paper's abstract from 97% to 98% and the computed messy-file improvement from "almost 22%" to "almost 20%". The compendium's own CI command regenerated the changed numbers, printed the change, and exited 0. The frozen mention pin caught the change, a one-line value lock on the committed outputs caught it too, and the pin saw neither number in the abstract. The paper's source reads this number from a file the build rewrites; the build does not recompile the PDF, and neither did we. Under 5 evidence-preserving maintenance edits the two frozen repairs fired on 3 of them (the pin on the reformatting and the moved citation, the directory lock on the renamed directory) and the value lock on 1; all cleared under the frozen update policy. 10 of the 10 locked predictions were confirmed and 0 refuted. This is one case. It is a mechanism with a worked example, not a rate.

Why this case exists. A reviewer asked two questions of this paper. First: *can a defect be shown to change an independent ML paper's scientific output, while the inherited gate misses it and the repair adds protection?* Second: *what does the same repair do under legitimate numeric reformatting, relocation, renaming or baseline updates?* App. M could not answer either. Its defects were deletions and prose edits, its controls were blank lines, comment lines and no-op re-saves, and none of its targets was a paper's claim-producing pipeline with a gate of its own. E121 takes one such pipeline. Its protocol was committed before any run (`experiments/121_consequential_protection/protocol.md`); every edit, every arm, every prediction and what each outcome would license was fixed there. One dated addendum, committed before the first run, fixed how six of the protocol's ambiguous sentences are read; it changed no prediction, arm, edit or definition.

The target and its claim path. The target is CSV_Wrangling (van den Burg et al., 2019), the compendium of a data-mining paper on CSV dialect detection, by authors unrelated to us, under an MIT licence, at one pinned commit with 150 tracked files. We surveyed 13 candidates against 6 requirements fixed first: unrelated authors, a permissive licence, a small tree, a CPU-only path from committed data to a number the paper prints, a gate the authors shipped, and no overlap with any earlier population of ours. CSV_Wrangling is the only one that met all 6. The survey ships with the artifact. The detector outputs the paper is computed from are committed, so `make output` needs no download. It writes 27 text fragments, 17 constants and 10 tables, and one figure. The paper's source, which is not in the repository, `\inputs` the fragments. Its abstract reads "achieves

97% overall accuracy" and "by almost 22%"; the committed fragments say the same, and so does the witness's pristine build (97% and 22%). Three facts about the `Makefile` decide what follows. First, two human-annotation targets are declared `.PHONY`, so everything downstream of them, 23 of the 27 fragments, is remade on every run whatever the file times say. Second, no recipe lists a script as a prerequisite, so a code edit reaches the outputs only through that remaking. Third, the detector recipes fire on file times. So before any run we set every tracked file's time to the commit time; `make -n` on the pinned tree then plans the same recipes under either make version. A validity gate re-checks the plan before every gate run and forbids the six corpus-building commands by name.

The inherited gate. The project's `.travis.yml` runs, verbatim, `make output && git diff`. `git diff` without `--exit-code` exits zero whether or not it prints a diff. So the gate can fail only when `make` fails. It cannot fail on a changed number: it regenerates the number, prints the change, and passes. We ran exactly that string (`gate_is_reconstructed` is false). Travis no longer runs this repository, and whether the gate was ever green is not known.

The nine edits and the five arms. One defect (D), one positive control (PC), one liveness control (C1) and six benign edits (B1–B6), each fixed with its exact span and expected effect before any run. D, PC and C1 share one line, line 69 of `scripts/common/dialect.py`, which returns the identity key of a dialect. D reduces the key from (delimiter, quote character, escape character) to the delimiter alone, a one-line refactoring error; every "detected dialect equals the reference dialect" test, and the set that counts distinct dialects, then see only the delimiter. PC makes every dialect equal to every other. C1 misspells an attribute so the first comparison raises `AttributeError`. B1 drops trailing zeros when two constants are printed (same values). B2 moves the README's citation block into a new file. B3 renames a data directory with `git mv`. B4 retracts one file from the GitHub corpus, which the README says can happen; it changes numbers on purpose. B5 renames an identifier. B6 rewrites an if/return pair as one boolean return, at a site like D's. The arms all read the same work tree after the same `make output`. **G0** is the inherited gate, verbatim. **G1a** is the naive one-flag fix, `git diff --exit-code`. **G1** is the value lock: `git diff --exit-code` on the 27 claim-set paths, the simplest gate that pins the paper's numbers using only what the repository already commits, chosen before any run as our strong comparator. **R** is the two frozen repairs of App. M, built once on the pinned tree: 5 locked directories, and a mention baseline of 242 distinct decimal literals with 263 mentions in 21 of the 35 prose files scanned. The generated `.tex` fragments count as prose by suffix, so the pin holds their decimals; that is the only way it can see a regenerated number. **W** is the witness, not a gate: in a separate tree it applies the edit, deletes the 27 fragments, runs the project's own recipes for them, and compares every constant and every table cell, 269 leaves in all, against the pristine build. It never reads a gate. A tracer records whether line 69 ran. Table 28 lists every gate run by arm.

The defect, derived before the run. D was chosen because its consequence could be derived from the committed detection records, not because it is typical. Before any run the protocol re-counted those records with the full key and with the delimiter-only key. Under the reduced key more detections count as matches, so the overall accuracy rounds up; fewer dialects are distinct; and the messy-file improvement over the baseline detector shrinks. The predicted values were written into the protocol: overall accuracy 98, messy-file improvement 19.0 with ceiling 20, distinct dialects 12 (GitHub), 5 (UK data), 13 (total).

What happened on D. W is WITNESSED: 118 of 269 leaves differ from the pristine build, and the tracer records line 69 as executed. Measured beside predicted: overall accuracy 98% (pristine 97%, predicted 98%); messy-file improvement 19.0% (pristine 21.4%, predicted 19.0%) with ceiling 20% (pristine 22%, predicted 20%); distinct dialects 12/5/13 (pristine 33/8/34, predicted 12/5/13). Every measured value equals its prediction. The overall improvement over the baseline, a fragment the paper does not `\input`, moved from 8.6% to 8.0%. The file counts and the two failure proportions are byte-identical to pristine. In each of the six accuracy tables the *Overall* row now equals the pristine *Delimiter* row, cell for cell, which is what a key reduced to the delimiter must produce. On the gate side, `D-frozen`: G0 regenerated the fragments, wrote 98% into the accuracy constant, printed a diff naming 23 paths (19 of them changed fragments; the rest are the summary JSON files and the figure), and passed. G1 failed. R failed with 105 mention alarms, among them the 21.4 of

run	edit	witness (leaves differing)	G0	G1a	G1	R	R alarms: pin / lock
`clean-01`	clean (no edit)	—	pass	**fail**	pass	pass	0 / 0
`clean-02`	clean (no edit)	—	pass	**fail**	pass	pass	0 / 0
`clean-03`	clean (no edit)	—	pass	**fail**	pass	pass	0 / 0
`clean-04`	clean (no edit)	—	pass	**fail**	pass	pass	0 / 0
`clean-05`	clean (no edit)	—	pass	**fail**	pass	pass	0 / 0
`C1-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`C1-frozen`	C1: first comparison raises	—	**fail**	pass	pass	pass	0 / 0
`PC-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`PC-frozen`	PC: every dialect made equal	witnessed (119)	pass	**fail**	**fail**	**fail**	105 / 0
`D-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`D-frozen`	D: dialect key reduced to its delimiter	witnessed (118)	pass	**fail**	**fail**	**fail**	105 / 0
`B1-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`B1-frozen`	B1: drop trailing zeros (same values)	no witness	pass	**fail**	**fail**	**fail**	2 / 0
`B2-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`B2-frozen`	B2: move the citation block to a new file	no witness	pass	**fail**	pass	**fail**	1 / 0
`B3-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`B3-frozen`	B3: rename a data directory	no witness	pass	**fail**	pass	**fail**	0 / 1
`B4-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`B4-frozen`	B4: retract one corpus file	witnessed (63)	pass	**fail**	**fail**	**fail**	39 / 0
`B5-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`B5-frozen`	B5: rename an identifier	no witness	pass	**fail**	pass	pass	0 / 0
`B6-pair`	clean pair of the edit below	—	pass	**fail**	pass	pass	0 / 0
`B6-frozen`	B6: equivalent refactor at a site like D's	no witness	pass	**fail**	pass	pass	0 / 0
`B1-policy`	B1, with its policy step	no witness	pass	pass	pass	pass	0 / 0
`B2-policy`	B2, with its policy step	no witness	pass	**fail**	pass	pass	0 / 0
`B3-policy`	B3, with its policy step	no witness	pass	**fail**	pass	pass	0 / 0
`B4-policy`	B4, with its policy step	witnessed (63)	pass	pass	pass	pass	0 / 0
`D-on-B1`	D on the B1 policy tree	witnessed (118)	pass	**fail**	**fail**	**fail**	105 / 0
`D-on-B2`	D on the B2 policy tree	witnessed (118)	pass	**fail**	**fail**	**fail**	105 / 0
`D-on-B3`	D on the B3 policy tree	witnessed (118)	pass	**fail**	**fail**	**fail**	105 / 0
`D-on-B4`	D on the B4 policy tree	witnessed (118)	pass	**fail**	**fail**	**fail**	105 / 0

Table 28: **Every E121 gate run, by arm.** 5 clean runs, 9 clean pairs with their 9 frozen edit runs (the edit, no policy step), 4 policy runs (the edit with its P1 rebuild or P2 regeneration), and 4 runs of D on top of a rebuilt policy tree. "Witness" is W's label for the edit with the number of leaves that differed from the reference build (C1 gets no witness run; clean runs and pairs carry no edit). G1a fails on every clean run because `latexmk` rewrites the figure. R's alarms are the mention pin's count and the directory lock's missing directories, against the pristine baseline for frozen runs and against the run's rebuilt baseline for policy and D-on-B runs.

the messy-file constant and cells of the GitHub accuracy table, and no alarm naming the accuracy constant, the ceiling constant or the two dialect-count constants: those four hold integers.

The controls. PC is WITNESSED with 119 leaves differing and line 69 recorded: overall accuracy 100%, messy-file improvement 9.3%, one distinct dialect (1), and in the GitHub accuracy table every *Overall* cell equals one hundred minus the same column's *No Result* cell, to within a hundredth. So the witness can see, and the site runs. C1 makes `make` exit non-zero at the first comparison; G0 failed, the shipped log tail holds `AttributeError`, and `C1-pair` passed. So the harness runs the project's scripts and the gate can fail. Both controls were decided before any label for D or a benign edit was read.

Maintenance: frozen, then with the policy. W is NO WITNESS on B1, B2, B3, B5 and B6 and WITNESSED on B4 (63 leaves; the GitHub file count reads 4385 instead of 4386). G0 passed all six frozen runs: no maintenance breakage. Without its policy step, R fired on B1, B2 and B3, 3 of the 5 evidence-preserving edits (nuisance alarms; the protocol calls them false alarms), and each has a named mechanism. B1: literal text. 0.30 and 0.3 are different tokens to the pin, so the two reformatted constants raised 2 alarms. B2: per-file counting. The DOI fragment moved to another file, and the pin counts per file, so 1 alarm. B3: directory identity. The lock lists `results/dev/detection` and the renamed directory is not it, 1 missing directory. B5 and B6 change no prose file and no directory, and R stayed quiet. G1 fired on B1 alone, 1 nuisance alarm in 5: the committed fragment bytes changed although the values did not. G1a failed on all five, as it does on every clean run (below). With the policy, B1–B4 each got one rebuild in the same commit as the edit. P1 rebuilt R's lock and baseline 4 times; the entries that changed were 3 (file,

literal) pairs on B1, 2 pairs on B2, 2 lock directories on B3 (the old name and the new), and 78 pairs on B4. P2 regenerated and committed the outputs 2 times, on B1 and B4. B5 and B6 got no step because the policy asks for none. G0, G1 and R then passed on all 6. Protection survived every rebuild: on each of the 4 `D-on-B` runs, D applied on top of the rebuilt policy tree, W is WITNESSED (118 leaves), G0 passed, G1 failed, and R's alarm list against the rebuilt baseline names the GitHub accuracy table.

The one-flag fix. A maintainer who notices the missing flag writes `git diff --exit-code`. That is G1a, and it failed on all 5 clean runs. Every path its diff named is under `results/test/analysis/figures/` (2 paths): `latexmk` rebuilds the runtime figure on every run, and on our toolchain its bytes differ from the committed figure. The 27 text fragments are not the reason. G1a passed on 3 of the 31 runs: `C1-frozen`, where `make` failed before the figure was drawn (an arm verdict; the workflow rejects C1, because the project's own check failed), and the B1 and B4 policy runs, whose P2 step had just committed a fresh figure that the gate's rebuild then matched byte for byte. So the rebuild is stable on our machine and differs only from the authors' bytes. Against the original committed figure, G1a failed the clean runs on our toolchain, because regenerated figure bytes differ. Those two policy runs passed because a figure regenerated by that toolchain had just been committed; the B2 and B3 policy runs, with no regeneration, failed; restricting the comparison to the claim files avoids the figure dependency. The value lock, which diffs only the claim set, passed clean as shipped.

What the pin cannot see, and what the value lock is. The mention pin matches decimal literals, `\d+\.\d+`. Both numbers in the abstract, 97% and 22%, are integers, and so are the two dialect counts. The pin never pinned any of them. It caught D through the decimals in the regenerated fragments, a side effect of the outputs being `.tex` files. The value lock sees every byte of the 27 committed fragments, integers included; it exists here only because the authors commit their outputs, and it fires on a legitimately reformatted byte (B1) until the outputs are regenerated and committed with the change. Neither gate knows which fragments the abstract quotes. On this compendium the frozen repairs did no better than the value lock: both rejected every fault the witness saw (D, PC and the 4 `D-on-B` runs), both fired on the authorized update B4, which needs its reference rebuilt and is not a fault, and the frozen repairs fired on 3 maintenance edits to the value lock's 1. The protocol said that if this happened the paper would say so; this is where it does.

Timing. Every run records gate, check and rebuild seconds with a `concurrent` flag, true when the machine's load average at the start of the run was above the protocol's threshold. The flag is true on 45 of the 45 runs; the artifact records the load, not its cause. The protocol makes no timing prediction, and this appendix makes no timing claim.

The ten predictions, locked wording and observed value.

1. **X1, the clean gate is stable** (CONFIRMED). Locked: all clean runs exit zero, the plan lists a recipe for each remade fragment and no corpus command, every remade file's time advances, and the claim manifest hash is unchanged. Observed: all 5 did.
2. **X2, the defect changes the paper's numbers** (CONFIRMED). Locked: W is WITNESSED and the accuracy, messy-improvement, ceiling and three dialect-count fragments read the derived values. Observed: 98%, 19.0%, 20%, 12/5/13; 118 leaves differ.
3. **X3, the witness is sensitive** (CONFIRMED). Locked: PC is WITNESSED, line 69 ran, and the accuracy, dialect-total and messy-improvement fragments read the all-equal values. Observed: 100%, 1, 9.3%.
4. **X4, the inherited gate misses the defect** (CONFIRMED). Locked: on `D-frozen` the gate exits zero, the accuracy fragment in the work tree reads the wrong value, and the diff stat names it and the GitHub accuracy table. Observed: exit 0, fragment 98%, both names in a stat of 23 paths.
5. **X5, the one-flag fix cannot be used as shipped** (CONFIRMED). Locked: G1a exits non-zero on every clean run and every named path is under `figures/`. Observed: 5 of 5, 2 figure paths each time.
6. **X6, the value lock works here** (CONFIRMED). Locked: G1 exits zero on every clean run and every pair, non-zero on `D-frozen`. Observed: as locked.

7. **X7, the frozen pin catches the defect through decimals and is blind to the four integers** (CONFIRMED). Locked: the alarm list names the messy-improvement decimal and a GitHub accuracy cell, names none of the four integer fragments, and the lock is silent. Observed: as locked, 105 alarms.
8. **X8, the gate can fail** (CONFIRMED). Locked: `C1-frozen` fails at `make` with `AttributeError` in the log tail; `C1-pair` passes. Observed: as locked.
9. **X9, what each arm does under maintenance** (CONFIRMED). Locked: W's labels on B1–B6; G0 passes all six; R fires on exactly B1, B2, B3 with the named alarm strings; G1 on exactly B1; with the policy all three arms pass on all six after four P1 rebuilds and two P2 regenerations. Observed: as locked, 3 and 1 false alarms, 4 rebuilds, 2 regenerations.
10. **X10, protection survives the policy** (CONFIRMED). Locked: on each `D-on-B` run W is WITNESSED, G0 passes, G1 fails, and R's alarms against the rebuilt baseline name the GitHub accuracy table. Observed: as locked, on all 4.

What this cannot show. One compendium, one defect, chosen by us after reading the code. This is a mechanism with a worked example. It is not a catch rate, not a false-alarm rate, and not a statement about ML compendia in general; this appendix prints counts and never a percentage of them. The defect was chosen because its consequence could be derived in advance, not because it is typical. The value lock exists here only because the authors commit their outputs; on a compendium that does not, it has nothing to compare, and this case says nothing about that. The defect is a code edit, and the mention pin was built for prose drift; both abstract numbers are integers the pin never held, and the case measures that gap rather than closing it. The benign edits are ours: six edits cover four maintenance classes once each, and a maintainer's real history would be the better population. The gate ran on our machine in a copied tree with normalised file times, not in the project's Docker image; the command is theirs, the machine and the file times are ours. The paper is a data-mining paper, not an ML paper. A fallback ML target with no gate of its own was declared in the protocol and was not needed. The witness shows that the printed number moved; it does not show that a reader would have noticed, or that the published PDF would have been rebuilt.

N.1 Lock, pin and a numeric comparison on the same target (E122)

E121 left one question open: if the frozen pin adds nothing over the value lock on this compendium, what *should* a compendium run? E122 answers it on the same target at the same commit, under a protocol committed alone and first. Six arms read one work tree after one `make output`: **G0** the inherited CI command; **G1a** `git diff --exit-code`; **G1** the value lock over the 27 claim paths; **R** split into `R_lock`, the directory lock, and `R_pin`, the mention pin; and **G2**, a numeric claim comparison built once on the pristine tree over the same 35 prose files the pin scans, which registers each number and compares them numerically, so a trailing zero does not make a second number and an integer is a number.

Eleven edits: D, PC, C1, B1 and B4 copied from E121's file; S1, a swap of two values inside one regenerated table; S2, the same swap across two committed fragments the build does not remake; I1, the integer percentage in the abstract's fragment; I2, a changed integer in the README (its compute-time estimate, which nothing regenerates); and L1, L2, deletions of a frozen baseline file, one for each repair design. Each runs with a clean pair, giving 28 runs and 168 arm-by-run cells. All 11 locked predictions were confirmed, no cell is unresolved and no edit failed to apply. A miss is scored only on the 6 edits whose class is defect, positive control or claim drift; a nuisance alarm only on the evidence-preserving one; a detection on any edit whose frozen run failed while its clean pair passed. Those are three different populations and the generator asserts each count against its own before writing a macro.

Per arm. G0 detected 1 (the crash control) and missed 6. G1a fails on every clean run, as in E121. G1 detected 6 and missed 2, namely S2 and I2, with the violation still in the tree. `R_lock` detected 0 and missed 6. `R_pin` detected 5, missed 3 and raised 1 nuisance alarm, on B1. G2 detected 6, missed 2 and raised 0. Setup, as declared in the protocol rather than measured: G1 needs 27 lines of claim paths, G2 needs 0 beyond running its builder.

What it licenses. X11's locked wording fired: *the frozen pin adds nothing over a numeric comparison here*, and the numeric comparison did strictly better on two cases — no alarm on B1, an alarm on I2. This may not be extended beyond this compendium and these 11 edits. X4: S1 is invisible to every count-based arm because counts are preserved, and the gate's own `make` undid it before any arm read the tree, so only the value lock fired; on a compendium that does not regenerate its fragments nothing here would have caught the swap. X5: S2 stayed in the tree through a passing gate and a passing value lock, and both per-file counting arms caught it. X6 and X7: the make had already restored the abstract's fragment before either arm read it, so G2 named that integer only in its pre-make reading, and I2, an integer in the README (the compute-time estimate) that nothing regenerates, was caught by G2 alone. X8: deleting a frozen baseline is loud in both designs, by an exception in R and by a declared alarm in G2, and no arm passes silently. 2 frozen runs ended with the generated output restored while the candidate commit still held the edit, which is why Table 29 prints the post-gate state for every run.

run	G0	G1a	G1	R_lock	R_pin	G2	post-gate state
`B1-frozen`	pass	**fail**	**fail**	pass	**fail**	pass	edit persists
`B1-pair`	pass	**fail**	pass	pass	pass	pass	none
`B1-policy`	pass	pass	pass	pass	pass	pass	edit persists
`B4-frozen`	pass	**fail**	**fail**	pass	**fail**	**fail**	edit persists
`B4-pair`	pass	**fail**	pass	pass	pass	pass	none
`B4-policy`	pass	pass	pass	pass	pass	pass	edit persists
`C1-frozen`	**fail**	pass	pass	pass	pass	pass	edit persists
`C1-pair`	pass	**fail**	pass	pass	pass	pass	none
`D-frozen`	pass	**fail**	**fail**	pass	**fail**	**fail**	edit persists
`D-pair`	pass	**fail**	pass	pass	pass	pass	none
`I1-frozen`	pass	**fail**	**fail**	pass	pass	pass	restored
`I1-pair`	pass	**fail**	pass	pass	pass	pass	none
`I2-frozen`	pass	**fail**	pass	pass	pass	**fail**	edit persists
`I2-pair`	pass	**fail**	pass	pass	pass	pass	none
`L1-frozen`	pass	**fail**	pass	**error**	**error**	pass	none
`L1-pair`	pass	**fail**	pass	pass	pass	pass	none
`L2-frozen`	pass	**fail**	pass	pass	pass	**fail**	none
`L2-pair`	pass	**fail**	pass	pass	pass	pass	none
`PC-frozen`	pass	**fail**	**fail**	pass	**fail**	**fail**	edit persists
`PC-pair`	pass	**fail**	pass	pass	pass	pass	none
`S1-frozen`	pass	**fail**	**fail**	pass	pass	pass	restored
`S1-pair`	pass	**fail**	pass	pass	pass	pass	none
`S2-frozen`	pass	**fail**	pass	pass	**fail**	**fail**	edit persists
`S2-pair`	pass	**fail**	pass	pass	pass	pass	none
`clean-01`	pass	**fail**	pass	pass	pass	pass	none
`clean-02`	pass	**fail**	pass	pass	pass	pass	none
`clean-03`	pass	**fail**	pass	pass	pass	pass	none
`clean-04`	pass	**fail**	pass	pass	pass	pass	none

Table 29: **E122: every arm on every run, one work tree, one `make output`.** Edits: D is the defect, PC and C1 the controls, S1/S2 swapped claim values, I1 the abstract's integer percentage, I2 the README's integer compute-time estimate, L1/L2 a deleted frozen baseline, B1 an evidence-preserving edit and B4 a legitimate evidence update. Arms: G0 the native command `make output && git diff`, G1a the naive `git diff --exit-code`, G1 the value lock, `R_lock` and `R_pin` our two frozen repairs, G2 the numeric claim comparison. Runs: `-frozen` is the edit, `-pair` its clean pair, `-policy` the edit after the policy rebuild. A detection is a **fail** whose pair passed; misses are scored only on D, PC, S1, S2, I1 and I2. "Restored" means the gate's own build rewrote each edited file to its unedited bytes before any arm read it, and every claim value read after the gate equals the unedited tree's (`make_tables.py` asserts both); the candidate commit still holds the edit; *edit persists* means the edited bytes are still in the tree after the gate, which is not a statement about a claim (App. U); whether the declared requirement is violated after the gate is in Table 43.

N.2 Typed claim identity on the same target (E124)

The result first. **At the reading point the other arms use, typed claim identity detected nothing the numeric comparison did not; what it added is the kind of each alarm, and a swap it saw only before the build.** All 17 of the 17 locked predictions were confirmed. This is one compendium, 12 edits and 270 registered claims, all chosen by us after reading the tree. It is a mechanism table, not a rate, and this subsection prints counts, never percentages.

Why this study exists. A reviewer asked for typed claim identity: register each claim with an identifier, its evidence location, its unit, its formatting rule and where it is expected to appear, so that two claims sharing a number are told apart and a changed value is told apart from a changed printed form. S1 of E122 shows why it is needed: a swap inside one table keeps every literal's count, so no count-based arm can see it. E124 builds the arm, **G3**, and runs it on E122's target, commit, environment and gate command, under a protocol committed alone and first. The arm labels G0–G3 of E121, E122 and E124 are not the operator ids G1–G7 of Table 13. E124 re-scores nothing of E121 or E122; where it prints one of their numbers, it prints it as locked.

The registry and its setup. The registry holds 270 claims. 269 are generated by E121's own extractor from the 27 claim files: 17 constants and 252 table cells. The remaining one is the README's compute estimate, the one prose claim; it was registered by hand, by us, after reading this repository's prose, and two of the 12 edits touch it (I2 and B2b). Each claim carries an identifier (the file for a constant; file, row label and column for a table cell), a locator, its value in canonical form, a unit that is recorded and never compared, and the token as printed. The evidence is not parsed: it is one digest over the 26 committed files under 2 evidence directories. A maintainer writes 29 lines of configuration, the 27 claim paths and the 2 directories, and 1 hand-written entry; the builder generates the rest. The value lock needs 27 lines and G2 0, as E122 declared.

What G3 reports. Every alarm is one string whose form carries its alarm kind, and any alarm fails the check. *Drift*: the value at the claim's own location differs from the registered value. *Moved*: a prose claim's anchors no longer match in its own file and the registered span occurs exactly once in another scanned file. *Missing*: the file, row, column, number or anchor is gone, or the anchor is no longer unique. *Evidence changed*: the digest differs. *Registry missing*: the registry file is absent, a declared alarm and never an exception. A changed printed form with an equal value is a note, not an alarm. Each reading then gets one kind, by the protocol's rule: *loud* if the registry is missing; else *update* if the evidence changed; else *drift* if any claim drifted or went missing; else *moved* if a claim only moved; else *none*.

Edits, runs and the two readings. The 12 edits are D, PC, C1, B1, B2 and B4, copied from E121's file; S1, S2, I1 and I2 from E122's; and two new ones. **B2b** cuts the README's compute-estimate sentence and the sentence after it, and appends both to `scripts/README.md`, a file the pin already scans. **L3** deletes the typed registry from the run's copy of the frozen baselines and leaves the tree alone. With a clean pair per edit, 4 clean runs and 4 policy runs there are 32 runs of 9 scored cells each (G0, G1a, the value lock, both frozen repairs and G2 at reading B; G3 at reading A, at reading B and combined), 288 in all, and none is unresolved; the reading-A results of the pin, G2 and the directory lock are recorded beside them. *Reading A* is taken after the edit is committed and before the gate's `make output`; *reading B* after the gate command, where E122 read. Arms are compared only at the same reading. A check run before the build is blind to every drift the build introduces, including D and PC, so Table 30 prints both columns. On every run E124 repeats from E122, E124's reading-B labels for E122's arms equal E122's locked labels, so the comparison with E122 is cell for cell.

Per arm, at reading B. Over the 6 edits a miss is scored on (D, PC, S1, S2, I1 and I2), G2 caught 4 and missed 2, S1 and I1, both restored by the build. G3 caught 4 and missed 2, the same two. The pin caught 3 and missed 3, S1, I1 and I2. A fail on any other edit is counted apart, never as a catch. Nuisance alarms, on B1 and B2: G2 raised 1 (B2), the pin 2 (B1 and B2) and G3 0. Relocation alarms, on B2b: G2 raised 1 and G3 1, a maintenance cost for both. On the evidence update B4, G2, G3 and the pin all failed, so each needs its baseline rebuilt, and all passed after the policy rebuild. On the deleted registry L3, only G3 failed. Over the 11 frozen runs other than L3, G3 failed on 0 runs where G2 passed, and G2 failed without G3 on 1, B2. That is the locked Y17, and the sentence at the head of this subsection is its mandatory wording; §4.4 carries the same sentence word for word, as the protocol requires.

What G3 added: the swap. A swap of two values inside one regenerated table (S1) was seen by typed claim identity and by no count-based check, but only at a reading taken before the project's own make ran; at reading B only the value lock saw it. The licence for Y4 says that at that reading nothing saw it; the locked Y4 itself records the value lock failing there, and so does Table 30, so

	reading A, before the `make`				reading B, after it					
edit	R_pin	G2	G3	G3 kind	G1	R_pin	G2	G3	G3 kind	post-gate state
`D`	pass	pass	pass	none	**fail**	**fail**	**fail**	**fail**	drift	edit persists
`PC`	pass	pass	pass	none	**fail**	**fail**	**fail**	**fail**	drift	edit persists
`C1`	pass	pass	pass	none	pass	pass	pass	pass	none	edit persists
`B1`	pass	pass	pass	none	**fail**	**fail**	pass	pass	none	edit persists
`B2`	**fail**	**fail**	pass	none	pass	**fail**	**fail**	pass	none	edit persists
`B2b`	pass	**fail**	**fail**	moved	pass	pass	**fail**	**fail**	moved	edit persists
`B4`	pass	pass	**fail**	update	**fail**	**fail**	**fail**	**fail**	update	edit persists
`S1`	pass	pass	**fail**	drift	**fail**	pass	pass	pass	none	restored
`S2`	**fail**	**fail**	**fail**	drift	pass	**fail**	**fail**	**fail**	drift	edit persists
`I1`	pass	**fail**	**fail**	drift	**fail**	pass	pass	pass	none	restored
`I2`	pass	**fail**	**fail**	drift	pass	pass	**fail**	**fail**	drift	edit persists
`L3`	pass	pass	**fail**	loud	pass	pass	pass	**fail**	loud	—

Table 30: **E124: every frozen edit at both readings.** Reading A is taken before the gate's own `make output`; reading B after it, where the labels of E122's arms are read, as in E122. G1 is the value lock and R_pin the mention pin. G0 fails only on C1, G1a fails on every clean run and the directory lock passes every run at both readings, so they are not shown. "G3 kind" is the kind G3 gives that reading: *drift*, *moved*, *update* (the evidence digest changed), *loud* (the registry is gone) or *none*. A check at reading A is blind to every drift the build introduces, which is why D and PC pass there. Post-gate state: *edit persists* means the edited bytes are still in the tree after the gate, which is not a statement about a claim, *restored* that the build rewrote each edited file to its unedited bytes before reading B and every claim value read then equals the unedited tree's, while the candidate commit still holds the edit, and — that the tree was not edited. Whether the declared requirement is violated after the gate is in Table 43. The clean, pair and policy runs pass every arm shown here; the supplement lists all 32 runs. Arm labels G0–G3 are not the operator ids of Table 13.

we write "only the value lock". The pin, G2 and the directory lock made 0 catches on S1 at either reading. The value lock fails at reading B because the build rewrote the swapped table, so the tree no longer matches the committed edit. Of the 2 edits the build restored, S1 and I1, G3 caught 2 at reading A, before the build restored them, and 0 at reading B, where only the value lock saw them; at reading A G2 caught 1 and the pin 0. Typed identity saw both integers, I1 and I2, and on the abstract's own fragment (I1) it did so only at reading A, before the build restored the value; at reading B only the value lock saw I1. G2 saw I1 at reading A too. So of the 6 edits a miss is scored on, S1 is the only one G3 alone saw at reading A.

What G3 added: the kind of each alarm. On S2, a swap across two fragments the build does not remake, both per-claim checks (G2 and G3) caught it and only the typed one named which claim was wrong; that is not a detection difference. When the registered prose claim moved to another file (B2b), typed identity named the claim and both files, while the numeric comparison returned the alarm it returns when that claim changes value: G2's alarm on B2b is the same string as its alarm on I2, while G3's alarm on B2b is of kind *moved* and its alarm on I2 of kind *drift*. The relocation is still an alarm, a maintenance cost. After the policy rebuild the registry holds the claim in its new file with the same value, and G3 passes. On B2, which moves the README's citation block, a BibTeX entry, typed identity raised no alarm where the numeric comparison and the pin did: a DOI is an identifier, not a registered claim. On B1 a change of printed form with the value unchanged was reported as a note, not a failure; this means a precision drift from 0.30 to 0.3 does not fail the check. On B4 typed identity distinguished a numerical change accompanied by an evidence-file change from one without, which no count-based arm here does: the reading's kind is *update* at both readings, never *drift*. The separation rests on a digest of 26 committed files, not on which claim each file supports, so it shows that evidence files changed, not that the update is legitimate; and because *update* is given ahead of *drift*, the reading's kind says *update* even while drift alarms sit in the same list. The protocol's licence calls this separating a legitimate update from a drift; we use the narrower wording. Typed identity classified the consequential defect D as a drift with its evidence unchanged (118 drift alarms), and saw nothing of it before the build. Deleting the typed registry (L3) is not silent: it is a declared alarm, of kind *loud* at both readings, while G2 and both frozen repairs passed. G3 passed all 4 policy runs.

What it costs. Setup: 29 lines and 1 hand entry, where G2 needs none. Maintenance: on B4, G3's reading-B list holds one evidence alarm, 35 drift alarms and 14 *missing* ones; the evidence alarm makes the reading's kind *update*. The missing ones are every cell of the *Standard* and *Total* rows of the GitHub standard-and-messy accuracy table: each row label prints its file count, the update removed one file, and a claim keyed by its row label no longer found its row. That is a maintenance cost of this locator, which the protocol requires us to report. The digest also cannot say which of the drift alarms the update explains.

What this cannot show. Every claim on this compendium has exactly one occurrence, so the case the schema exists for — one number printed in an abstract, a table and a sentence — could not be exercised here, because the paper's source is not in the repository. The predictions were written with E121's and E122's artifacts in the tree, so on the runs E124 repeats from E122, G3's reading was close to determined by facts already on disk; the cases G3 met for the first time are B2, B2b and L3. Reading A is not a point any shipped CI reads; it is where a pre-commit check would read. The one prose claim was registered by hand, by us, and two of the 12 edits touch it. The deleted-registry case tests G3's loader as written here. Runtimes are recorded and no claim is made about them.

N.3 Typed identity's diagnoses when conditions coexist (E125)

The result first. **E124's classification let an evidence change hide a coexisting drift, and reporting the flags simultaneously keeps it visible. Neither version decides whether an evidence change is legitimate. The variant changes the diagnosis, not the detection.** All 8 of the 8 locked predictions were confirmed, the two that could have refuted the variant (Z2 and Z3) among them. One compendium plus one synthetic fixture, all edits chosen by us; counts, never rates. It is a mechanism table, and this subsection prints counts, never percentages.

Why this study exists. E124's own artifact exposed three weaknesses in G3's diagnosis. Its kind is ordered (*loud*, then *update*, then *drift*, then *moved*), so an evidence change makes the reading *update* even when a claim drifted with it; the drift strings stay in the alarm list, and the kind does not say so. On B4 a set of table claims went missing because their row labels print the file count B4 changed. And every claim has one occurrence. E125 asks whether the diagnoses stay right when conditions coexist, under a protocol committed alone and first, on E121's target, commit, environment and gate command; it re-scores nothing of E121, E122 or E124.

The variant, G3m. G3m reads the registry G3 reads (E124's, rebuilt to the same bytes, and on B4P-D the one its policy step rebuilt), with G3's declared knowledge, at G3's two readings, and differs from G3 in two ways only. *Simultaneous flags*: each reading emits the set of flags that hold, from *evidence changed*, *claim drifted*, *claim missing*, *claim moved* and *registry missing*; no flag suppresses another, there is no kind, and the evidence alarm reads "evidence files changed; update requires review", never an authorised update. *Stable row keys*: a table row is keyed by its label with every parenthesised count removed, and the printed label is kept beside it; a key that then matches more than one row is ambiguous and fails, and nothing is guessed. G3 runs beside G3m, unchanged, on every run, so every difference is attributable to one of the two changes.

Edits, runs and the two readings. The 5 edits come from 4 edit records: D and B4 alone, from E121's file; **DB4**, D then B4, with no policy step; **U1**, one byte appended to a committed evidence file that no recipe of the build reads, then E122's I2, the README's integer; and **B4P-D**, B4 with E124's policy step (every reference rebuilt), then D. The design asked for U1's byte to go into a detection file; every detection file is read by a planned recipe, so the protocol's selection rule took the first evidence file that qualifies, a preprocessing list, and says so. With 4 clean runs and a clean pair per edit there are 14 runs of 12 scored cells each (E124's nine, and G3m before the build, after it and combined), 168 in all, none unresolved. Reading A is taken before the gate's `make output` and reading B after it, as in E124. On the 8 run ids E125 shares with E124, all 72 of E124's cell labels and all 4 of its G3 alarm lists came back identical, so the comparison with E124 is cell for cell.

The override, measured (Z2, Z3). On DB4 after the build, G3's kind is *update* and its list holds 138 alarms, D's drifts among them. G3m's list has as many, with both flags: *evidence*

	reading A, before the `make`				reading B, after it					
edit	G3	G3 kind	G3m	G3m flags	G3	G3 kind	alarms	G3m	G3m flags	alarms
`D`	pass	none	pass	none	**fail**	drift	118	**fail**	drift	118
`B4`	**fail**	update	**fail**	evidence	**fail**	update	50	**fail**	evidence, drift	42
`DB4`	**fail**	update	**fail**	evidence	**fail**	update	138	**fail**	evidence, drift	138
`U1`	**fail**	update	**fail**	evidence, drift	**fail**	update	2	**fail**	evidence, drift	2
`B4P-D`	pass	none	pass	none	**fail**	drift	118	**fail**	drift	118

Table 31: **E125: G3 beside G3m on every frozen edit, at both readings.** "G3 kind" is the one kind G3 gives the reading (App. N.2); "G3m flags" is the set of flags that hold, where *evidence* is *evidence changed*, *drift claim drifted* and *none* the empty set. "Alarms" counts each arm's alarm strings at reading B. Every clean run and every clean pair is silent for both arms at both readings. One compendium, all edits chosen by us; counts, never rates.

changed (11 of 26 committed evidence files differ) and *claim drifted*, naming D's drifts of `AccuracyOverallOurs` and `ImprovementOverSnifferMessyCeil`. On U1, before and after the build, each list holds exactly the README drift and one evidence alarm (1 of 26 files); G3's kind is *update* at both readings, and G3m's flags are *evidence changed* and *claim drifted*. E124's classification let an evidence change hide a coexisting drift, and reporting the flags simultaneously keeps it visible. Neither version decides whether an evidence change is legitimate. What hid the drift was the one-word kind: the drift strings were in G3's list all along. G3m keeps G3's evidence check, a digest over 26 committed files. It says that the evidence moved and that claims drifted, not which claim a file supports: after the build its flags on B4 alone and on DB4 are the same, so it cannot tell D's drifts from the drifts B4 itself causes.

The other predictions. *Refreshed reference (Z4).* On B4P-D after the build G3 and G3m both fail, and G3m's only flag is *claim drifted*, with 118 drift alarms: the evidence matches the refreshed registry. The defect stayed visible after the reference was refreshed for an authorised update. *Stable row keys (Z5).* On B4 after the build G3's list holds 50 alarms, 35 drifts, 14 missing claims and the evidence alarm; G3m's holds 42, 41 drifts and the evidence alarm, with 0 missing and 0 ambiguous keys. Of the 14 cells G3 lost, 6 drifted and 8 kept their value. The 14 missing claims on B4 were an artefact of row labels that carry a count, and a stable key removed them without an ambiguous match on this target. On DB4 the same key took G3's 14 missing claims to 0. *One condition alone (Z6).* On D after the build G3m's only flag is *claim drifted* (118 alarms), and on B4 before the build its only flag is *evidence changed*. With one condition alone, G3m reported that condition alone. *Clean runs (Z1).* All 4 clean runs are silent for both arms at both readings, with no format note. *Detection (Z8).* Over the 14 runs at both readings, G3 and G3m failed together on 8 of the 28 readings and passed together on the other 20; each failed on 5 of the 5 frozen edits, the authorised update B4 among them, with every clean pair passing. The variant changes the diagnosis, not the detection.

The synthetic fixture (Z7). A generated compendium of 5 files written for this study and never counted as a repository: one claim printed in an abstract sentence, a table cell and a figure caption, a second claim with another value in the abstract and the table (2 claims, 5 occurrences), a prose file with no registered number, and one evidence file both values are computed from. Its 7 cases, each on a fresh copy: (a) one occurrence drifts; (b) one occurrence is deleted; (c) a second row with the same stable key is added; (d) the two claims' values are swapped everywhere they occur; (e) one occurrence is reformatted to an equal value; (f) the sentence holding one occurrence moves to another file; (g) the evidence and every occurrence of the first claim change together. All 7 were applied at their declared hashes, and in all 7 G3m's alarm list, notes and flags were exactly the protocol's: (a) one drift naming that occurrence; (b) one missing claim naming it; (c) a failure naming the ambiguous key; (d) a drift on every occurrence of both claims; (e) a format note and no flag; (f) a move naming both files; (g) a drift on each changed occurrence and the evidence flag, together. The mechanism behaves as specified on a synthetic compendium with repeated occurrences, and the fixture is synthetic, written by us, and not an independent repository. G3 was not run on it, because E124's registry holds one locator per claim.

What this cannot show. One compendium plus one synthetic fixture, all edits chosen by us after reading the tree; counts, never rates. The predictions were written with E121's, E122's and E124's

artifacts in the tree, so on the clean, D and B4 runs G3's readings were close to determined by facts already on disk, and what E125 adds there is G3m's reading of the same trees; the cases G3m met for the first time are DB4, U1, B4P-D and the fixture. The stable key removes a parenthesised count and nothing else: a label that carries a number in another form is not handled, and two rows that differ only in their counts fail as ambiguous by design. U1's evidence change is one appended byte in a file no recipe reads, unrelated by construction, not found in the wild. The fixture's layout was chosen by a rule that suits the anchors G3 and G3m use. The work tree's pristine twin was hard-linked to the pinned checkout, for lack of disk space, and only ever read. Runtimes are this machine's, and no claim is made about them.

N.4 An ML compendium with no gate of its own (E123)

The second target is SGC, an ICML compendium by unrelated authors at a pinned commit, and the ML target E121's protocol had declared as a fallback. Its claim path is three commands, one per dataset, each of which trains and prints a test accuracy, read against a table in the README, with the declared test split as the population. It ships no test, no CI, no `Makefile` and no committed output. The gate is therefore *reconstructed* as the only check that exists: a documentation-derived rounded-accuracy comparator, which compares the printout with the README at one decimal. The value lock has nothing to lock, so all 42 of its cells are not applicable. Three clean runs print the same value on all 3 claims, and all 10 locked predictions were confirmed.

Table 32 is every cell. The witness has two clauses (App. U). Clause (i) recomputes the accuracy from the model's own predictions over the declared test population and reads no README; clause (ii) compares the printout with the README of the witness's own tree. It fired on 17 cells. The comparator caught 14 of them and missed 3. **Those 3 are a count on one target, never a rate.** They are one edit, D1b, which drops a single node from the declared test population: the printed accuracy moves off the model's true accuracy on every claim and still rounds to the README's printed value, so only a witness clause that recomputes the accuracy sees it. D1, accuracy computed over the validation indices, and D2, a wrong metric, move the printed value far enough that the comparator catches every claim. D3 changes the README number with the evidence untouched, which only clause (ii) can see: the frozen pin caught it, the comparator, which compares the printout with the README, caught it too on one dataset, and the numeric comparison, whose registry lives outside the document, did not. The pin fired on 3 runs with 1 nuisance alarm in two benign edits, and was silent on 5 of the 7 runs a witness fired on, every one of them a code edit it cannot read. The numeric comparison missed 5 of the 17 witnessed cells. The protocol's clause for a reconstructed gate that catches everything does not apply here: it missed, and the misses are reported.

One fact bounds what any repair of ours could have done. The target writes nothing to disk: after a run the tree is byte-identical, so nothing on disk records the wrong number and the only trace is the terminal. Both frozen repairs were built for prose drift and shipped data, and this pipeline has neither on its claim path; on this target they protect the README and nothing else.

N.5 Which checks can see a population requirement (E126)

The result first. On the graph-network compendium, an edit that evaluates every test node twice leaves every printed accuracy unchanged, so the README comparison and a full recomputation both accept it. A check that compares the loader's test id list with the declared ids, keeping counts, rejects it. It reads the loader's list, not the rows passed to the evaluation, so it accepts an edit made at the evaluation call. 41 of the 41 locked lines held, 0 failed and 0 were unresolved.

Why this study exists. E123's recomputation caught every population edit it saw, but only because each edit moved the accuracy. Evaluate every declared test node twice, and the number of correct predictions and the number of evaluated nodes both double, so the accuracy does not move. A check that uses only the unchanged accuracy cannot tell these runs apart; a check that reads the number of evaluated rows could. E126 asks which check can, and what that check in turn cannot see. Its protocol, with every prediction, was committed alone before any code or run.

Execution level. Every case retrains the model from the README command, on the pinned compendium, in E123's clone and environment on our machine. This is fresh target execution, not a

run	claim	printed	W	G0	G1	G2	R
clean-01	K_cora	0.8100	NO_WITNESS	PASS	n/a	PASS	–
clean-01	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	–
clean-01	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	–
clean-02	K_cora	0.8100	NO_WITNESS	PASS	n/a	PASS	–
clean-02	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	–
clean-02	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	–
clean-03	K_cora	0.8100	NO_WITNESS	PASS	n/a	PASS	–
clean-03	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	–
clean-03	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	–
D1	K_cora	0.7940	WITNESSED	FAIL	n/a	FAIL	–
D1	K_citeseer	0.7280	WITNESSED	FAIL	n/a	FAIL	–
D1	K_pubmed	0.8180	WITNESSED	FAIL	n/a	FAIL	–
D1b	K_cora	0.8098	WITNESSED	PASS	n/a	PASS	–
D1b	K_citeseer	0.7187	WITNESSED	PASS	n/a	PASS	–
D1b	K_pubmed	0.7888	WITNESSED	PASS	n/a	PASS	–
D2	K_cora	0.1480	WITNESSED	FAIL	n/a	FAIL	–
D2	K_citeseer	0.1260	WITNESSED	FAIL	n/a	FAIL	–
D2	K_pubmed	0.1850	WITNESSED	FAIL	n/a	FAIL	–
D3	K_cora	0.8100	WITNESSED	FAIL	n/a	PASS	fires
D3	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	fires
D3	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	fires
PC	K_cora	0.4050	WITNESSED	FAIL	n/a	FAIL	–
PC	K_citeseer	0.3595	WITNESSED	FAIL	n/a	FAIL	–
PC	K_pubmed	0.3945	WITNESSED	FAIL	n/a	FAIL	–
SC	K_cora	0.1130	WITNESSED	FAIL	n/a	FAIL	–
SC	K_citeseer	0.1820	WITNESSED	FAIL	n/a	FAIL	–
SC	K_pubmed	0.1800	WITNESSED	FAIL	n/a	FAIL	–
B1	K_cora	0.8100	NO_WITNESS	PASS	n/a	PASS	fires
B1	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	fires
B1	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	fires
B3	K_cora	0.8100	NO_WITNESS	PASS	n/a	PASS	–
B3	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	–
B3	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	–
B1-policy	K_cora	0.8100	NO_WITNESS	PASS	n/a	PASS	–
B1-policy	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	–
B1-policy	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	–
D3-on-B1	K_cora	0.8100	WITNESSED	FAIL	n/a	PASS	fires
D3-on-B1	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	fires
D3-on-B1	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	fires
clean-04	K_cora	0.8100	NO_WITNESS	PASS	n/a	PASS	–
clean-04	K_citeseer	0.7190	NO_WITNESS	PASS	n/a	PASS	–
clean-04	K_pubmed	0.7890	NO_WITNESS	PASS	n/a	PASS	–

Table 32: **E123: every run by claim.** "printed" is the accuracy the command printed; W is the gate-independent witness; G0 is the reconstructed documentation-derived rounded-accuracy comparator; G1 the value lock, which has nothing to lock; G2 the registered-claim check, which compares the printout with values written in the protocol and never reads the README (not E122's numeric comparison); R the frozen mention pin, reported per run rather than per claim. D1b is the one edit the comparator misses on every claim.

replay of saved predictions. It is a controlled diagnostic with edits we chose: not a naturally occurring defect, not a new repository and not an independent reproduction.

Three requirements. Each was declared before any run, with evidence that does not depend on any check's verdict. *Population*: each declared test node, read from the dataset's test index file and never from the loader, is evaluated exactly once. The benchmark permits no weights, so a repeated node is a violation. Its evidence is a record of the rows actually passed to the evaluation function, each named by a digest of its propagated features and its label. *Metric*: the printed accuracy equals the trained model's accuracy over the declared population. Its evidence is E123's recomputation. *Reported value*: the README value equals the printed value at the README's precision. Its evidence is the README number against the printout.

Four checks. The *README comparison* (K1) is E123's reconstructed comparator, run on a tree without the hook; it reads no id. The *full recomputation* (K2) recomputes the accuracy over the declared population from the model's outputs; it reads the declared ids, not the evaluated ones. The *loader-id check* (K3) compares the test id list the loader returns with the declared ids, and it keeps counts. K4 accepts only when K2 and K3 both accept; it reads no README. K1 is the reported-value evidence run as a check, and K2 is the metric evidence run as a check, so each agrees with

case	printed (Cora / Citeseer / Pubmed)	rejects				violated		
		README comparison (K1)	full recomputation (K2)	loader ids (K3)	K2 and K3 (K4)	population	metric	reported value
clean, 3 runs	0.8100 / 0.7190 / 0.7890	0/9	0/9	0/9	0/9	0/9	0/9	0/9
`D1b`: drop the last test node	0.8098 / 0.7187 / 0.7888	**0/3**	3/3	3/3	3/3	3/3	3/3	0/3
`DUP`: every test node twice	0.8100 / 0.7190 / 0.7890	**0/3**	**0/3**	3/3	3/3	3/3	0/3	0/3
`D1`: validation set at the call site	0.7940 / 0.7280 / 0.8180	3/3	3/3	**0/3**	3/3	3/3	3/3	3/3
`D2`: max values for argmax	0.1480 / 0.1260 / 0.1850	3/3	3/3	**0/3**	3/3	0/3	3/3	3/3
`B1`: README number reformatted	0.8100 / 0.7190 / 0.7890	0/3	0/3	0/3	0/3	0/3	0/3	0/3

Table 33: **E126: which check sees which requirement, on the graph-network compendium.** Every count is over the 3 claims; the clean row is 3 runs, so 9 claim cells. "Rejects" counts the claim cells a check rejected. "Violated" counts the claim cells in which the requirement's evidence shows a violation. A bold count marks a check that accepts a claim cell in which some requirement is violated. K1 agrees with the reported value and K2 with the metric by construction. The printed values are the same with the hook and without it.

its own requirement by construction. The informative cells are a check's verdict on a requirement it does not read. K3 and the population evidence read different objects: K3 reads the loader's list, and the evidence reads what reached the evaluation call.

Cases. 3 clean runs and 5 edits (Table 33). D1b, D1, D2 and B1 reuse E123's edit records, copied byte for byte, and are run afresh here. DUP is new and sits at D1b's site: the loader returns its test id list twice. Every run executes each of the 3 README commands twice, once with a hook that records the ids and the rows and once without it, 48 commands in all. All 24 run-by-claim cells printed the same value both ways, so the hook changed no printed value.

What the id records show. On DUP the loader's list has 2000 entries: each of the 1000 declared ids twice. Its set of distinct ids equals the declared set on all 3 claims. A comparison of id sets would accept the doubled edit; only a comparison that keeps counts rejects it. On D1 the loader's list is untouched, so the loader-id check accepts, while the evaluation call received the 500 validation rows. On D1b the list has 999 ids, the declared ones without the largest. The table has 15 claim cells in which a check accepts while a requirement is violated, exactly the set the protocol locked. K4 accepts in 0 of them. Checking the loader's id list and the metric together rejected every edit here that broke either requirement, and accepted the clean runs and the reformatted README. It rejected the call-site edit D1 only through the recomputation: neither part reads the rows that reach the evaluation call, so a call-site edit that keeps the accuracy would pass it; it reads no README, so it says nothing about the reported value.

The locked lines. 6 lines fix the printed value in each case, 24 fix each check's verdict on each case, 6 fix each case's requirement status, 4 describe the id records, and 1 fixes the set of bold cells. 41 of the 41 held. Post hoc, we narrowed the wording of two of the protocol's licensed sentences, the first and the one on K4: K3 reads the loader's list, not the rows that reach the evaluation call, and the licensed wording named it by the evaluated ids. Every locked line under the narrowed sentences held; the protocol's own words are in its file.

What this cannot show. One repository, 3 claims of one kind and 5 edits we chose after reading the code: a mechanism with worked cases, never a rate. The loader-id check is one population check with one observation point; a check that read the call site would see D1. K4 reads no README, so a README-only edit such as E123's D3 would pass it, and no such edit is in this set. The call-site record names rows by content, so test nodes with identical propagated features and label share a name: 77 on Cora, 132 on Citeseer and 0 on Pubmed. Every violation here changes the number of evaluated rows, so no locked status depends on telling such nodes apart. The environment is ours, as in E123.

O AN ML COMPENDIUM WITH ITS OWN CHECK (E128)

On the agent-failure compendium, the authors' claim verifier rejected the analysis-code defect, the input-data edit and the label edit. In this audit, it accepted 4 of the 7 claim faults: an edit to the failure rates under an unchanged label, an edit to a README count, and two edits to the verifier itself or to its expected values. Adding the frozen report comparison left 2 of them, and adding the numeric comparison 3. We chose the seven faults after reading the verifier, one per layer of the claim path; which ones escape follows from that choice, not from a sample. Two of the four that the authors' check accepted (WF1, F6) edit the verifier or its expected values. Of the locked prediction lines, 0 were refuted and 0 unresolved; the counts by group are under "The locked lines" below. This is one ML compendium: a case, not a rate. The repair study of App. P uses its own 7 claim faults on the same compendium and shares one of them with this audit (its B1 is F3b here), so the two sets are not one test set.

Why this study exists. The graph-network compendium (App. N.4) ships no check, so every comparator we ran there is ours. E128 takes an ML compendium whose authors ship a verification command, so that the check under audit is theirs. It asks which claim requirements that command protects, what three added checks add or lose, and which permitted maintenance edits each check blocks, always naming the artifact.

The selection rule and the survey. The rule was committed alone before any search query. A candidate had to meet six requirements: an ML paper whose repository is its research compendium; no author or repository owner in common with this paper or with any earlier GateCheck target; a permissive licence; a CPU-only path from committed code and data to a number the paper prints; a verification command the authors shipped that runs that path; and no overlap with the populations of the earlier studies. The requirement on this paper's authors was checked at selection and confirmed by the authors after the run (a dated addendum beside the protocol). Four kinds of search were fixed in advance: notebook and test checks in CI, build systems that regenerate paper numbers, reproducibility venues, and metric-checking CI. The searchers logged 159 queries, examined 323 repositories and screened 440 more by name only. Of the examined repositories, 252 were ML compendia; 90 of those ship a check of their own. Of these checks, 23 run the path to a printed number, 7 compare a produced number with a reference (3 on that path), and 4 compare a number the paper prints (3 on that path). 2 repositories met all six requirements on paper, and a frozen order put them in sequence before any smoke. The first passed its feasibility smoke on the clean checkout, so 1 smoke was run; the second, by the same author, was not tried until the transfer study (App. Q). No fault was run during selection. These searches look for repositories with checks, so these counts lean toward checks and are not an estimate of how common checks are among ML compendia.

The target, and the execution level. The agent-failure compendium is `DaoyuanLi2816/delta-mfp-local-agents` at commit `c01fbafc` (81 tracked files, MIT licence), the code of Li (2026a), a non-archival workshop paper. Its claims are counts over committed per-trace results of local language-model runs, such as the regime label of each failed trace and the failure rate at each replayed prefix; the paper's decision rule turns the rates into the label. Every run here is fresh target execution of the pinned repository on our machine (macOS, CPython 3.12.13), in a copy that is reset after each run. It is a controlled diagnostic with edits we chose after reading the code: not an independent reproduction of the paper and not a re-run of its experiments, which need a GPU. The model observations are the authors' committed ones.

The checks and their contracts. The authors' check (K0) is the three steps their CI runs after installing, stopped at the first failure: `pytest` (13 tests, one of which runs the claim verifier), the claim verifier, which compares 22 computed values with the literal `expected` dictionary in its own source, and a release scan for secrets and private paths. The value lock (K1) is E122's, run unchanged: `git diff --exit-code` over the committed records, the README and a results page, after the command. The frozen report comparison (K2) reads the verifier's report as JSON and compares its 33 numeric values, as decimals, with a copy of the pristine report frozen before any edit; it keeps the reference, stage and artifact the numeric comparison has on the CSV compendium, not its observation or acceptance rule. The numeric comparison (K3) is E122's, run unchanged against a frozen count of the numbers in the prose files; the command generates no prose here, so

	check verdict				requirement status (candidate commit / generated output)			
edit	K0	K1	K2	K3	stated value	computation	decision rule	README
clean (16 runs)	acc. 16	n/a	acc. 16	acc. 16	S/S	S/S	S/S	S/S
C1 (focal: computation)	rej.	n/a	error	acc.	S/U	**V/V**	S/U	S/S
F1 (focal: computation)	rej.	n/a	rej.	acc.	S/V	**V/V**	S/V	S/S
F2 (focal: stated value)	rej.	n/a	rej.	acc.	**V/V**	S/S	S/S	S/S
F3a (focal: stated value)	rej.	n/a	rej.	acc.	**V/V**	S/S	V/V	S/S
F3b (focal: decision rule)	acc.	n/a	acc.	acc.	S/S	S/S	**V/V**	S/S
F4 (focal: README)	acc.	n/a	acc.	rej.	S/S	S/S	S/S	**V/V**
F5	acc.	n/a	acc.	acc.	S/S	S/S	S/S	S/S
WF1 (focal: computation)	acc.	n/a	rej.	acc.	S/V	**V/V**	S/V	S/S
F6 (focal: stated value)	acc.	n/a	rej.	acc.	**V/V**	S/S	V/V	S/S
M1	rej.	n/a	acc.	acc.	S/S	S/S	S/S	S/S
M2	acc.	n/a	acc.	acc.	S/S	S/S	S/S	S/S
M3	acc.	n/a	acc.	acc.	S/S	S/S	S/S	S/S

Table 34: **E128: every check on every edit, and every requirement in both artifacts.** K0 is the authors' check, K1 the value lock, K2 the frozen report comparison, K3 the numeric comparison. S: satisfied, V: violated, U: unresolved (the crash control leaves no report). Bold: the edit's focal requirement, on which EscapedViolation is scored. The clean row covers every clean run; its statuses are the witness's two pristine runs. K1 accepted every run, but it is printed n/a: the command writes no tracked file, so each pass compared the candidate commit with itself (its raw verdict is kept in the current outcome layer, App. U).

K3 reads prose equal to the candidate commit's. A workflow is K0 alone or K0 with one added check, and it accepts only if each of its checks accepts; an error is never an accept.

Requirements and witness. Each requirement is stated for the candidate commit and for the generated output, which here is the verifier's report (the command writes no tracked file). *Stated value*: each of the 33 values equals its declared value, the rates read as numbers. *Computation*: the candidate's claim code, run over the pinned records, prints every declared value. *Decision rule*: each of the 17 regime-derived values equals what the paper's decision rule gives over the committed failure rates; on the pinned records our implementation reproduces all 239 committed labels. *README*: 10 rows of the README and the results page restate the declared counts. The witness runs in its own trees and never reads a check's verdict or log. Our own code recomputes the stated values, the rule labels and the README rows; for the generated output and the computation requirement the witness runs the candidate's verifier, as K0 and K2 do, so K2's verdict matches the stated-value status of the generated output by construction.

The edits. C1 points the verifier at a directory that does not exist, a crash control. The seven claim faults: F1 makes the survivor count read one field twice; F2 changes one task's difficulty; F3a relabels one natural failure; F3b changes that failure's rates so that the decision rule gives prefix-0, and keeps its label; F4 changes one README count; WF1 is F1 with the verifier's comparison weakened to "the claim was reported"; and F6 relabels the failure and changes its expected count to match. F5 is the weakened comparison alone, which violates nothing. The maintenance edits: M1 rewrites one repair rate from 1.000 to 1.0; M2 renames a result file and every reference to it; M3 rewrites a sum of booleans as the length of a list. Each edit ran once with its clean pair immediately before it; with three clean runs at the start and one at the end there are 16 clean runs and 12 edit runs.

How the checks caught what they caught. K0 stopped at its first step, the tests, on every edit it rejected. The crash control failed a test by a missing file; F1, F2, F3a and M1 failed by assertions that name the value that changed. K2 was an error on the crash control, where the report step printed nothing. K3 fired only on the README edit, after which the original count no longer appeared in the README. K1 fired on no run: across all 28 runs the command left no tracked path changed. Of the 112 check cells, 100 accepted, 11 rejected and 1 was an error; none was unresolved.

Trust settings and applicability (post hoc). For Table 3, a rule that reads only the edit sites puts an edit in the *co-edit* setting when one of its sites edits the verifier's comparison or its expected values, and in the *fixed-verifier* setting otherwise. F1, F2, F3a, F3b and F4 are fixed-verifier faults; WF1 and F6 are co-edit faults, and F5 is the co-edit control. The same current outcome layer gives

every added check an applicability flag per run: a value lock is applicable only on a run where the command wrote a locked file. K1 was not applicable on 28 of its 28 runs, that is, on none; on the CSV compendium the value lock was not applicable only on the crash control (3 of 36 runs). No locked line is re-scored; a dated addendum beside the protocol records the change of presentation.

The locked lines. The protocol was committed alone, after the target was chosen and before any edit was applied. It locked 150 prediction lines, which the scorer reads as 1107 clauses: 13 on requirement status (13 held), 20 on check mechanisms (20 held), 26 per check on verdicts and EscapedViolation (held: 26, 26, 26 and 26), 3 on nuisance alarms (3 held), 3 on the witness (3 held) and 7 on totals (7 held). 0 lines were refuted and 0 unresolved; every validity gate passed. This is a confirmation, not a discovery. We derived every edit's outcome from the code and records before the protocol was committed; the run confirms that derivation and our harness on the real repository.

What this does not show. One compendium and edits we chose: a mechanism table, not a catch rate, not a false-alarm rate and not a statement about ML compendia. The claims are counts over committed per-trace results, and nothing re-runs the models; the decision-rule requirement asks whether each label follows from the committed rates, not whether the rates would recur in a fresh replay, and a defect in the replay code upstream cannot be witnessed without a GPU. The value lock compared nothing here because this project commits no output its command rewrites; that is a property of this compendium, not of value locks. Only the numeric comparison reads the README, and it reads counts of numbers: it would pass two numbers swapped within one file. K0 is the three CI command strings as frozen; a weakening inside the CI file itself is outside this study. The authors' CI ran on Linux across three Python versions in the checkout itself; we ran on macOS with one interpreter, in a copy.

P A CHECK BUILT FROM THE DIAGNOSIS (E130)

E130 turns the diagnosis of App. O into a check and tests it on the same compendium. The authors' verifier counts the committed regime labels and compares the counts with literals; it never checks that a label follows from the evidence stored beside it. The contract names what is missing: a requirement (each stored label equals what the paper's rule gives on its stored evidence), a reference outside the commit (the rule, as the paper prints it) and an observation (each record's evidence and label). The protocol was committed alone before any edit was applied; every edit was chosen after reading the code. This is a confirmation of our reading, not a discovery. E130's 7 claim faults are its own set, not E128's: the two share one fault (B1 is E128's F3b). So the re-derivation check's 6 of 7 here and E128's counts are on different sets, not a before and after on one set.

The checks. K0 is the authors' three CI steps (App. O); K2 the frozen report comparison and K3 the numeric comparison, both unchanged from E128. K4, the re-derivation check, walks the 6 files that store a label beside its evidence and applies the paper's decision rule to each of the 239 stored labels: a trace is nontrivial when some later prefix fails at a rate of at least 0.6 and at least 0.3 above the rate from the start; prefix-0 when the rate from the start is at least 0.6 and no later prefix gives that increase; unstable when no prefix reaches 0.6; no-Δ otherwise. It compares decimal strings as exact fractions and lists every label that differs, with its file, line, stored label and rule label. K5, the frozen evidence lock, is the simplest competitor a reader would propose: it rejects when any of the five evidence files differs in bytes from its pristine hash. K1, the value lock, was not run. The value lock had nothing to observe on this compendium: the command writes no tracked file. We report it as not applicable, not as a pass.

Validation, before any edit. Before any edit, a second implementation that we wrote from the paper's text, sharing no code with the check and using exact fractions, and 32 hand-worked boundary cases agreed with the check on every case and on all 239 committed labels. The same team wrote both implementations. The check's rule function had already been run on 24 of the cases before they were locked. One case sits where binary floating point and exact arithmetic disagree ($p_0 = 0.4$, $p_k = 0.7$); no committed record sits there. The witness runs that second implementation in its own tree and never reads a verdict. On every edit run, the (file, line, field, stored label, rule label) set in the check's list equals the set the witness gives, so the witness graded the check's printed labels, not only its verdicts.

		check verdict					requirement status (candidate commit)		
edit	setting	K0	K2	K3	K4	K5	label	rule counts	stated value
clean (15 runs)		acc. 15	acc. 15	acc. 15	acc. 15	acc. 15	S	S	S
C1	control	rej.	error	acc.	acc.	acc.	S	S	S
B1	fixed	acc.	acc.	acc.	rej.	rej.	**V**	V	S
B2	fixed	acc.	acc.	acc.	rej.	rej.	**V**	V	S
B3	fixed	acc.	acc.	acc.	rej.	rej.	**V**	V	S
B4	fixed	acc.	acc.	acc.	rej.	rej.	**V**	V	S
I1	fixed	acc.	acc.	acc.	acc.	rej.	S	S	S
L1	co-edit	acc.	rej.	acc.	rej.	rej.	**V**	V	V
L2	fixed	acc.	acc.	acc.	rej.	rej.	**V**	S	S
M1	fixed	acc.	acc.	acc.	acc.	rej.	S	S	S
A1	co-edit	acc.	rej.	acc.	acc.	rej.	S	S	S (approved)
D1	co-edit	acc.	rej.	acc.	acc.	rej.	S	S	**V**

Table 35: **E130: every check on every edit, and each requirement's status.** K0 the authors' check, K2 the frozen report comparison, K3 the numeric comparison, K4 the re-derivation check, K5 the frozen evidence lock. S: satisfied, V: violated. Bold: the edit's focal requirement. The command writes no tracked file, so each status holds in the generated output too. The approved update A1 is judged against its approved values.

Edits and trust settings. Every edit ran once with its clean pair immediately before it; with three clean runs at the start and one at the end there are 15 clean runs and 11 edit runs, 26 gate runs in all, and 13 witness runs. C1, a crash control, points the claim code at a missing directory. In the fixed-verifier setting the verifier and its expected values stay as pinned: B1–B4 each move one rate across one threshold of the rule and keep the stored label (B1 is E128's F3b, byte for byte); L2 swaps two natural labels and keeps every count; I1 changes a rate and keeps its label right; M1 rewrites 0.600 as 0.6. In the co-edit setting the expected values may change too: L1 relabels one trace and sets the expected counts to match, with no new evidence; D1 removes one record and lowers the expected count to match; A1 is an approved evidence update, approved by a policy that requires each label to follow from its evidence and the expected values to equal the new counts. Table 35 gives every verdict and every requirement's status.

What the runs show. Re-deriving each stored label from its stored evidence rejected each of the four evidence edits we made, each crossing one threshold of the paper's rule. The authors' verifier and the frozen report comparison accepted all four. It also rejected a swap of two labels that kept every count the verifier compares. We turned the diagnosis from our previous study of this compendium into this check, and we chose the edits after reading the code, each to cross one of the rule's thresholds. Each of the check's verdicts follows from its definition; the run confirms the mechanism and the harness. This is one compendium: a case, not a rate. One of these edits, which we made, changed one replay outcome of one trace. Under the paper's rule, the abstract's 'only 1 of the 7' would then be 2 of 8. The authors' verifier still passed, because it counts the stored labels. The label swap kept every count the authors' verifier compares, so the verifier and the frozen report comparison accepted it. It moved one prefix-0 failure from one interface to another, which changes the per-interface split that the paper's Figure 2 plots. Both are consequences of edits we injected, not errors in the published paper.

Co-edits, and the approved update. When a change may also edit the verifier's expected values, the authors' verifier accepted both a relabelling made without new evidence and an approved evidence update, and the frozen report comparison rejected both. Of the checks we ran, only the re-derivation check separated them. It rejected the relabelling, whose label contradicts its unchanged evidence, and accepted the update, whose new label follows from its new evidence. The update was approved by a policy that requires each label to follow from its evidence, so the check's accept follows from the policy. In this case a frozen value blocked the approved update and a frozen rule did not. A frozen rule would block an approved change to the rule itself in the same way. An unapproved edit with the same bytes as the approved update would get the same verdict from every check; approval has to come from outside the repository.

The blind spot, and the simplest competitor. Removing a record and lowering the expected count to match passed the re-derivation check, because every label left still follows from its evidence.

The frozen report comparison and the frozen evidence lock rejected it. The re-derivation check tests whether each label present is right, not whether a label is missing. A frozen lock on the evidence files rejected all seven claim faults, including the removed record that the re-derivation check accepted. It also rejected the evidence change that kept its label, the equal rewriting of a rate, and the approved update. The re-derivation check rejected six of the seven faults and none of those three. Both results follow from the checks' definitions: the lock forbids every change to the evidence, and the re-derivation check allows a change whose labels still follow from it. The re-derivation check raised no alarm on a numerically equal rewriting of a rate that sits exactly on the 0.6 threshold, or on an evidence change that kept its record's label. It compares numbers, so its accept of the rewriting follows from its definition. The numeric comparison rejected no edit; no edit touches prose.

Costs. The re-derivation check is 143 non-blank lines of target-free logic and 130 lines of configuration for this compendium (15 items: the families, two evidence parsers, the two thresholds, the order of the tests and the label strings). The evidence lock is 35 lines and five frozen hashes. Of the 130 check cells on the gate runs, 109 accepted, 20 rejected and 1 was an error (the frozen report comparison on the crash control, where the report step printed nothing).

The locked lines. The protocol locked 88 prediction lines, which the scorer reads as 627 clauses: 12 on requirement status (12 held), 12 per check on verdicts and EscapedViolation (held: 12, 12, 12, 12 and 12 for K0, K2, K3, K4 and K5), 3 on nuisance alarms (3 held), 3 on the boundary validation (3 held), 3 on the witness (3 held) and 7 on totals (7 held). 0 lines were refuted and 0 unresolved; every validity gate passed, and no run was void. 27 lines are tagged as following from a check's definition. This is a confirmation, not a discovery: it confirms the derivation, the boundary validation and the harness on the real tree. It is a controlled diagnostic on the pinned repository on our machine, not an independent reproduction.

What this does not show. One compendium, edits chosen after reading the code: a case, not a rate. The check tests only whether each label follows from its stored evidence. It does not test whether the evidence is true, whether a record was removed, the fields derived from the labels, or whether two files agree about the same trace. A fabricated evidence update with consistent labels and expected counts has the approved update's bytes and passes the check. The co-edits here change expected values only; no edit changes the verifier's logic. The rule is our reading of the paper's text; it matches the authors' code on all 239 committed labels, and one team wrote the check and the second implementation, so a shared misreading would pass the validation.

Q THE FROZEN REPAIRS ON A SECOND COMPENDIUM (E131)

E131 asks whether the diagnosis, and the repairs it picks, carry over to a compendium we had not edited. Before this target was chosen, the two repairs that E128 and E130 built for the agent-failure compendium were frozen as text, with a closed list of what a target's configuration may set. R1 is the frozen report comparison: each value the project's verifier computes must equal the value it computed on the pinned commit, kept in a reference frozen outside the candidate. R2, the aggregate re-derivation, rebuilds every committed derived file from its committed inputs, with the derivation code taken from the pinned tree, and compares them cell by cell. A configuration may set four items only: paths, the two commands exactly as the project declares them, where the frozen reference is written, and the rounding the project declares. It names no value, no tolerance and no allowed disagreement.

The target. The prompt-injection compendium is `DaoyuanLi2816/RepoGuardBench` at commit `f1c1d28d` (101 tracked files, MIT licence for code), the code of Li (2026b), a non-archival workshop paper. It is rank 2 of the frozen order of E128's selection survey; rank 1 was the agent-failure compendium. It has the same author and the same verifier template. Its CI (K0, the native check) runs five steps: the tests, a smoke run of the aggregation code on mock data, table generation, a claim verifier that computes 18 values from six committed aggregate files and other committed files and compares each with a literal in its own source, and a release scan. Nothing in CI rebuilds the committed aggregates from the committed per-run results. Every run here is fresh execution of the pinned repository on our machine, in an environment built from the project's own requirements;

nothing re-runs the paper's model experiments. It is a controlled diagnostic, not an independent reproduction.

The diagnosis, written before any fault. In the four slots of the contract, the native check's reference is a set of literals inside the candidate commit, its observation the 18 values computed from committed aggregates, its stage the CI run on the aggregates as committed, and its artifact the candidate commit. The first slot picks R1 and the second picks R2. The contract named 5 gaps: G1, a change to the per-run results that leaves the aggregates alone; G2, a change to the aggregation code that still runs; G3, a change to an aggregate cell the paper prints but the verifier does not read; G4, a result changed together with its literal or with the verifier code that reads it; G5, a changed number in the README. On a second compendium, by the same author and with the same verifier template, the contract reading named five gaps before any fault was applied (two had already been noted in our earlier survey). Each fault we then built for a gap passed the native check, and the control was caught. The faults were built from the gaps. So this confirms our reading of this check. It does not measure how often the reading finds gaps.

Set-up. The two frozen repairs were set up on a second compendium, by the same author and with the same verifier template, using only the four configuration items the frozen logic allows: 36 lines, drafted 16 minutes after we first read the target. The freeze passed 66 minutes after that first read (it includes our own review). The set-up also needed 302 lines of new target-free code: a line-based report reader, exact string comparison, and the first code for R2's artifact form. The frozen logic describes these, but no earlier code implemented them. The rules of the logic were not changed. The new code is 57 lines for R1 and 245 for R2, and it names nothing of this target. The freeze registered 18 values for R1 and found R2 applicable to 21 of the 23 committed aggregate files; the other 2 are not written by the declared derivation, a finding about the release as published that we report as not applicable, never as a pass. R2 could not be set up for the per-run labels here. The project's scorer reads raw runs that are not shipped, and the frozen configuration has no item to point it elsewhere. This is a limit of the frozen interface. We did not widen it.

What transferred, and what was added. Table 36 separates the reused specification from the code and configuration the transfer needed. The specification carried over unchanged. The executable repairs did not: part of the frozen logic had no earlier implementation, so the transfer is reuse of a specification with new generic code, not an unchanged checker configured for a new target.

Edits and trust settings. Fixed-verifier edits leave the verifier and its literals as pinned: C1, the control, changes an aggregate cell the verifier reads; F1 flips one per-run label and leaves the aggregates alone (G1); F2 flips the sign of a difference in the aggregation code (G2); F3 changes one aggregate cell the verifier does not read (G3); F5 changes a README number (G5); P1–P3 are maintenance (a number rewritten as an equal one, a rename and a refactor in the aggregation code). Co-edits may also change the verifier: F4 is F1 with the aggregates rebuilt and the literal changed to match; F6 is C1 with the literal changed to match; F7 changes the committed headline file and points the verifier at an unchanged duplicate (all G4); A1 is F4 as an approved update, with the README and a changelog entry; P4 renames a verifier check and P5 refactors the verifier. The witness runs in its own trees, recomputes the printed values, the covered aggregate cells and the restatements with its own code, and never reads a verdict. Table 37 gives every run.

What the repairs caught. On this one compendium (same author and verifier template as E128's target), in FIXED-VERIFIER auditing, re-deriving the committed aggregates (R2) caught the two data faults the native check passed: a changed per-run label (F1) and a changed aggregate cell that the verifier does not read (F3). The frozen report comparison (R1) caught nothing the native check missed. On this one compendium (same author and verifier template as E128's target), in CO-EDIT auditing, R1 caught both co-edits that changed a result and its expected literal (F4, F6). R2 caught the one whose aggregates the inputs do not support (F6), and missed the one that rebuilt them (F4). Only R2 caught the co-edit that pointed the verifier at an unchanged duplicate file (F7): R1 reads what the candidate's own verifier prints. On this one compendium (same author and verifier template as E128's target), no check here caught the sign error in the aggregation code (F2): the committed aggregates were not rebuilt, and R2 re-derives with the pinned code on purpose. A check that ran

category	on the prompt-injection compendium (E131: same author and verifier template)	on UNIT (E134: other authors, a rebuild-and-compare verifier)
reused specification	the two repair rules, R1 and R2, frozen as text before the target was chosen, with the closed list of what a configuration may set; no rule changed	the same rules and E130's label re-derivation, frozen as code before any E134 target was read; no rule changed
reused executable code	E128's value comparison, imported unchanged (`as_number` and `leaf_equal` by R1 and R2, `shown` by R1); E122's numeric comparison and GateCheck's work-tree library, unchanged	E131's R1 and R2 and E130's K4 as byte copies, and E128's three helpers with the same source text; none changed
new generic code	302 non-blank lines that name nothing of this target: 57 for R1 (a line-based report reader and exact string comparison) and 245 for R2 (the first code for its artifact form)	344 non-blank lines of glue that name no target (221 to run the checks and compose workflows, 123 to load and validate a configuration), written before any E134 target was read
target configuration	36 non-blank lines in the 4 items the frozen logic allows: paths, the two commands as the project declares them, where the frozen reference is written, and the declared rounding (none is stated)	14 settings in 47 non-blank lines; R1 and R2 were set up with configuration only (the report command and a two-group pattern; the derivation command with its output directory)
adaptation code for this target	none	62 non-blank lines for K4: the per-record label rule and a reader for the JSON records (the frozen K4 has no configuration-only form)
unsupported requirements	R2 on the per-run labels (their raw runs are not shipped, and no item may point the scorer elsewhere); R2 on 2 of the 23 committed aggregate files, which the declared derivation does not write (not applicable, never a pass); the candidate's own aggregation code, which R2 does not run by design (so the sign error F2 passed)	no requirement lacked a witness, but no frozen repair reads the README (so none covers the one escape, FP#1) or checks which records are present; R2 does not run the candidate's own derivation code (K0 does here); the deleted-record family and the reordering edit had no site, because each record is its own file

Table 36: **E131 and E134: what each transfer reused and what it added.** From each freeze record, the configuration file, the imports of the repair modules (E131), and the frozen package's source record and adaptation ledger (E134). The studies' runners, edits and witnesses are experiment code, not part of the repairs, and are not counted.

run	setting	K0	R1	R2	K3	K0+R1+R2	requirements the witness finds violated
clean (3 runs)		acc. 3	acc. 3	acc. 3	acc. 3	acc. 3	none
C1	fixed	rej.	rej.	rej.	acc.	rej.	aggregates, code
F1	fixed	acc.	acc.	rej.	acc.	rej.	aggregates, claim, code
F2	fixed	acc.	acc.	acc.	acc.	acc.	code
F3	fixed	acc.	acc.	rej.	acc.	rej.	aggregates, code
F5	fixed	acc.	acc.	acc.	rej.	acc.	restatement
F4	co-edit	acc.	rej.	acc.	acc.	rej.	claim
F6	co-edit	acc.	rej.	rej.	acc.	rej.	aggregates, code
F7	co-edit	acc.	acc.	rej.	acc.	rej.	aggregates, code
A1	co-edit	acc.	rej.	acc.	rej.	rej.	none
P1	fixed	acc.	acc.	acc.	acc.	acc.	none
P2	fixed	acc.	acc.	acc.	acc.	acc.	none
P3	fixed	acc.	acc.	acc.	acc.	acc.	none
P4	co-edit	acc.	rej.	acc.	acc.	rej.	none
P5	co-edit	acc.	acc.	acc.	acc.	acc.	none

Table 37: **E131: every check on every run.** K0 the native check, R1 the frozen report comparison, R2 the aggregate re-derivation, K3 the numeric comparison. Witness requirements: the printed claims, the covered aggregate cells, the project's own derivation on the candidate's inputs, and the README restatements. None of the 68 check cells was an error or not applicable. The approved update is judged against its approved values.

the candidate's own derivation would catch F2, but it would pass a co-edit of the code and the aggregates. With the verifier fixed, K0 let 4 of the 5 fixed-verifier faults through, K0+R1+R2 2, and K0+R1+R2 with the numeric comparison 1; under co-edits K0 let all 3 through and K0+R1+R2 0.

Costs. R1 raised one false alarm (P4, a renamed verifier check). R2 raised none, but only one of the five maintenance edits (P1) could reach it. R1 and K3 both blocked the approved update. A frozen reference must be re-frozen, by someone outside the candidate, after every approved change. No check here can tell an approved update from a co-edit fault; only the approval can.

The locked lines. The protocol was committed alone after the configuration and before any fault was applied. It locked 92 prediction lines: 9 on set-up and applicability (9 held), 8 on the diagnosis (8 held), 53 on edit and check verdicts (53 held), 2 on nuisance alarms (2 held), 10 on the witness

per edit (10 held), 3 on the witness's validity (3 held) and 7 on totals (7 held). 0 lines failed and 0 were left unscored; no run was void. There were 17 gate runs and 16 witness runs. On the pinned tree the witness matched all 55 printed values it recomputes, all 399 covered aggregate cells and all 30 restatements.

What this does not show. This is one more compendium, by the same author as E128's target and with the same verifier template. It is a case, not a rate. It shows near transfer only. It does not show that the diagnosis or the repairs carry over to compendia by other authors or with other kinds of checks. The edits were chosen after reading the code, and their outcomes were derived from the code and records before this protocol; the runs test that derivation on the real tree. They are not a blind test, except for the R2 set-up lines (S03–S05). Nothing re-runs the model experiments, and R2 cannot judge whether the scored labels are true. A candidate file that shadows an import inside R2's copy is not tested. The frozen logic was written after reading the agent-failure compendium, so its fit to a second compendium by the same author is the easiest transfer to pass.

R A RECORD LIST BESIDE THE RE-DERIVATION CHECK (E132)

E132 asks whether a second check, built for the blind spot E130 measured, closes it without blocking the changes the requirements permit. The re-derivation check (K4) tests whether each stored label follows from its stored evidence; it passed a removed record (App. P). The record-list check (K_member) compares the records present in the five result files, by trace id and with multiplicity, with a list of the 189 required records frozen outside the repository. The combined check (K_composed) requires both. A simple baseline (K_count) keeps only a frozen row count per file and a rule that no trace id appears twice. Beside them run the authors' command (K0), the frozen report comparison (K2) and the frozen evidence lock (K5) of E130. The protocol was committed alone before any run, and the raw artifact alone before the scorer existed.

Edits, trust settings and approvals. The nine claim faults are E130's seven and two new ones: X1 copies natural line 2 over line 4, and X2 relabels line 4 as a trace the release does not hold (its seed 20008 becomes 20108). The five permitted changes are E130's three, a reordering of two records (R1), and P1, an approved withdrawal of one trace (its row, its raw trace and two expected values). A1 and P1 are co-edits. Each frozen reference has its own approval for each change that moves it: K2 new numbers, K_member a new record list (one id, for P1), K5 the new bytes. Every edit ran with a clean pair, under the pinned references and under the approvals: 34 gate runs (19 clean) and 17 witness runs. The witness adds a count of the records (W_member) and reads the release's own raw trace file as a second source for the natural results (W_xref). Table 38 gives every cell; Table 39 the workflows.

What the combined check did. The re-derivation check sees only the labels present, so it passed a removed record. We added a second check that compares the records present, by trace id, with a list of the required records kept outside the repository, and required both checks to pass. The combined check rejected all nine claim faults we made, including the removed record, a record copied over another, and a record relabelled as a trace the release does not hold. It accepted all five permitted changes: an evidence change that kept its label, an equal rewriting of a rate, an approved evidence update, a reordering of two records, and an approved withdrawal of one trace, whose new record list was approved outside the repository. We built the record-list check for the blind spot our previous run measured, and chose every edit after reading the code. Each verdict follows from the checks' definitions; the run confirms the design and the harness on the real tree. One compendium: a case, not a rate. The combined check protects two declared relations: each label follows from its evidence, and the records present are the approved ones. It does not show that the evidence is true. The withdrawal left the README, the results page and the PDF saying 13/25; we did not check those restatements.

Two relations, disjoint faults. On these nine faults the two relations did not overlap. The re-derivation check rejected the six faults that changed a label or its evidence, and none of the three that changed which records are present. The record-list check rejected exactly those three, and none of the six. Checking that each label follows from its evidence does not check that the right records are there, and the reverse also holds.

edit	setting	K0	K2	K4	K_ member	K_ composed	K_ count	K5	focal requirement
C1	control	rej.	error	acc.	acc.	acc.	acc.	acc.	none
B1	fixed	**acc.**	**acc.**	rej.	**acc.**	rej.	**acc.**	rej.	R_label V
B2	fixed	**acc.**	**acc.**	rej.	**acc.**	rej.	**acc.**	rej.	R_label V
B3	fixed	**acc.**	**acc.**	rej.	**acc.**	rej.	**acc.**	rej.	R_label V
B4	fixed	**acc.**	**acc.**	rej.	**acc.**	rej.	**acc.**	rej.	R_label V
L2	fixed	**acc.**	**acc.**	rej.	**acc.**	rej.	**acc.**	rej.	R_label V
X1	fixed	**acc.**	**acc.**	**acc.**	rej.	rej.	rej.	rej.	R_member V
X2	fixed	**acc.**	**acc.**	**acc.**	rej.	rej.	**acc.**	rej.	R_member V
L1	co-edit	**acc.**	rej.	rej.	**acc.**	rej.	**acc.**	rej.	R_label V
D1	co-edit	**acc.**	rej.	**acc.**	rej.	rej.	rej.	rej.	R_val V
I1	fixed	acc.	acc.	acc.	acc.	acc.	acc.	**rej.**	none
M1	fixed	acc.	acc.	acc.	acc.	acc.	acc.	**rej.**	none
R1	fixed	acc.	acc.	acc.	acc.	acc.	acc.	**rej.**	none
A1	co-edit	acc.	**rej.**	acc.	acc.	acc.	acc.	**rej.**	none
P1	co-edit	acc.	**rej.**	acc.	**rej.**	**rej.**	**rej.**	**rej.**	none
A1, approved	co-edit	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
P1, approved	co-edit	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
D1, P1 approved	co-edit	**acc.**	rej.	**acc.**	**acc.**	**acc.**	**acc.**	**acc.**	R_val V

Table 38: **E132: every check on every edit.** Pinned references, except the last three rows: each approved change judged against its approved references, and the incomplete removal D1 judged against P1's approved references. Bold: a claim fault accepted or a permitted change rejected. The clean runs and every pair were accepted by every check. K2 gives an error on the crash control C1, which writes no report.

workflow	claim faults escaped (of 9)	permitted blocked, no approvals (of 5)	with approvals
K0 alone	9	0	0
K0 + K2	7	2	0
K0 + K4	3	0	0
K0 + K_member	6	1	0
K0 + K_composed	0	1	0
K0 + K5	0	5	3
K0 + K2 + K4	2	2	0
K0 + K4 + K_count	1	1	0

Table 39: **E132: escapes and blocked changes per workflow.** A workflow accepts only if every check in it accepts. Escaped: accepted, with the focal requirement violated in the witness's run.

Faults that keep every count. Copying one record over another kept every count and the per-interface split, so the authors' verifier, the frozen report comparison and the re-derivation check all passed it. The table then held 25 rows but 24 traces, one of them twice: by trace, 12 of 24 were nontrivial, not 13 of 25. Relabelling one record as a trace the release does not hold kept even every id unique. The record-list check rejected both, and so did the release's own raw trace file, read as a second source. Swapping the same two rows, which changes nothing the paper reports, passed the record-list check and failed the byte lock. We chose the copied and the relabelled record so that every count stays the same.

The simple baseline. A simpler check, a frozen row count per file plus a rule that no trace id appears twice, gave the record list's verdict on every edit but one. It passed the relabelled record, which keeps every count and every id unique. On the other edits, the full list of ids added nothing over five counts. We made the relabelled record to separate the two. In all, the two verdicts agreed on 37 of the 38 comparisons (every gate run under the pinned list, and the four runs of the withdrawal and the removal under the approved list).

What each approval must hold. Every frozen reference blocks some permitted change until someone outside the repository approves a new version. What the approval must hold differs. For the frozen report values, it is the new numbers: two for the evidence update, two for the withdrawal. For the record list, it is the id of the withdrawn trace, and nothing for the evidence update. For the rule, it is nothing. For the byte lock, it is the new bytes of every changed file, which the candidate itself supplies, and without an approval the byte lock also stops the three changes that need none. Without

any approval, the frozen report values stopped the two changes that changed a reported number, the record list stopped the withdrawal, and the byte lock stopped all five. With the approvals, only the byte lock stopped anything.

The limit of an approved list. Once the withdrawal was approved, the record list, the row counts and the byte lock could no longer tell the incomplete removal from the approved withdrawal. The removal left the raw trace behind, so the frozen trace count still caught it, and so did the raw trace file. An approved record list is only as complete as the approval.

Validation, locked lines and cost. Before any edit, the record-list check gave the predicted message on each of five self-tests (an unexpected record, a duplicate, a missing file, a missing id column, an unreadable file), and a separate count of the records agreed with it on every run. The same team wrote both. The protocol locked 88 prediction lines (726 clauses): 7 on the requirement status of each candidate (7 held), 12 replicating E130's cells (12 held), 16 on E130's checks on the new edits (16 held), 25 on the new checks (25 held), 3 on the baseline (3 held), 8 on nuisance alarms and approvals (8 held), 3 on the list's validation (3 held), 4 on the witness (4 held) and 10 on totals (10 held). 0 lines were refuted and 0 unresolved; no run was void or repeated. 54 lines follow from the checks' definitions and 11 more in part, so this is a confirmation, not a discovery. The record-list check is 145 non-blank lines of target-free code and 23 lines of configuration (5 items); the list is 189 ids. Every gate run shared the machine with other studies, so we report no timing.

What this does not show. One compendium, edits chosen after reading the code: a case, not a rate. The record list is taken from the release as pinned: it checks that the records present are the approved ones, not that the approved ones are all that should exist. It reads the five result files only. The release's raw trace file is a second source for the natural results only. A record whose content is replaced under the same trace id passes it. On eight of the nine faults the full list adds nothing over five row counts. The withdrawal is our scenario, and its approval was given by our run plan; E132 tests the mechanism of an approved list, not who may approve it. The record-list check, the row counts and the record count share authors.

S THE CANDIDATE'S OWN COMPUTATION (E133)

E131 left one fault that no check caught: a sign error in the aggregation code (F2), with the committed aggregates left as they were. The aggregate re-derivation (R2) cannot see it, because it rebuilds with the approved code on purpose. E133 adds one comparison. For each derived cell there are three values: the committed one, the approved code's result, and the candidate code's result, both run by the approved command on the candidate's committed per-run results. R2 compares the first two (relation a). The new check, R3, compares the last two (relation b), in an isolated copy whose imports are audited. A plain rebuild-then-diff (RD) compares the first and the third (relation c). The target is E131's prompt-injection compendium at the same pin; R3 was applicable to the same 21 of the 23 committed aggregate files as R2. The protocol was committed alone before any run on the target; every outcome was derived from the code and E131's records before the runs, except the blind lines named below.

Edits. E131's runs were carried unchanged. The new ones are F8 (the sign error with the aggregates rebuilt by the wrong code), F9 (F8 with the verifier's literals edited to match, a co-edit), three refactors of the aggregation code (P6 rewrites the line F2 breaks; P7 writes a mean of differences as a difference of means, which changes only floating-point rounding), and A2, an approved change of the interval rule (the exact normal quantile in the Wilson interval), judged under the old rule and, after the new rule was frozen outside the candidate, under the new one (A2N); KN is the unchanged release under the new rule. There were 24 gate runs and 21 witness runs; all 120 check cells gave an accept or a reject (Table 40).

The sign error. On RepoGuardBench, the sign error in the aggregation code that no earlier check caught (F2) was caught by running the candidate's derivation files, by the approved command, on the candidate's committed per-run results, and comparing the result with the approved derivation's (R3). Among the checks of the workflow, R3 was the only one that caught it. A project's own

run	setting	K0	R1	R2	R3	RD	K3	K0+R1 +R2	+R3	requirements the witness finds violated
clean (3 runs)		acc. 3	acc. 3	acc. 3	acc. 3	acc. 3	acc. 3	acc. 3	acc. 3	none
F2	fixed	acc.	acc.	acc.	rej.	rej.	acc.	acc.	rej.	code, computation
F8	fixed	rej.	rej.	rej.	rej.	acc.	acc.	rej.	rej.	aggregates, computation
F9	co-edit	acc.	rej.	rej.	rej.	acc.	acc.	rej.	rej.	aggregates, computation
C1	fixed	rej.	rej.	rej.	acc.	rej.	acc.	rej.	rej.	aggregates, code
F1	fixed	acc.	acc.	rej.	acc.	rej.	acc.	rej.	rej.	aggregates, claim, code
F3	fixed	acc.	acc.	rej.	acc.	rej.	acc.	rej.	rej.	aggregates, code
F5	fixed	acc.	acc.	acc.	acc.	acc.	rej.	acc.	acc.	restatement
F4	co-edit	acc.	rej.	acc.	acc.	acc.	acc.	rej.	rej.	claim
F6	co-edit	acc.	rej.	rej.	acc.	rej.	acc.	rej.	rej.	aggregates, code
F7	co-edit	acc.	acc.	rej.	acc.	rej.	acc.	rej.	rej.	aggregates, code
P1	fixed	acc.	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
P2	fixed	acc.	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
P3	fixed	acc.	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
P4	co-edit	acc.	rej.	acc.	acc.	acc.	acc.	rej.	rej.	none
P5	co-edit	acc.	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
P6	fixed	acc.	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
P7	fixed	acc.	rej.	rej.	rej.	acc.	acc.	rej.	rej.	none
A1	co-edit	acc.	rej.	acc.	acc.	acc.	rej.	rej.	rej.	none
A2	fixed, old rule	acc.	acc.	rej.	rej.	acc.	acc.	rej.	rej.	none
A2N	fixed, new rule	acc.	acc.	acc.	acc.	acc.	acc.	acc.	acc.	none
KN	fixed, new rule	acc.	acc.	rej.	rej.	acc.	acc.	rej.	rej.	none

Table 40: **E133: every check on every run.** K0 the native check, R1 the frozen report comparison, R2 the aggregate re-derivation (committed against approved), R3 the candidate's code against the approved code on the same inputs, RD a rebuild with the candidate's code against the committed files, K3 the numeric comparison. The rule rows are judged under the approved interval rule named in their setting; a rejection there is a rule-contract rejection, neither a catch nor a false alarm. A1 is E131's approved value update, which a frozen reference blocks until it is re-frozen.

rebuild-then-diff (RD) also caught it, because the committed files were stale. Rebuilding with the candidate's code and comparing with the committed files (RD, what `make aggregate` followed by a diff checks) caught the sign error only while the committed files were stale (F2). Once the aggregates were rebuilt with the wrong code (F8, F9), RD passed them and R2 caught them. R3 caught all three. This confirms, as the edit definitions imply, E131's sentence that a check running the candidate's own derivation would pass a co-edit of the code and the aggregates. It also shows that R2 and RD together already catch all three. On F2, R3 named 141 cells, all in the seven files that hold the utility change.

Three relations. Any two of the three comparisons catch the same runs. R3 adds two things: it locates the fault (with R2, which two relations break says whether the code or the outputs changed), and it catches a code fault whether or not the outputs were rebuilt (F2, F8, F9). RD alone misses the rebuilt ones; R2 alone misses the stale one. Every run that broke a relation broke exactly two (by construction, never one), and which two says where the fault sits: the candidate's code differs from the approved code with stale outputs (F2); the candidate's code differs and the outputs follow it (F8, F9, and the rule changes A2 and KN); or the outputs or inputs changed without a rebuild (C1, F1, F3, F6, F7). F4 (an input changed, the aggregates rebuilt consistently) and F5 (prose) broke none: the relations see only code, inputs and derived files. Of the 24 runs, 11 broke two relations and 13 broke none.

Refactors and the cost of exact comparison. R3 flagged none of the three refactors of the aggregation code, including a rewrite of the very line the sign error breaks (P6). These refactors give byte-equal outputs (P2, P3) or run the same pandas operation (P6), so they test only that R3 does not read code text. A refactor that changes only floating-point rounding (P7: a mean of differences written as a difference of means, aggregates rebuilt) was flagged by R3, and by R2, on 38 cells, each by less than 1e-12. No number the paper defines moved by more than 1e-12 (W04). This is a false alarm of exact comparison; a declared tolerance would remove it.

A rule change. An approved change of the interval rule (A2: the exact normal quantile in the Wilson interval, aggregates rebuilt) passed the native check and R1, because no value the verifier prints changed. R2 and R3 blocked it, because the approved rule was still the old one. After the new

rule was frozen outside the candidate, the same commit passed, and the unchanged release failed the new rule (KN). Freezing values (R1) and freezing the rule (R2, R3) protect different things. A rule change needs a newly approved rule, not only new expected values. R2 and R3 block any rule change under the old rule by construction; the informative cells are the native check and R1 on A2, and the runs under the new rule. On A2 and on KN, R2 and R3 each named 3037 interval cells.

The workflow. With R3 added, the repaired workflow let 1 of 6 FIXED-VERIFIER faults through (a README change, F5, which only the prose count K3 sees) instead of 2 of 6. Adding RD instead of R3 gives the same 1 of 6. In CO-EDIT auditing neither added anything: R1 or R2 already caught every co-edit fault here. Of the 7 maintenance edits, R3 flagged 1 (P7).

Isolation. E131's R2 runs the frozen files inside the candidate's full copy, and the aggregation script puts the copy's root first on `sys.path`. So R2 can run candidate modules that shadow an import (for example a candidate `numpy/` at the root). R3's isolated copy and its audit close this. No edit tests it on the target. On the real target, a trusted derivation that reached outside its copy (the package left out of the copy, so the editable install supplied it from the pinned clone) wrote the same 21 files. Only the audit showed that it had loaded code from outside the copy.

Locked lines. The protocol locked 51 lines: 6 whose outcome was not known in advance, 17 that follow by construction and 28 derived from the code and E131's records. 51 held, 0 failed and 0 were left unscored. E131's 17 runs, repeated in the same environment, gave the same verdicts and the same lists.

What this does not show. R3 shows agreement on the candidate's inputs, not that two programs are equal. A code change that alters results only on inputs the compendium does not contain passes R3. R3 runs the candidate's derivation files by the approved command, so a changed Makefile recipe is not seen. This is the compendium of E131. The edits were chosen after reading the code, and every outcome above was derived from the code and E131's records before the runs, or follows by construction, except the blind lines (S02–S04, S06, N22, T06). The approval of A2 was declared by us, not by the compendium's author. It is one compendium and a new check on it, not a transfer. A candidate that changes its dependencies, writes outside its copy or detects the check is not tested, and the witness does not cover the interval bounds.

T THE FROZEN REPAIR CODE ON A DIFFERENT VERIFIER (E134)

E134 asks whether the contract reading, and the repairs it picks, carry over to a compendium whose authors and verifier both differ from the agent-failure and prompt-injection compendia. Four rules separate it from E131. The repairs were frozen as executable code (R1, R2 and K4, with the glue that runs them) before any E134 target was read in depth. The target was chosen by a rule. The diagnosis and a predicted verdict for every site of every fault population were committed before the seed of the draw existed, and the seed was the hash of that commit. Every repair that could be set up was run, picked or not.

Selection. The pool was the 323-row survey of E128 (App. O); no new search was made. A mechanical filter on the recorded fields (an ML compendium; an author-made check that runs the path to a printed number; no author in common with the two earlier compendia; no recorded "no" on E128's first five requirements) left 1 row: `PsychometricsMZ/UNIT`, the code of a stat.ML paper on semiparametric causal mediation analysis by other authors (433 printed values in two tables). A re-read on paper admitted it by reading E128's sixth requirement by repository identity; E128's reading by bare name had excluded it, and the selection file disclosed this before it ran. It passed the selection's smoke, whose runs had a three-minute ceiling. Its native check (K0), `python scripts/verify.py`, locks the hash of each of its 14000 input records and of its 3 table files, counts the records, rebuilds the tables from the records with the candidate's own code, and compares bytes (class A2 in the selection's terms: a rebuild with a byte-exact comparison and hash-locked inputs). The two earlier compendia compare committed results with literals.

The diagnosis, before the draw. In the four slots, K0's reference is the record hashes, the committed tables and the count constants, all inside the candidate commit, with a documented command

that rewrites them; its observation is the record count, the record hashes and the hashes of the rebuilt and committed tables; its stage is a fresh rebuild at check time; its artifact is the candidate commit. The reading picked R1 (a reference inside the commit that a documented command rewrites) and K4 (records store labels next to their evidence), and did not pick R2, because K0 already rebuilds every committed table. It named six gaps, among them a README number (K0 never reads the README), code K0 never runs, and two costs: every byte of every record is hash-locked, and the table comparison is byte-exact.

The draw. Five fault families were fixed before any candidate was read: analysis code (FA), input data (FI, a changed cell or a deleted record), committed results (FE), reported prose (FP) and expected values (FV, a co-edit that also changes every equal number in the tables), with four permitted families (P1 drops trailing zeros of a table number, P2 renames a local, P3 inserts a comment line, P4 reverses the order of records). Every site was enumerated and predicted: 633129 sites. Two sites were drawn from each fault population and one from each permitted one; the deleted-record and record-order populations were empty, because each record is its own file. To these came a control (C1, a deleted file K0 reads) and an approved update (A1: an input cell changed, then the project's own documented rebuild and rewrite). Each ran once, 15 runs in all; none was void. Table 41 gives every run.

run	family	setting	K0	R1	R2	K4	all	requirements the witnesses find violated	reading
C1	C1	fixed	rej.	rej.	acc.	acc.	rej.	restatement unresolved	unresolved
FA#1	FA	fixed	acc.	acc.	acc.	acc.	acc.	none	no violation
FA#2	FA	fixed	acc.	acc.	acc.	acc.	acc.	none	no violation
FI-cell#1	FI-cell	fixed	rej.	acc.	acc.	acc.	rej.	none	no violation
FI-cell#2	FI-cell	fixed	rej.	acc.	acc.	rej.	rej.	label	violation
FE#1	FE	fixed	rej.	acc.	rej.	acc.	rej.	aggregates, code	violation
FE#2	FE	fixed	rej.	acc.	rej.	acc.	rej.	aggregates, code	violation
FP#1	FP	fixed	acc.	acc.	acc.	acc.	acc.	restatement	violation
FP#2	FP	fixed	acc.	acc.	acc.	acc.	acc.	none	no violation
FV#1	FV	co-edit	rej.	acc.	rej.	acc.	rej.	aggregates, code	violation
FV#2	FV	co-edit	rej.	acc.	rej.	acc.	rej.	aggregates, code, restatement	violation
P1#1	P1	fixed	rej.	acc.	rej.	acc.	rej.	none	no violation
P2#1	P2	fixed	acc.	acc.	acc.	acc.	acc.	none	no violation
P3#1	P3	fixed	acc.	acc.	acc.	acc.	acc.	none	no violation
A1	A1	fixed	acc.	acc.	acc.	acc.	acc.	none	no violation

Table 41: **E134: every check on every run.** K0 the native check; R1 the frozen report comparison, R2 the aggregate re-derivation and K4 the label re-derivation, as frozen code; "all" is K0 with the three repairs. A run with no violation that a check rejects is a blocked edit. C1's restatement status is unresolved because the witness cannot read the file C1 deletes.

Set-up and cost. On one compendium by a different author, with a verifier of class A2, the frozen repair code was set up with configuration only for R1 and R2: 14 settings, 47 lines, 86 minutes from the first read to a passing freeze. K4 also needed 62 lines of adaptation, for: stating the per-record label rule and reading JSON records. The package's 344 lines of glue were written for E134 before any target was read. The minutes run from the start of the selection's read of this compendium to the start of the freeze that was used (a passing freeze had been made earlier and was repeated after a fix to our runner). The freeze registered 9 printed values for R1, reproduced 3 of 3 committed table files for R2, and froze 25 label families (196550 labels) for K4. Table 36 (App. Q) separates what was reused from what was added.

What the diagnosis predicted. The contract reading, committed before the sites were drawn, predicted the native check's verdict on 10 of 10 randomly drawn faults (0 unclassified); always predicting K0's most common verdict gets 6 of 10. On the permitted edits and the control it predicted 4 of 4; the same baseline gets 2. The protocol locked 80 prediction lines: 69 hit, 0 missed and 11 were unresolved. Each unresolved line had been predicted "unknown": the rule could not tell in advance whether a changed standard error flips a coverage label, whether the co-edited numbers fall on cells the witness checks, or what a deleted file restates.

A simpler rule (post hoc). We wrote this rule after reading every native verdict, so it is not a test; we committed it alone before applying it. It sorts each edited file by what K0 does with it:

hash-locked (the records), rebuilt and compared (the three table files), never read (for example the README and the simulation code), or part of the check itself (`verify.py`, `make_tables.py` and `MANIFEST.json`). It predicts a rejection if any edited file is hash-locked or rebuilt, an acceptance if the check reads none of the edited files, and nothing otherwise. It matches K0 on 10 of 10 drawn faults, as the contract reading does. On the 5 runs that are not faults it matches 3, and the contract reading matches 5. The simpler rule makes no prediction for C1, which deletes the reference file. It predicts a rejection for A1, which K0 accepts, because the approved update rewrites the hash list inside the commit being checked; the rule has no slot for a reference the candidate can rewrite. We knew A1's outcome, so this one run is evidence for neither rule. Comparing predictions only, over all 633129 sites, the two readings never predict opposite verdicts. The simpler rule is silent on 317 sites, 316 in the table script and 1 in the reference file, and no drawn fault landed on one. So on this compendium the contract reading's result shows that the contract describes what K0 observes; it does not show better reasoning than such a rule.

What the checks let through and blocked. In FIXED-VERIFIER auditing, the native check let 1 of 4 drawn faults through with a witnessed violation; the workflow K0 AND {R1, R2, K4} let 1 through. The escape is a reported-prose fault: a README restatement of the number of seeds per cell, the gap the diagnosis named. Leaving out the requirement and check pairs that agree by construction (the code requirement with K0, the label requirement with K4), the full workflow let 1 of 3 through. In CO-EDIT auditing, the native check let 0 of 2 drawn faults through with a witnessed violation; the workflow K0 AND {R1, R2, K4} let 0 through. The other 4 fixed-verifier faults violated nothing the witnesses check: both analysis-code sites landed in simulation code the check never runs, one input cell feeds no table, and one README number restates nothing. K0 rejected that input cell anyway, because every byte of every record is hash-locked. The repairs blocked 1 of 3 permitted edits: P1#1 (a table number 1.20 rewritten as 1.2), by K0 and R2. The approved update passed every check.

What the repairs added. The contract reading picked R1, but on the drawn sites R1 caught nothing K0 let through. The contract reading picked K4, but on the drawn sites K4 caught nothing K0 let through. The reading did not pick R2, and R2 added no catch. On the drawn sites the three repairs added 0 catches and 0 blocks over the native check. The gap R1 was picked for (a harmful change followed by the documented rebuild and rewrite) is not produced by the frozen fault families; only its approved form, A1, was run. No frozen repair reads the README, so none covers the one escape. The reading predicted the native check's verdicts before the draw, but the two repairs it picked added nothing on the drawn sites; no frozen repair added protection on this verifier.

A step of the released code that the paper's text does not state. The witness recomputes the printed values with the exclusion rule the paper states (a fit is dropped when its second stage is skipped, when it does not converge, or when its estimate or standard error is not finite). Under that rule, 48 of the 433 printed values and 397 of the 3143 committed table cells do not come out. The released table code excludes more fits than that rule; among its steps are a cap that drops fits whose standard error exceeds 2 (`scripts/make_tables.py`, line 58, a step its docstring explains) and a divergence guard (line 55). The committed tables follow that code: K0 and R2 rebuild them exactly. The paper's text does not state the standard-error cap. A check made after the study was scored traces all 48 values to that cap alone (App. T.1). We fixed these values as outside the witness's scope before the draw, so no verdict here rests on them.

What this does not show. One compendium, with sites drawn from five declared fault families. It is a case, not a rate. The families and the operators were fixed by us; the sites were drawn at random within them. It does not show that the repairs protect claims in general, or that the committed evidence is true. One person did the selection reading, the diagnosis, the configuration and the witnesses in one evening; the review estimated one to three days or more for this study. The draw makes the sites independent of the diagnosis, not representative of real errors. The frozen package cannot see an error in the candidate's own derivation code when the committed outputs are left alone, or a record removed together with its derived rows. The same team wrote the repairs, the diagnosis and the witnesses.

T.1 WHICH STEP OF THE RELEASED CODE ACCOUNTS FOR THE LEFT-OUT VALUES (POST HOC)

We planned this check after E134 was scored, and committed the plan alone before running it. It is not a prospective result. It changes no E134 count, scope or verdict, and counts nothing as a detection. The released table code adds four steps to the exclusion rule the paper states (Table 42). We added each step alone to the stated rule, and removed each step alone from the released rule. Adding only the standard-error cap to the stated rule reproduces all 48 printed values and all 397 table cells that the stated rule does not. Removing only the cap from the released rule brings back exactly those values and cells. The cap drops 105 fits that the stated rule keeps. Every one of them is stored as covering the true value; the code's docstring gives the reason for the cap, that an interval this wide can only cover by accident. The other three steps change no value: they act only on fits the stated rule already drops. None of that paper's comparisons we could check changes, including its 1.45 to 1.51 fold reduction of the standard error; one sentence about skip counts could not be scored, because its intended scope is not clear to us. A rebuild check cannot see this gap: the tables follow the code byte for byte. A witness built from the paper's text does see it. We report it as an unstated step of the released code, not as an error in that paper's results.

step of the released code	fits it acts on	fits it drops beyond the stated rule	printed values it accounts for (of 48)	table cells it accounts for (of 397)
error flag	0	0	0	0
weak-identification flag	259	0	0	0
standard-error cap (SE $>$ 2)	146	105	48	397
divergence guard	18	0	0	0

Table 42: **E134, post hoc: the four steps the released code adds to the stated exclusion rule.** Each step is added alone to the stated rule; "accounts for" counts the left-out printed values and table cells it then reproduces.

U THE OUTCOME OVERLAY AND THE MATCHED STUDIES (POST HOC)

Everything in this appendix is post hoc. It reads only shipped records, it runs no target, gate or checker, and it re-scores no locked count. It has three outputs: the archived overlay (`data/review_20260923_gc/ledger_overlay.json`) and the matched studies (`data/review_20260923_gc/matched.json`), written by `experiments/review_20260923_gc/`, and the current outcome layer (`data/review_20260925c_gc/current_outcomes.json`), written by `experiments/review_20260925c_gc/`. Each records its version, and the verifier regenerates all three byte for byte.

What the overlay adds, and what it keeps. The overlay is a versioned layer beside the historical labels. It copies every label untouched and asserts, row by row, that the copy equals the ledger. To each unit it adds four fields. The *record role*: fault (a defect or a claim drift), control, maintenance (an edit that preserves the evidence, or a relocation of a registered claim) or authorized update (an intended change of evidence). *Edit persists*: true when the edited bytes are still in the work tree after the gate, false when the build rewrote them, and null when no record covers the edited files. This is what E122 and E124 used to print as "violation present"; it is not a statement about a claim, and both tables now print it as *edit persists*. The *witness status*: witnessed, structural (the violation is the edit itself, such as a crash by construction or a changed prose statement no build regenerates), preserved, no change, no record, no claim, or not applicable. The *requirement status* after the gate: violated, satisfied or unresolved.

The outcome population and the four outcomes. A workflow is the native command plus at most one added check, and it accepts only if both pass; an error is never a pass. A unit is in a workflow's outcome population when it puts a violation in front of the workflow, the workflow accepts its clean pair, and the workflow's verdict on it is scorable. A unit puts a violation in front of the workflow when a witness recorded one (witnessed or structural), whatever its record role, or when it is an authorized update whose approved revised requirement is violated. A role says what an edit was

meant to be; it never exempts a recorded violation. Inside the population, a unit is *rejected*, *accepted and restored* (the build rewrote the edited bytes and the claims read after the gate equal the clean pair's, in the work tree; the candidate commit still holds the edit), *accepted and persisting* (the main text calls this an escaped violation), or *accepted and unresolved* (the final state was not recorded). For E123 the native command is the reconstructed comparator and the observed result includes the terminal output. Units are kept apart: a ledger row is one injection, an E121, E122 or E124 unit is one frozen run read under one workflow, an E123 unit is one run-by-claim cell, and E121's four runs of D on a rebuilt policy tree are counted on their own. Table 43 prints the arm studies.

Fixtures for the classifier (post hoc, 25 September 2026). Each fixture builds a small record in the overlay's own schema and calls the overlay's own classification functions, and those behind Table 5. (a) A prose number changes while its evidence stays fixed: the requirement is violated and the edit is scored as a fault, whether the study filed it as a claim drift or as maintenance. (b) Only the formatting changes: permitted maintenance, satisfied, outside every outcome population. (c) The build replaces one wrong value with another: persisting, although the edited bytes are gone. (d) The build fixes the output while the commit keeps the wrong value: two outcomes, one per artifact. (e) An authorized update carries an unintended defect: the defect is kept and scored. (f) The evidence is missing: unresolved, never satisfied. The 6 fixtures hold 17 test cases. On the classifier as it stood, 15 passed and 2 failed, and both failures were a record role exempting a recorded violation: a maintenance label in (a) and an authorized-update label in (e). The classifier now applies the fault rule whatever the role, and judges an authorized update against its approved revised requirement; 17 of the 17 cases pass. In the shipped overlay this changed no record role, witness status, requirement status, edit persistence, verdict, population or outcome: 0 changes over 316 unit-by-workflow entries. Only the stated reason for being outside a population changed. The 99 entries that read "not a fault or control" now read "no witnessed violation" (80) or name an authorized update not shown to violate its approved requirement (19). The fix changes what the code would do, not what this paper reports. `experiments/review_20260925_gc/fixture_record.py` measures both states, and the verifier re-runs it.

Which artifact holds each violation: the five claim faults. Table 5 is generated from `data/review_20260924_gc/publication_state.json`, written after the reviews of 24 September 2026 by `experiments/review_20260924_gc/publication_state.py`. It reads only E122's shipped records, and E124's for agreement; it runs nothing and re-scores nothing. It names two artifacts. The *candidate snapshot* is the accepted candidate commit, released as committed, without a build. The *generated output* is the work tree the accepted build leaves: the build's regenerated outputs together with every file the build does not write, the README included. Each fault's requirement is written as a relation, not as an old number. For the analysis defect, the two swaps and the abstract edit it is: every stated value equals the value the declared scoring rule computes over the declared records. For the README edit it is agreement with an approved reference, the pinned README, because nothing computes the compute-time estimate. The record gives each fault's declared value, its value in the candidate commit, before the build and after it, and each value is cross-checked between the claim diffs, the pre-build hashes and the witness's stated files; 0 artifact states are unresolved. Then $\mathrm{EscapedViolation}_i(p)$ of §2 is scored for each workflow and each artifact. The project's own check alone rejects 0; 4 violations escape into the commit and 3 into the output. With the value lock it rejects 3, and 2 escape into the commit and 2 into the output. With the numeric comparison it rejects 3, and 2 escape into the commit and 0 into the output. The faults are wrong in both artifacts for 2 (S2, I2), in the commit only for 2 (S1, I1), and in the output only for 1 (D). D is the one subtle row: its committed claim file still states 97, so the snapshot's stated value satisfies the requirement, but its analysis code no longer implements the declared rule and its own build restates 98; the record flags this. Table 5's requirement type is read from a second record, `data/review_20260925_gc/fivefault_requirements.json`, which also holds the computation requirement the table does not score: for D it is violated in the commit as well as in the output, because the commit's code breaks the declared scoring rule. The stage matters too. The numeric comparison's reading before the build, logged and deciding no label, sees 3 of the faults (S2, I1 and I2); of the 2 faults the build restores, it sees only I1 there, because S1 keeps the count of every number. E124, a second run of the same edits, agrees with E122 on all 5 faults: the verdicts, the post-build claim files and the post-build file states.

					accepted		
target (study)	unit	workflow	n	rejected	restored	persisting	unresolved
CSV_Wrangling (E121)	frozen run	G0	3	1	0	2	0
CSV_Wrangling (E121)	frozen run	G0 + G1	3	3	0	0	0
CSV_Wrangling (E121)	frozen run	G0 + R	3	3	0	0	0
CSV_Wrangling (E122)	frozen run	G0	7	1	2	4	0
CSV_Wrangling (E122)	frozen run	G0 + G1	7	5	0	2	0
CSV_Wrangling (E122)	frozen run	G0 + R_lock	7	1	2	4	0
CSV_Wrangling (E122)	frozen run	G0 + R_pin	7	4	2	1	0
CSV_Wrangling (E122)	frozen run	G0 + G2	7	5	2	0	0
CSV_Wrangling (E124)	frozen run	G0 + G2	7	5	2	0	0
CSV_Wrangling (E124)	frozen run	G0 + G3 (after build)	7	5	2	0	0
CSV_Wrangling (E124)	frozen run	G0 + G3 (before build)	7	5	0	2	0
SGC (E123)	run × claim	G0	17	14	0	3	0
SGC (E123)	run × claim	G0 + G2	17	14	0	3	0
SGC (E123)	run × claim	G0 + R	17	14	0	3	0

Table 43: **The four outcomes, per workflow** (post hoc overlay, App. U). n is the number of violations, witnessed or structural, whose clean pair the workflow accepts, whatever the unit's record role. A workflow accepts only if the native command G0 and the added check both pass. Rejected; accepted with the build restoring the edited value in the work tree before any check read it; accepted with the violation persisting; accepted with no record of the final state. Restored and persisting are read on the work tree after the build, not on the candidate commit; on SGC the final state includes the terminal output. Units are frozen runs on CSV_Wrangling and run-by-claim cells on SGC, and no row is added to another. G0 + G1a is not shown: G1a fails every clean pair, so its n is zero. The full run-by-run matrices are Tables 29 and 30.

B4 is an authorized update, not a surviving error. E121's B4 retracts one file from the GitHub corpus, which the README says can happen; it changes numbers on purpose. E121's witness labels it witnessed, so the E121 line of Table 12 counts it among its 3 witnessed-and-surviving rows, of which 2 are a fault or a control (D and PC). The overlay gives B4 the role authorized update in E121, E122 and E124, carried from E121's edit record, and scores its requirement satisfied after the gate, because the claims read after the gate equal the witness's rebuild from the updated evidence. It is outside every outcome population.

					accepted		
population	defect class or control	rows	n	rejected	regen.*	persisting	unresolved
4 chosen targets (E76, E77)	seven portable classes	98	15	8	0	0	7
System A (E75)	twelve own classes and one variant	43	11	5	4	0	2
13 JOSS repositories (E94, E98)	prose number, weakened assertion, wrong constant	115	0	0	0	0	0
5 of the JOSS 10 (E96, E116)	swapped call	18	6	4	0	0	2
10 JOSS repositories (E96)	deleted step	32	0	0	0	0	0
15 public compendia (E114)	deleted directory, prose drift	22	16	16	0	0	0
15 public compendia (E114)	controls: blank line, comment, re-save	43	0	0	0	0	0
System A, held out (E107, E110)	deleted directory, prose drift	18	18	17	0	0	1
CSV_Wrangling (E121)	one code defect, its positive control, 5 maintenance edits and one authorized update	8	2	0	0	2	0

Table 44: **The historical rows under the four outcomes.** One line per line of Table 12, under the gate each study recorded. n counts the witnessed violations whose clean pair the gate accepts. Where the historical runner reset the tree after the gate, an accepted violation is *accepted, unresolved*: edit persistence is recorded for 12 of the 397 rows of Table 12 (13 of the whole ledger's 435, the rows the table keeps out included). *Regenerated; final validity unverified: the overlay's *accepted, restored*, which for these rows rests on a regeneration record, not on a witness. There is no total row.

The historical rows. Table 44 reads the ledger through the overlay. Two facts carry the finding of §4.4. A witnessed violation is 15 of 98 rows for the chosen targets and 0 of 115 for the JOSS prose, assertion and constant draws. And the final state of an accepted violation is almost never recorded, so the historical studies cannot distinguish a violation that persisted from one the build put back; System A's 4 rows in the regenerated column (4 in all, the same rows as the regenerated column of Table 11) rest on its protocol's record that the verifier regenerates the mutated file. Their archived label is *repaired during execution*, and the overlay records them as accepted and restored with the requirement satisfied; that status is inferred from the regeneration, not observed. Under the definition of §2 restoration needs a witness, so the paper presents them as regenerated, with their final validity unverified.

The current outcome layer. The overlay above is kept as an archive. Every present-day table reads a second file, the current outcome layer, whose labels follow the definitions of §2 exactly: rejected, accepted and restored (only with a witness), accepted and escaped, or accepted and unresolved, per requirement, artifact and workflow, with maintenance alarms counted apart. It differs from the archive in one place: the 4 regenerated System A rows are accepted and unresolved. Over the 68 witnessed rows of Table 12 with a passing clean pair, it gives 50 rejected, 0 restored, 2 escaped and 16 unresolved. Table 45 summarizes it for the three contract compendia. We chose the seven faults after reading the verifier, one per layer of the claim path; which ones escape follows from that choice, not from a sample. Two of the four that the authors' check accepted (WF1, F6) edit the verifier or its expected values.

compendium	workflow	artifact	n	rejected	restored	escaped	unresolved	maintenance blocked
CSV	own check	commit	4	0	0	4	0	0/1
CSV	own check	output	5	0	2	3	0	0/1
CSV	+ value lock	commit	4	2	0	2	0	1/1
CSV	+ value lock	output	5	3	0	2	0	1/1
CSV	+ numeric comparison	commit	4	2	0	2	0	0/1
CSV	+ numeric comparison	output	5	3	2	0	0	0/1
graph network	README comparison	run output	12	6	0	6	0	0/3
graph network	recomputation	run output	12	9	0	3	0	0/3
graph network	loader-id check	run output	12	6	0	6	0	0/3
graph network	combined check	run output	12	12	0	0	0	0/3
agent failure	authors' check	commit = output	7	3	0	4	0	1/3
agent failure	+ value lock	commit = output	n/a	n/a	n/a	n/a	n/a	n/a
agent failure	+ frozen report comparison	commit = output	7	5	0	2	0	1/3
agent failure	+ numeric comparison	commit = output	7	4	0	3	0	1/3

Table 45: **The current outcome layer, per compendium, workflow and artifact.** CSV: the requirement Table 5 scores, one unit per claim fault (the analysis defect leaves the committed value right, so it is not a commit unit); maintenance: one permitted edit. Graph network: one unit per edit and claim, any requirement broken. Agent failure: one unit per claim fault, its focal requirement; the command writes nothing, so commit and output agree. n/a: the value lock was not applicable, because the command wrote no file for it to compare.

E123: three requirements, and the witness's two clauses. The E123 text used to say that the witness never reads the README, while D3, a README-only edit, is labelled witnessed on one claim. The code is right and that sentence was wrong. The witness W has two clauses, both declared in the protocol before any run. Clause (i) fires when the accuracy W recomputes (correct predictions over the declared test ids, argmax of the model's own logits) differs from the printed one; it reads no README. Clause (ii) fires when the printed value, at the precision of the README literal, differs from that literal, which W reads from the README of its own tree. D3 on cora fires clause (ii) alone: printed and recomputed agree, and the README literal was edited. Of 42 cells, 17 are witnessed: clause (i) fires on 12, clause (ii) on 14, both on 9, clause (i) alone on 3 (D1b) and clause (ii) alone on 5 (SC on all three claims, D3 and D3-on-B1 on cora). The reconstructed comparator G0 equals clause (ii) on all 42 cells, which is why we call it a documentation-derived rounded-accuracy comparator. The three requirements then read as follows. *Evaluation-population integrity*, the printed accuracy is computed over exactly the declared test ids, is witnessed by clause (i) on 6 cells (D1, D1b). Clause (i) is a numeric consistency check against a recomputation over the declared population, not a comparison of populations: it witnessed these two edits because each moved the accuracy, and it would pass a population edit that leaves the accuracy unchanged. App. N.5 runs

such an edit: every test node evaluated twice. *Metric correctness*, the printed number is the argmax-accuracy fraction of that population, is clause (i) on 6 cells (D2, PC). The code cannot tell these two apart; we split them by each edit's declared class, and two recorded facts back the split on every cell: population edits keep the clean weights and the clean validation accuracy, and metric edits keep the weights but move the validation accuracy. *Reporting consistency*, the README value equals the printed value at its precision, is clause (ii) on 14 cells. D1b is a population discrepancy below the comparator's resolution: the printed value moves by at most 0.03 percentage points against a README resolution of 0.1, and the rounded headline is unchanged on 3 of 3 claims. One further reading is derived, not implemented anywhere: "the README equals the recomputed accuracy" is false on 5 cells (D3, SC on all three claims, D3-on-B1). On D1 the README is true of the trained model; only the printout is wrong.

Matched scope: G3's silence on B2. G2r is G2 cut down to the 249 (file, number) pairs that G3's 270 registered claims hold, in 28 files; those pairs cover 270 of G2's 309 mentions in 34 files. It is computed exactly from G2's shipped alarm lists, because G2 emits one alarm per registered pair and each alarm depends only on that pair's count in its file, and every canonical value G3 holds equals G2's canonical token for the same claim. The population is 27 runs at both readings, 54 cells, all against the pristine reference; 5 runs are kept out because their reference is not the pristine one (the deleted-registry run and the four policy runs). On B2, G2 raises 11 alarms at each reading, and they are exactly the numbers of the BibTeX block B2 moves (year, volume, issue, pages, ISSN and DOI fragments); the README's 1 registered pair is not in that block. G2r raises 0 and G3 0. G2r and G3 agree on 52 of 54 cells and on all 27 after the build. Scope separates G2 from G2r on 2 cells (B2 at both readings); identity separates G2r from G3's claim alarms on 1 (S1 before the build); the evidence digest separates G3 from its own claim alarms on 1 (B4 before the build).

arm	compares with	scope	reading	passes when	own reference deleted
G0	none	exit status of the build	after build	exit 0 within 1800 s	no frozen reference
G1a	candidate commit	every tracked file	after build	no tracked file differs	no frozen reference
G1	candidate commit	27 claim files	after build	no claim file differs	no frozen reference
R_lock	pristine, frozen	5 directories exist	after build; before logged	no locked directory gone	L1 (E122): error
R_pin	pristine, frozen	263 decimals in 21 files	after build; before logged	no pinned count drops	L1 (E122): error
G2	pristine, frozen	309 numbers in 34 files	after build; before logged	no registered count drops	L2 (E122): alarm
G3	pristine, frozen	270 claims; 26 evidence files	before and after build	no alarm of any kind	L3 (E124): alarm
G0 (E123)	candidate commit's README	3 printed accuracies	terminal output	equal at one decimal	no frozen reference
G2 (E123)	protocol's values	3 printed accuracies	terminal output	equal at one decimal	no frozen reference

Table 46: **What each check compares with, with codes.** The full form of Table 1: one line per arm, as implemented; R, in E121 and E123, is R_lock and R_pin run as one arm. G1a and G1 compare the build with the candidate commit; R_lock, R_pin, G2 and G3 compare with a reference frozen on the pristine tree. On SGC, G0 compares the printout with the candidate commit's README, and G2, the protocol's registered-claim check, with values written in E123's protocol, which are the pristine README's. The last column is the control that deletes the arm's own frozen reference, and the arm's result, which is the same before and after the build (post hoc summary, this appendix). "Before logged": the reading before the build is recorded and decides no label. Every count in the table is recounted from the frozen baselines.

Each arm with its own frozen reference deleted. L1 (E122) deletes the directory lock and the mention baseline; R_lock and R_pin raise `FileNotFoundError`, which the runner records as an error, before and after the build. L2 (E122) deletes G2's numeric manifest; G2 fails with its declared alarm, "numeric manifest missing". L3 (E124) deletes G3's registry; G3 fails with its declared alarm, "typed claim registry missing". So all 4 arms reject, 2 by an exception and 2 by a declared alarm, and all 4 clean pairs pass. In each run the other frozen files are byte-identical to the pristine ones. L3 alone compares G3 without its reference with R_lock, R_pin and G2 with theirs, so it is not a like-for-like control, and we do not read it as G3 against G2. The 3 arms G0, G1a and G1 keep no frozen state, so no deletion control applies to them.

What E124's protocol requires wherever its counts are quoted. E124 is one compendium, 12 edits and 270 registered claims, all chosen by us after reading the tree: a mechanism table, not a rate,

work	what is audited	what makes a violation meaningful	what the outcome refers to	stage, reference trust and final artifact
mutation testing (Just et al., 2014; Humbatova et al., 2021)	a program's test suite	the mutant changes the program's behaviour	the mutant: killed or surviving	not part of the outcome
MLReproMutate (Shulepov, 2026)	a repository's own validation workflow	the mutation is confirmed to change behaviour	whether the workflow detects it	not part of the outcome
executable papers (Luger & showyourwork contributors, 2021)	an article's build	the build fails	the rebuilt article	the rebuilt artifact only
claim checkers (Nuijten & Wicherts, 2024)	printed statistics	a recomputed value differs	the printed claim	the reference is the printed statistics
GateCheck (ours)	each check's contract, per requirement	a witness that does not use the verdict	the requirement's state in a named artifact	each stated in the contract

Table 47: **What each line of work audits**, for the works we compare with in §5. The last column says whether the stage, the trust in the reference and the final artifact enter the outcome.

in counts, never percentages. Every claim on it has exactly one occurrence, so the case the schema exists for — one number printed in an abstract, a table and a sentence — could not be exercised, because the paper's source is not in the repository. The one prose claim was registered by hand, by us, after reading this repository's prose, and two of the 12 edits touch it. A check that runs before the build is blind to every drift the build introduces, including D and PC, and Table 30 prints both readings. The swap S1 was seen by typed claim identity and by no count-based check, but only at a reading taken before the project's own make ran; after it, only the value lock saw it.

V Running the other tool on the same repositories

Shulepov (2026) asks a related question — does a research project's own checking catch a defect that would change its results? — and releases a tool. We ran it on the same repositories.

The other lines of work in Table 47. Just et al. (2014) seeded 230,000 mutants into 5 Java programs with 357 real faults. A test suite that detects more mutants also finds more real faults, independently of code coverage. 17% of their real faults match no mutant at all, mostly changed algorithms and deleted code; in our setting these are a wrong *analysis* and a *dropped step*, and we add both as operators (App. G). Humbatova et al. (2021) build 35 mutation operators from real deep-learning faults, and Panichella & Liem (2021) warn that in ML the line between code and test data blurs, so the realism of an operator can be challenged. All of this work scores a program's test suite. The showyourwork tool rebuilds an article's figures and PDF from its data and scripts on every push (Luger & showyourwork contributors, 2021); it does not ask what a defect in the pipeline would do to a claim. The statcheck tool recomputes reported p-values from the statistics printed beside them. Over 7,000 articles, journals that adopted it saw a steeper fall in reporting inconsistencies than matched controls (Nuijten & Wicherts, 2024). That study evaluates one check by its effect on published outcomes; we evaluate many checks by what they would stop, and in which artifact.

What it changes, and what we change. Its four operators edit a *configuration*: an integer random seed becomes the next integer, an exact dependency pin == becomes >=, a `stratify=` argument becomes `None`, a fold count becomes one larger. Our seven edit a *released artifact*: a number in the prose, a value in a data file, the sign of a result, a cross-reference, an assertion, a directory of evidence, a constant in the analysis code.

The rule we report by. A reviewer asked for this in one sentence: *a baseline need not support every surface; report overlap and unique coverage explicitly instead of comparing unmatched aggregate percentages.* So Table 48 is counts, not rates. There is no single number in this appendix comparing its detection rate with ours, because the two tools are not looking at the same places.

Before trusting a zero. We first ran the tool on its own shipped fixture and checked the outcome against its own README: the seed change survives the unguarded validation command and is killed by the guarded one. Both reproduced. Without that check, a target where it finds nothing could just as easily mean we drove it wrong.

Table 48: What each tool finds on the same five repositories. The four middle columns are Shulepov (2026)'s operators; **GC** is the total site count of our seven. **Shared** is the number of `(file, line)` pairs both tools would edit. **Killed** is how many of its mutations failed that repository's own gate on a single run, out of how many we ran; lifelines' one failure is kept as recorded and its cause is unresolved (text).

repository	seed	pin	split	folds	GC	shared	killed
harnessdrift	0	62	0	0	33,609	0	0 / 62
lifelines	4	0	0	0	87,322	0	1 / 4
pingouin	22	0	0	0	24,515	0	8 / 22
probfsm	0	0	0	0	2,943	0	—
ruptures	0	0	0	0	147	0	—

What the table says. Three things.

First, **the two tools never touch the same line**. Across all 5 repositories the overlap is 0 out of 148,536 of our sites and 88 of its candidates. This is not an accident of sampling. Our wrong-constant operator only edits a literal with a decimal point in it, so it cannot reach an integer seed or an integer fold count, and no repository lists a requirements file among the source files we mutate. The check is still worth running: on pingouin the two tools edit thirteen of the same *files*, just never the same line.

Second, **a tool built for training pipelines finds little in a research compendium**. On 2 of the five — probfsm and ruptures — it finds nothing at all, because neither splits data, sets a seed, or pins a fold count. That is a fact about these repositories, not a verdict on the tool, and it is exactly why a rate comparison would be meaningless here.

Third, and this one goes against us: **its mutations are sometimes caught**. Of the 88 we ran, 9 failed the target's gate, as recorded. Pingouin's suite caught 8 of 22 reseeds; that is the evidence for this sentence. lifelines' gate failed on 1 of 4. We keep that count as recorded and do not read it as a detection: lifelines' clean gate is stochastic (App. H), the tool validates the clean baseline once and runs each mutant once, and the stored record for that mutant is an exit status of 1 with a generic reason, no failing-test trace and no repeated control. Whether the mutation or the flakiness failed that run is unresolved in both directions. The 9 of 88 aggregate includes that one run. The mechanism we can name is Pingouin's, and it is the same one we report for our own operators: the reseed dies where a test pins a number — `test_bayesian`, `test_correlation`, `test_effsize`, `test_multivariate` — and survives where a test only checks a shape — `test_plotting`, `test_regression`, `test_utils`. Our own accounting never credited these suites with constraining a seed, because we never injected one. That is a gap in our coverage, found by running somebody else's tool, and it is the reason to run one.

Where our own repository lands. All 62 of its candidates on System A are dependency pins in `requirements-lock.txt`, and our verifier caught 0 of them. It does not rebuild the environment, so relaxing a pin changes nothing it looks at. We report that as a hole in our own gate rather than a limit of the operator.

W THE LIMITS, IN FULL

This is the argument §6 points to, unabridged.

The honest reading depends on the limits, so they go here rather than in a closing paragraph. The first one is the largest, so it goes first.

We can prove that only 3 of our 19 missed constants would have changed anything. A missed injection is only interesting if the injected defect mattered, and reading the tested gate's own output cannot tell us whether it did. So we built a probe that never reads the gate's verdict: call the project's own worked example, compare the value it returns, never look at the gate (App. I). Under the rule we locked in advance it witnesses 0 of the 19. A post-hoc rule that also reads the project's documentation witnesses 3. For the rest we cannot tell: in 9 cases the probe cannot see a *tenfold*

change either, and in 3 the mutated line never ran. A probe that was built is not a probe that can see its constant; App. I gives the split. The miss counts in this paper are what they are; what they support is narrower than a reader might assume, and this is the sentence that says so.

Our own release shipped the defect this paper is about, and the paper's own catalogue names it. A reviewer said the release failed a directly relevant check. It did, and the cause is worth printing. One generator recovered a historical line count by reading git history. Extracted from the archive there is no git history, so every lookup failed, the generator wrote a 0 and logged a warning, and the released file no longer matched what its own generator produced — the archive failed step 1 of its own twenty-one-step verifier. That is operator `D6`'s shape exactly: a check that goes quiet instead of failing when the thing it reads disappears. We found it the same way we find them in other people's repositories — by running the gate somewhere the inputs are missing. The fix ships the historical counts as data and raises instead of writing a zero when neither git nor the snapshot is there. What the archive checks without git is narrower than a full recount, so we state it exactly. The regeneration step reruns every generator and compares every JSON and TeX file it writes byte for byte. The figure-toolchain manifest is compared field by field; its matplotlib version is the one field exempt. The four figure PDFs are byte-compared only when the reader's matplotlib is the version that drew them; on any other version the check says so and does not count them. Two scorers are rerun on their shipped inputs and their outputs compared byte for byte. The two chronology checks that read git history skip outside git and print that they skipped. The 3,017 lines and 37 refusal sites are archived counts, taken from the eight checker files at the injection commit and stored in `at_injection_snapshot.json` with the commit hash and one entry per file; the archive ships the counts and the file names, not the eight historical files, so a reader with the archive alone cannot recount them. Before release we recounted them from the pinned commit in git, and all eight per-file entries match. We report this here rather than quietly repairing it, because a paper that measures whether other people's checks can fail should say when its own could not.

And 10 of our 12 wrong-analysis misses could not have been caught. The third operator, checked the same way. `np.sum` and `np.nansum` agree on any array without a NaN, so a swap between them is only a defect if a NaN turns up. We measured it: of 8 such swaps, 5 executed, all 5 returned the same value on every observed call, and 0 ever saw a NaN (App. K). Across all 16 scored sites, 7 never differed on an observed call and 3 never ran. Counting only sites that demonstrably change a value, the record reads 4 caught and 2 missed rather than 4 and 12. Every catch survives that filter; the misses are what shrink.

And every one of our 11 decidable weakened assertions was redundant. The same question, asked of the other operator we lean on. Weakening an assertion cannot open a hole unless the assertion was the thing holding it shut. So we paired each weakened assertion with a defect its own test exercises and ran the gate three times: perturb only, weaken only, both (App. J). Of 20 sites we could decide 11, and in every single one the gate still failed with the assertion weakened — in 6 the assertion's own test still failed, in 5 only other tests did; these are read from failing test ids, not tracebacks. **0 were necessary.** Four of the 11 are on `lifelines`, whose gate is stochastic; those four rest on one run each and are kept as recorded, not read, so we report 7 on the four targets. The miss count is what it was; what it means is that for the one perturbation we could pair with each, the gate did not depend on the weakened assertion. The 9 unpaired sites, and every other defect class, are unresolved.

Every rate here is catalogue-weighted, and the catalogue is finite. Our catalogue still cannot express a wrong formula whose every call is right, and an operator we did not write could be easier or harder to catch than the ones we did, so incompleteness gives a "nothing caught this" count no direction. The public repositories showed one direction in practice: our original four under-represented what real suites catch (App. G).

The catalogue and one gate share an author. The deepest limitation, and we cannot argue it away. The mitigation is chronological: 8 of the twelve classes were found *after* the gates existed, and the gates missed them then. So they are not classes the gates were built around. Pingouin and Ruptures unconnected to us show the same three blind spots; ten more show the prose-number one

in all ten and the weakened-assertion one in the 6 where our rule drew a site, and three of the ten catch the third.

The overall catch rate is catalogue-weighted, so we do not lead with it. Three more prose operators would move it with nothing changing about the gate.

Three more limits, and where they are argued. A changed byte is not automatically a changed result, and we classify the perturbed constants only after seeing whether the gate fired. Clean re-runs bound flakiness at 0.14 and sample no benign edit. Our own checks have gaps we enumerate rather than discover one review at a time: 20 of 22 surfaces are gated, and 100 prose numbers are right today with nothing checking them. All three are argued in full in App. H.

17 repositories is a finite design, and two are ours. The frame is also narrower than the question: it requires a CI test runner, so every public gate here is `pytest` and a compendium like System A could not enter it. All but those two rebuild from a recorded URL and commit; System A ships whole, System B is private. The public ones came from one frame — research software with a JOSS paper whose suite passes here in under fifteen minutes. That many cases license the bounds we report, no more.

What we could settle, and what we could not. Repositories with no author in common now exist (App. F), and operators for a wrong analysis and a deleted step (App. G). A pre-registered human baseline does not. The run of the overlapping study's catalogue now does: App. V reports it. The two tools share 0 sites, and 9 of the 88 runs with its operators ended in a failed gate: eight on Pingouin, whose gate our own accounting had credited with nothing of the kind, and one on `lifelines`, whose gate is stochastic and whose single failed run we do not attribute either way (App. V). The Pingouin eight are a hole in our coverage, and it took somebody else's tool to find it.

X Reproducibility, in full

Every number this paper *pins* is produced by a script in the supplement and re-derived by one command, `paper2/verify_paper2.sh`, which regenerates the figures from the released result files, rebuilds the PDF, checks the build had no errors, no box overflowing its page by more than 5pt, and fits the page limit, recomputes each pinned quantity from its artifact, and runs an anonymity scan. It does not check the numbers nobody pinned; App. H says how many those are. Two of its steps guard the release itself rather than the paper. One regenerates every derived file we ship — the result records, the generated tables and macros, and the figures — and fails if a single byte comes back different, so a file that was corrected by hand instead of in its generator cannot pass. Two things need an exception, and each exception is cut down to exactly what needs it. The figures' bytes depend on the drawing library as well as on the data, so they are compared only when the reader's matplotlib is the version that drew them, and the check says which case it is instead of passing quietly. One small file records that version; it is compared field by field, with only the version string exempt. An earlier version exempted that whole file and still counted it as checked, so editing its text by hand passed — an exemption wider than its reason, which is the failure this paper is about. The other opens all 42 released run logs, re-hashes them, and reads each verdict back out of the bytes.

The supplement ships the injection harness, the operator catalogue, every target description, every protocol with its amendments, and every raw result file. Target descriptions carry no machine paths: roots are environment variables, documented in `gatecheck/targets/README.md`. Of the 17 gated repositories, all but our two are public. Each public description records the clone URL, the exact commit and the environment, so a reader can rebuild it from scratch; the 13 frame-sampled ones carry theirs in `data/e94_frame` and `data/e98_frame_extension`. Figures are drawn from the released files by `paper2/make_figures.py`; no figure number is typed by hand.

Two things a reader cannot reproduce independently, said here rather than left to be discovered: System B is a private repository, and System A is the repository this method was built in, released as a concurrent submission's supplement rather than this one's. Both are marked in their `provenance` blocks. lifelines is shipped as a withdrawn target with its diagnosis, because what happened to it is a result (App. H, "The withdrawn fifth target, in full").

A bounded fresh replay of the central defect. Everything above re-derives numbers from shipped records; none of it re-runs a target. For the central case of §4.3 we also re-ran the target. From the pinned CSV_Wrangling checkout, after checking its commit, its clean status and the hashes of all eight files E121 fixed, we ran 3 runs with E121's unedited modules: a clean run, the defect D and its clean pair. Every output was redirected with E121's own `E121_DATA` and `E121_WORK` settings into `data/review_20260924_gc/replay_e121/`, the way E122 and E124 ran; nothing was written into E121's data directory. All 8 of the 8 facts we checked reproduce: the accuracy is 97 on the clean runs and 98 after D; 118 claim values change; `make output && git diff` exits 0 on D; every arm's label is the shipped one; the claim files are byte-identical; `git diff` names the same paths; and all 32 of the 32 files hashed after the build, the figure PDF included, equal the shipped runs. The frozen repairs rebuilt for the replay are byte-identical to the shipped ones. This is one replay of one case, on our machine, by us; it is not an independent reproduction. The comparison is `data/review_20260924_gc/replay_e121_compare.json`.

What replays offline, at three levels, and what does not. The 153 inspected compendium files ship with their hashes: a reader can re-hash them and re-run the marker detector, but the whole-tree counts (files per tree, scripts not read) are archived records, not reconstructed from that subset. The nine drawn prose sites ship with their full source files: a reader can re-hash each file, re-slice the recorded span, replay the historical first-substring edit and the corrected exact-span edit or refusal, and the packager runs that check inside its own extraction (`experiments/review_20260915/check_e114_site_replay.py`). The E116 call records ship as JSONL: every observed call was compared during the run, and the all-call aggregate counts (2,085 comparisons across 6 final aggregate records) and every site label can be re-derived from the shipped records; the per-call examples are capped at 380, so individual call values are not all retained, and none of this is a fresh run of any target suite. Two probe files carry two hashes on purpose: the one named `as_run` is the historical probe the withdrawn E96 reading used, and the corrected run's records match the current probe; a reader choosing by pathname should choose by hash.