EVIDENCE BEFORE ASSERTION: CALIBRATED LLM-ASSISTED VERIFICATION OF CODE CHANGES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models can explain code changes fluently, but fluency is a poor substitute for evidence when a reviewer asks whether a route is affected, whether a test actually proves the changed behavior, or whether an unexecuted check is safe to ignore. We present an LLM-assisted verifier for pull requests built around an explicit epistemic contract: a verification claim may be reported as established only when it is grounded in inspectable structural or execution evidence; LLM outputs may propose missing facts, but cannot silently promote them to proof. The system represents behavioral claims as verification obligations, attaches evidence at the strongest scope that can be demonstrated, and degrades attribution rather than confidence when exact ownership cannot be established. We study this design through controlled pull requests on eight real open-source repositories spanning Python, TypeScript, Java, Go, and Rust. The study surfaced seven reproducible failure classes at the boundaries between structural analysis, LLM recovery, test mapping, execution, and reporting. Two cases are particularly informative. First, an apparently intermittent call-resolution failure was shown, after isolating the deterministic subsystem, to be a fully reproducible $0/20$ structural defect masked by occasional LLM recovery; after three generic fixes, the same harness resolved $20/20$ twice and the full pipeline resolved the path in $3/3$ runs. Second, whole-suite execution produced 13 passing and 4 unrelated environment failures for a narrowly scoped change; after wiring mapped-test execution and improving assertion recognition, the system executed exactly one relevant test and attributed a pass to the changed behavior without running the unrelated suite. These results are not a benchmark-scale accuracy claim. They are an empirical systems study of a recurring problem in hybrid LLM tools: correctness failures concentrate at evidence boundaries, and reliable behavior requires explicit provenance, scope-aware evidence ownership, and a calibrated handoff when the environment cannot complete verification.

1 INTRODUCTION

LLMs are increasingly used to review pull requests, summarize diffs, and reason about repository-scale changes. The attractive property is obvious: an LLM can interpret intent, connect semantically related code, and explain a change in language useful to a human reviewer. The dangerous property is equally obvious: the same model can produce a plausible claim about a repository that is structurally false. A statement such as "this endpoint is affected" or "this test verifies the new validation" is useful only if the system can answer the follow-up question: *how do you know?*

The verification setting is therefore different from ordinary code summarization. A wrong summary may be inconvenient; a wrong verification claim can cause a reviewer to merge a change under false confidence. Conversely, a verifier that marks every uncertain neighboring behavior as a failure is technically conservative but practically uninformative. The central problem is not simply hallucination reduction. It is *calibration of assertion*: deciding which claims may be elevated to established facts, which claims should remain hypotheses, and what to tell the developer when the verifier cannot execute the relevant evidence in its own environment.

We present a system for evidence-grounded verification of code changes. Its design is based on four principles.

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.

1. **Facts and interpretations are different objects.** Structural relations, executed tests, and LLM inferences retain distinct provenance throughout the pipeline.
2. **Evidence has scope.** Evidence proven for a changed symbol is not silently promoted to an exact route-level claim. When ownership is uncertain, attribution weakens before confidence does.
3. **LLMs recover; they do not certify themselves.** The LLM may search for missing paths or tests, but recovered claims must survive structural checks before they can become established.
4. **Verification is allowed to stop.** If a mapped test requires a database, native dependency, secret, or toolchain the verifier cannot reproduce, the correct output is a handoff to the developer's environment, not a fabricated pass and not necessarily a red failure state.

The paper focuses on the consequences of enforcing this contract in a real multi-language system. We evaluate controlled changes on forks of eight real open-source repositories spanning five languages. Rather than presenting the deployment only as a success rate, we use failures as instrumentation: each discovered inconsistency was reduced to a reproducible cause, addressed generically, and guarded by a regression test targeting the identified failure mode. The resulting defect taxonomy reveals that the hardest bugs were not "the LLM was wrong" in isolation. They occurred where one evidence type was converted into another: a same-named symbol in another file acquired the changed symbol's identity; recovered evidence updated an aggregate count but not the owning obligation; a complete mapped-test execution mechanism existed but was bypassed by the pipeline; a language-specific test recognizer silently collapsed annotated methods; and an LLM fallback masked a deterministic call-resolution defect.

Our contributions are therefore threefold:

- an **evidence contract** for hybrid code verification, including scope-aware evidence ownership and a rule that LLM recovery can propose facts without laundering them into proof;
- a **multi-language implementation** that composes structural analysis, obligation derivation, test mapping, targeted execution, and verified LLM recovery while preserving provenance end to end; and
- an **empirical failure study** across eight real repositories, including a controlled isolation experiment ($0/20 \rightarrow 20/20$, repeated, with $3/3$ end-to-end confirmation) and a targeted-execution case showing how unrelated suite failures can contaminate verification unless execution is attributed at the same granularity as the claim.

We deliberately do not claim benchmark-scale superiority over LLM-only or static-analysis baselines. The present study asks a more foundational question: what engineering and epistemic constraints are required before a hybrid verifier's claims are trustworthy enough to be shown to a human reviewer?

2 Problem Formulation: Claims, Evidence, and Scope

We model verification as a relation between *claims*, *evidence*, and the *scope* at which evidence can be attributed. Let C denote candidate behavioral claims induced by a code change, E the evidence items available to the verifier, and S the set of attribution scopes. A claim can be as specific as "`POST /orders` rejects a quantity greater than available stock" or as broad as "the changed service method has regression coverage." The verifier's task is not only to judge plausibility, but to preserve why a claim is believed and how precisely that belief can be attached to the changed system.

2.1 Verification obligations

Each falsifiable behavioral claim is represented as a *verification obligation*

$$o = (b, q, s),$$

where b identifies the behavior under consideration, q is the property to establish (for example, rejection, response status, side effect, or state transition), and $s \in S$ is the strongest currently established owner. An obligation also carries its supporting evidence and current verification state. This representation is intentionally narrower than free-form code review: an observation can be useful without being mechanically provable, while an obligation is a claim for which the system must track evidence and an explicit state.

2.2 ADMISSIBLE EVIDENCE AND PROMOTION

Let $A(e) \in \{0, 1\}$ indicate whether evidence item e is admissible as proof, and let $\text{supports}(e, c)$ denote that e directly supports claim c. The core assertion rule is

$$\text{Established}(c) \iff \exists e \in E : \; A(e) = 1 \wedge \text{supports}(e, c).$$

Structural facts derived from repository analysis and execution facts produced by running a test can be admissible when their identity and ownership checks succeed. An LLM output is different: by itself it is a proposal $\hat{c}$, not proof,

$$\text{LLM}(x) \to \hat{c}, \qquad A(\hat{c}) = 0.$$

Only an independent structural or source-backed check may promote that proposal into admissible evidence. Generation may therefore expand the search space, but generation cannot certify itself.

2.3 EVIDENCE SCOPE AS A PARTIAL ORDER

Evidence is attached at the strongest scope the system can actually establish. We use the attribution order

$$\text{exact flow / obligation} \succ \text{changed symbol} \succ \text{change-level}.$$

For evidence e, let $\text{owned}(e, s)$ mean that ownership of e at scope s is established. The attachment rule is

$$s^*(e) = \max\{s \in S : \text{owned}(e, s)\}.$$

If a regression test clearly exercises a changed method but route composition is unresolved, the test remains useful at symbol scope; it is neither discarded nor promoted to an exact endpoint. This yields the practical invariant:

> *Evidence may move downward in attribution specificity without moving upward in confidence.*

Failure to prove exact ownership therefore weakens attribution before it erases evidence or invents precision.

2.4 THREE-WAY VERIFICATION SEMANTICS

For an obligation o, the verifier records

$$V(o) \in \{\text{VERIFIED}, \text{CONTRADICTED}, \text{UNRESOLVED}\}.$$

A passing admissible test can establish VERIFIED; a failing admissible test can establish CONTRADICTED. Missing execution does neither. In particular,

$$\neg\, \text{executed}(t) \not\Rightarrow V(o) = \text{CONTRADICTED}.$$

This distinction matters in real repositories, where the relevant test may require a database, native dependency, secret, or toolchain unavailable to the verifier. In that case the correct state is unresolved with an explicit handoff, not a fabricated pass and not a failure attributed to the changed behavior.

2.5 A RUNNING EXAMPLE

Consider a small pull request in an order service. The change adds an inventory check so that requests exceeding available stock are rejected. The diff states which lines changed; the verifier instead constructs a sequence of claims and evidence:

Stage	Example
Change	reject an order when requested quantity exceeds available stock
Affected path	`POST /orders` → `create_order`
Obligation	insufficient stock ⇒ HTTP 400 with no successful order creation
Test evidence	a regression test exercises the route and asserts rejection; execution status is tracked separately
Report	established path, mapped evidence, unresolved execution if the test was not run, plus independent review observations

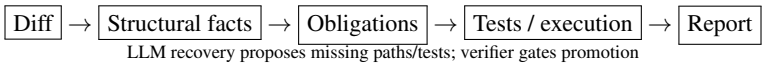

Figure 1: High-level pipeline. LLM recovery augments structural analysis but does not bypass provenance or verification checks.

The example illustrates the separation enforced throughout the paper: a path can be established even when execution remains unresolved; a discovered test can be relevant without yet proving the behavior; and a code-review observation can be useful without being promoted into a verification fact. Appendix B shows a richer anonymized end-to-end report drawn from one of the real repository cases.

3 System Design

Figure 1 summarizes the system. The LLM is deliberately downstream of deterministic analysis and upstream of a verifier, rather than being the sole authority.

3.1 Structural facts and normalized boundaries

The structural layer indexes files, symbols, imports, route declarations, and available call edges. Language- and framework-specific syntax is normalized into generic concepts such as boundary, handler, call edge, and test case. A boundary can be an HTTP route, gRPC method, GraphQL operation, queue consumer, event handler, cron job, or CLI command. Framework recognizers may discover these facts, but they do not directly construct final impact claims; those are composed in a common representation.

Two implementation modes supply structural facts. A native backend performs lightweight language-aware extraction and bounded on-demand following. A graph-backed backend can supply a richer call graph. Some relations remain difficult for either representation, especially dynamic or typed indirection such as constructor-injected service fields. For these cases, the verifier adds conservative synthetic bridges only when identity is unambiguous. The failure study in Section 4 shows why these bridges must obey the same evidence rules as ordinary edges.

3.2 From paths to obligations

Once a changed symbol is associated with an affected path, the verifier derives obligations from changed conditional behavior, route contracts, and observable sinks. Multiple route aliases sharing the same handler are canonicalized to a single obligation identity, preventing alias fan-out from multiplying verification requirements.

A useful distinction is between *change-scoped* obligations and *route-neighborhood* obligations. The former correspond to behavior introduced or modified by the diff; the latter describe relevant surrounding behavior on the same path. Both are useful, but they should not be presented as if they have the same decision significance. Section 4.7 returns to this distinction.

3.3 Test mapping is semantic, not only structural

A test that invokes a changed function does not necessarily verify the changed behavior. The verifier therefore separates three evidence tiers:

- **Tier A: direct verification.** The test exercises the relevant route or changed symbol and asserts the obligation's behavior, such as a rejection, thrown error, status code, or named side effect.
- **Tier B: supporting behavioral evidence.** The test asserts a relevant sink or side-effect token but does not directly prove the full obligation.
- **Tier C: flow exercise only.** The test reaches the flow but contains no assertion that proves the obligation. It is useful context but cannot satisfy the claim.

The mapping vocabulary is intentionally conservative. During evaluation, this conservatism exposed recognizer gaps—for example, a Java method annotated with `@Test` but lacking "test" in its name, and a Jest validation test proving rejection with `.rejects` rather than a literal numeric status code. These became generic recognizer fixes rather than repository-specific exceptions.

3.4 Targeted execution and environmental handoff

When mapped tests are executable, the verifier runs them individually and attaches pass/fail results to the obligations they support. If no mapped test can be invoked individually, it may fall back to a repository-level test command, but the fallback is explicit. If only some mapped tests are executable, the system reports the mixed state rather than replacing missing evidence with a suite-wide result.

This matters because repository test environments are not universally reproducible. A verifier may encounter native dependencies incompatible with the runner, a database unavailable in the sandbox, an old JDK requirement, or a secret intentionally absent from pull-request CI. Such an environment failure is not evidence that the changed behavior failed. The correct output is a handoff: the relevant test was found; execution could not be completed here; run the named test in the repository's normal environment or CI.

3.5 LLM recovery as proposal under verification

The LLM recovery stage runs when the deterministic pipeline leaves a meaningful gap. It receives bounded repository context and may propose a missing boundary, path, or relevant test. The proposal is parsed into a typed candidate and subjected to structural checks before it can be merged. For example, a zero-hop candidate claiming that a decorated entrypoint is itself the changed target must agree with the changed file and symbol identity; a same-named method in another file cannot be substituted merely because the bare names match.

The design goal is not to eliminate LLM uncertainty. It is to prevent uncertainty from being silently converted into deterministic-looking evidence.

4 Empirical Study

4.1 Corpus and protocol

We evaluated the verifier through controlled pull requests on forks of eight real open-source repositories spanning five languages (Table 1). The changes are intentionally small enough that expected behavior can be manually inspected, while covering distinct backend structures: REST handlers, service-layer validation, route composition, database-facing logic, and framework-specific test conventions. Some cases were derived from real upstream changes; others were constructed as calibration changes against real repository code. All evaluation occurred on forks; no upstream project was modified.

The study is defect-driven rather than benchmark-scale. When a run produced an inconsistent or incorrect claim, we traced the responsible layer, reduced the failure to a reproducible test or harness, implemented a generic fix, and reran the relevant case with a regression test targeting the identified failure mode. The final study snapshot incorporates all fixes described in this paper. Final acceptance runs used `openai:gpt-6-sol` with a fixed configuration across the reported cases.

The study is organized around three research questions:

- **RQ1:** Where do correctness failures arise when structural analysis, LLM recovery, test mapping, execution, and reporting are composed into one verifier?
- **RQ2:** Which evidence-boundary rules prevent plausible but insufficiently grounded information from becoming established verification claims?
- **RQ3:** When a failure is isolated and repaired generically, does the relevant subsystem and end-to-end pipeline exhibit the expected before/after behavior?

Table 1: Evaluation corpus. Controlled changes on real open-source repositories; identities are anonymized for double-blind review.

Repo.	Lang.	Domain	Evaluated change
R1	Python	monitoring	reject uppercase identifier input
R2	Python	REST API	cap an aggregation input
R3	Python	inference API	normalize model-name input
R4	TypeScript	feature flags	reject duplicate parameters
R5	TypeScript	REST API	reject non-positive numeric input
R6	Java	service API	ignore blank/duplicate user identifiers
R7	Go	banking API	reject amount above available balance
R8	Rust	web framework	reject zero-length identifier

Table 2: Seven reproducible failure classes. The rightmost column states the verification risk, not merely the implementation symptom.

ID	Observed failure	Boundary	Verification risk
D1	provider rejects hard-coded sampling temperature	configuration → LLM client	recovery becomes unavailable for reasons unrelated to the code change
D2	unrelated same-named symbol receives recovered evidence	symbol identity → ownership	false established impact through cross-file identity collision
D3	"tests verify" coexists with "no relevant test"	recovery → obligation state	contradictory report from stale evidence ownership
D4	unrelated suite failures contaminate narrow verification	mapped evidence → execution	unrelated environment failures appear to invalidate the changed behavior
D5	annotated Java test collapses to file-level pseudo-test	framework syntax → test identity	direct evidence becomes undiscoverable or too coarsely attributed
D6	rejected-error assertion is treated as flow exercise	assertion → evidence tier	direct behavioral proof is understated
D7	constructor-injected service call never resolves	synthetic edge → call resolution	affected path becomes a false negative, occasionally masked by LLM recovery

4.2 Ground truth and adjudication

The evaluation oracle is external to the LLM. For each controlled change, we inspect the diff, relevant source, existing tests, and executable behavior where the repository environment permits it. Structural claims are adjudicated against source and extracted relations; test claims against test bodies and, when runnable, execution artifacts. When exact ownership cannot be independently established, the reference state is *unresolved at that scope* rather than forced positive or negative. LLM-generated text is never used as ground truth for another LLM-generated claim. Appendix A records the per-case oracle categories and execution-state interpretation used in the study.

4.3 Failure taxonomy: bugs at evidence boundaries

Table 2 summarizes seven failure classes uncovered by the study. The pattern is more important than the count: most defects occurred where one representation was converted into another—identity to evidence ownership, recovered evidence to obligation status, mapped tests to execution, framework syntax to canonical test cases, or synthetic structural edges to call resolution.

D2 is illustrative of the epistemic risk. An LLM recovery candidate named a real decorated method in one file. The diff changed a different method with the same bare name in another file. A later merge step matched only the bare name and thereby "laundered" a true structural fact about the wrong symbol into an established impact claim for the diff. The fix was not better prompting; it was a stronger identity invariant requiring file agreement or already-disambiguated qualified identity.

D3 exposed the dual problem on the reporting side. Recovered symbol-scoped tests incremented aggregate evidence counts, but the owning obligation retained its stale pre-recovery reason, allowing the same report to say both "two tests verify this behavior" and "no relevant test found." A reconciliation pass now applies the same evidence-scope hierarchy to both aggregate and per-obligation state.

4.4 CASE STUDY I: WHEN LLM RECOVERY MASKS A DETERMINISTIC DEFECT

The most instructive failure involved a TypeScript controller calling a constructor-injected service field: `this.petService.create(...)`. In repeated end-to-end runs, the path appeared to resolve only occasionally. Because LLM recovery is stochastic and sometimes rediscovered the behavior independently, the initial interpretation was that the structural result itself might be nondeterministic.

We removed the LLM from the experiment entirely and built a direct harness over the deterministic structural index, synthetic member-call bridge, and call resolver. Across 20 repetitions against the identical checkout, the result was **0/20 resolved**. The bug was therefore not intermittent: the structural path failed every time, while LLM recovery occasionally masked the failure in the full pipeline.

Three compounding defects explained the result. First, the bridge expected a constructor symbol, but the real language extractor deliberately did not emit constructors as ordinary methods; the bridge's hand-written tests had fabricated an impossible index shape. Second, the otherwise backend-agnostic bridge was gated behind a capability check for a graph backend, preventing its edges from reaching the native path. Third, after those two fixes, the fallback lookup still keyed edges by a bare method name while the native call extractor produced a receiver-qualified key such as `petService.create`.

After fixing all three, the direct harness resolved **20/20**; repeating the experiment produced another **20/20** $(40/40$ total). Three full end-to-end runs with LLM recovery enabled then resolved the same path **3/3**. The methodological lesson is broader than this repository: in hybrid systems, an LLM fallback can improve apparent robustness while simultaneously hiding a deterministic defect. Fault isolation therefore requires evaluating the deterministic subsystem directly, not inferring its behavior from end-to-end variance.

4.5 CASE STUDY II: VERIFICATION GRANULARITY MUST MATCH EVIDENCE GRANULARITY

A second case exposed a mismatch between what the verifier knew and what it executed. For the same TypeScript change, the system discovered a direct regression test for the new invalid-age behavior. Yet the execution pipeline ignored the mapped test mechanism and ran the repository's entire Jest suite. The result was **13 passed, 4 failed**; the four failures were unrelated environment problems involving SQLite/database setup. A developer looking only at the top-level execution result would see a failed verification even though the changed behavior had a relevant unit test.

Inspection showed that a complete per-test execution implementation already existed but had no callers in the verification pipeline. We wired it into a three-way policy: execute mapped tests when possible; fall back to the whole suite only when none can be invoked individually; and preserve per-obligation outcomes when executability is mixed. This change then exposed a second gap: the relevant Jest test asserted rejection with `.rejects.toMatchObject(...)` and checked that persistence was not called, but the validation mapper recognized only a literal numeric HTTP status. A generic rejection-assertion recognizer was added for patterns such as `.rejects`, `toThrow`, `assertThrows`, `assertRaises`, and `pytest.raises`, while retaining a negative control in which mere invocation remains supporting evidence only.

In the final acceptance run, the verifier mapped exactly the invalid-age regression test, executed **1** mapped test, obtained an attributable **pass**, and did **not** execute the unrelated regression suite. This case supports a simple design rule: if evidence is owned per obligation, execution results should be attributed at the same granularity. Suite-level success or failure is too coarse to certify a specific behavioral claim.

4.6 INTERVENTION EVIDENCE

The study is not a component ablation in the conventional benchmark sense, but two failures admit controlled before/after interventions because the responsible subsystem can be isolated. Table 3 summarizes these measurements. They support a causal engineering claim narrower than overall accuracy: repairing the identified boundary defect changes the measured subsystem behavior in the expected direction while holding the repository checkout and target relation fixed.

Table 3: Controlled before/after evidence for the two principal interventions.

Intervention	Before	After
Member-call resolution	deterministic harness resolves target $0/20$	resolves $20/20$ twice; full pipeline $3/3$
Execution granularity	whole suite: 13 pass, 4 unrelated environment failures	exactly 1 mapped relevant test executed and passed; unrelated suite not run
Assertion recognition	rejection assertion treated as non-direct evidence	rejection/throw pattern credited to the validation obligation

4.7 Reporting semantics: change evidence is not route completeness

The deployment also revealed a presentation problem. The underlying verifier conservatively tracks obligations on the affected route, including pre-existing behavior that the diff did not modify. Consequently, a narrowly scoped change could be directly verified while the route still contained unrelated, unexecuted obligations. Treating both as a single top-level severity signal caused qualitatively different PRs to receive similarly cautionary output.

We did not change the core verdict semantics immediately before submission. Instead, the renderer was changed to expose the distinction already present in the data: what concerns *this change* versus what is merely *also on this route*. The report now uses analysis-oriented wording when no blocking failure has been found, separates change-scoped verification from surrounding checks, and frames unexecutable tests as handoffs to the repository's own environment. A failed relevant test or blocking review finding still receives strong "action required" language.

This is a product-facing change, not a human-subject study, and we do not claim measured improvements in developer trust. The empirical observation is narrower: one aggregated state had insufficient discriminative value across the evaluated PRs, while the underlying canonical model already contained the information needed to separate change-scoped and route-scoped incompleteness.

5 Related Work

Repository-level agents and semantic reasoning. SWE-bench (Jimenez et al., 2024), SWE-agent (Yang et al., 2024), and AutoCodeRover (Zhang et al., 2024) evaluate systems that navigate repositories to produce or repair code, while CodeSense (Roy et al., 2026) targets semantic reasoning over real-world code. SWE-agent also shows that interface design materially affects agent behavior. Our setting is complementary: the patch already exists, and the system must decide which impact and verification claims are sufficiently grounded to show a reviewer.

Tests and executable evidence. SWT-Bench (Mündler et al., 2024) studies whether code agents can generate tests that reproduce real issues, and EvalPlus (Liu et al., 2023) shows that insufficient tests can overstate functional correctness. Regression-test-selection work asks which tests should run after a change (Rothermel & Harrold, 1997; Memon et al., 2017); our verifier additionally asks what a selected test actually proves. Repo2Run (Hu et al., 2025) shows that reconstructing executable repository environments is itself a systems problem, supporting our distinction between behavioral contradiction and an unresolved execution handoff.

External evidence and structural grounding. LLM self-correction without external feedback can be unreliable (Huang et al., 2024). Classical program slicing and call-graph analysis provide structural reachability (Weiser, 1984; Lhoták & Hendren, 2003), while retrieval-augmented and neuro-symbolic systems motivate combining learned inference with external evidence or symbolic constraints (Lewis et al., 2020; Garcez & Lamb, 2020). Hallucination in generative models is well documented (Ji et al., 2023), and automated code-review work studies generated review comments and issue detection (Tufano et al., 2021). Our contribution is a typed evidence contract: model output may guide search, but promotion to an established claim requires independently attributable evidence, and the system may stop at unresolved rather than invent precision.

6 DISCUSSION

6.1 WHAT THE STUDY SUGGESTS ABOUT HYBRID LLM SYSTEMS

Three conclusions recur across otherwise different failures.

First, **LLM quality is not the only source of epistemic error**. D2, D3, D4, and D7 all produced misleading end-user states even when the LLM output itself was reasonable. The failure occurred in identity, ownership, reconciliation, execution, or composition around the model. Improving the model would not have repaired these bugs.

Second, **fallback intelligence can hide deterministic defects**. In Case Study I, the LLM was useful precisely because it sometimes reconstructed a missing path. That robustness made the deterministic bug look intermittent until the LLM was removed from the measurement path. Hybrid-system evaluation should therefore include component isolation, not only repeated end-to-end trials.

Third, **the unit of evidence should match the unit of decision**. If the decision is whether a particular changed behavior is verified, a whole-suite result is often the wrong unit. Likewise, if evidence is only symbol-scoped, a route-level claim is the wrong unit. Much of the verifier's architecture can be understood as preserving this alignment across representations.

6.2 LIMITATIONS AND THREATS TO VALIDITY

Eight repositories provide cross-language engineering coverage, not a statistical estimate of code-review accuracy. The controlled cases may underrepresent large migrations, generated code, metaprogramming, and highly dynamic dispatch. The study is mechanism-focused rather than comparative: it does not establish superiority over LLM-only, static-analysis-only, or commercial systems in recall, precision, productivity, or defect detection.

LLM recovery is model- and prompt-dependent; we hold the final acceptance configuration fixed but do not characterize sensitivity across providers. The strongest quantitative result therefore comes from the deterministic harness with the LLM removed, while the $3/3$ end-to-end runs are confirmation only. Test mapping remains framework-dependent and intentionally favors calibrated omission over unrelated evidence. Execution can also require toolchains, databases, secrets, or services unavailable to a hosted verifier, requiring explicit handoff to repository CI. Finally, some coupling remains between route completeness and top-level verdict logic; fully orthogonal impact, confidence, and verification axes remain future work.

7 CONCLUSION

We introduced an evidence-grounded approach to LLM-assisted code-change verification in which the model may help search and interpret, but cannot silently convert a plausible repository claim into proof. The implementation combines structural facts, scope-aware obligations, semantic test mapping, targeted execution, and verified LLM recovery. An empirical study across eight real repositories and five languages revealed that the most consequential defects clustered at evidence boundaries rather than exclusively inside the model. Isolating one hybrid failure changed the diagnosis from apparent stochasticity to a deterministic $0/20$ bug and, after repair, to $20/20$ repeated structural resolution and $3/3$ end-to-end confirmation. A second case reduced a noisy 13-pass/4-fail whole-suite result to one attributable mapped-test pass. These results motivate a broader principle for hybrid developer tools: before optimizing how much an LLM can infer, define exactly what evidence is required for the system to assert that an inference is true.

REFERENCES

Artur d'Avila Garcez and Luis C. Lamb. Neurosymbolic AI: The 3rd wave. *arXiv preprint arXiv:2012.05876*, 2020.

Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2Run: Automated building executable environment for code repository at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, 2025.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 9459–9474, 2020.

Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC)*, pp. 153–169, 2003.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 233–242, 2017.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024.

Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6:173–210, 1997. Preliminary version at ICSE 1996.

Monoshi Kumar Roy, Simin Chen, Benjamin Steenhoek, Jinjun Peng, Gail Kaiser, Baishakhi Ray, and Wei Le. CodeSense: A real-world benchmark and dataset for code semantic reasoning. In *International Conference on Learning Representations (ICLR)*, 2026.

Rosalia Tufano, Luca Pascarella, Michele Tomova, Fabio Palomba, and Gabriele Bavota. Towards automating code review activities. In *International Conference on Software Engineering (ICSE)*, pp. 163–174, 2021.

Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984. Originally presented at ICSE 1981.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024.

AI USE STATEMENT

Generative AI was used extensively throughout this work. AI coding and reasoning systems were used to generate and edit implementation code and regression tests; assist with root-cause analysis and experimental design; invoke, inspect, and interpret evaluation runs; conduct literature search and positioning; draft and edit manuscript text; and prepare the anonymized artifact and reproducibility documentation. Quantitative claims reported here were checked against source code, concrete test executions, logs, or other recorded artifacts rather than accepted from model output alone. We reviewed the resulting code, experiments, claims, citations, and manuscript and take responsibility for the final content.

ETHICS STATEMENT

Evaluation used public open-source repositories through dedicated forks. No pull request was submitted to the upstream projects as part of this study. No private or sensitive user data was used. The work does not involve human subjects.

REPRODUCIBILITY STATEMENT

The submission is accompanied by an anonymized artifact containing the final system snapshot, regression tests for the reported failure modes, and instructions for reproducing the corrected structural and mapped-test behaviors. The reported $0/20 \rightarrow 20/20$ measurements are historical results from the original controlled evaluation. The original ad-hoc pre-fix harness was not preserved; the artifact therefore provides the corresponding regression test for the corrected behavior rather than claiming exact reconstruction of that historical run. From a fresh artifact extraction, the test suite completes with 1676 passing and 4 skipped tests, with environment-dependent checks skipped rather than reported as behavioral failures.

A Adjudication Protocol and Execution Semantics

Table 4 summarizes the principal reference evidence used for each controlled case. The oracle is intentionally heterogeneous because the repositories expose different amounts of executable and structural evidence; the common rule is that no LLM output serves as its own oracle.

Table 4: Principal adjudication evidence for the eight controlled cases.

Repo.	Language	Principal reference evidence
R1	Python	source behavior and regression assertion
R2	Python	route/source inspection and repository tests
R3	Python	source normalization behavior and existing tests
R4	TypeScript	source behavior and assertion semantics
R5	TypeScript	structural path plus directly executed mapped test
R6	Java	endpoint/service source plus focused unit-test source
R7	Go	handler/service source plus repository tests
R8	Rust	route/source inspection plus repository tests

Execution evidence is interpreted at the same ownership granularity as the obligation. A relevant mapped test that runs and passes contributes positive execution evidence; if it runs and fails, it contributes contradictory evidence. A relevant test that is discovered but cannot run remains unresolved and is handed off to repository CI. A failure from an unrelated suite test or unavailable environment is not evidence against the changed obligation unless ownership of that failure is independently established. This distinction is particularly important for the targeted-execution intervention in Section 4.5.

B Worked Pull-Request Analysis

This appendix gives a richer end-to-end example drawn from repository R6. To preserve double-blind review, repository, package, class, method, route-segment, workflow, and tool-brand identifiers have been normalized while preserving the structure of the reported evidence. The underlying change removes blank and duplicate user identifiers before constructing datastore keys and processing forced session termination. The report shown here is a paper rendering of the machine-readable result, not a screenshot of the public pull request.

Analysis complete – Medium impact

Change. The session-termination service now ignores blank and duplicate user identifiers when generating datastore keys and iterating over users to terminate. A focused unit test was added for blank, null-element, whitespace-only, and duplicate inputs.

What it may affect – Established.

```
DELETE /api/session/user/kickout
  -> kickoutOnlineUser
  |- ApiResponse.success
  |- SecurityException
  `- SessionService.kickout
     ... +1 traced call
```

The service-level input filtering is relevant to the route above, but route composition is not complete enough to claim that every boundary reaching the changed symbol has been enumerated.

Test evidence.

Relevant regression test	Found
API behavior	Found, not executed
Test executed by verifier	No (execution disabled for this run)
Route coverage	Incomplete

One mapped unit-test case is associated with the affected route. The test invokes the changed service behavior with repeated identifiers, empty strings, a null element, and whitespace-only input, and verifies the resulting datastore-key operation for the retained identifiers. In this

run, however, the verifier does not promote the discovered test to execution-verified evidence because test execution is disabled.

What is still unknown. Before merge, the relevant mapped test should be executed in the repository's normal environment. In addition, unresolved route composition means the report does not claim that the displayed endpoint is the only boundary that can reach the changed service method.

Independent code review observations.

- The method assumes that the input list itself is non-null. The added test covers a null *element* inside a non-null list, not a null list argument.
- Blank filtering intentionally changes behavior for whitespace-only identifiers. The report surfaces this as a semantic boundary worth confirming rather than treating it as a verification failure.
- Parallel iteration remains in the service. This is not introduced by the filtering change, but future non-thread-safe or ordering-dependent logic inside the loop could make the behavior relevant to concurrency review.

Technical evidence. The changed production symbol is the normalized `SessionService.kickout` operation; the report keeps this identity distinct from the controller entrypoint and from the mapped test identity.

This example demonstrates the evidence contract at several levels simultaneously. The endpoint-to-handler-to-service path is reported as established because it is structurally supported. The unit test is reported as relevant because its source is inspectable, yet its status remains unexecuted because this run supplied no execution artifact. Route completeness remains unresolved rather than being inferred from the single established path. Finally, the independent review observations are intentionally not promoted into failed verification obligations: they are useful questions for a reviewer, but they do not constitute contradictory evidence for the changed behavior.

The distinction is important operationally. A conventional summary could collapse these facts into a single statement such as "the change is covered by tests" or "verification is incomplete." The structured report instead preserves four separate statements: an affected path is established; relevant source-backed test evidence exists; that test was not executed in this run; and the known structural exploration does not prove route completeness. These statements can coexist without contradiction because they answer different questions and are backed by different evidence scopes.

C Defect Details

This appendix records the concrete failure mode and generic correction for each defect in Table 2. The purpose is not to count bugs as a performance metric, but to make the empirical basis for the architectural claims inspectable.

D1: provider-parameter propagation. A class of models rejects explicit non-default sampling temperatures. One LLM client-construction site inherited a numeric default even after sibling call sites had been corrected; a subsequent audit found additional construction sites with the same latent pattern. The generic correction is to leave temperature unset and allow the provider/model default. Regression tests intercept client construction and assert the unset contract.

D2: cross-file bare-name evidence misattribution. Recovery could establish a real decorated method in one file and later attach it to an unrelated changed method in another file because both shared a bare name such as `get`. The fix removes bare-name-only promotion and requires file agreement or an already-disambiguated qualified identity. A related zero-hop collapse now also requires the candidate node to correspond to an actual changed symbol.

D3: stale obligation state after recovery. Recovered evidence could update aggregate counts while leaving an obligation's earlier "no test" reason untouched. Exact-flow attachment now updates the obligation directly; broader-scope evidence triggers a reconciliation pass that changes the stale reason without promoting the obligation beyond the evidence's scope.

D4: whole-suite-only execution. Per-test execution code already existed but was disconnected from the verification pipeline. The corrected policy executes individually mapped tests when possible,

labels whole-suite fallback explicitly when none are invocable, and records mixed executability per obligation. This change also required verdict escalation for an individually executed mapped test that fails.

D5: Java test recognition. The Java extractor required a method name containing "test", a JUnit-3-era naming assumption. Consequently, methods such as `@Test public void saveRejectsBlankUsername()` were not indexed individually. The corrected extractor recognizes JUnit 4/5 annotations while preserving naming-based compatibility.

D6: rejection assertion semantics. A validation test using a symbolic expected error and a rejected-promise assertion did not satisfy a validation obligation because the mapper looked for a literal HTTP status. The generic fix recognizes rejection/throw constructs across several common testing idioms while retaining a negative control in which plain invocation remains supporting evidence only.

D7: constructor-injected member-call resolution. The member-call bridge expected a constructor symbol that the real extractor never emits; the bridge was incorrectly gated behind a graph-backend capability; and the final lookup used a bare method key where native extraction produced a receiver-qualified key. All three defects were required to reproduce the failure. Each has an independent regression test.

D Key Acceptance Traces

D.1 Historical deterministic call-resolution experiment

The controlled experiment reported in Section 4.4 invoked the structural index, member-call bridge, and call resolver directly on a fixed repository checkout, excluding LLM recovery and command-line orchestration. The pre-fix implementation resolved the target call in 0 of 20 repetitions. The corrected implementation resolved 20 of 20, and a second independent 20-run repetition produced the same result. Three subsequent full-pipeline runs with LLM recovery enabled resolved the call in all three runs. The original ad-hoc harness used for those historical measurements was not preserved. The anonymous artifact instead contains the corresponding regression test for the corrected member-call behavior, including 20 repeated resolution checks, so that the post-fix property can be independently re-exercised without overstating reconstruction of the original pre-fix run.

D.2 Mapped-test execution acceptance

For the invalid-age TypeScript change, the final acceptance trace contained a change-scoped validation obligation, a directly mapped unit test asserting rejection, and one individually executed test with an attributable pass. The repository-level Jest suite was not executed in this acceptance run. Earlier whole-suite execution of the same repository produced 13 passing and 4 failing tests due to unrelated local environment dependencies, motivating the per-obligation execution policy.

E Reporting Contract

The user-facing report is designed to preserve the distinction between analysis, verification, and handoff. When no blocking failure is observed, the report may state that analysis is complete while separately indicating whether the changed behavior was verified. If a relevant test is found but cannot be executed in the verifier's sandbox, the report names the test and instructs the developer to run it in the repository's own environment or existing CI. Strong warning language is reserved for a failed relevant test, blocking review finding, or unresolved issue that materially affects the changed behavior. This rendering does not alter the canonical evidence or verdict computation described in the main paper.

F Reproducibility Checklist

The anonymized artifact supports the claims that can be reproduced from the final system snapshot without access to private infrastructure. It includes: (i) the verifier snapshot used for the final study,

(ii) regression tests for D1–D7, (iii) the corrected member-call regression test corresponding to the historical $0/20 \rightarrow 20/20$ experiment, (iv) instructions for the mapped-test execution acceptance behavior, and (v) reviewer-facing instructions for installation, test execution, and running the verifier against an unrelated public pull request. The original ad-hoc pre-fix call-resolution harness is not included, so the artifact does not claim exact reconstruction of that historical run. From a fresh extraction, the artifact test suite reports 1676 passed and 4 skipped tests; environment-dependent checks are skipped explicitly when their required runtime is unavailable. Repository identities used in the double-blind manuscript are mapped only within the private author record and are not required to interpret the reported failure mechanisms.