Measure the Right Thing, at the Right Grain: Three Measurement Traps in Evaluating Code-Repair Agents

João Pedro Wieland
COPPE/UFRJ
Rio de Janeiro, Brazil
jpvbwieland@cos.ufrj.br

Diego Castro
CEFET/RJ
Rio de Janeiro, Brazil
diego.castro@cefet-rj.br

Cláudia Werner
COPPE/UFRJ
Rio de Janeiro, Brazil
werner@cos.ufrj.br

Abstract

Code-repair agents get graded on whether they "fixed" a bug, but the machinery that decides that almost never gets checked. Building a tool that repairs web-accessibility problems with local LLM agents, we hit three measurement traps, and each flipped a result we were ready to publish. Trusting the agent's own report of success held the fix rate near 58% where an independent re-scan puts it near 50%, and hid the fact that 71% of the "bugs" were not bugs, just noise from our test harness. Counting violations that share a file as separate data points, the mistake called pseudo-replication, dressed up an ablation as a finding: the fix rate seemed to jump from 58% to 89% as we stripped the system down, and redone at the right level, the file, the effect vanished. And even a real verifier stayed foolable wherever it ran the detector's own rules, so an audit flagged 17 of the 168 confirmed violations as false positives or pointless fixes. We tell each story with the numbers on both sides and say plainly which parts of the audit have been run. Observability and construct validity are things you design into an agent, not footnotes you add at the end, and none of this is really about accessibility: any agent judged by a signal it can push around is exposed to all three.

Keywords

Agentic Systems, Evaluation, Construct Validity, Pseudo-replication, Verification, Automated Program Repair

1 Introduction

An agent reads a problem, makes a plan, does something, and then gets a grade. For code-repair agents like SWE-agent [1] and OpenHands [2], that grade comes from benchmarks such as SWE-bench [3], which ask one flat question: did it fix the task? The whole field is now lining up behind numbers like that, and the pipeline that produces them has quietly become load-bearing. Nobody stress-tests it, and that is the problem: the success signal is produced by machinery the team built, inside a loop the agent can influence, and aggregated at a grain the team chose. When any of those three is wrong, no amount of statistical care downstream repairs the result, and nothing in the usual reporting format makes the error visible to a reviewer. This paper is what we learned when ours broke three separate times.

None of the three failures we report is new to science, and we say so in Section 2. What is new is the amplification, worth stating up front, because it is why these known hazards deserve attention now. Agents optimize against feedback, so any checker placed inside the loop stops being a mirror and becomes a target: construct validity is no longer a passive property of an instrument but something an optimizer actively erodes. Agents act in coarse chunks but are scored in fine ones, one patch deciding the fate of many scored items, which makes the mismatch between the unit of action and the unit of analysis structural. And agents narrate their own state fluently while the loop already halts on that narration, which leaves reading the self-report as the outcome the path of least resistance.

Our objective, then, is not to evaluate accessibility repair. It is to characterize, with before-and-after numbers from one system, three measurement failures that changed our own conclusions, and to derive from them what an agentic framework should provide by construction. We were not looking for a methods paper. We had built a system and wanted two ordinary answers: which model repairs best, and which parts of our own framework actually help. The answers came back clean, confident, and publishable, and then each one collapsed the moment we looked at how it had been measured. Once, we had let the agent grade its own homework. Once, our statistics were textbook-correct and computed on the wrong rows. Once, a proper verifier was rubber-stamping fixes nobody could use. What follows is those three stories, each with the numbers on both sides of the correction (C1); a crediting rule based on the net per-criterion drop in findings, which a displaced violation cannot game (C2); a construct-validity audit of repair trajectories, its leakage of 17 in 168, and what it does not yet establish (C3); and domain-free practices with the logging schema they need, released with the logs, audit rules and scripts (C4).

2 Related Work

The gap between a plausible patch and a correct one is settled ground in program repair, where passing patches are often mere deletions [5], repairing against the weak signal you judge with produces overfitting [6, 8], and classifiers attack the same gap from the tooling side [9]; a review of 189 LLM-repair studies finds the field still reporting pass-style numbers [7], and conversational repair improves outcomes while pulling harder on the loop's own signal [10]. Pseudo-replication was set down in ecology [11] and is seldom named in agent evaluation, though its cause, one intervention and many scored items, is everywhere in it; construct validity we take from software engineering [12]. The platforms we build on [1–3] supply ability and scoreboards, not run-level observability or outcome auditing, so our contribution is their synthesis for the agentic setting.

3 Background and Setting

Everything below happens inside *a11y-autofix*, a system where local LLM agents repair violations of the Web Content Accessibility Guidelines [4] in React/TypeScript code. The shape is simple. A scanner finds a problem in a component; we wrap it in a structured prompt and hand it to an agent, which writes a patch; and before we

apply anything the patch has to clear four gates: L1 syntax, one well-formed block that parses; L2 interface preservation, props, exports and handlers unchanged; L3 domain, the attribute the fix needs is present; and L4 quality. The models are small and open, so they run on one mid-range GPU: CodeLlama-7B and Qwen2.5-Coder at 3B and 7B, all at four-bit precision, temperature 0.1.

The corpus was built to be re-runnable rather than merely large. Repositories came from the GitHub Search API, stratified over seven application domains, requiring a public React or Next.js codebase with at least 100 stars, a commit within eighteen months and ten or more TSX/JSX files exporting renderable components, and excluding templates, forks, archived repositories, generated UI and rootless monorepos. Each accepted repository was pinned to a commit SHA and snapshotted with per-file SHA-256 digests, so the corpus does not drift if maintainers fix a violation upstream. Thirty-six repositories passed, contributing 1,249 component files, each scanned by Pa11y, Axe-core and Playwright+Axe against the rendered component. Findings were deduplicated by the triple (file, selector, WCAG criterion) and kept at HIGH confidence, two tools agreeing, or MEDIUM, one tool at serious or critical severity; files with no finding stay in as regression guards.

Two cuts of that collection carry the results below, and they are easy to confuse. The model-comparison cut applies a density filter, keeping the highest-violation-count files per domain stratum, and yields 168 confirmed violations across 33 files on three models; the ablation cut keeps every file with at least one violation, and yields 227 violations across 66 files on Qwen2.5-Coder-7B alone. Both draw on the same 36 repositories with the same scanner configuration and the same oracle, so their fix rates are comparable; the two overlap but are not nested. Each experiment ran three times end to end, seed 42 for the model comparison and seeds 42, 137 and 2025 for the ablation. Those repetitions are not additional observations: they resample stochastic behavior on a fixed corpus, so they are averaged within each (condition, file) cell before any inferential statistic, never pooled as independent rows, and the ablation is then tested per file against the full pipeline with a Wilcoxon signed-rank test over 66 pairs, Holm–Bonferroni corrected, with 95% bootstrap intervals.

The measure that carries the paper is the issue fix rate, IFR: the share of confirmed violations that an accepted patch really removes, credited by the rule in Section 4. That word "really" is where the trouble lives, and pinning it down is most of the work. Web accessibility is a good place to study it, oddly, because a tool decides pass or fail without consulting the agent, yet the real target, what a screen-reader user experiences, is soft enough that a passing tool and a helped person are not the same thing. That gap is where all three traps live. Which model repairs best, then? None of them, really: the three landed within four points of one another, at 50–54% IFR, tied after correction, so we started leaning hard on the numbers to find something that did, and that is when the cracks in the numbers themselves started to show.

4 Trap One: Self-Report Is Not an Outcome

A repair agent hands back a patch and, tucked alongside it, a claim of victory: it stopped, it wrote code, it declared the problem handled. Reading that claim as your result is nearly automatic inside a loop that already halts when the agent decides it is done, but it measures the agent's confidence, not the world. Program-repair researchers charted this ground years ago [5, 6]; we knew the literature and fell in regardless, which is the reason to tell it rather than cite it.

In our setup the honest result is not "the agent says it fixed it" but "an independent re-scan no longer reports the violation," and the distance between those sentences turned out to be enormous. An early build had a bug that kept the four gates from ever running as a gate, so patches drew credit from the agent's plausible output alone. That run announced a fix rate near 58%, which we believed and nearly wrote up; rebuilding the pipeline so that credit flowed only from a verified re-scan brought the same models down to about 50%.

Rebuilding it also exposed something worse, and the two findings differ in kind. The drop from 58% to 50% is an *instrument* defect: one corpus, scored by an honest rule instead of a credulous one. The second is a *corpus* defect: 71% of what the old corpus had logged as violations were never real, but leftovers from our test harness, phantom problems that shipped in no application and that the agent had spent its budget cheerfully "repairing." Neither defect caused the other; they arose independently, one in the scoring code and one in the rendering harness. What connects them is detection, because credit never depended on comparing a before-scan with an after-scan, so nothing in the system ever had cause to look at what the findings actually *were*. The missing verifier did not create the phantom corpus, but it is the reason that corpus survived an entire experiment unnoticed, and correcting the instrument is what made the corpus defect visible. An independent outcome check is not only a better score, it is the observation point from which other kinds of invalidity become findable.

The harness bug deserves daylight, because this kind of thing hides well. To let browser scanners inspect a live DOM, our harness renders each component alone inside a tiny preview page, so rules meant for a whole page ran against the harness wrapper, and one page-level ARIA rule tripped by the harness root rather than by any project code produced 240 of the 338 findings in the old corpus, which is where the 71% comes from. Beneath it, a hand-rolled type-stripping step built on regular expressions occasionally wrecked a JavaScript object, so a few components rendered blank and got scanned empty. Excluding those rules and replacing the regex hack with a real parser settled the corpus at 168 violations across 33 files.

Bolting on a verifier fixes the headline problem and introduces a subtler one. A re-scan notices that a *finding* disappeared, but a patch that reshuffles the page can erase a finding while the same underlying violation resurfaces under a new selector, so "the finding is gone" and "the problem is solved" come apart again. We plug that hole by crediting only the net drop per criterion: for criterion c in a file, with n_c^{before} and n_c^{after} findings before and after, the credited fixes are $\max(0,\ n_c^{\text{before}} - n_c^{\text{after}})$, and finding ids never enter the accounting, so three link-name findings that come back as two under new selectors earn one credit, not three. The general point is small and firm: where an agent loop halts and how an experiment scores it are different objects and must not be the same object.

Table 1: Ablation at the right unit, the file. Per-file mean IFR with 95% bootstrap CI; Wilcoxon signed-rank vs. the full pipeline, paired by file (n=66), Holm–Bonferroni corrected. Nothing is significant, where the pseudo-replicated reading of the same data had called these removals a decisive climb from 58% to 89%.

Condition	IFR	95% CI	Δ	p_{corr}
Full pipeline (control)	0.749	[0.64, 0.85]		
− Reflection feedback	0.740	[0.63, 0.84]	−0.9pp	0.539
− Few-shot examples	0.769	[0.66, 0.86]	+2.0pp	0.570
− WCAG rule context	0.766	[0.66, 0.86]	+1.7pp	0.570
− Internal validation	0.808	[0.71, 0.89]	+5.9pp	0.286

5 Trap Two: The Unit of Analysis

This one is dangerous because it looks rigorous. Means, standard deviations, significance tests, all done correctly, all done on the wrong rows. We ran an ablation to answer the question every builder owes their own system: of the machinery we wrap around a bare "here is the code, here is the problem, fix it" prompt, which parts pull their weight? We switched off, in turn, the reflection loop that feeds each rejection reason into the next attempt, the retrieved few-shot examples, the prose explaining the violated rule, and the four gates. Treat every violation across every repetition as its own data point, and the picture is thrilling and wrong: the fix rate appears to climb from 58% for the full system to 89% with internal validation disabled, with p-values small enough to pass for a finding.

Hurlbert wrote this up for ecology forty years ago and called it pseudo-replication: running inference as though observations were independent when they are not [11]. Two dependencies sink the analysis. Violations in one file are repaired by one patch, so they are not separate outcomes but the same intervention counted many times, and a file with 25 violations that a single patch clears buries 25 genuine single-violation files under 25 correlated "successes"; pooling three repetitions then triples the apparent sample while adding no new information. Between them, they mint significance out of correlation. Move the unit to where the patch actually lands, the file, average the repetitions within each cell, compare paired per-file rates with a Wilcoxon signed-rank test and a Holm–Bonferroni correction, and the story falls apart. Table 1 is the sober version, and the raw counts tell it better than the p-values: of 66 paired files roughly sixty are exact ties, so three to five pairs per comparison carry everything. The 58-to-89 climb and the quiet null are one dataset read at two grains.

The file is the right unit *here* for one specific reason: a single patch is written for, and applied to, exactly one file, so the file is what one agent action decides. That reason does not transfer automatically. An agent that emits a multi-file patch has the patch as its unit; one solving a repository-level task has the task instance, whose intermediate actions are not independent replicates of it; while an agent invoked once per violation with no shared state genuinely does have the violation as its unit, and analyzing at the file would throw away information. The transferable rule is not "use files" but *ask what a single agent action decides, and make that the observation.* Repeated runs are a separate axis: they carry real information about stochastic variation and should be reported as such, or modeled as a random effect, but never counted as independent observations at the item level.

6 Trap Three: A Verifier Can Still Measure the Wrong Thing

Escaping the first trap with a verifier opens a third, meaner one. Construct validity is the old question of whether an instrument measures what it claims to [12]. Ours claims to measure accessibility repair; it actually measures scanner silence. How circular the setup is caps what "verified" can mean: the re-scan runs the same engines, and the same rule code, that produced the findings, and since every finding here came from a single tool, detection and verification lean on one engine end to end. A second engine would shrink that circularity without ending it, since accessibility engines share most of the same rules.

Two actual fixes show the failure running both ways. An agent "repaired" a `<video>` with no captions by adding `aria-label="Video content"`: it reported success, the re-scan agreed because the rule that had fired went silent, and a deaf user gained nothing, since captions need a `<track>` element and a label is not one. The reverse happens too, since some components hold their accessible name in a runtime prop and, rendered alone with that prop empty, get flagged, so the agent "repaired" barriers that never existed in a running app. One success rate absorbs both without a mark.

Because the oracle cannot watch itself, we audit the saved trajectories for two leaks. Detection false positives come from a cross-check: line up each finding's rendered context, where the attribute is missing and the scanner therefore fired, against the component source in the agent's prompt log, where that attribute may be tied to a dynamic prop; present in source and empty only when rendered alone marks a harness artifact rather than a barrier. Suspect fixes come from reading the patch for generic names, references to absent ids, wrong roles, captions "fixed" with a label, and edits that change nothing. The unit of that audit is the confirmed violation, keyed by a stable issue id, and its denominator is the whole corrected corpus: all 168 confirmed violations of the model-comparison cut, deduplicated across the three models and three repetitions, each carrying a repair attempt in the trajectory log. It is therefore neither a count over successful repairs nor one over run-level rows. On that base the pass flagged 17 of 168, or 10.1%: five detection false positives and twelve suspect fixes, so the verified rate is a ceiling with a checked band of at most ten points beneath it.

That band is only as good as the flagger, though, and the flagger is triage, not truth: its accuracy has to be earned. Matching a patch to the finding it addressed is imperfect: 57 of the 168 findings were linked through rendered context, the other 111 by file basename alone, which cannot separate two same-named components. More importantly, the human adjudication that would turn the flags into precision and recall **has not been executed**. Its protocol is specified and its instrument built and released: the flagger emits a worksheet in which every flagged case and a matched sample of unflagged ones is labelled independently by two raters under a written rubric grounded in the official W3C techniques, agreement is scored with Cohen's κ, and a third rater settles disputes. The worksheet ships

with the replication package, its rater columns empty, and until they are filled 10.1% is an upper bound on *auto-detectable* leakage and nothing stronger, with the corpus itself scanner-confirmed rather than expert-verified. A verifier that runs the detector's own rules is circular, so report the checked band instead of a lone estimate, and say plainly which parts of the check have actually been run.

7 Practices, and the Position They Support

Across the three stories, one condition made every escape possible: observability. You cannot audit what you did not record, and each recovery here rests on plain engineering, an agent wired so its whole run can be read back later. Concretely, that is one structured JSON line per model call, keyed by model, repetition, file and attempt, holding the prompts, the raw response, token counts, latency, the gate that rejected the patch when one did, and the re-scan verdict. Drop the component source it carries and the harness false positives go invisible; drop the per-gate record and every failure collapses into the single word "failed," where the breakdown is itself a finding: rejections cluster at L1 and L2 while L4 almost never fires. Once that log exists, the three habits the traps argue for cost almost nothing: the logging is a write-through file, the audit is a set of rules, and the right denominator is free. The logs, audit rules, adjudication worksheet and scripts, including the unit-of-analysis redo, live in the replication package.[1]

Under the anecdotes runs the argument we opened with, and the evidence now lets us put it concretely: trap three is Goodhart's law handed a retry budget, trap two is a scoring grain that cannot match the action grain, and trap one is a fluent narrator sitting at the exact point where the loop stops. Stack those together and a field that standardizes on self-reported, pooled, stand-in-measured success will accumulate results that do not replicate. We are not alone in seeing it: Agent-Fence judges deep-research agents with trace-auditable, outcome-based checks instead of trusting a run to explain itself [13], and the systematization of DARPA's AI Cyber Challenge observes that its telemetry was built for live scoring, not for reconstructing after the fact why a system behaved as it did [14]. The position is thus a step past hygiene: observability and construct validity are requirements for agentic systems, not reporting niceties, and a framework should ship the means to reconstruct why it acted as it did and to confirm a credited success is real, the way a serious distributed system ships tracing.

8 Threats to Validity and Conclusion

Our own second trap applies to this paper. The evidence is one domain, one corpus, small samples, and these are surely not the only three traps; the strongest way to reinforce the case is to reproduce one elsewhere, in security linting or test-based repair. Our verifier shares rule code with the detector, and the human adjudication is packaged but not executed, so 10.1% is an automated upper bound. The ablation null could hide a tiny real effect, so read it as absence of evidence rather than proof of no effect, and a mixed-effects model would handle the nesting of repetitions within files more fully than the paired test we chose for transparency.

Three measurement traps each flipped a headline result while we built a code-repair agent: crediting self-report over a verified outcome, scoring co-located outcomes as independent, and trusting a verifier that measures its own stand-in. None is new to science, all are easy to fall into, and all are cheap to escape once the agent is observable. A result a corrected measurement can flip was never a result, so the cheapest useful thing many of us can do is make sure that when an agent says it succeeded, we have our own reason to believe it.

[1] https://github.com/jpwieland/a11y_experiment

Acknowledgments

We thank the SE4AS 2026 reviewers, whose comments on the unit of analysis and the audit denominator shaped this version. During the preparation of this work, the authors used Generativa AI to revise grammar, clarity, and phrasing of the text, and to help draft specific excerpts. All content generated or revised was reviewed and edited by the authors, who take full responsibility for the content of this publication.

Artifact Availability

The logs, audit rules, adjudication worksheet, and scripts behind every number reported in this paper, including the file-level ablation, are publicly available in the replication package at https://github.com/jpwieland/a11y_experiment.

References

[1] J. Yang, C. E. Jimenez, A. Wettig, et al. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *NeurIPS*, 2024.
[2] X. Wang, B. Li, Y. Song, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *ICLR*, 2025.
[3] C. E. Jimenez, J. Yang, A. Wettig, et al. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR*, 2024.
[4] W3C. Web Content Accessibility Guidelines (WCAG) 2.1, 2018.
[5] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. *ISSTA*, 24–36, 2015.
[6] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. *ESEC/FSE*, 532–543, 2015.
[7] Q. Zhang, C. Fang, Y. Xie, et al. A Systematic Literature Review on Large Language Models for Automated Program Repair. *arXiv:2405.01466*, 2024.
[8] J. Petke, M. Martinez, M. Kechagia, A. Aleti, and F. Sarro. The Patch Overfitting Problem in Automated Program Repair. *Companion Proc. FSE (IVR)*, 2024.
[9] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying Patch Correctness in Test-Based Program Repair. *ICSE*, 789–799, 2018.
[10] C. S. Xia and L. Zhang. Automated Program Repair via Conversation. *ISSTA*, 2024.
[11] S. H. Hurlbert. Pseudoreplication and the Design of Ecological Field Experiments. *Ecological Monographs*, 54(2):187–211, 1984.
[12] P. Ralph and E. Tempero. Construct Validity in Software Engineering Research and Software Metrics. *EASE*, 13–23, 2018.
[13] S. Puppala, I. Hossain, M. J. Alam, et al. Agent-Fence: Mapping Security Vulnerabilities Across Deep Research Agents. *AAAI Symp.*, arXiv:2602.07652, 2026.
[14] C. Zhang, Y. Park, F. Fleischer, et al. SoK: DARPA's AI Cyber Challenge (AIxCC). *arXiv:2602.07666*, 2026.