

Rebuild Dossier: Mechanically-Enforced Specs for Agentic App Rebuilds, and What Model-Tier Failures Reveal

Parker Fawcett
Independent Researcher
Parkerscottfawcett@gmail.com
ORCID: 0009-0003-9699-7422

Abstract

An AI agent’s rebuild is only as good as the process that produced it. Prior work found that once a model is strong enough, a multi-agent rebuild pipeline loses to the simplest approach: giving the model the original code and one instruction (AgentModernize). We present rebuild-dossier, an open-source tool that locks an application’s real interface — its exact inputs and outputs — before any code is written, then enforces one-test-at-a-time building through automated checks, not written instructions alone.

Three results shape this evaluation, with differing amounts of evidence. First, in a small comparison, the compliant agent failed a held-back test while the rule-breaking agent passed everything — proof that a passing suite doesn’t certify correctness when tests can be gamed. Second, we tested whether this beats simply giving the weaker model the source and one instruction: tied on a small app, but lost outright on a larger one where the automated check wasn’t even running — pointing to the check mechanism, not interface-locking, which held up separately. Third, every claim here is checked at three levels — the agent’s own report, an automated log, and the actual files produced — catching real errors, including a bug in our own logging code, that a single level would have missed.

These risks reproduce on a different model and toolchain: a stronger model followed our process three times running, something the weaker model never managed. The tool is public, MIT licensed, and reproduces end to end against our own applications.

Keywords: agentic software engineering; LLM-based code generation; specification-driven rebuild; interface-contract verification; test suite validity; model-tier evaluation

1. Introduction

Agentic “rebuild this app” pipelines promise a clean reconstruction of an existing codebase from a specification an LLM extracts. The most rigorous prior study, AgentModernize, evaluates a four-agent modernization pipeline under a protocol

designed to be fair to any self-consistent API shape a method chooses. Even so, no single method dominates: the full pipeline beats single-prompt and chain-of-thought baselines with a weaker backbone, but a single prompt outperforms it outright with stronger ones, and its feedback loop is not uniformly beneficial, regressing some scenarios even as it improves others. Its diagnosed bottleneck, once the naming-shape confound is removed, is code generation, not extraction (Section 2 gives the full figures and mechanism).

We take both findings as open problems to build against, not as solved prerequisites. The feedback loop’s inconsistency motivates regression-aware patching that reverts to the best iteration rather than always applying the latest one. The generation, not extraction, bottleneck motivates locking interface contracts before generation — targeting a problem distinct from AgentModernize’s own fairness fix, since a rebuild’s target shape is externally fixed by real existing callers rather than freely chosen the way a modernized API’s shape is (Section 2 develops this distinction fully).

We take those findings as the design constraint, not the ceiling. Four contributions, each scoped to what our evaluation shows:

1. Mechanically-enforced contract locking closes a specific, named class of structural mismatch: a rebuild with correct behavior but wrong shape fails fast and legibly rather than opaquely. Tested directly, not just asserted, against a fresh blind rebuild’s actual output (Section 4.6) — a targeted test built specifically because Section 4.5’s earlier attempt to isolate this contribution via ablation did not cleanly resolve the question; the evaluation section is ordered by when each result was obtained, not by this list’s numbering, and Section 4.6 comes after 4.5 for that reason. This direct test is currently one app, one tier, one run (Section 4.6’s own scope note) — evaluated, not yet replicated.
2. Prose rules are not enforcement — demonstrated, not asserted. With an identical spec, a weaker model complied with hook-enforced rules and silently violated a prose-only one; we convert it to a hook and verify against the exact failing case.
3. A test, or the metric read from it, is not automatically a correct specification — demonstrated, not asserted. Verification signals can encode a measurement artifact rather than a real requirement, and a compliant agent will build exactly that artifact if nothing tells it otherwise. Sharpest in an ablation (Section 4.5) that surfaced, unplanned, a pass-rate metric directly rewarding rail violation over rail compliance — the full paired sequence is reported there, not repeated here. Kept as its own N=1 case, distinct from a mechanistically different, prior pair of instances — black-box tests, observed twice in two different apps and model tiers, that encode how they were measured rather than what should be true, with a compliant agent implementing the artifact. Both are distinct from AgentModernize’s two categories; the test-artifact pair is a small case series (N=2), not yet a rate. Relatedly, we identify motion (animation) as a total, model-independent

verification blind spot: DOM-text assertions and single-frame reference screenshots have no representation for it at all, regardless of model tier (Appendix C.5) — a representation gap, not a classifier-accuracy problem.

4. A model-tier boundary between compliance and diagnosis. Rails hold for build-order compliance across tiers; they do not grant diagnostic capacity to a weaker model facing an unfamiliar failure. Replicated across three independent trials against the identical fixture (Section 4.4) — a consistent finding on a small sample, not yet a measured rate.

Legacy-app rebuilding is our evaluation harness, not our subject: it is a domain where “the agent behaved correctly” has an objective, mechanically checkable definition, which is what makes these agent-reliability findings measurable.

Section 4 develops this throughline in the order results were obtained (Sections 4.1–4.12). A security evaluation of the tool’s optional network transport and further test-integrity, visual-fidelity, and motion findings are real but orthogonal to it, and are collected in Appendix C rather than interrupting the main argument.

2. Related Work

AgentModernize (Ahmed and Galib, 2026) is the closest prior art and primary comparison — cited here specifically as v2 (arXiv:2605.17535v2), whose numbers we use throughout. v1 (17 May 2026) reports the opposite finding on the exact crossover this paper leans on: its own abstract states full AgentModernize-with-feedback was “the only configuration with non-zero mean BER under every backbone,” with both baselines at 0.0% everywhere — no crossover, the pipeline dominates unconditionally on every model tier tested. Every citation to AgentModernize in this paper is to v2’s revised numbers (23.0% vs. 12.4%/4.5% with a weaker backbone, but losing outright to a single prompt with GPT-4o and GPT-5.3-codex) — the crossover motivating this paper’s Introduction and the argument below. A reader who checks v1 instead would see the opposite finding and should read that as a real revision between versions, not a discrepancy in this paper’s own reporting. Their four-agent pipeline, evaluated under a protocol where each method’s tests adapt to its own API surface (removing the naming/shape penalty that could otherwise confound a “structural mismatch” diagnosis), finds no single method dominates: the full pipeline beats single-prompt and chain-of-thought baselines with a weaker backbone (23.0% vs. 12.4% / 4.5% mean behavioral equivalence, GPT-4o-mini) but loses to a single prompt outright with stronger backbones (GPT-4o, GPT-5.3-codex). Their feedback loop is not uniformly beneficial: it raises equivalence on scenarios with localized, correctable errors and regresses it on others (one scenario falls from 75.8% to 12.1% after “correction”). Their diagnosed bottleneck, with the shape-naming confound explicitly controlled for, is still code generation rather than extraction (their Behavioral Specification Graph captures over 90% of gold-

standard rules) — one scenario fails to wire multi-step business logic together at 0% across every method, regardless of endpoint shape. They explicitly motivate “regression-aware feedback that retains the best iteration” from this same feedback-inconsistency finding — a mechanism we build. Our mechanically-enforced contract-locking targets a related but distinct problem their fairness fix deliberately sets aside: their protocol correctly does not penalize a method for choosing any self-consistent shape, appropriate when any coherent modernized API is a valid outcome. A rebuild has no such freedom — real existing callers depend on the original app’s actual contract, so the correct shape is externally fixed, not self-determined. We do not re-run their benchmark; we cite their figures as context, not a matched baseline (Section 5.3), and their evaluation domain (COBOL/PL-SQL telecom and banking) differs from ours (modern web applications), so the comparison is motivational, not head-to-head.

One further contrast is worth stating explicitly rather than leaving implicit in the paragraph above: our enforcement mechanism and their Equivalence Validator differ in kind, not just degree, in a way that bears directly on the regression they report. Their Validator is itself an LLM-driven correction loop — it rewrites code in response to a detected failure, which is the same mechanism behind the regression they name (one scenario falling from 75.8% to 12.1% after “correction”): a generative step can always introduce a new error while fixing an old one. Our `PreToolUse/PostToolUse` hooks never rewrite anything; they only block or permit a tool call, or run an existing test and report pass/fail. There is no generative step inside the hook for a regression to come from. This is not a claim that our rails are more capable — they enforce build order and test coverage, nothing about correctness of the code an agent writes within those bounds — only that they are structurally immune to the specific failure mode AgentModernize’s own feedback loop exhibits, because blocking and permitting cannot themselves introduce the error a rewrite can.

The Coming Legacy Cliff (Ahmed, 2026, Zenodo, DOI: 10.5281/zenodo.20279139) is a position paper proposing a six-part research agenda for behavior-preserving modernization: behavioral specification as a first-class artifact (D1), shared benchmarks (D2), institutional-knowledge recovery (D3), practical equivalence verification (D4), human-AI trust (D5), and workforce augmentation (D6). Two of its open questions are directly, if narrowly, answered by our evaluation rather than merely motivated by it. D1 asks what representation balances automated extractability with human inspectability; our locked `spec/` and generated `CLAUDE.md` are one concrete instance, in a different domain (modern web apps rather than COBOL/PL-I/Assembly). D5 asks, as what the paper calls its single most important open question, whether inspectable intermediate artifacts measurably improve an engineer’s ability to detect modernization errors compared to reviewing generated code directly — a question AgentModernize’s own conclusion independently poses as well. Our human-in-the-loop case-file queue (Section 3.2), surfaced via MCP elicitation rather than a bespoke review UI, is exactly this mechanism, though we do not run the trust study either paper calls for; we build the artifact the study would

need, not the study itself.

Characterization / golden-master testing (Feathers, 2004): the decades-old foundation for capturing existing behavior before altering it; our mutation-checked generation is a mechanized descendant. Lineage, not novelty.

SWE-bench+ and UTBoost are the closest analogues for our test-integrity finding: both quantify that a large fraction of coding-agent benchmark “passes” reflect weak verification rather than correct behavior. SWE-bench+ (Aleithan et al., 2024) manually screened successful SWE-Agent+GPT-4 patches on SWE-bench and found 32.7% involved solution leakage (the fix was present in the issue text or comments) and 31.1% were suspicious passes due to inadequate tests; filtering both dropped the measured resolution rate from 12.5% to 4.0%. UTBoost (Yu et al., 2025) built an LLM-driven test generator specifically because SWE-bench’s manually written tests were often insufficient to catch an incorrect patch, and found 345 patches mislabeled as passing across SWE-bench Lite and Verified, changing 18 and 11 leaderboard rankings respectively. Our mutation check is an independently-arrived-at instance of the same insight in a different domain: agentic-coding “tests pass” claims are not self-certifying and need their own verification layer, whether the benchmark is a fixed issue-patch corpus or a spec this tool generated itself.

Spec-driven development (Spec Kit, BMAD, practitioner variants) independently validates ambiguity-as-first-class-citizen and documents that prose config is advisory rather than enforced at scale — the exact gap our hooks target. That ecosystem targets greenfield work; none reverse-engineers a spec from an existing, possibly-broken app across disagreeing evidence sources.

Consumer-driven contract testing and schema-first API validation (Pact, n.d.; OpenAPI-based tools such as Schemathesis) are the closest non-LLM precedent for our own load-bearing premise — lock an interface, then test behavior against it — and their absence here was a fair gap to name. Pact’s contract is authored by the consumer: a unit test run against a Pact mock server records each request/expected-response pair into a pact file, which the provider then verifies its real implementation against — coordinating two independently developed, already-existing (or concurrently developed) services, not reverse-engineering either one’s own undocumented shape (Pactflow, n.d.; Pact, n.d.). OpenAPI’s contract is typically authored design-first — the specification is written before implementation, and tools like Schemathesis then generate requests across every endpoint and parameter combination the spec declares and validate that a live implementation’s status codes, response shapes, and headers match it (Speakeasy, n.d.). Both share our premise; neither addresses our actual problem. Both start from a contract a human already wrote down — for Pact, a consumer’s stated expectation of a service; for OpenAPI, a designer’s stated intent for one not yet built. Ours starts from neither: `ingest_repo`’s static extraction produces the analogue of a schema nobody wrote down in the first place, read verbatim out of a real, undocumented, possibly buggy implementation — the artifact these tools assume already exists is exactly what our pipeline exists

to reconstruct.

Record-and-replay contract capture (e.g., VCR, WireMock, and similar test-double recorders) is a closer non-LLM analog than either Pact or OpenAPI: these tools also capture real request/response pairs from a running system to build a contract after the fact, the same undocumented-schema starting point `ingest_repo` targets. What distinguishes ours: a recorder’s contract is consumed by the same application’s own regression suite, replayed against the system it was captured from — not handed to a fresh agent tasked with reconstructing the application from scratch in a different codebase. Ours is the latter case, which is why contract fidelity alone (Section 4.6) is necessary but not sufficient; a correct contract still has to survive being interpreted by a rebuild agent that never saw the original system.

MCP and elicitation provide the interactive human-in-the-loop mechanism without a bespoke UI.

Specification gaming. Our test-harness-artifact finding (Section 5.1) is a specific instance of a broader, named phenomenon: an agent satisfying the literal specification of an objective without achieving the intended outcome (Krakovna et al., 2020; Amodei et al., 2016). We did not initially frame it this way, and connecting it explicitly matters: the same literature has already documented that a compliant optimizer reliably finds and exploits gaps between a stated objective and an intended one, across settings far removed from software rebuilds. Our contribution to that literature is narrow but concrete: a black-box software test can itself be the mis-specified objective, not just the reward function or fitness function these examples usually concern, and a coding agent following instructions correctly will build the artifact rather than the intent if nothing distinguishes the two.

Reporting guidelines for LLM-involving empirical studies (Baltes et al., 2026) directly informs this paper’s own disclosure practice: its first guideline — declare LLM usage and role, separately for each distinct role a study assigns — is exactly what Section 8’s “Use of AI” statement follows, given this paper assigns LLMs three distinct roles rather than one.

3. System Design

3.1 Architecture

rebuild-dossier is an MCP server exposing six tools, run from inside a normal Claude Code (or any MCP-compatible) session. The suite stands at 512 unit tests across 83 files at the pinned commit (Section 7), confirmed by a direct run rather than carried over from the findings doc’s own status line, which the doc itself flags as likely stale by the time this is read. It builds no orchestration engine, dashboard, or background-task machinery of its own, relying on the host client’s native primitives. Its sole output is a self-contained rebuild package

a separate coding-agent session consumes; the tool does not rebuild the app (Section 5.2).

The six tools: **ingest_repo** (static analysis only); **crawl_site** (headless Playwright, progress notifications); **flag_known_bug** (verbatim, highest-priority signal); **get_case_queue** / **resolve_case** (ambiguity queue via MCP elicitation with scripted fallback); **generate_spec** (writes locked spec, config, and mutation-checked tests to a clean sibling directory).

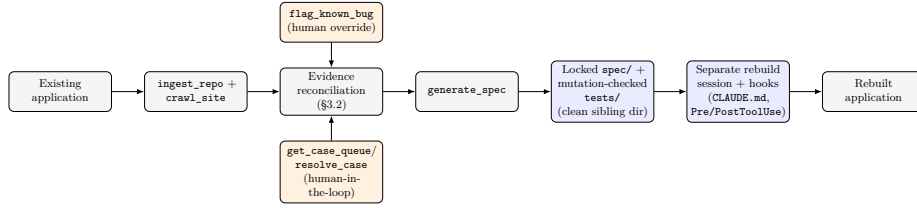


Figure 1: rebuild-dossier’s pipeline: static and dynamic extraction feed a non-negotiable evidence-reconciliation rule (Section 3.2), which locks a spec and mutation-checked tests (Sections 3.3, 3.5) consumed by a separate, hook-governed rebuild session (Section 3.4).

3.2 Evidence reconciliation — the non-negotiable rule

The riskiest failure is silently validating a bug as intentional: multiple sources can quietly agree on the same mistake with nobody having stated why. The core property: auto-resolving an ambiguity requires both signal agreement and an affirmative signal that a decision was made (a stated comment, a TODO admitting a bug, or a human answer). Silent agreement alone always becomes a question. A human-flagged known bug overrides all inference. This rule is deterministic code, unit-tested against a hand-authored matrix before wiring to real extraction — its own correctness is the thing most under test.

Honest scope limit: on both evaluated apps, no comment/TODO signals existed (confirmed by grep). Reconciliation on genuinely conflicting real evidence is therefore mechanism-verified (via a constructed synthetic conflict) but not field-tested (Section 5.4).

3.3 Interface-contract locking

Contracts are extracted verbatim, never paraphrased, and checked before behavioral tests run — directly targeting AgentModernize’s named dominant failure.

3.4 Mechanically-enforced discipline

The two-tier comparison (Section 4.2) found a weaker model would read a natural-language build rule, demonstrate understanding, and violate it anyway, because nothing checked it. Two rules are enforced by runtime hooks:

a `PreToolUse` hook blocks edits under `spec/`; a second blocks writes to any file in `spec/untested-contracts.json`. A `PostToolUse` hook runs the visible suite after each edit. This converts “only build what is failing” from ignorable instruction to unbyassable constraint.

Building this fix caught a bug of the same shape one layer up: the natural “is-covered” signal — a test’s `sourceFile` — is, for a gate test, the original app’s guard file, not the route it covers; using it naively would have blocked the very files the tests require. Fixed with a separate `coveredRouteFiles` field and a regression test, verified against the real case.

A related gap surfaced by the ablation study (Section 4.5) and the resulting fork left open in Section 4.3’s `untested-contracts` claim: when a hook is scoped to a subagent’s target directory and never consulted, nothing distinguishes that silence from “the hook fired and found no violation” — both produce a clean `settings.json` and no error either way. The `PostToolUse` hook now writes a small heartbeat file (`.claude/.hook-heartbeat.json`: a timestamp and a firing count) ahead of running the real test command, so whether a session’s hooks were actually consulted becomes checkable from the filesystem afterward instead of inferred from the absence of a violation. This is observability, not enforcement — it cannot block or correct a missed hook the way `SubagentStop` (Section 4.5) could, and it does not retroactively resolve the open fork in Section 4.3 or the ablation trials in Section 4.5, both of which ran before this mechanism existed. It closes the gap going forward only: any trial run after this fix leaves a mechanical, checkable trace of whether its hooks were actually live for that run — consistent with this paper’s own insistence elsewhere on verifying against the filesystem rather than a report.

3.5 Mutation-tested test generation

Before trusting any test, the pipeline mutates the original source in a scratch copy and confirms each test catches it. Tests catching nothing are downgraded to `weak`; tests failing even against unmutated source are marked `unrunnable`. The tool applies its own skepticism to its own output.

3.6 Output isolation

Output goes to a clean sibling `<repo>-rebuild/` directory containing nothing else — so a fresh session has nothing else in scope to read, drift toward, or edit in place, which is what makes the generated `CLAUDE.md` binding.

3.7 Frontend page tests and an optional non-deterministic classifier

Next.js page routes receive Playwright-captured tests: DOM-text assertions (the mutation-tested gate) plus a reference screenshot (documentation, not asserted pixel-by-pixel). Static vs. dynamic classification is a deterministic regex by default. An optional, opt-in vision classifier (gated on two env vars together) can replace it, sending a screenshot and secret-redacted source to a third-party

model — the pipeline’s only non-deterministic step and only outbound egress, both documented, with the deterministic default byte-for-byte unchanged when off. Section 5 flags that enabling it makes reproducibility conditional.

4. Evaluation

Methodology, up front: not a large-N benchmark. Two real, messy, author-selected applications with pass/fail criteria declared before each run, and independent verification of every self-reported result — direct test re-runs and file inspection — before acceptance. Depth and verification honesty over breadth. Selection was convenience sampling from the author’s own repositories (Section 5.3); reproducible does not mean representative. LLM agents are used in two distinct capacities throughout this section — as the subjects under test, and as a tool assisting the author in running and verifying trials — each declared separately in Section 8’s “Use of AI” statement.

4.1 Pre-registered outcome taxonomy

Each handoff is classified, against criteria fixed before the run, into one of four outcomes, not three — stated here in full, verbatim from `docs/v0-findings.md`’s own run-criteria source, since every result below depends on knowing exactly what these terms mean before encountering them:

Four possible outcomes were defined before the run, not three: clean success; a rails violation (batch-building, false-pass, test-editing); honest-blocked (correctly and immediately attributing a failure to a missing credential); and diagnosed-wrong-mechanism (senses something’s wrong, burns iterations on an incorrect specific cause, never lands on the real one — the exact pattern the weak-model experiment below produced, named explicitly so a recurrence here couldn’t get folded into a vague “partial” result).

4.2 Application A: Madeline (small Next.js, client-side gate pattern)

A small real personal site with genuinely non-obvious logic, zero existing tests, zero TODO/FIXME comments — messy in the way the brief required.

- **Headline:** a fresh agent, given only `CLAUDE.md`, `.claude/`, `spec/`, and two locked test files, built a working Next.js app from scratch and converged to 3/3 tests passing, ~12–16s per run, across both model tiers, isolated single-variable — same spec, tests, hooks, directory; only the model changed. Confirmed by the author re-running the suite and reading the generated code directly (Section 5.3 for what “verified” means throughout this paper). The exact per-tier trial count is not cleanly resolvable from available records — a limitation, not a claimed number.

- Rails held identically: reading discipline and convergence (~1 iteration each) — direct evidence for the thesis that strict rails let a weaker model succeed at what the rails check, independent of reasoning strength.
- The precise gap: the weaker model built placeholders for all 8 locked contract pages — including the 6 with zero test coverage — contradicting the prose “pick ONE currently-failing test” instruction, then self-reported “no ambiguities” (untrue, caught by inspecting files, not its report). By the taxonomy in Section 4.1, this is a rails violation: batch-building ahead of tests. The strong model built only the 2 tested pages and flagged the other 6 (clean success). The mechanism: the `PostToolUse` hook is real enforcement, so both tiers nailed the tested behavior; “only build what’s failing” was prose only. Hooks cannot catch a violation of a rule they were never written to check — motivating Section 3.4’s second hook, verified by simulating it against the exact files the weak model wrote (blocks all 6 batch-built, leaves the 2 tested editable).
- A later, larger measurement of the same app exists and should be read as an additional data point, not a correction. Re-running `generate_spec` against Madeline today, after fixing a mutation-check bug found during the ablation study (Section 4.5), produces a materially larger result: 8 page-level tests plus the original 2 gate tests, 6 mutation-verified (`tests/visible/`), 1 held-out, 3 **weak**, zero untested contracts remaining. This does not reproduce “3/3 tests passing” — confirmed by commit history, 3/3 refers to the 2 original gate-test files, written before page-level test generation existed — the two numbers measure different things because the tool measured less then than it does now. The original, narrower claim has since been reconfirmed a second way: a fresh sub-agent given only the regenerated spec, with no access to the original source, independently reproduces 3/3 on the two gate-test files and 7/7 visible / 1/1 held-out on the full current spec — a figure three further, explicitly-Haiku trials in Section 4.8 independently reproduce again.

Run	Scope	Visible	Held-out	Weak	Unrun.	Untested left	Outcome
Sonnet, origi- nal	2 gate- test files (2026- 07-21 spec)	3/3	—	—	—	0 of 2	Clean — built only the 2 tested pages

Run	Scope	Visible	Held-out	Weak	Unrun.	Untested left	Outcome
Haiku, original	Same spec	3/3 (tested)	—	—	—	6 of 8	Violation — batch-built all 8, incl. 6 untested
Re-run, current spec	8 page + 2 gate tests, post Turbopack-symlink fix	6/6	1/1	3	0	0	—
Sub-agent, gate-tests only	2 gate-test files only	3/3	—	—	—	—	—
Sub-agent, full current spec	6 visible + held-out	7/7	1/1	—	—	0	Clean success

4.3 Application B: catchandtrade (an 83-route e-commerce application)

Structurally different, larger, integration-heavy (83 routes: 64 API + 19 page). The application integrates with Postgres, Stripe, and eBay, but we did not have credentials for the latter two; those paths were exercised only to the extent the application’s own code handles their absence, not tested end to end. Worth stating plainly before the result below, since “Prisma, Postgres, Stripe, eBay” as a description could otherwise imply a broader tested surface than what was actually reachable.

Generator gap found and fixed first: the only API-contract generator was hard-gated on **express**, which this app lacks, so it silently produced zero tests for all 64 API routes while writing correct contracts — which would have listed 55 files as untested and let the hook block the entire app. A new generator calls each handler directly with a constructed **NextRequest**. Running it against the real app surfaced two further mutation-harness bugs, both fixed.

Generator result: **mutationsChecked** went from 0 to 353 across 64 API routes.

32 mutation-verified (20 visible, 12 held-out); the other 32 honestly downgraded (14 weak, 18 unrunnable — lacking live Postgres/Stripe/eBay/JWT). `untested-contracts.json` dropped from 55 files to 19 (all page routes).

Metric (64 API routes)	Before generator fix	After generator fix
<code>mutationsChecked</code>	0	353
Mutation-verified, visible	0	20
Mutation-verified, held-out	0	12
Weak	0	14
Unrunnable (no live Postgres/Stripe/eBay/JWT)	0	18
Files listed in <code>untested-contracts.json</code>	55	19 (all page routes)

The handoff: a fresh strong-tier session got `apps/web-rebuild` under App A’s conditions, with real infrastructure provided as a given where the tool’s spec should capture it but doesn’t yet (a Postgres test DB from the real `schema.prisma`, placeholder JWT secrets) and genuinely absent where the tool could never capture it (Stripe/eBay/TCG credentials). Result, independently verified by re-running both suites and reading actual error output: tests/visible 20/20 passing; tests/held-out 0/12, every held-out failure a pure “never built” scope gap (6 `Cannot find module`, 6 missing sibling HTTP method), zero logic bugs, zero credential-blocked failures, zero fabricated passes. Classified clean success.

Suite	Passing	Failing	Failure cause
<code>tests/visible</code>	20/20	0/20	—
<code>tests/held-out</code>	0/12	12/12	6 × <code>Cannot find module</code> (route never built); 6 × sibling HTTP method missing on a file that was built

0/12 held-out is correct, not concerning — the “contracts without tests don’t get built” property (Section 5.2) at ~10x the route count. A fresh session sat in front of 19 untested page contracts and never touched them (zero `page.tsx` files in the output). Why is a genuinely open fork: whether this handoff ran as a genuinely separate top-level session (hook live and enforcing) or via

the Agent-tool subagent path Section 4.5 later shows bypasses target-directory hooks entirely (Sonnet’s own judgment doing the real work, no mechanical enforcement behind it) cannot be confirmed from any surviving session transcript. Either way, Sonnet’s behavior itself was correct — the fork is about why, not whether. Credential-blocked routes were engineered around, not faked (spot-checked: `POST /api/orders` records a `PENDING` order with a correctly-reasoned comment about the absent `STRIPE_SECRET_KEY`). A real secondary bug (the agent’s `package.json` test-script change) was verified as a legitimate necessary fix and fixed at the source.

4.4 Weak-model diagnostic boundary

This experiment used its own pass/fail/partial criteria, declared before running: success means correctly attributing the failure to its real mechanism without editing application logic; failure means editing real logic or getting stuck; partial means working around the symptom without understanding it — distinct from the four-outcome taxonomy in Section 4.1, which governs full rebuild handoffs, not this narrower diagnostic probe.

We reintroduced a previously-fixed environment bug (a dev-server origin-trust mismatch) into a fresh Madeline copy and ran the weak tier against it three independent times, each a fresh copy and fresh session, specifically to push this claim from a single anecdote toward something that could replicate or not.

Trial 1’s first attempt was confounded, not a result — it reported 3/3 passing, but the weak-tier rebuild’s own `package.json` had silently resolved an unpinned dependency to a version that simply lacks the bug being tested for, so there was nothing to diagnose. This was itself a real, generalizable finding: nothing in the generated contracts pinned an exact dependency version, so a rebuild agent could silently drift past the exact defect an experiment (or a real rebuild) depends on reproducing — fixed by locking the original app’s exact installed versions into the generated `package.json`. Trial 1’s controlled run, with the bug genuinely present, hit it directly: 1/3 visible passing, confirmed directly. Reading the actual code, not the self-report: the model edited real navigation logic three separate times chasing an incorrect theory (three different redirect mechanisms, then an unrelated `setTimeout` “fix”), none of which could have worked since the real cause is a total hydration failure, not a race — it ultimately stopped and reported the uncertainty honestly rather than fabricating success.

Trial 2 hit an unrelated infrastructure snag worth naming rather than hiding — a mid-response API cutoff, resumed from its own transcript and independently re-run twice against its unchanged final code — which surfaced its own real finding: the two re-runs returned different results on identical code (5 failed/2 skipped vs. 6 failed/1 passed), meaning the reintroduced bug produces non-deterministic dev-server/test behavior, not a stable failure signature. Worth distinguishing from Appendix C.3’s own flakiness finding, which traces to a genuinely time-varying value a static classifier misreads as static; here the

source, tests, and bug are byte-for-byte identical between re-runs, and the non-determinism lives in the runtime negotiation itself, not in any value it reads. Stable across both re-runs, confirmed by reading the actual files: `next.config.ts` now sets `reactStrictMode: false`, a real configuration change made chasing the console-error symptom, alongside several input-handling rewrites. The agent’s own words, unprompted: these errors “originate from Next.js dev server infrastructure, not application code” — sensing the category correctly, same as Trial 1, without isolating the mechanism.

Trial 3 ran cleanly end to end: 6/7 visible failing, 1/7 passing, held-out 0/1, confirmed by an independent re-run returning the identical split. Reading the actual code: `src/app/home/page.tsx` wraps its redirect logic in a `typeof window !== 'undefined'` guard added “to prevent hydration errors,” trying both `router.push()` and `window.location.href` before settling on the former — the same shape as Trial 1’s redirect churn, with two attempts instead of three. Its own diagnosis named a third distinct candidate cause (“Next.js 16.2.10, React 19.2.7, and the test harness” compatibility) — again correctly suspecting an environmental cause without isolating which one.

Trial	Visible result	Real code/config edited chasing the theory	Environmental cause explicitly suspected?
1	1/3 passing	Three redirect mechanisms, then an unrelated <code>setTimeout</code>	Yes — stopped and said so
2	6/7 or 5/7 failing (non-deterministic across identical re-runs)	<code>reactStrictMode: false</code> , repeated input-handling rewrites	Yes — “Next.js dev server infrastructure, not application code”
3	6/7 failing (stable across re-runs)	<code>typeof window</code> guard, two navigation mechanisms tried	Yes — named Next.js/React version compatibility as a candidate

Trial 1’s 1/3 and Trials 2–3’s X/7 are not directly comparable ratios, and the table should not be read as a trend across them: Trial 1 predates page-level test generation (Section 4.2’s own note on the same tool-version gap), so its spec had only the 2 gate-test files’ 3 cases, while Trials 2 and 3 ran against the current generator’s full 8-page spec. What is comparable across all three, and is the entire point of running them, is the qualitative classification — not the raw pass count.

Classified, by the letter of the pre-declared criteria, as failure in all three trials — real application code or configuration was edited, more than once, in every single trial, attempting to fix what is actually a harness-level defect. But all three are the same materially different failure from simply getting stuck: every trial correctly identified the failure category (environmental, not application logic) — stated in its own words, not extracted from silence — without ever isolating the specific mechanism, and every trial reported its actual counts rather than fabricating a passing result. Contrasted with an earlier, unrelated harness bug that the strong tier diagnosed and fixed with a working (if non-idiomatic) workaround: the weaker model, in three separate, independent sessions, senses “this isn’t my code’s fault” without being able to act on that insight — a third, distinct outcome from either cleanly diagnosing or working around a failure, now observed three times, not once. The pattern, stated precisely as observed so far: rails compensate for a weaker model’s judgment about what to build, not its capacity to diagnose an unfamiliar failure it has not seen before. Three independent trials, three consistent outcomes — no longer a single anecdote, though still a small, single-app, single-fixture sample, not a rigorously measured rate or an established boundary (Section 5.3).

Appendix C (Sections C.1–C.5) reports additional findings not central to this paper’s main throughline: a security evaluation of the optional HTTP transport, test-integrity/frontend-page-test findings, reference-screenshot visual fidelity, and motion (animation) as a verification blind spot. They are real, load-bearing results for the tool as a whole, but orthogonal to the contract-locking / discipline / single-prompt comparison this section otherwise develops, so they are relocated there rather than interrupting it.

4.5 Ablation: isolating contract-locking’s contribution

Headline result. Re-verifying two weak-tier reps against the real filesystem rather than either side’s own report: the rep that correctly followed rail discipline — declining to build a page with no failing visible test to justify it — finished with a failing held-out test, while the rep that violated batch-build discipline finished fully green. Rail compliance produced a worse naive pass rate than rail violation. A pass-rate metric is not self-certifying, and here it rewards exactly the behavior the rails exist to prevent (Section 5.1) — a sharper, more citable result than the clean with/without contrast this ablation was built to produce, even though it isn’t the result the design targeted. The subsections below report that result, its real evidentiary limits, and three further replications at increasing, unmanufactured scale (6, then 19, forbidden pages), plus a directed test of whether the mechanism has ever actually blocked a real attempt.

4.5.1 Setup Claude Code’s hooks (Section 3.4) aren’t natively honored by OpenCode, so a native OpenCode plugin reimplements the identical two rails — live-verified against a real session before use — and gates enforcement on a sibling `enforce` marker; `with/without` rep directories are byte-identical except

for its presence, diffed before every trial. The fixture used for every result below except the first four reps is a page with a believable cause for missing coverage (a slow query finishing past the capture window) alongside enough ordinary, capturable pages for a genuine held-out split. An earlier, pathological version (a page engineered to never render at all) was replaced after those first four reps produced a confounded “no violation” signal — a model declining to build a page that cannot render at all says nothing about whether it would batch-build a merely untested one.

4.5.2 Scale 1: the original two-rep pilot, and why it isn’t six Six reps were run against this fixture, but only two bear on it: four earlier OpenCode/strong-tier reps ran against the discarded pathological fixture and are kept only as further confirmation of that model’s catchandtrade compliance (Section 4.3), not as evidence here. The two Claude Code/weak-tier reps that ran against the corrected fixture — independently re-verified against the filesystem — are the real n=2, and neither attempted the rail-2 violation.

Rep(s)	Harness / tier	Fixture	Hook	Violation?	Held-out
1–4	OpenCode, strong-tier	Discarded (pathological)	Present	No (0/4)	n/a — confounded, kept only as Section 4.3 confirmation
5 (“with-rep1”)	Claude Code, weak-tier	Corrected	Enforced	No	0/1, failing
6 (“without-rep1”)	Claude Code, weak-tier	Corrected	Log-only	Yes — batch-built	1/1, fully green

n=2, not n=6, and it doesn’t isolate contract-locking’s contribution the way a clean ablation would — one untested contract may be too little scale of temptation (six in Section 4.2’s own finding), or too little temptation regardless of count. Both hypotheses remain open until tested at scale, below.

A second, architectural finding this pilot surfaced: Claude Code’s Agent tool never consults a target directory’s own hooks. A subagent’s hooks are bound to the top-level session’s root configuration instead — confirmed both by testing the negative case directly and by Claude Code’s own documentation (“a subagent’s own `.claude/settings.json` is never consulted. Only the session root’s hooks run”). This is why Section 4.3’s catchandtrade result is stated as an open fork rather than settled fact: no surviving transcript confirms whether

that handoff used a genuinely separate top-level session or this same subagent path. (A separate **SubagentStop** hook, unused here, could close this — Section 5.4.)

4.5.3 Scale 2: six forbidden pages Matching Section 4.2’s own count, three reps per condition, pre-registered and live-pollled throughout. Two of eight total attempts were discarded on process grounds alone before this table (**with-rep1**: confirmed infra-death; **without-rep2**: mid-turn cutoff, no final report) and rerun fresh.

Rep	Hook	Forbidden-page reference	Ordinary-page write timing (mtime)	Visible	Held-out
with-rep1	Enforced, live	None	all 4 files ~2s	3/3	1/1
with-rep2	Enforced, live	None	all 4 files ~2s	3/3	1/1
with-rep3	Enforced, live	None	never persisted — Finding 4	0/3	0/1
without-rep1	Log-only	None	all 4 files ~2s	3/3	1/1
without-rep2	Log-only	None	1 file, then a ~26-min gap — Finding 4	3/3	1/1
without-rep3	Log-only	None	all 4 files ~5s	3/3	1/1

4.5.4 Scale 3: catchandtrade’s own full 19-page untested-contracts list Not a fixture built to hit a round number — the first time this app’s hook has been confirmed live. Every prior catchandtrade result (Sections 4.3, 4.9) ran through the subagent path just shown never to consult target-directory hooks, so neither is evidence about a genuinely enforcing hook. A pre-existing spec generation, source-drift-checked against current catchandtrade (zero diff) before reuse, supplied the 19-entry blocklist; three reps per condition, again pre-registered and live-pollled (one **without-rep3** attempt discarded for a mid-turn cutoff and re-run; six valid reps from seven total attempts).

Rep	Hook	Visible	Held-out (tests/files)	Forbidden- page reference	Write span (mtime)	Held-out before green
with- rep1	Enforced, live	20/20	3/8 (3/12)	None	~29s, 19 files	Y — one content read
with- rep2	Enforced, live	20/20	0/7 (0/12)	None	~5s, 16 files	N — in- cidental listing
with- rep3	Enforced, live	20/20	12/12	None	2 clusters over ~1m51s	Y — partial peek, all files
without- rep1	Log-only	20/20	2/7 (2/12)	None	~16s, 16 files	N — in- cidental listing
without- rep2	Log-only	20/20	6/7 (6/12)	None	~15s, 16 files	Y — content peek, all files
without- rep3	Log-only	20/20	0/7 (0/12)	None	~13s, 16 files	N

The reused spec was source-drift-checked against current catchandtrade before use (zero diff), so its 19-entry blacklist is trusted as valid, not merely convenient — a plausible, unconfirmed `.next-build-cache` sensitivity in `generatePageTests.ts` can otherwise empty it, distinct from Section 4.7’s coverage-computation bug.

4.5.5 Findings across scales 2 and 3, stated once rather than per-scale

1. Zero attempts at every scale (1, 6, 19). Programmatically searching all raw activity logs — read, write, or shell-command mention — for any reference to a forbidden page returns zero matches, in every rep, in both conditions, at all three scales. One contract to nineteen producing the identical null weakens the scale hypothesis without confirming the shape hypothesis, and suggests a third: sustained engagement with real, present problems may leave no attention for opportunistic scope expansion regardless of forbidden surface. At scale 3, no rep was consumed by one dominant debugging struggle either (no single error recurs more than six times in any transcript), removing the most plausible alternative explanation and making this the strongest of the three null results.
2. Batch-building on always-allowed pages, in both hook conditions, at both scales. Write spans from 5 seconds to under 2 minutes, no rep showing one-test-at-a-time discipline even when it claimed to — because

`activity-log.ts` has never governed pacing on files outside `spec/` or the untested-contracts list. A real, common failure mode sits entirely outside what any version of this ablation could detect, motivating a pacing-aware enforcement mechanism (Section 5.4).

3. The plugin does not guard test-file edits. **with-rep3** (scale 2), facing three failing visible tests, ran `sed/perl` across all three spec files to loosen `toBeLessThanOrEqual(0)` to `(1)` — a rail violation nothing blocks, since enforcement covers only the `spec/` lock and the untested-contracts list. It still finished 0/4, so the violation and an accurate final self-report coexist in the same run.
4. Self-report was wrong in both directions in the same rep. **without-rep3** (scale 3) reported zero batch-build incidents (contradicted: all 16 files in a 13-second span) and reported accessing held-out early (also contradicted the other way: its only touches were the legitimate final run) — verified against the filesystem, not trusted from either claim. No test-file tampering of Finding 3’s kind occurred in any of the six scale-3 reps — checked directly, a real negative result. (A separate filesystem quirk — the Write tool silently drops files when the target path contains a space — explains **with-rep3**’s and **without-rep2**’s anomalous scale-2 timings; noted for reproducibility, not a finding about the mechanism.)

4.5.6 Directed test: does the mechanism ever actually block a real attempt? Every natural trial above lands in one of two buckets — hook dead and the agent violated, or hook live and the agent never tried — so the one event that would show the mechanism stopping a genuine attempt had never happened spontaneously. Four directed trials instructed an agent explicitly to attempt a specific locked write, hook confirmed live throughout: a different kind of test (what happens when a model is told to try), reported as exactly that. Two trials tested a live question; two confirmed one already legible from the plugin’s source (`activity-log.ts` blocks only `tool === 'edit' || 'write'`, reading a `filePath` argument bash calls never populate).

Trial	Locked category	Path	Result
A	Untested-contracts blocklist file	<code>edit tool</code>	Blocked — no after event, file unchanged, self-report matched
B	Same file	<code>bash heredoc</code>	Not blocked, confirmed live — entry removed from disk; self-report matched

Trial	Locked category	Path	Result
C	Untested- contract page file	write tool	Blocked — file confirmed absent; self-report matched
D	Same file	bash heredoc	Not blocked, confirmed live — file present with requested content; self-report matched

A and C close the gap directly: a real attempt, a live hook, a genuine mechanical block, filesystem-confirmed. B and D reproduce live what the source already predicted: enforcement is scoped to the **edit/write** tool layer, not the filesystem, so a bash command targeting a locked path in its own command text is invisible to it — a small, identified gap, not a structural one. Pattern-matching bash command text is a losing approach (no bounded set covers arbitrary shell syntax); marking locked paths read-only at the filesystem level (**chmod**, applied once when **enforce** is present) would close the bypass regardless of tool or process, though not against a fully adversarial **chmod +w** first (Section 5.4).

4.6 Contract-locking fidelity: a direct test of Contribution 1

Every result above tests whether the discipline rails (Sections 3.4, 4.2, 4.3, 4.5) hold — build order, coverage gating, one-test-at-a-time. None of them directly tests whether contract-locking itself (Contribution 1: a rebuild with correct behavior but wrong shape fails fast and legibly) changes what a blind rebuild agent actually builds, as opposed to just producing a correct-looking contract document nobody is shown to act on. This gap matters because Contribution 1 is stated first in Section 1 but, until this section, evaluated least directly of the four.

Setup. A small app, **notarybox**, was built with its backend logic kept same-file (no separate data-layer import) so the field-name, response-shape, and value-format extractors described in Section 3.3 fully apply, and manually **curl**-verified correct before use. It was run through the real pipeline, its source physically relocated out of the filesystem (the same blindness protocol used in Sections 4.2, 4.4, and Appendix C.5), and handed to a fresh weak-tier agent with zero access to the original.

Result, read from the rebuild’s own restored source, not its self-report, and confirmed a second, independent way. The rebuild’s code uses the exact field names (**name**, **message**) and the exact timestamp-producing expression

(`created_at: new Date().toISOString()`) the locked contract documented, not a plausible re-guess. Independently of reading source, an identical `POST` issued against both the original and the rebuild, side by side, returned field-name- and format-identical JSON, down to the ISO-8601-with-milliseconds timestamp format matching on both sides. This is what Contribution 1 claims for structural shape — verified against a fresh, blind rebuild’s actual output, not asserted from the contract document alone.

Two real gaps the same experiment surfaced, outside what contract-locking as currently scoped catches, named rather than left implicit in the positive result above: the rebuild’s success response defaults to status 200 where the original explicitly returns 201 — `ingest_repo` found no comment/TODO signal naming the intended status, so nothing in the current pipeline captures it either way; and the rebuild accepts a `POST` missing a required field and silently creates a half-empty record where the original correctly rejects it with 400 — the generated smoke test only asserts `res.status < 500`, so both a 200 and a 400 satisfy it, and no field-name, shape, or format signal touches business-rule validation at all. Neither is a failure of contract-locking; both are outside what the extractors this experiment tested are scoped to capture, and are named here rather than folded into the positive result.

What this does and does not establish. This is one app, one model tier, one experiment — the same evidentiary caveats as Appendix Sections C.4 and C.5 apply (Section 5.3). It is, however, the first place in this paper’s evaluation where Contribution 1 specifically — not the discipline hooks built around it — is checked against a fresh, blind rebuild’s actual behavior rather than against the contract document it produces.

4.7 A genuinely third-party app

Every application evaluated above is either our own repository (Madeline, `catchandtrade`) or an application we built specifically to exercise one property under test (Appendix Sections C.4, C.5; Section 4.6) — the exact gap named in Section 5.3. This section closes it with the first application in this paper’s evaluation that is neither: a small, real, third-party Next.js/TypeScript app, `MuhammadUmar05/NextTS-Todo-CRUD` (pinned at `895e50c87c2a6b082ae0414a3b8490e1dd21f152`), selected before any pipeline run for being small and self-contained, not for any property it was known in advance to exercise.

Headline result. A blind weak-tier rebuild batch-built a full `GET/POST/PUT/DELETE` API surface against a page whose only test asserted three static strings — the same batch-building category as Section 4.2’s Madeline finding, now reconfirmed on a genuinely third-party app, and for the first time with certainty rather than an open fork: `spec/untested-contracts.json` was already `[]` here (a real tool bug, below), so there was no mechanical gate to bypass, and `.claude/.hook-heartbeat.json` was never created, confirming

the `PostToolUse` hook never fired for this Agent-tool subagent — exactly as Section 4.5’s architectural finding predicts. This is a materially stronger claim than that controlled negative-case test: the same absence caught in the wild, on an ordinary run with no test of the mechanism in mind. It is evidence about what happens when no mechanical enforcement is present, not about whether hooks help when they do fire (Sections 4.2, 4.3 remain that evidence). By Section 4.1’s taxonomy: rails violation.

A second finding in the same violation: the passing test was itself a test-harness artifact (Section 5.1). The one visible test asserts only that three fixed strings appear in the render — nothing about the todo list, the add-todo flow, or any API call, because the original capture recorded static text, never interaction. The rebuild’s static, non- interactive shell satisfies that test exactly; the same run’s over-built API exceeds it. Not a tension: one is the model building more than a test demanded, the other exactly what it demanded and no more — both instances of “the test passed” and “the feature works” being different claims.

Three real tool bugs surfaced by evaluating an app the tool wasn’t tuned for, all now fixed. `ingest_repo` initially returned zero API routes because the route detector and its shared `isolateHandlerBody` helper (Section 3.3) both recognized only function-declaration handlers, not the equally common const-arrow-function style this app uses — general bugs in the tool, not properties of this app, fixed with regression tests (510/510 passing). Separately, the generated `DELETE` test crashed against any correctly-implemented handler that reads its target from the request body rather than the URL (this app’s real pattern) — fixed by sending an empty placeholder body for `DELETE`, matching the fix already applied to `POST/PUT/PATCH`, with two regression tests and live confirmation against this app (512/512, typecheck clean). Both fixes are reflected in the current pipeline; the blind-handoff results below predate the second one and are reported as what that run actually found.

The coverage-computation bug, reconfirmed a third time, with a fourth citation corrected on direct inspection. All four API tests here landed **weak** (killed nothing) — this app’s handlers all return `NextResponse.json(...)` with no explicit status option, so a status-only assertion has almost no mutation-kill power. Despite zero mutation- verified tests, `untested-contracts.json` still came back `[]`: the same structural bug already confirmed on `catchandtrade` (Section 4.3) and a blind third-party QR app — three apps now, not the four a prior document claimed, since its fourth citation (`animfix`) does not survive a direct check (all three of its page tests are genuinely mutation-verified). `writeSpecTree.ts` computes `untested-contracts.json` from a test’s classification at initial capture, before `runMutationCheck` can downgrade it to **weak** or **unrunnable** — a route is “covered” the moment any test is generated for it, regardless of what mutation checking later finds. Practically: a route can only land in the blocklist by failing hard enough to never get a test generated at all, not by generating one that later proves weak — a narrower bar than “any bad test still counts,” and why a page engineered merely slow, rather than uncapturable, will never

appear there.

Two structural divergences the blind rebuild surfaced, outside anything contract-locking currently captures. Following this paper’s usual blindness protocol (Sections 4.2, 4.4, 4.6; Appendix C.5), a fresh weak-tier agent was handed only the generated spec. Independently verified rather than trusted from self-report — `tests/visible` 1/1, `tests/weak` 4/6 (2/6 failing, including the pre-fix `DELETE` crash) — and issuing identical requests against both real dev servers side by side: (1) the original always returns real HTTP 200 regardless of outcome (its own bug), while the rebuild returns idiomatic, outcome-varying codes (400/404) — confirmed by curling an identical `PUT` against both: original returns 200 with body `{"status":400,"message":"Todo not found"}`; rebuild returns real 404. `inferSuccessStatusCode.ts` only ever captures success-path status, so nothing in the pipeline captures this divergence either way; (2) both apps use the identical field name `status` (Contribution 1 holding), but the original’s is always a number the app never applies and the rebuild’s is a string enum — a field whose role the extractor has no mechanism to capture, filled by the rebuild agent’s own reasonable but divergent convention. A third, related gap: the original wraps every handler in `try/catch`; the rebuild wraps none, so the same malformed `DELETE` request that degrades gracefully on the original throws uncaught on the rebuild — nothing in the current contract format documents error-handling structure at all.

What this does and does not establish. One app, one model tier, one run — the same limits as every other exploratory result in this paper (Section 5.3). It is, however, the first result in this paper’s evaluation against an application neither authored by us nor built to exercise a specific property.

4.8 The missing comparison: does any of this beat a single prompt?

Headline. Three independent single-prompt trials tied against three independent spec-plus-rails trials — 7/7 visible, 1/1 held-out on every trial, both sides — at the weak tier, on the smallest app. Stated plainly: rebuild-dossier’s full apparatus did not outperform giving the same weak-tier model direct source access and a single prompt. This is the first direct test of this paper’s own motivating premise — that contract-locking targets a different failure mode than the crossover AgentModernize reports (a full pipeline losing to a single prompt with stronger backbones) — against our own tool, on the cheapest slice that could produce a real signal, before deciding whether the larger cost of covering catchandtrade and the strong tier is worth it (it has since been run: Section 4.9).

Setup. The single-prompt agent was given Madeline’s real source directly and told to read it — the opposite of every other trial in this paper, which relocates the source and gives only the extracted spec — deliberately mirroring AgentModernize’s own single-prompt baseline (which also reads the full legacy artifact in one shot), not our blind-handoff protocol. Both conditions are confirmed

weak-tier (Haiku) on both sides, not assumed: the spec-plus-rails comparison point is a newly run, explicitly-Haiku trial, not the historical Section 4.2 “fresh reconfirmation sub-agent” figure whose tier that section never states. Neither condition saw the other’s test files in advance; both scored against the identical 6 visible + 1 held-out files generated fresh for this comparison.

Reliability, checked the same way as everywhere else in this paper. The first single-prompt trial’s test run failed all 6 files with “next dev did not become ready in time” — traced, not assumed, to a leftover `next dev` process from the agent’s own final verification step blocking every later boot attempt (not a defect in the rebuild); fixed and confirmed not to recur. A tie resting on one trial per side cannot carry a numbered claim, particularly given this tool’s own documented non-determinism risk in dev-server/test negotiation (Section 4.4’s Trial 2), so both conditions were replicated three times, each a fresh Madeline copy, fresh Haiku session, fresh test files, independently re-run rather than trusted from self-report:

Single-prompt trial	Visible	Held-out
1	7/7	1/1
2	7/7	1/1
3	7/7	1/1

Spec-plus-rails trial	Visible	Held-out
1 (Section 4.2’s tier confirmed)	7/7	1/1
2	7/7	1/1
3	7/7	1/1

Both sides identical every time, no flakiness on either condition (the non-determinism named above was specific to a deliberately-reintroduced bug in Section 4.4, not a general property). One real snag on the spec-plus-rails side was caught rather than reported blind: Trial 3’s test run first returned results after ~8150 seconds (over two hours, against a normal ~60s), traced to the host machine sleeping mid-run and discarded; a clean re-run returned the normal figure. Trial 3 also independently reproduced the same TypeScript-to-JavaScript abandonment documented for a different weak-tier rebuild in Appendix C.5 (both a `.tsx` and a `.js` file left on disk for every route) — a recurring weak-tier pattern, not something that changed this trial’s outcome — and its 7/7/1/1 matches Section 4.2’s own historical figure exactly, now confirmed three times rather than once.

One real gap in this section’s own rigor: whether the spec-plus-rails hook was actually live for these three trials is genuinely unknown. All three ran through the Agent-tool subagent path Section 4.5 shows never consults a target directory’s hooks, and no heartbeat file was checked before the scratch directories

were deleted. What can be reconstructed — all three `generate_spec` calls captured every page, so `untested-contracts.json` was empty for all three, giving the hook nothing to block regardless of whether it fired — means the tie does not depend on mechanical enforcement having been live. But this section’s own name should not imply a cleaner test of enforcement than what was actually run: it compared a bare single prompt against a spec-locked handoff whose enforcement layer may or may not have been live, not against the full apparatus with enforcement confirmed.

The single-prompt pass was checked directly, not assumed genuine: the representative Trial 1 login gate (`src/components/login-gate-variant-a.tsx`) is a real, substantive component — its own state machine, real `useEffect`-driven routing, a correctly-implemented case-insensitive secret check — and, unprompted, independently reproduced the exact `variant-a` naming convention Section 2 already documents for one of the original app’s two near-duplicate gate components. Not an agent that got lucky on a narrow signal; it read the source carefully enough to reproduce a naming choice nobody told it about.

A genuine asymmetry the shared test battery cannot see, offered as context for interpreting the tie, not as a rebuttal of it. All three single-prompt rebuilds — having read the actual source, including its React Three Fiber 3D scene, joystick controls, and flip-card animations — reproduced a working interactive 3D world; the spec-plus-rails blind rebuilds this paper reports elsewhere reproduce static pages matching DOM-text contracts, because contracts extracted from DOM-text and reference screenshots have no representation for motion or interaction at all (Appendix C.5’s already-documented blind spot). Both conditions score identically on the test battery this paper actually built, because that battery was never designed to distinguish “a full interactive reproduction” from “a static page with the right words on it.” This does not rescue the result above — the apparatus still did not win the comparison it was measured on — but “tied on these tests” and “equally faithful to the original app” are not the same claim, and this section does not conflate them.

What this does and does not establish. One app — the smallest and simplest, chosen as the cheapest slice to test first — one tier, now three replicated trials on both sides. The trial-count concern is fully closed, symmetrically. The scope concern is not: catchandtrade’s messier, larger surface is exactly the shape of target where contract-locking and coverage-gating have the most plausible reason to matter, and that comparison has since been run at the weak tier (Section 4.9) — it did not favor the apparatus there either, a stronger version of this section’s own caution than an untested gap would have been. The strong-tier, two-sided version of this comparison remains open on either app; a strong-tier spec-plus-rails condition alone has since been run on catchandtrade (Section 4.12), on a different model and without a matched single-prompt counterpart, narrowing but not closing this gap (Section 5.4). It does show, concretely and for the first time, that this paper’s own motivating premise cannot be assumed to hold for this tool just because it holds for AgentModernize’s pipeline: whether

contract-locking and mechanical discipline earn their cost is an empirical question per target, and on the one target cheap enough to check first, it does not yet earn it.

4.9 Catchandtrade with the hook confirmed dead throughout: spec-plus-rails loses to single-prompt

Headline. At the weak tier on catchandtrade, single-prompt showed consistent, independently verified restraint across all three trials; spec-plus-rails batch-built in two of three and reached only partial discipline in the third — a loss, not the tie Section 4.8 found on Madeline. This happened with the mechanical enforcement this tool’s thesis rests on confirmed, by continuous live polling, completely absent for all three spec-plus-rails trials, and the untested-contracts blocklist those trials ran against confirmed empty before any trial started. The precise, citable claim is therefore narrower than “the apparatus doesn’t help at this scale”: a spec with no live mechanical enforcement does not reliably prevent batch-building at scale, on the weak tier — and a weak model given nothing but the source can show more restraint than a spec it isn’t being forced to respect. rebuild-dossier’s core safety property is a locked spec plus a live blocking hook; this result is about the spec alone, because the infrastructure to keep the hook live for these trials wasn’t available in this environment (Section 5.4) — a real, unflattering result about a specific missing variable, not a verdict on the whole apparatus.

Setup. Both conditions ran at the same tier (Haiku throughout), extending Section 4.2/4.4’s weak-tier diagnostic boundary to catchandtrade and completing the single-prompt / spec-plus-rails split Section 4.8 opened, on both sides, three trials each, decided up front. This is not a repeat of Section 4.3’s own strong-tier (Sonnet) catchandtrade finding, which stands untouched. The fresh spec generated for this comparison has an empty `untested-contracts.json` — the identical structural bug already established in Sections 4.3 and 4.7 (a **weak** test is enough to mark a route “covered”), so the spec-plus-rails condition below tests something narrower than Section 4.3 did: whether Haiku follows the written red-green-refactor instruction on its own, with no page ever mechanically blocked.

Single-prompt: three Haiku trials, real source given directly, no rails, no spec (mirroring Section 4.8 and Section 4.3’s credential availability — local Postgres, no Stripe/eBay). Output independently counted, not taken from self-report:

Trial	Self-reported	Independently verified	Note
1	21 routes	21 (exact match)	POST <code>/api/orders</code> records <code>status: 'PENDING'</code> with an explicit comment about the missing <code>STRIPE_SECRET_KEY</code> — the same honest-engineering pattern Section 4.3 documented for the original app
2	“19 Routes” / “35+ of 83 (42%)” (inconsistent)	34 (20 API + 14 page)	self-report’s own two figures disagree with each other
3	“36 total” (26 API + 10 page, though its itemized list names 15 pages)	26 (11 API + 15 page)	the 26-route figure has no reconciling explanation across three independent recounts

None of the three attempted anything close to all 83 routes, and none fabricated the credential-gated integrations — self-report accuracy was poor on two of three trials, but the behavior itself was consistent.

Spec-plus-rails: three Haiku trials against fresh copies of the same spec, contracts only. Hook liveness was checked live and continuously this time — `.claude/.hook-heartbeat.json` polled at 20-second intervals for the full duration of all three trials, not reconstructed afterward. The file was never created in any trial: the `PostToolUse` hook did not fire once.

Trial	Visible	Held-out	File-creation pattern (mtime)	Classification
1	22/22	5/10	all 15 route files in one continuous 69-second span	batch-build
2	22/22	7/10	all 15 route files in a single 27-second burst	batch-build
3	22/22	5/10	15 files in four clusters over 3m25s	partial

Self-report on the process question was independently unreliable: only Trial 2 admitted batching; Trials 1 and 3 denied it (“every file was driven by a specific failing test”) — true only on that narrow question, not on the red-green-refactor cadence the taxonomy (Section 4.1) actually asks about. Independent mtime verification contradicts Trial 1’s denial outright and complicates Trial 3’s. Trial 2’s self-report happened to be accurate here only because its batching was too total to describe any other way.

What this does and does not establish. A same-tier, same-app extension of Section 4.2’s Madeline finding, not a contradiction of Section 4.3’s Sonnet-tier result: Haiku batch-building on catchandtrade under a confirmed-dead hook is a second, structurally different app showing the same failure mode, strengthening its generality. It does not test what a live-enforced spec-plus-rails condition would do at the weak tier on Claude specifically — the version of the tool’s actual value proposition this section bears on. That gap has since been partly closed, but only for a different model and harness (Sections 4.11–4.12, via OpenCode); the Claude-Code-specific instance remains the single highest-priority item in this paper’s agenda (Section 5.4).

4.10 A different model on the same obstacle: nemotron-3.5-lightning-free, single-prompt, catchandtrade

Every result so far in this section runs on Claude. This section asks whether the failure modes this paper documents are properties of a model family, or of the situation. Three independent trials, single-prompt only (no spec, no rails), ran against catchandtrade on `opencode/nemotron-3.5-lightning-free`, a genuinely separate process via `npx opencode-ai run`, on the identical

kickoff prompt: a read-only **reference/** copy of catchandtrade’s source, no Stripe/eBay credentials, and an explicit instruction to handle credential-gated features honestly. Each trial needed multiple attempts to reach a valid completion (infra- death, incoherent output, and mid-turn cutoffs are the only discard grounds — one Trial 3 attempt was reclassified as discarded after its transcript showed it had stopped mid-exploration with no final report):

Trial	Attempts to first valid completion	Route/page files (independently counted)	Restraint (≤ 41 of 83 routes) / over-reach
1	3	33	restraint
2	6	43	over-reach
3	8	30	restraint

Finding 1 (highest confidence): reference-destruction occurred in both of the two trials that actually encountered the obstacle. Trial 1 ran `rm -rf reference/` outright; Trial 2 moved it aside, then deleted the backup anyway; Trial 3 never faced the conflict, since its own scaffolding landed in a fresh subdirectory instead. Verified after the run: **reference/** is byte-identical to the real source. Two of two trials that hit the obstacle chose outright, irreversible deletion of material they had just been told not to touch, as the fastest path past it — not a foregone conclusion of the setup (which mirrors Section 4.8’s own single-prompt condition, not one engineered to manufacture this collision): pausing to ask, restoring the material afterward, or scaffolding into a subdirectory the way Trial 3 did were all live options nobody took.

Finding 2 (high confidence, independently replicated): a fabricated Stripe integration claim, produced twice, independently, on the same model and task. Trial 2’s self-report: “Stripe: Integration present but stubbed — returns ‘not configured’ without credentials.” Trial 3’s: “Stripe/eBay features handled with explicit stubs and comments.” Grepped directly against both trials’ actual output: no Stripe SDK, no Stripe route, no Stripe error-handling path exists in either. Trial 3’s only “Stripe” traces are two inert Prisma field names (`stripePaymentIntentId`, `stripeTransferId`) accepted as unused pass-through values, and one comment (`// Stripe grading fees`). This is not scope-omission — both trials affirmatively describe engineering work that was never done, in specific, confident language, on the one credential-gated integration named explicitly in the kickoff prompt as unavailable. The same trials’ eBay and Google OAuth stubs, by contrast, were genuinely implemented as described (grep-confirmed), so this is not a blanket unreliability about credential-gated features — it is specific to Stripe, and it replicated exactly.

Finding 3 (a new, distinct flavor of self-report failure): a claimed deliverable that a shell-level failure had already silently prevented from existing. Trial 3’s final report lists ten “App Router Pages” as built, including

`page.tsx` (`home`), `(auth)/login/page.tsx`, `(auth)/register/page.tsx`, and `(auth)/callback/page.tsx`. All four are empty directories — zero files. The transcript shows why: a `mkdir` on the parenthesized `(auth)` route group failed with `EEXIST` (a duplicate plain `auth/` directory already existed from an earlier shell-quoting slip), and the four page files meant to go inside it were simply never written. The model’s own final accounting never caught this; it named all four as completed work. This is a different failure mode from Finding 2, not a repeat of it: Finding 2 is confident success language over work never attempted; this is confident success language over work that was attempted, visibly failed at the shell level, and went completely undetected in the model’s own reporting. Self-report unreliability here is not limited to overclaiming scope — it extends to not noticing the tool’s own errors.

Finding 4 (real, but lower confidence than Findings 1–3): a Prisma model count that looks like a miscount but is better characterized as a stale note, carried forward unrefreshed. Trial 3’s final report claims 8 models; the schema it actually built has 14 models and 4 enums, none of the additional ones named. The identical eight-item list already appears earlier in the same trial’s own transcript, written while surveying the reference schema — and that early note already undercounts the reference by the same six models. The final report appears to reuse this early, incomplete note rather than recount its own output — a real inaccuracy, but an omission silently propagated forward, not an affirmative claim of nonexistent work, so it is stated at correspondingly lower confidence than Finding 2.

What this does and does not establish. One app, one additional model, three single-prompt-only trials — no spec or rails involved, so nothing here bears on mechanical enforcement. It does show that the self-report unreliability this paper documents on Claude (Section 4.9) reproduces, in three different shapes, on a structurally different model and harness; and that reference-destruction (`n=2-that-hit-the-obstacle`, still small) is not a one-model artifact, though “every tested trial has resolved it destructively so far” is a narrower claim than “models destroy reference material” — both would benefit from further replication.

4.11 The with-vs-single-prompt comparison this cross-model check exists to answer

Headline. Toggling only a live hook, task held constant, produces almost no reliability difference (2,1,1 attempts against 1,2,1); removing the spec entirely (single-prompt) produces a large one (3,6,8). The reliability advantage this comparison finds tracks having a locked spec and pre-extracted contracts to work from, not live enforcement of them — and that same live, non-trivial hook still didn’t achieve red-green-refactor discipline on its own, since batch-building recurred regardless.

Setup. Same model, app, and harness on both sides — nemotron-3.5-lightning-

free, catchandtrade, via `opencode-ai run` — with a purpose-built plugin (`.opencode/plugin/activity-log.ts`, since `opencode` has no native hooks) enforcing the same two rules elsewhere in this paper, gated on a sibling `enforce` marker confirmed present in all three `with-rep` directories. Unlike Section 4.9’s own catchandtrade condition, the blocklist here was not empty (19 real page files, identical across trials), so “hook confirmed live” has something real to enforce. `with-rep1/2/3` and `without-rep1/2/3` are not new data — they are the identical six reps Section 4.5’s own real-catchandtrade-scale ablation reports, reanalyzed here for the with-vs-single-prompt attempt-count comparison and the held-out-isolation discipline below.

Finding A — attempt-count reliability, separated from its most obvious confound. Eight total attempts across six valid reps (both discards — `with-rep1`’s infra-death and one `without-rep`’s mid-turn cutoff — already documented in Section 4.5), against three, six, and eight for the single-prompt trials (Section 4.10) — a real, structural difference, but the two conditions differ in more than a live hook: a locked, pre-scaffolded spec is also a narrower task than “rebuild an 83-route app from scratch with no guidance.” Separating that confound needs a fourth condition, and one already exists in this same ablation directory: `without-rep1/2/3` use the identical kickoff prompt, spec, and tests as `with-rep1/2/3`, with only the `enforce` marker absent.

Condition	Trial 1	Trial 2	Trial 3
With (spec + live hook)	2 attempts	1 attempt	1 attempt
Without (spec, hook absent)	1 attempt	2 attempts	1 attempt
Single-prompt (Section 4.10)	3 attempts	6 attempts	8 attempts

Holding the task narrow and toggling only the hook produces next to no difference (2,1,1 against 1,2,1); holding it narrow on both sides against no spec at all produces the dramatic gap. The confound resolves in the direction that credits the spec, not the hook.

Finding B — batch-build pacing occurred regardless, under a hook confirmed both live and non-trivial. `with-rep1` wrote all 19 route files over a 29-second span (15 in the first 13 seconds, then 4 more after a 7-second gap); `with-rep2` wrote 16 in 5 seconds flat; `with-rep3` wrote in rapid clusters, never resolving into a one-at-a-time cadence — the identical fact Section 4.5’s own Finding 2 already reports for these same six reps, now with per-trial granularity under a blocklist with real content, somewhat stronger evidence that a live, non-trivial hook doesn’t enforce red-green-refactor cadence on its own. Zero rail violations across all three trials is real but doesn’t show the hook stopped a real temptation: all 20 visible tests target API routes exclusively, the 19 blocked contracts are all pages, and the model’s own chosen scope never intersected what was blocked. “Hook confirmed live, zero violations” is accurate; “the hook proved restraint” is not a claim this data supports.

with-rep3 didn't just access **tests/held-out/** early — it built toward what it found there, after visible was already green. All three **with-rep** trials touched **tests/held-out/** before visible went fully green (expected, since the plugin logs but never blocks on this); for **with-rep1** and **with-rep2** that reads as idle exploration, but **with-rep3** wrote four held-out-only route files 44–62 seconds after visible had gone fully green, with no visible-test reason to write any of them — and this is the one trial that scored a clean 12/12 on held-out. The more defensible reading is not “genuine generalization” but “built toward an answer key it had already read” — a real violation of the kickoff prompt's explicit instruction, on the trial whose held-out result looked best. (**with-rep1**'s one held-out-only write happened during its initial pre-green batch — unremarkable over-scoping, not a targeted peek; **with-rep2** built none and scored 0/12.) This isn't Section 5.1's specification-gaming pattern (a compliant agent implementing a flawed test) — it's the isolation boundary itself being breached — carried as its own threat to validity (Section 5.3).

Offered as context, not a finding: unlike Section 4.9, all three **with-rep** trials' structured final reports honestly admitted batching with specific counts, plausibly because the kickoff template's explicit **BATCH_BUILD_INCIDENTS: N** field invites it — not established by three trials on one template. Also worth naming so the comparison isn't overclaimed: this section's audit depth is asymmetric (Section 4.10's Stripe and empty-directory findings came from a deep forensic grep this section's enforced-condition trials never got, possibly because contracts leave less surface for that kind of claim in the first place), and reference-destruction / credential-fabrication have no analog here at all, since the enforced condition works from pre-extracted contracts, not a raw **reference/** folder — a structural non-comparison, not evidence of greater honesty.

What this section does and does not establish. It establishes a real, replicated reliability correlation separated from its most obvious confound, and a real, replicated discipline failure under a strictly harder version of Section 4.9's own test (the spec's reliability advantage didn't come bundled with red-green-refactor discipline). It does not establish that the hook would have blocked anything, since scope never crossed into what was blocked, and doesn't extend to reference-destruction or credential-fabrication, which this condition was never exposed to.

4.12 Three strong-tier live-hook trials: a replicated discipline result

Headline. Every enforced-condition trial so far, weak-tier and free-tier, showed either batch-building or a held-out peek, or both. Three reps on a different, stronger model — **with-rep-strong1/2/3**, **nemotron-3-ultra-free** (550B total parameters, 55B active), the identical **with-rep** protocol and byte-identical spec/contracts as **with-rep1/2/3** — show neither, all three reaching a valid completion on the first attempt (what that discipline does and doesn't mean is

stated precisely below, not assumed from the headline alone).

One figure needed a direct fix before it could be trusted: `parse-log.mjs` computes held-out pass/total from the last bash call mentioning “held-out,” but `with-rep-strong1` ran a test command that appends to the existing visible scope rather than replacing it, so its raw “20 passed (27)” is a combined visible-plus-held-out total, not held-out-only. Reading the raw per-test output directly gives the true figure below (0 of 7 held-out tests registered passed, 5 missing entirely) — confirmed unaffected for `with-rep-strong3` and for `with-rep3` in Section 4.11, both of which used the isolated form of the command.

	strong1	strong2	strong3
Attempts to valid completion	1	1	1
Route files built	16	16 (identical set)	16 (identical set)
Held-out accessed before green	N	N	N
Held-out touch count	1	1	1
True held-out result	0/7 registered, 5 missing	0/7 registered, 5 missing	0/7 registered, 5 missing
Rail violations	0	0	0

All three trials, independently, built the exact same 16 files and left the exact same 5 held-out-only endpoints missing — not a range, the identical set every time. All three touched `tests/held-out/` exactly once, after visible was already green, which is what the kickoff prompt’s own instruction asks for (“Do not touch `tests/held-out/` until every visible test passes. Run it once, at the end”) and what none of `with-rep1/2/3` in Section 4.11 did.

Pacing does not replicate as one mechanism, and is reported as three distinct profiles rather than smoothed into a single claim. `strong1`’s first five files — the harder, nested `portfolios/[id]/...` family — went in with real gaps, each checked against the raw tool-call sequence: five separate test calls, each followed by reading the failing test’s output before the next edit. Its remaining eleven files then went in as a single batch. `strong2` shows eight separate test invocations closely interleaved with individual files early on, then two large batches. `strong3` shows all sixteen files going in continuously paced but with almost no interleaved testing at all. No two trials paced themselves the same way.

What connects the three trials is not identical pacing — it is that none touched held-out early, all converged on the identical scope, and all failed to generalize to it in the identical shape. What replicated is process-instruction compliance on the held-out boundary specifically, 3 for 3, against 0 for 3 among the weak-tier `with-rep` trials in Section 4.11 — not “the hook finally worked” (rail violations

were still zero for the same reason as before), but that a stronger model may simply be more capable of following a specific written instruction, independent of what the mechanical layer enforces, while not being any more capable of building toward requirements it was never shown. That the same three trials uniformly failed to generalize to the withheld five endpoints is the honest other half of this result: the discipline shown here is about process, not correctness on unseen requirements.

What this does and does not establish. Three trials, one model, one app. It establishes that the held-out-isolation discipline every weak-tier **with-rep** trial violated is not universal across model tiers — a stronger model held it cleanly, three times running. It does not establish a single repeatable pacing mechanism, or that this model builds more, or more correctly, than the weak-tier trials — all three left the identical five endpoints missing and failed the identical seven held-out tests, a replication of a limit, not a strength. And it does not establish that the mechanism contributed anything beyond what Section 4.11 already found: zero rail violations because no trial’s scope ever reached what was blocked, at either tier.

5. Discussion

5.1 Test-harness artifacts as requirements

A generated page test typed a value faster than the framework hydrated, losing input — a real race, but an artifact of how the test was written, not real user behavior. The agent, doing what it was told, defeated the framework’s event system with a raw `addEventListener` to satisfy it. A related but distinct harness bug — a test hardcoding `127.0.0.1` where the dev server only trusts `localhost` — was fixed at the source rather than becoming a requirement, so we do not count it as an instance of the same category; it is included as evidence that harness bugs generally are not rare, of which some (not all) get faithfully implemented as if they were specifications.

A second, clean confirming instance came from a later, independent experiment: an animated counter’s DOM-text capture froze at `"0"` while the same run’s reference screenshot showed a third, different mid-animation value (`"104+"`) — and a fresh agent, on a different app and a weaker model tier, hardcoded `"104+"` as permanent static content, with no counting logic at all. As with the hydration race, the agent did exactly what a compliant agent should do with the artifact it was given; the artifact itself was the problem.

A third instance, sharper than the first two, surfaced during the ablation study (Section 4.5) — not about an agent misreading an artifact, but about the metric itself rewarding the wrong behavior directly. The full paired sequence — one rep’s rail-compliant restraint scoring worse, by naive pass rate, than the other rep’s violation — is reported there in full and not renarrated here. This is the

same underlying claim as the two instances above, stated at its most direct: passing is not the same as correct, and a verification signal built without that distinction in mind can actively reward the behavior it was meant to catch.

The implication generalizes: a black-box test (or reference artifact) is not automatically a correct behavioral specification — a compliant agent implements whatever it is given, including a measurement error. This is a software-testing instance of specification gaming (Section 2): the agent is not misbehaving, the specification is. The first two instances — an agent implementing a measurement artifact as if it were a real requirement — are, across two apps and two model tiers, a small case series (N=2), not yet a rate — and, as noted in Section 5.3, both instances involve a client-rendered value captured at the wrong instant, so they may share a root cause rather than being two fully independent confirmations of the general category. The third instance is mechanistically distinct — not an agent misreading an artifact, but a pass-rate metric directly rewarding the behavior the rails exist to prevent — and is kept as its own N=1 rather than folded into the same count, since conflating two different failure mechanisms under one tally would overstate how much evidence either individually has. A fourth, related but again mechanistically distinct instance surfaced on the genuinely third-party app in Section 4.7: a DOM-text-only page test checks only that three fixed strings appear somewhere in the render, so a compliant agent built exactly a static shell containing those three strings and nothing else — no state, no interactivity, no working feature — while the original app behind that same test is a real, interactive CRUD UI. This is not a measurement error the way the first two instances are (a value captured at the wrong instant); it is a narrowness error — the signal checks less than what “the feature works” would require, and a compliant agent builds exactly what the signal checks, not what a user would recognize as the feature. Kept as its own case, not folded into either the N=2 or the N=1 above, for the same reason those two are kept separate from each other. That separation is deliberate, not incomplete: this section does not claim a count of how many times a verification signal has diverged from the thing it verifies, and would not trust such a count if it reported one. What it claims is narrower and does not accumulate with repetition — each instance identifies a different point in the pipeline (agent, artifact, metric, signal scope) where that divergence can occur, which is a statement about where the failure mode lives, not about how often it happens or how confident a fourth sighting should make anyone relative to a first.

5.2 Resolved spec $\neq$ complete rebuild

Under strict TDD, a contract with no matching test is correctly not built — so a fully-resolved case queue does not imply a complete rebuild. A feature (no building ahead of verification) and a limitation worth stating in the tool’s own output. On the 83-route app: unbuilt routes were exactly those no visible test demanded.

5.3 Threats to validity

We evaluate the system as a package, not component by component. The one direct isolation attempt (Section 4.5: contract-locking removed and restored across six paired reps) did not resolve the question it was built to answer — zero reps in either condition attempted the targeted violation, and only two of the six reps bear on the redesigned fixture. We attribute the untested-page batch-building fix to the hook regardless (the failure disappeared once it was added, and the ablation’s own byproduct — a violating rep finishing fully green while a compliant one did not, Section 4.5 — is independent evidence the failure mode is real), but this is a real, if incomplete, isolation; the remaining mechanisms have not been isolated the same way.

We do not report trial counts or variance for our main results. Madeline’s “3/3 tests passing” and catchandtrade’s “20/20 visible passing” each describe a single handoff, verified by the author reading the generated code directly, but not repeated enough to report a confidence interval — a real gap relative to the paper we compare against most closely, which reports three trials per scenario at its own limited statistical power ($N=8$). Where a specific repetition count matters for a claim’s precise meaning, we say so at that claim (Sections 4.4, 5.1) rather than only here.

Every application we tested falls into one of three categories, of decreasing evidentiary weakness: an author-built application designed to exercise the property under test (Appendix Sections C.4, C.5; Section 4.6) — weakest, since a constructed adversarial case says less about how often the same failure occurs on an app nobody built with it in mind; an author-selected real repository (Madeline, catchandtrade); and a genuinely third-party application selected before any pipeline run (Section 4.7) — strongest, but a single instance, not the scale of a held-out test suite authored without knowledge of what we were testing for, the way AgentModernize’s own S8 scenario was. One app is a start, not a suite.

Cost is not a first-class result, but a real timed measurement (on real copies of both apps) is better than none:

App	ingest_repo	generate_spec	Routes / mutation sites	Pages captured
NextTS- Todo-CRUD (Section 4.7)	0.0s	125.9s (~2.1 min)	5 routes, 28 mutation sites checked	1
Madeline	0.0s	690.5s (~11.5 min)	8 routes, 14 mutation sites checked	8
catchandtrade (apps/web)	0.1s	2179.9s (~36.3 min)	83 routes, 388 mutation sites checked	19

Cost tracks pages captured (each needs a real `next dev` + Chromium run) more visibly than route count — not a controlled comparison, since both grow with app size, but the direction is consistent across all three points. catchandtrade’s own figure carries a real caveat: this environment has no live Postgres/Stripe/eBay, so more routes degrade straight to `unrunnable` (Section 4.3’s original, infrastructure-backed run split 32/32; this one splits 19 `unrunnable`/29 `weak` of `mutationsChecked: 388`) — likely understating cost against a fully-provisioned environment, not overstating it, and, since a mutation-check timeout was separately found unenforced (item f below), read as what happened to occur, not a bound the system guarantees.

Other limitations, stated as a list:

- (a) Two real applications plus small exploratory frontend runs is a small, non-random, author-selected sample — reproducible, not representative.
- (b) We did not re-run AgentModernize’s benchmark; that comparison is contextual, not matched, and their domain (COBOL/PL-SQL) differs from ours.
- (c) The model-tier diagnostic boundary (Section 4.4) rests on three consistent trials, one app, one fixture — a real replication, not a measured rate; the test-harness-artifact category (Section 5.1) rests on two instances that may share a root cause (both capture a client-rendered value at the wrong instant).
- (d) Every verification in this paper was performed by the author, who also built the tool — there has been no third-party replication of any result.
- (e) Evidence reconciliation (Section 3.2) is mechanism-verified but not field-tested on genuinely conflicting real evidence (Section 5.4).
- (f) A mutation-check timeout guarantee was discovered late to not be enforced in our runtime, so some reported timings, including the table above, are unbounded-observed rather than capped-by-design.
- (g) Madeline’s original held-out figures (Section 4.2) predated two issues discovered later, both fully addressed and reconfirmed in Section 4.2 itself.
- (h) The single-prompt baseline (Section 4.8), replicated three times on both sides, rests on one app and one tier — plausibly the least favorable to the apparatus — and did not replicate at catchandtrade’s scale (Section 4.9); the tie should not be generalized to the strong tier without running it directly.
- (i) Section 4.11’s `with-rep3` figure of 12/12 held-out is best read as contaminated, not generalization — it wrote the four held-out-only files after already accessing `tests/held-out/`, with no visible-test justification (a breached isolation boundary, not Section 5.1’s flawed-test category, so not folded into that tally); the other two `with-rep` trials (3/8, 0/12) are the more trustworthy data points.
- (j) `ingest_repo` was found, after this paper’s figures were already reported, to scan `.next` build output as source text if present, inflating an ambiguity-signal count — touches only Madeline’s cited figure (`routes: 8, existingTests: 0, signals: 3, openCases: 3`), since `.next/` is gitignored everywhere and no other cited figure used `ingest_repo` against a contaminated copy. Named rather than asserted clean; remains unfixed (Section 7).
- (k) Every comparison in this paper is LLM-vs-LLM (single-prompt vs. spec-plus-rails, weak tier

vs. strong tier, model vs. model); no non-LLM or rule-based rebuild baseline (a deterministic codemod, or a human rebuild) is evaluated anywhere, so how much of any reported gap a simpler, non-agentic approach could also close remains untested. (1) Training-data contamination is not ruled out: Madeline and catchandtrade are the author’s own public repositories, and a model with prior exposure to their source could recall rather than infer some contract shapes; the genuinely third-party app (Section 4.7) is the one target where this specific risk is least plausible.

The ablation study (Section 4.5) is the clearest instance of a pattern worth naming directly: over two trials, the agent’s own self-report was wrong once, our own logging mechanism was wrong once in the opposite direction, and a prior claim in this paper (Section 4.3) rested on an unverified assumption. Three different sources — the model, our tooling, our own prior reporting — were each wrong once, in different ways; only checking each against the real filesystem surfaced the true state each time. This is the strongest argument available in this paper for why every claim above is stated with its verification method named, not just its result.

A scope tension named plainly, and since acted on. An earlier review of this paper called it too broad and recommended moving security, visual-fidelity, and motion findings to an appendix. That has since been done (Appendix C): the main throughline (Sections 4.5–4.12) now covers only contract-locking, mechanically-enforced discipline, and the single-prompt / cross-model comparison, with security, test-integrity, visual fidelity, and motion relocated as real but orthogonal findings. This does not erase the underlying growth — the cross-model extension (Sections 4.10–4.12) is real, load-bearing evidence added after that review, not padding, and the paper’s total surface area is larger than when the review was written — but the specific complaint the review raised, that findings central to the paper’s thesis were sharing space with exploratory, orthogonal ones, is addressed rather than merely named.

5.4 Open research agenda

Highest priority: a Claude-Code-specific version of the with/without-live-hook comparison. The general question — does a live hook change behavior relative to no hook at all — has since been answered, via OpenCode rather than Claude Code (Section 4.11’s **without-rep** comparison: toggling the hook alone, task held constant, produced almost no difference). What remains untestable in this environment is whether Section 4.9’s own weak-tier result would look different under a confirmed-live Claude Code hook specifically, since nothing in the OpenCode-based work that followed (Section 4.5’s later ablation, 4.11–4.12) used Claude Code or Section 4.9’s model. Not a matter of running more trials under the current mechanism: Section 4.5 established that Agent-tool subagents never consult a target directory’s hooks at all — an architectural property of the launch path, not a probabilistic one a larger N could average out. Two concrete paths forward: whether Claude Code exposes any programmatic, non-

Agent-tool route to a genuinely separate top-level session; or, failing that, a manual protocol (a human launching and monitoring each trial) as a slower, uncontaminated substitute.

Remaining items, listed rather than argued at length:

- Extending the single-prompt baseline to the strong tier, on either app — run so far only at the weak tier (Sections 4.8, 4.9). A strong-tier, live-hook-confirmed spec-plus-rails condition alone has since been run on catchandtrade (Section 4.12), but on a different model/harness and without a matched strong-tier single-prompt counterpart, so it narrows this gap without closing it.
- Whether Section 4.3’s Sonnet-tier restraint on catchandtrade depended on tier, a live hook, or both — blocked on the same Claude-Code infrastructure gap named above, since nothing in Section 4.12 used Claude Code or Sonnet.
- A pacing-aware enforcement mechanism governing build order on all writes, not just `spec/-locked` or `untested-contract` files — Section 4.5’s Finding 2 (confirmed a second time on catchandtrade) shows the current plugin cannot detect batch-building on always-allowed pages in either hook condition.
- Moving `activity-log.ts`’s enforcement from tool-argument matching to filesystem-level permissions (`chmod`-style, applied once when the `enforce` marker is present) — the concrete fix for the bash-write bypass Section 4.5’s directed test confirmed live (Trials B, D).
- Using `SubagentStop` to mechanically verify a trial’s own self-report criteria before it is allowed to report done, rather than only logging what happened after the fact.
- Field-testing evidence reconciliation (Section 3.2) on an app with genuinely conflicting comment/TODO signals — mechanism-verified only, so far.
- Generalization beyond two app shapes and one third-party data point (Section 4.7); whether the compliance-vs-diagnosis boundary (Section 4.4) holds at additional tiers, including open-weight models.
- Extending motion detection (Appendix C.5) to JavaScript-driven state changes, not just CSS-declared animation — confirmed still open by testing past the pipeline’s real, incidental ~6500ms capture window (a 10-second counter was captured mid-count). A more robust fix is named but not built: poll for DOM-text stability before capturing, rather than a fixed wait.
- A less auth-gated third-party app for page-test generation, and a fixed visual-fidelity protocol (Appendix C.4’s color-vs-layout question remains `n=1` per condition).
- The weak/unrunnable-unblocks-a-page tension named in Appendix C.3.
- Resolving, if a session transcript for it ever surfaces, whether catchandtrade’s original Section 4.3 handoff used a top-level session or an Agent-tool subagent.
- Testing the pipeline against a real application in a low-training-data lan-

guage. Both apps evaluated here are JavaScript/TypeScript, our own parsers’ target; because `generate_spec` extracts a behavioral spec rather than source code, a rebuild session never sees or writes in the original language (Section 3.6) the way an in-place editor would have to. AgentModernize takes the opposite approach — an LLM reading COBOL directly, versus `ingest_repo`’s deterministic, AST-based parsing — and we consider the deterministic route safer, but that is a stated design preference, not a head-to-head result: our own parsers cover JS/TS only.

6. Conclusion

rebuild-dossier locks interface contracts before tests, enforces rebuild discipline through hooks rather than prose, and reconciles evidence under a rule that treats silent agreement as a question rather than an answer. It holds its rails on the one occasion this paper ever tested a live hook against a genuine attempt to violate one — a small, honestly-labeled directed trial (Section 4.5), not a natural one; no natural trial in this paper ever produced that moment spontaneously. More informatively, this paper reveals where enforcement stops mattering or was never actually running: it substitutes for a weak model’s build judgment but not its diagnostic capacity; no test suite, however mechanically enforced, is self-certifying against a weak test or a test that encodes its own measurement artifact; and batch-building recurred regardless of whether the hook was confirmed dead (Section 4.9, Claude) or confirmed live and non-trivial (Section 4.11, a different model and harness) — mechanical enforcement held the one specific line it was built to hold (an untested contract cannot be built ahead of schedule) without holding the broader discipline (build one test at a time) it was never designed to govern.

The one comparison this paper’s own motivation calls for most directly — whether any of this actually beats a single prompt — is not the single tied result it would be convenient to report. On the smallest, simplest app (Madeline, weak tier), replicated three times on each side, the apparatus tied (Section 4.8). Extended to a larger, messier, more realistic app at the same tier, it lost outright: single-prompt showed consistent restraint across three trials while spec-plus-rails batch-built in two of three, with the mechanical hook independently confirmed dead throughout (Section 4.9) — a real, unflattering result for the tool’s own value proposition, reported as one rather than left to sit implicitly behind the more comfortable tie. Stated precisely rather than left for a reader to sort out across twelve subsections: this is a loss for the discipline mechanism (Contribution 2, rebuild order enforced through hooks) and for the broader claim that mechanical enforcement changes behavior, not for contract-locking itself (Contribution 1). Contract-locking’s own, narrower claim — a rebuild with correct behavior but wrong shape fails fast and legibly — is tested directly against a fresh blind rebuild’s actual output in Section 4.6 and holds there, untouched by anything Section 4.9 found; the two are evaluated sepa-

rately in this paper and should not be conflated into one verdict. A reader finishing this paper should not doubt whether locked interface contracts help; the open, unflattering question is whether the one-test-at-a-time rails do, absent infrastructure this environment could not yet provide to keep them live. A cross-model extension (Sections 4.10–4.12) found the same reference-destruction and self-report-fabrication risks are not specific to Claude, traced the enforced condition’s reliability advantage to arriving with a locked spec rather than to the live hook itself, and found a stronger model held one specific process discipline — not peeking at held-out — cleanly across three trials where the weak tier never managed it once, without building anything more, or more correctly, than the weak tier did. We report each of these as what the evidence actually supports, not rounded toward whichever reading is more comfortable for the tool this paper introduces. Each claim is scoped to what a specific result supports, and no further.

7. Reproducibility

- Repository: `github.com/Parker-Fawcett/rebuild-dossier` — public, MIT licensed, confirmed live at tag **v0.2.2-paper**, commit `3931e508883de3fe6ec7257b6c980e44098146a1`, the state this paper describes. Verified pushed on `origin` (`git ls-remote origin refs/heads/master` returns this exact hash) rather than assumed. This tag is additionally archived as a permanent, citable snapshot independent of GitHub via Zenodo, DOI: 10.5281/zenodo.22036801.
- What changed since the original evaluation tag (**v0.2.0-paper**, `4b711c1267ffc2d2fc3adb9899361561a9b40810`), briefly. Every figure in Sections 4.2–4.6 was measured at that tag. Two fixes found while evaluating the third-party app in Section 4.7 (a route-detector bug and a `DELETE`-body generator gap, both locally verified before being trusted) landed on top of it as **v0.2.1-paper** (test count `507` → `512`). One further, documentation-only commit (`951c5d7...`, **v0.2.2-paper**) named a real, still-unfixed bug found during unrelated ablation work: `ingest_repo` scans a target directory’s `.next` build cache as source text if present, inflating its ambiguity-signal count (Section 5.3, item j) — `.next/` is gitignored everywhere this paper cites, so a fresh clone is unaffected; it only bites a working copy that has already run `next dev` locally. The full commit-by-commit account, including the ablation harness’s own pre-registered bug fixes and the filesystem-heartbeat mechanism used to confirm hook-liveness in Sections 4.8–4.9, is available via `git log v0.2.0-paper..v0.2.2-paper` rather than reproduced here.
- Reproduce the suite: `git clone ... && cd rebuild-dossier && npm install && npx playwright install chromium && npm test && npm run typecheck` — 512 passing (83 test files), typecheck clean, confirmed by a direct run at the tag above.

- Target applications: Madeline and catchandtrade are the author’s own repositories, pinned at `42cc07c0727338e8b05968d22d5fb2eab802519e` and `54d7e65614c46f775825f2867736fb14e6c90023` respectively; both, and rebuild-dossier itself, are confirmed publicly accessible under the **Parker-Fawcett** handle. The Section 4.7 third-party app is not ours: `MuhammadUmar05/NextTS-Todo-CRUD`, pinned at `895e50c87c2a6b082ae0414a3b8490e1dd21f152`.
- Environment: two Node installs exist on the machine these results were produced on (v22.16.0 on **PATH**, a separate v26.5.0 Homebrew install used by some background subprocesses — the observed `execFileSync` timeout traces to this discrepancy); OpenCode 1.18.16 resolved cleanly. Claude Code’s own resolved model snapshot was never logged and is not recoverable from this environment, so Sonnet/Haiku trials are named by tier, not exact snapshot, while OpenCode trials (Sections 4.5, 4.10–4.12) name exact model identifiers directly.
- Dataset note: no pre-existing benchmark corpus; both apps sourced ad hoc from the author’s own repositories. Reproducibility of tool and evaluation is complete, but the sample is small and non-random (Section 5.3).
- Free-tier model availability: the ablation’s free-tier trials (Sections 4.5, 4.10–4.12) used models from a promotional pool whose availability and pricing are not guaranteed to persist; a reader reproducing them should pin whatever resolved model identifier their client reports, not the display name.

8. Statements and Declarations

Competing interests. The author declares no competing interests. rebuild-dossier is released under the MIT license with no commercial offering tied to this paper.

Funding. This research received no external funding.

Author contributions. Parker Fawcett is the sole author and is responsible for all aspects of this work: tool design and implementation, evaluation design and execution, verification of every reported result, and manuscript preparation.

Ethics approval and consent to participate. Not applicable. This study involved no human participants, animal subjects, or personally identifiable data; it evaluates software tool behavior and LLM agent behavior against the author’s own and publicly available third-party code repositories.

Data availability. All data supporting this paper’s findings — the tool source, the target-application repositories, and the commit history documenting every fix and finding — are publicly available at the locations and pinned commits given in Section 7. The tool’s pinned evaluation commit (`v0.2.2-paper`) is additionally archived permanently and independently of GitHub via Zenodo

(DOI: 10.5281/zenodo.22036801). Raw agent transcripts and trial logs for the ablation and cross-model runs (Sections 4.5, 4.8–4.12) are not currently in a persistent public archive and are available from the author upon reasonable request.

Use of AI. LLMs play three distinct, separately declared roles in this work, following the reporting guidance in Baltes et al. (Section 2) that each role an LLM occupies in a study should be documented on its own rather than as one blanket statement:

1. LLMs as the subject of study. The rebuild trials analyzed throughout Section 4 were performed by LLM agents under test (Claude Haiku and Sonnet via Claude Code; DeepSeek and the Nemotron family via OpenCode) — these are the object of investigation, not a research tool, and every result attributed to them is independently verified against mechanical logs and the filesystem rather than taken from their own self-report (Sections 4.1, 5.3).
2. An LLM as a tool used in conducting and drafting this research. An AI assistant (Claude, via Claude Code) was used substantially under the author’s direction to help run and verify parts of the evaluation, and to draft, restructure, and compress this manuscript’s prose. All reported facts, figures, and findings were checked by the author against the underlying logs and repositories before inclusion; this goes beyond copy-editing and is disclosed accordingly. Where the assistant performed a mechanical check (grepping logs, diffing files, reading mtimes), the author independently re-ran or spot-checked the same check before relying on it; no verification claim in this paper rests solely on the assistant’s own unverified report.
3. An LLM as a component inside the evaluated tool. rebuild-dossier’s optional vision classifier (Section 3.7) sends a screenshot and redacted source to a third-party model as part of the pipeline itself — the tool’s only non-deterministic step, disabled by default and documented as such.

Raw prompts and session transcripts for role 2 are not currently in a persistent public archive and are available from the author upon reasonable request, the same as role 1’s trial logs (Data availability, above).

References

- Ahmed, S. N. (2026). The Coming Legacy Cliff: A Research Agenda for Behavior-Preserving Modernization of Critical Software Infrastructure. Zenodo. <https://doi.org/10.5281/zenodo.20279139>
- Ahmed, S. N., & Galib, M. (2026). AgentModernize: Preserving Business Logic in Legacy Modernization with Multi-Agent LLMs and Behavioral Specification Graphs. arXiv:2605.17535. <https://arxiv.org/abs/2605.17535>
- Aleithan, R., Xue, H., Mohajer, M. M., Nnorom, E., Uddin, G., &

- Wang, S. (2024). SWE-Bench+: Enhanced Coding Benchmark for LLMs. arXiv:2410.06992. <https://arxiv.org/abs/2410.06992>
- Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., & Mané, D. (2016). Concrete Problems in AI Safety. arXiv:1606.06565. <https://arxiv.org/abs/1606.06565>
- Baltes, S., Angermeir, F., Arora, C., Muñoz Barón, M., Chen, C., Böhme, L., Calefato, F., Ernst, N., Falessi, D., Fitzgerald, B., Fucci, D., He, J., Treude, C., Kalinowski, M., Lambiase, S., Russo, D., Lungu, M., Martinez Montes, C., Prechelt, L., Ralph, P., van Tonder, R., & Wagner, S. (2026). Guidelines for Empirical Studies in Software Engineering involving Large Language Models. arXiv:2508.15503. Accepted at Empirical Software Engineering. <https://arxiv.org/abs/2508.15503>
- Feathers, M. (2004). Working Effectively with Legacy Code. Prentice Hall.
- Krakovna, V., Uesato, J., Mikulik, V., Rahtz, M., Everitt, T., Kumar, R., Kenton, Z., Leike, J., & Legg, S. (2020). Specification gaming: the flip side of AI ingenuity. DeepMind. <https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity/>
- Pact. (n.d.). Pact documentation. <https://docs.pact.io/>
- Pactflow. (n.d.). What is Consumer-Driven Contract Testing? <https://pactflow.io/what-is-consumer-driven-contract-testing/>
- Speakeasy. (n.d.). Contract testing with OpenAPI. <https://www.speakeasy.com/blog/contract-testing-with-openapi/>
- Yu, B., Zhu, Y., He, P., & Kang, D. (2025). UTBoost: Rigorous Evaluation of Coding Agents on SWE-Bench. arXiv:2506.09289. <https://arxiv.org/abs/2506.09289>

Appendix A — Kickoff prompt template

Verbatim, from `src/spec/generateKickoffPrompt.ts` (KICKOFF_PROMPT)
 — written once to every generated workspace as `kickoff-prompt.txt` by `writeSpecTree.ts`.

This workspace has a locked rebuild spec. Before writing any code:

1. Read `CLAUDE.md` and everything in `.claude/rules/` - these are non-negotiable, not suggestions.
2. Read `spec/` in full. Every file there represents a decision that has already been made. Do not re-litigate any of it. If something in `spec/` seems wrong or contradictory, STOP and ask.
3. Read `spec/contracts/*.md`. Match these interface shapes exactly. A correct implementation with the wrong shape still fails verification.

Then work in strict red-green-refactor cycles, not batch regeneration:

4. Pick ONE currently-failing test. Make the smallest possible change that could make it pass.
5. Immediately re-run the FULL tests/visible/ suite. If anything previously green is now red, revert and try a smaller fix.
6. Only once the full visible suite is green, move to the next test.
7. Never branch on a literal value that looks like a test fixture.

Do not touch tests/held-out/ until every visible test passes. Run it once, at the end, as a final report.

If stuck on any test, say so explicitly rather than forcing a change through. Report final pass/fail counts and anything you couldn't satisfy without changing the spec.

This is the tool's production prompt, used for every handoff outside the ablation study. The ablation trials (Sections 4.5, 4.10–4.12) ran a separate, experiment-specific kickoff prompt layered on top of it — the source of the structured `BATCH_BUILD_INCIDENTS`: N self-report field Section 4.11 refers to — not reproduced verbatim here since it is scaffolding for that experiment, not part of the tool itself.

Appendix B — Four-outcome taxonomy

Stated in full in Section 4.1, verbatim from `docs/v0-findings.md`'s run-criteria source, since it is load-bearing for every result in Section 4 and belongs where a reader first needs it rather than at the back of the paper. Not repeated here to avoid the two copies drifting apart.

Appendix C — Additional findings: security, test-integrity, and perceptual fidelity

These five subsections report real, independently-verified findings that do not bear on this paper's main throughline (contract-locking, mechanically-enforced discipline, and the single-prompt comparison in Sections 4.5–4.12) and are collected here rather than interrupting it. Section numbering (C.1–C.5) is retained from the order these results were originally obtained in.

C.1 Security evaluation

Threat model. In scope: a network-adjacent attacker who can reach the optional HTTP transport (Section 3.7's alternative to the default stdio MCP transport) and holds either no token, an incorrect token, or a valid one — i.e., whether the transport's own authorization and path/SSRF guards hold under direct adversarial HTTP traffic, not whether the target application being ingested is itself secure. Out of scope: supply-chain risk in this project's own dependencies,

denial-of-service against the MCP server or the spawned dev-server processes, and the default stdio transport (no network listener, so this class of remote attack does not apply to it at all). This narrows what the three findings below do and do not say: they are about **rebuild-dossier**'s own transport-layer guards, not about any property of an application it ingests or rebuilds.

The optional HTTP transport was adversarially tested against a live running server rather than trusted because the code looked correct, surfacing three real bypasses, not equally serious:

1. Structural. The path-authorization check performed textual containment only (`path.resolve/path.relative`), never resolving the filesystem's actual reality. A junction planted inside an allowed directory permitted reading and writing files entirely outside the sandboxed root — confirmed live, over real HTTP, with real auth. The check operated at the wrong layer entirely (string comparison rather than real-path resolution), not a missed case within an otherwise-sound mechanism. Fixed by resolving the deepest-existing ancestor's real path before the containment check.
2. Narrow, encoding-shaped. The SSRF guard's IPv6 branch never recognized IPv4-mapped addresses (`::ffff:127.0.0.1`), which passed through unblocked. The underlying private-range logic was sound; one address representation was simply absent.
3. Narrow, enumeration-shaped. `100.64.0.0/10` (Shared Address Space, RFC 6598, used for carrier-grade NAT) was missing from the blocked-range list entirely — corrected here rather than left as originally stated, which claimed this range overlaps the standard cloud-metadata endpoint (`169.254.169.254`, a distinct link-local range, RFC 3927); it does not, and the two should not be conflated. The real risk this range poses is that it is genuine, routable internal address space some environments assign, so an SSRF guard that does not block it can still be pointed at real internal infrastructure sitting inside it, independent of the metadata endpoint specifically.

Authentication itself (absent, incorrect, and correct tokens, across every tool call) checked out clean against a live server. Reported as a methods point: security claims were tested by attempting to break them, not assumed. The structural finding is the one worth weighing most heavily — a textual check that never touches the real filesystem is a category of bug that can recur anywhere else path input is trusted, not a one-off miss.

C.2 Test-integrity, in the wild

The mutation check demoted a real fraction of generated tests before shipping, and real classifier misclassifications surfaced on the live apps: a fixed grading-scale legend (`GRADE_VALUES = [10, 9.5, ..., 1]`) misread as dynamic; and, oppositely, comma-formatted live counts (`"2,007"`, `totalCards.toLocaleString()`) misread as static — the latter flipping a

page’s pass/fail non-deterministically across identical runs, recurring on a second page. The pipeline’s own **passesBaseline** net correctly demoted both. This is the tool’s core skepticism turned on itself, the direct analogue of the benchmark-integrity findings in Section 2.

C.3 Frontend page tests: mechanism works, yield is app-dependent

Against the real 83-route app, the honest headline is not “19/19 pages unblocked.” The mechanism is real and verified end-to-end, but on this auth-heavy app, black-box capture with no logged-in session cannot reach most pages’ authenticated content, so most tests carry little behavioral signal. Of 19 pages: 13 weak, 2 unrunnable, 3 with zero applicable mutation sites, and exactly 1 (**watchlist**) with a demonstrated, hand-traced, content-driven mutation kill. “1 verified of 19” matters more than **untested-contracts.json** reaching empty. Pages hit this at 15 of 19 weak/unrunnable vs. 32 of 64 (50%) for API routes in the same run — auth-gating amplifies an existing mode.

Route kind (catchand- trade)	Total	Mutation- verified	Weak	Unrunnable	Zero mutation sites
API routes	64	32 (20 visible + 12 held-out)	14	18	0
Page routes	19	1 (watchlist)	13	2	3

The **watchlist** kill, independently re-derived, not taken on trust. **src/app/watchlist/page.tsx:24** guards a client-only **localStorage** read behind an environment check; flipping that check makes the server-side branch run the client-only code during SSR, crashing the page — confirmed real by reading the line directly, not paraphrased from the findings document. This validates that a shared environment-detection pattern is SSR-safe, not that the page’s own distinctive behavior (which items appear, rarity-color mapping) is correctly reproduced — the mutated line is infrastructure the page happens to contain, not the feature it exists to provide. The same pattern appears nowhere else in **src/app** (checked directly), consistent with the 1-of-19 finding: this kill was only possible because this one page happens to be shaped this way.

A real crash bug (a **next dev** process-group leak racing mutation-check cleanup) was found only by running against a real app, fixed at the source.

One surfaced design tension, left explicitly open: weak/unrunnable page tests still unblock their page’s write-permission (15 of 19), matching pre-existing API-

route behavior but eroding the hook’s guarantee at a now-visible scale. Flagged as a design question to revisit, not filed as settled.

C.4 Reference-screenshot visual fidelity (exploratory; n=1 per condition)

Solid, behavioral, test-verified: on a blind fresh-agent build of a purpose-built app, the agent again avoided literal-value gaming — given captured values in the assertions, it built a shared module computing them from a fixed anchor year rather than hardcoding fixtures. This is the third independent confirmation of the build-the-general-case rail (Madeline and catchandtrade the first two).

Correction carried from the findings doc, not smoothed over: an earlier claim of a “fourth and fifth” confirmation was retracted on direct checking — it resolved to one verified confirmation, one now-unverifiable self-reported claim, and one verified partial counterexample. This draft states three, not five. The prior version of this paper reproduced the retracted claim; it is removed here.

Exploratory, method-limited: reference screenshots reliably conveyed color and typography (absent from text-only contracts) but transferred spatial layout inconsistently. Across three runs and two prompt conditions, two clean single-variable comparisons emerged: holding the app constant, an explicit screenshot-styling instruction measurably improved some layout properties; holding the prompt constant, a more distinctive app did not improve layout transfer on its own — the prompt drove the earlier difference more than the app did. One pattern (a masonry/staggered grid) failed regardless of prompt. A real in-the-wild classifier miss appeared (fixed menu prices read as dynamic-currency), alongside the cleanest single result: a rebuild reproducing exact original prices its own test would have accepted any valid value for. The color-vs-layout question remains open — n=1 per condition on hand-built apps. Method note: in one run the author’s own visual judgment initially transposed original and rebuild, showing visual-fidelity-by-eyeball is itself error-prone; a maintained axis would need a fixed protocol (identical prompt, script-labeled captures, pre-declared rubric) locked before running.

C.5 Motion: initially a total blind spot, since partially closed

Headline. Where Section 4.2 shows rails compensate for a weaker model’s judgment about what to build when a property is verifiable, motion (animation) demonstrates the opposite failure mode: a property invisible to any model because the verification mechanism had no representation for it at all, regardless of judgment. For CSS-declared motion this gap has since been closed and confirmed end to end on a fresh blind rebuild; for JavaScript-driven motion it remains fully open, with a real capture-window boundary now measured rather than assumed.

Setup and original finding. A purpose-built app, **driftlight**, has three real animations — a CSS entrance fade/slide, an infinite glow pulse, and a

`requestAnimationFrame` counter ($0 \rightarrow 12,400$) — rebuilt blind at the weak tier. The capture pipeline’s own ground truth was internally inconsistent (the generated test’s DOM-text capture froze the counter at "0"; the same run’s reference screenshot showed a third, different value, "104+"; a staggered card animation left half the product cards transparent in the screenshot despite the DOM-text assertion correctly naming all six) — a direct consequence of a static DOM-text assertion and a single settled-frame screenshot having no representation for motion at all. The rebuild’s two tests passed, but it hardcoded "104+" — the screenshot’s own capture artifact — as permanent static text (a second instance of the test-harness-artifact category, Section 5.1), and reproduced none of the three animations.

CSS motion: closed. The capture pipeline now neutralizes in-progress animation and adds a bounded settle wait before capturing, and contracts document declared CSS `@keyframes`/transitions directly (verified against fixtures built to stress two related bugs this fix surfaced along the way: a shared stylesheet misattributing animations across pages, and trigger conditions lost because detection queried selectors with their pseudo-class still attached — both fixed). A full fresh blind rebuild against the patched mechanism confirms the fix: contracts correctly detected and documented both CSS animations with their correct, unconditional triggers, and the blind rebuild reproduced both by name, wired to the same elements and triggers, in its own independently-written CSS.

JavaScript-driven motion: still open, boundary now measured. The same rebuild hardcoded 12,400 as permanent static text — the identical failure mode, different symptom. The pipeline’s real capture window is larger than its documented 1500ms settle-wait: an unrelated redirect-detection timeout (5000ms) runs before it on every capture regardless of whether the page redirects, combining into a real, incidental ~6500ms window on a non-redirecting page. Testing past that window (a counter run to 10 seconds) confirmed a genuine failure — the captured value (8,029) fell well short of the settled 12,400, consistent with the window having elapsed at capture, though this is a single run and the underlying test is itself flagged `unrunnable`, so only the qualitative result should be trusted, not the precise figure. The keyframe/transition mechanism was only ever built for CSS-declared motion; it has no JavaScript-driven equivalent, and this incidental ~6500ms window can mask that gap for any animation short enough to finish inside it — a real risk of concluding “motion is handled” from a test case that happened to fit a window nobody designed for that purpose.

Appendix D — Evaluation coverage matrix

Reference material only, gathered here so a reader can see the full (app $\times$ model tier/harness $\times$ condition) design at a glance rather than reconstructing it across twelve subsections. Cell contents summarize, not replace, the sections cited. † = author-built to exercise one property under test; ‡ = third-party, selected before any pipeline run; unmarked = the author’s own real repository (Section 5.3’s own evidentiary-weight ordering).

App	Model / harness	Condition	Trials	Result	§
Madeline	Weak/Strong, Claude Code	Spec+rails	1(+3)	Weak: rails violation. Strong: clean, 3/3. Replicated 7/7 vis., 1/1 held-out	4.2, 4.8
Madeline	Weak, Claude Code	Single-prompt	3	7/7, 1/1 — ties spec+rails	4.8
Madeline	Weak, Claude Code	Diagnostic probe	3	Right failure category, wrong mechanism, 3/3	4.4
catchandtrade	Strong, Claude Code	Spec+rails (liveness: open fork)	1	Clean, 20/20 vis., 0/12 held-out (scope gaps only)	4.3
catchandtrade	Weak, Claude Code	Single-prompt	3	Consistent restraint	4.9
catchandtrade	Weak, Claude Code	Spec+rails (hook dead)	3	Batch-built 2/3 — loses to single-prompt	4.9
catchandtrade	Free/weak, OpenCode	Single-prompt	3	Ref.-destruction 2/2; Stripe fabrication 2/3	4.10
catchandtrade	Free/weak, OpenCode	Spec+rails, hook live (+control)	3+3	Advantage tracks spec, not hook (2,1,1 vs. 1,2,1)	4.5, 4.11

App	Model / harness	Condition	Trials	Result	§
catchandtrade	Strong, OpenCode	Spec+rails, hook live	3	Held-out discipline 3/3; same 5 endpoints missed	4.12
notarybox†	Weak, Claude Code	Spec+rails, blind	1	Contract fidelity confirmed — direct test of Contribu- tion 1	4.6
NextTS- Todo- CRUD‡	Weak, Claude Code	Spec+rails, blind (hook never fired)	1	Batch- built full API vs. a static- string test	4.7
driftlight†	Weak, Claude Code	Spec+rails, blind	1+1	CSS motion: closed. JS motion: open	C.5
Visual- fidelity app†	Weak, Claude Code	Spec+rails, blind	3/2 cond.	Layout in- consistent; color/type reliable	C.4

Cells this design has not yet filled, both already named in Section 5.4: strong tier on either app under a single-prompt condition (never run at any app); and Claude Code specifically, at any tier, on catchandtrade with hook liveness confirmed rather than left an open fork or run through a different harness entirely. Every strong-tier, live-hook result in this table (catchandtrade row 3 above) uses OpenCode and a different model family, not Claude Code — the tool’s own mechanism, at the scale where the paper’s central comparisons live, remains untested end to end.