# REPAIR OR REBUILD? WHAT VALIDATORS MISS WHEN AGENTS REVISE EXECUTABLE WORLD MODELS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

When the rules of its world change, an agent can repair its executable world model or rebuild it, and a validator decides which revision its planner will trust. To test whether that trust is earned, we executed the confident plans of accepted programs in a pre-registered Crafter-OO audit. Acceptance did not protect them: they failed on new tasks about as often as rejected candidates' plans (19.0% against 23.3%, exploratory), and a pre-registered replication gives 17.3% (10.8–25.0%). Repair solved the most tasks, and *collateral regressions*, errors at rules the change left valid, came almost only from rebuilds; 136 of the 265 failing plans were shortcuts through a world that exists only in the model (post hoc). The pre-registered strictest gate, perfection on sealed held-out transitions, removed every failure and admitted 7 of the 30 accepted programs. We then turned to code, where no engine defines correct behaviour. For 11 of 62 resolved SWE-bench Verified patches, the issue or the unpatched code certifies a counterexample, against 1 of 13 coder-cleared controls. On one selected issue, all 8 top systems ship the same crash that every released test misses, and we prove that agreement among revisions cannot flag a defect most revisions share. A detector anchored on the unpatched code raised fewer false flags than a matched auditor on every backbone (pre-registered). Gate revisions on a regression check and a sealed check, never on agreement alone.

## 1 INTRODUCTION

Suppose the rules of a world change: a crafting table now costs three wood instead of two. An agent that plans with a learned executable world model, a program that predicts the next state from the current state and action, must revise that program. It can repair the program it has, keeping what still holds, or rebuild one from new evidence. In practice a validator decides. The candidate that best predicts recent transitions is kept, and the planner trusts it. We set out to learn which of the two an agent should prefer, and found that the answer depends on what the validator lets through.

In some of our four runs the validator rejected every repair attempt and kept the unchanged program as the "repair", with zero accepted edits. Most of its plans still worked, yet of 60 new tasks it was *confident* it could solve, meaning it predicted that its plan would reach the goal, 8 failed when executed, each at the first table placement, which the program still priced at two wood (Figure 1a).

We wanted to know whether this case was typical, so we executed every confident plan of every accepted and rejected program on new tasks. Acceptance did not protect the plans. Accepted programs solved more tasks, but their confident plans failed 19.0% of the time, as often as those of rejected candidates (23.3%, exploratory). A pre-registered replication with four new runs, two from a second source program, gives 17.3% (program-clustered 10.8–25.0%).

The two arms fail in different places. Repaired programs mostly miss the change itself, while rebuilt programs mostly err at rules the change never touched, whether they inherited the error or introduced it. We call these *collateral regressions* (Tayebati et al., 2026), which marks where the error lies and leaves its cause open (Section 4.3).

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

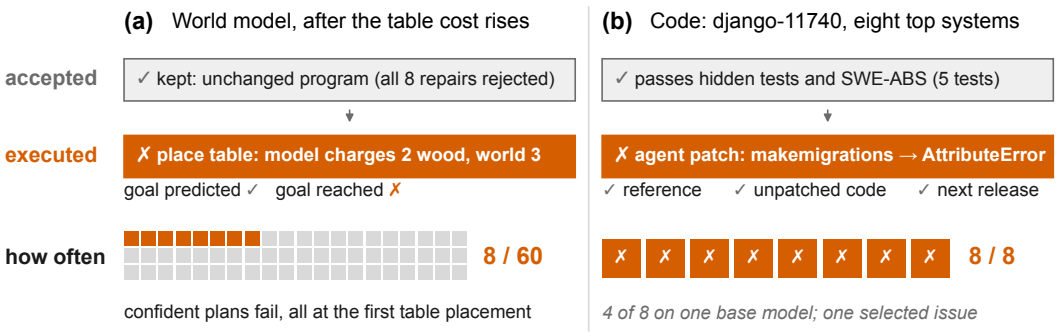



Figure 1: **The validator accepted each of these revisions; each fails when executed.** Grey ✓: passes the validator; vermillion ✗: fails when executed. (a) After the first change the validator rejected every repair and kept the unchanged program. (b) django-11740, one selected issue: 8 top systems pass the hidden and SWE-ABS tests (PROBE and UTBoost have none); one executed test crashes `makemigrations` on every patch and passes on the reference and unpatched code.

This matters because agents that write their world models as programs (Tang et al., 2024; Khan et al., 2026; Piriyakulkij et al., 2025) plan with the accepted one and can use its predictions in place of reward. Model-based RL makes the same bet (Chua et al., 2018; Janner et al., 2019; Lambert et al., 2020), and Vafa et al. (2024) show that a model can pass the usual diagnostics while its implicit world model is incoherent. We instead test the validator, executing plans of revisions it accepts and rejects, and so that every failure can be observed we study one LLM reviser on one controlled Crafter-OO slice (Hafner, 2022; Khan et al., 2026) (scope: Section 8).

We then asked whether the same failures appear where no engine can say what correct behaviour is. In code, our second testbed, we audit patches that already passed the validator, the hidden tests (SWE-Proof overturned about 31% of one model's (Ma et al., 2026)). The issue or the unpatched code certifies a counterexample for 11 of the 62 patches (17.7%) and for 1 of 13 coder-cleared controls, and as in world models the flags mostly mark regressions: 13 of the 18 flagged patches fail a test the unpatched code passes. Released augmenters (Yu et al., 2026; Li et al., 2026; Yu et al., 2025) leave 5 of 62 patches flagged and uncaught. On django-11740, one selected issue, all 8 top systems ship the same crash, which every released test misses (Figure 1b). We prove this *consensus blind spot*: agreement among revisions cannot flag a defect that most of them share.

We make three contributions; every pre-registered test, with its hash and verdict, is in Appendix I.

- **Validators keep revisions whose plans fail.** Executed confident plans of accepted world-model revisions fail about as often as rejected candidates' (exploratory), and that rebuilds fail mostly at rules the change left valid (Sections 4.2 and 4.3).
- **Repair solves the most tasks after a change, and a sealed gate removes every failure:** in one pipeline, the pre-registered gate requiring perfection on held-out transitions admitted 7 of the 30 accepted programs (Sections 4.1 and 4.4).
- **Agreement cannot see shared defects.** We prove it with definitions common to both testbeds. A detector anchored on the unpatched code raised fewer false flags than a matched auditor on all three backbones (pre-registered, Holm-adjusted), and its flags carry more information in these data (exploratory; Sections 2 and 6 and Appendix H).

## 2 Validator-surviving revisions and the consensus blind spot

A world model and a patched repository are one object: an executable artifact, revised after a change and kept by a validator that sees only part of reality (Table 1).

**Definition 1** (Validator-surviving revision)**.** After a change, a *repair* revises a prior executable artifact $\alpha_0$ using evidence, and a *rebuild* writes a revision $\alpha_1$ from that evidence alone. A *validator* accepts or rejects $\alpha_1$ with checks fixed before the revision is scored, possibly including the artifact's own self-report, such as a planner's prediction that its plan reaches the goal. $\alpha_1$ is a *validator-*

Table 1: **Two testbeds, one formal object (Definition 1).** World models alone let us execute reality.

| | World models (Crafter-OO) | Code (SWE-bench Verified) |
|---|---|---|
| **artifact** | program `predict` | repository after an agent's patch |
| **change** | hidden rule change: table, pickaxe cost | issue text |
| **reviser arms** | repair; rebuild (same archive; post-change) | leaderboard systems (one patch each) |
| **validator (gate)** | strict gain on 26 dev transitions, then planner self-report | hidden tests, then released augmented tests |
| **reality oracle** | native execution in the pinned engine | none; the reference is one realisation |
| **requirement / regression check** | change matters / does not matter at the divergent state | issue judged by two LLM raters / test passes on the unpatched code |
| **oracle error measured by** | replay reproduced 8 of 8 recorded plans | decoys rejected 32 of 32; rater $\kappa$ 0.926 vs 0.49; controls flagged 4 of 13 |

*surviving revision* if the validator accepts it, and one *with a defect* if it also behaves unacceptably at some query after the change.

**Two checks that need no reference solution.** Suppose a revision misbehaves at a query: if the change demanded the expected behaviour there, the failure is a *missing requirement* (the *requirement check*). If the pre-change reference already behaved as expected there and the change left that behaviour unchanged, the failure is a *collateral regression* (the *regression check*), the term Tayebati et al. (2026) use for self-modifications that fix one case and break another. We compare with the pre-change reference rather than the prior artifact.

In world models the engine is reality, so both checks run and *attribute* each failure that execution has already measured; we estimate how often confident plans of accepted programs fail, clustered by program. In code, where no engine says what the issue requires, the checks *establish* each failure: an executed counterexample, a test the patch fails and the reference passes, which we read as a defect once the issue or the unpatched code certifies it.

Why not let revisions check one another? Agreement is blind where they converge.

**Observation 1** (The consensus blind spot)**.** *Suppose a check sees only a pool of revisions of one task, treats them anonymously, and flags a revision only where it departs from behaviour the pool shares. Then the check never flags a defect the pool shares, and if all revisions behave alike it flags all of them or none.*

Ensemble disagreement over learned dynamics (Chua et al., 2018) can be such a check, yet in our world-model runs no failure signature recurs in more than 2 of the 9 to 11 accepted programs at a stage, while leaderboard patches often coincide byte for byte (Appendix E). Proofs and the soundness of both checks are in Appendices G and H.

## 3 Two audits that execute what the validator accepted

**World models.** An LLM writes a Python `predict(state, action)` model of a controlled slice of Crafter-OO (Hafner, 2022; Khan et al., 2026). The world then changes twice, unannounced: a table costs three wood instead of two, then the wooden pickaxe also needs one stone, while an unchanged control stage runs beside them. At each stage, *repair* edits the retained program, *rebuild (same archive)* writes a new one from the full time-labelled archive, and *rebuild (post-change)* writes one from post-change evidence only. Each arm makes 2 proposals and up to 6 repair rounds, stopping early only at a perfect development score. The gate keeps the program with the most exact predictions on 26 development transitions (the *incumbent*) and replaces it only on a strict gain; the primary run also rejects repairs that break a transition the incumbent predicted (Algorithm 1). The validator is this gate plus a beam planner's self-report: a plan is *confident* when the kept model predicts it succeeds. The four runs differ in evidence format, endpoint and acceptance rule, giving 36 cells (run, stage, arm), each scored on 4 tasks (Appendix A).

To test whether acceptance protects the plans, we audit every *candidate*: every program–stage pair that was ever a cell's incumbent; there are 72 of them. The 30 still kept when a cell ends are *accepted*,

Table 2: **Repair solved the most tasks after a change.** Descriptive (one source program, 4 tasks per cell). Changes: of 32 tasks; all: of 48. Fail: accepted programs' confident plans that fail, change stages; dev: of 26.

| | tasks solved | | | adaptation tokens | | audit | |
|---|---|---|---|---|---|---|---|
| arm | control | changes | all | changes | all | fail, changes | dev |
| repair | 16 | **28** | 44 | 1.34M | 1.47M | 6.7% | 20–26 |
| rebuild, post-change | 8 | 24 | 32 | 1.27M | 1.92M | 3.8% | 13–26 |
| rebuild, same archive | 7 | 8 | 15 | 2.05M | 2.85M | 44.5% | 9–26 |

and the rest are *rejected candidates*. Each runs its confident plans in the pinned engine (*native execution*) on its own stage's natively reachable tasks, giving 4,024 plans and 0 errors. For each failing plan, replay finds the first step where prediction and execution differ. We pre-registered the sealed gates as the strongest simple baseline, scoring the same programs on 27 held-out transitions no reviser saw.

**Code.** We execute counterexample tests instead of plans, and each auditor sees the issue and the patch, never the reference patch or the tests (Appendix B). A counterexample must fail on the patch, pass on the reference patch and reproduce. We also run the released augmented tests of SWE-ABS, PROBE and UTBoost (Yu et al., 2026; Li et al., 2026; Yu et al., 2025). Rates come from 150 random resolved patches from 8 top SWE-bench Verified systems (Jimenez et al., 2024; OpenAI, 2024), one per instance, analysed as the first 75 in draw order. Witnesses (counterexample tests) and witness-confirmed positives come from a 2024 sample of 150 instances whose patches were selected as plausibly non-divergent from the reference, and its follow-ups (Section 5.2). A cross-system re-run supplies the shared crash and no rate.

**What was fixed in advance.** Every study was hashed before its analysis (Nosek et al., 2018), and post hoc analyses are labelled (Appendix I). The world-model audit (hash `e385508c`) was frozen before its new tasks ran. The code headline count (two auditors, one attempt each) was designated after an interim look. World-model intervals are program-clustered bootstrap intervals that resample programs. Code has one patch per instance, so we use Wilson intervals, and Newcombe's hybrid interval for differences (Newcombe, 1998).

## 4 Repair or rebuild, and what the validator lets through

### 4.1 After a change, repair solved more tasks in three of four runs

**Repair solved the most tasks, largely by keeping what it already knew.** Across the two change stages, repair solved 28 of 32 tasks, against 24 and 8 for the post-change and same-archive rebuilds (Table 2). Retained knowledge carried much of the lead: before adaptation, the retained programs already solved 22 of the post-change tasks. Adaptation then added 6 more, at about 223k tokens per added task against 53k for the post-change rebuild (exploratory), and at every per-stage token budget repair had solved at least as many tasks as either rebuild.

The large gap is to the same-archive rebuild, while in the full-state run the post-change rebuild solved more on the change stages, at fewer tokens. All runs share one source program, so the comparison measures repair together with the competence it inherits.

**Transition accuracy tracks task competence only loosely.** Of the 46 accepted changes that raised development accuracy (24 more have no development score), tasks solved rose in 20, stayed level in 18 and fell in 8.

### 4.2 Accepted programs license confident plans that fail

Executing every confident plan in the pinned engine shows what acceptance buys, and it did not protect them: confident plans of accepted programs failed about as often as those of rejected candidates (19.0% and 23.3%; difference exploratory, Figure 3a).

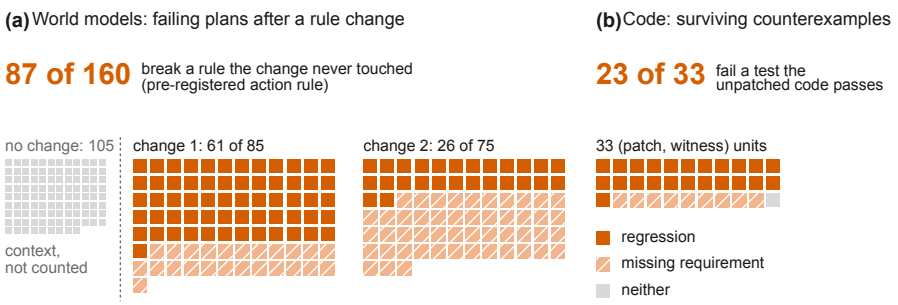



Figure 2: **After a rule change, most failures break what the change never touched.** (a) Under the pre-registered action rule, 87 of 160 failing confident plans after a change are regressions (grey: unchanged control, context only). (b) 23 of 33 surviving code counterexamples fail a test the unpatched code passes (selected units).

In this pipeline, 265 of the 1,392 confident plans of accepted programs failed on new tasks (program-clustered interval 9.5–29.8%), and 16 of the 30 accepted programs fail at least once. The result replicates from both source programs (four new runs, two from a second source program, analysed separately), at 14.6% (program-clustered 6.5–23.8%) from the second and 20.3% (10.4–33.3%) from the first, and 18 of 33 programs fail at least once (Appendix A).

Most failures come from same-archive rebuilds (203 of the 265), which the gate kept with 9–26 development transitions right, so per-arm rates are confounded with acceptance leniency. On the change stages, confident plans of same-archive rebuilds fail on 44.5%, against 6.7% for repairs and 3.8% for post-change rebuilds (exploratory).

**The gate tracks competence more than plan reliability.** Rejected candidates fail on 23.3% of their confident plans (accepted minus rejected $-4.3$ points, program-clustered 95% interval $[-17.2, +9.5]$; exploratory), while accepted programs solve more of their tasks (68.2% against 52.9%). Holding the cell or the development score fixed does not reverse this: accepted programs have the lower failure rate in 13 of 23 cells and the higher rate in 3 of the development bands where both occur (exploratory).

**Most failures lie on tasks the change did not affect.** Confident plans fail more often where the change matters (37.2% on tasks whose old-rule plan no longer works, against 16.5% elsewhere), but the second group is larger. The unchanged control alone holds 105 of the 265 failures, and there rebuilt programs fail at rules they never learned. Validator-admitted failures in this pipeline are therefore not specific to revising after a change (Appendix A).

**Search selects the errors, and they are overconfident.** Of the 265 failing plans, 136 are shorter than any plan that works: the planner found a shortcut that exists only in the model (Figure 8), yet of all 4,024 plans, just 4 were predicted to fail and then succeeded (post hoc, Appendix A.1).

### 4.3 Repairs miss the change; rebuilds err where nothing changed

To locate each failure, we ask whether the change touched its first divergent step. The pre-registered *action rule* calls the failure a missing requirement if the rule of the divergent action changed, whereas the *state rule*, a logged refinement, does so only if the old and new rules give different outcomes at the divergent state. Every other failure is a collateral regression: measured against the pre-change engine, the program errs at a step where the change makes no difference. The label locates the error and says nothing about its cause.

**Over the two change stages, most failures are collateral regressions.** Their share is 54.4% under the action rule and 60.0% under the state rule, although after change 2 alone missing requirements are the majority under either rule (Figure 2a).

The regressions are mundane. Under the state rule, 91 craft with no table nearby, a precondition that held before the change and still holds, and 96 mispredict movement or facing. The missing requirements are just as plain: of the 64, 35 predict exactly the old rule.

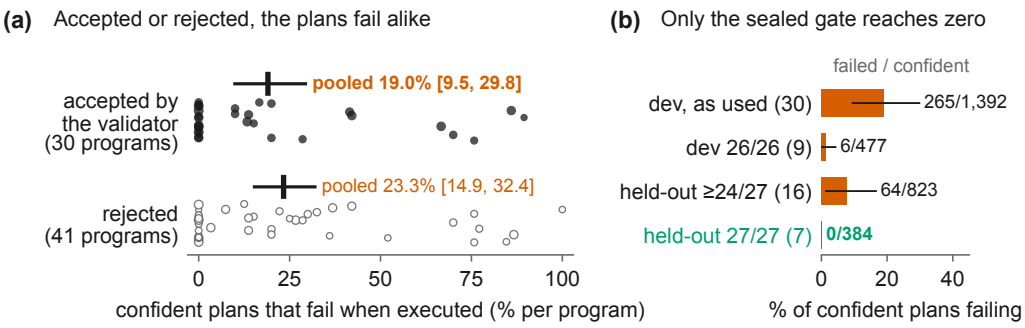



Figure 3: **Rejected candidates' confident plans fail about as often as accepted programs'. The sealed gate reaches zero and admits 7 of the 30 accepted programs.** (a) Each dot is one program, sized by its confident plans: 30 accepted and 41 rejected. Bars: pooled rate, program-clustered 95% interval. (b) Stricter gates on 26 development and 27 sealed held-out transitions, accepted programs only, with program-clustered 95% intervals.

**Regressions sit almost only in rebuilds, and repairs miss the change.** Under the action rule every collateral regression is in a rebuilt program, and under the state rule 197 of the 201 are. Repairs fail mostly on missing requirements (12 of 16). This holds partly by construction, since repair inherits rules the source program already had right, and it is confounded with acceptance leniency. Accepted rebuilds often lack knowledge that is still valid, whether they lost it or never learned it.

### 4.4 A SEALED PERFECT GATE REMOVED EVERY FAILURE

**The pre-registered sealed gate removed every failure.** We fixed it in advance as the strongest simple baseline, accepting only programs perfect on the 27 sealed held-out transitions, which no reviser optimised against. Programs passing it made 384 confident plans, 0 of which failed (Figure 3b). The gate is strict: it admits 7 of the 30 accepted programs, after change 1 a single repair at 26 of 26 development transitions, and no same-archive rebuild after either change.

The zero rests on 7 programs (program-level Wilson upper bound 35.4%), all of them also perfect on the development transitions. The development-perfect programs this gate removes include the only one that fails, so these data do not separate sealing from perfection. The replication reproduces the zero: programs perfect on the sealed transitions again made no failing confident plan, nor did development-perfect programs. Perfection is what counts: in the main audit, high but imperfect held-out scores did not suffice. Programs at or above 24 of 27 held-out transitions still fail on 7.8% of their confident plans (64 of 823, program-clustered interval 1.4–16.6%), and on 14.6% among those short of perfect.

**Thresholds on the reviser's own data cut failures but leave cells empty.** Post hoc, floors on development accuracy lower the failure rate among accepted programs (2.5% at a near-perfect floor, 1.3% at a perfect one), but applied to all candidates they leave 7 and 10 of the 12 cells after the first change with no admissible program. A development-perfect program still fails 6 of its 36 confident plans, because it never updates its facing after moving down (Table 17).

## 5 CODE, WHERE REALITY CANNOT BE EXECUTED

### 5.1 FLAGGED PATCHES MOSTLY BREAK BEHAVIOUR THE UNPATCHED CODE HAD

Code gives us no engine, so we built a certificate instead: a flagged patch counts only if the issue or the unpatched code backs its counterexample. We drew resolved patches of 8 top systems at random, outside our selected 2024 sample, and two auditors searched the 62 auditable ones among the first 75 for a failing test.

Most of what they found is a regression. Of the 18 patches they flag, 13 fail a test that the unpatched code and the reference both pass, 9 fail one that two LLM raters judged the issue requires, and 0 have

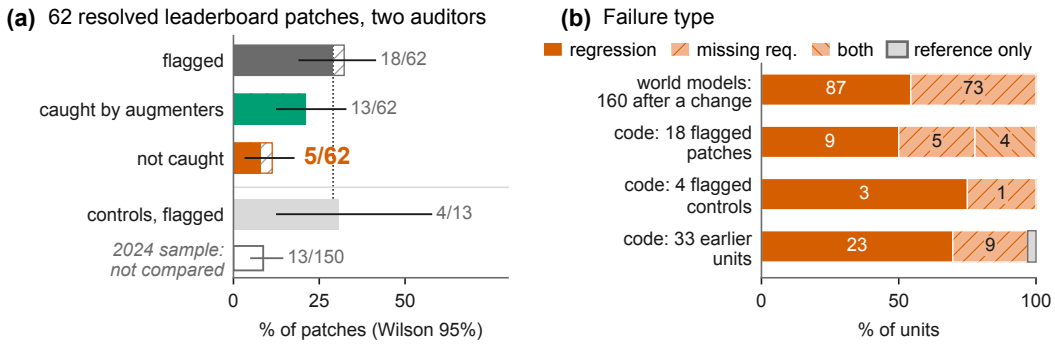



Figure 4: **Most counterexamples fail a test the unpatched code passes.** (a) Two auditors, one attempt each (hatched: six attempts), on 62 resolved patches and 13 coder-cleared controls, before certification; outline: the 2024 sample. (b) World models: change stages, pre-registered action rule. Code: flags and earlier (patch, witness) units; flagged controls carry the same content. The categories are analogous across testbeds.

neither. Flags cover 29.0% of patches (Wilson 19.2–41.3%; Django 7 of 34, others 11 of 28). On 13 Django controls that two LLM coders judged correct (*coder-cleared*; not known to be correct), they flag 4, as often (30.8%; difference $-1.7$ points, Newcombe 95% $-30.3$ to $+20.1$), with the same content (3 fail a test the unpatched code passes, 1 one the issue requires), so we certify flags before counting them. Specificity was measured only on Django.

The rate survives a change of auditors: a disjoint, pre-registered panel (GPT-6 Sol and three open models from other families) flags 29.2% of 130 built units (Wilson 22.1–37.6%), matching our pair. A concurrent route lands close: with the same counterexample rule, auditing against a formal specification lowers Claude Opus 4.8's resolution from 85.0% to 58.2%, or about 31% of its test-passing patches (Ma et al., 2026). Neither rate alone shows how many flags fail their issues; our certificate and controls address that, while SWE-Proof measures no specificity.

**Certification keeps most patch flags and few control flags.** 11 of the 18 flagged patches keep a certified counterexample resting on the issue or unpatched code (17.7% of 62, Wilson 10.2–29.0%; worst case in Appendix B). On the controls, certification fires on 1 of 13, down from 4 flagged (one-sided 95% upper bound 31.6%). The certificate is conservative: either rater can veto it, and a veto tightened after the input lock only lowers counts. The pre-registered comparison of certified rates cannot separate patches from controls, and the count is compatible with no true defects at the controls' specificity bound.

Released augmenters do not close the gap: SWE-ABS, PROBE and UTBoost catch 13 of the 18 flagged patches (72%), which leaves 5 of 62 patches flagged and uncaught (8.1%, Wilson 3.5–17.5%; 3 in Django), counted before certification. Six auditor attempts raise it to 7 of 62 (Figure 4a).

### 5.2 Certificates that do not rest on the reference alone

We asked how often a certificate needs the reference patch at all. Of the 17 surviving witnesses (counterexample tests) from our earlier samples, the unpatched code or the issue certifies 16, and all 17 pass on the next upstream release. On the agent patches they came from, they give 33 (patch, witness) units: 23 rest on the unpatched code alone, 9 on the issue alone, 0 on both and 1 on the reference alone (Figure 4b). The samples were selected, so the split gives no rate. The pre-registered reference-independence test stopped at its pre-set count (3 of 150) without passing. Independence matters because the reference itself fails: two reference commits in our samples were later fixed upstream as regressions. A certified counterexample is an executed test that fails on the patch, passes on the reference and reproduces over five re-runs. Its issue branch needs two blind raters from different model families (Panickssery et al., 2024), either of whom can veto it.

Table 3: **Fewer false flags on every backbone.** Detector (det) and matched auditor (aud) on 17 reference patches and 17 positives.

| | DeepSeek | | GPT-6 Sol | | Qwen | |
|---|---|---|---|---|---|---|
| | det | aud | det | aud | det | aud |
| false flags, reference | **4** | 10 | **2** | 15 | **1** | 9 |
| positives found | 6 | 12 | 7 | 15 | 6 | 10 |

### 5.3 THE CONSENSUS BLIND SPOT: TOP SYSTEMS SHARE A CRASH

On django-11740, all 8 top systems ship the same `makemigrations` crash (Figure 1b). The instance is selected; the sharing is not: 231 of 739 leaderboard units share a byte-identical patch with another system, in 64 groups, and 30 of these groups span model families (exploratory). Consistent with Observation 1, a frozen majority-vote detector flags 0 of 16 surviving units at 26 of 80 false flags (Appendix E).

## 6 CHECKS ANCHORED OUTSIDE THE GATE CUT FALSE FLAGS

Both checks that catch these revisions use evidence the gate never saw, which matters because a gate scored on the evidence the reviser optimises against cannot act as an independent check (Gao et al., 2023; Huang et al., 2024; Kamoi et al., 2024). In world models, the pre-registered sealed gate removed every failure (Section 4.4).

**In code, a regression check against the unpatched code cut false flags on every backbone.** Code has no sealed transitions but has the unpatched code, which the reviser never optimises against, so we built a detector from the two checks of Section 2. Its main rule, a regression check, counts a probe only if it is stable, is judged a regression from the unpatched code and lies outside the issue's scope; a patch-blind requirement check is secondary. Frozen, it met a matched agentic auditor once on held-out units and raised fewer false flags on reference patches on all three backbones (pre-registered; Holm-adjusted exact McNemar $p = 0.031$, 0.0007 and 0.016; Table 3). The detector was developed on units disjoint from the held-out ones, and the auditor was not tuned (Appendix C). On 20 coder-cleared controls the counts point in the same direction (3, 7 and 0 against 8, 16 and 3; not separately tested). The auditor finds; the certificate confirms (Section 5.1).

GPT-6 Sol shows why this matters: its auditor flags 15 of 17 reference patches, as often as it finds positives (15 of 17). Pooled over backbones, a detector flag is 2.4 times as likely on a positive as on a negative (95% CI [1.2, 5.1]), and an auditor flag 1.3 times ([1.0, 1.7]). In these data a detector flag carries more information, at \$4.65, \$2.63 and \$4.66 per detection (exploratory; difference not tested; Appendix C).

At its operating point the detector attributes fewer positives than the auditor (Table 3); the paired interval excludes zero on two backbones. The data do not rule out a recall loss of up to 0.59, 0.67 and 0.47. Held to the detector's false-flag rate, the auditor recovers 24% of positives against the detector's 37% (exploratory; on DeepSeek the auditor is ahead).

## 7 RELATED WORK

**Program world models.** WorldCoder, PoE-World and GIF-MCTS write world models as code and refine them online (Tang et al., 2024; Piriyakulkij et al., 2025; Dainese et al., 2024), and OneLife infers laws from one episode of Crafter-OO, our environment (Khan et al., 2026). Updates that fix new cases can break old ones (Kirkpatrick et al., 2017; Yoo & Harman, 2012), as our collateral regressions do. Concurrent work gates revisions on certified model-error bounds (Zhu et al., 2026); we evaluate the validator, executing the plans its verdicts license.

**Gates and agreement.** Self-modification gates reject changes that fix one case and break another (Tayebati et al., 2026); we count those an accuracy gate still accepts. Execution consensus picks programs and patches by agreement (Chen et al., 2023; Shi et al., 2022; Li et al., 2022; Xia et al., 2025). Voting helps only when errors are weakly correlated (Ladha, 1992); independently

written programs and language models fail together (Knight & Leveson, 1986; Kim et al., 2025; Goel et al., 2025).

**Validity of agent benchmarks.** Test adequacy, oracles, overfitting patches and benchmark shortcuts are well studied (Zhu et al., 1997; Barr et al., 2015; Liu et al., 2023; Qi et al., 2015; Smith et al., 2015; Xin & Reiss, 2017; Ye et al., 2021; Xiong et al., 2018; Zhu et al., 2025; Zhong et al., 2026). The closest audits check passing patches against the reference patch, directly in SWE-Bench+, PatchDiff, UTBoost, SWE-ABS and PROBE, and through a formal specification built from it in SWE-Proof, whose audit overturns about 31% of one model's passing patches (Aleithan et al., 2024; Wang et al., 2026; Yu et al., 2025; 2026; Li et al., 2026; Ma et al., 2026). Our auditors never see the reference patch, certificates rest on the issue or the unpatched code, and the same audit runs on coder-cleared controls (Appendix K).

## 8 Limitations, recommendations and conclusion

**Limitations and scope.** Three checks stand behind the central rates. The world-model audit replicates in four new runs, two from a second source program, where 17.3% of confident plans fail (program-clustered 10.8–25.0%). A disjoint four-auditor panel flags 29.2% of the 130 built code units, as often as our pair, and concurrent formal auditing overturns about 31% of the passing patches from one model (Ma et al., 2026). Now the edges. For world models we used one learner, one Crafter-OO slice and two changes we picked by hand. Our replay cannot tell an inherited collateral regression from an introduced one. In code, specificity comes from 13 coder-cleared Django controls, and the issue branch of the certificate rests on LLM raters who can veto; human rating of certificates is not part of this submission. A patch identical or nearly identical to the reference cannot be flagged, and we did not measure how often that happens. Unless stated, intervals cover instance sampling, not LLM run-to-run variation. Pre-registered tests that failed include prevalence (4 of 150, inconclusive, and only a lower bound, since 13 panel-flagged items were never executed), reference-independence (stopped on its count) and a majority-vote detector (Appendix I). Inside that scope, we executed every world-model failure.

**What to check before trusting an accepted revision.**

(1) *Add a regression check against pre-change behaviour.* Collateral regressions came almost only from rebuilds, and in code the check that cut false flags compares against the unpatched code (Section 6). In world models, distrusting any plan that uses an action the program mispredicted on its own development transitions leaves 0.9% of the remaining confident plans failing (post hoc).
(2) *Keep a sealed check that the reviser cannot optimise against.* The pre-registered sealed gate removed every world-model failure (Section 4.4).
(3) *Do not accept on agreement alone.* When most revisions share a defect, agreement among them provably cannot flag it (Observation 1).

**Conclusion.** In our pipeline, repair solved more tasks after a rule change in three of four runs, yet accepted programs' confident plans failed about as often as rejected candidates' (exploratory), mostly at rules the change left valid, and in code most flagged patches fail a test that the unpatched code passes. Gate revisions on a regression check and a sealed check, never on agreement alone.

### Reproducibility statement

Every study has a design file hashed before its analysis; Appendix I lists each with its hash, hash time and verdict, including the studies whose rules were not met. A manifest of the earlier design hashes carries an OpenTimestamps proof, which bounds when those hashes existed, not when each design was frozen. Definitions are in Section 2 and Appendix G, the world-model protocol in Appendix A and the code protocol in Appendix B. The anonymous supplementary material on OpenReview contains the world-model programs, task panels, plans and replays; the code unit lists with instance and system identifiers and patch hashes; every auditor, rater, panel and execution record; every witness; the prompts; the deviation logs; and the scripts that regenerate every figure, table and number of the paper from those records. Agent patches are referenced by identifier and hash, not redistributed. Witnesses and world-model programs re-run without any LLM. We propose MIT for

code and CC-BY-4.0 for records. The experiments used commercial LLM APIs and one server with two A100 GPUs for the self-hosted world-model runs. The benchmarks we build on are public: SWE-bench Verified and Crafter are released under the MIT licence.

### ETHICS STATEMENT

The code study analyses patches that their developers submitted to a public leaderboard, with public issues and public test releases; it uses no personal data. Untrusted code ran only in a network-isolated sandbox without credentials. We report results per public submission because the failures are reproducible and a benchmark can be repaired only if its failures are specific; we do not rank systems, since per-system counts are small and several systems share a base model. Our findings concern the benchmark's acceptance criterion, not the competence of any team. We will send the witnesses to the benchmark maintainers, the augmenter authors and the affected teams before the camera-ready deadline; no disclosure had been made at submission. Each released witness file and the README of the supplement carry a canary GUID so that they can be kept out of training data. The main risk we see is that per-submission counterexamples are read as a ranking of teams, which is why we report them without ranking and plan disclosure before release. Models from vendors of audited systems also served as our instruments; the supplement lists which model played which role. No human-subject data were collected for this submission.

### AI USE STATEMENT

Generative AI was central to this work, as a research instrument and as a research assistant. We answer each use that the ICLR 2027 policy requires authors to disclose, then the recommended ones.

**As research instruments.** An LLM writes and revises the world-model programs under study. In code, LLM agents served as static coders, adjudicator and witness author for the 2024 sample; as auditors (two models for the primary configuration, four more for the comparison); as blind issue raters and packet redactors; as writers of the rating cards for the blind certificate check and as a blind checker of their faithfulness; as a seven-family judge panel; as writers of the decoy expectations; as the oracle agents and judges of the detector; and as the coding agents of the frontier-model extension. Exact model identifiers, dates and prompt hashes are in the supplement.

**Required disclosures.** LLM agents (Anthropic Claude models, chiefly Claude Opus 5.5, as coding, analysis and writing agents; OpenAI, Google and open-weight models as instruments and external reviewers; exact identifiers per role are in the supplement) were used as follows. *Hypotheses:* yes; agents proposed and refined hypotheses, and the authors chose which to test. *Methodology and experiments:* yes; agents drafted experiment designs and pre-registrations and critiqued them, and the authors approved each design before it was hashed. *Implementing methods:* yes; agents wrote the world-model harness, sampling, sandbox, execution, statistics and figure scripts. *Conceptual framework, mathematical claims and proofs* (Section 2 and Appendix G): yes; agents drafted the definitions, propositions and proofs. *Synthetic data:* only as the instrument outputs listed above (witness tests, decoy expectations, rating cards); no model was trained on generated data. *Cleaning and reformatting data:* yes (result records and tables). *Qualitative analysis:* yes; agents categorised failures and drafted the case analyses. *Interpreting results:* yes; agents drafted interpretations, and every verdict is the one fixed by its pre-registered rule. *Translation:* not used.

**Recommended disclosures.** Agents also wrote and edited software, created the figures, searched and summarised the literature, formatted references, and drafted and edited this paper, including its structure and title.

**Checks on AI-assisted work.** Every pre-registration was hashed before the outcomes it governs were analysed. Separate agent runs recomputed reported outcomes from raw records and re-ran witnesses, and integrity audits checked model calls for errors, truncation and outages. Every number in the paper is generated by script from the released records, and a model with no prior context re-checked the numbers against those records before submission. Every reference was checked by script against arXiv, Crossref, OpenReview or the publisher's page. We reviewed all AI-assisted work and take full responsibility for the content of this paper.

## REFERENCES


Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs. arXiv preprint arXiv:2410.06992, 2024.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2207.10397.

Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. arXiv:1805.12114.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by Monte Carlo tree search. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. doi: 10.52202/079017-1933. arXiv:2405.15383.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10835–10866, 2023. arXiv:2210.10760.

Shashwat Goel, Joschka Strüber, Ilze Amanda Auzina, Karuna K. Chandra, Ponnurangam Kumaraguru, Douwe Kiela, Ameya Prabhu, Matthias Bethge, and Jonas Geiping. Great models think alike and this undermines AI oversight. In *International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 19621–19678, 2025. arXiv:2502.04313.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2109.06780.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.01798.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. arXiv:1906.08253.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. When can LLMs actually correct their own mistakes? A critical survey of self-correction of LLMs. *Transactions of the Association for Computational Linguistics*, 12:1417–1440, 2024. doi: 10.1162/tacl_a_00713.

Zaid Khan, Archiki Prasad, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2510.12088.

Elliot Kim, Avi Garg, Kenny Peng, and Nikhil Garg. Correlated errors in large language models. In *International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 30038–30066, 2025. arXiv:2506.07962.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017. doi: 10.1073/pnas.1611835114.

John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1): 96–109, 1986. doi: 10.1109/TSE.1986.6312924.

Krishna K. Ladha. The Condorcet jury theorem, free speech, and correlated votes. *American Journal of Political Science*, 36(3):617–634, 1992. doi: 10.2307/2111584.

Nathan Lambert, Brandon Amos, Omry Yadan, and Roberto Calandra. Objective mismatch in model-based reinforcement learning. In *Learning for Dynamics and Control (L4DC)*, volume 120 of *Proceedings of Machine Learning Research*, pp. 761–770, 2020. arXiv:2002.04523.

Chenglin Li, Yisen Xu, Zehao Wang, Shin Hwei Tan, and Tse-Hsun Chen. Probe to generate: Program variant-guided test augmentation for repository-level repair benchmarks. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2026. arXiv:2604.01518.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. doi: 10.52202/075280-0943. arXiv:2305.01210.

George Ma, Benjamin Mikek, Haoyu Li, Ferhat Erata, Yuhao Zhang, Zeren Shui, Behrooz Omidvar Tehrani, Jun Huan, Murali Krishna Ramanathan, Somayeh Sojoudi, Hao Zhou, and Anoop Deoras. SWE-Proof: Can language models resolve real-world issues with machine-checked proofs? arXiv preprint arXiv:2609.21190, 2026. We cite v2 (22 September 2026); v1 (18 September 2026) reports two models.

Robert G. Newcombe. Interval estimation for the difference between independent proportions: Comparison of eleven methods. *Statistics in Medicine*, 17(8):873–890, 1998. doi: 10.1002/(SICI)1097-0258(19980430)17:8<873::AID-SIM779>3.0.CO;2-I.

Brian A. Nosek, Charles R. Ebersole, Alexander C. DeHaven, and David T. Mellor. The preregistration revolution. *Proceedings of the National Academy of Sciences*, 115(11):2600–2606, 2018. doi: 10.1073/pnas.1708274114.

OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024.

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. doi: 10.52202/079017-2197. arXiv:2404.13076.

Wasu Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. PoE-World: Compositional world modeling with products of programmatic experts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. doi: 10.52202/085713-0896. arXiv:2505.10819.

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 24–36, 2015. doi: 10.1145/2771783.2771791.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022. doi: 10.18653/v1/2022.emnlp-main.231. arXiv:2204.11454.

Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 532–543, 2015. doi: 10.1145/2786805.2786825.

Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. doi: 10.52202/079017-2243. arXiv:2402.12275.

Sina Tayebati, Divake Kumar, Nastaran Darabi, Ranganath Krishnan, and Amit Ranjan Trivedi. Self-healing harness for runtime oversight of agent self-modification. arXiv preprint arXiv:2609.24130, 2026.

Keyon Vafa, Justin Y. Chen, Ashesh Rambachan, Jon Kleinberg, and Sendhil Mullainathan. Evaluating the world model implicit in a generative model. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. doi: 10.52202/079017-0846. arXiv:2406.03689.

You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? An empirical study. In *International Conference on Software Engineering (ICSE)*, 2026. doi: 10.1145/3744916.3764576. arXiv:2503.15223.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi: 10.1145/3715754. Published version of arXiv:2407.01489, "Agentless: Demystifying LLM-based Software Engineering Agents".

Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 226–236, 2017. doi: 10.1145/3092703.3092718.

Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *International Conference on Software Engineering (ICSE)*, pp. 789–799, 2018. doi: 10.1145/3180155.3180182.

He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2), 2021. doi: 10.1007/s10664-020-09920-w.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012. doi: 10.1002/stvr.430.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-bench. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. doi: 10.18653/v1/2025.acl-long.189. arXiv:2506.09289.

Boxi Yu, Yang Cao, Yuzhong Zhang, Liting Lin, Junjielong Xu, Zhiqing Zhong, Qinghua Xu, Guancheng Wang, Jialun Cao, Shing-Chi Cheung, Pinjia He, and Lionel Briand. SWE-ABS: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2603.00520.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2510.20270.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. doi: 10.1145/267580.267590.

Huatai Zhu, Qiang Chen, Ziqian Kou, Wenhao Li, Fei Wang, Yichao Cao, Xiu Su, and Yi Chen. Dual-frontier: When can an agent trust its world model? arXiv preprint arXiv:2609.26293, 2026.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, et al. Establishing best practices in building rigorous agentic benchmarks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. doi: 10.52202/085713-5547. arXiv:2507.02825.

## APPENDIX AT A GLANCE

Every number in the main text can be traced here and regenerated from the supplement: the full protocol and results, the checks of how much they depend on our choices, the theory, and what is needed to audit or rerun the work. Part I is the evidence: for world models, the revision pipeline, every cell and every breakdown of the failures the gate admitted; for code, how patches were sampled, audited and certified; and the detector's frozen method and its held-out comparison. Part II measures how the code counts depend on the auditor, the stopping point, the controls and the raters, how often failures are shared across systems and programs, and follows four failures from the change to the certificate. Part III states the objects of Section 2 in full and proves why agreement among revisions cannot flag a shared defect, and when the two checks that read the change or the unpatched code do imply one. Part IV lists every pre-registered study with its hash and verdict, including those whose rule was not met, the hash of every prompt, and the relation to prior work. Analyses marked † are exploratory or post hoc; everything else follows a design frozen before its data (Appendix I).

**PART I THE EVIDENCE BEHIND THE HEADLINE RESULTS**

**A World models: protocol and full results** ..... 15
How programs were revised, gated, planned with and audited (Algorithm 1); all cells of the comparison; the failure rate by arm, stage, failure type and gate; the pre-registered replication from a second source program; how accepted programs fail, mostly through shortcuts that exist only in the model (Appendix A.1, post hoc). *Backs:* Section 4, Figure 1a.

**B Code: sampling, audit and certification** ..... 21
How the two samples were drawn, what an auditor sees and how a counterexample becomes an executed, rerun certificate (Algorithm 2); the certification counts and the blind check of certificates. *Backs:* Section 5.1–5.2.

**C The detector in full** ..... 23
The frozen method (Algorithm 3), its development and selection, and the held-out comparison in which it raised fewer false flags than a matched agentic auditor on all three backbones. *Backs:* Section 6.

**PART II ROBUSTNESS, SHARING AND WORKED CASES**

**D Statistics and robustness** ..... 26
Code counts by sample and stratum; how the counts hold up across the auditor, the stopping point, the controls and the raters; the same audit on frontier-model patches; why no failed API call counts as a negative. *Backs:* Section 5.

**E Shared failures** ..... 28
Byte-identical patches and shared crashes across systems, shared failure signatures across programs, and what a majority vote does with shared behaviour. *Backs:* Section 5.3, Observation 1.

**F Four worked cases** ..... 31
One template per case: the change, the validator's verdict, the executed witness and the certificate, in both testbeds. *Backs:* Figure 1, Section 4.3.

**PART III THEORY**

**G Formal definitions and proofs** ..... 31
Executable artifacts, validators, witnesses and the two checks, instantiated for code and world models (Table 19); Observation 2 and Observation 1 with proofs. *Backs:* Section 2.

**H Why flags from the two checks carry information** ..... 34
Soundness of the requirement and regression checks (Observation 3), the error of an estimated certificate (Observation 4) and flag precision from the likelihood ratio (Observation 5). *Backs:* Section 6.

**PART IV REPRODUCIBILITY, INTEGRITY AND CONTEXT**

**I Pre-registration registry** ..... 35
Every study bearing on this paper, with its hash, freeze time and verdict under its frozen rule; tests not met, stopped or never run are reported alongside. *Backs:* every "pre-registered" in the paper.

**J Prompts** ..... 35
The file and hash of the fixed prompt of every LLM instrument; the files are released verbatim. *Backs:* Section 3.

**K Relation to prior work** ..... 36
What each related study checks a verdict against: augmenters judge patches against the reference patch, SWE-Proof audits one model against a reference-derived specification, world-model evaluations test a fixed model. *Backs:* Section 7.

# A WORLD MODELS: PROTOCOL AND FULL RESULTS

*What this establishes: how programs were revised, gated and audited (Algorithm 1), every cell of the comparison, the audit's breakdowns by arm, stage, failure type and gate, and a pre-registered replication in four new runs, two from a second source program, in which 17.3% (10.8–25.0%) of confident plans fail.*

**Pipeline.** The substrate is a controlled 16×16 slice of Crafter (Hafner, 2022) whose transitions come from the pinned functional Crafter-OO engine (Khan et al., 2026): nine actions, no creatures, horizons of at most eight steps. A model is a Python function `predict(state, action)` returning the full next observation. Change 1 raises the table cost from two to three wood; change 2 also makes the wooden pickaxe cost one stone. At each stage each arm proposes candidates; the gate keeps the one with the most exact predictions on the stage's 26 development transitions and replaces an incumbent only on a strict improvement, so ties keep the incumbent (Algorithm 1). A stage runs its fixed number of repair rounds and stops early only when the incumbent is exact on every development transition. In the primary run a repair is also rejected if it breaks a development transition the incumbent predicted exactly; the other runs require only the aggregate gain. The planner is a shared beam search (width 64, at most 4,608 model queries) that runs the program in a sandbox and stops at the first predicted goal state. Each (run, arm, stage) cell is evaluated natively on 4 tasks.

**The four runs.** The primary run and three variants share the source program, the development panels and the task panel. The three variants replace the primary run's acceptance rule, which admits no regression on the development panel, with an aggregate-best rule; the full-state and compact runs also change the evidence format shown to the model (full states versus compact deltas). Table 4 gives all 36 cells. The costs of fitting the source program are accounted separately and are not in any adaptation total.

Table 4: All 36 cells of the repair-or-rebuild comparison: final tasks solved (of 4), cumulative adaptation tokens, final development accuracy (of 26) and number of accepted program changes, per run, arm and stage.

| | | unchanged control | | | | change 1 | | | | change 2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| run | arm | tasks | tokens | dev | chg | tasks | tokens | dev | chg | tasks | tokens | dev | chg |
| primary | repair (warm) | 4 | 27k | 26 | 0 | 3 | 181k | 20 | 0 | 4 | 36k | 26 | 1 |
| primary | rebuild, post-change evidence | 1 | 164k | 13 | 1 | 1 | 142k | 24 | 2 | 4 | 82k | 26 | 2 |
| primary | rebuild, same archive | 1 | 202k | 9 | 1 | 1 | 178k | 16 | 2 | 0 | 235k | 11 | 1 |
| aggregate | repair (warm) | 4 | 33k | 26 | 0 | 3 | 172k | 20 | 0 | 3 | 232k | 20 | 1 |
| aggregate | rebuild, post-change evidence | 2 | 169k | 26 | 5 | 3 | 193k | 25 | 1 | 2 | 184k | 26 | 4 |
| aggregate | rebuild, same archive | 1 | 187k | 20 | 3 | 1 | 178k | 14 | 1 | 1 | 210k | 10 | 2 |
| full state | repair (warm) | 4 | 51k | 26 | 0 | 4 | 223k | 26 | 2 | 3 | 381k | 24 | 1 |
| full state | rebuild, post-change evidence | 4 | 157k | 26 | 4 | 4 | 248k | 24 | 5 | 4 | 44k | 26 | 2 |
| full state | rebuild, same archive | 4 | 206k | 26 | 2 | 1 | 349k | 19 | 4 | 3 | 452k | 20 | 4 |
| compact | repair (warm) | 4 | 28k | 26 | 0 | 4 | 75k | 26 | 1 | 4 | 36k | 26 | 1 |
| compact | rebuild, post-change evidence | 1 | 155k | 23 | 2 | 3 | 173k | 21 | 4 | 3 | 207k | 24 | 3 |
| compact | rebuild, same archive | 1 | 212k | 23 | 4 | 1 | 188k | 12 | 3 | 0 | 258k | 9 | 1 |

**The audit's design.** The audit was pre-registered (hash prefix `e385508c`) before the new panel was generated and before any new plan was run or failure replayed. It fixed the estimands, program clustering and the stopping rule: drop the world-model claim if the Wilson upper bound of the plan-level failure rate on the new panel fell below 5% or fewer than two accepted programs failed. The observed upper bound is 21.2%, and 16 accepted programs fail. It also fixed the strongest simple baseline, a gate requiring a perfect held-out score.

**The audit panel.** Part A (the original four tasks per stage) is retrospective. Part B, the new panel, is a full factorial of four task templates (build a tool and mine; restore consumed stone; build a stone pickaxe; place a table and craft) crossed with inventories, an optional tree and four rotations: 204 tasks, of which the 168 reachable by breadth-first search over native actions are kept; 20 are change-sensitive (the shortest source-rule plan fails under the current rules). All 72 distinct programs were planned and executed on their stage's tasks, 4,024 plans with 0 sandbox errors. Before the run, the copied planner and executor reproduced 8 of 8 original plans exactly.

**Algorithm 1** World-model revision, gate and audit (one run). $K$ initial proposals and at most $R$ repair rounds per (stage, arm); both are fixed in the run configuration and identical across the four runs.

**Input:** source program $\alpha_0$; stages control, change 1, change 2; for each stage $s$: 26 development transitions $D_s$, 27 sealed held-out transitions $H_s$ that no reviser sees, and audit tasks $T_s$

**for** each stage $s$ and arm $a \in$ {repair, rebuild (same archive), rebuild (post-change)} **do**
  $E \leftarrow$ the time-labelled archive (repair, same-archive rebuild) or post-change transitions only
  $P \leftarrow$ {the program retained by $a$'s lineage} if $a$ is repair, else $\emptyset$
  add to $P$ $K$ proposals written from $E$ (repair: shown the retained program)
  $\hat{\alpha} \leftarrow$ the member of $P$ with the most exact predictions on $D_s$; ties keep the earliest
  **for** $r = 1, \ldots, R$ **do**
    **if** $\hat{\alpha}$ is exact on all of $D_s$ **then break**
    **end if**
    $\alpha' \leftarrow$ an LLM revision of $\hat{\alpha}$, shown $E$ and $\hat{\alpha}$'s errors on $D_s$
    **if** $\alpha'$ is exact on strictly more of $D_s$ **and** meets the run's regression rule **then** $\hat{\alpha} \leftarrow \alpha'$ ▷ ties keep $\hat{\alpha}$
    **end if**
  **end for**
  accept $\hat{\alpha}$ for $(s, a)$; every program scored above is a *candidate*
**end for**
*Audit, frozen before the new panel existed; all candidates, accepted or not*
**for** each candidate $\alpha$ of stage $s$ and each natively reachable task $t \in T_s$ **do**
  $\pi \leftarrow$ beam search over $\alpha$ in the sandbox; $\pi$ is *confident* iff $\alpha$ predicts it reaches the goal
  execute $\pi$ in the pinned engine under the rules of $s$
  **if** $\pi$ is confident and fails **then**
    $q^* \leftarrow$ the first step at which $\alpha$'s prediction on the native state differs from the native next state
    missing requirement iff the source and current rules differ at $q^*$, else collateral regression (state rule); stale rule iff the prediction equals the source rule's outcome
  **end if**
**end for**
report failing / confident plans of accepted and of rejected programs, clustered by program; apply the pre-registered gates on $H_s$ and post hoc gates on $D_s$ to the same programs

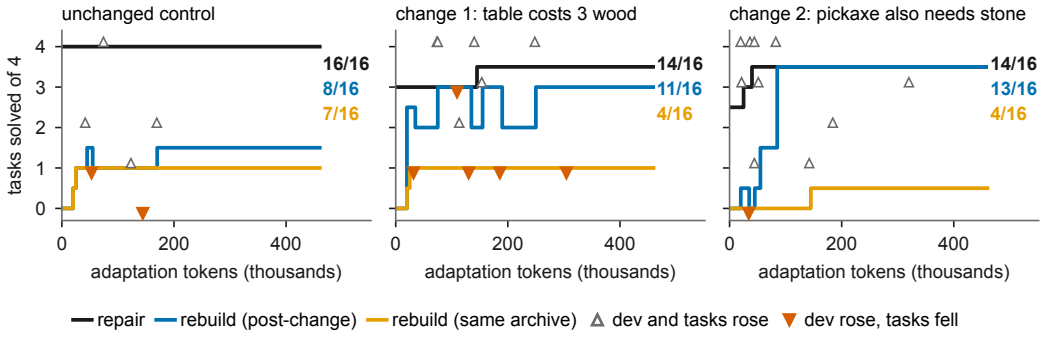



Figure 5: **Repair usually solved more after a change, at about the token cost of the post-change rebuild and a third less than the same-archive rebuild.** Tasks solved (of 4) against cumulative adaptation tokens, per stage; thin lines are the four runs, thick lines the median per arm, and the numbers at the right are stage totals of 16. Token totals also reflect the early stop at a perfect development score (Algorithm 1). Open triangles mark accepted program changes where development accuracy and tasks rose (20); vermillion triangles, where development accuracy rose and tasks fell (8); the 18 with no change in tasks and the 24 changes without a development score are not marked.

**Taxonomy rules.** The replay finds the first step at which the program's prediction on the native state differs from the native next state. The pre-registered *action rule* calls a failure a missing requirement when the divergent action is one whose rule changed. The *state rule*, a logged refinement, runs the

Table 5: Failure anatomy: the 265 failing confident plans of accepted programs, by the first divergent step.

| class (state rule; action rule in the last row) | plans |
|---|---|
| collateral regression | 201 (75.8%) |
| craft with no table nearby | 91 |
| movement or facing mispredicted | 96 |
| mine stone without a pickaxe | 10 |
| craft with too little wood | 4 |
| missing requirement | 64 |
| table placement, wrong wood deduction | 48 |
| wooden pickaxe at the old cost after change 2 | 16 |
| of these, stale rule (predicts the old rule exactly) | 35 |
| action rule: collateral regression / missing requirement | 192 / 73 |

Table 6: Confident plans of accepted programs that fail natively, by arm: all stages (program-clustered 95% interval) and the two change stages only†.

| arm | all stages | cluster interval | change stages |
|---|---|---|---|
| repair | 16 / 308 (5.2%) | 1.1–10.3% | 16 / 240 (6.7%) |
| rebuild (post-change) | 46 / 585 (7.9%) | 1.3–17.0% | 14 / 372 (3.8%) |
| rebuild (same archive) | 203 / 499 (40.7%) | 21.2–61.1% | 130 / 292 (44.5%) |
| all accepted | 265 / 1,392 (19.0%) | 9.5–29.8% | 160 of the 265 failures |

same action at the divergent state under the source rules and calls the failure a missing requirement iff the source and current rules give different outcomes there. Table 5 gives the counts under both rules and the subtypes.

**Deviations from the registration.**

D0 Part A deduplicated by (stage, program, task).
D1 State rule added; both rules reported.
D2 An inherited flag marked two programs as accepted at change 2, where they were only the warm starting point; corrected, which moved the headline from 285 of 1,472 to 265 of 1,392.
D3 Development-score thresholds added after outcomes were seen†.
D4 Program-cluster intervals for the threshold subsets.
D5 Precondition identity recorded for every precondition-omission divergence†.
D6 Descriptive evidence from other runs reported but not pooled.

**By arm (accepted programs, Part B).** Table 6 gives the rates. The pooled rates include the control stage, where repair is the unchanged source program; after change 1 the post-change rebuilds failed on 0 of 216 confident plans. The arm contrast is confounded with acceptance leniency (after a change, same-archive rebuilds were accepted at 9–20 of 26 development transitions) and is not a causal repair-versus-rebuild effect.

Table 7: The programs that pass the sealed gate (27 of 27 held-out transitions), out of all candidates on the audit panel.

| stage | arm | dev | accepted | sha256 prefix |
|---|---|---|---|---|
| unchanged control | rebuild, same archive | 26/26 | yes | `81177af4` |
| unchanged control | rebuild, post-change evidence | 26/26 | yes | `30e5294e` |
| unchanged control | repair (warm) | 26/26 | yes | `53b9ceb9` |
| change 1 | repair (warm) | 26/26 | yes | `11e94854` |
| change 2 | rebuild, post-change evidence | 26/26 | yes | `6aba1042` |
| change 2 | rebuild, post-change evidence | 26/26 | yes | `b5ef3b86` |
| change 2 | repair (warm) | 26/26 | yes | `2476083d` |

Table 8: **Accepted programs fail no more often after a change than before it.** Confident plans of accepted programs that fail natively, by arm and stage: failures / confident plans (%). Exploratory breakdown of the pre-registered rate.

| | control | change 1 | change 2 | all |
|---|---|---|---|---|
| repair (warm) | 0/68 (0.0) | 8/120 (6.7) | 8/120 (6.7) | 16/308 (5.2) |
| rebuild, post-change evidence | 32/213 (15.0) | 0/216 (0.0) | 14/156 (9.0) | 46/585 (7.9) |
| rebuild, same archive | 73/207 (35.3) | 77/177 (43.5) | 53/115 (46.1) | 203/499 (40.7) |
| all accepted | 105/488 (21.5) | 85/513 (16.6) | 75/391 (19.2) | **265/1392 (19.0)** |
| program-clustered 95% CI | [4.8, 43.6] | [2.7, 34.2] | [7.1, 36.2] | [9.5, 29.8] |
| change-sensitive tasks | – | 33/65 (50.8) | 31/107 (29.0) | 64/172 (37.2) |
| failing: regression / missing | 105 / 0 | 61 / 24 | 35 / 40 | 201 / 64 |

Table 9: **The sealed held-out gate removes every failure and keeps fewer programs.** Acceptance gates on the same programs. Change-1 cells: (run, arm) cells at change 1 whose accepted program passes the gate. †Exploratory: development thresholds were chosen after the audit's outcomes were seen.

| acceptance gate | reviser sees? | fail / confident | cluster 95% CI | programs | change-1 slots |
|---|---|---|---|---|---|
| as used (argmax dev) | yes | 265/1392 (19.0) | [9.5, 29.8] | 30 | 12/12 |
| dev $\geq$24/26† | yes | 18/713 (2.5) | [0.0, 5.9] | 14 | 5/12 |
| dev 26/26† | yes | 6/477 (1.3) | [0.0, 4.3] | 9 | 2/12 |
| held-out $\geq$24/27 | no | 64/823 (7.8) | [1.4, 16.6] | 16 | 5/12 |
| **held-out 27/27** | no | **0/384 (0.0)** | Wilson $\leq$1.0 | 7 | 2/12 |
| *rejected candidates, for contrast:* 381/1632 (23.3), cluster CI [14.9, 32.4], 41 programs | | | | | |

**Movement coverage.** The development panel of 26 transitions contains 3 movement transitions (two downward, one rightward) and none leftward or upward; 96 of the 265 failures diverge on a movement action, 74 of them on actions the panel contains. The audit panel has 68 control, 60 change-1 and 40 change-2 tasks.

**Retrospective panel.** Part A, the original four tasks per stage, gives the same rate as the new panel (17 of 85, 20.0%).

**Accepted against rejected, by development accuracy†.** In bands of development accuracy (0–15, 16–20, 21–23, 24–25 and 26 of 26), confident plans of accepted programs fail at 52.6, 28.8, 27.1, 5.1 and 1.3%, and those of rejected candidates at 32.4, 15.3, 24.2 and 11.9% (no rejected candidate reaches 26). The sign of the difference changes between bands, so the pooled contrast reflects which programs fall in which band; among accepted programs the rate falls with development accuracy.

**By change†.** Table 8 gives the failure rate of accepted programs by stage and arm. The rate is no higher after a change than before it (21.5%, 16.6% and 19.2%; intervals overlap). At change 1 the post-change rebuilds had no failure in their confident plans, and all of repair's change-1 failures come from the unchanged source.

**The sealed gate by stage.** The programs perfect on all 27 sealed held-out transitions are 3 at the control, 1 after change 1 and 3 after change 2 (Table 7).

**Cost per task gained†.** Per task gained over its starting point, repair spent about 223k tokens (6 tasks gained) and the post-change rebuild, which starts from no program, 53k. In the run that shows the reviser full states as evidence, the post-change rebuild solved more than repair on the change stages (8 of 8 against 7) at half the tokens (292k against 605k).

**A repair can be the old program.** The change-1 "repair" of Figure 1a is the unchanged source program, kept by the tie rule although the gate had seen the stale price (Table 16).

**Development thresholds on all candidates†.** Applied to all 72 candidates, the 24-of-26 development threshold gives 4.8% (program-clustered interval 0.4–12.6%) but leaves 7 of the 12 (run, arm) cells at the first change with no admissible program; at 26 of 26, 10 of 12 are empty.

**Further evidence, not pooled.** Two configured WorldCoder runs (Tang et al., 2024) on a structurally changed task each predicted success and failed natively; in both traces `make_wood_pickaxe` is

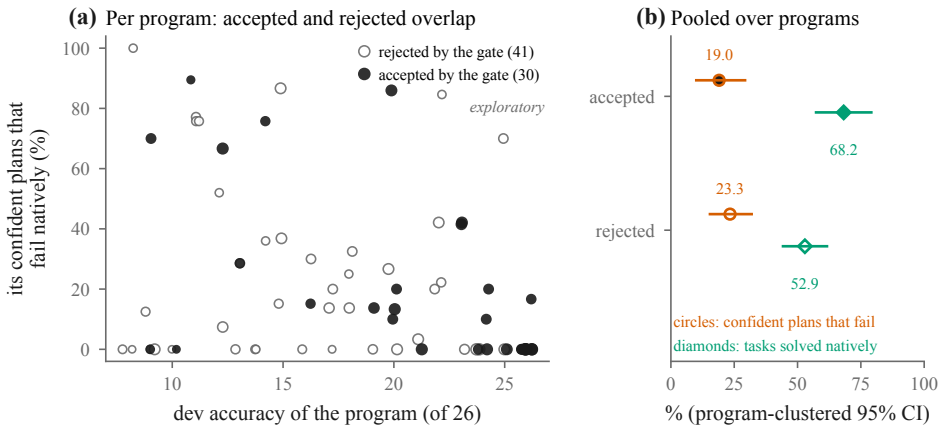



Figure 6: **Acceptance buys competence, not calibration (exploratory).** (a) Each point is one candidate program at one stage; filled: accepted by the pipeline's gate; area grows with the number of plans it predicted would succeed. (b) Pooled rates with program-clustered 95% intervals. Exploratory except the two failure rates.

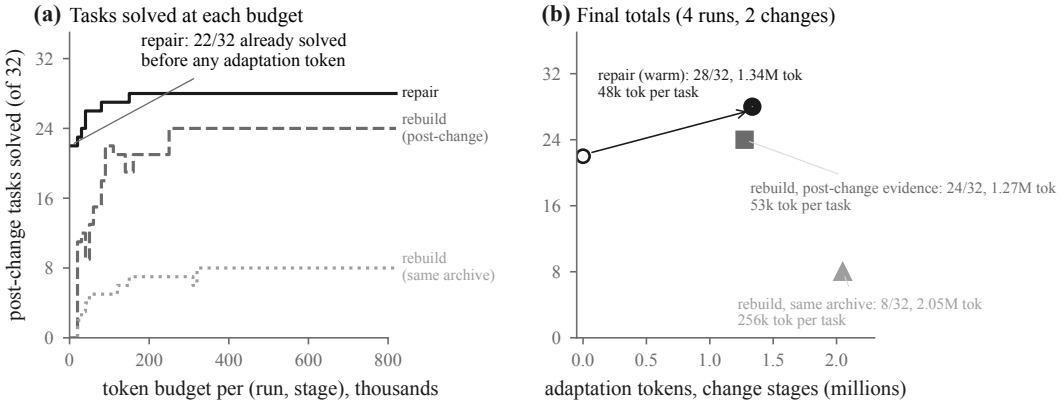



Figure 7: **Repair started from competence the source already had (exploratory).** (a) Post-change tasks solved (of 32, summed over 4 runs and 2 changes) by the incumbent after spending at most the given tokens in a stage. (b) Final totals. Repair's adaptation tokens bought 6 tasks (from 22 to 28), about 223k tokens per added task; the post-change rebuild spent 53k per task. The per-task figures in (b) average over all solved tasks, including those solved before any adaptation token. Token totals also reflect the early stop at a perfect development score (Algorithm 1).

issued with one wood and no stone and has no effect. The textual comparator (one run) solved 2, 1 and 0 of 4 tasks at the control and the two changes. It asks the model for one complete action sequence instead of running beam search over an executable model, so it differs from the program arms in controller as well as representation; its change-2 held-out accuracy is a parse failure.

**Replication with a second source program (pre-registered; never pooled with the main audit).** Four new revision runs of the primary configuration were frozen before any of them started: two from the source program that the source-selection tie-break did not choose (it scores the same as the original on the sealed panel, so it is a second learned program, not a different law) and two with new seeds from the original source, evaluated on the same frozen task panel with the same planner, executor and program clustering. Confident plans of accepted programs failed at 14.6% (114 of 780; program-clustered 6.5–23.8%) from the second source and at 20.3% (142 of 701; 10.4–33.3%) from the original source; over all new runs 17.3% (10.8–25.0%); per run 23.6, 3.5, 18.8 and 17.3%. 18 of 33 accepted programs fail at least once, and programs not already among the main audit's fail at 18.3%. 4 of 4 runs finished by the pre-registered cut. Under the pre-registered replication rule,

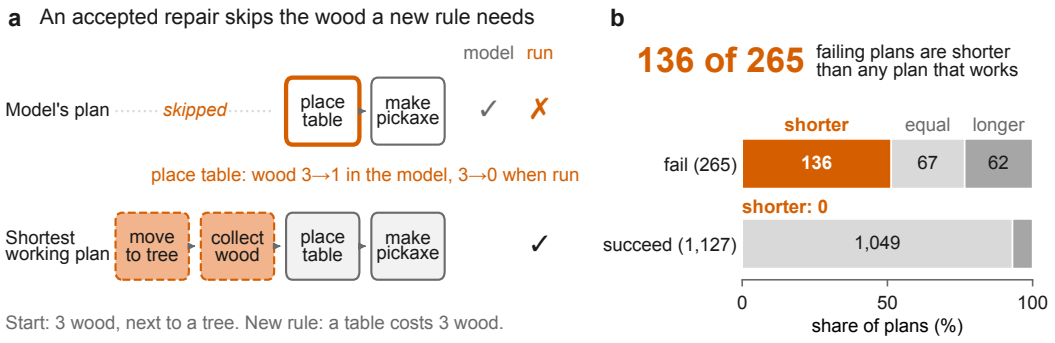



Figure 8: **Failing plans take shortcuts that no working plan takes** (post hoc). (a) An accepted repair skips the wood a new rule requires; the model predicts the table leaves wood that the engine does not. (b) 136 of the 265 failing confident plans, and none of the 1,127 successful ones, are shorter than the shortest plan that works.

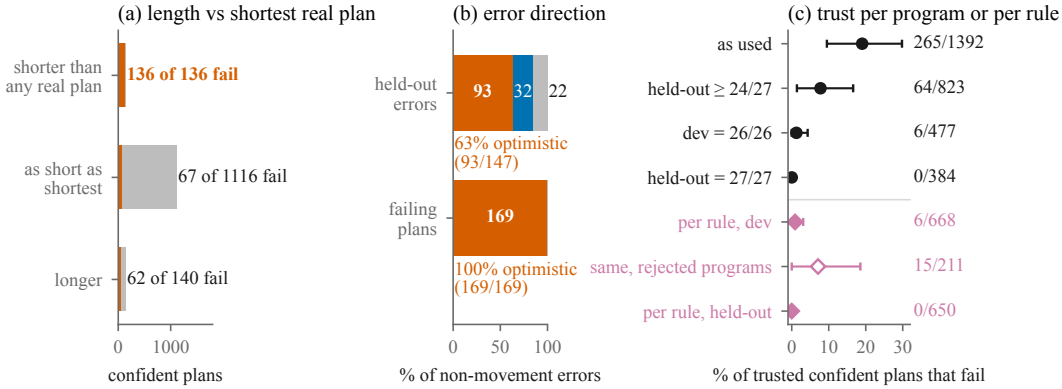



Figure 9: **Failing plans are shortcuts through optimistic errors, and a per-rule check removes most of them (exploratory).** Accepted programs, new panel. (a) Confident plans by length against the shortest plan that works natively (breadth-first search over native actions); vermillion: fails natively. (b) Direction of non-movement prediction errors: on the sealed held-out transitions of the same accepted programs, and at the first divergent step of failing confident plans. Vermillion, optimistic: the model lets an action take effect, or charges less, than the world does; blue, pessimistic: the reverse; grey: neither. (c) Share of still-trusted confident plans that fail, with program-clustered 95% intervals, when programs are gated as a whole (black) or when only plans that use an action the program mispredicted are distrusted (purple; hollow: the same dev rule applied to the rejected candidates, which were not used to devise it).

replicated: the pre-stated failure rule (cluster interval entirely below 5%) was not met. None of these is pooled with the 19.0% above. A fifth run from the second source, added by a pre-registered addendum before any outcome was seen and finished before its cut-off, is reported in its own row and never used for the replication rule: its confident plans failed at 5.4% (22 of 405); with it, the five new runs give 15.0% (program-clustered 9.1–21.6%, 39 programs) and the second source 11.4% (4.6–19.2%).

### A.1 How accepted programs fail: shortcuts, selected optimism, one bug per program†

*What this establishes (post hoc): most failing plans are shortcuts that exist only in the model, the planner selects optimistic prediction errors, each failing program has about one rule-level bug, and distrusting that rule instead of the whole program removes most failures.*

**51% of the failures are shortcuts through a world that does not exist.** The beam planner returns the first plan that reaches the goal in the model. Of the 265 failing confident plans, 136 (51%; program-clustered interval 29–77%) are strictly shorter than the shortest plan that works natively, so no execution could have succeeded; 11 of the 16 failing programs produce one (Figure 9a). No successful plan among all 4,024 is shorter than the native optimum. The shortcuts ride on permissive rules: crafting with no table nearby (64), the stale or wrong table price (48), the old wooden-pickaxe price (16). 16 of repair's 16 failures are shortcuts, against 120 of 203 for same-archive rebuilds and 0 of 46 for post-change rebuilds, whose failures are movement errors. The other 129 failures are mostly movement or facing errors (92).

**Search keeps the optimistic errors and discards the pessimistic ones.** At the first divergent step of all 169 non-movement failures (12 programs), the model is optimistic: it lets the action take effect where the world does not (105) or charges less wood or stone than the world does (64). None is pessimistic. The same 30 programs' non-movement errors on the sealed held-out transitions are 93 optimistic, 32 pessimistic and 22 neither (optimistic share of signed errors 74%, interval 59–87%); 8 programs make pessimistic errors. Pessimistic errors cost tasks, not safety: two same-archive rebuilds with 9 and 10 of 27 held-out transitions right, whose signed errors are 8 of 10 pessimistic, had no failure in 25 and 16 confident plans, while the unchanged source at change 1 (20 of 27, every error optimistic) failed 8 of 60. The one-sidedness of Section 4.2 is therefore a property of planning with the model, not only of the programs. Counting only optimistic held-out errors ranks programs by failure rate at Spearman $\rho = 0.74$ (0.50–0.88), against 0.59 (0.26–0.85) for all held-out errors; the paired difference is not resolved.

**Each failing program has about one bug, and the planner finds it many times.** For 13 of the 16 failing programs, every failure diverges in one rule family, and the most frequent family of each program accounts for 243 of the 265 failures. Each such bug was visible per rule: every failure diverges on an action the same program already mispredicts on at least one sealed held-out transition (191) or on a movement the held-out panel never contains (74: `move_down`, `move_right`). All 101 precondition failures (table nearby 91, pickaxe 10) come from programs that also fail the held-out transition probing that exact precondition. The aggregate score hid a bug the per-rule score showed.

**Distrusting rules instead of programs keeps more plans (post hoc).** Distrust a confident plan only if it uses an action the program mispredicted on its own 26 development transitions, data the reviser already has. Among accepted programs, 6 of the remaining 668 confident plans fail (0.9%; Wilson 0.4–1.9%; program-clustered 0.0–3.1%), against 265 of 1,392 as used, 6 of 477 for the program-level gate at 26 of 26 development transitions and 0 of 384 for the sealed perfect gate (Figure 9c). The rule keeps trusted plans from 22 programs, including 10 of the 12 cells after change 1, and at least one trusted plan on each of the 168 tasks. Using held-out transitions per action instead leaves 0 of 650. On the 42 rejected candidates, which we did not use to devise the rule, it leaves 15 of 211 (7.1%; 0.0–18.6%), above the 3% we predicted before computing it. All 21 failures that survive the development rule, accepted or rejected, are facing errors after `move_down` or `move_left`, movements the development panel exercises in 2 transitions (`move_down`) or none (`move_left`). Trusting only actions that the development panel exercised without error, a variant chosen after seeing these residuals, leaves none of 476 (accepted) and none of 192 (rejected) failing, but covers only 108 of the 168 tasks. Here each rule is one action, which favours a per-action check; in richer worlds the unit would be a rule, not an action.

**What does not predict failure.** The size of the edit does not: lines changed relative to the source program correlate with a program's failure rate at $\rho = 0.04$ ($-0.30$ to 0.38; 30 programs). Development accuracy does ($\rho = -0.58$, $-0.84$ to $-0.24$), about as well as held-out accuracy ($-0.59$).

## B CODE: SAMPLING, AUDIT AND CERTIFICATION

*What this establishes: every certificate rests on an executed counterexample stable over five re-runs; how the two code samples were drawn, what an auditor sees, and how a counterexample becomes certified (Algorithm 2); and the protocol for a blind check of the certificates (supplement).*

**The 2024 sample.** 150 instances whose late-2024 resolved patches were selected as plausibly non-divergent from the reference, each coded by two LLM coders with an adjudicator writing one ex-

**Algorithm 2** Code audit and certification of one resolved patch. The auditor never sees the reference patch or the hidden tests; the reference enters only when a test is run. One script applies the certification to every sample.

**Input:** issue $\iota$; unpatched tree $b$; resolved agent patch $p$; reference patch $g$; auditors $M_1, M_2$ from two providers
**Output:** flagged; caught by released tests; certified, with its branch

**for** $M \in \{M_1, M_2\}$, first attempt **do** ▷ witnesses
    $M$ reads $\iota$ and $b{+}p$ in a sandbox without network or credentials, runs code and tests, and writes an audit test file
**end for**
**for** each audit test $t$ **do** ▷ replication
    $t$ is a *counterexample* iff it fails on a fresh $b{+}p$, passes on a fresh $b{+}g$, and both reproduce once
**end for**
flagged $\leftarrow$ some counterexample exists
caught $\leftarrow$ some released augmented test fails on $b{+}p$, passes on $b{+}g$ and reproduces
**for** each counterexample $t$ **do** ▷ certificate branches
    issue $\leftarrow$ two raters of different families, shown $\iota$ and a redacted $t$ with its failure (never $p$ or $g$), both judge the failed expectation entailed by $\iota$
    unpatched code $\leftarrow$ $t$ passes on $b$
    **drop** $t$ if neither branch holds
    **drop** $t$ unless its outcomes on $b$, $b{+}p$ and $b{+}g$ agree over five re-runs ▷ stability
    **drop** $t$ if it also fails on a unit whose source hunks equal the reference, same repository and test file; outside Django, such a failure removes every certificate in that repository ▷ harness
    **drop** $t$ unless it fails on an assertion about a value or state, or on an exception raised in code under test; an interface or message failure counts only if its name appears verbatim in $\iota$ ▷ construct
    **drop** $t$ if only the unpatched-code branch holds and either rater judges the changed behaviour requested or permitted by $\iota$ ▷ veto
    mark $t$ *pending* if a rater call errored or a panel vote is missing ▷ post-lock; can only lower counts
**end for**
certified $\leftarrow$ some counterexample survives; branch: issue, unpatched code, or both; local resolution of $p$ is recorded, and units where it is unknown are counted both ways

ecuted witness (a same-prompt re-code flipped 3 of 50 of one coder's binary labels, and the coder flagged 1 of 150 reference patches shown in place of agent patches).

**The leaderboard sample.** Resolved SWE-bench Verified patches of 8 top public systems (submitted between August 2025 and February 2026), one per instance, drawn at random from the 299 Verified instances outside the 2024 sample; the analysis set is the first 75 in draw order, fixed from completion status alone.

**Exclusions.** The 3 unbuildable and 7 sanity-failing units come from scikit-learn, Sphinx, xarray and Matplotlib-heavy instances, so the audited set is weighted towards Django and SymPy. 3 units lack the second auditor because its provider credit ran out; they are reported as API errors and never as negatives. The cross-system slice re-runs witnesses on the patches of all eight systems for selected instances.

**Auditor.** Each auditor gets the issue text and the agent patch in a sandboxed copy of the patched tree, with no network and no credentials. It may read code, run Python and run tests, within 40 tool calls, and writes an audit test file. A test is a counterexample if it fails on a fresh patched tree and passes on a fresh reference tree, reproduced once. The two auditors are models from two different providers, with three independent attempts each; the primary uses the first attempt of each.

**Released augmenters.** A patch is caught if any released augmented test of SWE-ABS, PROBE or UTBoost fails on it, passes on the reference and reproduces. Augmented tests that fail on the reference are indeterminate and never catch.

**Certificates.** A counterexample is certified by the issue if two blind LLM raters from different families, shown the issue and a redacted form of the test and its failure but never the patch or the reference, both judge the failed expectation entailed by the issue; and by the unpatched code if the test passes on the unpatched tree. The pre-registered certification adds the checks of Algorithm 2.

**Certification counts.** Before certification the leaderboard counts are 18 counterexamples, 18 with an issue or base certificate, and 18 stable over five re-runs; no certified unit has an unknown local resolution. One rater judged the changed behaviour requested or permitted by the issue in 13 of the 14 base-certified units and the other in 2; with the pre-registered rule that either objection removes a base certificate, 11 units are certified, and without it 19 over all 75 drawn units. By stratum, 5 of 34 Django and 6 of 28 other patches are certified. The worst case over all 75 drawn units (8.4–24.4%) sets the pre-registered wording band: a certified rate above 5%; band words describe the certificate rate, not a defect rate.

**A rule change after the lock.** The certification inputs and code were locked before the final run. We then made the veto rule stricter: a rater call that errored, or a judge panel with a missing vote, now leaves a certificate pending instead of letting it stand, because an error is not a verdict. The change can only lower the count. It leaves 1 unit pending (one judge's call failed on every retry, although the other votes already fixed the panel's outcome); the rule as locked gives 12. Both code versions are in the supplement.

**Decoys and judges.** Decoy expectations were constructed to be the reported bug, its negation, or a plausible behaviour in the same component that the issue does not ask for. Both raters rejected 32 of 32; a seven-family judge panel, run as a robustness layer, changed no frozen verdict and rejected every decoy.

**Blind check of certificates: the instrument.** A first instrument asked raters whether the issue entails the failed expectation of a counterexample. It was abandoned before any rating: in pilots, neither a qualified Python developer nor an author could apply it reliably, even to worked examples, so judging entailment in the abstract is expert adjudication rather than a rating task. The replacement was pre-registered before any rating file existed and keeps the same 27 items: 12 certified when drawn (under the certification rule as locked), 7 uncertified, 4 coder-cleared controls as hard negatives and 4 twins. Each item shows the issue, one concrete input with a short code snippet, and two results for that input, the reference's and the audited patch's, unlabelled and on sides fixed by a seeded balanced shuffle; a twin is the same card with the sides swapped. Raters pick the result the code should produce (either one, both acceptable, or cannot tell).

**Blind check of certificates: the gate.** An item is endorsed when a strict majority of valid raters picks the reference's result. The gate, a veto evaluated only with at least three valid raters and eight certified items, requires endorsement of certified items (PPV) of at least 0.70 with a Wilson lower bound of at least 0.50, consistency between twins of at least 0.70 (a check on position bias), and PPV of at least 0.70 in every certificate branch with at least four items. An LLM wrote the cards from the public test source and the raw failure logs; a second model, checking them blind against the same sources, flagged wording in 17 of 23 cards, all fixed before the freeze. No rating was completed at submission; cards, app and analysis script are in the supplement.

## C THE DETECTOR IN FULL

*What this establishes: the detector's method (Algorithm 3), how it was developed and selected, and the held-out comparison behind Section 6. Frozen before that comparison and compared once with a matched agentic auditor on three backbones, it raised fewer false flags on reference patches on all three (pre-registered, Holm-adjusted); the summaries of its flags that follow are exploratory.*

**Method.** Algorithm 3 gives the frozen method. Oracle B works per instance and is patch-blind. Every variant with B lowered development Youden's $J$, so B is only in the secondary rule.

**Development and selection.** Development used 29 positives (witness-confirmed positives that survive the released tests) and 26 negatives (reference patches and coder-cleared controls, half each). Four early iterations were stopped by provider credit exhaustion or a rate limit and never scored. The selection rule changed once during development: first positive recall minus the reference false-flag rate, then, a few minutes later and before any score of the final variants was read, Youden's $J$ over

**Algorithm 3** The detector (frozen before its held-out comparison). It never sees the reference patch, hidden tests, witnesses or labels. Oracle B enters only the secondary rule.

**Input:** issue $\iota$; unpatched tree $b$; candidate patch $p$

1: Oracle A: an agent writes probes and runs each on $b$ and on $b{+}p$ in the sandbox, with feedback on which changed hunks each probe executed
2: keep a probe whose outputs differ on $b$ and $b{+}p$, if the difference is stable under a re-run with volatile tokens masked
3: a patch-blind judge labels each kept difference requested, regression or benign; a fresh patch-blind call checks that a regression lies outside the scope of $\iota$
4: **primary flag** iff at least two probes hit changed hunks, are stable, and are out-of-scope regressions at judge confidence of at least 0.7
5: Oracle B (secondary rule only): an agent writes acceptance checks from $\iota$ against $b$; they run on $b{+}p$, and a strict judge rules on each failure

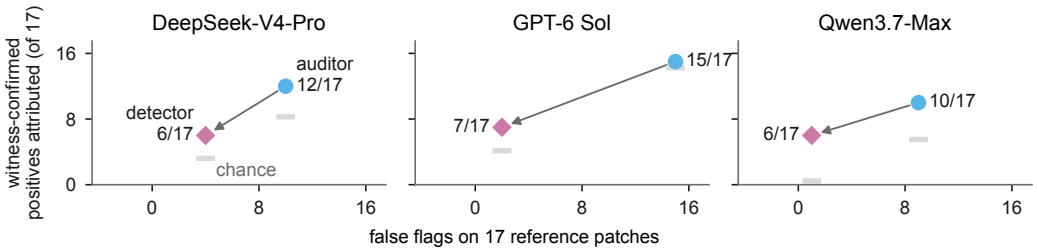



Figure 10: **Fewer false flags on reference patches on all three backbones; lower recall, established on two.** Held-out units, per backbone. Arrows run from the matched agentic auditor to the detector; bars mark the recall expected from each arm's false-flag rate alone.

all negatives; both rules and what had been read are in the supplement. The chosen rule reached development $J$ of 0.497 over 24 configurations, an optimistic figure. A volatile-timestamp bug in the stability filter was found through the django-11740 witness and fixed before the freeze.

**Held-out comparison.** 54 units per backbone: 17 positives (witness-confirmed controls of the leaderboard sample), 17 reference patches and 20 source-identical controls. The confirmatory test is an exact McNemar test on the reference patches, Holm-adjusted over three backbones. The matched auditor uses its deployable (reference-free) flag, first attempt; a third attempt was stopped at a pre-declared prefix and is not reported. The detector's disagreement with the auditor exceeds the auditor's own run-to-run disagreement on every backbone, overall and on reference patches. False flags on reference patches were identical across repeated attempts where these were run. The same holds for the leaderboard auditors: each of the 5 controls they flag is flagged on all three attempts, whereas the Gemini auditor detects 11 of the 13 witness-confirmed positives on at least one attempt but 7 of them on only one or two. Agreement across attempts would therefore filter true detections, not false flags, and neither instrument uses it.

**The auditor with the reference, and likelihood ratios.** The agentic auditor with the reference stopped at 248 of 432 planned episodes: no unit has all six, and eight episodes voided by an integrity audit were not re-run. Its first attempt with one model alone finds 29 of 29 witness-confirmed positives that survive the released tests and flags 4 of 13 controls, so its pre-registered verdict is partial and its counts are not estimates at full coverage. Without the reference the auditor flags 38 of 62 random patches, against 20 when its tests must also pass on the reference. On GPT-6 Sol the auditor finds 15 of 17 witness-confirmed positives and 15 of 17 reference patches.

**Likelihood ratios$^{\dagger}$.** Pooled over backbones, a detector flag is 2.4 times as likely on a witness-confirmed positive as on a negative ($LR^{+}$, 95% interval [1.2, 5.1]) and an auditor flag 1.3 times ([1.0, 1.7]); the intervals overlap, and we did not test the difference (exploratory). The recall each arm would reach by chance at its false-flag rate is lower for the detector on every backbone (Figure 10). The detector was tuned on one backbone's development units while the auditor's prompts were not; its positives are the witness-confirmed controls of the leaderboard sample, and a prior result file was read during development; one positive fires only under Python 3.8, while held-out runs used 3.9.

Table 10: Flag summaries on held-out units†. †Rule fixed on development units but written after the analyst had seen held-out aggregates. The rule keeps an auditor flag when the detector's main or secondary check also flags the unit. Its 26 attributed hits include one attributed only through the detector's evidence, so it keeps 25 of the auditor's 37; under the pre-registered conservative attribution (detector evidence only) its precision is 0.33, below the auditor's 0.35.

| | | flags | | | precision of a flag | | |
|---|---|---|---|---|---|---|---|
| backbone | method | attr. hits | ref. | C/C | held-out mix | at 17/150 | $LR^+$ |
| DeepSeek-V4-Pro | detector | 6/17 | 4/17 | 3/20 | 0.38 [0.15, 0.60] | 0.19 [0.08, 0.40] | 1.9 [0.7, 5.2] |
| | auditor @1 | 12/17 | 10/17 | 8/20 | 0.35 [0.22, 0.49] | 0.16 [0.10, 0.23] | 1.5 [0.9, 2.3] |
| | aud. ∧ sec. rule† | 8/17 | 6/17 | 2/20 | 0.42 [0.21, 0.65] | 0.22 [0.11, 0.41] | 2.2 [1.0, 5.5] |
| gpt-6-sol | detector | 7/17 | 2/17 | 7/20 | 0.37 [0.16, 0.59] | 0.18 [0.08, 0.33] | 1.7 [0.7, 3.9] |
| | auditor @1 | 15/17 | 15/17 | 16/20 | 0.31 [0.22, 0.39] | 0.12 [0.10, 0.14] | 1.1 [0.8, 1.3] |
| | aud. ∧ sec. rule† | 10/17 | 5/17 | 8/20 | 0.40 [0.22, 0.59] | 0.18 [0.10, 0.30] | 1.7 [0.8, 3.3] |
| Qwen3.7-Max | detector | 6/17 | 1/17 | 0/20 | 0.60 [0.30, 0.89] | 0.63 [0.33, 1.00] | 13.1 [3.9, ∞] |
| | auditor @1 | 10/17 | 9/17 | 3/20 | 0.42 [0.26, 0.57] | 0.19 [0.11, 0.30] | 1.8 [1.0, 3.4] |
| | aud. ∧ sec. rule† | 8/17 | 1/17 | 0/20 | 0.73 [0.46, 1.00] | 0.69 [0.40, 1.00] | 17.4 [5.3, ∞] |
| Pooled | detector | 19/51 | 7/51 | 10/60 | 0.42 [0.22, 0.62] | 0.24 [0.13, 0.39] | 2.4 [1.2, 5.1] |
| | auditor @1 | 37/51 | 34/51 | 27/60 | 0.35 [0.24, 0.44] | 0.14 [0.11, 0.18] | 1.3 [1.0, 1.7] |
| | aud. ∧ sec. rule† | 26/51 | 12/51 | 10/60 | 0.47 [0.30, 0.63] | 0.25 [0.16, 0.36] | 2.6 [1.5, 4.4] |

Table 11: Attributed recall at a matched false-flag rate†: the auditor's recall read off its own score at the detector's false-flag rate.

| | matched on reference false flags | | matched on all negatives (37 per backbone) | |
|---|---|---|---|---|
| backbone | det. vs aud. recall | Δ 95% CI | det. vs aud. recall | Δ 95% CI |
| DeepSeek-V4-Pro | 0.35 vs 0.41 | −0.06 [−0.35, +0.22] | 0.35 vs 0.44 | −0.08 [−0.34, +0.20] |
| gpt-6-sol | 0.41 vs 0.20 | +0.22 [−0.14, +0.48] | 0.41 vs 0.47 | −0.06 [−0.33, +0.25] |
| Qwen3.7-Max | 0.35 vs 0.18 | +0.18 [−0.14, +0.47] | 0.35 vs 0.18 | +0.18 [−0.11, +0.47] |
| Pooled | 0.37 vs 0.24 | +0.14 [−0.07, +0.36] | 0.37 vs 0.30 | +0.07 [−0.14, +0.25] |

**Summaries of the frozen held-out flags†.** Table 10 gives flag precision and positive likelihood ratios; Table 11 compares recall at matched false-flag rates, using the auditor's own graded score. At the detector's false-flag rate on reference patches, the auditor's score recovers 24% of defects against the detector's 37% (pooled difference +14 points, 95% interval [−7, +36]; +7 [−14, +25] when matched on all negatives; on DeepSeek the auditor is ahead). At the frame's rate of witness-confirmed counterexamples (17 of 150, not a defect rate), a pooled detector flag would point to a witness-confirmed counterexample about 24% of the time. Qwen's $LR^+$ of 13 rests on one false flag. None of these is a pre-registered test.

**Failure analysis.** Four positives are missed on all three backbones: a `makemigrations` change detected only in a project-level scenario the probes never built, a SymPy precomputed-value error, a missing dependency entry and a formset default; in the last three the flagged behaviour appears to lie inside the issue's scope, which the frozen scope gate excludes. One positive fires only under Python 3.8, while held-out runs used 3.9, so both arms flag it for another reason. Some reference patches that the detector flags may carry real defects: during development two reference patches and one control carried behaviour that Django later fixed as a regression.

**A simpler variant†.** Keeping only the auditor's failing tests that pass on the unpatched code cuts its false flags on reference patches from 34 to 15 of 51 and keeps 28 of its 37 attributed detections: the comparison with the pre-change artifact is what cuts false flags, and this auditor variant captures part of the detector's gain. The detector's development was less blind than the auditor's (above).

**Cost.** Each attributed detection of the detector cost $4.65, $2.63 and $4.66, 2.4 to 4.5 times the auditor's, with fewer flags to dismiss per detection (2.4 against 2.9).

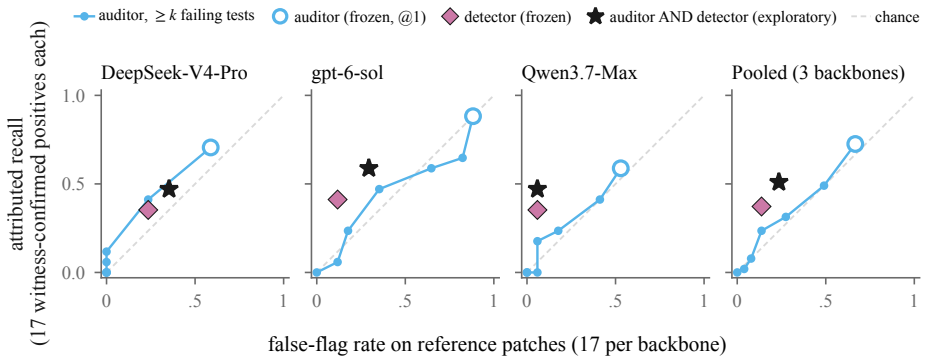



Figure 11: **Held-out flags in ROC space†.** Attributed recall against the false-flag rate on reference patches, per backbone and pooled; the auditor's curve uses its number of failing kept tests as a score.

Table 12: Code counts by sample. Counts before certification unless stated.

| | all | Django | other repositories |
|---|---|---|---|
| leaderboard sample, audited | 62 | 34 | 28 |
| flagged (two auditors, one attempt each) | 18 | 7 | 11 |
| flagged, not caught by released tests | 5 | 3 | 2 |
| certified counterexample (strict rule) | 11 | 5 | 6 |
| coder-cleared controls, flagged | 4 of 13 | 4 of 13 | – |
| coder-cleared controls, certified | 1 of 13 | 1 of 13 | – |
| difference, sampled minus controls (Newcombe) | −1.7 [−30.3 to +20.1] | −10.2 [−38.9 to +14.1] | – |
| 2024 sample, executed witness not caught (other instrument) | 13 of 150 | – | – |
| earlier surviving witnesses certified by issue or unpatched code | 16 of 17 | – | – |
| reference patch flagged by an agentic auditor (three backbones) | 10, 15 and 9 of 17 | – | – |

## D STATISTICS AND ROBUSTNESS

*What this establishes: the code counts by sample and stratum, how much they depend on the auditor, the stopping point, the controls and the raters, the rule that no failed API call counts as a negative, and the same audit on patches of frontier models (Appendix D.1).*

**Code counts by sample.** Table 12 collects the code counts. In the Django stratum, sampled patches are flagged at 7 of 34; non-Django patches at 11 of 28, where specificity was not measured. With six attempts the auditors flag 20. Exclusions (unbuildable, failing their own reference tests, an auditor outage; reported as errors, not negatives) are listed in Appendix B; none is Django or SymPy, so exclusions are not random. Certification counts are in Section 5.1 and Table 12: 5 of 34 Django and 6 of 28 other patches carry a certified counterexample, against 1 of 13 Django controls.

Table 13: Leaderboard sample by auditor configuration: flagged patches (Wilson 95%), flagged and caught by released augmented tests, flagged and not caught (before certification); recall on two sets of witness-confirmed positives from earlier samples (set A survives the released tests); flags on the coder-cleared Django controls.

| configuration | flagged | caught | not caught | positives A | positives B | controls |
|---|---|---|---|---|---|---|
| auditor G, 1 attempt | 9/62 (14.5%, 7.8–25.3) | 7 | 2 | 8/13 | 15/17 | 1/13 |
| auditor X, 1 attempt | 18/65 (27.7%, 18.3–39.6) | 13 | 5 | 13/13 | 17/17 | 4/13 |
| **both, 1 attempt each (primary)** | **18/62 (29.0%, 19.2–41.3)** | 13 | 5 | 13/13 | 17/17 | 4/13 |
| auditor G, 3 attempts | 11/62 (17.7%, 10.2–29.0) | 9 | 2 | 11/13 | 16/17 | 1/13 |
| auditor X, 3 attempts | 20/65 (30.8%, 20.9–42.8) | 13 | 7 | 13/13 | 17/17 | 4/13 |
| both, 3 attempts each | 20/62 (32.3%, 22.0–44.6) | 13 | 7 | 13/13 | 17/17 | 4/13 |

**Auditor dependence.** Table 13 gives every configuration. A second pre-registered comparison ran four further auditors, one attempt each, on the same draw-order prefix: they flag between 6 and 16 of 65 patches, and no auditor reaches full recall on the witness-confirmed positives with at most one control flag (the two with full recall flag 4 of 13 controls).

**Which model audits.** On the 62 units that all six auditors covered, which model audits matters more than how many do. The two GPT-6 models flag 15 (GPT-6 Sol) and 17 (GPT-6 Astra), against 6 to 9 for each of the other four; every patch that only one auditor flags was flagged by a GPT-6 model (5 by GPT-6 Astra, 4 by GPT-6 Sol); and they carry the control false flags, 4 of 13 each against 0 or 1 for each of the others. The other four flag fewer controls and 7 to 10 of the 13 witness-confirmed positives, against 13 of 13 for each GPT-6 model. All six together flag 22 of 62, against 17 for the best single auditor. API spend per flag, over all episodes each auditor ran, ranges from \$0.56 for GPT-6 Sol to \$3.21 for Gemini; GPT-6 Astra's spend was not logged per episode.

**The four further auditors on all built units.** The four further auditors also ran, as pre-registered, on every built unit of the 150 drawn: 130 units, the same for all four; the other 20 failed to build or failed their own reference tests, so only buildability selects units. On them GPT-6 Sol flags 34 of 130 (26.2%, Wilson 19.4–34.3%), DeepSeek 16 of 130 (12.3%, Wilson 7.7–19.1%), Qwen 11 of 130 (8.5%, Wilson 4.8–14.5%) and GLM 17 of 130 (13.1%, Wilson 8.3–19.9%); the four together flag 38 of 130 (29.2%, Wilson 22.1–37.6%; 20 of 130 not caught by released tests), and all six together flag 34 of 84 units covered by all six. This is a measurement without a decision rule. On the units beyond the prefix, processed before a pause and not random, the auditors flag 12 of 22 with six attempts, an enriched set that we do not pool.

Table 14: Stopping disclosure: primary-configuration rates on draw-order prefixes. Complete case uses audited units; worst case divides by all units drawn (unaudited counted as not flagged).

| prefix | audited | flagged (complete case) | flagged (worst case) | flagged, not caught |
|---|---|---|---|---|
| 40 | 36 | 10/36 (27.8%) | 10/40 (25.0%) | 3/36 (8.3%) |
| 50 | 44 | 12/44 (27.3%) | 12/50 (24.0%) | 3/44 (6.8%) |
| 58 | 50 | 12/50 (24.0%) | 12/58 (20.7%) | 3/50 (6.0%) |
| 65 | 55 | 15/55 (27.3%) | 15/65 (23.1%) | 4/55 (7.3%) |
| 75 | 62 | 18/62 (29.0%) | 18/75 (24.0%) | 5/62 (8.1%) |

**Stopping.** The sample size was fixed by cost; the primary configuration was designated with 44 of 75 units visible, which is optional stopping in the timing sense. Table 14 gives the rates on every pre-declared prefix; the final complete-case flag rate lies just above the range of the smaller prefixes, and the not-caught rate inside it.

**Controls.** All 13 controls are Django patches that two LLM static coders judged correct; they are not known to be correct. A blind upstream check found a later upstream fix of a regression in a function the patch modifies for 3 of 13 controls under a strict criterion and 5 of 13 under a broad one; in each case the regression test of that fix also fails on the reference patch. Specificity for other repositories was not measured.

**Raters.** On the 619 counterexamples of the leaderboard sample the two raters agree with $\kappa = 0.49$; on the 17 earlier witnesses, $\kappa = 0.926$. The issue branch of the certificate is therefore more reliable on witnesses than on auditor output at scale: one rater judged the expectation required for 30% of the witnesses but 56% of auditor counterexamples, and the decoy check was run on witnesses only, so it does not license the issue branch at scale. In the 2024 sample, 6 of 13 surviving witnesses depended on one of the two static coders, so a coder gate would have missed persisting classes.

**Latent-class model.** A pre-registered latent-class model over the static coders and the execution-based instruments estimates the share of resolved patches that fail the correctness rubric at 25–29% across dependence structures, with an identified set of 22–48% when dependence within method families is unrestricted; the estimate depends on the assumed dependence and does not enter the main text. The fit used an earlier snapshot of the auditor data (19 flagged of 63 units) than the final count (18 of 62); we did not re-fit it.

**What the flags contain.** Of the 18 flags, 13 fail a test that the unpatched code and the reference both pass and 9 have a counterexample both raters judged required by the issue; 0 have neither. Of the 4 control flags, 3 fail a test their unpatched code passes and 1 has an issue-required counterexample; content therefore describes what a flag is, not how often flags are correct. Of the 5 flagged patches that no released test catches, 4 fail a test the unpatched code passes. Post hoc, auditors and released augmenters agree on 13 flagged patches against about 5 expected if they were unrelated (Cohen's $\kappa = 0.61$); both require passing on the reference, so the agreement shows reproducible divergence only. With six attempts, 13 of the 20 flags are caught by released tests and 1 has neither property;

the 2024 sample's rate (13 of 150, 8.7%, Wilson 5.1–14.3%) comes from a different instrument and frame.

**API failures.** A unit whose model call failed, was truncated or returned no verdict is never scored as a negative; it is re-run or reported as an error. The rule matters because provider failures look like negatives. Of 18 leaderboard-auditor episodes that failed on provider errors and had been recorded as finished and unflagged, 10 were flagged when re-run (1 in the failed records); in the auditor-with-reference study, 2 of 5 cleanly re-run episodes flipped to flagged; and an interim certification run had counted 112 errored rater calls as no objection, which the final run replaced with real verdicts. None of these failed records enters a reported count.

### D.1 The same audit on frontier-model patches

On 16 tasks from a 65-task pool of instances whose build had finished, a four-auditor union flagged 9 of 81 resolved patches from seven frontier models (11.1%; Wilson 95% 6.0–19.8%; design-effect lower bound 1.3%); the same union flags 4 of 13 coder-cleared Django controls. All 9 flagged patches were also caught by released validators, and 8 of them fail a test the unpatched code passes. The analysis set ends at 16 tasks because one auditor episode on the next task did not finish before the pre-registered lock; the five pilot tasks are reported separately.

## E Shared failures†

*What this establishes: how often failures are shared across systems in code and across programs in world models, and what a majority vote does with shared behaviour (Observation 1). Selected instances, not prevalences.*

**Monoculture.** In the cross-system slice, django-11740 is shipped by 8 of 8 systems (Table 15). 231 of 739 leaderboard units share a byte-identical patch with another system, in 64 groups, 30 of which span model families.

**Instance-level catching.** In the selected cross-system re-run, whether released augmenters catch a patch is largely a property of the instance: in 23 of 26 instances with audited patches from several systems, all were caught or none. The leaderboard sample has one patch per instance, so its intervals are already at the instance level.

**A non-neutral reference.** In a five-task pilot, all 7 frontier models that submitted implemented the rounding that the scikit-learn-14496 issue asks for, where the reference truncates; 5 submitted byte-identical patches, and all 7 were resolved: a shared behaviour and a non-neutral reference, not a defect.

**Majority vote.** A frozen majority-vote detector, which flags patches whose probe outputs depart from the pool's majority, flags 0 of 16 surviving units at 26 of 80 false flags, as Observation 1 predicts when the surviving behaviour is shared. On django-16661 a post hoc probe finds the shared behaviour in 57 of 89 public patches.

**World models.** Post hoc, we grouped the 265 failing plans of accepted programs by stage, task, first divergent action and the predicted versus native change. They form 237 signatures; 28 are shared by two programs (56 plans) and none by more than 2 of the 9 to 11 accepted programs at a stage. At task level, on 3 change-1 tasks 5 of the 9 programs confident on the task fail, so failures can be common on a task without sharing a signature.

**Pairwise sharing study.** A pre-registered study of paired leaderboard samples (eight systems, 100 instances, 739 units) was not complete at submission: 97 units are still pending, so both of its frozen decision rules are incomplete. Its frozen analysis script, run as an exploratory look on the completed rows, finds an executed witness that no released test catches in 43 of 739 units (5.8%, instance-cluster interval 2.8–9.3%); pending units stay in the denominator and never count as such, so this is a floor on the rate over all units.

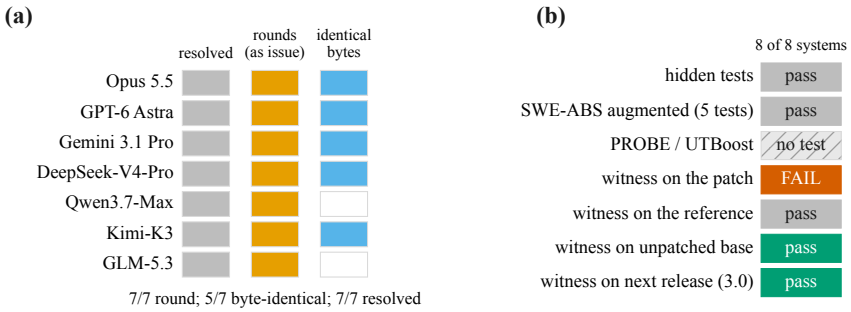



Figure 12: **Shared behaviour across systems.** (a) scikit-learn-14496 in the five-task frontier pilot: every model that submitted implements the rounding the issue asks for (the reference truncates), most with byte-identical patches, and all are resolved; a shared behaviour and a non-neutral reference, not a defect. (b) django-11740 in the cross-system slice: eight top systems, built on Claude, Doubao, Gemini and GPT models, pass every released check that exists and fail one executed witness that the reference, the unpatched code and the next release pass. Selected instances, not prevalences.




Figure 13: **One anatomy in two domains (summary).** (a) Failure and flag rates with their contrasts. (b) Most failures (world models) and flags (code) are regressions; 3 of the 4 control flags also fail a test their unpatched code passes. (c) Sharing: no world-model failure signature is shared by more than 2 of the 9 to 11 accepted programs at a stage†, while all 8 top systems ship one crash on django-11740 (Claude, Doubao, Gemini and GPT models). (d) The detector against the matched auditor.

# F FOUR WORKED CASES

*What this establishes: what a validator-surviving failure looks like end to end, from the change to the certificate, in each testbed.* Each case uses one template. The four are chosen to illustrate, not sampled: a crash shared by every top system, a "repair" that kept the old program, a collateral regression in a program perfect on its development data, and a certified regression from the random leaderboard sample. In the outcome row, *unpatched* and *source rules* are the pre-change reference $\rho_0$, and *reference* and *engine* are $\tau^*$ (Table 19).

Table 15: Case 1: one crash shipped by all 8 top systems (django-11740; a selected instance, not a rate).

| | |
|---|---|
| *change (issue)* | Changing a `UUIDField` to a `ForeignKey` should make `makemigrations` record a dependency on the target model. |
| *reference idea* | Compute the new dependency from the bound new field. |
| *agent change* | Compute it on the unbound clone of the field that the null-to-not-null branch builds, where the target model is still a string (read in the top-ranked patch). |
| *witness* | *Input*: make a nullable foreign key required (also: a nullable `UUIDField` changed to a required foreign key), then run `makemigrations` non-interactively. *Expected*: the migration is written. *Actual*: `AttributeError: 'str' object has no attribute '_meta'`. |
| *outcomes* | unpatched: pass; reference: pass; patch: fail, on every system's patch (8 of 8); next upstream release: pass. |
| *validators that missed it* | The hidden tests (every patch is resolved) and every released augmented test: all SWE-ABS tests for the instance pass on each patch and on the reference; PROBE keeps no test and UTBoost has no entry for it. |
| *certificate* | Unpatched code: the witness passes on the unpatched tree, so the patch broke behaviour that worked before. Both raters judged the expectation *not* entailed by the issue, so the issue branch does not certify it. |
| *shared by* | All 8 systems ship it; 4 of them run the same model. A majority-vote check cannot flag it (Observation 1). |

Table 16: Case 2: the change-1 "repair" that is the unchanged source program (Figure 1a).

| | |
|---|---|
| *change* | Change 1: a table costs three wood instead of two. |
| *reference* | The pinned engine under the changed rules. |
| *agent change* | None kept. In the primary run the repair arm started from the source program at 20 of 26 development transitions and 3 of 4 tasks, and made 8 proposal and repair calls (180,928 tokens). No revision scored above the source's 20 of 26, so the tie rule kept the source. The aggregate run repeated the pattern (171,507 tokens). |
| *witness* | The first failing plan in task order, a rule fixed before drawing: `place_table`, `make_wood_pickaxe`, `move_up`, `do`, starting with three wood. *Input*: the start state and `place_table`. *Expected* (engine): no wood left. *Actual* (program): one wood left, so it plans a wooden pickaxe that cannot be crafted. |
| *outcomes* | source rules: one wood left; engine: none; program: one. The program predicts the old rule exactly: a missing requirement, and a stale rule. |
| *validators that missed it* | The development gate saw the stale price: the source mispredicts 6 of the 9 table placements on the development panel, yet ties kept it. The planner's self-report predicted success. |
| *certificate* | Executed: the plan fails natively, and the replay locates the divergence at `place_table`. The program is confident on 60 change-1 audit tasks and fails on 8 (13.3%; Wilson 6.9–24.2%), each at `place_table` with the source rule's prediction. |

# G FORMAL DEFINITIONS AND PROOFS

The math here is short on purpose. It fixes one vocabulary for both testbeds (Table 19) and writes down the few facts the experiments lean on. Each is elementary, so we call it an observation. The weight sits in the data: whether real defects are shared (Section 5.3) and how often each check fires.

Table 17: Case 3: a collateral regression in a program perfect on its development transitions.

| | |
|---|---|
| *change* | Change 2: the wooden pickaxe also needs one stone. Movement is unchanged. |
| *reference* | The pinned engine under the changed rules; for movement it agrees with the source rules. |
| *agent change* | A post-change rebuild: a new program written from post-change transitions only, accepted with all 26 development transitions correct and 25 of 27 sealed held-out transitions. |
| *witness* | *Input*: the start state of a build-a-tool-and-mine task and `move_down`, the first action of the plan `move_down`, `place_table`, `make_wood_pickaxe`, `move_left`, `do`. *Expected* (engine): the player's facing changes. *Actual* (program): the predicted facing differs from the engine's, so the later actions target the wrong cell. |
| *outcomes* | source rules and engine agree at this step; the program alone differs, so the failure is a collateral regression under both classification rules. |
| *validators that missed it* | The development gate, which it passes perfectly: the panel holds 3 movement transitions and the program predicts them, but not this state. The planner's self-report predicted success. |
| *certificate* | Executed: 6 of its 36 confident plans on the audit panel fail, all diverging at `move_down`. Of the 9 accepted programs perfect on development data, it is the only one that fails; the sealed gate, which requires all 27 held-out transitions, rejects it. |

Table 18: Case 4: a certified collateral regression from the random leaderboard sample (django-11163).

| | |
|---|---|
| *change (issue)* | `model_to_dict(instance, fields=[])` should return an empty dict, because no field was requested; it returns all fields. The issue proposes testing `fields is not None` instead of the truth value of `fields`. |
| *agent change* | A resolved patch from one of the eight top systems, drawn at random. |
| *witness* | An audit test written by the GPT-6 Astra auditor, blind to the reference. *Input*: `fields = []`; build and instantiate an inline formset from `fields`, then build a model form for the child model from the same list. *Expected*: the form has no fields. *Actual*: `AssertionError: Lists differ: ['parent'] != []`. |
| *outcomes* | unpatched: pass; reference: pass; patch: fail; identical on five re-runs of each tree. |
| *validators that missed it* | The hidden tests (the patch is resolved); every released SWE-ABS and PROBE test for the instance; and the Gemini auditor, which reported no violation. |
| *certificate* | Unpatched code. Both raters answered that the issue does not request the changed behaviour, so the veto did not fire; the issue branch does not certify it. The failure is an assertion on a value, so it passes the construct gate. On the same unit, another counterexample was removed because one rater judged the change permitted by the issue: the veto removes, it never adds. |

**Vocabulary.**

*Confident plan:* its model predicts it reaches the goal.
*Stale rule:* a missing requirement that predicts the old rule exactly.
*Certified counterexample:* its certificate survives the checks of Algorithm 2.

**Definition 2** (Revision task and defect)**.**

*Artifact:* answers a query $q$ with a reproduced behaviour $\beta(\alpha, q)$.
*Task:* a prior artifact $\alpha_0$, a change $\Delta$, and a pre-change reference $\rho_0$ that was correct before the change.
*Acceptable set:* fixed by reality, $\mathcal{A}_q$ for each $q$.
*Defect:* a revision $\alpha_1$ has a defect, $D = 1$, iff

$$\beta(\alpha_1, q) \notin \mathcal{A}_q \quad \text{for some } q.$$

**Definition 3** (Witness and executed counterexample)**.**

*Validator:* $V$ accepts or rejects a revision using checks fixed before scoring.
*Witness:* $W = (q_W, e_W)$ fails on a system $\tau$, written $W(\tau) = 1$, iff $\beta(\tau, q_W) \neq e_W$.

Table 19: Notation, with its instantiation in the two testbeds (Definitions 2 to 4).

| symbol | meaning | (I) code | (II) world models |
|---|---|---|---|
| $\alpha_0$, $\alpha_1$ | prior artifact; its revision | unpatched tree $b$; $b$ plus the agent patch | retained source program; the revised program |
| $\Delta$ | the change | issue text $\iota$ | hidden rule change (table cost; pickaxe recipe) |
| $\rho_0$ | pre-change reference | $b$ | engine under the source rules |
| $q$, $\beta(\alpha, q)$ | query; behaviour | test input; value or exception | state–action pair; predicted next state |
| $\mathcal{A}_q$ | acceptable behaviours | what the issue requires (not executable) | $\{\mathcal{R}_1(q)\}$, engine under the changed rules |
| $\tau^*$ | executable reference | $b{+}g$, the reference patch applied | $\mathcal{R}_1$ |
| $V$ | validator | hidden tests; released augmented tests | development-accuracy gate, then the planner's self-report |
| $W = (q_W, e_W)$ | witness | an audit test with its expectation | first divergent step $q^*$ of a failing plan, with $\mathcal{R}_1(q^*)$ |
| $O_{\text{req}}$ | requirement check | two blind raters: issue entails $e_W$ | $\rho_0(q^*) \neq \mathcal{R}_1(q^*)$ (state rule) |
| $O_{\text{reg}}$ | regression check | $W$ passes on $b$, fails on the patch | $\rho_0(q^*) = \mathcal{R}_1(q^*)$ and the program errs |

*Executed counterexample:* $W$ is one for $\alpha_1$ iff, each reproduced,

$$W(\alpha_1) = 1 \quad \text{and} \quad W(\tau^*) = 0.$$

In our two testbeds:

- World models: $\tau^* = \mathcal{R}_1$, so a counterexample is a defect.
- Code: $\tau^* = b{+}g$, so it is one only if the reference does what the issue requires; the checks below settle that.

**Definition 4** (Requirement and regression checks)**.**

*Requirement check:* $O_{\text{req}}(W) = 1$ iff the change requires $e_W$ at $q_W$.
*Regression check:* $O_{\text{reg}}(W) = 1$ iff the change leaves $q_W$ alone ($G(W) = 1$) and

$$W(\rho_0) = 0, \qquad W(\alpha_1) = 1.$$

*Certified:* $W$ is certified without a reference solution iff $O_{\text{req}}(W) = 1$ or $O_{\text{reg}}(W) = 1$.

In our two testbeds:

- Code: $O_{\text{req}}$ is two raters from different model families; $G$ comes from the reference (in the measurement) or from patch-blind issue judges (in the detector).
- World models: both checks run on the engine, $G(W) = \mathbf{1}[\rho_0(q^*) = \mathcal{R}_1(q^*)]$ and $O_{\text{req}} = 1 - G$, so the checks attribute failures that execution has already certified.

**What we estimate.** Over accepted revisions (resolved patches; confident plans, clustered by program):

- $A = 1$ when the instrument certifies a counterexample;
- $\text{Rej} = 1$ when the validator rejects.

If every certified witness is valid, $\Pr(A{=}1)$ is at most the defect rate.

**Observation 2** (Caught share and sensitivity)**.** *Among the defects our instrument finds, the share the validator caught overstates its sensitivity on all defects exactly when the instrument and the validator tend to find the same defects. Formally, if* $\{A{=}1\} \subseteq \{D{=}1\}$*, then*

$$\Pr(\text{Rej}{=}1 \mid A{=}1) - \Pr(\text{Rej}{=}1 \mid D{=}1) = \frac{\text{Cov}(A, \text{Rej} \mid D{=}1)}{\Pr(A{=}1 \mid D{=}1)}.$$

*Why it matters:* the caught shares we report are read as lower bounds on what slips through only under this condition.

*Proof.* Because $A = 1$ implies $D = 1$, $\Pr(\text{Rej}{=}1 \mid A{=}1) = \Pr(A{=}1, \text{Rej}{=}1 \mid D{=}1) / \Pr(A{=}1 \mid D{=}1)$. Write $\Pr(\text{Rej}{=}1 \mid D{=}1)$ over the same denominator and subtract. □

**Proof of Observation 1.** We read the terms as follows.

*Anonymous:* the flag is a fixed function of the revision's probe profile, the multiset of all profiles and side information that does not reveal the acceptable sets.
*Departs from shared behaviour:* a flag needs some valid probe $q$ where the revision's output differs from the value $\mu_q$ held by a majority or plurality of the pool.
*Shared defect:* the revision outputs $\mu_q$ on every valid probe that reveals it.

Then no revealing probe can trigger a flag, and a revision that outputs $\mu_q$ on every valid probe is never flagged. If all profiles coincide, the function gets the same input for every revision, so it flags all or none.

When $\mu_q$ itself is unacceptable, the revisions that behave correctly at $q$ are the ones that depart; the majority vote does this on django-16661 (Appendix E). None of this applies to checks that read the change or the pre-change reference, which is why the detector of Section 6 uses $O_{\text{req}}$ and $O_{\text{reg}}$.

## H WHY FLAGS FROM THE TWO CHECKS CARRY INFORMATION

Three short facts sit behind the detector results: when the two checks imply a defect, how rater errors limit a certificate's error, and why a flag's precision tracks its likelihood ratio rather than its recall. Notation follows Definition 4; $\mathcal{A}_q^0$ is the acceptable set before the change.

We use two assumptions.

*Frame:* where the change neither asks for nor permits a new behaviour ($G(W) = 1$), the acceptable set is unchanged, $\mathcal{A}_q = \mathcal{A}_q^0$; this is the convention behind pass-to-pass tests.
*Tightness:* the witness only rejects behaviour that was already unacceptable, $\mathcal{A}_{q_W}^0 \subseteq \{e_W\}$.

The construct and stability steps of Algorithm 2 and the detector's masking and benign/regression judge aim at tightness, though nothing guarantees either.

**Observation 3** (When the checks imply a defect)**.** *A witness certified by the requirement check marks a defect. So does one certified by the regression check, if frame and tightness hold at $q_W$. Without the scope guard $G$ the regression check fails: if the change asks for a new behaviour at $q_W$, every correct revision fails a witness the unpatched code passes.*

*Why it matters:* it is the reason the detector compares with the unpatched code and keeps a scope judge.

*Proof.* If $O_{\text{req}}(W) = 1$ then $\mathcal{A}_{q_W} \subseteq \{e_W\}$, and $W(\alpha_1) = 1$ puts $\beta(\alpha_1, q_W)$ outside it. Under frame and tightness $\mathcal{A}_{q_W} = \mathcal{A}_{q_W}^0 \subseteq \{e_W\}$, and the same step applies. Without the guard, take $\mathcal{A}_{q_W}$ disjoint from $\{e_W\}$. □

**Observation 4** (Error of an estimated certificate)**.** *With two raters, a certificate on a revision without a defect needs a rater error on the requirement side, a rater miss on the scope side, or a failure of frame or tightness. Requiring both raters to agree, or letting either veto, caps each rater term at the smaller of the two raters' error rates, with no independence assumption.*

*Why it matters:* it justifies the two-rater veto used for the certified counts.

*Proof.* By Observation 3, a certificate on a revision with $D = 0$ must have passed through one of the three routes; each rater-dependent route requires both raters to err. Take the union. □

The veto costs valid certificates: one rater objects on 13 and the other on 2 of 14 base-certified units, and the count is 11 with the veto and 19 without it. We do not measure the rater error rates on the leaderboard sample; on coder-cleared controls the certificate fired on 1 of 13 (one-sided 95% upper bound 31.6%).

**The detector.** A probe whose output differs between $b$ and $b{+}p$ is a witness whose expectation is the unpatched output, so a detector flag is the regression check with a patch-blind scope judge. By Observation 3, a primary flag on a correct patch needs a judge error, or a failure of frame or tightness, on at least two of its probes. A reference-free auditor sets the expectation itself and gets no such guarantee, which fits the gap on reference patches (Section 6) and the post hoc variant that keeps only tests the unpatched code passes (34 to 15 false flags of 51).

**Observation 5** (Flag precision is Bayes' rule)**.** *Precision depends on recall and false-flag rate only through their ratio. For a flag with sensitivity $s$, false-flag rate $f > 0$ and prevalence $\pi$, with $\mathrm{LR}^+ = s/f$:*

$$\Pr(D = 1 \mid \textit{flag}) = \frac{\mathrm{LR}^+ \pi}{\mathrm{LR}^+ \pi + 1 - \pi}.$$

*Why it matters:* it is why we compare the two checks by likelihood ratio (Table 10). A flag with $\mathrm{LR}^+ = 1$ leaves the odds of a defect where they were, however high its recall.

*Proof.* Bayes' rule, dividing numerator and denominator by $f$. □

In our code testbed, with $s$ the attributed recall and $f$ the false-flag rate on all held-out negatives, pooled $\mathrm{LR}^+$ is 2.4 for the detector and 1.3 for the auditor. At $\pi = 17$ of 150, the frame's rate of witness-confirmed counterexamples (not a defect rate), this gives 24% and 14% (difference $[-1, +24]$ points; exploratory). On GPT-6 Sol the auditor flags 15 of 17 witness-confirmed positives and 15 of 17 reference patches, so its $\mathrm{LR}^+$ against reference patches is at the no-information value.

## I Pre-registration registry

*What this establishes: each study was frozen and hashed before its data, and every verdict is reported, those whose rule was not met included.* Each study froze, before its first model call or execution, a design file with the sample rule, the instruments, the outcomes and, for tests, a decision rule; measurements report their estimands whatever their value. Table 20 lists every study in the campaign that bears on this paper, including those whose rule was not met. Hash times are modification times of the hash files, in UTC, on 25–26 September 2026; some times written inside our logs came from a drifting clock, so the hashes and the timestamp proof, not logged times, are authoritative. A manifest of the earlier hashes carries an OpenTimestamps proof, which bounds when those hashes existed but not when each design was frozen. A post-hoc power audit found the prevalence test underpowered (probability 0.79 of an inconclusive result at a true rate of 3–4%), and recorded the patch-blind screening test as an instrument failure under its pre-registered stopping rule. The early screening tests that decided which directions to pursue cost $28.93 in API spend together, recomputed from the logged token counts.

## J Prompts

*What this establishes: the exact fixed text every LLM instrument received can be checked against the released files.* Table 21 lists, for each instrument, the file that holds its fixed prompt and that file's hash. Per-item content (issue text, patches, tests, transitions) is inserted where the templates show placeholders. The files are in the supplementary material, verbatim.

Table 20: Pre-registration registry. "Test" studies have a decision rule; "measurement" studies report estimands. The verdict is the frozen rule applied as written. Hash times: day of September 2026 and time, UTC; for the world-model replication, the recorded freeze time.

| study | hash | hashed | verdict |
|---|---|---|---|
| residual in the 2024 sample (test) | `a4e538ad` | 25, 03:11 | rule met: 14 of 150 (13 in the witness wording used here); 2 of 150, not met, under a broader reading of its attribution rule |
| patch-blind screening (test) | `d6c6eb98` | 25, 05:35 | not met (instrument failure) |
| cross-system slice (test) | `69a2163e` | 25, 06:32 | rule met on a selected slice |
| prevalence, static reading (test) | `21bc1c2a` | 25, 07:40 | inconclusive, recorded as not met: 4 of 150, a lower bound (13 items a later judge panel flagged were never executed) |
| reference-independence (test) | `44384543` | 25, 18:17 | stopped: 3 of 150 |
| majority-vote detector (test) | `56f32a95` | 25, 18:24 | not met: 0 of 16, 26 of 80 false flags |
| plan-stage predictions | `0b19515e` | 25, 18:35 | hashed, *never run* or scored |
| pairwise sharing (test) | `ea8c48dd` | 25, 18:38 | incomplete (97 units pending); exploratory look in Appendix E |
| blind certificate check, first protocol | `f1fb3f88` | 25, 18:40 | *never rated* |
| agentic auditor with the reference (test) | `e5ab8605` | 25, 18:51 | partial: 29 of 29 recall, 4 of 13 control flags, on 248 of 432 episodes |
| leaderboard audit (measurement) | `440d4384` | 25, 19:05 | reported (Section 5) |
| detector split (test) | `1f34851f` | 25, 19:06 | held-out units fixed |
| decoy specificity (test) | `930940e4` | 25, 19:12 | rule met: 32 of 32 |
| latent-class model (measurement) | `e9a7a49a` | 25, 19:32 | reported (Appendix D) |
| coder noise floor (measurement) | `84062368` | 25, 20:08 | reported: 1 of 150 false flags on reference patches; 3 of 50 binary labels flip on re-code |
| judge panel (measurement) | `239710e4` | 25, 20:44 | no frozen verdict changes |
| further auditors (measurement) | `1f16585d` | 25, 22:51 | reported (Appendix D); one deviation: working directories moved off a memory file system after a host restart |
| frontier models, pilot and main (measurement) | `6f15bec4` | 26, 00:01 | reported (Appendix D.1) |
| statistical analysis plan | `2240c01c` | 26, 00:51 | governs the code wording and certification |
| detector method and Holm amendment (test) | `66db067a`, `fdd2a3ce` | 26, 01:42 | rule met on false flags, 3 of 3 backbones |
| world-model audit (test) | `e385508c` | 26, 02:42 | rule met: upper bound 21.2%, 16 programs fail |
| blind certificate check, entailment instrument | `e85ab3c5` | 26, 04:32 | *abandoned* before any rating (Appendix B) |
| blind certificate check, paired results (test) | `d18e5121` | 26, 05:30 | no ratings at submission |
| world-model replication (test) | `0575b7e1` | 26, 06:12 | replicated: the pre-stated failure rule (cluster interval entirely below 5%) was not met (Appendix A) |
| world-model replication, extra runs (addendum) | `e6028dec` | 26, 07:58 | reported in separately labelled rows, never pooled (Appendix A) |

## K Relation to prior work

*What this establishes: prior studies audit a patch or a fixed model; this work audits the validator that accepts revisions, with executed evidence, in both code and world models.* Table 22 places this work beside studies of test-passing patches and of learned world models: what each checks a verdict against, whether its evidence is executed, and whether it covers world models, revision and the validator that accepts revisions.

Table 21: Fixed prompts of every LLM instrument. Each file is in the supplementary material, verbatim; the hash is the sha256 of the released file (first 16 hex digits).

| Instrument | Prompt file (supplement) | sha256 |
|---|---|---|
| Leaderboard agentic auditors, 2 arms (SYSTEM, USER, ENV_*) | prompts/auditor_leaderboard.txt | 7a9843fbc25c6ba3... |
| Four further auditors (same prompt text; SYSTEM, USER) | prompts/auditor_further.txt | 796bb2619f67ce46... |
| Auditors on frontier patches (same prompt text; SYSTEM, USER) | prompts/auditor_frontier.txt | a87d69e5dc0bd94f... |
| Certification issue raters (ENTAILED / NOT_ENTAILED); also the out-of-family re-rating | prompts/certification_rater.txt | 7aa547fcaa3030a5... |
| Certification raters for the leaderboard audit (byte-identical copy of the rater prompt above) | prompts/certification_rater_leaderboard.txt | 7aa547fcaa3030a5... |
| Witness redactor | prompts/witness_redactor.txt | 1ffef264639ff172... |
| Decoy checker | prompts/decoy_checker.txt | 919269b84dcfdd02... |
| Maintainer veto rater, REQUESTED / PERMITTED / NOT_REQUESTED | prompts/veto_rater.txt | eb1b1552da3463e6... |
| Seven-family judge panel (TRAILER; reuses the rater prompt and coding rubric) | prompts/judge_panel.txt | cb9abf89fcffffae... |
| Detector Oracle A: probe agent, patch-blind judge, scope check (A_SYSTEM, A_USER, A_JUDGE, A_SCOPE) | prompts/detector_oracles.txt | c8e0ef911d74e337... |
| Detector Oracle B: acceptance-check agent, strict judge (B_SYSTEM, B_USER, B_JUDGE_V2) | prompts/detector_oracles.txt | c8e0ef911d74e337... |
| Static-coding rubric shared by both static coders (identical bytes in every static-coding run) | prompts/static_coding_rubric.txt | 2d2df1b5ad854ed2... |
| Static coder, Gemini (inline JSON trailer); the coder noise floor reruns it unchanged | prompts/static_coder_gemini.txt | d1fdae8adb58cafb... |
| Static coder, Gemini, frozen copy (inline JSON trailer) | prompts/static_coder_gemini_frozen.txt | 99431e60df30df55... |
| Second static coder, Codex (TRAILER) | prompts/static_coder_codex.txt | 89c81e03177e3a4b... |
| World-model reviser, system prompt (SYSTEM_CODE) | prompts/wm_reviser.txt | 36be558e0f247e3f... |
| World-model repair/rebuild templates, first run | prompts/wm_repair_first_run.txt | 429032c8c3e4464f... |
| World-model repair/rebuild templates, format and policy runs | prompts/wm_repair_format_policy.txt | cdc88d23db7b48d4... |

Each prompt source file is released byte for byte as plain text under the name shown; rows that share a name share one file.

Table 22: **How this work relates to studies of test-passing patches and of world models.** "Reference": what a flag or verdict is checked against. "Executed": whether the evidence is a run of the artifact (test or environment), as opposed to reading or judging it. W/R/V: covers learned world models (W), revision of an existing artifact (R), the validator that accepts revisions (V). Headline numbers are as each paper reports them, on its own population and conditions; they are not comparable across rows. p. = PDF page of the arXiv version read.

| Work | Measures | Reference | Executed | Sample; headline (page) | W/R/V | Not measured |
|---|---|---|---|---|---|---|
| SWE-Bench+ (Aleithan et al., 2024) | Solution leakage and weak tests among resolved patches | Developer PR | Manual screening | One system (SWE-Agent+GPT-4); 31.08% of passed patches suspicious from weak tests (p. 1) | – – – | Executed counterexamples; many systems |
| PatchDiff (Wang et al., 2026) | Behavioural divergence of plausible patches from the developer patch | Developer patch | Yes, plus manual review | Three 2024 tools; 260 of 877 plausible patches (29.6%) diverge; 7.8% fail the developer suite (pp. 1, 5) | – – ✓ | Patches that survive later augmenters; shared failures |
| UTBoost (Yu et al., 2025) | Passing patches rejected by generated tests | Gold patch (intramorphic) | Yes, plus manual review | 36 instances with weak tests; 345 patches labelled passed are rejected (Lite and Verified) (p. 1) | – – ✓ | Patches its tests do not reach |
| SWE-ABS (Yu et al., 2026) | Passing patches rejected by coverage- and mutation-driven tests | Gold patch | Yes | Top-30 agents; 2,184 of 11,041 passing patches rejected (19.78%) (p. 1) | – – ✓ | Patches its tests do not reach |
| PROBE (Li et al., 2026) | Passing patches rejected by tests from surviving variants of the reference | Reference patch | Yes | Top-10 agents; resolved rates fall by 4.2–9.0% (p. 1) | – – ✓ | Patches its tests do not reach |
| SWE-Proof (Ma et al., 2026) (concurrent) | Formal verification of patches; counterexample audit of test-passing patches | Formal spec. built from gold; claim must hold on gold | Yes (re-run) | One model (Opus 4.8), own scaffold, 500 Verified; audit lowers resolution from 85.0% to 58.2% (p. 7); violations in 122 of 421 resolving submissions of a second setting (p. 44) | – – ✓ | Public leaderboard patches; residual after augmenters; shared failures |
| Vafa et al. (2024) | Coherence of world models implicit in sequence models | True automaton | Yes | Taxi, Othello, logic puzzles; shortest-path model's valid traversals fall from 0.99 to 0.08 at 10% random detours (p. 9) | ✓ – – | Explicit programs; revision; acceptance gates |
| WorldCoder (Tang et al., 2024) | Learning and refining program world models for planning | Environment | Yes | Sokoban (incl. added teleport rules), Minigrid curriculum, AlfWorld; sample efficiency and transfer (pp. 6–7) | ✓ ✓ – | Failures of accepted programs' plans |
| PoE-World (Piriyakulkij et al., 2025) | Products of programmatic experts as world models for planning | Environment | Yes | Pong, Montezuma's Revenge and altered versions; game score (pp. 6–7) | ✓ ✓ – | Failures of accepted programs' plans |
| OneLife (Khan et al., 2026) | Symbolic world models from one unguided episode | Environment | Yes | Crafter-OO; beats a baseline on 16 of 23 mechanics by state ranking (pp. 1–2) | ✓ – – | Hidden rule changes; plan failures |
| **This work** | **Executed failures of what a validator accepts; which checks still catch them; agreement** | **World models: environment. Code: flag needs the reference to pass; certificates use base or issue** | **Yes** | **World models: 265 of 1,392 confident plans of accepted revisions fail (19.0%; rejected 23.3%). Code: 18 of 62 flagged (29.0%; controls 30.8%); 5 of 62 not caught by released augmenters** | ✓ ✓ ✓ | **Correctness of any patch; population-scale rates; formal proofs** |