# SNAPSHOT-CORRECT, REPLAY-WRONG: GRADING GENERATED DATA PIPELINES OVER TIME

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

## ABSTRACT

Agent-written data pipelines can pass the test they ship with and still corrupt the tables they maintain: executed a second time on the same records, or on records delivered late, twice or out of order, they go silently wrong, and a benchmark that runs a generated artifact once cannot see it. Grading eleven model arms on 40 tasks we built against a fixed snapshot, as seven of nine agent benchmarks we audit do, passes 86.0–100.0% of the pipelines that run; re-executing those same artifacts under duplicate, delayed, reordered and retried delivery finds 7.0–79.2% of the passing ones wrong (1.7–78.1% without five tasks an independent reviewer found ambiguously worded). dbt's default grain tests, placed on each task's key, catch 4 of 110 such failures: a uniqueness test on a business key is satisfied by exactly the upsert that double-counts. In exploratory analyses, the algebra of the update predicts which hazard class a pipeline fails, though not how often: additive folds double-count a redelivered record where naturally idempotent ones absorb it, a gap of +42.2 [+26.6, +58.4] points that survives a port to Python. The defect also has a locus we can move: rewriting uniqueness to the event-arrival boundary raises idempotency pass rates 74.7 and 67.0 points on DuckDB and PostgreSQL where an entity-keyed rewrite and a syntax-preserving one do not. Telling a model about the property narrows the gap without closing the spread between models: the share of passing pipelines that are wrong falls in all 11 arms (significantly in 10 after adjusting for multiplicity) and reaches zero on 3, yet all-in (silent-or-crash) failure still spans 0.0–71.0%, and on Sonnet 5 much of the fall is silent wrongness traded for crashes on a constraint the model itself declares. Evaluating stateful generated code therefore means perturbing its execution history rather than grading one snapshot of it.

## 1 INTRODUCTION

Suppose a model writes a pipeline that computes yesterday's revenue, and the pipeline passes the test that ships with the task. If the upstream system then delivers a batch twice, as at-least-once delivery allows, the revenue comes out wrong and nothing in the run says so.

Production data arrives late, twice and out of order by design. Kafka, for instance, promises at-least-once delivery, in which "messages are never lost but may be redelivered" (Apache Software Foundation, 2025a; Debezium Community, 2025; Apache Software Foundation, 2025b). We call a pipeline *replay invariant* when the state it reaches under any such delivery schedule equals a batch recomputation over the same records. A crash is easy to notice because the run exits with an error. A wrong table is not, and most agent benchmarks for data work cannot see one, because they execute the generated artifact once and grade what it leaves behind (Section 2).

We measure how often this happens. Eleven model arms from eight organisations write pipelines that a single execution certifies and re-execution shows to be wrong. The grain tests a dbt project declares by default do not catch these failures (Section 5.1). The algebra of the target predicts which hazard a pipeline fails, and rewriting where the generated program enforces uniqueness moves one locus of the defect. Telling the model about the property lowers the failure rate in every arm but leaves the models as far apart as before, and on some arms it changes the kind of failure instead of removing it.



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.

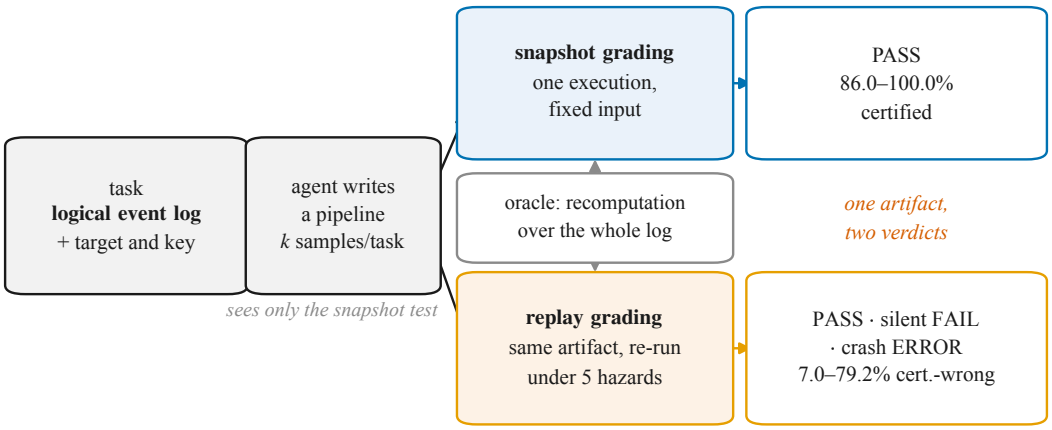


Figure 1: Overview. The agent sees only the snapshot test while it writes a pipeline. The stored artifact is then graded twice: on a fixed input, as seven of the nine benchmarks in Table 1 do, and under schedules that deliver the same records late, twice, out of order, after a crash or as a backfill. The oracle recomputes the target from the complete log, so a wrong table and a crash are separate verdicts.

**Contributions.**

1. **A replay-invariance benchmark with a recomputation oracle** (Sections 3–5.1). Graded once, 86.0–100.0% of the pipelines that run pass. Re-executed under delivery hazards, 7.0–79.2% of those passing pipelines are wrong, and dbt's grain tests catch 4 of 110 such failures. Two oracles that share no grading code agree on 63,995 verdicts.
2. **The algebra of the update predicts which hazard fails** (Section 5.2; exploratory). Additive folds double-count a redelivered record where naturally idempotent ones absorb it, a margin of +42.2 [+26.6, +58.4] points that survives a port to Python. The algebra explains which hazard fails, not how often: it accounts for 7.2% of the variance in idempotency outcomes, against 20.3% for the model.
3. **One locus of the defect can be moved** (Sections 5.3 and 6.2; exploratory). Enforcing uniqueness at the event-arrival boundary raises idempotency pass rates by +74.7 and +67.0 points on two engines with ordering unchanged. Applied deterministically, the same rewrite cuts all-in failure by 26.5 and 23.7 points without a model call, provided the pipeline carries a stable per-record identity.
4. **Stating the property narrows the gap but not the spread** (Sections 5.1 and 6.1). The certified-wrong rate falls in all 11 arms (significantly in 10 after adjusting for multiplicity) and reaches zero in 3, yet the arms stay apart. On Sonnet 5 much of the fall is silent wrongness traded for crashes on a constraint the model itself declares.

## 2 Related Work

Table 1 compares nine agent benchmarks for data work. Two are text-to-SQL (Lei et al., 2025; Chen et al., 2024), where a single execution is the right design because a query keeps no state between runs. The other seven (Jin et al., 2025; Lai et al., 2025; Huang et al., 2024; Sun et al., 2026; Huang et al., 2026; dbt Labs, 2025; UC Berkeley EPIC Lab, 2025) grade an artifact that does keep state, and each runs it once before grading. Two of them vary something around that run, but not delivery: ADE-Bench changes the database the agent works against, and DataAgentBench runs the *agent* five times on identical data, which tests the agent's stability rather than the artifact's. Three suites released alongside this work keep the same design (Snowflake AI Research, 2026; Meng et al., 2026; Pham et al., 2026). We instead re-execute the same artifact under up to 8 delivery schedules and separate a wrong table from a crash.

Table 1: Agent benchmarks for data work whose grading code we read. *Runs* is how many times the artifact executes before it is graded, and *what varies* is what differs between those runs. None of the stateful suites claims to test replay, so the table describes what they measure rather than a flaw in them.

| Benchmark | Artifact | Runs | What varies | Verdicts |
|---|---|---|---|---|
| Spider 2.0 | query | 1 | none | pass/fail |
| BEAVER | query | 1 | none | pass/fail |
| DA-Code | stateful | 1 | none | pass/fail |
| ELT-Bench | stateful | 1 | none | pass/fail |
| KramaBench | stateful | 1 | none | pass/fail |
| CDR-Bench | stateful | 1 | none | pass/fail |
| AgenticDataBench | stateful | 1 | none | pass/fail |
| ADE-Bench | stateful | 1 | database | pass/fail |
| DataAgentBench | stateful | 1 | agent run | Pass@1 |
| **This work** | stateful | up to 8 | delivery schedule | typed |

SPARK (Xu et al., 2026) shows that frontier models leave secure defaults unused under a plain prompt and recover them when told to; we find the same, and show where the recovery stops. Tyen et al. (2024) report that models can correct an error once told where it is but cannot locate it themselves. Our setting has the same shape, with the location a program variable and the ground truth executable. In method, the benchmark is metamorphic testing (Chen et al., 2018) with an environmental transformation, and it borrows from fault injection with a history checker (Kingsbury & Alvaro, 2020) and from deterministic simulation testing (Zhou et al., 2021). Deduplication by record identity is standard engineering; what we measure is where generated code puts it. Appendix S1 covers the rest.

## 3 Benchmark Design

**Replay invariance.** A task defines a logical event log, and a *schedule* splits it into delivery batches, possibly duplicating records, reordering them, holding them back or aborting a batch midway. Write $\sigma \oplus B = P(\sigma, B)$ for the state a pipeline $P$ reaches from state $\sigma$ after batch $B$. Then $P$ is replay invariant over a family of schedules if

$$\text{final_state}(P, s) \;=\; \text{batch_recompute}(\text{complete log})$$

for every schedule $s$ in the family. We can only run a finite set of schedules, so a pass means that none of them falsified the pipeline; it is not a proof of invariance. Ordering hazards require $\oplus$ to be commutative and associative over batches, and idempotency hazards also require $\sigma \oplus B \oplus B = \sigma \oplus B$. Generated code treats the two very differently: idempotency fails far more often, and every pipeline in our corpus that fails an ordering hazard also fails an idempotency one (Appendix S15).

**Coverage of the schedule family.** For each of the 1027 cells the fixed family certifies, we drew 20 random delivery histories. They falsify 9 (0.9%), 8 of them silently. As a positive control, the same draws catch 60 of 60 cells the fixed family already falsifies. The generator therefore finds failures of the kinds the family targets, though it cannot rule out failures of other kinds (Appendix S14).

**Tasks and hazards.** The suite has 40 tasks, each with declared hazards. The task set was fixed before any arm ran: 50 tasks exist and the 40 used predate every arm. Of these, 10 were written by hand, 19 drafted with the LLM assistant used to build this project and 11 templated. All passed a mechanical check of their gold pipelines and broken controls, and an independent data engineer then reviewed all 40 without being told how each was made (Appendix S8). The certified-wrong rate is 48.3, 42.0 and 53.2% across the three groups (Appendix S15). We rewrote 10 starter scripts after the unhinted baselines ran, so the 7 later +hint arms and Opus 5's +mech saw the new text; on the 30 unchanged tasks each of their effects holds (Appendix S8). Five hazards carry the results: two ordering hazards (late arrival, out-of-order CDC) and three idempotency hazards (duplicate delivery, crash-retry, backfill), beside a benign steady stream that replays nothing. A sixth, schema evolution, is graded on two tasks as a loud hazard and belongs to neither group. The benchmark covers pipelines

that consume a delivery stream; pipelines that aggregate other maintained models are out of scope, since the arrival boundary is not where their guard would go.

**Execution.** All batches of a schedule run on one database connection, and only crash-retry reopens it. This models a long-lived consumer, whereas dbt, Airflow and cron open a fresh session for each invocation. Section 5.1 re-grades every unhinted arm with a fresh session per batch (Appendix S2). Every arm is told that its script *runs once per delivery batch against a database that persists between runs*, to use `CREATE TABLE IF NOT EXISTS`, and that any other tables it creates also persist. Within a cell the agent has 3 turns: it writes a script, the harness runs the task's snapshot test, and after a failure the agent sees the error or a typed diff of the target and may revise. It never sees a hazard, a schedule or a replay verdict, so the snapshot pass rates in Table 2 are the strongest form of a snapshot baseline we could give. Cells that took more turns fail replay +0.2 [−6.6, +6.6] points more within an arm and task, an interval too wide to rule out a small effect.

## 4 Experimental Setup

**Models and arms.** An *arm* is a model under one prompt condition. We grade eleven models from eight organisations under the unhinted prompt, all eleven again under +hint, a paragraph that lists the delivery hazards, and a subset under two further instructions (+arch and +mech, Section 6.1). Each arm writes $k = 5$ pipelines per task. Sample 0 is drawn at temperature 0 and the others at 1.0; splitting by sample index separates none of the 8 arms that vary. No reasoning-effort or thinking parameter is sent, so reasoning models run at their provider's default in every arm (Appendix S6).

**Metrics.** A *cell* is one generated pipeline, a (task, sample) pair within an arm, graded under every schedule. Verdicts are typed: PASS, FAIL for a wrong final state and ERROR for a crash. Over a cell, *crash* means the snapshot or a graded hazard raised an error, *silent* means the snapshot passed, nothing crashed and a hazard produced a wrong state, and *all-in* is their union. Our main metric, the *certified-wrong rate*, is silent failure among the cells whose snapshot passes, computed per task and averaged over tasks. It asks how many of the artifacts a snapshot test certifies are wrong under replay. Intervals are 95% task-clustered bootstraps.

**Oracles.** The oracle recomputes the target from the complete log, so it needs no classifier to decide a verdict. A second implementation that shares no grading code agrees on all 46,763 DuckDB verdicts of 30 arms and all 17,232 PostgreSQL verdicts, and, separately, on the 8,076 and 17,232 verdicts of the rewritten PostgreSQL pipelines (Appendix S8). Classifiers are used in four places, none of them a verdict: one selects the population of the scope rewrite, two sort the cells of Section 6.1 by route and by constraint, and one labels each task's algebra from its reference SQL. Two blind raters labelled the additive and idempotent tasks ($\kappa = 0.95$) and agreed on 37 of 38; the labelling function matches them on all 37. Appendix A lists each claim with its design and status.

## 5 Results

### 5.1 Snapshot grading misses replay failures

Table 2 is our main result. Snapshot grading scores the models almost identically and almost perfectly, while under replay the same artifacts span 7.0–79.2% certified-wrong. Without Codestral, the one arm that fails broadly rather than only under replay, the range is still 7.0–73.7%. Without the 5 tasks whose wording the independent reviewer found ambiguous, whose prompts are unchanged in the graded corpus, it is 1.7–78.1%. Three of them define a balance, and they hold 9.4–64.3% of each arm's unhinted certified-wrong cells (Appendix S8). Re-grading all 11 arms with a fresh session per batch leaves the range unchanged and moves no arm by more than 5.7 points (Appendix S2). Adjacent rows are not resolved: their intervals overlap, and a single draw per task misorders 8.0% of arm pairs, with the ranking settling at $k = 3$ (Appendix S15). The table supports tiers rather than an exact ranking.

Table 3 shows where the gap sits. Every arm passes duplicate delivery less often than either ordering hazard, and ordering alone separates the weaker arms from the stronger ones.

Table 2: Snapshot and replay grading of the same pipelines, eleven arms, ordered by unhinted certified-wrong. *ord.* and *idem.* are mean pass rates over every verdict of each hazard group, crashes included. Certified-wrong is silent failure among cells whose snapshot passes; all-in also counts crashes, so certified-wrong can exceed all-in on an arm that crashes often (Codestral under +hint). A 0.0% is a bound, not a proof (Appendix S14). Seven later +hint arms saw rewritten starters on ten tasks; every hinted fall holds on the other thirty (Appendix S8). Snapshot pass counts pipelines that run (79.5–100.0% if crashes count as failures). Table S9 gives silent, crash, all-in and any-failure rates for every arm on one denominator.

| | unhinted | | | | | under +hint | |
|---|---|---|---|---|---|---|---|
| | snapshot | replay grading | | | | | |
| Model | pass | cert.-wrong (95% CI) | all-in | ord. | idem. | cert.-wrong | all-in |
| GPT-5.6 Sol | 100.0% | 7.0% [1.5, 14.5] | 7.0% | 100.0% | 93.0% | 0.0% | 0.0% |
| Opus 5 | 100.0% | 16.7% [8.5, 26.4] | 21.0% | 94.5% | 89.3% | 0.0% | 1.5% |
| Kimi K3 | 99.5% | 35.2% [22.7, 48.8] | 36.0% | 97.3% | 69.3% | 0.0% | 0.5% |
| DeepSeek V4 Pro | 100.0% | 45.3% [31.9, 59.0] | 52.0% | 87.4% | 55.8% | 13.1% | 35.0% |
| Sonnet 5 | 100.0% | 48.2% [32.8, 63.6] | 51.0% | 98.4% | 67.2% | 10.5% | 44.0% |
| MiniMax M3 | 99.5% | 50.2% [38.0, 62.6] | 56.5% | 84.3% | 57.8% | 10.9% | 32.5% |
| Haiku 4.5 | 98.9% | 52.2% [37.4, 66.9] | 67.0% | 67.9% | 44.5% | 35.1% | 53.5% |
| DeepSeek V4 Flash | 97.9% | 52.4% [38.6, 66.2] | 62.5% | 72.8% | 45.2% | 28.9% | 49.5% |
| Gemma 4 26B | 99.5% | 54.7% [39.1, 70.4] | 70.0% | 82.7% | 47.8% | 20.4% | 61.0% |
| Qwen3-Coder-Next | 99.2% | 73.7% [60.6, 85.7] | 78.5% | 43.2% | 31.8% | 58.1% | 71.0% |
| Codestral | 86.0% | 79.2% [67.2, 89.9] | 72.0% | 36.6% | 30.0% | 72.7% | 70.0% |

Table 3: Pass rate per hazard over every verdict, crashes included, with the mean crash share. *Hinted* is the mean pass rate under +hint, crash share in brackets. Rows are ordered as in Table 2.

| | ordering: pass% | | idempotency: pass% | | | all hazards | |
|---|---|---|---|---|---|---|---|
| Model | late | o-o-o | dup | retry | backfill | crash% | hinted |
| GPT-5.6 Sol | 100 | 100 | 93 | 93 | 93 | 0 | 100 (0) |
| Opus 5 | 95 | 94 | 93 | 80 | 95 | 5 | 98 (2) |
| Kimi K3 | 97 | 97 | 66 | 72 | 71 | 2 | 99 (1) |
| DeepSeek V4 Pro | 88 | 87 | 49 | 60 | 58 | 9 | 73 (21) |
| Sonnet 5 | 98 | 98 | 49 | 76 | 76 | 2 | 90 (8) |
| MiniMax M3 | 86 | 83 | 46 | 63 | 64 | 6 | 87 (10) |
| Haiku 4.5 | 68 | 67 | 38 | 50 | 46 | 19 | 62 (22) |
| DeepSeek V4 Flash | 75 | 71 | 41 | 50 | 44 | 13 | 60 (24) |
| Gemma 4 26B | 84 | 82 | 34 | 55 | 55 | 18 | 59 (36) |
| Qwen3-Coder-Next | 46 | 41 | 28 | 34 | 34 | 23 | 37 (37) |
| Codestral | 40 | 33 | 28 | 30 | 32 | 12 | 36 (11) |

**Default grain tests.** We placed dbt's `unique` and `not_null` grain tests on each task's declared key and ran them over the final tables of 110 sampled silent failures. They catch 4, with 0 false alarms on 55 passing controls, and 0 of the 61 upsert failures that are this paper's mechanism (Appendix S13).

**Stating the hazards.** If the spread were only a gap in the specification, stating the hazards would close it. We therefore ran +hint on all 11 arms. Certified-wrong falls in every one, significantly in 10 after Holm adjustment (Appendix S9), by 24.1 points on average, and 3 arms reach zero. The arms nevertheless stay spread across 0.0–72.7% with their order nearly intact (Goodman–Kruskal $\gamma = 0.81$, a rank correlation over the pairs the hint leaves untied), and the bottom arm moves least: Codestral falls 6.5 points ($p = 0.077$ before adjustment) against 39.3 in the middle. A zero is a bound of at most 7.5% at 95% if failures cluster by task, and composing hazards breaks no certified hinted cell in any arm, so every arm keeps its hinted rate and rank (Appendix S14). Seven of these arms saw the rewritten starters of Section 3, and every fall holds on the 30 unchanged tasks. Certified-wrong excludes crashes, so a fall does not mean the pipelines work: under +hint, all-in failure still spans

0.0–71.0%. With the hazards stated, the spread that remains lies below the three arms at zero; under the unhinted prompt it reaches the strongest arms too (Table 2).

### 5.2 The algebra of the update predicts which hazard fails

Ordering hazards need commutativity, and redelivery also needs idempotence. This is settled systems knowledge. Helland & Campbell (2009) name associativity, commutativity and idempotence as what lets work succeed "at least once, anywhere, in any order"; CRDTs are built on these properties (Shapiro et al., 2011); CALM ties coordination-freedom to monotonicity (Hellerstein & Alvaro, 2020); equality with a batch recomputation is the correctness criterion of incremental view maintenance (McSherry et al., 2013; Budiu et al., 2022); and exactly-once delivery has its own lineage (Akidau et al., 2013; 2015). A sum is commutative but not idempotent, so it double-counts a redelivered record, whereas a last-write-wins upsert keyed on the entity, or a maximum, absorbs it. Our observation is that *generated* code follows this algebra, so the target's algebra predicts which hazard a pipeline fails before anyone reads the pipeline. We noticed this after seeing the outcomes, and the analysis is therefore exploratory.

The analysis uses 59 tasks with verdicts in both hazard groups: 38 from the corpus, written before the hypothesis, and 10 plus 11 written later, the latter to balance the algebra classes. The margin between ordering and idempotency pass rates is +39.3 points on 20 additive tasks and −4.2 on 18 naturally idempotent ones. The difference in mean per-task margin is +42.2 [+26.6, +58.4], and +48.8 [+29.8, +67.5] on the 38 tasks written before the hypothesis. The three balance tasks the reviewer flagged are additive and in this contrast; without them it is +35.4 [+19.8, +52.0] over 17 additive tasks, and the other two flagged tasks are not in it (pre-registration §6qq). The margin also survives a change of language. On 15 tasks ported to a Python state store, where the agent writes a `run_batch` function against a key-value interface, the additive margin is +65.8 [+48.4, +77.2] against +69.0 [+46.2, +89.3] in SQL (−3.2 [−16.9, +9.3]). Idempotence under redelivery is a property of the fold, and it fails the same way in both languages (Appendix S7).

### 5.3 Where uniqueness is enforced

An association between algebra and failure does not say where in the program the failure lives. To find out, we change where uniqueness is enforced, add the state needed to enforce it there, and leave everything else as the model wrote it. We chose this variable after seeing the outcomes, so this result is exploratory as well.

For every stored pipeline whose target accumulates, the rewrite admits each record at most once at the arrival boundary. It adds a persistent side table holding the identities admitted so far, and the pipeline reads a filtered view of the batch instead of the batch itself, so a target that groups its dimensions away never has to carry an identity. Aggregations, joins, filters and the output schema stay as the model wrote them. On DuckDB, over 362 pipelines, idempotency pass rates move +74.7 points and ordering verdicts 0.0. On PostgreSQL, over 339 pipelines written in that dialect, they move +67.0 and 0.0 (Figure 2). No snapshot verdict changes on either engine.

Two control rewrites show that the effect does not come from rewriting as such. The first enforces uniqueness at the *entity* level, with the same syntax and the same amount of code. It destroys ordering correctness (−81.0 points on DuckDB, −71.6 on PostgreSQL) and repairs nothing, while leaving the clean run intact, because it filters only against entities admitted in earlier batches. The second touches the same statements but applies no filter, and moves idempotency by +0.2 points. The scope of the filter is therefore *one* locus of the redelivery failure, though not necessarily the only one.

## 6 Interventions

### 6.1 Instructions

The models know something about the property. Asked whether their own pipelines still match a batch recomputation, most arms answer above chance, though only once the property is named (Appendix S5). An instruction also has a clear target, because one route nearly decides the outcome:

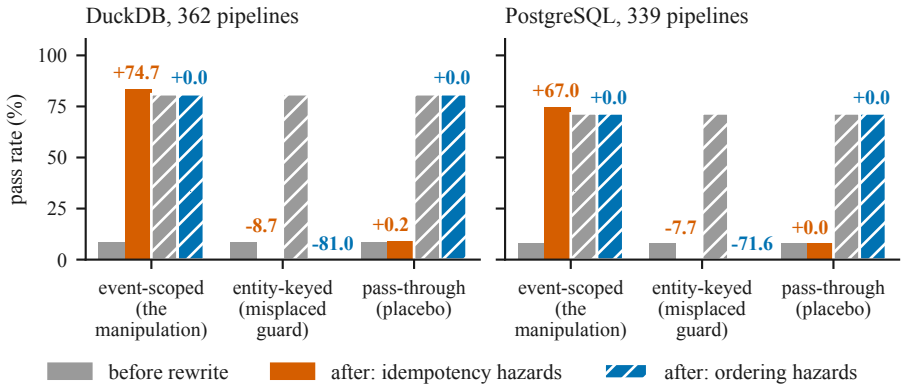


Figure 2: The scope rewrite and two control rewrites on both engines. Enforcing uniqueness at the arrival boundary repairs idempotency and leaves ordering unchanged; an entity-keyed guard destroys ordering instead, and a pass-through rewrite of the same statements changes nothing. Exploratory: the variable was chosen after the outcomes.

Table 4: Instruction arms scored by typed outcome. Δ silent and Δ crash are disjoint and sum to Δ all-in. *: against a same-time baseline; other rows are against stored baselines (Section 7). Task-clustered bootstraps, 4,000 resamples.

| Instruction | Model | Δ silent | Δ crash | Δ all-in |
|---|---|---|---|---|
| +hint | Haiku 4.5 | −12.5 [−20.0, −5.5] | −1.0 [−7.5, +5.5] | −13.5 [−22.0, −5.5] |
| | Sonnet 5 | −35.0 [−47.5, −22.5] | +28.0 [+17.0, +39.0] | −7.0 [−19.5, +6.0] |
| | Opus 5 | −16.0 [−25.0, −8.0] | −3.5 [−7.5, +0.0] | −19.5 [−29.0, −11.0] |
| | GPT-5.6 Sol | −7.0 [−14.5, −1.5] | +0.0 [+0.0, +0.0] | −7.0 [−14.5, −1.5] |
| +arch | Haiku 4.5 | −36.5 [−49.0, −24.5] | +1.5 [−7.5, +10.5] | −35.0 [−49.5, −20.5] |
| | Sonnet 5 | +5.5 [−5.0, +16.0] | +5.0 [−5.0, +15.0] | +10.5 [+2.5, +20.0] |
| | Opus 5 | −16.0 [−25.0, −8.0] | +1.0 [−3.5, +6.5] | −15.0 [−25.5, −6.0] |
| +mech | Haiku 4.5* | −10.0 [−20.0, −1.0] | +2.0 [−6.5, +10.5] | −8.0 [−17.5, +1.0] |
| | Sonnet 5* | −23.5 [−36.5, −11.5] | +37.5 [+25.0, +50.5] | +14.0 [+3.5, +25.5] |
| | Opus 5 | +17.5 [+7.5, +28.0] | −4.0 [−8.5, +0.0] | +13.5 [+3.0, +24.5] |

pipelines that accumulate into existing state pass 1.4% of idempotency verdicts over the 168 cells that take that route (Figure S3).

**Typed outcomes.** Scored on all-in alone, 2 of the 10 rows of Table 4 look like no change and 3 look like regressions. Split by type, they look different. Across all 11 arms, naming the hazards lowers silent failure in 10 but *raises* crashes in 6, from 5 organisations. The clearest case is Gemma 4 26B, which played no part in forming the hypothesis: silent wrongness falls −26.0 points while crashes rise 17.0, which all-in reports as −9.0. Enumerating the hazards lowers Haiku's and Opus's failure without adding crashes and brings Opus near zero, although Haiku's drop did not replicate in fresh samples. On Sonnet 5 it moves silent wrongness by −35.0 [−47.5, −22.5] while crashes rise +28.0 [+17.0, +39.0], so all-in reads −7.0 [−19.5, +6.0] and looks unchanged, even though the crash rise matches about 80% of the silence removed. Prescribing the level of uniqueness (+mech) goes further: against a same-time baseline, Sonnet moves −23.5 [−36.5, −11.5] silent and +37.5 [+25.0, +50.5] crash, which all-in scores as a +14.0 [+3.5, +25.5] regression. Opus goes the other way, with silent failure rising +17.5 [+7.5, +28.0]. We report the two outcomes separately because all-in nets a wrong table against a crash, and we did not measure which costs more in practice.

**Source of the added crashes.** A redelivered record violates a uniqueness constraint the model wrote into one of its own tables. On Sonnet these violations fire on 0.0% and 0.0% of verdicts under the benign schedule, which replays nothing, against 35.5% and 51.0% under duplicate delivery,

so they are not an engine checking constraints eagerly. The instruction also writes the repair of Section 5.3 into the code: the share of Sonnet's declaring cells that constrain the *event identity* rises from 38.3% to 85.0% under +hint and 93.5% under +mech. Opus does this without being told (83.7%).

**The window of the guard.** An anti-join against identities already consumed catches a batch delivered twice, but not two copies of an event that arrive in the same batch. Only duplicate-delivery schedules put two copies in one batch, and all of Sonnet's added crashes occur there. We tested this with a pre-registered intervention (pre-registration §6kk), re-running every stored Sonnet and Opus program on its duplicate-delivery schedule with only the same-batch copies removed. Sonnet's crash rate under +hint falls from 36.5% to 1.0%: 71 of its 73 crashes pass, none turns silent, and its crash excess over the unhinted arm falls from 28.0 to 0.5 points. Opus's crashes do not move. The mechanism is Sonnet's, not a general one: the same intervention removes 66.7% of MiniMax M3's excess and at most 4% of the other arms', whose added crashes have other causes (Appendix S15). Adding `ON CONFLICT DO NOTHING` to every constrained insert turns 98.6% of the +hint crashes caused by Sonnet's constraints into passes, but 20.5% of its +mech crashes into silent wrong answers (Appendix S2).

**Other levers.** Preference tuning on oracle-labelled pairs leaves failure where it was, a tool that lets the model re-run its own script shifts Haiku's failures toward silence, and a self-critique pass lowers all-in failure no more than a placebo review does (Appendix S5).

### 6.2 A deterministic repair

If the defect lies in where the guard is placed, placing the guard deterministically should repair the idempotency component and nothing else. We apply a transformation $R$ that filters each batch against the record identities the pipeline has already consumed. All-in failure falls 26.5 points on DuckDB and 23.7 on PostgreSQL (Table 6). On PostgreSQL, $R$ fixes 522 pipelines, breaks 3 that passed and changes 0 snapshot verdicts. On 200 failing and 50 passing cells, it fixes 56.5% and breaks 0 controls.

$R$ absorbs redelivery only under empty-batch neutrality, $P(\sigma, \emptyset) = \sigma$, which holds for 335 of the 338 accumulating DuckDB pipelines that run (24 crash throughout). The 3 without it still fail duplicate delivery after $R$, and so do 26 with it. $R$ gives no ordering guarantee, and it leaves every ordering verdict unchanged. Its statements also commit one at a time, so a run killed before the final insert into the consumed set retries over its own committed prefix, and 28 of 38 rewritten pipelines whose prefix commits a write fail crash-retry (Appendix B). Committing the body and that insert in one transaction fixes this and nothing else. Over 353 accumulating cells, crash-retry passes rise from 79.0% to 85.3%, the targeted pipelines go from 10 of 38 to 32 of 38 passing, and the only verdicts that change are 22 crash-retry failures that now pass. The atomic variant cannot be applied to the 9 pipelines that open their own transaction. We use $R$ to locate the defect; it is not an exactly-once scheme.

## 7 Limitations

We wrote the tasks, and all of them have one arrival interface and a stable record identity by construction. 11 of 12 incremental models in the public dbt projects that parse read external data. The one that does not fails by the mechanism of Section 6.1, but loudly, caught by the project's own uniqueness test, so it shows the mechanism outside our tasks and says nothing about how often it is silent (Appendix S16). None of our results speaks to real projects at scale, and we withdrew a census of public models (Appendix S12). The independent review shows that the tasks are well formed, not that they are representative, and we have not shown that a score here predicts pipeline work elsewhere. The mechanism was found after the fact and is labelled exploratory throughout; it most needs a prospective test on tasks written by people who have not seen it. With the contract stated, three frontier arms reach zero, so at the top the suite works better as a diagnostic protocol than as a leaderboard, and harder tasks are the obvious next step.

The silent-to-crash trade appears in 6 arms from 5 organisations, but the new arms are compared with stored baselines. Re-running 5 stored baselines at the same sample indices moved silent failure

down 0.5–5.0 points and crashes by −4.0 to +7.0, with no interval excluding zero. Against those later baselines, the silent fall holds in all 4 trade arms that have one, and the crash rise in 2 (Sonnet 5, DeepSeek V4 Flash); it spans zero for Qwen3-Coder-Next (+6.5 [−3.5, +16.5]) and MiniMax M3 (+4.5 [−1.5, +10.5]). The trade does not occur under +arch and is reversed on Opus 5, so it belongs to these instructions on these arms, not to instructions in general. Finally, we measure what a model knows with the property named in the prompt, which is weaker than knowing it unprompted.

## 8 Conclusion

A generated pipeline can pass every check a benchmark runs and still corrupt its table the second time it executes, on two engines and under two execution models. The tests a project carries by default do not catch this, and telling the model about the hazards narrows the gap without closing it. Stateful generated code should be evaluated by perturbing its execution history, not by grading one snapshot of it.

## AI use statement

We used generative AI tools in this work. They helped implement the evaluation harness and the analysis scripts, helped list candidate hypotheses and failure modes that we then specified, pre-registered and tested, and helped edit the prose. The LLM assistant used to build the project also drafted 19 of the 40 benchmark tasks; each passed the mechanical check described in Section 3, and an independent data engineer then reviewed all 40 tasks without being told which were drafted (Appendix S8). The research questions, the experimental design, the pre-registered readings, the interpretation of every result and the decisions about what to claim are our own. We reviewed all AI-assisted work. The analysis code was checked against two independently implemented oracles, which share no grading code and agree on 63,995 verdicts, and every number in the paper is generated from a stored analysis output by a script, with a test that fails when a number and its source disagree. The evaluated models are the objects of study; their outputs are experimental data and are not used as evidence about the work itself. We take responsibility for all text, claims and artifacts.

## Ethics statement

**Human participants.** Appendix S10 reports a small anchor in which four practising data engineers wrote pipelines from the unhinted prompt given to the models, producing 19 usable cells. Participation was voluntary, unpaid and by invitation among professional contacts. Participants were told what the task was, that their scripts would be graded by the same harness as the models, and that anonymised pass rates might be published; they consented on that basis and could withdraw. No personal or demographic data was collected, and the released artifact holds only their scripts and verdicts, checked for incidental identifiers. The exercise carried no foreseeable risk beyond ordinary professional work. It was not reviewed by an institutional board, and we report it as a check on a task-design objection, not as a comparison of humans and models.

**Mined code.** Appendices S12 and S16 read public dbt repositories under licences that permit reuse and report aggregate properties only. No repository, author or organisation is named, since the finding concerns a common idiom and not any team's work.

**Use and misuse.** The benchmark measures whether generated pipelines corrupt data under replay, and its foreseeable use is defensive. The transformation of Section 6.2 is a mitigation, the tasks are synthetic, and no component targets a deployed system. The remaining risk is that the benchmark is read as a certificate; passing a finite set of schedules is not a proof of invariance (Section 3).

## Reproducibility statement

The anonymous supplementary material contains the tasks and their hazards, the harness, both oracles and the analysis scripts, together with every stored generation (transcript, extracted SQL and served model id), every verdict, the analysis outputs the paper's numbers are read from, the run logs and the

graded SQL of every cell. A SHA-256 checksum covers each of its 34,039 files. It leaves out the mined third-party dbt repositories, the SQL embeddings of one probe and two adapter weight files. Every figure, table and number in the paper is generated from those outputs by a script. In a fresh copy of the release, the outputs behind the benchmark table, the instruction arms and the multiplicity family regenerate exactly (13,913 numbers in 32 files, pre-registration §6pp.3). Section S20 of the supplementary material lists what each result needs: the tables, both DuckDB oracles, the rewrites, the interventions and the delivery histories need only Python and DuckDB; the grain-test baseline needs dbt-core 1.10.23 with dbt-duckdb; the PostgreSQL results need a PostgreSQL server; and regenerating a pipeline needs the providers' APIs and cannot be exact for hosted models.

## References


Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11), 2013.

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12), 2015.

Apache Software Foundation. Apache Kafka documentation: Message delivery semantics. `https://kafka.apache.org/documentation/`, 2025a.

Apache Software Foundation. Apache Spark structured streaming programming guide. `https://spark.apache.org/docs/latest/`, 2025b.

Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic incremental view maintenance for rich query languages. *arXiv preprint arXiv:2203.16684*, 2022.

Peter Baile Chen, Devin Yang, Weiyue Li, Fabian Wenz, Yi Zhang, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. BEAVER: An enterprise benchmark for text-to-SQL. *arXiv preprint arXiv:2409.02038*, 2024.

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):1–27, 2018.

dbt Labs. ADE-Bench: A benchmark for analytics development engineering. Software repository, 2025.

Debezium Community. Debezium documentation: Frequently asked questions. `https://debezium.io/documentation/`, 2025.

Pat Helland and Dave Campbell. Building on quicksand. In *Conference on Innovative Data Systems Research (CIDR)*, 2009.

Joseph M. Hellerstein and Peter Alvaro. Keeping CALM: When distributed consistency is easy. *Communications of the ACM*, 63(9), 2020.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. DA-Code: Agent data science code generation benchmark for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024.

Yuchen Huang, Xiang Li, Zhenqing Ling, Sijia Li, Qianli Shen, Daoyuan Chen, Yi R. Fung, and Yaliang Li. CDR-Bench: Evaluating faithful execution of compositional, order-sensitive data refinement recipes. *arXiv preprint arXiv:2606.31435*, 2026.

Tengjun Jin, Yuxuan Zhu, and Daniel Kang. ELT-Bench: An end-to-end benchmark for evaluating AI agents on ELT pipelines. *arXiv preprint arXiv:2504.04808*, 2025.

Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proceedings of the VLDB Endowment*, 14(3):268–280, 2020.

Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton A. Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, Jun Chen, Markos Markakis, Matthew Russo, Weiyang Wang, Ziniu Wu, Michael J. Cafarella, Lei Cao, Samuel Madden, and Tim Kraska. KramaBench: A benchmark for AI systems on data-to-insight pipelines over data lakes. *arXiv preprint arXiv:2506.06541*, 2025.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *International Conference on Learning Representations (ICLR)*, 2025.

Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

Debin Meng, Jiaming Yang, Zefang Zong, Tengyue Xu, Haining Xie, Yang Li, and Peng Chen. DataClawEval: A benchmark for data engineering agents in real industrial harness. arXiv:2607.28033, 2026.

Quang Hieu Pham, Yang He, Ping Nie, Canwen Xu, Davood Rafiei, Yuepeng Wang, Xi Ye, and Jocelyn Qiaochu Chen. FlexSQL: Flexible exploration and execution make better text-to-SQL agents. arXiv:2605.02815, 2026.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems (SSS)*, 2011.

Snowflake AI Research. data-eng-bench: A data engineering benchmark for AI agents. Software repository and engineering blog post, 2026.

Zhaoyan Sun, Shan Zhong, Daizhou Wen, Jiaxing Han, Guoliang Li, Ying Yan, Peng Zhang, Yu Su, Xiang Qi, Baolin Sun, Chengyuan Yang, Tao Fang, and Huaiyu Ruan. AgenticDataBench: A comprehensive benchmark for data agents. *arXiv preprint arXiv:2607.01647*, 2026.

Gladys Tyen, Hassan Mansoor, Victor Cărbune, Peter Chen, and Tony Mak. LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics (ACL Findings)*, 2024. arXiv:2311.08516.

UC Berkeley EPIC Lab. DataAgentBench. Software repository, 2025.

Xiaoyun Xu, Lichao Wu, Jona te Lintelo, Siyu Zhang, and Stjepan Picek. SPARK: Security knowledge priming and representation-guided knowledge activation for LLM-based secure code generation. *arXiv preprint arXiv:2606.16244*, 2026.

Jingyu Zhou, Meng Xu, Alexander Shraer, et al. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, pp. 2653–2666, 2021.

## A Claims and their status

Table 5 lists each headline claim with where it is shown, its design, the unit its uncertainty is computed over, the section of the pre-registration that fixed its reading, and its status. *Exploratory* means the analysis was chosen after the outcomes were known; the algebra contrast and the scope rewrite are both of that kind. Appendices S1–S16 are in the supplementary material, which opens with a table of contents.

Table 5: Headline claims and their status.

| Claim | Where | Design | Unit | Registered | Status |
|---|---|---|---|---|---|
| Snapshot grading passes pipelines replay shows wrong | Tab. 2 | descriptive, 11 arms | task | §2.2 | confirmatory |
| Stating the property lowers certified-wrong; the ranking survives | Tab. 2, §5.1 | randomised between arms | task | §5, §6d, §6g, §6dd, §6ii | registered arms; Holm family set after outcomes |
| Sonnet 5 trades silent failure for crashes | Tab. 4 | randomised between arms | task | §6d, §6ii | post-hoc split; Holm-adjusted |
| The trade is same-batch duplicates | §6.1, Tab. S10 | intervention on the schedule | pipeline | §6kk, §6ll | registered test of a post-hoc idea |
| The algebra predicts which hazard class fails | §5.2, Fig. S2 | observational; labels rater-checked | task | — | exploratory |
| Uniqueness scope is a causal locus | Fig. 2 | within-program edit, two controls | pipeline | — | exploratory |
| The transformation $R$ repairs without a model | Tab. 6 | within-program edit | pipeline | — | exploratory |
| Snapshot repair neither helps nor harms replay | App. S15 | within arm and task | cell | — | questions fixed in its script |
| The fixed family suffices; the frontier does not separate | App. S14 | re-grading under new histories | cell | §6ff, §6mm | registered |
| A public model fails by the same mechanism | App. S16 | execution of one real model | schedule | §6gg | registered; one model |
| Grain tests on the key catch few silent failures | App. S13 | tests run on stored final tables | cell | — | exploratory |

## B The repair transformation

Table 6: The transformation $R$ applied to every stored pipeline on both engines, without a model call. Rates are over all graded pipelines; drops quoted in the text use unrounded rates.

| Engine | pipelines | all-in failure | | silent failure | |
|---|---|---|---|---|---|
| | | before | after | before | after |
| DuckDB | 2,179 | 51.7% | 25.1% | 35.1% | 14.8% |
| PostgreSQL | 2,188 | 56.4% | 32.6% | 31.2% | 13.1% |

**What $R$ does not guarantee.** $R(P)$ runs as statements that commit one at a time, with the update of the consumed set last. If a run is killed before that update and the script is retried, the retry re-executes the committed prefix on a state that already holds its effects, so $R(P)$ is replay invariant under that crash only if those writes are themselves idempotent. We state this as a prediction and test it: among rewritten pipelines whose crash prefix commits a write, 28 of 38 still fail crash-retry under the non-atomic variant, and 32 of 38 pass once the updates and the consumed-set insert commit together (Section 6.2). The atomic variant closes this gap but gives no ordering guarantee.