

STRUST-v1: A Small-Sample Evidence Audit and Unvalidated Verification-Slice Proposal

George Weale

Columbia University, 2025

george.weale@columbia.edu

Abstract—Two correct implementations can agree while neither reproduces its declared target observation. We evaluate this failure with executable event closure and a reduced, externally specified order-processing workflow. The new workflow follows NewOrder, OrderStatus and StockLevel semantics from TPC-C, without claiming benchmark compliance. SQLite and a separately written Python specification agree on all 1,920 declared full-history observations. Among 240 regularly selected targets, a filtered-stock-read ablation reproduces 236 contexts: four of 15 StockLevel targets give shared wrong-count matches. Instrumented examined-row keys restore all tested contexts; supplied predicate keys add no further correctness here. The full-history oracle already stores both unchanged implementations’ target outputs: a cached-oracle baseline answers all 240 questions with 1,920 events per implementation. Oracle-plus-slice replay costs 18,007 events even with last-writer keys. Its 10.7% reduction against naive repeated prefixes does not establish validation economy; cached outputs are sufficient and cheaper. A known-schema transfer study separately shows how density erases conditional replay savings and how deliberately omitted aggregate reads fail. An actual GnuCOBOL/Java seam tests observation channels, not the slice planner. The contribution is an inspectable audit of declared-observation fidelity and validation cost, not a new general slicing algorithm, TPC-C performance result or industrial COBOL migration system. Historical title and ledger are retained; the broader architecture remains unvalidated.

Index Terms—replay fidelity, observation contracts, database provenance, differential testing, validation cost

I. RESEARCH QUESTION AND SCOPE

Testing one transaction in isolation can compare two implementations under the same wrong state. Equality between the two runs does not establish that either represents the target business observation. This paper tests that distinction with executable replay, rather than treating a green isolated test as evidence of preserved context.

The evaluated mechanism is deliberately narrow: sequential event replay in a known-schema account-transfer study and an independently specified order-processing subset, each with a separately written state reference and a real SQLite implementation. The access schema is supplied, not extracted from COBOL or JCL. A separate GnuCOBOL/Java experiment evaluates which observation channels expose authored errors; it does not validate the slice planner. The broader verification architecture named in the retained title remains a proposal. Its historical ten-row ledger is summarized in Appendix A and is not evidence for the new mechanism.

We ask: **RQ1** Under which explicit assumptions does typed event closure preserve a target observation? **RQ2** How

do shared-state density and an omitted global read change context validity and replay work? **RQ3** In a separate actual-language seam, which state errors are invisible to receipt-only comparison? **RQ4** On externally specified transaction semantics, which observed-read model preserves targets, and how does oracle-validated slicing compare with cached answers and repeated prefixes? The answer to RQ2 is not simply that closure works: sparse known state permits substantial replay reduction, dense or aggregate state removes that saving, and incomplete keys can create a shared wrong-context match unless a separate context check intervenes.

Classical program slicing already studies dependencies and executable projections [9], [10], [11]. Executable slicing via procedure specialization addresses richer language and parameter-mismatch issues [14]; differential slicing studies causal execution differences [15]. Our event closure is simpler and does not claim to originate executable slicing or causal localization. The empirical contribution is a reproducible validation boundary for this supplied schema, including an explicit counterexample to agreement without context fidelity.

Transactional reenactment is nearer to our database setting than source slicing alone. Niu et al. replay transactions over the database states and concurrent interactions of their original executions, using temporal data and audit logs for debugging and provenance [16]; GProM supplies query/update/transaction provenance middleware [17]. We do not claim to originate history-based replay. Our smaller sequential experiment compares two supplied-key plans against each implementation’s complete-history target and abstains on context failure. It neither implements GProM nor replaces its concurrency/provenance machinery; the paired context gate and finite-key tradeoff are the scope of the present evaluation.

a) *Relation to why and why-not provenance.*: Lee et al. explain successful and failed query derivations, including existing and missing intermediate results, using SQL middleware [20]. Thus state absent from returned rows affecting a result is an established provenance problem. Our StockLevel witness does not discover that general phenomenon; it exposes its consequence for a particular supplied replay-key model. Niu et al. reconstruct original transactional environments for debugging and what-if questions [16]. We instead audit whether two selected histories preserve specified observations and whether obtaining the validation evidence already answers the same task.

TABLE I

DIFFERENT QUESTIONS REQUIRE DIFFERENT PRESERVED INFORMATION. THE CITED SYSTEMS ARE RELATED WORK, NOT EXECUTED COMPARATORS.

Approach	Preserved information	Question answered
Why/why-not provenance [20]	Successful/failed derivations and relevant existing/missing intermediates	Why is a query answer present or absent?
Transaction reenactment [16]	Original database versions and transaction interactions	What happened in the historical environment, including what-if debugging?
This supplied-key audit	Declared paired observations, logical read/write ledger, plans and acquired oracle answers	Does each slice preserve its own observation; which key ablation fails; does direct oracle reuse dominate validation cost?

II. KNOWN-SCHEMA EVENT AND OBSERVATION MODEL

Let e_i be a deterministic state transition with receipt r_i . For each implementation a , declare a target observation $q_t^a(s_t^a, r_t^a)$ of its post-target state and receipt. Here *context validity means equality of this declared observation with its full-history value*; it does not mean equality of the complete database or hidden runtime. A receipt-only rollback observation, for example, says nothing by itself about every database row after rollback.

For each transition, supplied keys K_i conservatively cover state read or changed by *either* implementation, including rejected paths. Additionally, O_t must cover every state read by *either target observation*, including projection-only reads and conservative aggregate/range dependencies. A comparison can require more state than the target transition touches. We represent the criterion as a read-only observation event after e_t . Its root has an edge to t and to the latest preceding touch of every key in O_t . Transition nodes link to preceding touches of their keys. Backward closure from the observation root, replayed in original order from identical initial state, is the sufficiently conservative plan. This is the ordinary dependence-criterion requirement, not a new slicing algorithm.

A. Sufficient assumptions and preservation argument

The guarantee requires deterministic sequential transitions and observations, identical initial state and event order, and conservative transition *and observation* keys for both implementations. Clocks, randomness, files and services must be pinned or modeled. Dynamic observed accesses do not certify this condition for untested contexts. A schema valid only for the reference, or only for transition reads, is insufficient.

For an included transition, repeated last-touch edges retain every previous touch to its relevant keys. Induction in event order then supplies identical relevant input state and receipt. The same argument applies to the explicit observation root: all prior touches of its covered read keys are retained, and its required target receipt is supplied by the mandatory target edge. Consequently q_t^a is preserved for each implementation separately. This does not establish whole-state preservation unless the declared observation and its conservative read criterion cover the whole state. Nor does the argument validate the supplied keys; the empirical ablations test their behavior under the observations actually declared.

B. Regression for an observation-only dependence

A separate two-event regression records the missing condition explicitly. Initially $A = B = 0$. Event 0 writes $B = 1$;

target event 1 writes only $A = 1$. Complete transition keys give the target-only plan [1], but a post-target observation of B is then zero rather than its full-history value one. Adding an observation root reading B gives $[0, 1]$ and restores that observation. Two identical deterministic implementations agree even in the incomplete plan; the per-implementation observation check abstains. These are formal regression cases, not additional benchmark samples. A new small helper implements the root construction; frozen benchmark runners and results are unchanged. Their tested contracts are enumerated below rather than extended retroactively to arbitrary projections.

C. Two separate gates

The differential verdict compares reference and candidate observations under the contract. The context gate compares *each replay's* target with that implementation's full-prefix observation. A differential match with failed context is INCLUSIVE, not MATCH. Unknown declared frontiers also abstain; blocked frontiers prohibit execution. Appending nonmatches to the cumulative ledger prevents a later match from erasing a previous failure. These are explicit policies, not discovery of unknown dependencies or proof of enterprise recomposition.

Full-prefix targets are validation oracles acquired in this research design. Their acquisition and storage are additional work. The replay event counts below compare selected replay with prefix replay conditional on those oracles already being available; they do not establish a net end-to-end speedup or a cheap production context gate. For the unchanged-pair transaction question below, the full oracle already answers every target directly. Other operational purposes would require a different task and oracle policy before an economy claim could be assessed.

III. EXECUTABLE REPLAY AND THE SPARSITY BOUNDARY

A. An implemented dependence-closure planner

The slice mechanism is evaluated separately using an integer specification and an actual SQLite transaction implementation. SQLite's explicit commit and rollback behavior is part of this local execution substrate [13]; it is not a CICS, DB2 or VSAM emulator. Each event reads or writes its source/destination account and transaction ID. A conservative temporal graph adds an edge to the last event touching each key, including failed reads. For a target observation, transitive closure selects events whose ordered replay begins from the same initial account state. This implements dependency selection and replay, not

a parser for arbitrary source code: the read/write schema is supplied and known.

Twenty new seeds are crossed with 4, 16 and 64 accounts, four subjects (correct, floor-fee, omitted-credit and rollback-leak), 96-event sequences, and eight target checkpoints. The target contract compares its receipt, touched balances and journal entries for its ID. Three plans execute from fresh state: the complete prefix, transitive dependency closure, and the target event alone. Before crediting a slice verdict, both reference and candidate target observations are checked against that implementation’s full-prefix observation. This checks context preservation separately from whether the candidate agrees with the reference.

Every seventeenth target under a fixed schedule carries an explicitly unknown external frontier. The planner returns INCONCLUSIVE and executes no slice for such a target. This tests a declared frontier policy; it does not test discovery of unknown dependencies. Stage recomposition appends prior divergences and unresolved frontiers to a cumulative ledger, and later matches cannot clear them. The artifact stores each plan result and cumulative record.

B. Replay correctness and sparsity boundary

The three methods yield 5,760 target evaluations: 5,412 complete plans and 348 unknown-frontier abstentions. Prefix and closure reproduce both implementations’ original context in all 1,804 complete targets each. Target-only replay reproduces context in 0/600 targets with four accounts, 18/600 with 16 accounts, and 98/604 with 64. Its raw, ungated differential comparison agrees with the full-prefix target in 57.5%, 70.2% and 87.3%, respectively. Failed context receives an INCONCLUSIVE credited verdict regardless of that raw agreement. An apparently plausible isolated test therefore can both misrepresent its runtime context and give a misleading comparison result.

Figure 1 shows a deterministic cost boundary. At four accounts, closure retains 52.59 of 53.68 mean prefix events; at 16 it retains 43.84 of 53.84; at 64 it retains 14.76 of 54.12, a 72.7% reduction. Dense shared state eliminates most slicing benefit. Exploratory mean wall time is 5.10 versus 5.22 ms at four accounts and 2.12 versus 8.86 ms at 64. These timings include planning and paired SQLite replay, but use fixed method order on one shared host. They are not a production throughput estimate or a causal speedup measurement. Event counts provide the firmer evidence.

These results establish an executable bounded mechanism under the supplied schema. They do not establish correct dependence extraction from legacy programs, dialect/encoding fidelity, unmodeled external effects or savings on an industrial dependency graph. The language seam and SQLite slice planner are separate experiments; the planner does not yet automatically extract and replay slices from the COBOL worker. That integration remains necessary before claiming a COBOL modernization verification system.

IV. WHEN A KEY MODEL OMITTS AN AGGREGATE READ

A separately recorded mechanism protocol adds a global balance rule to 20 two-event sequences over 16 accounts. The

TABLE II
DELIBERATELY TRIGGERED OMITTED-GLOBAL-READ CHECKS. BOTH IMPLEMENTATIONS ARE CORRECT; CONTEXT VALIDITY IS THE TESTED OUTCOME.

Model	Mean events	Context valid	Gated match
Account/ID only	1	0/20	0/20
Aggregate key	2	20/20	20/20
Complete prefix	2	20/20	20/20

prior event transfers between two accounts and deducts a positive fee. The target touches two *different* accounts but reads the sum of all balances before allowing its transfer. The threshold is fixed strictly between the initial and post-fee totals, so this design intentionally exercises the omitted read. It is a counterexample family, not an estimate of how frequently production schemas are incomplete.

The account/ID key model selects only the target. In both the integer reference and actual SQLite implementation, replay from initial state then permits a transfer that the full prefix rejects with `global_limit`. The implementations still agree with one another: a blind differential comparison returns MATCH in all 20 wrong contexts. The context gate compares both projections with their respective full-prefix targets and makes every case INCONCLUSIVE. A conservative aggregate-balance key connects the potential total-changing event and the global read, restoring the full target on all 20 cases while retaining both events.

This check supplies the model-boundary result missing from a closure-only demonstration. Dependency closure cannot compensate for an omitted read; agreement can fail open because the same context is missing on both sides. A sufficiently conservative aggregate model restores correctness but couples otherwise disjoint accounts. Together with the density experiment, this shows why replay reduction depends on the precision and completeness of modeled state, not simply on taking a smaller execution unit.

V. EXTERNALLY SPECIFIED ORDER-PROCESSING WORKFLOW

A. Frozen subset and separately implemented reference

The transfer generators establish mechanisms but do not test a practical key-model decision. We therefore freeze a new protocol before inspecting its full-history or slice outcomes, using TPC-C 5.11.0 clauses 2.4.2, 2.6.2 and 2.8.2 and the published py-tpcc SQLite query shapes [18], [19]. The supplier sources are pinned and hashed; their repository build code is not executed. An internal pre-outcome source audit by the parent agent in the same research process checks the declared subset. This is not an external-team or blinded validation. Our owned SQLite implementation and separately written Python state transitions implement the subset, rather than importing the original driver, whose invalid-item path contains an unfinished rollback check.

NewOrder allocates a district order ID, inserts the order/new-order/line records, updates each stock quantity using the specified subtract-or-add-91 rule, and updates year-to-date

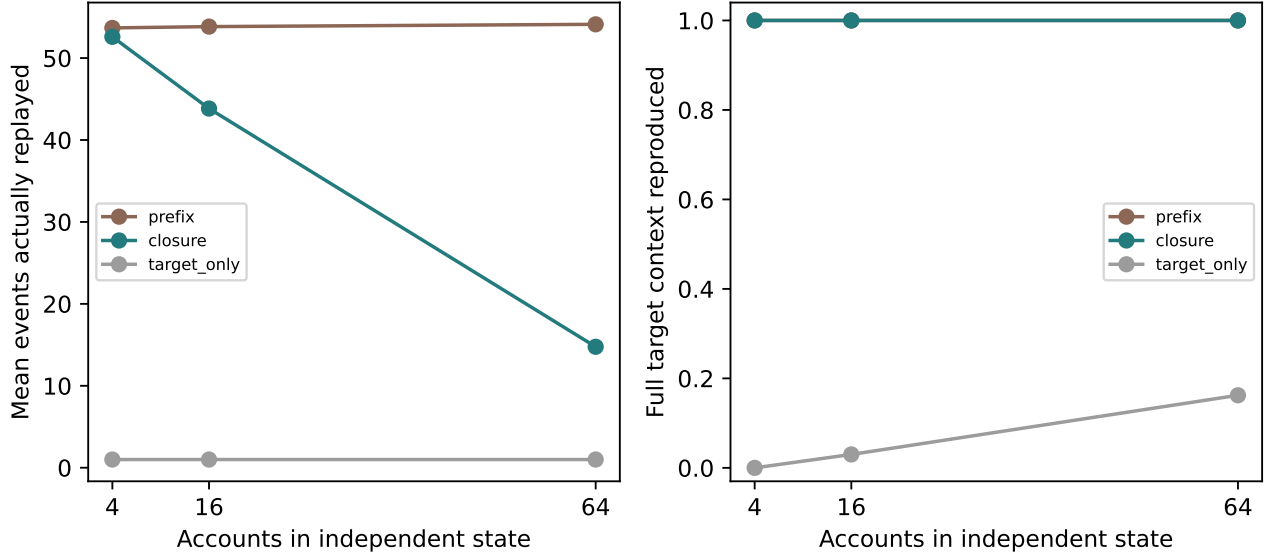


Fig. 1. Implemented dependency-closed replay versus complete prefixes and context-free targets. The known account/ID dependence model preserves target behavior in all complete closure cases. Fewer shared accounts make closure dense and remove most event-count savings. Unknown frontiers abstain.

quantity and order count. An invalid final item causes complete rollback after earlier valid line writes. OrderStatus by customer ID returns the latest order and its lines. StockLevel counts distinct items with home-warehouse stock below a random threshold among the district’s last 20 orders. This last predicate naturally reads quantities absent from its low-stock result set.

There is one warehouse, four districts, 20 customers per district and 100 items. Each customer starts with one five-line order. Twelve frozen seeds generate 160 events each, using normalized NewOrder:OrderStatus:StockLevel weights 45:4:4. Districts/customers/items are uniform; NewOrder has 5–15 unique items, quantities 1–10 and a 1% invalid-final-item probability; StockLevel thresholds are uniform 10–20 and independent of observed stock. Every eighth event is a target. These choices are not tuned to slice outcomes. We omit Payment/Delivery, remote supply, surname lookup, NURand, monetary tax/discount totals, display fields, benchmark scale and concurrency. This is an order-processing semantic subset, *not a compliant TPC-C benchmark*, and no throughput or TPC score is reported.

a) *Declared observation schemas.*: NewOrder compares its allocated order ID, next district ID, line tuples (item, quantity, amount), and requested stock tuples (item, quantity, year-to-date quantity, order count). OrderStatus compares the latest order ID and its line tuples. StockLevel compares its distinct low-stock item count. Rollback compares only the ROLLBACK receipt. These outputs arise from the supplied transition/query evaluations; no general post-target database snapshot is compared in this transaction study. The transfer study separately compares its receipt, touched account balances and transaction-ID journal entries, whose reads are covered by its account/ID criterion. Observation reads in a wider contract would require additional roots or keys before any preservation guarantee. The actual-language seam has its own account/journal/ receipt contract and does not use this planner.

B. Actual logical access ledger and key models

SQLite functions in executed row-value expressions log logical reads; update and insert triggers log writes. A materialized StockLevel query stage records every examined stock quantity before the quantity filter, including excluded rows. These are actual instrumented SQL evaluations, not a physical database access oracle. Query text/parameters, returned observations and separate read, write and predicate sets are retained for every full-history event. Attempted writes from rolled-back events remain conservative dependencies.

The *filtered-stock-read ablation* (stored as `result_rows` in frozen files) retains every write. For StockLevel it retains all non-stock reads, including the district counter and every relevant order-line read, and drops only stock keys whose quantities fail the low-stock filter. Successful NewOrder and OrderStatus retain their instrumented reads and writes; rollback retains attempted writes under its receipt-only contract. This is not a generic extractor of output-row provenance from a returned count, and no such extractor is evaluated. The *examined-row* model includes all instrumented reads and writes. A *predicate-aware* model additionally links order insertions to supplied district line-range and customer latest-order predicates, covering changed or absent membership. These are manually supplied query-schema keys, not automatically inferred SQL dependencies. Dynamic observed reads alone are not certified complete for other execution contexts.

We evaluate the existing last-touch closure and a second, pre-outcome source-audit diagnostic: last-writer closure. In the latter, reads/writes depend on the preceding writer, while read/read accesses do not connect events. Identical workloads and targets are used for all models. Each selected plan runs in order from fresh initial state in both implementations. A target is credited only when both results equal their own full-history targets; otherwise the shared raw match remains diagnostic

and the verdict abstains.

C. Natural omitted reads and null predicate benefit

All 1,920 full-history observations agree across implementations: 1,622 NewOrder, 135 OrderStatus and 163 StockLevel events, including 24 rollbacks. Regular target selection yields 207 NewOrder, 18 OrderStatus and 15 StockLevel targets. Both filtered-stock-read planners reproduce 236/240 contexts. Their four failures are all StockLevel: both correct implementations return the same wrong count, so blind comparison matches while the context gate abstains. Post-hoc disaggregation of the retained output pairs confirms 236/240 reference-observation fidelity and 236/240 SQLite-observation fidelity for each filtered-stock-read planner; their 240 paired matches hide the same four failures. The affected histories have seeds 91001, 91007 and 91010. Both examined-row planners and both predicate-aware planners reproduce 240/240 contexts. Predicate keys therefore add no correctness benefit on these targets; we retain that null result rather than infer that predicates are generally unnecessary.

An outcome-aware explanation, outside any new test denominator, traces each failure to omitted history and retained SQL reads. For seed 91001, target 135, threshold 16, item 52 has quantity 100 in the full history but 12 in the slice. Its last writer, event 132, was omitted because the full query examined that item but excluded it from its low-stock results. The full count is three; both slice implementations return four. This is an ordinary generated predicate case, unlike the deliberately triggered global-sum counterexample. It demonstrates that dropping filter-excluded stock reads can break replay fidelity even when other query reads are retained. This is a counterexample to treating returned stock rows as sufficient, not an evaluation of a general output-provenance extractor or a failure-prevalence estimate. Targets within each of the twelve histories are dependent; 240/240 is not a population guarantee.

a) *Why the supplied predicate tags add no writer-plan benefit here.:* NewOrder reads and increments its district allocation counter. StockLevel reads that same counter before its last-20-order query. The last-writer chain therefore retains every preceding successful same-district NewOrder for a StockLevel target; last-touch also follows that chain, with possible extra read/read links. OrderStatus reads the returned order row and lines. If that order was created in the workload, those keys lead to its creating NewOrder and then to the successful same-district NewOrders through that creator. They do not require every later same-district insertion. An initial order has no creating event in the workload.

A read-only inspection of the frozen ledger/plans confirms those scoped retention statements for all 15 StockLevel and 18 OrderStatus targets under all six key/planner combinations; nine OrderStatus targets return newly created orders and nine return initial orders. Examined and predicate writer plans are identical, not merely equal in length, for all 240 targets. The supplied range/latest-order tags are thus redundant with existing retained paths for these observations; last-touch tags can still add read/read over-retention, as their higher event count shows. This workload does not isolate changing membership without

TABLE III
240 UNCHANGED-PAIR TARGETS PER ARM. TOTAL EVENT COUNTS ARE PER IMPLEMENTATION ACROSS ALL TWELVE WORKFLOWS; SLICES INCLUDE ORACLE ACQUISITION. FOR STOCKLEVEL, FILTERED STOCK RETAINS ALL NON-STOCK READS AND WRITES, DROPPING ONLY FILTER-EXCLUDED STOCK READS. CACHED ANSWERS PREDATE SLICE REPLAY.

Keys	Planner	Valid	Events	/Cached
Filtered stock	Touch	236	20,188	10.515
Examined	Touch	240	20,287	10.566
Predicate	Touch	240	20,561	10.709
Filtered stock	Writer	236	17,968	9.358
Examined	Writer	240	18,007	9.379
Predicate	Writer	240	18,007	9.379
Complete	Repeated prefix	240	20,160	10.500
Complete	Cached oracle	240	1,920	1.000

counter-chain protection. The result does not establish that such tags are unnecessary when those paths are absent, under deletions, or in another query schema. The post-hoc inspection executes no transaction and changes no key or benchmark outcome.

D. Cached answers defeat a replay-economy interpretation

For targets $t_1, \dots, t_K$, naive repeated-prefix execution costs $B_K = \sum_j (t_j + 1)$ events per implementation. Oracle-plus-slice replay costs $C_K = H + \sum_j |S_j|$ with $H = 160$ per workflow. At twenty targets, last-writer examined/predicate models cost 18,007 events across workflows versus 20,160 naive repeated-prefix events: the retained 10.7% relative reduction and fourteen-target pooled crossover describe only that inefficient comparator. Last-touch models cost 20,287/20,561 events and do not even improve on it.

The necessary baseline is to cache the oracle's existing answers. Collection already executes both unchanged implementations and retains the full-history observations used to validate every slice. Those cached outputs answer all 240 same-pair target questions directly, without any additional replay. Their acquisition costs only $12 \times 160 = 1,920$ events per implementation. The post-hoc baseline analysis verifies every cached target against the retained paired full-history output; it executes no transaction and changes no context/key result. Correct last-writer slicing therefore costs 9.38 times the sufficient cached-oracle baseline. The study demonstrates *no operational or net validation-efficiency benefit* for its unchanged pair.

The workload remains informative about context preparation and isolation: last-writer examined keys retain 70.48 versus 83.29 repeated-prefix events for NewOrder, 24.39 versus 92.89 for OrderStatus, and 70.53 versus 83.20 for StockLevel. These are conditional replay sizes, not validation savings. Fault isolation, changed implementations or unavailable full observations could motivate other slice uses, but their benefits are not demonstrated here. Figure 2 places the cached baseline beside all replay costs. Planning/setup/instrumentation timings and storage costs add further work; no wall-time or production efficiency claim follows.

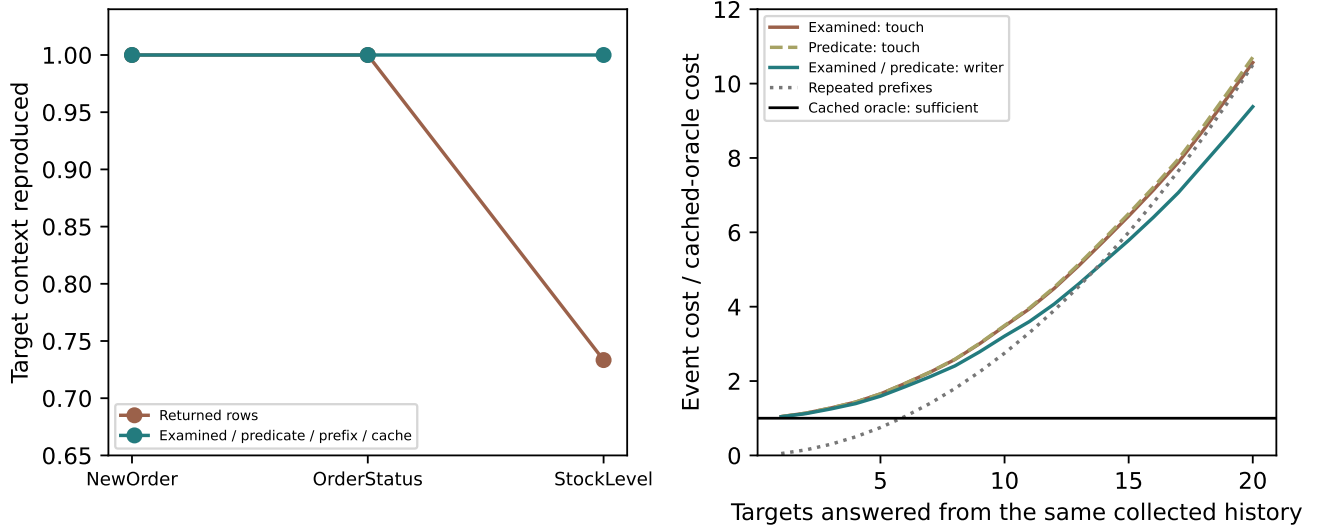


Fig. 2. Same retained context results and corrected cost baselines. Left: The plotted returned-row label denotes the filtered-stock-read ablation: all non-stock reads/writes remain. Four StockLevel targets fail; examined stock reads restore them. Right: the full-history cache already answers every unchanged-pair target and requires no slice replay. Lower cost relative to naive repeated prefixes does not imply validation economy.

VI. SEPARATE ACTUAL-LANGUAGE OBSERVATION STUDY

A. New COBOL/Java transaction seam

This new study uses actual GnuCOBOL 3.2 and Temurin Java 17, not a Python substitute for either language. Both workers maintain four account balances and a committed-transaction journal across 24-event sequences. The contract specifies integer cents, half-up fees, duplicate IDs, missing destinations, insufficient funds, same-account transfers and rejection without state change. The reference is also checked against a separately written integer-state specification. Source, compiler identities, exact commands, input sequences, raw standard streams and output hashes are retained in the language manifest. These executions took place in the new revision cycle, not the original project year or the date of the ten-case ledger reconstruction.

Eighty workflows contain an authored boundary prefix followed by seeded transactions. The prefix intentionally triggers the seven mutation classes; the observed detection rates therefore are mechanism checks, not unbiased fault recall. The COBOL reference agrees with the integer specification at all 1,920 checkpoints. Correct and equivalent reverse-update Java modes match the reference in all 80 workflows. Full state and receipts detect floor-fee, round-to-even, omitted-credit, missing-journal, rollback-leak, stale-receipt and duplicate-reapplication candidates in every workflow. Receipt-only comparison misses both rounding variants completely; account-only comparison misses stale receipts. Figure 3 shows why observation scope matters. Checkpoints add no detection beyond final full state plus receipts on this deliberately small corpus; we retain this null result.

B. A reusable fidelity-and-cost audit

The transferable result is a task definition and falsifiable evaluation sequence for supplied-key replay, rather than a new

dependence extractor. First declare the observation and its read criterion before choosing a slice. Report each implementation’s fidelity to its own full-history observation separately from cross-implementation agreement; an observation failure cannot receive a credited match. Then ablate filter-excluded stock reads while holding non-stock reads and writes fixed on identical inputs, retaining dependent history units and inconclusive outcomes. Finally identify what the acquired oracle already answers, and compare direct cached answers before claiming validation economy. These requirements are grounded here by the observation-root regression, four natural StockLevel failures in three of twelve histories, the null extra-predicate effect, and the cache comparator that defeats the apparent repeated-prefix saving.

A researcher applying this audit can reject the filtered-stock-read ablation for StockLevel under the declared count contract, distinguish counter-chain redundancy from evidence about unprotected membership, and decline to deploy oracle-validated slices as a cheaper way to answer these unchanged-pair queries. That is more specific than a general recommendation to log more state. Diagnostic isolation or altered-execution uses remain possible, but need their own task, observation and oracle-cost definition. None is a demonstrated operational benefit of this experiment. Known provenance and replay techniques supply the concepts; this bounded failure/cost audit tests their interaction under explicit comparison decisions.

VII. INTERPRETATION, LIMITS AND REPRODUCIBILITY

The planner and language experiments answer different questions. SQLite evaluates known-schema event selection and context; the actual COBOL/Java workers evaluate observation contracts. No operation here extracts the SQLite graph from COBOL source or automatically replays COBOL slices. Language feasibility must not be inferred as slice-planner efficacy, or conversely.

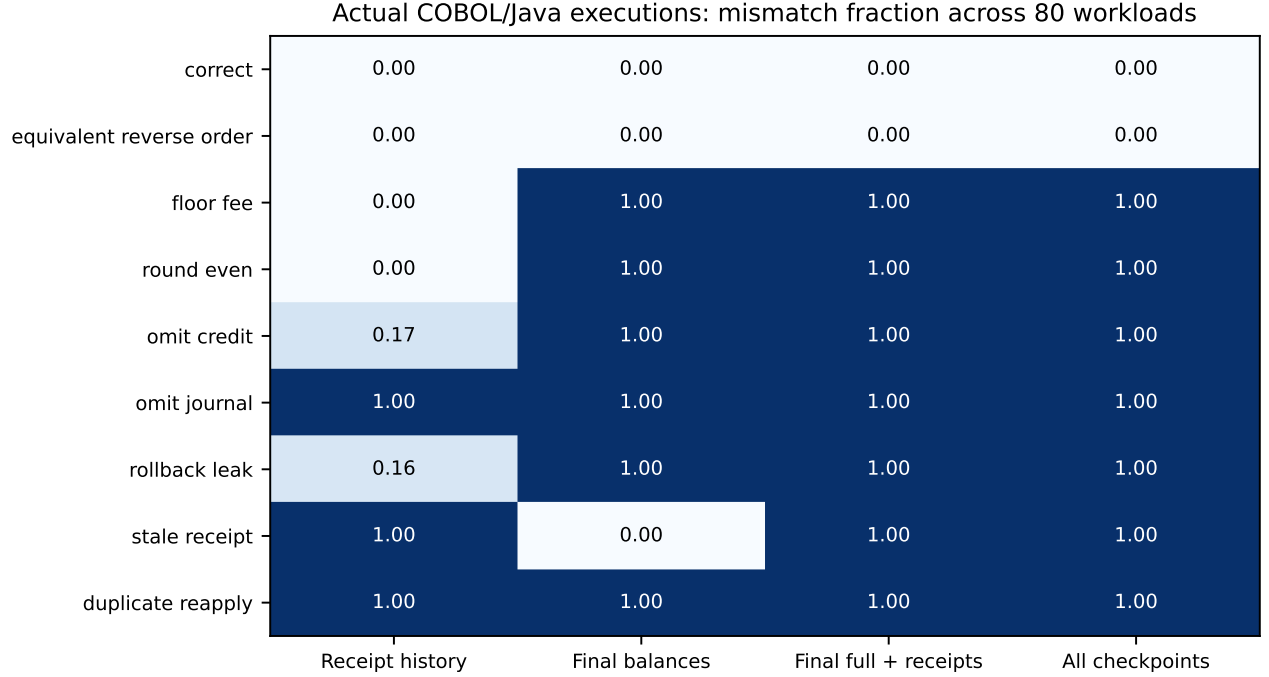


Fig. 3. New executable COBOL/Java observation-contract study. Rates concern 80 deliberately fault-triggering authored workflows for each Java mode; they do not estimate accuracy on unseen migration candidates. The historical ten-row ledger is separate.

The preservation argument applies only to the supplied deterministic key model. The known-schema seeds, twelve external-subset workflows, dependent targets and deliberately authored faults do not represent industrial migrations. Unknown-frontier abstention is an assigned policy input; it is not evidence that unknown dependencies were discovered. Cumulative ledger retention is tested as a policy invariant, not as full business-flow recomposition. The aggregate experiment identifies a specific assumption failure, not a new general source-analysis method.

Event counts give a stronger cost result than the exploratory wall times: dense state retains almost the prefix, sparse local state permits much less replay, and coarse global dependencies can remove the saving. Method order and ambient host load are uncontrolled. The corrected external-subset analysis charges oracle acquisition and compares its cached answers directly: slicing cannot reduce validation cost for this unchanged pair once all target outputs are available. Planning, setup, instrumentation and storage costs do not rescue that conclusion.

The reference can itself be wrong. A modernization may intentionally repair an old behavior; approved behavioral changes need an explicit exception rather than quiet normalization [12]. Fixed decimal scale, rounding, encoding, error behavior and persistent state remain contract decisions, not properties established by compilation alone. GnuCOBOL is not arbitrary Enterprise COBOL or z/OS subsystem compatibility.

Protocols, selected event indices, paired raw states, full-prefix targets, source/runtime identities, commands and hashes are retained. The new execution cycle is recorded in manifests; it does not recover the historical demonstrator or inherit a project-origin date. Final human scientific approval remains pending.

VIII. CONCLUSION

Closure rooted in a declared observation preserves that observation under the stated conservative transition- and observation-key assumptions, while incomplete keys can make two correct programs agree in the wrong replay state. Four naturally generated StockLevel targets expose that failure under the filtered-stock-read ablation; examined stock reads restore their contexts. Extra predicate keys add no correctness on these targets. The collected oracle already answers every unchanged-pair question with 1,920 events per implementation, versus 18,007 for correct last-writer oracle-plus-slice replay. The latter's 10.7% advantage over naive repeated prefixes is not an operational or net validation-efficiency benefit. The authored density and aggregate checks explain complementary boundaries. The separate language seam shows why receipts alone omit relevant observations, without validating an integrated COBOL slicing system. The historical ledger and enterprise proposal remain distinct from this bounded executable evidence.

APPENDIX A

HISTORICAL LEDGER AND BROADER PROPOSAL

The supplied v1 demonstrator preserves ten final rows, not its original subjects, paired traces or environment manifests. Its reconstructed ledger has $TP = 3$, $FP = 1$, $FN = 1$, $TN = 5$, accuracy 8/10 and mismatch F1 0.75. The Wilson interval is [0.49, 0.94]; the one-sided exact tail versus 0.5 is 0.0547 and the paired comparison with always-match is $p = 0.625$. These post-hoc small-sample calculations neither validate the architecture nor measure the new replay mechanism. The earlier architecture proposal and complete row-level audit are retained separately in the artifact.

The complete ten-row reconstruction remains in `historical/paper-before-r2.tex`, including each scenario and assigned verdict. Its two errors concern rounding and encoding. This historical table does not establish the new replay or language-study results.

REFERENCES

- [1] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
- [2] H. Zhu, P. A. V. Hall, and J. H. R. May, “Software unit test coverage and adequacy,” *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.
- [3] S. Rapps and E. J. Weyuker, “Selecting software test data using data flow information,” *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367–375, 1985.
- [4] T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.
- [5] IBM, “Controlling how numeric data is stored,” *Enterprise COBOL for z/OS Documentation*, 2024.
- [6] GnuCOBOL Project, “GnuCOBOL Programmer’s Guide,” ver. 3.2, 2022.
- [7] Oracle, “Class BigDecimal (Java SE 11 & JDK 11),” 2024.
- [8] Docker, Inc., “Docker documentation,” 2024.
- [9] M. Weiser, “Program slicing,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.
- [10] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.
- [11] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.
- [12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.
- [13] SQLite Project, “Transaction,” official documentation, accessed Oct. 20, 2026. https://www.sqlite.org/lang_transaction.html
- [14] M. Aung, S. Horwitz, R. Joiner, and T. Reps, “Executable slicing via procedure specialization,” University of Wisconsin, Technical Report 1711. <https://research.cs.wisc.edu/wpis/abstracts/tr1711.abs.html>
- [15] N. Johnson et al., “Differential slicing: Identifying causal execution differences for security applications,” in *IEEE Symposium on Security and Privacy*, 2011.
- [16] X. Niu et al., “Debugging transactions and tracking their provenance with reenactment,” *PVLDB*, vol. 10, no. 12, 2017. <https://arxiv.org/abs/1707.09930>
- [17] UIC Database Group, “GProM,” primary project documentation, accessed Oct. 20, 2026. <https://www.cs.uic.edu/~bglavic/dbgroup/projects/gprom.html>
- [18] Transaction Processing Performance Council, “TPC Benchmark C, standard specification, revision 5.11.0,” clauses 2.4.2, 2.6.2, 2.8.2. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf
- [19] A. Pavlo et al., “py-tpcc SQLite driver,” pinned commit db36d72dfbb6bd800d257279be9bbc1a22095ff9, accessed Oct. 20, 2026. <https://github.com/apavlo/py-tpcc/blob/db36d72dfbb6bd800d257279be9bbc1a22095ff9/pytpcc/drivers/sqlitedriver.py>
- [20] S. Lee, S. Köhler, B. Ludäscher, and B. Glavic, “A SQL-middleware unifying why and why-not provenance for first-order queries,” primary author manuscript, accessed Oct. 20, 2026. <https://www.cs.uic.edu/~bglavic/dbgroup/assets/pdfpubs/LS17.pdf>