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

George Weale
Columbia University
New York, NY, USA
george.weale@columbia.edu

***Abstract*—Modernizing COBOL to Java requires preserving observable behavior, not merely compilation. This paper audits the only preserved evaluation evidence for STRUST-v1: a ten-row historical verdict ledger. Executable subjects, raw traces, and environment digests are unavailable. We ask whether the reconstructed ledger provides evidence that the demonstrator outperforms chance and a trivial majority-class comparator. V1 classifies 8/10 cases correctly (Wilson 95% interval 0.49–0.94) with mismatch F1 0.75. The exact one-sided tail probability against 0.5 is 0.0547, and the paired exact comparison with the 60%-accurate majority baseline is $p = 0.625$; neither crosses $\alpha = 0.05$. Leave-one-case-out accuracy ranges from 0.778 to 0.889. Thus the ledger does not establish comparative effectiveness. The two errors expose decimal-tolerance and EBCDIC-normalization defects. A contract-driven verification-slice architecture is retained only as unvalidated design context; this ledger supplies no evidence about dependency discovery, recomposition, scalability, or production applicability.**

***Index Terms*—legacy modernization, COBOL, Java, differential testing, program slicing, interprocedural analysis, call graph, test oracle**

I. INTRODUCTION

COBOL-to-Java modernization is a semantic preservation problem. A conversion can compile cleanly, satisfy a style checker, and still be operationally unsafe if it changes a rounding boundary, a field-padding rule, a collation order, an error path, or the state written to a downstream system. These hazards are unsurprising: COBOL data descriptions and operational environments encode behavior that is often implicit in a Java translation. In particular, fixed-point decimal fields, packed-decimal representations, EBCDIC text, record-oriented I/O, and dialect-specific runtime behavior require an explicit compatibility decision rather than a syntactic rewrite [5], [6].

Traditional modernization assurance combines unit tests, integration tests, code review, and vendor-specific checks. Those practices remain necessary, but none independently answers a narrow and important question: *given the same approved workload and declared environment, did the candidate preserve the reference program's observable behavior?* Differential testing provides a useful foundation because independently executed implementations can act as mutual test oracles [1]. Its value, however, depends on disciplined control of inputs, environments, and comparison semantics.

This work audits a historical STRUST (*semantic trust*) ledger and records its proposed contract-scoped differential-verification model as design context. The audit does not establish that the proposed implementation existed in the form described below or that it localizes disagreement, preserves evidence, or scales beyond the ten supplied verdict rows.

The paper makes three bounded contributions:

1) an auditable reconstruction of the supplied ten-case ledger, with corrected diagnostic metrics and preserved errors;
2) exact null comparisons, leave-one-case-out sensitivity, and explicit negative evidentiary conclusions; and
3) a separation between what the ledger supports and an unvalidated verification-slice proposal whose claims are stated as future hypotheses.

The empirical question is deliberately narrower than the architecture: **RQ1:** is 8/10 ledger accuracy distinguishable from a declared 0.5 chance comparator? **RQ2:** does v1 outperform an always-MATCH majority-class rule on these same ten paired cases? Because the audit was added after outcomes were available, exact p-values are descriptive rather than preregistered confirmation. A result above 0.05 is reported as a negative result, not converted into an architectural success claim.

II. PROBLEM MODEL AND DESIGN PRINCIPLES

A. Observable Equivalence Is Contractual

The proposed model requires a future paired execution to capture a trace bundle: exit classification, standard streams, declared output files, declared persistent-state deltas, timeout state, and run provenance. No such trace bundles are preserved in this artifact. The intended observation scope is broader than console output because a program that prints the expected line but writes the wrong record layout, database row, or return condition has not preserved the business behavior that matters.

A *test contract* fixes the adapters, canonicalization policy, comparator, resource limits, and observation scope. Exact-byte comparison is appropriate for an approved binary artifact; it is not a defensible default when one side emits EBCDIC and the other UTF-8. Conversely, unconstrained whitespace, timestamp, or numeric "normalization" can hide a real defect. Every transformation must therefore be declared, versioned, and attributable to a compatibility decision.

B. Three Verdicts, Not a Forced Boolean

The proposed verdict algebra classifies each fixture as MATCH, DIVERGE, or INCONCLUSIVE. A MATCH would mean

that all required observations compare equal after authorized normalization. A DIVERGE would mean that both executions reached a comparable terminal state and at least one required observation differs. An INCONCLUSIVE result would represent a timeout, compilation failure, missing adapter, unapproved dependency, or incompatible environmental precondition. The ledger records only final verdicts and cannot verify that the historical demonstrator implemented these distinctions correctly.

C. Evidence, Not a Magic Number

A future implementation should report agreement rate, divergence rate, inconclusive rate, covered business observations, contract identity, and dependency frontier. The present ledger permits only case counts and derived classification metrics; it contains no coverage, contract-identity, or dependency-frontier evidence. High agreement would remain evidence for an enumerated workload, not proof that every untested path is equivalent [2], [3].

III. PROPOSED SYSTEM ARCHITECTURE (UNVALIDATED)

Figure 1 shows the proposed architecture. No preserved implementation or execution evidence validates this design. It is included to make future hypotheses concrete. The proposal centers on the contract manifest rather than the containers: container isolation would be useful only when runtime identity, fixture materialization, and comparison policy are reproducible.

A. Contract Manifest

The contract is a version-controlled manifest with six required sections:

1) *Identity*: SHA-256 digests for source, copybooks, build files, images, and comparator plug-ins.
2) *Execution*: compiler/JDK versions, dialect flags, locale, timezone, environment variables, CPU and wall-clock limits, and network policy.
3) *Fixtures*: immutable input files, command-line arguments, file allocations, and declared initial state.
4) *Adapters*: mappings for file paths, encodings, external calls, and simulated subsystems.
5) *Observation*: streams, files, return conditions, state snapshots, and any interaction trace that must be compared.
6) *Comparison*: canonicalization transformations, schema-aware comparators, tolerances, and forbidden normalizations.

This structure makes the usual hidden assumptions inspectable. For example, a contract can state that a DISPLAY field is transcoded from EBCDIC to UTF-8 before textual comparison, while a COMP-3 field is decoded by a field-aware adapter rather than treated as text. IBM documents packed decimal as a storage representation distinct from DISPLAY and NATIONAL data; a generic character transcoder is therefore not an adequate comparator for such data [5].

Algorithm 1 Contract-driven differential verification

Require: Contract Π, fixture x, reference P_C, candidate P_J

$\Pi \leftarrow$ VALIDATEANDDIGEST(Π)
$(e_C, e_J) \leftarrow$ BUILDPINNEDCAPSULES(Π)
$(o_C, o_J) \leftarrow$ EXECUTEINPARALLEL(P_C, P_J, x, e_C, e_J)
if NOTCOMPARABLE(o_C, o_J, Π) **then**
    return (INCONCLUSIVE, PERSISTRAWEVIDENCE())
end if
$(\hat{o}_C, \hat{o}_J) \leftarrow (N_\Pi(o_C), N_\Pi(o_J))$
$d \leftarrow C_\Pi(\hat{o}_C, \hat{o}_J)$
$v \leftarrow$ **if** d **then** MATCH **else** DIVERGE
return (v, PERSISTEVIDENCE($\Pi, o_C, o_J, \hat{o}_C, \hat{o}_J$))

B. Execution Capsules and Trace Capture

The design assumes separate, pinned container images for the COBOL and Java toolchains, but no historical image or manifest is preserved. A proposed COBOL capsule could use GnuCOBOL for portable cases [6]; a field study would need the actual compiler, dialect options, and runtime layer. A proposed Java capsule would pin the JDK, dependency lockfile, and JVM options, with input isolation between capsules. These are requirements, not observed properties of v1.

Each run emits a content-addressed trace bundle. At minimum, it contains the manifest digest, source and image digests, invocation, exit class, captured streams, declared output artifacts, timing metadata, and comparator version. The collector writes raw evidence before normalization. This ordering is important: a later change to canonicalization policy must not destroy the ability to inspect what each program actually produced.

C. Canonicalization and Comparison

Canonicalization is a compatibility boundary, not a cosmetic clean-up step. The proposal distinguishes:

- *representation normalization*, e.g., CRLF-to-LF conversion or a declared EBCDIC code-page transcode;
- *schema normalization*, e.g., decoding a fixed-length record into named fields before comparison; and
- *semantic comparison*, e.g., an approved decimal scale and rounding mode, or order-insensitive comparison of an explicitly unordered collection.

The engine applies the narrowest comparator consistent with the contract: byte comparison for invariant bytes, record-aware comparison for mainframe files, structured comparison for JSON/XML/CSV, and state comparison for declared persistence adapters. A transformation that masks a business-visible difference is rejected as a contract violation. Timestamp stripping, for example, is permitted only for a named field that has been classified as non-semantic.

IV. PROPOSED SCALING DESIGN (UNVALIDATED)

A. The Verification Slice

The proposed scaling hypothesis uses a middle abstraction, the *verification slice*, between disconnected snippets and an opaque whole-estate test. A slice is defined as the smallest

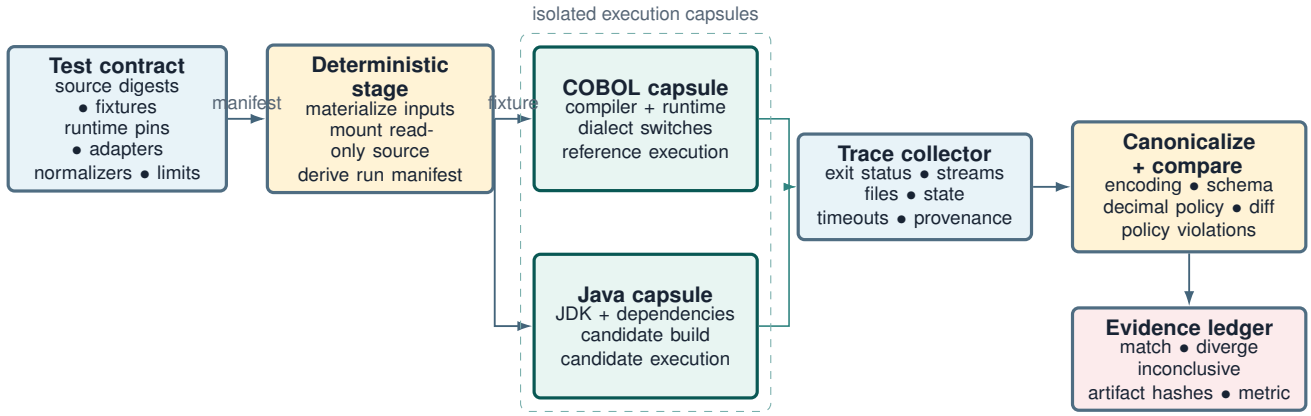

Fig. 1. Unvalidated STRUST architecture proposal. The diagram states intended responsibilities; no preserved implementation or execution artifact demonstrates them.

TABLE I
PROPOSED STAGED VERIFICATION LADDER. REUSE AND MONOTONIC EVIDENCE ACROSS STAGES ARE NOT EVALUATED IN V1.

Stage	Verification unit	Primary evidence
Record	Named field or record layout	Raw bytes, encoding, scale, sign, schema.
Program	One compilation unit and its paragraphs	Return condition, files, streams, selected storage.
Chain	Context-specific program/call path	Parameter mapping, interaction trace, call context.
Job	Batch steps and allocated datasets	Step ordering, dataset state, restart/error behavior.
Flow	End-to-end business scenario	Cross-boundary state transitions and release evidence.

context-bounded execution unit intended to establish one named business observation together with required dependencies and environmental assumptions. The ten-row ledger does not test whether such slices can be discovered, executed, or recomposed.

For example, an observation might be "the settlement output record for a rejected payment," "the updated balance field after interest accrual," or "the return condition and journal record produced by a batch step." Under the proposal, a slice would include relevant COBOL paragraphs or programs, copybooks, file layouts, input records, initialized storage, call context, and boundary adapters. Whether such a slice remains executable and preserves context is an untested hypothesis.

This ladder lets a team validate the hard, high-risk parts first and then assemble the evidence upward. A green program slice does not automatically make a green batch job; the job-level contract adds JCL parameters, allocated datasets, step ordering, restart behavior, and external boundary conditions. Conversely, when a flow diverges, the retained lower-stage evidence makes the failure localizable instead of forcing a blind end-to-end debugging exercise.

B. Discovering Call and Data Context

The proposed planner would begin with a repository inventory and construct a unified graph with control, data, and build edges. Program-dependence and interprocedural models motivate this design [10], [11], while program slicing motivates tracing a business observation backward through possible dependencies [9]. This paper does not implement or evaluate graph construction, resolution accuracy, or slice closure [9].

For COBOL, the useful dependency targets may be a paragraph, section, entry point, subprogram, CICS transfer, or JCL `EXEC` step. The proposed context key would include job or transaction identity, program identifier, normalized call path, fixture version, and declared dataset state. No evidence here shows that v1 recorded or resolved these fields.

Static analysis will not resolve every edge. The proposal would supplement it with observed edges from reference traces and surface unresolved dependencies as frontier items. The ten-case ledger contains neither traces nor frontier records, so this behavior is wholly unvalidated.

C. Boundary Control and Bytewise Evidence

The point where a slice meets the rest of the system is its *boundary frontier*. Each frontier edge is classified before execution: (i) included in the slice, (ii) represented by a deterministic adapter, (iii) observed but not yet modeled, or (iv) blocked. Only the first two states permit a MATCH verdict. The third produces INCONCLUSIVE evidence with an explicit follow-up item; the fourth blocks the slice. This rule prevents a convenient mock from masquerading as a verified integration.

Bytewise comparison is useful only at the correct representation layer. The proposal would compare identical physical records byte for byte and would decode legitimately different encodings through a contract-pinned schema before semantic comparison. Preserving original bytes and exact offsets is a design requirement. The v1 ledger contains none of this evidence, and T10 shows that the historical normalization policy failed on its EBCDIC case.

D. Context-Preserving Construction and Recomposition

Algorithm 2 is pseudocode for a proposed scale-out procedure, not an implemented algorithm evaluated here. Its intended safety property is monotonic evidence: a larger stage should not discard a lower-stage divergence, contract violation, raw

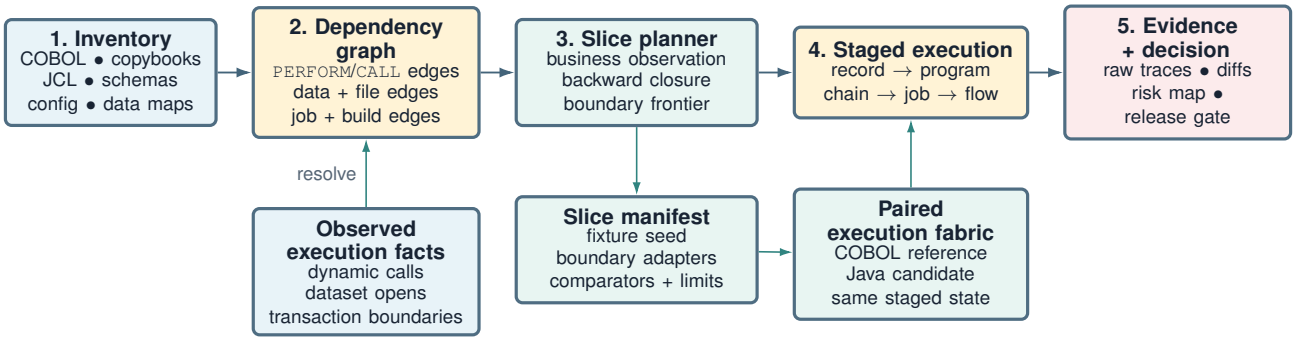

Fig. 2. Unvalidated system-scale flow proposal. The transformation, staged execution, and dependency retention shown here are future evaluation targets, not v1 results.

Algorithm 2 Proposed slice planning and staged recomposition (unvalidated)

Require: Inventory I, business observations Z, contract templates Π

$G \leftarrow \textsc{MergeEdges}(I)$
$Q \leftarrow [\,]$ ▷ planned verification slices
for all $z \in Z$ **do**
  $S \leftarrow \textsc{SliceClosure}(G, z)$
  $B \leftarrow \textsc{BoundarySet}(S, G)$
  $K \leftarrow \textsc{BindContract}(S, B, \Pi)$
  $Q \leftarrow Q \cup \{\textsc{Slice}(S, B, K, z)\}$
end for
$L \leftarrow \textsc{TopologicalStages}(Q)$
for all stage $\ell \in L$ **do**
  for all slice q in ℓ **do**
    $v \leftarrow \textsc{Verify}(q)$
    if $v \in \{\textsc{Diverge}, \textsc{Inconclusive}\}$ **then**
      $\textsc{PersistBlock}(q, v)$
    end if
  end for
end for
return $\textsc{ReleaseEvidence}(Q, L)$

trace, or unresolved boundary. Future work must test whether this property holds in implementation and recomposition.

V. Historical Version-1 Ledger Context

A. Scope

The supplied v1 paper calls its source the "Verified Snippet Demonstrator" and describes self-contained COBOL/Java programs plus a Docker workflow. The present package preserves none of those programs, container manifests, raw traces, or comparator outputs. Runtime details are therefore unverified historical descriptions. The ten final verdict rows are the entire empirical evidence base, and they say nothing about the proposed system-dependence graph, trace collection, slice planner, subsystem adapters, or production runtime emulation.

The proposed policy would label an unsupported subsystem as a contract failure or INCONCLUSIVE until an adapter and observation rule are supplied. The historical ledger does not test that behavior.

Listing 1. Contract-aligned fixed-point calculation in Java.

```
BigDecimal amount = new BigDecimal(line.substring(0, 8)).
     movePointLeft(2);
BigDecimal rate = new BigDecimal(line.substring(8, 12)).
     movePointLeft(2);
BigDecimal interest = amount.multiply(rate).multiply(
     BigDecimal.valueOf(term))
    .divide(BigDecimal.valueOf(100), 2, RoundingMode.HALF_UP)
        ;
```

B. Decimal Semantics: A Representative Failure Mode

Financial code exposes why source-level similarity is insufficient. A COBOL field such as `PIC 9(6)V99` denotes a fixed decimal scale in the program model; a Java `double` is a binary floating-point value. Formatting a `double` at the end of a calculation does not restore the intermediate decimal semantics. The Java platform's `BigDecimal` API exists specifically for arbitrary-precision decimal arithmetic and explicit rounding behavior [7].

Listing 1 illustrates the candidate-side discipline required by a contract that specifies a two-decimal stored scale and `HALF_UP` rounding. The substring offsets, scale, denominator, and rounding mode are not implementation trivia: each is part of the equivalence surface and must be derived from the source record definition and contract.

VI. Evaluation

A. Methodology

The v1 supplied test suite contains ten manually constructed COBOL/Java pairs. It includes arithmetic, string handling, conditionals, iteration, sequential file I/O, numerical precision, order dependence, error handling, rounding behavior, and character encoding. Five cases were designed to agree; four were deliberately non-equivalent; and one EBCDIC case was intended to agree after normalization. This is an artifact-level evaluation, not a benchmark of conversion systems and not a scalability study.

The machine-readable ledger reconstruction and plotting companion for this revision were executed on July 11, 2026. All ten case identifiers, scenario labels, ground-truth assignments, and demonstrator verdicts were fixed from Table II before metric calculation; the companion recomputes the reported statistics but does not rerun the historical COBOL/Java programs.

For diagnostic calculations, a *detected mismatch* is treated as the positive class. Thus, a true positive is a genuinely non-equivalent pair reported as DIVERGE; a false positive is an equivalent pair reported as DIVERGE; and a false negative is a non-equivalent pair reported as MATCH. We report accuracy, precision, recall, and F1 because a raw pass rate can conceal whether the comparator is failing open or failing closed.

B. Results

Table II yields the confusion matrix in Table III: $TP = 3$, $FP = 1$, $FN = 1$, and $TN = 5$. The ledger records eight correct classifications among ten cases, giving accuracy 0.80. For the mismatch class, precision, recall, and F1 are each 0.75; specificity is $5/6 \approx 0.83$. The Wilson 95% interval is wide ($[0.49, 0.94]$). The defensible conclusion is a descriptive pilot ledger with known errors, not implementation feasibility or production accuracy.

Figure 3 makes the two errors and the resulting uncertainty visible at case level. The interval is a binomial interval over case-level correctness; it does not account for how representative the manually constructed scenarios are.

The two failures are technically informative. T9 demonstrates that tolerance policies must never silently override source-defined decimal scale and rounding rules. A converter may compute a financially material value incorrectly while remaining within an arbitrary epsilon. T10 demonstrates that byte-wise or Unicode-only text comparison is insufficient when the reference contract permits EBCDIC data. The remedy is not to disable comparison; it is to use a declared code-page and field-schema adapter, preserving raw bytes in the evidence bundle.

C. Exact Small-Sample Research Audit

The retrospective audit evaluates two explicit null comparisons. Against a Bernoulli accuracy of 0.5, observing at least eight correct cases among ten has one-sided exact probability $56/1024 = 0.0547$. Against an always-MATCH baseline, v1 improves accuracy from 6/10 to 8/10. The paired table has three v1-only wins and one baseline-only win; the two-sided exact McNemar probability is 0.625. Neither comparison rejects at $\alpha = 0.05$. These tests do not show equivalence to the null; they show that this ledger is too small to establish the claimed advantage.

Comparator sensitivity is outcome-informed and therefore not evidence of generalization. Correcting only the diagnosed T9 rounding policy or only the T10 encoding policy would mechanically raise this ledger to 9/10; correcting both yields 10/10. Those values demonstrate policy sensitivity, not validated improvement. A clean test requires freezing the revised comparator and evaluating new blinded pairs.

D. Scale-up Evaluation Plan

The proposed slice architecture requires a separate evaluation; the ten-case demonstrator does not validate it. A credible system-scale study should report at least five dimensions. *Closure* measures the percentage of required static and observed dependencies either included or explicitly modeled at the frontier. *Recomposition yield* measures how many green lower-stage slices remain green when assembled into a chain, job, or business flow. *Localization quality* measures the distance from an injected or discovered divergence to the first failing slice. *Cost* measures wall-clock time, parallel capacity, container reuse, and evidence-storage growth. Finally, *diagnostic utility* measures whether the retained trace, record diff, and context key let an engineer identify the responsible conversion boundary without rerunning the entire workload.

The study should preserve both reference and candidate artifacts, preregister the comparator policy, include deliberately seeded faults at record, call-boundary, and job-order levels, and separately label behavior changes approved by domain owners. The test-oracle literature makes clear why an old implementation is evidence rather than an infallible specification; a modernization program still needs an explicit mechanism for accepting intentional corrections [12].

VII. DISCUSSION

A. What a Validated Version Would Need to Establish

A future validated implementation could establish bounded behavioral agreement only for an executed workload under a declared contract. The present artifact cannot establish that capability: it lacks executable subjects, raw paired observations, trace bundles, and environment manifests. Applicability as a quality gate and fair comparison across conversion candidates remain hypotheses for a larger blinded study.

The framework cannot establish that a legacy program is correct, that every possible input is covered, or that a Java program is universally equivalent to its COBOL counterpart. The reference program may contain a latent defect; a conversion may intentionally repair it. That scenario deserves a signed behavioral-change exception, not an unannotated test failure or a quietly altered normalizer. Similarly, a green result for a GnuCOBOL capsule is not evidence of compatibility with an arbitrary z/OS configuration. Compiler dialect, NUMPROC behavior, collation, and subsystem semantics must be encoded in the contract or reported as limitations.

B. Positioning with Other Assurance Techniques

Differential verification complements rather than replaces static analysis, human review, property-based testing, security assessment, and integration testing. Static analysis is useful for dependency discovery and unsupported construct detection; differential execution is useful for detecting runtime semantic drift. Coverage instrumentation can prioritize fixtures, but coverage alone does not guarantee fault detection [2], [4]. The strongest modernization program uses these techniques together and treats STRUST's evidence as one decision input rather than a universal oracle.

VIII. LIMITATIONS AND THREATS TO VALIDITY

External validity. Ten small, manually prepared snippets do not represent enterprise call graphs, batch schedules, or stateful transaction systems. The reported metrics should not

TABLE II
VERSION-1 RESULTS, RECONSTRUCTED FROM THE SUPPLIED TEST LEDGER. "CORRECT" EVALUATES THE DEMONSTRATOR VERDICT AGAINST THE INTENTIONALLY ASSIGNED GROUND TRUTH.

ID	Scenario	Ground truth	Verdict	Correct	Diagnostic interpretation
T1	Arithmetic	Equivalent	MATCH	Yes	Baseline fixed-operation agreement.
T2	String processing	Equivalent	MATCH	Yes	Concatenation and substring behavior agreed.
T3	Conditional logic	Equivalent	MATCH	Yes	Selected branch outcomes agreed.
T4	Iteration	Equivalent	MATCH	Yes	`PERFORM VARYING` translation agreed.
T5	Sequential file I/O	Equivalent	MATCH	Yes	Declared output files compared equal.
T6	Numerical precision	Non-equivalent	DIVERGE	Yes	Detected decimal/binary precision divergence.
T7	Order dependence	Non-equivalent	DIVERGE	Yes	Detected sequence-sensitive output difference.
T8	Error handling	Non-equivalent	DIVERGE	Yes	Detected different failure behavior.
T9	Rounding behavior	Non-equivalent	MATCH	No	False negative; minor rounding difference escaped policy.
T10	Character encoding	Equivalent	DIVERGE	No	False positive; EBCDIC normalization was incomplete.

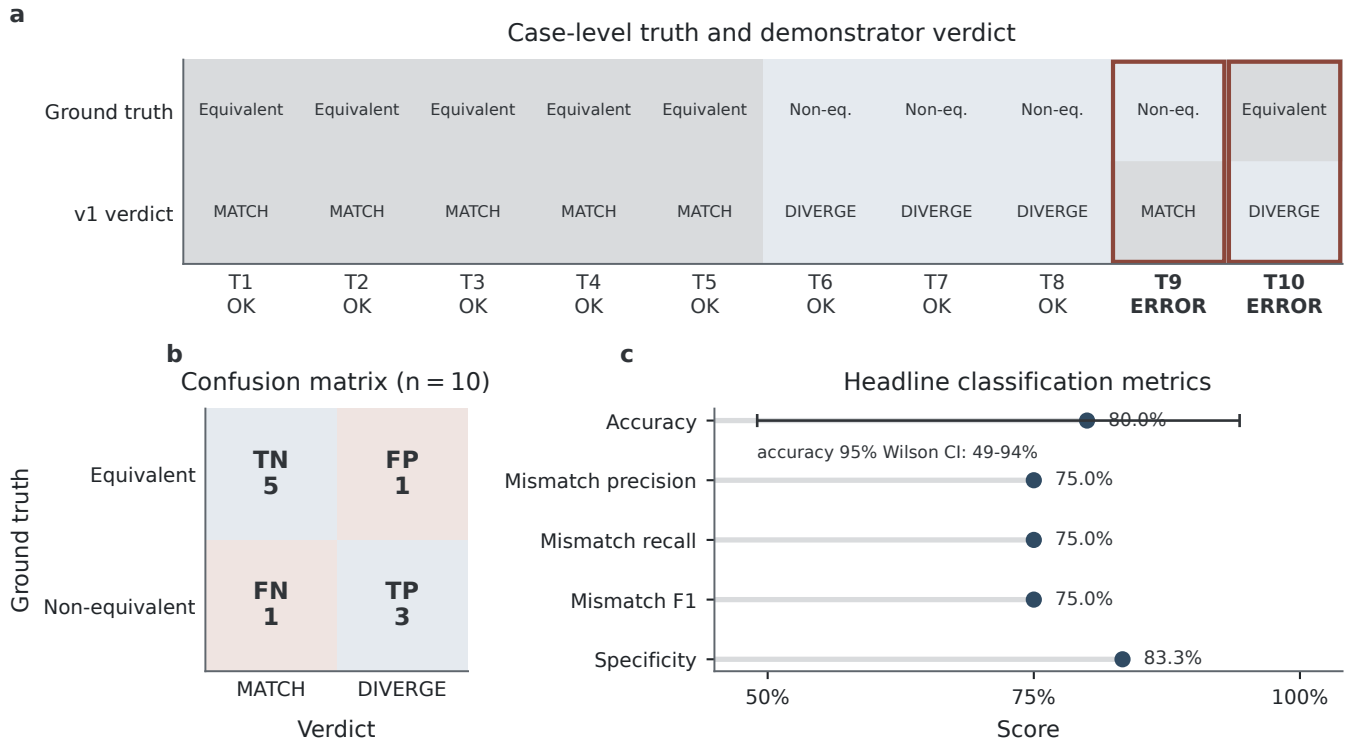

Fig. 3. Reproducible reconstruction of the supplied v1 ledger (analysis and plot executed July 11, 2026). Top: predeclared ground truth and demonstrator verdict for every case; T9 and T10 are the only errors. Bottom left: confusion matrix with mismatch as the positive class. Bottom right: headline metrics, with the Wilson 95% interval shown for accuracy. The small sample and hand-authored case inventory preclude a production-performance interpretation.

TABLE III
DIAGNOSTIC METRICS FOR V1 (MISMATCH IS POSITIVE).

Measure	Value
Accuracy	$8/10 = 0.80$
Mismatch precision	$3/(3+1) = 0.75$
Mismatch recall	$3/(3+1) = 0.75$
Mismatch F1	0.75
Specificity	$5/(5+1) = 0.83$

be generalized to a vendor, a language model, or a production portfolio. The 2026 companion recomputes metrics and figures from the supplied ledger rather than independently rerunning the historical demonstrator, so it adds auditability but no new external-validity evidence. The exact tests were specified after inspecting the ledger and have very low power at $n = 10$; their non-significance is a negative evidentiary result, not proof that the demonstrator is no better than either null comparator.

Oracle validity. Differential comparison assumes that the COBOL behavior is the desired reference behavior. Known legacy defects require an explicit exception mechanism; otherwise, a correct repair appears as a divergence.

Environment validity. GnuCOBOL is appropriate for an accessible demonstration but is not a substitute for every Enterprise COBOL dialect or z/OS subsystem. CICS, IMS, DB2, VSAM, JCL, load-module linkage, and scheduler behavior remain outside v1.

Comparator validity. Normalizers can produce false positives when too weak and false negatives when too permissive. This is why STRUST hashes the comparator and retains the pre-

normalized traces.

Scalability. Per-test container startup, compilation, artifact storage, and state reset can dominate runtime for small programs. The slice planner and staged execution fabric are a proposed architecture, not a demonstrated throughput result. Version 1 does not establish graph-extraction accuracy, parallel-scheduling behavior, frontier-closure rate, or the storage cost of long-lived evidence ledgers.

IX. Roadmap

The next implementation phase should prioritize trustworthiness before breadth. First, build an inventory parser and symbol resolver for copybooks, paragraphs, static and dynamic calls, JCL steps, datasets, and compiler options. Second, construct the system dependence graph and import observed call/data edges from reference traces, retaining unresolved edges as first-class frontier items. Third, implement schema-aware adapters for EBCDIC, zoned decimal, COMP-3, fixed-length records, relational state, and selected VSAM-like abstractions. Fourth, add deterministic fixture synthesis, staged state reset, and context-keyed paired execution so slices can be recomposed without losing provenance. Fifth, integrate the contract and verdict API into continuous integration, where DIVERGE and INCONCLUSIVE outcomes can enforce different release policies. Finally, evaluate on a blinded, larger corpus with published fixtures, seeded faults, and a preregistered comparison policy.

X. Conclusion

STRUST-v1 reframes COBOL-to-Java verification as a disciplined differential-evidence problem, but the preserved evidence does not validate the proposed system. The ten-case ledger yields 80% accuracy, yet exact comparisons against chance and a majority baseline do not reject at 0.05, and one-case removal materially changes the estimate. That negative result is the paper's defensible empirical conclusion. A context-aware verification slice remains a testable architectural hypothesis; evaluating it requires executable subjects, raw paired observations, blinded cases, environment digests, and a preregistered comparator. Until then, STRUST is a research proposal with an auditable pilot ledger, not a demonstrated migration quality gate.

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.