Evidence-Based Legacy Modernization: Contract-Scoped Software Equivalence for AI-Assisted Migrations

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

***Abstract*—AI-assisted modernization can produce code that compiles, reads clearly, and still changes business-critical behavior. Assurance therefore centers on whether a candidate preserves declared observations of a reference system under a declared workload and environment, rather than on source similarity alone. This paper develops a contract-scoped method for that comparison. It defines observable equivalence over execution traces, a verification slice that binds business observations to necessary code and data context, a tri-state verdict, and an evidence ledger that makes each comparison reproducible. The method supports AI-generated candidates by binding source, build, environment, fixture, comparator, and outcome provenance. Its empirical evaluation methodology uses seeded behavioral changes, staged recomposition, and independently reviewable evidence to assess migration assurance.**

***Index Terms*—legacy modernization, AI-assisted migration, differential testing, program slicing, software equivalence, evidence, regression testing**

I. Introduction

Modernization changes more than syntax. Data encodings, fixed-point arithmetic, record layouts, exception behavior, scheduling, and surrounding adapters may encode requirements that a new implementation does not make explicit. A candidate can therefore pass a style review and still alter an observable business outcome. The problem is heightened when a generative system produces broad code changes quickly: review attention may shift to the candidate's plausibility rather than the reference system's contract.

This paper defines *contract-scoped equivalence*: agreement over an explicit workload, environment, and observation set. The contract binds each assurance claim to finite, inspectable evidence instead of treating observed agreement as universal program equivalence. Differential testing provides a practical foundation because independently executed programs can expose disagreement [1]; program slicing supplies a way to make the tested context inspectable [2], [3].

II. Motivation and Related Work

Legacy migration is often framed as a translation problem: replace a language, runtime, database interface, or deployment target. The practical risk is broader. A business flow may depend on defaulted data fields, rounding and overflow semantics, ordering, batch restart behavior, or a particular error classification. These conditions can be scattered across programs, schemas, configuration, and operations manuals. A modern candidate that is cleaner than its reference may still be incorrect for the flow that pays invoices, reports balances, or reconciles inventory.

The legacy-modernization literature emphasizes incremental transition and explicit interfaces rather than a single all-at-once rewrite [6]. This paper adopts the same incremental instinct but makes the assurance artifact explicit: each completed step leaves behind an inspectable statement of what was compared, what agreed, and what evidence supports that statement. Containers, emulators, and generative coding assistants can improve repeatability or speed while also creating new version, configuration, and provenance obligations. The contract records those obligations as part of the comparison.

Testing research supplies useful but incomplete ingredients. Differential testing compares independently executed implementations [1]; property-based testing broadens input exploration [7]; metamorphic testing derives relations when a direct oracle is difficult [8]; and coverage research clarifies why executed lines alone are not a sufficient adequacy argument [9]. Software-engineering benchmarks such as SWE-bench use real repositories and executable tests [10], while automated-repair studies show the importance of distinguishing plausible patches from semantically correct ones [11]. The framework combines those lessons for migration: observed agreement has meaning under a stated observation boundary and test contract.

Modern testing standards also motivate separating process requirements from a claim of correctness. ISO/IEC/IEEE 29119 describes test-process and documentation concepts [12]; ISO/IEC 25010 offers a quality vocabulary [5]. Their relevance here is methodological: a credible modernization record identifies tested behavior, artifacts, assumptions, and residual risk rather than concealing uncertainty behind a pass/fail dashboard.

III. Equivalence as a Declared Contract

Let P be a reference implementation and Q a candidate. For fixture x and environment η, an execution emits a trace

$$\tau = \mathrm{Run}(P, x, \eta) = (z, o, f, \sigma, \pi), \qquad (1)$$

where z is an exit classification, o standard output and error, f declared emitted artifacts, σ declared persistent-state deltas, and π provenance. A test contract is

$$\mathcal{C} = (W, \eta, \mathcal{O}, \nu, \kappa, \beta, \varrho), \tag{2}$$

where W is the workload; $\mathcal{O}$ selects observations; ν is a versioned normalization function; κ is a comparator; β sets resource bounds; and ϱ identifies the source, candidate, fixtures, and runtime revisions.

For one fixture, the scoped equivalence predicate is

$$P \equiv_{\mathcal{C},x} Q \iff \kappa(\nu(\mathcal{O}(\tau_P)), \nu(\mathcal{O}(\tau_Q))) = \mathsf{true}. \tag{3}$$

The workload claim $P \equiv_{\mathcal{C},W} Q$ requires (3) for every declared fixture that completes within bounds. Normalization must be explicit. Transcoding an approved character encoding or masking a declared nondeterministic timestamp may be reasonable; suppressing a numeric difference, a record-field mutation, or an error path is not.

The verdict set is $\{\mathsf{MATCH}, \mathsf{DIVERGE}, \mathsf{INCONCLUSIVE}\}$. MATCH is agreement under the contract, not proof of universal equivalence. DIVERGE retains a minimal witnessed difference. INCONCLUSIVE covers timeout, unavailable dependency, incomplete observation, or a contract breach. The third state prevents a failed observation from being counted as a pass.

The contract should identify which differences are intentionally tolerated and why. For example, a modern adapter may serialize an approved Unicode representation where the reference adapter emits a legacy encoding, provided a versioned boundary conversion and comparator rule are approved. A different rounding mode, output record, authorization decision, or persistent-state delta should not be normalized away. This forces the migration team to convert implicit compatibility assumptions into reviewable artifacts.

IV. Verification Slices and Evidence

Whole-estate equivalence is rarely a tractable first unit. A *verification slice* for business observation y is the context-bounded closure

$$\mathcal{S}_y = (N, E, D, F, A, \mathcal{O}_y), \tag{4}$$

where N, E are relevant control and data dependencies; D is data schema and copybook-equivalent context; F fixtures; A adapters; and $\mathcal{O}_y$ the outcome being compared. Static dependence graphs explain potential influence; dynamic traces identify which paths a fixture actually traversed. Interprocedural slicing therefore guides the verification boundary while preserving explicit review of slice completeness [3].

For every paired execution, the evidence ledger stores the contract digest, source and build digests, environment manifest, fixture digest, raw observations, canonical observations, comparator version, verdict, and timestamps. Let H be a collision-resistant hash function. A chain record may be represented by

$$h_i = H(h_{i-1} \parallel H(\mathcal{C}) \parallel H(x_i) \parallel H(\tau_i^P) \parallel H(\tau_i^Q) \parallel V_i). \tag{5}$$

The chain provides tamper-evident ordering; it does not establish that inputs were correct, that a runtime emulator is faithful, or that hashes substitute for access controls. Evidence remains useful only if the underlying artifacts are retrievable and access-controlled.

An evidence record should make reproduction possible without exposing prohibited data. At minimum, it should contain content-addressed fixture references, masked or synthetic-field policy, source and binary identifiers, declared environment variables, adapter revisions, timeout values, and a comparator configuration. Where production data cannot leave a protected system, the ledger may retain a signed digest and a controlled replay reference rather than the raw value. Such a record does not replace access control, but it lets a reviewer distinguish a candidate regression from a changed fixture or runtime.

V. Staged Recomposition

The protocol expands verification in stages: (i) record or function; (ii) program or component; (iii) call chain; (iv) job or service flow; and (v) end-to-end business journey. At each stage, the contract names the observation boundary and adapters. A stage may advance only when its predecessor's divergent and inconclusive cases are accounted for; passing a local unit does not automatically certify its composition.

Slice selection needs a business-facing rationale. A slice begins with a named observable such as "customer balance after approved posting" rather than with a convenient module boundary. Static dependencies identify fields and routines that can influence that observable; dynamic traces reveal the subset traversed by each fixture. The protocol retains both views. A trace-only slice risks omitting an unexercised error path; a static-only slice can become too broad to diagnose. Evaluation compares reviewer effort and defect localization under trace-derived, static, and combined slice boundaries.

This staging creates an explicit trade-off. Small slices localize differences and admit controlled fixtures, while larger flows reveal ordering, integration, and state effects. Migration teams accumulate small, inspectable evidence claims and then test their composition at progressively larger boundaries.

VI. Evaluation Methodology

The empirical evaluation uses representative legacy-to-modern candidate pairs in isolated sandboxes or a transparent benchmark designed for this purpose. It seeds changes across observable categories: decimal rounding, padding or encoding, sort collation, field defaulting, return codes, timeout behavior, exception mapping, and persistent-state mutation. Seeds are labeled by type and severity before evaluator results are viewed.

The evaluation compares a baseline regression suite, raw-output differential testing, and the contract-plus-slice protocol. Primary outcomes are seeded-divergence detection recall, false-positive rate on approved variations, inconclusive rate, time to localize a defect, evidence completeness, and runtime

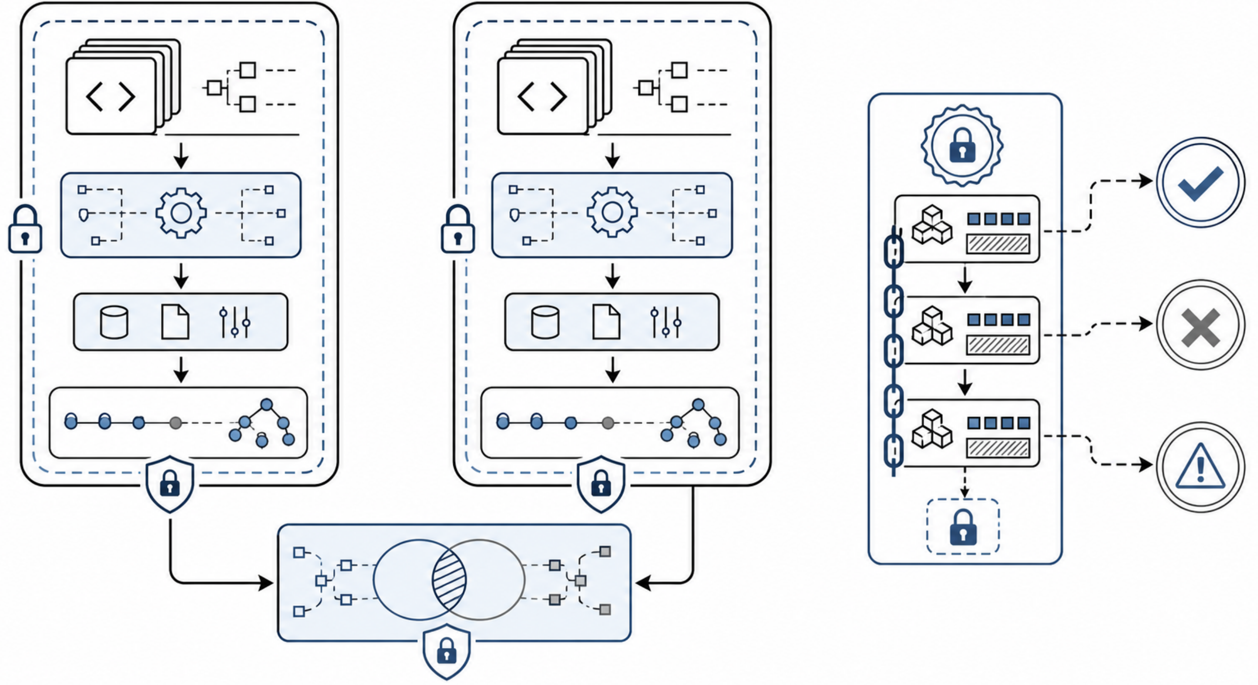

Fig. 1. Paired execution capsules, comparator, evidence chain, and tri-state verdict ledger.

TABLE I
EVIDENCE ARTIFACTS FOR CONTRACT-SCOPED MODERNIZATION VERIFICATION.

Artifact	Minimum recorded content	Assurance purpose
Test contract	Workload, observation boundary, normalizer, comparator, resource bounds, source and runtime revisions	Binds an equivalence claim to explicit conditions.
Verification slice	Business observation, static dependencies, exercised paths, schemas, adapters, and fixtures	Connects a business-visible outcome to inspectable context.
Paired trace bundle	Exit class, declared output artifacts, persistent-state deltas, timeout state, and provenance	Captures more than console output or compilation success.
Comparator dossier	Raw observations, canonical observations, policy version, first witnessed difference, and verdict	Lets reviewers examine why a pair matched, diverged, or was inconclusive.
Evidence ledger	Contract and fixture digests, build manifests, artifact references, timestamps, access policy, and chain record	Supports controlled reproduction without treating a digest as proof.
Stage review	Entry criteria, unresolved divergences, residual risks, approver role, and next recomposition boundary	Prevents local agreement from being misrepresented as end-to-end assurance.

cost. Let M be the set of seeded semantic changes and $\widehat{M}$ detected changes. Report

$$\text{Recall} = \frac{|M \cap \widehat{M}|}{|M|}, \qquad \text{Precision} = \frac{|M \cap \widehat{M}|}{|\widehat{M}|}. \tag{6}$$

Metrics should be stratified by change category and verification stage, with bootstrap confidence intervals rather than one pooled point estimate [4]. Independent reviewers should inspect a sampled set of matches, divergences, and inconclusives with access to the same ledger artifacts.

The evaluation pre-registers fixtures, mutation operators, comparator rules, and stopping conditions before running the candidates. It uses multiple migration styles or language pairs only where the same observation contracts remain meaningful, and it publishes a failure dossier for every seeded change not detected. The dataset includes both expected divergences and approved representational differences so that a method is not rewarded merely for flagging everything.

VII. Protocol Governance and Review

The protocol should establish ownership before comparison begins. A domain owner approves the business observation and fixtures; a migration owner approves the candidate revision and build configuration; an environment owner approves adapters and secrets handling; and an assurance reviewer approves the comparator and normalization policy. One person may hold several roles in a small project, but the responsibilities should remain distinct in the ledger. This separation makes it possible to locate a disputed match: was the input wrong,

the environment inconsistent, the candidate defective, or the comparison rule underspecified?

Every contract needs an expiration and change policy. Source revisions, schemas, third-party adapters, and regulations change. A contract written for one release should not silently persist after a field definition, authorization rule, or data-retention policy changes. The ledger should therefore bind a contract to version ranges and mark it stale when a dependent artifact changes. Reusing a stale contract is allowed only through an explicit review event, not by an unnoticed CI default.

AI-assisted migration calls for particularly strict provenance. The record need not retain sensitive prompts or proprietary code in a general-access store, but it should identify the tool class, model or generator revision when available, input artifact digests, generation parameters that affect determinism, post-generation edits, and the human reviewer who accepted the candidate. This is not an attribution exercise. It lets later reviewers determine whether a behavioral divergence follows from the candidate source, a regenerated variant, a changed prompt template, or a build-chain change.

Review should be risk-tiered. A formatting-only report conversion may need a small fixture set and one domain review. A change that affects money movement, identity, pricing, medical information, or access rights requires a richer observation set, adversarial fixtures, and explicit approval for any production-like execution. Organizations define risk scores, while the method requires the selected tier and its acceptance rule to appear in the contract so that a passing low-risk test cannot be misrepresented as high-risk assurance.

A controlled migration corpus uses public synthetic fixtures, versioned candidates, and independently adjudicated seeded changes. It compares a conventional regression process with contract-scoped differential evidence at multiple recomposition stages, reports the full false-positive and false-negative dossier, and measures review burden. This design ties any assurance improvement to inspectable comparisons rather than a single aggregate score.

VIII. Reference Verification Architecture

The reference implementation has four separable services: a contract registry, paired execution capsules, a comparator service, and an evidence store. The registry version-controls contracts, fixture descriptors, and normalization rules. Each execution capsule receives an immutable fixture reference and a constrained environment manifest, then emits the trace bundle in (1). The comparator receives paired bundles and the exact registry version, computes the contract-scoped verdict, and writes an evidence record. Separating these services prevents an informal local test setting from becoming an unreviewed production assurance claim.

Execution capsules should be isolated but not idealized. A useful capsule models relevant encodings, locales, clock policy, filesystem behavior, message ordering, and adapter responses to the extent the contract requires. When an emulator or test double replaces a real dependency, that replacement is an explicit adapter with its own version and limitations. The protocol must record where the reference and candidate use different adapters. Otherwise an apparent program difference may actually be an environment difference, and an apparent match may be caused by a shared but incorrect stub.

The comparator should operate over typed observations rather than a single serialized output whenever possible. A file output can be decomposed into record structure, fields, ordering, and byte-level representation. A database observation can be decomposed into row identity, field values, and transaction outcome. A service response can be decomposed into status, body, side effects, and downstream event emission. These decompositions let a review contract choose exact equality where appropriate while preserving a narrow, documented representation boundary where exact bytes would be misleading.

Evidence storage must make a distinction between integrity and availability. A hash chain can show that a retained record follows a previous record, but it cannot recover a deleted fixture, authorize access to a sensitive trace, or prove that a source system was faithfully captured. The implementation should therefore use durable artifact references, access controls, retention periods, and an evidence-verification command that checks availability as well as digests. A missing underlying artifact changes a result to inconclusive for any review that depends on it.

A minimal implementation uses one source-candidate pair, a synthetic fixture suite, an explicit adapter layer, and a comparator that can emit all three verdicts. It tests the whole evidence path with seeded rounding, encoding, ordering, error-path, and state-mutation differences before expanding to a broad estate. This sequence validates protocol operability and preserves the evidentiary boundary around any specific migration claim.

IX. Reproducibility and Audit Record

A verification claim should be exportable as a bounded audit package. The package includes the contract version, source and candidate build digests, fixture manifests, adapter revisions, execution commands, trace bundles, comparator configuration, and final verdict. It should also record which parts of the environment were simulated, unavailable, or inherited from a controlled runner. The goal is not to retain every byte indefinitely; it is to preserve the chain of decisions needed to reproduce a conclusion or narrow it when a dependency cannot be recovered.

The package should separate immutable evidence from derived views. Raw output and state snapshots are captured once under the declared retention policy. Canonical records, difference summaries, slice graphs, and reports are derived artifacts with their own version identifiers. A reviewer can then rerun a comparator after changing an approved rule without overwriting the original verdict. This is particularly important when a candidate is regenerated or a migration tool changes behavior: a new comparison is a new record, not a retroactive edit to a prior evidence claim.

Reproducibility also requires negative controls. The fixture suite should contain known-equivalent pairs, known-divergent

pairs, approved representation differences, missing-evidence cases, and deliberate environment mismatches. Each control checks a different protocol promise: detect a real mismatch, tolerate an approved boundary conversion, avoid treating missing observation as a pass, and localize configuration drift. A study that only injects obvious source-code bugs can make a comparator look strong while leaving its most consequential false-acceptance modes untested.

Finally, audit review issues a short bounded-assurance statement with every milestone. It names the business observation covered, the fixtures exercised, the strongest supported claim, unresolved divergences, and excluded behaviors. A reviewer who reads only that statement can distinguish agreement in a narrow slice from full-system equivalence. This reporting norm makes incremental progress visible while preserving the line between demonstrated agreement and untested risk. Empirical review measures whether these packages improve reviewer consistency and defect localization.

X. Threats to Validity and Limits

Fixture coverage limits every finite claim. Test doubles may differ from production services; environmental nondeterminism may cause spurious deltas; normalizers may hide meaningful differences; and data restrictions may exclude the cases that matter most. AI-generated code introduces additional risks, including underspecified prompts, hidden tool versions, and nonreproducible regeneration. These are reasons to strengthen provenance, not to assert certainty. Product-quality models such as ISO/IEC 25010 help enumerate quality concerns, but they do not turn observed agreement into a proof [5].

The approach also has governance limits. Production data must not be copied indiscriminately into test fixtures, evidence artifacts need retention and access policies, and a passing candidate still requires domain-owner approval. The method supports verification under a declared contract; deployment remains a separate authorization decision.

XI. Conclusion

Evidence-based modernization replaces the broad assertion “the migration worked” with a testable contract-scoped claim: preserve declared observations for declared fixtures under a declared environment, retain the evidence, and label uncertainty honestly. Verification slices, paired execution, tri-state verdicts, and an auditable evidence ledger make that claim reproducible across staged recomposition. The framework gives migration teams a disciplined basis for reviewing AI-assisted changes without confusing plausible code with demonstrated behavioral agreement.

References

[1] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[2] M. Weiser, “Program slicing,” in *Proc. 5th Int. Conf. on Software Engineering*, 1981, pp. 439–449.

[3] 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.

[4] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. New York, NY, USA: Chapman and Hall, 1993.

[5] ISO/IEC 25010:2023, *Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Product Quality Model*, 2023.

[6] M. L. Brodie and M. Stonebraker, Eds., *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. San Francisco, CA, USA: Morgan Kaufmann, 1995.

[7] K. Claessen and J. Hughes, “QuickCheck: A lightweight tool for random testing of Haskell programs,” in *ICFP*, 2000, pp. 268–279.

[8] T. Y. Chen, F.-C. Kuo, H. Liu, P. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, “Metamorphic testing: A review of challenges and opportunities,” *ACM Computing Surveys*, vol. 51, no. 1, art. 4, 2018.

[9] 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.

[10] C. E. Jimenez et al., “SWE-bench: Can language models resolve real-world GitHub issues?,” in *ICLR*, 2024.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[12] ISO/IEC/IEEE 29119-1:2022, *Software and Systems Engineering - Software Testing - Part 1: General Concepts*, 2022.