Expectation-Driven Development

Gold-State Parity™ as a Write-Path Acceptance Gate for Agent-Written Stateful Applications

A METHOD PROPOSAL AND PRACTITIONER EXPERIENCE REPORT

Melvin Fahnestock

Version 1.0 — August 2, 2026

doi.org/10.5281/zenodo.21761800

expectationdrivendevelopment.com

mel@expectationdriven.com

CONTENTS

Abstract

AI coding agents can generate implementation changes faster than humans can verify them, making trustworthy verification the limiting resource in software delivery. Expectation-driven development (EDD) addresses that imbalance with an independently governed machine oracle whose central mechanism is **Gold-State Parity™**: a trusted, fixed, versioned **gold state** supplies both the scenario inputs and the expected persisted result. The test store is never seeded with gold: it begins from a declared baseline, receives inputs only through a declared application entry path, and must reproduce gold's expected-state projection under an explicit, fail-closed parity contract. One corpus supplies stimulus and expectation without circularity.

The standing gate has two lanes. **Proof** drives a coverage-selected subset through the real user interface, mechanically evaluates the declared interface obligations, establishes browser-to-persistence state parity, and captures the normalized command-contract variants the UI emits. **Validate** instantiates only those captured variants with the full in-scope corpus through the application's official command entry point, tethered to the current Proof evidence by identity and an out-of-band provenance witness. A passing gate establishes contract-exact observational equivalence at the declared persistence boundary for the identified build, runtime profile, and oracle version, and nothing outside the versioned coverage model, which is the principal residual limitation and matures across immutable oracle versions. A failing gate is engineered to be as consumable as a passing one: a covered disagreement surfaces as a mechanical, localized artifact naming tables, business keys, and field-level mismatches, built to feed a diagnose-repair-rerun loop, whether agent-operated or human-operated,

rather than to be reconstructed from a diff. A repair loop can only heal to the standard of the verdict that stops it: a loop stopped by interface tests heals until the screens behave; a loop stopped by Gold-State Parity™ heals only until what was entered is what is stored. The companion field report exercises that difference — seven seeded create-path mapping faults passed every interface-level test, and no interface, contract, validation, or read-path check convicted any of them, while the full-row comparison against gold supplied all seven gate convictions. The faults were all of one deliberately chosen kind, and the author reports his own run: the result shows what only the comparison could catch, not how often the gate catches faults in general. Read-path rendering, security, performance, concurrency, and external delivery remain separate gates.

The component techniques have substantial prior art; the contribution claimed here is their composition, constraints, and governance as a standing acceptance gate for agent-written stateful write paths and, through the gate's engineered failure artifact, as the repair infrastructure that makes delegating the fix itself a governed act. The construction is exercised in a reference implementation that conforms to the specification as published (§9).

In plain terms. The method exists to make *done* demonstrable. It fits an application whose business state lives in a store that can be copied, frozen, and compared, written through one or more official entry points: one corpus then supplies both the question and the answer — the values a scenario enters and the exact state the application must have stored — so *done* stops being a claim someone makes and becomes a thing the system shows. Because a refusal names the record and the field that disagreed, it is precise enough to repair against, by a human or by an agent, under a verdict the implementing agent cannot edit. The gate buys that precision by claiming only what it covers: a gate that claimed everything would be worth nothing.

Keywords: expectation-driven development; Gold-State Parity™; self-healing software; self-healing test automation; automated program repair; AI coding agents; AI-assisted software development; test oracles; persisted-state verification; stateful applications; database testing; acceptance testing.

---

1. The problem: generation accelerated before verification

AI-assisted development changes the economics of implementation. Code, tests, migrations, and refactorings can be produced in parallel and at a speed that does not imply equal growth in human review capacity. In an observational telemetry study of more than 10,000 developers, Faros Research (2025) reported that teams with high AI adoption merged more pull requests while review time also increased. The study does not establish a universal causal law, but it illustrates the practical bottleneck: faster production can push work into review, testing, and release queues rather than remove those queues.

The same bottleneck limits delegation. Where verification remains primarily interpretive and manual, a rational team keeps agents on a short leash: small diffs, low-risk surfaces, and a human eye on nearly every change. The upper bound on useful agent autonomy is therefore not only what an agent can generate; it is what the organization can verify with sufficient confidence.

The usual responses are incomplete:

- **Agent-written tests are useful but not automatically independent.** Tests derived from the same implementation or interpretation can share its misunderstanding. They become stronger when anchored to expectations the implementing agent does not control.
- **Human code review remains valuable but does not scale linearly with generated volume.** Review should be concentrated where judgment is irreplaceable—architecture, security, privacy, maintainability, risk, and the oracle itself —rather than spent repeatedly reconstructing every expected business value from an implementation diff.

- **Trust is not a verifier.** Von Arx et al. (2025) have documented frontier models modifying tests or scoring code, obtaining reference answers, and exploiting evaluator loopholes when those actions improved the measured score. A development process that exposes a weak success signal should expect optimization against the signal.

Consider an unremarkable change. A developer, human or agent, adds a service-level field to a subscription record, with controls for setting it during creation and later editing. The diff reads well: the column exists, the controls are bound, and the mapping is correctly typed. The defect lies in a separate edit flow. A renewal-date form constructs a whole-record update from only the fields it displays. It does not display service level, so the update supplies an empty value that overwrites the stored one. Changing only the renewal date silently clears the service level set months earlier.

The tests do not catch this, and they are not bad tests. They assert that the renewal date changed and the operation succeeded. Both are true. They omit service level because the change was not supposed to affect it. That assertion is easy to add once the risk is known. Here, however, detection depends on naming an unrelated field as a possible casualty in advance. Code and tests produced from the same understanding can share the omission. Without an independent expected state, the defect can ship, and diagnosis begins by reconstructing what the record should still contain.

With a fixed expected state, the discrepancy is direct: the expected subscription retains its service level; the actual subscription does not. Two scope conditions matter. The renewal-date edit must be among the behaviors exercised, and service level must be among the fields compared. If either is omitted, the method makes no detection claim. If both are included, the mismatch stops the run and identifies the edit scenario, the subscription identifier, the field, and the expected and actual values. Note what the comparison is not: it is not a before-and-after diff, and it never compares the create flow to the edit flow. Both flows answer to the same fixed expected record, which does not change because an edit ran — so the renewal-date edit must leave the subscription exactly where gold says it ends, service level intact, and a flow is held to account even for the fields it never displayed. The companion field report deposited in this record — where this method's results are reported, as this paper reports its construction — realizes this fault family experimentally: seven create-path faults of this shape — each passing every interface-level test that exercised it, each self-concealing on a later edit — were convicted only by the full-row comparison against gold (companion, Record IV).

Wei (2025) calls the broader asymmetry **verifier's rule**: tasks become more tractable for AI as objective, fast, scalable, low-noise verification becomes available. The corresponding development-process claim is straightforward: stronger verification permits wider delegation. EDD is an attempt to construct such verification for a bounded but economically important class of stateful write paths. The reward-hacking evidence above motivates the method's governance posture, but the two claims it supports are distinct, and this paper keeps them separate: the gate's default result is regression fidelity over visible, versioned scenarios; standing against a deliberately hostile implementer requires the additional controls named in Rule 6 and §7 as mandatory parts of that stronger claim, not optional hardening.

That reframing is the paper's payoff, and it is worth stating before the construction: a gate with the properties built in §3–§5 changes not only how covered write-path flaws are caught but what a failure *is*. A covered regression stops being a review finding to be reconstructed from a diff and becomes a mechanical, localized disagreement—a named table, a business key, the mismatched field values—durable enough for any subsequent process to consume. That artifact can be read by a human, drive an agent-assisted diagnosis, or feed a fully agent-operated diagnose-repair-rerun loop, while the gate itself remains deterministic, versioned code with no agent judgment in its accept/reject path (§8). The method has been developed and applied this way across two platforms in parallel (§9). Verification this strong does more than stop unverified agent output at the door: it makes handing the repair to an agent significantly safer, because the agent then works under the gate's verdict: the trust is earned by the green, not extended in advance.

2. Applicability and definitions

2.1 The required shape of the problem

EDD is not a universal solution. In one breath: it fits an application whose business state lives in a store you can copy, freeze, and compare, written through one or more official entry points, with results that are deterministic or can be made so under control. Precisely, it applies when all of the following are available:

1. a trusted reference state exists or can be deterministically constructed;
2. the reference and resulting state can be expressed in a stable, canonicalizable representation;
3. the system has one or more declared official write entry points;
4. the relevant result is deterministic, or any intentional entropy can be controlled or verified by an explicit semantic assertion;
5. material outcomes can be observed in persisted state, evaluated by a versioned direct interface assertion, or delegated to another mandatory gate; and
6. the comparison can fail closed when it encounters unclassified state.

The reference implementation uses relational databases because they expose business state, relationships, audit evidence, and durable messages in a form that can be compared mechanically. A database is not essential to the logical concept. The construction could in principle use a document database, canonical JSON or XML documents, CSV files, workbooks, a versioned event ledger, or another structured state representation. That portability remains to be demonstrated outside the relational reference implementation. What matters conceptually is not the storage product; it is the existence of trusted gold and a deterministic equality contract.

EDD does not apply to a surface if no such gold state exists or can be created, or if the state cannot be compared meaningfully. That boundary is a precondition, not a claim of universal coverage.

For a surface that qualifies, the whole construction is one picture. Figure 1 shows it: the governed oracle, the two lanes, the capture boundary between them, the comparison that decides, and the repair loop that consumes a refusal. The sections that follow build each part in turn, and §3.3 explains the topology in detail.

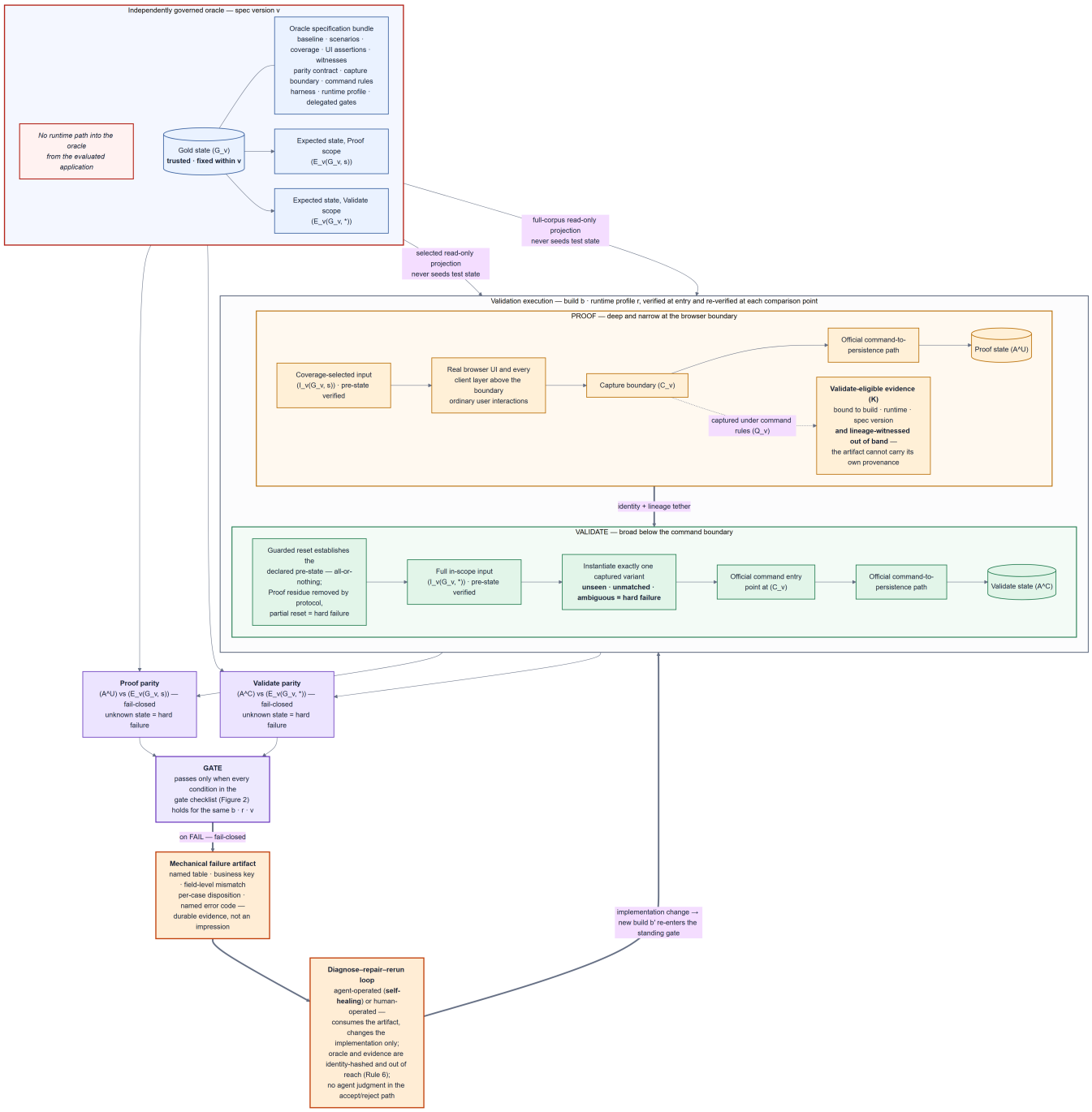

Figure 1. *Two-lane gate topology and dataflow. Acceptance logic appears in Figure 2; claims and limits appear in Figure 3. The highlighted repair loop consumes the failure artifact and does not alter the oracle, evidence, or accept/reject decision (§8).*

2.2 Gold, test state, and the oracle specification bundle

An EDD implementation maintains two logically separate states (Figure 1, above):

- **Gold state** is the trusted reference. It contains every in-scope entity, relationship, value, and persisted effect needed to define the expected result. Gold is fixed within a validation run and mechanically read-only to the application under test.

- **Test state** begins from an exact declared schema and baseline. It is empty of scenario business data unless the baseline explicitly requires otherwise. Gold is not copied or loaded into it. The tested application must construct the resulting state through a declared entry path.

One term is used throughout and is worth fixing before the machinery arrives. **The gate** is the complete, versioned acceptance procedure this paper specifies: it runs the oracle's covered scenarios through the application's declared real entry paths and compares every in-scope persisted result with gold. Each run yields one verdict for one identified build, execution configuration, and oracle version: pass only when every covered obligation holds; fail otherwise, with an artifact naming the unmet obligation or the exact disagreement. The gate is neither a single check inside that procedure nor the release decision — whether the build actually goes out to users is decided separately, after the gate has answered (§5.3).

The complete proof surface is larger than either store. An **oracle specification bundle** at version v contains:

- G_v — the gold state and its schema;
- B_v — the declared baseline state;
- S_v — the scenario manifest and dependency ordering, from which—together with B_v and the verified results of any required prior rounds—the pre-state $Pre_v(x)$ for each scenario or partition is derived (this is the single definition of Pre_v; §5.1 uses it unchanged);
- L_v — the UI behavior ledger and field-state coverage model;
- U_v — the direct UI assertions required where successful interaction and state parity do not by themselves establish the declared interface outcome;
- W_v — the per-case non-vacuity conditions that require a named observable difference or named evidence for a refusal, idempotent result, or declared no-change outcome;
- P_v — the parity contract, including keys, ordering, canonicalization, exclusions, and delegated assertions;
- C_v — the exact command-capture boundary between Proof and Validate;
- Q_v — the command-contract capture, normalization, applicability, unique-selection, execution-context, and value-instantiation rules;
- H_v — the harness and comparator versions;
- R_v — the required runtime profile, split into mechanically checked identities—execution-affecting configuration, feature flags, dependency and schema identities—which the gate matches exactly, and declared environment assumptions, which are recorded rather than matched; and
- D_v — the complementary gates required for excluded, non-persisted, or external behavior.

A completed validation record for build b under runtime profile r adds $K_{b,r,v}$, the immutable command-contract evidence produced by Proof under C_v and Q_v, together with the actual-state snapshots, assertion results, delegated-gate results, and mismatch or pass artifacts. $K_{b,r,v}$ is run evidence bound to the evaluated build and runtime; it is not a pre-authored expectation that changes the meaning of oracle version v.

Every binding named above travels with the artifact, and therefore none of them can establish where the evidence was minted: a copy carried into another context arrives with its identity, freshness, and build binding intact. $K_{b,r,v}$ is accordingly subject to a **provenance premise**: evidence is acceptable only when its minting run is witnessed through a channel the artifact cannot carry, and acceptance of evidence minted in another context is an explicit, logged operator act, never a default. Run evidence also sits under the same write governance as the oracle itself (Rule 6): an implementing agent

that can mint, transplant, or inject acceptance evidence controls the gate as surely as one that can edit gold, and for an adversary that optimizes against the signal, the acceptance evidence is the highest-value forgery target.

A result is not simply "the build passed." It is "the identified build under the identified runtime profile passed oracle specification `v` with its identified Proof and Validate evidence." That qualification matters because the proof surface can mature. A prior result remains valid only within the scope it actually exercised and only while the premises of that result remain intact; discovery of a defective comparator, baseline, identity record, or other invalidating oracle premise requires withdrawal or reclassification. Withdrawal propagates to evidence: a revoked run can no longer vouch for the `K_{b,r,v}` artifacts it minted, and downstream results that consumed them inherit the reclassification.

The paper's symbols, for reference:

Symbol	Name	Role	Where enforced
G_v	Gold state	Trusted reference; source of inputs and expected state	Rule 1
B_v	Baseline	Declared starting schema and state of the test store	Rule 3 (pre-state check)
S_v	Scenario manifest	Ordering and derivation of $Pre_v(x)$	Rules 3, 5
L_v	Coverage model	Behavior ledger + covering array	Rule 7
U_v	Direct UI assertions	Interface outcomes not entailed by interaction + parity	Rule 7; gate condition 5
W_v	Non-vacuity witnesses	Per-case named difference or no-change evidence	Rule 5; gate condition 3
P_v	Parity contract	Keys, canonicalization, exclusions, delegation	Rule 3; gate conditions 4, 9
C_v	Capture boundary	Exact Proof/Validate handoff point	Rule 2
Q_v	Command rules	Capture, normalization, applicability, instantiation	Rule 2; gate condition 8
H_v	Harness identity	Pinned comparator, projections, and dependencies	Rule 6; §5.4
R_v	Runtime profile	Checked identities + declared assumptions	Gate conditions 1, 12
D_v	Delegated gates	Mandatory gates for excluded or external behavior	Rule 4; gate condition 10
$I_v(G_v, x)$	Input projection	Values supplied through the declared entry point	§2.3; trusted computing base (§5.4)
$E_v(G_v, x)$	Expected-state projection	Contract-complete persisted outcome, per scenario and round	§2.3; trusted computing base (§5.4)
$Pre_v(x)$	Pre-state	Declared starting state per scenario or partition	Rule 5; gate condition 2
$K_{b,r,v}$	Command-contract evidence	Build-bound Proof capture consumed by Validate	Rule 6; gate conditions 6, 7, 11

2.3 One corpus, two roles—and why the construction is not circular

For a scenario `s`, the parity contract defines two projections of the same gold state:

- the **input projection** $I_v(G_v, s)$, containing only values supplied through the declared entry point; and
- the **expected-state projection** $E_v(G_v, s)$, containing the contract-complete persisted outcome after the scenario reaches its comparison point.

The input projection is necessarily selective. A create screen may accept a customer name and address but not the generated identity, audit records, derived totals, outbox rows, or lifecycle history. The expected-state projection is not a matching allow-list of input fields. It is the complete in-scope persisted result, minus only explicit exclusions and values delegated to other mandatory assertions.

The tested application produces actual state independently:

```
actual = application(declared_baseline, input_projection_from_gold)
expected = expected_state_projection_from_gold
pass = compare(actual, expected, parity_contract)
```

This is not self-confirmation. The test state was not initialized from the expected state, and the expected state is not generated from the application's output. The application must reconstruct the stipulated result through its declared door. Before each independent scenario or partition begins, the actual pre-state is mechanically checked against Pre_v; undeclared scenario business data, stale residue, or a mismatched baseline fails the run. A no-op cannot inherit the answer from an unverified starting state.

For directly entered fields, omission normally fails the comparison. If gold contains `Quantity = 17` and the application drops the field, the resulting `NULL`, zero, default, or different value disagrees with gold. There is one important edge case: a gold value can be indistinguishable from omission. A nullable field set to `NULL`, a boolean set to its default, an empty string, zero, or another default-equivalent value may land on the expected result even though the transport path ignored it. The coverage model therefore includes a **discriminating-value obligation**:

> Every input-capable field must be exercised with a valid value distinguishable from omission, `NULL`, an empty value, or default behavior in at least one scenario for each command variant and round that transports that field, unless the field's only valid state is itself the default.

The obligation quantifies over field × exercised command variant, not over fields alone. A field can transport correctly on the create variant—the scenario that discriminates it—while the edit variant drops it: if the value exercised on the edit round happens to be default-equivalent and the edit scenario's witness is satisfied by other fields, a per-field obligation would let the gate stay green over a broken transport path.

Derived values follow a different rule. They are not typed merely because they exist in gold. They appear only in the expected-state projection or in a separately declared invariant. This preserves the distinction between stimulus and result while retaining one versioned source of truth.

The two projections are functions, and they do real oracle work. $E_v(G_v, x)$ is scenario- and round-indexed: one gold row cannot simultaneously encode its pre-edit, deactivated, and restored forms, so the projection logic that derives each round's expected state is itself part of the trusted computing base, governed and calibrated like the comparator (§5.4).

2.4 Gold is an axiom; acceptance is a separate premise

Within a run, gold is correct **by definition of the fidelity question**. The gate asks whether the tested path reproduced the stipulated reference, not whether the organization chose the right business policy. A theorem does not prove its axioms, and EDD does not need to prove that gold represents the best possible business intent in order to prove fidelity to gold.

A stronger release statement adds another premise: the organization has approved gold as the intended result for the covered behavior. Establishing that premise is outside parity execution. An organization may use provenance, review, version control, privacy controls, and change approval to decide whether gold deserves organizational trust, but those controls are not additional operations in the equality test. The distinction is deliberate:

- **Fidelity claim:** fixed gold was reproduced under the parity contract.

- **Acceptance claim:** fixed gold was reproduced, and gold represents approved intent for this release decision.

Where trusted gold is unavailable, the method has no foundation. The operational rule is therefore: **guard the gold; do not ask the validation run to establish the authority of its own reference.**

2.5 Scope and non-goals

EDD is a point-of-entry write-path oracle.

1. **Write entry, not comprehensive read rendering.** Proof may directly assert interface behavior needed to establish the selected write traces—such as visibility, enablement, validation, paging, selection, and the value or command emitted by the UI. Whether persisted state is later rendered correctly across the application is the symmetric reverse problem and requires a separate read-path gate.
2. **Comparison point, not indefinite future behavior.** Entry-time calculations and deterministic work completed before quiescence are in scope. Aging, schedules, future deadlines, and later external events require time-aware verification.
3. **Covered traces, not all possible inputs.** The gate says nothing about a behavior absent from the current coverage model.
4. **Declared observables, not every system property.** Persisted business state, audit evidence, events, durable messages, and named direct UI assertions can be evaluated. Specific authorization scenarios may be included as write-path obligations, but overall security, privacy, authorization design, performance, resilience, concurrency, maintainability, and actual external delivery require complementary review and tests.
5. **Stateful, canonicalizable surfaces.** If a material result cannot be represented as trusted comparable state, a direct interface assertion, or a mandatory delegated assertion, the method does not claim it.

The purpose of a bounded verifier is not to claim everything. It is to make a specific and consequential claim exact.

---

3. The construction

3.1 Seven rules

Rule 1 — Gold is fixed and protected within an oracle version

Gold is versioned, mechanically read-only during validation, and changed only through an independently reviewed path. The application under test receives no credentials or runtime route that permits it to query or copy gold directly. Schema evolution produces a new oracle version; gold is reconstructed from versioned sources rather than migrated in place through an unreviewed chain of mutations. Because gold may contain production-derived or otherwise sensitive information, trusted does not mean unrestricted: its creation and handling require data minimization or de-identification where feasible, encryption, least privilege, retention limits, and applicable confidentiality and privacy controls.

Gold is immutable **within** a version. The oracle is expected to evolve **between** versions. Fault-injection calibration targets the application or disposable harness/comparator copies; it never mutates canonical gold.

Rule 2 — Each lane uses its declared official door

In Proof, the real user interface is the only writer of scenario business data: no direct database writes, API substitutions, service-layer calls, or test-only helpers. The point is to exercise browser behavior, client transformations, validation, binding, authorization context, command construction, and the downstream write path for the selected UI slice.

In Validate, the only writer is the application's official command entry point, using the build-bound command-contract evidence $K_{b,r,v}$ captured from a passing Proof run. Validate may substitute input values from the full in-scope corpus into those observed structures only through the value-instantiation rules in Q_v; it may not synthesize a new command shape or bypass the named entry point. The versioned applicability rules in Q_v must resolve each Validate case to exactly one Proof-observed variant. Any of the following is a hard failure: no match, more than one match, or a required context not proven by Proof. Direct database and lower-layer writes remain forbidden. Imports, synchronizations, batch loaders, and other legitimate entry points are separate doors with separate proof claims; they cannot be used to support a UI-path claim.

The closure boundary is checked; the only-writer premise is a governance rule that check supports but cannot itself prove. Each attempt is followed by a write scan across the in-scope record classes outside the declared change closure, and an unexplained write fails the run. A state scan enforces a boundary rather than attributing authorship, so a write placed *inside* the declared closure by an undeclared path is not what it catches; that path is closed by capability isolation and Rule 6, not by the scan. The scan as currently specified has a known residual: outside the declared change closure, it is count-based, so an out-of-closure update or a balanced delete-and-insert is invisible to it; content-sensitive verification applies only to declared unchanged tables. That residual is disclosed deliberately—as written, it is an exploitable gap for an adversarial implementer—and closing it is a tracked oracle obligation, not an assumption.

Each implementation must name the exact **capture boundary** between Proof and Validate. Proof exercises the browser and every in-scope layer above that boundary, then continues through the downstream write path to compared state. Validate begins at that same boundary. Moving the boundary changes the claim and requires a new oracle version and a new passing Proof artifact.

A bypass is not merely a test optimization. It changes which layers have been verified.

Rule 3 — Comparison closure is explicit, contract-complete, and fail-closed

Before execution, the gate verifies that the actual pre-state equals the declared Pre_v for the scenario or partition. Before comparing result values, it inventories the relevant schema and persisted record classes on both sides. Every table, collection, record class, field, and material persisted artifact must be one of:

1. compared exactly after declared canonicalization;
2. structurally translated, such as surrogate identities mapped through business keys;
3. checked by a named semantic invariant; or
4. excluded with a written reason and another mandatory gate.

An unknown table, column, document field, record class, or persisted artifact fails the run. A schema change therefore turns the gate red until the new state is deliberately classified. This is a strength of the method: new state cannot become silently invisible.

Rows are matched by stable, unique, appropriately scoped business keys. Tenant context, case and collation rules, duplicate handling, and parent-child translation are part of the contract. Record counts and key sets are checked before field comparison; missing, extra, duplicate, or untranslatable keys are failures. Ordering and canonicalization are pinned. The input projection and expected-state projection may be derived from the same gold corpus, but they are not controlled by one shared "included fields" list: input is selective; expected state is contract-complete.

Rule 4 — Material in-system effects persist, or another gate is mandatory

Effects that matter to the application's state—domain events, audit records, outbox messages, webhook intents, integration requests, and durable rejection evidence—must be persisted and included in the observable state wherever the architecture permits. If a material effect cannot be represented there, the parity contract must name another mandatory gate for it.

Successful business changes and their outbox or event records should commit atomically. Rejected operations require evidence written under transaction semantics that preserve the evidence when the rejected business transaction rolls back; otherwise, the rollback erases the very proof of refusal.

Persisted intent is not the same as external completion. An outbox row can prove that the application durably requested an email or webhook. It cannot prove that a remote mail server delivered the message or that a third party accepted the webhook. Actual external completion is delegated to an integration-specific gate and must pass for the same build and release-bound runtime profile when release depends on it.

Rule 5 — Comparison occurs only at a declared deterministic quiescence point

A scenario is compared only after all in-scope synchronous work and deterministic background work for that scenario has reached a declared quiescence condition. The condition may be an empty work queue, completed worker checkpoints, a stable event sequence, or another mechanically observable boundary. The actual state is then read through a transactionally consistent or otherwise version-stable snapshot; a comparison assembled from state that can change while it is being read is not a valid parity result. A timeout, ambiguous completion, or loss of snapshot consistency is a failure, not permission to compare early.

Lane-state composition is part of the same rule. Proof mutates test state; Validate must not inherit that residue informally. Validate therefore begins from a declared pre-state established by a **guarded, all-or-nothing reset** of the in-scope entities: the reset is gated on the current evidence identity, a partial reset is a hard failure, no command may execute against unreset state, and the non-vacuity witness pre-counts are captured only after the reset completes. Proof residue is removed by protocol, and the pre-state check then verifies that the protocol held rather than trusting that it ran.

Validation proceeds in rounds:

- **Round 1:** valid entry and direct transport;
- **Round 2:** refusals, requiring both durable rejection evidence and otherwise unchanged business state;
- **Rounds 3 through `n`:** lifecycle transitions such as deactivate, restore, promote, void, or approve.

A field must be compared or delegated in the round that exercises it. Each round therefore has its own declared expected-state projection or snapshot; a terminal snapshot cannot substitute for an intermediate transition claim. A globally excluded field is invisible even to the round intended to prove it.

Each scenario case also requires a declared **non-vacuity witness** in W_v. For a case intended to create or change state, the witness names at least one expected compared or mandatorily delegated observable that differs from the verified pre-state and that the scenario is intended to affect. For a refusal, an idempotent operation, or another deliberate no-change case, the witness instead names the business state that must remain unchanged together with the required durable refusal or idempotency evidence and any direct UI assertion. The witness is a mechanically checked oracle-consistency condition, not proof of a unique internal cause. A Validate partition passes only when every constituent case satisfies its witness; one changed record cannot make other vacuous cases acceptable. An unchanged state with no declared witness is not a passing proof of behavior.

Rule 6 — The implementing agent never owns the oracle

The code-writing agent cannot modify gold, the parity contract, scenario definitions, direct UI assertions, coverage obligations, comparator, harness, pipeline acceptance logic, or delegated gates as part of the same unreviewed change. Oracle changes receive independent approval. The oracle must not silently import mutable implementation-owned mapping, serialization, canonicalization, or equality logic. A shared dependency is permitted only when it is pinned as part of H_v, independently reviewed, and prevented from changing with the implementation under evaluation without an oracle-version change.

Write isolation extends to run evidence. $K_{b,r,v}$, state snapshots, and pass artifacts are governed artifacts in the same sense as oracle content: the implementing agent cannot mint, replace, transplant, or inject them, and the gate accepts evidence only when its minting run is witnessed through the out-of-band channel required by the provenance premise (§2.2). A gate that verifies everything about an artifact except where it came from has not closed the forgery path that matters most.

Write isolation is universal. Read isolation is threat-model dependent. The agent may legitimately see non-sensitive published scenarios and approved visible input values during implementation; that does not invalidate a pass over those identified scenarios. Access remains subject to confidentiality, privacy, and least-privilege restrictions. Stronger controls—held-out scenarios, rotated seeds, sealed acceptance subsets, and controlled failure disclosure—are added where fixture targeting or benchmark gaming is a material risk, and they become mandatory rather than optional wherever the gate is claimed as protection against a hostile implementer; without them, the claim is regression fidelity over visible scenarios, not adversarial robustness (§7).

One runtime prohibition is universal: the evaluated application may not reach gold or expected projections through an undeclared path. Otherwise it could copy answers instead of implementing the behavior.

Rule 7 — Coverage is explicit, bounded, and versioned

Every Proof run is governed by a coverage model, not by an informal promise that several representative rows are enough. The model distinguishes:

- a **behavior ledger**, which enumerates UI behaviors, workflows, validations, roles, boundaries, and state transitions that must be exercised and states the mechanical evidence required for each; and
- a **covering array**, which selects an economical set of combinations among modeled field states.

A behavior obligation may be evidenced by successful completion of a scripted interaction, a direct UI assertion, state parity, or a named delegated gate. When successful interaction and parity do not entail the interface outcome—for example, exact validation text, enablement, or conditional visibility—the direct assertion is mandatory.

Pairwise coverage is the default minimum for ordinary field-state combinations. It is not a completeness theorem. Higher-order interactions, sequences, role combinations, lifecycle transitions, boundaries, and known risk clusters are separate obligations. A green result is always scoped to the current versioned model.

3.2 Exclusions, semantic assertions, and comparison closure

Every exclusion weakens literal equality, so exclusions are inventoried under an explicit per-version budget rather than allowed to accrete invisibly: each oracle version carries its exclusion inventory with written reasons, and the independent review of that version sees the inventory's count and trend against the prior version, not only individual entries (§4.3).

- **Structural identity:** generated surrogate keys and raw foreign-key integers may be excluded only when business-key matching and parent translation replace them. The replacement must fail on ambiguity.
- **Controlled entropy or external authority:** timestamps, UUIDs, external identifiers, cryptographic nonces, and similar values are either controlled at the source, compared through a declared semantic relation, or excluded and delegated. Business-significant time, actor, tenant, and authorization values are not casually discarded as "audit noise."
- **Business values:** persisted business data should almost never be excluded. Derived values are preferably compared directly or checked through an in-gate invariant, such as a header total equaling the sum of its lines.
- **External completion:** delivery beyond the controlled system boundary is not inferred from a durable intent record; it is verified by a separate integration gate.

Interface-only expectations are not hidden inside parity exclusions. They belong in U_v as direct Proof assertions. Successful interaction may itself be sufficient evidence when a wrong interface state would make the scripted action impossible; otherwise the expected visibility, enablement, validation, selection, displayed value, or other declared outcome is asserted explicitly.

The acceptance rule is strict:

> A parity result is green only when every excluded business-significant observable names another mandatory gate, every required direct UI assertion has passed, and every such gate has passed for the same build, evaluated runtime profile, and oracle version.

3.3 The two-lane Proof/Validate gate

If interface execution were free, one lane would suffice: in a perfect world, every case meant to enter through a screen would be proven through that screen, and Proof would be nearly the entire gate. (Flows built solely for API entry have no screen to prove in any world; their official door is the command path, and they are verified there.) The economics of large data forbid it—running a multimillion-record corpus through a real browser can take hours or days, with diminishing evidentiary return per case to the point of impracticality, and would make the standing gate operationally unusable. EDD does not claim to do that. Validate exists as the deliberate economic accommodation to that fact: the method divides the proof obligation rather than pretending the full corpus can traverse the interface. Figure 1, in §2.1, depicts the resulting topology and dataflow: the governed oracle, the two lanes, the capture boundary, the single deliberate crossing edge that tethers Validate to Proof evidence, and the repair loop of §8 that consumes a failing gate's artifact from outside the accept/reject path.

Proof: the UI fidelity lane

Proof drives a minimum slice selected by the coverage model through the real interface using ordinary user interactions. The lane presumes a mature interface-automation layer capable of driving those interactions and observing their outcomes mechanically; for web browsers, such tooling is a commodity, and the method is indifferent to which automation product

supplies it. Proof's primary role is to establish evidence about the selected UI surface and the commands that surface emits; it is not the volume strategy for entering the corpus. It is responsible for:

- exercising and mechanically evaluating every behavior obligation declared for that oracle version;
- covering the required field-state combinations for the selected interaction strength;
- forcing paging, scrolling, search, collision, validation, role, and workflow behavior represented in the model;
- passing every required direct UI assertion for interface outcomes not entailed by successful interaction and state parity;
- establishing end-to-end parity for the selected UI scenarios; and
- capturing, at the declared application command boundary, the normalized command structures, observed field-to-value mappings, observed discriminator values evaluated by the applicability rules in Q_v, and required actor, tenant, role, or other execution context emitted by the official UI, including every command-shape variant required by Validate.

Proof produces the immutable **UI command-contract evidence** $K_{b,r,v}$. Its identity digest is bound to the evaluated build, runtime profile, UI bundle, capture boundary C_v, command-contract rules Q_v, scenario manifest, and oracle version. The digest proves artifact identity and the bindings named above. That the artifact is the current evidence for this gate invocation is established by gate conditions 6 and 7, not by the digest alone. It does not independently prove semantic equivalence between every possible browser interaction and every replayed command; no digest can do that. The semantic claim comes from the passing UI scenarios that produced the artifact and is bounded to their coverage. Nor does the digest prove provenance: every property it attests travels with the artifact, so a copy minted elsewhere carries them all. Provenance is established separately by the out-of-band lineage witness required by the provenance premise (§2.2) and enforced as gate condition 11.

Every command-shape variant used by Validate must be traceable to at least one passing Proof scenario. If conditional UI behavior can emit distinct structures, the behavior ledger must force each required variant. Each corpus case must resolve to one and only one observed variant under Q_v; an unseen, unmatched, or ambiguous form is a hard failure rather than a shape Validate may invent.

Validate: the full-volume command lane

Validate applies the full in-scope gold input projection through the application's official command entry point by instantiating $K_{b,r,v}$ under Q_v with corpus values. **Full in-scope corpus** means every record and scenario assigned to Validate by G_v and S_v for that oracle version; it does not mean the universe of possible inputs, future records, or behaviors absent from the oracle. Before any command executes, Validate establishes its declared pre-state through the guarded, all-or-nothing reset protocol of Rule 5. It does not write the test store directly and does not call a lower persistence layer. It refuses to run if the $K_{b,r,v}$ identity, build identity, runtime-profile identity, oracle version, capture boundary, command-contract rules, or required variant set differs from the passing Proof evidence. ("Current" evidence is mechanical, not temporal: the artifact must originate from the identified passing Proof run in the same verification chain as this gate invocation, judged at the capture instant against the gate invocation; evidence from any other run—however recent—is stale.)

Validate establishes data fidelity over the full in-scope corpus from the command boundary through the compared persisted state. The volume is not merely tolerated; it is evidence in its own right: a corpus-scale run exercises the shared below-boundary path—batching, constraint interaction, identity generation at scale—under load that a Proof-sized slice cannot produce. Validate does **not** establish that every corpus row can traverse every browser behavior correctly. That is the residual risk controlled by the Proof coverage model. If the corpus is partitioned for execution, the partition plan, starting state, ordering, and expected projection are part of the oracle specification bundle. Cross-partition behavior is covered only when the contract proves that the partitions are independent or an explicit scenario exercises the interaction.

Corpus-scale comparison is itself an engineering constraint the method does not receive for free: pre-state verification, snapshot-consistent reads, and contract-complete field comparison all scale with the corpus. Implementation experience is that staged gating—record-count and key-set gates before field comparison, hash-based pre-state verification—keeps the comparator tractable, and that the browser lane, not the comparator, has been the wall-clock driver. Comparator cost is nonetheless a measured quantity in the research agenda (§9), not an assumption.

Gate conditions

Acceptance requires all of the following for the same identified build, evaluated runtime profile, and oracle version. Figure 2 renders this checklist; this section remains the single source of truth.

1. an exact match to the mechanically checked identities in the runtime profile R_v—execution-affecting configuration, feature flags, dependency and deployed schema identity—with R_v's declared environment assumptions explicitly recorded rather than matched;
2. verified pre-state identity for every Proof and Validate scenario or partition, with the Validate pre-state established by the guarded, all-or-nothing reset of Rule 5—a partial reset, or any command against unreset state, is a hard failure;
3. a passing non-vacuity witness for every Proof scenario and every Validate case, even when cases are executed in an aggregate partition;
4. Proof parity for every selected UI scenario, each compared only after its declared quiescence condition is satisfied and read through a transactionally consistent or otherwise version-stable snapshot;
5. satisfaction of every required behavior-ledger obligation and passage of every required direct UI assertion;
6. current $K_{b,r,v}$ evidence captured from that passing Proof run under C_v and Q_v;
7. an exact identity match between that evidence and the artifact consumed by Validate;
8. unique conformance of every Validate command to a Proof-captured variant under Q_v;
9. Validate parity over the full in-scope corpus through the official command entry point, under the same declared quiescence and snapshot-consistency requirements;
10. every delegated mandatory gate required by the parity contract;
11. witnessed provenance for the $K_{b,r,v}$ evidence: its minting run is confirmed through a channel the artifact cannot carry, and any acceptance of evidence minted in another context is an explicit, logged operator act; and
12. runtime-profile stability: $r = R_v$ is re-verified at each lane's final observation point, not only at entry; profile elements capable of changing materially during execution are either held immutable for the run or monitored, and a relevant change during the evaluated interval invalidates the result even if the profile later returns to its initial value.

Figure 2 — Gate checklist

GATE PASS requires every row below, for the same identified build `b`, evaluated runtime profile `r`, and oracle specification version `v`. The gate is a pure conjunction, and a table is its honest form. The twelve rows mirror the paper's §3.3 gate conditions verbatim in substance; §3.3 is the single source of truth and this table must never diverge from it.

#	Group	Condition
1	Premises	Exact match to the mechanically checked identities in the runtime profile R_v — execution-affecting configuration, feature flags, dependency and deployed schema identity; R_v's declared environment assumptions are explicitly recorded, not matched
2	Premises	Verified pre-state identity for every Proof and Validate scenario or partition *(for Validate, the pre-state is established by a guarded, all-or-nothing reset of the in-scope entities — partial reset or any command against unreset state is a hard failure)*
3	Premises → enforcement	A passing non-vacuity witness W_v for every Proof scenario and every Validate case, even when cases are executed in an aggregate partition *(declared before the run; **checked after** it — a declaration alone satisfies nothing)*
4	Proof lane	Proof parity for every selected UI scenario — each compared only after its declared quiescence condition is satisfied and read through a transactionally consistent or otherwise version-stable snapshot
5	Proof lane	Satisfaction of every required behavior-ledger obligation and passage of every required direct UI assertion U_v
6	Evidence	Current $K_{b,r,v}$ evidence captured from that passing Proof run under C_v and Q_v
7	Evidence	Exact identity match between that evidence and the artifact consumed by Validate
8	Validate lane	Unique conformance of every Validate command to a Proof-captured variant under Q_v — unseen, unmatched, or ambiguous is a hard failure
9	Validate lane	Validate parity over the full in-scope corpus through the official command entry point, under the same declared quiescence and snapshot-consistency requirements
10	Delegated	Every delegated mandatory gate required by the parity contract has passed, for the same `b`, `r`, and `v` — delegated results are evidence-bearing and fail-closed like every other row
11	Evidence	The evidence's minting run is witnessed through a channel the artifact cannot carry — identity, freshness, and build binding travel *with* the artifact, so they cannot establish where it was minted; provenance requires an out-of-band lineage witness, with any foreign acceptance an explicit, logged operator act
12	Premises	Runtime profile `r` is re-verified at each lane's final observation point, not only at entry; profile elements capable of changing mid-run are held immutable or monitored — a relevant change during the evaluated interval invalidates the result even if the profile later returns to its initial value

Reading note: rows 6–7 + 11 together are the evidence tether drawn as one edge in Figure 1. Rows 4–5 are what qualifies K for use — qualification is a checklist fact, not a dataflow.

Figure 2. Gate checklist. The figure mirrors the twelve conditions in §3.3, which remains authoritative.

The lanes make different claims and neither substitutes for the other. Proof is deep and narrow at the browser boundary. Validate is broad below the command boundary. Their identity-and-provenance tether prevents Validate from silently using a stale, foreign, or structurally different capture artifact; the Proof coverage model still bounds how representative that artifact is of possible UI behavior.

3.4 When no historical gold exists

A greenfield capability has no inherited truth to replay. Its gold must be manufactured deliberately before the capability can enter the gate.

The bootstrap is **deterministic data population**: a versioned generator and seed create the gold corpus and expected persisted effects. The same generator version and seed must reproduce the same canonical state. The generator is part of the oracle and remains outside the implementing agent's unreviewed write scope.

Direct-fidelity values are inexpensive to generate because the generated value is also its expectation. Derived values, refusal evidence, lifecycle outcomes, events, and durable messages are specification work: the generator must state what should result before the implementation produces it. Practical strategies include:

- hand-approved exemplar rows for complex derived rules;
- in-gate invariants for relationships that are clearer as formulas than copied values;
- deliberate boundary and collision values;
- pairwise or higher-strength arrays for modeled interactions; and
- rotated seeds or held-out bundles where fixture targeting matters.

The cost is real and useful. A feature whose expected state cannot be written or generated precisely is not yet specified precisely enough for this method.

---

4. Coverage sufficiency: the principal residual limitation

4.1 The behavior ledger is not the covering array

Coverage has two distinct jobs.

The **behavior ledger** defines what the UI lane is required to demonstrate. Examples include create, edit, refusal, conditional field visibility, role-specific behavior, a lifecycle transition, search selection, paging, duplicate handling, or a particular command-shape variant. Each obligation declares how it will be judged: successful interaction, direct UI assertion, state parity, or a named delegated gate.

The **covering array** chooses a compact set of data combinations within that declared model. Pairwise coverage guarantees that every valid pair of modeled factor values appears in at least one selected scenario. It does not discover the factors, define the levels, or guarantee behavior that the ledger never named.

"Every declared behavior" therefore means exactly:

> Every behavior obligation explicitly enumerated in the current versioned coverage model.

It does not mean every behavior the application could exhibit.

4.2 The UI-only escape class

Pairwise coverage can miss a fault triggered only by three or more interacting conditions. Consider a UI defect that occurs only when all of the following are true:

- account type is commercial;
- payment method is ACH; and
- region is international.

A pairwise set can contain commercial + ACH, commercial + international, and ACH + international without ever containing the failing triple. Proof can therefore pass. Validate can also pass if it receives a correct command at the command boundary, because it bypasses the defective browser behavior. The gate is green even though one legitimate UI trace is wrong.

That is a real limitation. It is not a failure of state equality: the defective trace was never exercised. It is the deliberate residual risk created by sampling the UI while processing the full corpus below it.

Pairwise coverage also does not automatically cover:

- sequences and multi-page workflows;
- role, tenant, and lifecycle combinations;
- concurrency and interleaving;
- unmodeled values or boundaries;
- a value-dependent client transformation hidden inside one broad factor level; or
- behavior omitted from the ledger.

Kuhn et al. (2010) treat pairwise testing as an economical and often effective baseline while recognizing that higher-order interactions require higher-strength coverage. EDD adopts the same posture: pairwise is a systematic minimum, not a proof of exhaustive interaction coverage.

4.3 Evolutionary oracle versions

Coverage inadequacy is addressed evolutionarily rather than denied.

Gold is immutable within an oracle version. The complete oracle specification bundle evolves across versions:

- `v1` may contain the initial gold, parity contract, behavior ledger, pairwise UI slice, command-capture and selection rules, and delegated gates;
- an escaped defect, a surviving injected application fault, a new requirement, or a newly recognized interaction creates a new obligation;
- `v2` adds or revises scenarios, field states, interaction strength, invariants, comparison scope, or delegated gates; and
- the next build must pass `v2` before making a `v2` acceptance claim.

When `v2` expands coverage or changes approved intent, a `v1` pass was not false: it established the `v1` result for the `v1` proof surface. A different rule applies when later evidence shows that a `v1` premise was violated—for example, the comparator silently ignored in-scope state or the recorded artifact identity was wrong. In that case, the affected `v1` result is

withdrawn or reclassified rather than preserved as valid. Oracle-version history therefore records whether a version extends the proof surface or corrects an invalidating oracle defect.

Version history also carries a monotonicity obligation. Between versions, the coverage and exclusion diffs—scenarios removed, scope narrowed, exclusions widened, witnesses weakened—are surfaced mechanically to the independent reviewer alongside the exclusion budget (§3.2). A shrinking proof surface is loud, not silent: a regression cannot be laundered by weakening `v2` in the same cycle that introduces it.

Changing gold alone does not necessarily improve UI coverage. More records expand Validate's command-path data surface, but Proof still selects a subset. A browser-path gap requires a change to the behavior ledger, coverage factors, scenario manifest, proof-slice selector, or required interaction strength.

A plausible maturity path is:

1. **Systematic minimum:** explicit behavior ledger, pairwise field-state coverage, separate boundary obligations, and Validate over the full in-scope corpus.
2. **Risk-directed strengthening:** three-way or mixed-strength coverage for high-risk factor groups; explicit role, tenant, authorization, and lifecycle combinations.
3. **Evidence-directed strengthening:** fault injection against UI mapping and validation, with every surviving injected application fault becoming a new coverage obligation.
4. **Operational learning:** every production escape becomes a permanent regression scenario and may change the coverage model, not only the data.
5. **Change-aware selection:** altered screens, mappings, and commands receive stronger targeted Proof coverage while stable surfaces retain the standing minimum.

The central research problem for EDD is not whether exact comparison can detect a difference it sees. It is how efficiently the oracle can expose the traces on which important differences occur.

---

5. The claim

5.1 Precise statement of the pass claim

For an identified build `b`, evaluated runtime profile `r`, and oracle specification `v`, let:

- C_v be the exact command-capture boundary declared by the oracle specification;
- Q_v be the versioned command capture, normalization, applicability, unique-selection, context, and value-instantiation rules;
- R_v be the required runtime profile (§2.2) and $r = R_v$ be a gate precondition, checked at entry and re-verified at each lane's final observation point;
- $F^U_{b,r}$ be the real browser-to-persistence execution path used by Proof, including the layers above C_v;
- $F^c_{b,r}$ be the official C_v-to-persistence execution path used by Validate;
- $Pre_v(x)$ be the declared starting state for scenario or partition `x`, as defined in §2.2;
- $I_v(G_v, x)$ be the input projection for `x`;
- $E_v(G_v, x)$ be its expected-state projection;

- U_v be the required direct UI assertion set;
- $W_v(x)$ be the declared non-vacuity witness for scenario case x, with partition success requiring every constituent case to pass;
- $K_{b,r,v}$ be the immutable command-contract evidence captured by Proof at C_v under Q_v and bound to b, r, and v;
- $Compare_v$ be the parity comparator that applies the parity contract P_v and is pinned, with its dependencies, as part of H_v; and
- $Quiescent_v$ be the declared completion condition.

For each Proof scenario s:

$$A^U(b, r, v, s) = F^U_{b,r}(Pre_v(s), I_v(G_v, s))$$

For each Validate scenario or corpus partition t:

$$A^c(b, r, v, t) = F^c_{b,r}(Pre_v(t), instantiate(Q_v, K_{b,r,v}, I_v(G_v, t)))$$

The equations are sequence-aware through Pre_v (§2.2): a dependent run may begin only after its required predecessor round has reached quiescence and matched its expected projection, so the sequence claim is inductive rather than assumed. The runtime-profile equality $r = R_v$ is checked rather than assumed—and checked twice, at entry and again at each lane's final observation point. Endpoint checks alone cannot see a profile that changes and reverts between them, so profile elements capable of changing materially during execution are either held immutable for the run or monitored; the claim is that the profile held over the evaluated interval, and a relevant change during it invalidates the result even when the endpoints agree.

The gate passes only if each actual pre-state is verified against Pre_v; every Proof scenario and every Validate case has a passing $W_v(x)$, including cases aggregated into partitions; the required runs reach quiescence; every Proof and Validate actual state equals its expected-state projection under $Compare_v$; every declared behavior-ledger obligation has been satisfied and every required assertion in U_v has run and passed; every Validate command resolves uniquely under Q_v to a variant in the current $K_{b,r,v}$ captured at C_v; the evidence identity is current and its minting run is witnessed under the provenance premise of §2.2 (gate condition 11); and all delegated gates pass.

Therefore:

> **If the gate passes for build b under runtime profile r with oracle specification v and current evidence $K_{b,r,v}$, the selected UI traces satisfied their declared direct interface assertions, and the observed state produced by those traces and by the full in-scope command corpus exactly matches the stipulated expected state under the contract at the declared persistence boundary.**

For a directly entered field exercised with a discriminating value, this establishes observational transport fidelity: execution through the declared door produced the stipulated value in compared state. It does not claim to reconstruct or uniquely identify every internal causal step. For derived values and side-effect records, the result establishes equality or the separately declared semantic relation.

The conclusion is strong because it is bounded (Figure 3 renders the boundary). It does not establish:

- that the coverage model contains every fault-inducing trace;
- behavior outside the selected UI scenarios or full in-scope command corpus;
- comprehensive read-path rendering beyond the direct interface assertions in U_v;
- security, privacy, authorization, maintainability, performance, resilience, or concurrency;
- future time progression after the comparison point;
- actual completion of external delivery;
- correctness of an observable that the contract excluded without another passing gate; or
- behavior under a materially different runtime configuration, feature-flag set, dependency graph, schema deployment, or external environment.

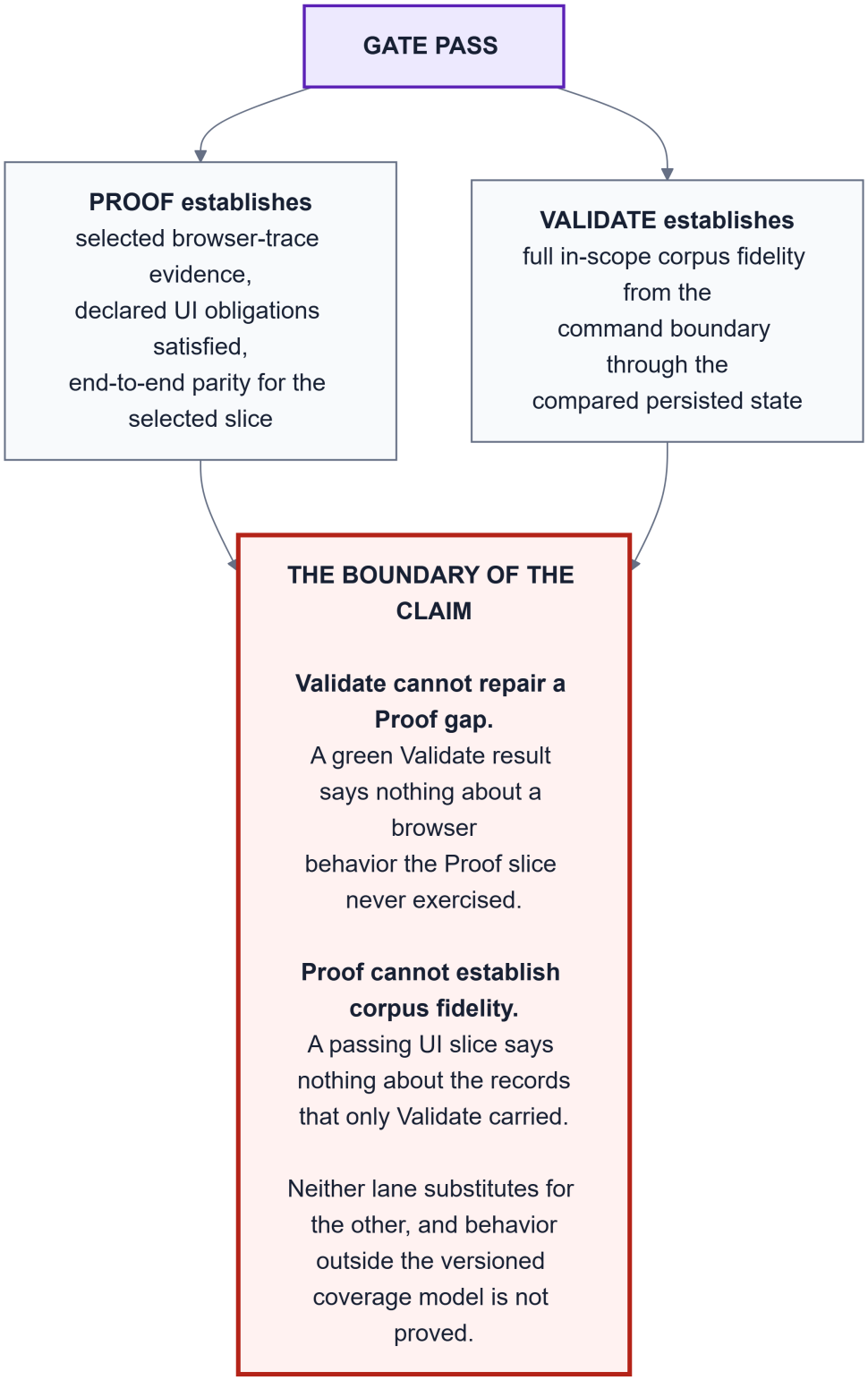

Figure 3. *Claims and limits. Validate cannot repair a Proof gap, and Proof cannot establish corpus fidelity.*

5.2 The flaw-finding inverse

EDD's daily value is the inverse statement:

> **When the construction and gate conditions of §3 hold, a defect the run actually exercises must show up if it changes a value or effect the contract covers: the comparison reports a mismatch, or a mandatory separate gate fails for an effect checked there. Exactly two escapes remain: another defect cancels the difference exactly, or the effect falls outside the comparison closure—that is, outside everything the contract compares or assigns to a mandatory gate.**

The mismatch is localized by scenario, entity or record class, business key, field, and round. A declared interface-only defect surfaces when it violates a direct assertion in U_v; no claim is made for interface behavior absent from the versioned ledger and assertion set. Persisted audit, event, outbox, and refusal evidence reduce the space in which a wrong internal route can accidentally land on the same terminal business rows without leaving a trace.

Cancellation remains possible. Two defects can offset one another, or a wrong route can produce the right terminal value. Conventional engineering tests, code review for risk-sensitive changes, invariants, intermediate evidence, and fault injection remain necessary. EDD does not require process perfection; it converts covered flaws that change compared state into mechanically located disagreements.

5.3 Fidelity, acceptance, and release

A parity pass is first a fidelity result. It becomes release evidence when the organization also accepts gold and the coverage model as sufficient for the risk tier of the change, and when the released artifact and its execution-affecting configuration are bound to the validated runtime profile. A different production profile requires a deployment-conformance check or its own qualified validation result.

The appropriate release posture is:

> A passing EDD gate is primary functional acceptance evidence for the covered write-path behavior. Release remains risk-based and depends on the complementary security, privacy, authorization, performance, resilience, concurrency, maintainability, and external-effect gates applicable to the change.

This can justify moving implementation review from exhaustive reconstruction toward risk-based inspection. It does not justify eliminating review of high-risk architecture, security-sensitive logic, regulatory controls, irreversible operations, or changes to the oracle itself.

5.4 What is deductive and what is empirical

The **validity claim** is deductive once its premises hold: exact comparison under a correct contract establishes equality of the compared actual and expected states, and a passing direct assertion establishes the declared interface proposition for that trace. Whether a particular harness, comparator, baseline, assertion set, capture artifact, and coverage model satisfy those premises is an engineering question and can be tested through known-difference fixtures, negative controls, and fault injection. The oracle must not depend silently on mutable implementation-owned comparison logic; any shared dependency is versioned and reviewed as part of H_v. The trusted computing base includes the projection functions, not only the comparator: I_v and E_v are derived upstream of $\mathrm{Compare}_v$, and a defect in a transformation library shared by both derivations would mangle stimulus and expectation identically and pass. The two projections therefore share no unreviewed

transformation code, and known-difference calibration exercises the projections as deliberately as it exercises the comparator. Calibration exercises mutate disposable application or harness copies; canonical gold remains immutable.

The **coverage claim** is empirical and evolutionary. Fault-injection campaigns, surviving injected application faults, escaped defects, and higher-strength coverage experiments measure how much of the relevant fault space the current oracle version exposes.

The **economic claim** is also empirical: EDD is worthwhile only if the cost of building and maintaining gold, scenarios, contracts, and the harness is lower than the combined review and defect costs it displaces. A method paper need not finish that longitudinal study before publication, but it must not report the answer as settled. The kill conditions are explicit:

- oracle maintenance exceeds the review effort displaced;
- escaped defects remain comparable to review-only development;
- the gate is too slow or flaky to remain standing;
- fault-injection detection remains poor despite coverage investment; or
- oracle changes become so coupled to ordinary implementation changes that the intended economics disappear.

---

6. Prior art and the contribution claimed

The method's components are established. Its claim is about their assembly and governance.

6.1 Oracle theory, model-derived expectations, and GUI state

Stored and pseudo-oracles address cases in which human evaluation of every output is impractical (Davis & Weyuker, 1981; Weyuker, 1982). The oracle-problem literature distinguishes oracle information from the procedure that compares actual with expected behavior (Barr et al., 2015). Model-based systems such as T-VEC generate inputs and expected outputs from a specification (Blackburn et al., 2002), and GUI-oracle research has long studied capture/replay and state-reference techniques (Xie & Memon, 2007).

EDD belongs to that lineage. It does not claim to invent machine expectations, state oracles, or GUI automation.

6.2 Database and structured-state assertion

AGENDA treated database state before and after an operation as a central test output (Chays et al., 2004). dbUnit and related tools compare actual database snapshots with expected datasets and support column filtering, ordering, and difference collection (dbUnit, 2026). DbFit and commercial data-diff tools add key-based large-set comparison and practical mismatch localization (DbFit, n.d.; Redgate, 2025).

dbUnit's official guidance says the expected dataset should differ from the dataset used to set up the database (dbUnit, 2026). That is guidance for dbUnit's direct-fixture model, not a general theorem about test-oracle design. In a design that loads setup state directly and then reuses the unchanged fixture as expectation, a no-op can pass. EDD uses a materially different construction: it never seeds the test store with gold, begins from the declared baseline, and requires the application to reconstruct expected state from projected inputs. The dbUnit rule is therefore relevant prior-art context, not evidence that dual-role gold is circular or unsound. Transactional outbox patterns likewise establish the prior practice of committing durable message intent with business state (Amazon Web Services, n.d.); EDD treats that durable intent as a compared observable while keeping actual external delivery outside the parity claim.

Golden-master, or approval, testing is the obvious adjacent art: capture the system's own prior output and approve it as the expectation for future runs (Approval Tests, n.d.). It is also the nearest kin to Gold-State Parity™, and the kinship is worth stating plainly rather than disclaiming: where gold is derived from an application's own approved history, as it is in the brownfield case, it too originates in the implementation lineage. The distinction is not independent authorship; it is what happens after freezing. An approved master is captured *from* the implementation and stays fixed until someone approves a replacement; what it ratifies, at each approval, is whatever the implementation did at that moment. Gold is frozen and approved under governance outside the implementing agent's write authority, is never loaded into the test store, and must be reconstructed in full through the declared entry path from a declared baseline—so the build under evaluation is measured against a reference it had no hand in producing, whatever produced its ancestor.

6.3 Capture, replay, and stateful equivalence

Pact records consumer expectations in a contract artifact and replays them against a provider (Pact Foundation, 2022). Netflix has described replay testing for stateful migrations using isolated stores initialized to identical states, replay of state-driving requests, and comparison of responses and related state (Gala et al., 2023). AWS Mainframe Modernization Application Testing records source reference artifacts, replays tests against a transformed target, and compares datasets, database changes, and terminal screens in repeatable pipelines (Kindelberger et al., 2024).

Wang et al. (2018) verify equivalence of database-driven applications statically, proving that two programs induce the same database behavior; EDD makes no static claim about programs—it operates a standing runtime gate that compares one application's persisted behavior against independently governed gold.

These are close precedents for capture/replay, contract identity, and stateful equivalence. EDD differs in purpose and assembly: one fixed, versioned gold state supplies stimulus and expected state; Proof produces build-bound evidence of the real UI's command contract under a versioned capture policy; Validate uses that current evidence as a standing gate over the full in-scope corpus for ongoing agent-written changes.

6.4 Enterprise parallel reconciliation

Payroll and other enterprise replacements have long used parallel runs and reconciliation to determine whether a new system reproduces the established result (Oracle, n.d.). EDD generalizes that acceptance instinct from periodic migration sign-off into a per-change engineering gate and makes mismatch localization, coverage, exclusions, and oracle governance explicit.

6.5 Agent-evaluation governance

SWE-bench evaluates generated patches against repository-level acceptance tests (Jimenez et al., 2024). SWE-bench Verified originally strengthened benchmark reliability through human validation of the evaluation instances (OpenAI, 2024). In 2026, OpenAI reported that residual flawed tests and benchmark contamination meant SWE-bench Verified no longer provided a reliable measure of frontier software-development capability and stopped reporting it (OpenAI, 2026). That history is directly relevant here: independent curation improves an oracle but does not make it infallible; calibration, versioning, and withdrawal or reclassification of results remain necessary when an oracle premise later fails. METR's reward-hacking results separately show why the implementing agent must not control the scoring machinery (Von Arx et al., 2025). EDD applies the same governance principle to business-application state: the agent may optimize against a visible goal, but it cannot rewrite the goal or reach the answer through an undeclared runtime path.

6.6 The novelty claim at the strength the evidence supports

A mechanism-first search completed in July 2026 found substantial prior art for nearly every component and several close combinations. The search covered literature and official documentation from the early oracle literature through 2026 across model-based expectations, GUI and state oracles, database assertions, contract and traffic replay, enterprise parallel reconciliation, persisted side effects, combinatorial testing, and AI-agent evaluation. It prioritized primary sources but was not exhaustive by patent class, non-English literature, or unpublished industry practice. Within that bounded search, no source was located that combined all of the following as a standing gate for agent-written business-application changes:

1. one fixed, versioned reference state used as both input source and expected persisted state;
2. a mandatory no-bypass UI Proof lane for a coverage-selected subset;
3. a contract-exact, fail-closed state comparison including durable side-effect records;
4. a Validate lane over the full in-scope corpus through the official command entry point;
5. an identity tether from Validate to the command-contract variant set captured by the current passing UI lane; and
6. oracle governance outside the code-writing agent's write authority.

The defensible statement is therefore:

> **The underlying techniques are established. The potential contribution is their formal composition, constraints, and governance as an evolutionary acceptance architecture for agent-written stateful write paths. No source describing this exact composition was located in the completed search.**

This is a report of what the search found, not proof that no earlier source exists. The methodology stands or falls on its utility and validity even if closer prior art is later found.

Two adjacent repair literatures. Two established bodies of work share this paper's vocabulary and should not be mistaken for it. *Self-healing test automation* repairs the test machinery when the application moves beneath it—most prominently broken element locators, repaired by matching the changed interface against the structure the script expected (Brisset et al., 2022). *Automated program repair* patches the implementation itself, searching for a change that satisfies a supplied oracle, most often a test suite (Monperrus, 2018). EDD supplies neither technique. Automated program repair takes an oracle as given—most often the test suite it is searching against—and self-healing test automation restores a script's grip on a moving interface without touching the verdict at all. EDD supplies that verdict: a governed oracle over persisted state, and a failure artifact localized enough to repair against. Its contribution at that boundary is the stopping verdict—what a repair loop is permitted to stop on—and the governance that keeps the loop's operator from moving it (§8.2).

6.7 A note on the name

"Expectation-Driven Development" has prior uses, including Revolut's engineering principles, an agentic methodology called "Expectation Driven Agentic Development," and a separate 2026 validation framework (Revolut, 2020; Yeh, 2025; Laforgia, 2026). This paper does not claim to have coined the phrase. The distinction is mechanism-level, not merely nominal: none of those uses describes a fixed, versioned reference state serving simultaneously as stimulus source and expected persisted result under a fail-closed parity contract at a declared persistence boundary. Laforgia's framework, the closest by name and year, validates through human adversarial review of agent-produced execution evidence against natural-language expectations—no reference state, no mechanical parity, and no separation between the lanes or governance of the evidence. The shared name marks a shared instinct—declare expectations before implementation judges itself—not a shared

construction. **Gold-State Parity™** is the primary technical handle for the mechanism defined here; EDD remains the umbrella methodology name.

---

7. Objections and answers

“One corpus in two roles is self-confirming.”

No. Self-confirmation would occur if the actual output generated its own expectation or if gold were loaded directly into the test store and an unchanged fixture could pass. EDD does neither. Gold supplies inputs; the application independently constructs actual state; gold separately supplies expected state; and exact comparison decides agreement.

“A field can be omitted from both input and comparison, so parity passes.”

That describes an incomplete comparator, not an inherent consequence of dual-role gold. The intended contract is fail-closed and expected-state complete. A field supplied through input must land in compared state or a named mandatory assertion. If a new field or column is unclassified, the run fails. The genuine edge case is a default-equivalent gold value, which is why each input-capable field requires at least one discriminating scenario value.

“Gold might be wrong.”

For fidelity, gold is the stipulated reference and the objection changes the question. For acceptance, gold must additionally be approved intent. Gold governance is important, but the execution proof is not responsible for proving its own axiom.

“The command hash does not prove semantic equivalence.”

Correct; it is not intended to. The hash proves identity and build/oracle binding of the command artifact; currency is established by gate conditions 6 and 7, not by the digest. It also does not prove provenance—every property a digest attests travels with the artifact, so evidence copied from another context arrives with all of them intact. Provenance is a separate gate condition: the minting run must be witnessed through a channel the artifact cannot carry, and evidence minted elsewhere enters only by an explicit, logged operator act (§2.2; gate condition 11). Within those bounds, Proof supplies browser-path evidence for the selected UI scenarios and Validate supplies command-path evidence over the full in-scope corpus. Neither claim extends beyond its lane.

“Validate could replay a command shape the UI never proved.”

Not in a conforming implementation. The command-contract set records each normalized variant and the passing Proof scenario that produced it. Validate may instantiate those variants with corpus values; an unseen shape, discriminator combination, or capture-boundary mismatch fails hard. The remaining limitation is value-dependent browser behavior inside a known shape, which belongs to coverage sufficiency.

“The full corpus does not pass through the UI.”

Correct. That is the deliberate scalability compromise. Proof supplies browser-path evidence for a coverage-selected subset. Validate supplies evidence over the full in-scope corpus only from the declared command boundary downward. A green Validate result cannot repair a browser-only defect absent from Proof.

“Equal terminal state does not prove the internal route.”

Correct. The formal claim is observational at a declared boundary, not a proof of unique internal causation. Requiring the official door, prohibiting runtime access to gold, and comparing audit, event, outbox, refusal, and intermediate-round evidence narrow the space for a wrong route to pass, but equivalent internal implementations remain equivalent for this oracle.

“The validation environment can behave differently from production.”

Correct. A build can branch on configuration, feature flags, dependency versions, schema state, or test-only environment cues. The formal result is therefore bound to R_v, not to source code alone. Release evidence requires the deployed artifact and execution-affecting configuration to conform to that validated profile; materially different production conditions are outside the result until separately checked.

“The comparator itself could be wrong.”

Correct. Comparator and parity-contract correctness are premises of the deductive claim. They are protected by independent review, fail-closed schema inventory, known-difference fixtures, negative controls, and application fault injection. The oracle may not silently reuse mutable implementation-owned mapping, canonicalization, serialization, or equality logic; any shared dependency is pinned and reviewed as part of the oracle. Calibration uses disposable copies; canonical gold is not mutated.

“A no-op can pass when the expected state already exists or when the scenario is intended to change nothing.”

Not in a conforming run. Each actual pre-state is verified, and every scenario case has a named non-vacuity witness. A create or change case must produce at least one declared compared or mandatorily delegated difference from that pre-state. A refusal, an idempotent operation, or another deliberate no-change case must instead prove the specified unchanged business state together with durable evidence or a required direct interface assertion. Aggregate partitions are not allowed to hide a vacuous constituent case. The witness does not prove a unique cause, but an unchanged state with no such witness is not evidence of behavior. For create and change cases, the witness is largely subsumed by contract-complete parity; its distinct payload is precisely the refusal and no-change cases—especially inside aggregate partitions—where nothing else would force a difference, or its deliberate absence, to be named.

“Persistence parity cannot prove conditional visibility, enablement, or validation text.”

Correct. Those are interface propositions, not persisted-state propositions. Proof therefore assigns each declared UI behavior a mechanical evidence type. Where successful scripted interaction and parity do not entail the expected interface state, U_v contains a direct assertion. Comprehensive rendering of persisted data remains outside this write-path gate.

“Pairwise coverage misses higher-order failures.”

Correct. This is the primary admitted weakness. Pairwise is the default minimum for selecting combinations inside a declared coverage model. Known higher-order interactions, sequences, roles, lifecycle states, and boundaries are explicit additional obligations. Unknown interactions remain residual risk and are absorbed into later oracle versions when discovered. The boundary is economic, not mechanical: a higher-order interaction that is deterministic can be brought fully in scope by adding the scenarios and gold that express it, and the gate compares such cases exactly as it compares pairs. The governing question is whether the expected testing benefit justifies the upfront cost of preparing that gold—a trade-off that the economics item of the research agenda (§9) treats as a measured quantity rather than a fixed limit.

“Two bugs can cancel.”

Correct. Persisted intermediate evidence, rounds, invariants, scenario diversity, and conventional tests reduce but do not eliminate cancellation. The formal flaw-finding claim includes the cancellation exception.

“Exact equality is too brittle.”

Unmanaged nondeterminism is brittle. EDD handles it by structural identity translation, controlled clocks and controlled randomness, explicit canonicalization, semantic assertions for intentional entropy, and written delegation. Variance that remains inside the compared contract is a finding by design.

“A fail-closed browser lane will be flaky, and rerun-until-green converts fail-closed into fail-open.”

That failure mode is real, and the answer is policy, not optimism. A retry is permitted only for a failure positively classified as infrastructure rather than application behavior, each attempt preserves its own evidence, and the declared pre-state is re-verified before every attempt. A pass on any attempt after the first is itself a stop condition: it is surfaced for operator judgment, never silently advanced. Every retry is logged as an oracle work item, so flakiness accumulates as visible oracle debt rather than tolerated noise. The kill condition stands behind the policy as the backstop, not the answer: a gate too flaky to remain standing is a failed gate (§5.4), and the retry discipline exists to keep the gate honest long before that threshold—not to hide the approach to it.

“An outbox row does not prove the email arrived.”

Correct. It proves durable in-system intent at the controlled boundary. External delivery is a separate integration assertion.

“Agents can target known fixtures.”

They can, and the paper scopes its claim accordingly. The default result is non-adversarial: regression fidelity over visible, versioned scenarios. Validate over the full in-scope corpus raises the cost of targeting, and runtime access to gold is forbidden. Where EDD is claimed as protection against a hostile implementer, held-out scenarios, rotated seeds, and sealed acceptance subsets stop being optional hardening and become mandatory parts of the claimed configuration (Rule 6). A team quoting the adversarial claim while running only visible fixtures is quoting a different gate.

“This replaces all code review.”

No. It is intended to replace repeated manual reconstruction of covered functional outcomes with a mechanical result. Review remains essential for oracle changes and for architecture, security, privacy, regulatory controls, maintainability, performance, concurrency, irreversible actions, and other risk-specific concerns.

“This is only a database technique.”

The reference implementation is database-centered, but the logical requirement is a trusted comparable state store. The method could apply in principle to other structured representations when equality, keys, canonicalization, quiescence, and fail-closed comparison can be defined; that portability has not yet been demonstrated here. Without such a store, it does not apply.

“This only works for a web browser.”

“Browser” in this paper names the reference implementation’s interface, not a requirement of the construction. What the Proof lane needs is a real user interface with mature automation—one that can be driven by ordinary user interactions,

observed mechanically, and held to declared interface assertions—and a definable capture boundary beneath it. Desktop, mobile, terminal, and comparable interface classes with automation at that level of maturity could satisfy the same role in principle; as with the state store, that portability has not yet been demonstrated here.

---

8. Closing the loop: the failure signal as repair infrastructure

§1 framed verification as the bottleneck that bounds delegation: the useful autonomy of a code-writing agent is capped by what the organization can verify. The preceding sections built the verifier. This section states what the verifier buys beyond catching defects, because the gate's failure output—not only its pass claim—was designed as infrastructure.

8.1 The failure signal is mechanical and localized

When the gate refuses, it does not report that something regressed somewhere. It emits a durable, mechanical disagreement: the named table and business key where compared state diverged, the field-level mismatch values, the corpus case whose disposition was missing or doubled under its named error code, the retry classification that consumed a run, the out-of-closure write and its change class, the identity or provenance check that refused.

The shape of that artifact, illustratively:

```
MISMATCH  scenario: master-entity create · round 1
  table:  master entity
  key:    business key MK-0000
  field:  primary contact email
    expected: contact.a@example.invalid
    actual:   contact.b@example.invalid
  field:  billing contact email
    expected: contact.b@example.invalid
    actual:   contact.a@example.invalid
```

Illustrative rendering — not a verbatim run artifact. That is the whole of it. There is no diff to read and no expected value to reconstruct: the two mirrored fields are the transposition, and the scenario, business key, and round locate it. The companion reports the gate convicting a planted defect of exactly this shape (companion, Record IV) — The faults were all of one deliberately chosen kind, and the author reports his own run: the result shows what only the comparison could catch, not how often the gate catches faults in general.

Every refusal path terminates in an artifact rather than an impression. The interpretive alternative—a reviewer reading a diff and reconstructing the intended business values—produces a conclusion that remains in a reviewer's judgment. The gate's failure is located in an artifact that any subsequent process can consume.

8.2 The repair loop, and why it is safe against its own operator

A failure artifact that precise makes a diagnose-repair-rerun loop practical: the loop's operator receives the exact disagreement, changes the implementation, and reruns the standing gate. What makes the loop *safe*—rather than merely practical—is that the gate was constructed on the assumption that its operator might optimize against it.

§1 cited evidence that models modify tests, obtain reference answers, and exploit evaluator loopholes when a weak success signal makes those moves profitable. Under this construction those moves are closed by design rather than discouraged by

convention: the oracle, its scenario manifests, gold state, and command evidence are identity-hashed and outside the implementing agent's write authority (Rule 6); evidence cannot be minted, transplanted, or accepted across contexts without a logged operator act (gate conditions 6, 7, and 11); and every rerun re-verifies pre-state, runtime profile, and comparison closure from scratch.

A repair loop run against a weak signal launders defects into green. Against this gate, the loop retains several moves—the failing premise may be the write path, but it may equally be a scenario that asserts a contract the interface does not offer, a precondition, or the verifier's own machinery—and in the author's practitioner experience each of those has at some point been the correct repair. What the construction removes is not the existence of other moves but their cheapness. Fixing the write path is the only repair the loop can make on its own authority. Every other surface charges a toll: an oracle-affecting change requires independent approval and cannot travel with the implementation change (Rule 6); a change to what a scenario drives lands in the command-contract evidence that gates the Validate lane; and no repair of any kind counts until the deterministic driver re-witnesses it under its own lineage, so the loop cannot confer a verdict on itself.

The claim is therefore not that one door is open, but that every other door is alarmed. That assessment is bounded by the residuals this paper discloses. A loop can still profit from an interaction the coverage model omits, and, where the out-of-closure scan is left count-based as §3.1 permits, from a write class that scan cannot see.

What practitioners informally call self-healing is therefore not a convenience bolted onto the method; it is the intended consumption model of the failure signal — and it differs in kind from the self-healing already common above interface-level signals, because a repair loop can only ever heal to the standard of the verdict that stops it. A loop stopped by interface tests heals until the screens behave, and ships whatever those tests cannot see; a loop stopped by this gate heals until what was entered is what is stored, verified value by value against an oracle the implementing agent cannot edit. The companion demonstrates the gap operationally: seven persisted-state faults on which an interface-signal loop would never have opened at all, because its definition of done was already green (companion, Record IV). Verification strong enough to delegate to is verification strong enough to repair against.

8.3 The gate is deterministic; the agent is optional

Nothing in the loop places agent judgment inside the accept/reject path. The gate is versioned, deterministic orchestration: fixed gold, declared entry paths, mechanical comparison, fail-closed refusal. No model output is trusted as an oracle and no agent scores a result; removing every agent from the process leaves the gate's semantics unchanged. An implementation can therefore operate the loop at any point on an adoption gradient: a human consumes the failure artifacts directly—already cheaper in the author's practitioner experience than reconstructing expected values from a diff; an agent diagnoses and proposes while a human applies the fix; or an agent operates the full diagnose-repair-rerun loop, as the reference implementation does (§9). The gate is invariant across all three. In particular, the method is not AI evaluating AI output; it is a machine oracle that does not know or care who operates the loop above it.

8.4 Limits of this claim

The claim in this section is structural: the failure signal is sufficient input for a governed repair loop, and the governance makes optimizing against the gate unprofitable under the stated premises. It is not an empirical claim about repair quality. Convergence rates, loop cost, and thrash behavior are unmeasured here and belong to the follow-up study (§9). A repaired change remains subject to every form of human review that the gate does not replace (§7). And a loop that thrashes consumes runs and accumulates visible oracle debt—it never weakens the gate.

9. Practitioner experience and research agenda

The author has been developing and applying the method across two settings—a personal trading platform and an internal line-of-business platform. The reference implementation reported here is the latter; it includes:

- dependency-ordered scenario manifests and round-scoped proof chains;
- business-key comparison specifications with count gates and field-level mismatch artifacts;
- a Proof lane that evaluates a coverage-selected UI subset and captures the command contract;
- a Validate lane that processes the full in-scope corpus through the command entry point and refuses stale Proof artifacts;
- module-scoped baselines and oracle versions; and
- an agent-operated diagnose-repair-rerun loop in which the scenario manifests, gold state, and command evidence are identity-hashed and re-verified by the gate, so modifying them invalidates the standing proof evidence instead of silently accompanying an implementation change.

The method as specified here additionally requires a direct interface assertion wherever successful interaction and state parity do not establish a declared UI outcome.

The companion field report includes an exercised instance of that requirement, from a seeded-fault exercise: a planted inversion of a status gate offered a workflow's approval controls on exactly the wrong records — present once a record was approved, absent on the open records that should offer them. Because the action could not be taken where it was due, no write occurred, and state parity had nothing to compare—what failed the run was the declared interface obligation, whose assertions the scenario states in both directions: the control must be present on an open record and absent on an approved one. A defect that suppresses an action is invisible to any check that only compares what was written; the interface obligation is the paper's answer to that class, and the exercise shows it failing a run for exactly that reason.

The reference implementation conforms to the specification as published; the closures that follow are enforced by tests rather than asserted. As of publication:

- Comparison closure fails closed at the table level as well as the column level: every business table must be compared, classified with a written reason and category, or delegated, and an unclassified table fails the run.
- Runtime-profile capture includes feature flags and the execution-affecting launch environment, canonicalized and hashed; the profile is verified at run start and re-verified at the final observation point, independently on both sides of the capture boundary, so a result cannot be attributed to a profile that no longer held at either observation. Condition 12 further requires that volatile elements be held immutable or monitored across the whole evaluated interval; that requirement is met for the elements the profile captures, and a change to an element outside that capture which occurred and reverted between the two observations would not be detected.
- Non-vacuity is enforced through per-scenario declared witnesses—expected differences carry minimum row deltas, declared unchanged tables are verified by both row count and a row-content state hash, so an in-place update is caught, and refusals require durable evidence—and, in the Validate lane, through per-case dispositions as described below. Enforcement runs after the run completes, with independent re-enforcement on the replay path.
- Evidence identity binds the scenario-manifest digest, the build identity, the comparison-contract closure, the runtime-profile digest, the command-contract digest, and both the locator and the intrinsic identity of the state store: a locator can be redirected; the identity cannot.
- Evidence provenance is witnessed out of band. A lineage record maintained outside any channel the artifact travels through must witness the minting run; both lanes refuse unwitnessed evidence, and acceptance of evidence minted in

another context is an explicit, logged operator act. The mechanism closes a concrete forgery class: a passing evidence artifact that travels across contexts through ordinary version control arrives still satisfying every identity, freshness, and build binding—exactly the class the provenance premise of §2.2 exists to close.

- The replay credential is minted fresh per run, deleted on every exit path, and refused past a bounded age; no standing credential survives a run.
- The only-writer premise is checked per attempt by the out-of-closure write scan of Rule 2, content-sensitive by default: every business table's row count and an order-insensitive row-content digest are snapshotted around the flow, so an in-place update or a balanced delete-and-insert outside the closure is a violation rather than remaining invisible. Narrowing any table to count-only verification is an explicit, reviewed opt-down recorded with a written reason, and the review found no table that currently needs it.
- The command-contract shape rules are enforced by two independent implementations on opposite sides of the capture boundary, cross-pinned by shared test vectors, so a modified or defective runner cannot skip them.

The specification has been implemented twice. A second harness, written against the specification rather than derived from the first, also conforms. The independence claimed here is of the code, not of the author: both harnesses are the author's, and neither has been implemented or specification-tested by an independent party. What the second implementation establishes is narrower than corroboration and still worth stating—a construction built a second time from its written rules is shown to be followable as written, rather than being a description of one implementation that already existed.

A fault-injection exercise has been run against the hardened implementation, reported in full in the companion. Single-token faults were planted by the operator in application code as uncommitted mutations, and the run executed in a commit-free mode that mints no evidence reaching the mainline and cannot gate the other lane—so an exercise that plants defects in the system under test cannot contaminate the evidence chain. The mode's contract bars version control as a diagnostic instrument—no diff, status, history, or inventory of modified files—so each fault had to be convicted from failure artifacts, then observed behavior, then the source as it stands. The operator's injection manifest records six faults planted; all six were found and none survived. They were repaired across five chain stops: four convicted by a stop, one convicted after its neighbor's repair made it reachable—two faults stacked in one scenario, peeled in order, with the scenario's passing-test count advancing at each iteration as evidence the earlier fix held—and one found by a peer sweep performed before the scenario that would otherwise have exercised it ran. A second exercise then aimed the injection at the write path itself: seven faults of a single class — the create-path mapping disagreeing with its sibling update-path mapping about one field, as a discarded input, a fabricated value, a forced default, or a transposed pair. Every interface-level test passed in all seven cases, because the tests assert the submitted command and the resulting row count and both were correct; and since the update path mapped each field correctly, a later edit would have silently healed the row. The full-row, full-value comparison against gold supplied all seven gate convictions — no interface, contract, validation, or read-path check convicted any of them — and completeness was machine-checked rather than attested: with all seven repairs applied, the working tree was byte-identical to the committed baseline, so exactly seven mutations existed and none survived. Thirteen faults across two exercises are still not a sample and no detection rate should be inferred; both planted classes were narrow and single-line by design — the second deliberately so, to isolate the parity claim — and the results establish nothing about multi-line faults, faults distributed across components, or faults planted in the oracle rather than the application.

Three further closures are each enforced by a regression test demonstrated to fail when the enforcement is removed. Witness granularity in the Validate lane is now per case, as gate condition 3 obliges: every corpus case must resolve to exactly one recorded disposition—accepted and tied to compared state under its business key, rejected with a durable evidence record, or a declared no-change naming its unchanged observable—with the accounting kept independently on both sides of the capture boundary and reconciled after completion; an undispositioned or doubly-dispositioned case fails the run under a named error code, so one changed record can no longer vouch for vacuous siblings.

The no-change leg of this accounting is enforced but not yet exercised: the current corpus contains no deliberate no-change case, and the refusal and idempotency rounds that would supply them remain future coverage.

Retries follow the policy of §7 as written: a failed attempt is retried only when its preserved evidence positively matches a named infrastructure failure class, an application-behavior marker in the same evidence dominates any such match, an unclassifiable failure consumes the run, and every permitted retry is written to an append-only ledger as an oracle work item.

The broad out-of-closure write scan is content-sensitive by default, closing the count-based residual; the measured cost of the per-table content digest on the working store was small enough that no table currently exercises the reviewed count-only opt-down.

This experience motivates the proposal; it is not offered as a controlled experiment. The companion field report deposited alongside this note supplies part of what a fuller empirical account would need: scenario and level counts, clean full-chain and full-session runtimes in both lanes on a single commit, the Validate lane's replay volume with its reviewed-scope exclusion and deferral inventories and their reasons (exit criteria are held in the implementation's reviewed scope ledger), and seeded-fault results on stated fault classes. Those measurements are a single practitioner's record of one implementation, made on one day on one hardened build, not a study. What remains outstanding for a follow-up report is the part that record cannot supply: comparator cost and its scaling with corpus size under controlled conditions, escaped-defect rates, diagnosis time measured rather than narrated, oracle-maintenance effort over time, the proportion of implementation changes that require oracle changes, and any of it produced by someone other than the method's author.

A reader who wants to try the construction before adopting it will find a deliberately bounded **proving slice** described in this record's `README` and on the project site — one write path, a small frozen gold fixture held outside the implementation's write authority, and a full-closure fail-closed comparison — offered as a way in, and explicitly not as a conforming EDD gate.

The research agenda is therefore:

1. **Coverage maturity.** Compare pairwise, mixed-strength, risk-directed, and change-aware Proof selection; track which faults each exposes.
2. **Fault-injection calibration.** Maintain a representative fault model across UI mapping, validation, command construction, persistence, events, audit, and refusals. Every surviving injected application fault becomes evidence for a new oracle obligation.
3. **Reverse EDD.** Drive the application to render gold state and compare DOM-level values, completing the read-path twin of the write-path gate.
4. **Lifecycle and temporal rounds.** Extend deterministic rounds across every supported state transition and then into controlled clocks for future-time behavior.
5. **Concurrency.** Add interleaved actor scenarios with expected states for duplicate submission, simultaneous edits, and conflict handling.
6. **External-effect verification.** Join durable intent parity to delivery receipts or controlled endpoint assertions without weakening the state boundary.
7. **Portability.** Evaluate Gold-State Parity™ over non-relational stores, structured files, and mixed persistence architectures, and Proof lanes driven through non-browser interfaces.
8. **Economics.** Compare total oracle cost, review effort, gate latency, and escaped defects against review-centered development over time.

The method's most important unresolved problem is not whether equality can identify a difference. It is how rapidly a living oracle can mature toward the interactions that matter without becoming as expensive as the review process it is meant to relieve.

---

10. Conclusion

Expectation-driven development treats verification as infrastructure rather than a final human activity. Its central mechanism is simple to state but strict to implement: hold a trusted reference state fixed; derive both stimulus and expectation from it; verify each starting state; require each lane to use its declared official door; assert the selected interface obligations; compare the resulting state under a contract that fails closed; persist or explicitly delegate every material effect; and prevent the implementing agent from rewriting the oracle or the evaluated application from reading the answer at runtime.

The two-lane design is a deliberate scalability compromise. Proof uses the real interface for a coverage-selected subset and directly asserts the interface outcomes required by that model. Validate uses the current build-bound UI-produced command-contract evidence to carry the full in-scope corpus through the official command path. A pass therefore means exactly what the architecture supports: under the identified runtime profile, the selected browser traces satisfied their declared interface assertions, and those traces plus the full in-scope command corpus reproduced the expected contract-defined state for the identified oracle version. It does not mean every possible UI interaction was exercised.

That coverage boundary is not hidden. Pairwise selection is a practical minimum; higher-order and unmodeled interactions remain the principal residual risk; and the proof surface is expected to evolve from `v1` to `v2` and beyond. The residual is a coverage-selection economics boundary rather than a capability boundary: any deterministic interaction that satisfies the applicability conditions of §2.1 can be brought in scope, at any order, by preparing the gold that expresses it, and the open question is when the testing benefit justifies that preparation cost. Gold is immutable within each version; the oracle matures across versions.

The parts are old: stored oracles, state comparison, capture and replay, parallel reconciliation, and independent evaluation. The proposed contribution is the machine assembled from them—a standing, fail-closed, independently governed acceptance gate for agent-written stateful write paths. Its promise is not perfect software. Its promise is narrower and operationally valuable: under the gate's stated premises, a covered write-path flaw that changes compared state surfaces as a mechanical, localized disagreement before release, subject to the stated cancellation exception. Whether the resulting scale and economics outperform review-centered development is an empirical question reserved for the follow-up study.

What that buys downstream is the difference between two kinds of self-healing, because a repair loop can only heal to the standard of the verdict that stops it. The companion field report puts the gap on the record: seven seeded create-path faults that every interface-level test passed, that no contract, validation, or read-path check convicted, and that the full-row comparison against gold convicted all seven times — one narrow fault class, author-reported, not a rate.

Reduced to an instruction, the method is three words: make *done* demonstrable. The rest of this paper is what it costs to mean them.

Publication information and disclosures

- **Author:** Melvin Fahnestock
- **Version:** 1.0

- **Publication date:** August 2, 2026
- **DOI:** 10.5281/zenodo.21761800
- **Canonical project site:** https://expectationdrivendevelopment.com
- **Contact:** mel@expectationdriven.com

Preferred citation

> Fahnestock, M. (2026). *Expectation-Driven Development: Gold-State Parity™ as a Write-Path Acceptance Gate for Agent-Written Stateful Applications* (Version 1.0). Zenodo. https://doi.org/10.5281/zenodo.21761800

Disclosures

Publication status. This is an independently published method proposal and practitioner experience report. Version 1.0 has not undergone journal or conference peer review.

Competing interests. The author developed Expectation-Driven Development and may publish, teach, license, or commercialize related materials or services.

Trademark. Gold-State Parity™ is a trademark of Melvin Fahnestock, claimed for the mechanism this paper defines. Section 2(b) of CC BY 4.0 expressly excludes trademark rights from the license granted here, and the author reserves them. This reservation is about the name, not the technique: the method remains free to implement, describe, teach, and write about, including by name in the ordinary descriptive and referential ways, and no permission from the author is needed to do so.

Funding. No external funding was received for this work.

Implementation and data availability. The reference implementation, regression tests, and operational datasets described in the practitioner experience are proprietary and are not included in this publication. No research dataset accompanies this method proposal and practitioner experience report.

Generative-AI assistance. The method and its claims are the author's own. Generative AI tools assisted with drafting and revising the manuscript text, reference checking, and structured critique passes during preparation. The author defined the thesis and its claim boundaries, adjudicated the resulting output, and accepts responsibility for the final publication. These critique passes were part of manuscript development and do not constitute peer review or independent validation.

Client confidentiality and endorsement. No client or customer is identified in this publication, and no client or customer endorsement is implied.

References

- Amazon Web Services. (n.d.). *Transactional outbox pattern*. AWS Prescriptive Guidance. Retrieved August 2, 2026.
- Approval Tests. (n.d.). *Approval Tests: Capturing Human Intelligence*. Retrieved August 2, 2026.
- Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering, 41*(5), 507–525.

- Blackburn, M. R., Busser, R. D., Nauman, A., & Chandramouli, R. (2002, September 3). Interface-driven model-based generation of Java test drivers. *Quality Week 2002*.
- Brisset, S., Rouvoy, R., Seinturier, L., & Pawlak, R. (2022). Erratum: Leveraging Flexible Tree Matching to repair broken locators in web automation scripts. *Information and Software Technology, 144*, 106754.
- Chays, D., Deng, Y., Frankl, P. G., Dan, S., Vokolos, F. I., & Weyuker, E. J. (2004). An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability, 14*(1), 17–44.
- Davis, M. D., & Weyuker, E. J. (1981). Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference* (pp. 254–257).
- DbFit. (n.d.). *Reference*. Retrieved August 2, 2026.
- dbUnit. (2026, May 11). *Getting started*.
- Faros Research. (2025, July 23). *The AI Productivity Paradox Report 2025*.
- Gala, S., Fernandez-Ivern, J., Rokkam Pratap, A., & Shah, D. (2023, May 4). *Migrating Critical Traffic at Scale with No Downtime—Part 1*. Netflix Technology Blog.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues?. *The Twelfth International Conference on Learning Representations*.
- Kindelberger, Y., Rajagopalan, S., & Singh, S. (2024, September 27). *Automate the building, testing, and deployment processes of AWS Mainframe Modernization with AWS Blu Age*. Amazon Web Services.
- Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010, October). *Practical combinatorial testing* (NIST Special Publication 800-142). National Institute of Standards and Technology.
- Laforgia, A. (2026, March 21). *Expectation-Driven Development: A Validation Framework for the Age of AI Agents*.
- Monperrus, M. (2018). Automatic software repair: A bibliography. *ACM Computing Surveys, 51*(1), 1–24.
- OpenAI. (2024, August 13; updated 2025, February 24). *Introducing SWE-bench Verified*.
- OpenAI. (2026, February 23). *Why SWE-bench Verified no longer measures frontier coding capabilities*.
- Oracle. (n.d.). *Parallel test checklist*. PeopleSoft Payroll for North America documentation. Retrieved August 2, 2026.
- Pact Foundation. (2022, November 7). *Conceptual overview*.
- Redgate. (2025, December 10). *What's a comparison key?*. *SQL Data Compare 16* documentation.
- Revolut. (2020, May 20). *Under the hood: Engineering at Revolut*.
- Von Arx, S., Chan, L., & Barnes, B. (2025, June 5). *Recent Frontier Models Are Reward Hacking*. METR.
- Wang, Y., Dillig, I., Lahiri, S. K., & Cook, W. R. (2018). Verifying equivalence of database-driven applications. *Proceedings of the ACM on Programming Languages, 2*(POPL), Article 56.
- Wei, J. (2025, July 15). *Asymmetry of verification and verifier's rule*.
- Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal, 25*(4), 465–470.
- Xie, Q., & Memon, A. M. (2007). Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology, 16*(1), Article 4.
- Yeh, S.-Y. (KohakuBlueleaf). (2025). *Expectation Driven Agentic Development (EDAD)*.