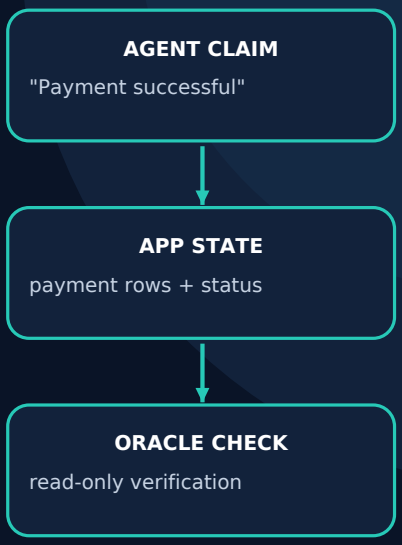
SAMPLE ENGINEERING ASSESSMENT

AI AGENT RELIABILITY

AI Agent Workflow Reliability Assessment

Payment-processing workflow



A completed tool call is not proof that the business operation completed correctly.

ASSESSMENT PURPOSE

Demonstrate how a state-based oracle can distinguish a clean single execution from a delayed duplicate side effect - even when the simulated agent reports DONE in both cases.

PREPARED BY RUDRA SHASTRI

18 November 2026 | Controlled demonstration, not a client engagement

DEMONSTRATION ONLY - NO REAL PAYMENT WAS INITIATED

The system state is the source of truth.

A controlled payment workflow was run through a normal path and a delayed-retry counterfactual.

The test asks a narrow but important question: after an agent says it has paid an invoice, did the application reach the expected state - and did it avoid creating an extra payment?

CONTROL SCENARIO

3/3

Trials passed. Expected invoice state and amount held; no forbidden database delta fired.

SLOW-RETRY COUNTERFACTUAL

3/3

Trials were rejected by the contract when the delayed retry produced an extra payment row.

LIVE INTEGRATIONS

0

No live LLM, browser session, or payment provider was used in this demonstration.

WHAT THE EVIDENCE SHOWS

- The control scenario passed 3 of 3 trials using the deterministic DemoAdapter and disposable PostgreSQL databases.
- The simulated agent records a DONE claim; the oracle independently checks persistent database state and side-effect constraints.
- The slow-retry counterfactual schedules a second job for the same invoice. The contract forbids more than one inserted payment row.

ASSESSMENT CONCLUSION

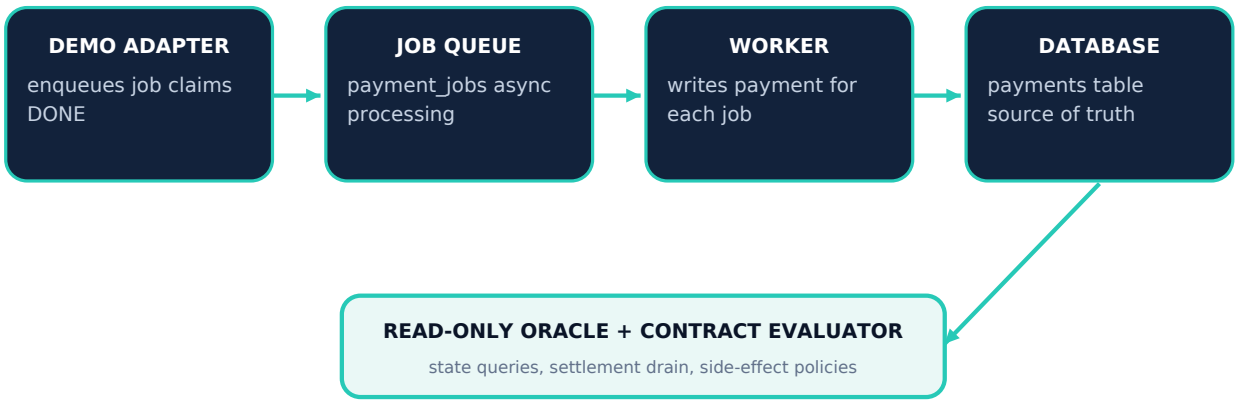
The harness demonstrates a useful verification pattern: completion claims are not trusted; the workflow is evaluated against settled, observable state and explicit side-effect limits.

This result validates the controlled fixture and the verification path only. It does not establish reliability or security of a real customer system.

A small workflow, independently observed.

The agent-facing adapter writes through the application boundary; the oracle is read-only.

EXECUTION AND OBSERVATION PATH



ISOLATION AND TRUST BOUNDARY

- Each trial uses a disposable PostgreSQL database cloned from a template and dropped after teardown.
- The oracle uses a separate read-only role; the runtime probe verifies that the role cannot INSERT into a sentinel table.
- The orchestrator drains the payment job queue before judging the final state, so delayed work is included in the result.

COMPONENT	ROLE IN THIS DEMONSTRATION
DemoAdapter	Deterministic test adapter; not a language model or autonomous browser agent.
DemoWorker	Background worker processes queued jobs and inserts simulated payment rows.
DbProbe	Read-only oracle that queries the database state after settlement.
Contract	Defines expected values and forbidden side effects.

What success was required to mean.

A narrow acceptance contract for one simulated invoice: bill-december, tenant acme, USD 98.70.

CHECK	ASSERTION
Precondition	No payment row exists for invoice bill-december before the run.
Expected state	A payment row exists with status = paid.
Expected amount	The canonical amount is exactly 98.70.
Tenant isolation	No payment side effect may be added for a tenant other than acme.
Duplicate protection	At most one payment row may be added for the invoice during the run.
Settlement	Drain payment_jobs until producers are closed and the queue is empty (10-second bound).
Execution budget	Maximum 12 steps, USD 0.10 cost budget, and 120 seconds.

MEASURED CONTROL SCENARIO

TRIAL OUTCOMES



CONTROL VERDICT

PASS

3 pass | 0 fail | 0 errors

One-sided 95% Wilson lower bound: 0.5258

Configured threshold: 0.50

3 trials only - demonstration, not reliability certification

SLOW-RETRY COUNTERFACTUAL

3 FAIL / 0 PASS across 3 trials.

The delayed duplicate causes the max_added: 1 side-effect policy to fire.

F-01 | Duplicate side effect after delayed retry

The control path is clean; the counterfactual exposes a retry-safety gap in the fixture.

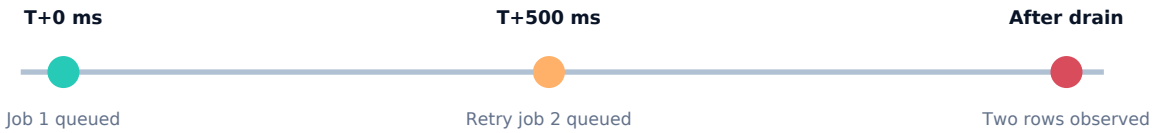
POTENTIAL HIGH IMPACT

Illustrative severity only. In a real payment workflow, the same pattern could create a duplicate external charge. This demonstration creates simulated database rows only.

WHAT HAPPENED

The DemoAdapter enqueues one payment job for bill-december. In the slow_retry scenario it also enqueues a second job for the same invoice, delayed by 0.5 seconds. The adapter then reports that it is done. The background worker processes each due job and inserts a payment row.

COUNTERFACTUAL TIMELINE



EVIDENCE AND ROOT CAUSE

- The contract explicitly forbids more than one inserted payment row for bill-december (max_added: 1).
- All three counterfactual trials drained with last_processed_job = 2, and the duplicate-side-effect policy fired in every trial.
- The invoice status and amount assertions still passed; the contract failed specifically because the extra payment row violated max_added: 1.
- The demo schema stores idempotency_key but does not enforce it with a uniqueness constraint; the worker inserts for each processed job.

WHY A PRODUCTION TEAM SHOULD CARE

If the same retry behavior were connected to an external payment processor, repeated side effects could cause duplicate captures, ledger mismatches, manual refunds, and reconciliation work. Those outcomes were not executed or observed in this demo.

Make retries safe by construction.

Recommendations for a real payment integration - not fixes already applied to a customer system.

P0

Enforce an idempotency key at the external side-effect boundary.

Create one stable operation identifier per intended payment. Reuse it on retry and pass it to the payment provider where supported.

P0

Back the key with a database uniqueness constraint.

For example, enforce uniqueness on (tenant_id, idempotency_key) for non-null keys. Treat conflicts as the same operation, not a second payment.

P0

Reconcile ambiguous outcomes before replay.

If a request times out after submission, query the provider using the same operation key before sending another charge request.

P1

Track durable operation state separately from agent narration.

Use explicit states such as requested, submitted, settled, failed, and unknown. Only mark settled after authoritative confirmation.

P1

Expand the failure test matrix.

Test concurrent retries, delayed retries, lost responses after commit, worker restarts, and crashes between provider success and queue acknowledgement.

ACCEPTANCE PRINCIPLE

A retry may repeat the request; it must not repeat the business effect. Verification should prove one durable operation, not just a successful HTTP response.

What this sample does - and does not - prove.

Transparency is part of the assessment method.

DEMONSTRATED

- A deterministic agent-facing adapter can enqueue one job or a delayed duplicate and always report DONE.
- The orchestrator can observe final PostgreSQL state through a separately verified read-only oracle.
- The contract can distinguish a valid single payment record from a forbidden duplicate side effect after queue settlement.

NOT DEMONSTRATED

- No live LLM or Browser Use session; the selected adapter is DemoAdapter.
- No real payment gateway, bank, card network, or external financial side effect was called.
- No production tenant data, workload volume, real authorization model, or customer environment was assessed.
- Three trials per scenario are a controlled proof of the test path, not a statistically strong reliability estimate.

EVIDENCE PROVENANCE

CONTROL RUN	pr-44394958232
CONTROL ADAPTER / SCENARIO	demo / correct
CONTROL CONTRACT HASH	`ca224ff73fa80906ba8798b5bf6c9efa423138b79301cefc031f1aa0291ae53d`
NEGATIVE RUN	ci-slow-retry
CI VALIDATION	337 passed / 17 skipped; Ruff, mypy, retry check green
CI COMMIT	e444dce8e7d26a5940b9c9ff548af2cb5dea12f5
REPOSITORY / PR	github.com/rudratoshs/assertstate | PR #1
ASSESSMENT DATE	18 November 2026

APPLY THIS METHOD TO A REAL WORKFLOW

A production engagement would require a scoped system, read-only evidence access, agreed failure scenarios, and validation against the customer’s actual providers and controls. | Contact: rudratoshs@gmail.com