Replay Verification: Re-Deriving the Decision Instead of Reading the Log

Intentrax Labs Research Paper | Replay Verification
Published 2026-07-09 - Intentrax Labs - www.intentrax.com/labs

The method by which an independent verifier reconstructs an execution context, re-evaluates policy, and confirms the result by hash equality rather than trust.

Reading a log versus re-deriving a decision

Conventional assurance reads a record and trusts the writer. Replay verification does the opposite: it reconstructs the execution context from stored evidence, re-evaluates the governing policy, and confirms the outcome by comparing recomputed hashes against the hashes stored in the proof. The verifier does not ask the executor what happened; it derives what must have happened from the evidence and checks that the derivation matches the proof. Replay verification is a first-class capability of the protocol, not an add-on. Its guarantee is precise: identical inputs must produce identical results, and any divergence is treated as verification failure rather than as noise to be reconciled.

The five-step replay contract

The replay verification contract has five ordered steps. First, retrieve the stored proof artifact. Second, retrieve the execution and policy snapshots, that is, the Execution Event Object and the Policy Object Definition bound to the proof. Third, reconstruct the execution context from those snapshots. Fourth, re-evaluate the policy engine against the reconstructed context. Fifth, compare the computed hashes with the stored proof hashes. Because the policy evaluation VM operates without external dependencies and all policy evaluation is deterministic, step four yields the same decision object every time. Step five converts that decision into a binary integrity check: the proof either recomputes to the same hash or it does not.

Why replay must not depend on mutable state

Replay verification must reconstruct the original execution context and produce the same evaluation result without relying on mutable system state. This is what separates it from re-running the agent. The verifier does not call the model, execute tools, contact the network, or read a live clock; it re-derives the decision from immutable snapshots. Evaluation timestamps and other operational fields are recorded but excluded from canonical hashing, so wall-clock differences at verification time cannot change the result. The deterministic execution model forbids randomness, time and environment dependency, and external I/O precisely so that replay is possible; the deliberate cost is reduced flexibility inside the engine in exchange for a system that can be checked by anyone, later, offline.

The independent, offline verifier

AEVN requires that verification be independent and reproducible externally, which is why the reference verifier runs standalone. It recomputes hashes and fails closed on authority drift, and it does so without runtime access, network access, or private key material. The production evidence-bundle verifier, for example, reads only repository-local JSON files and performs SHA-256 and canonical JSON recomputation; it does not import engine packages, call a runtime service, open the network, or require key material. This is the operational meaning of trustless verification: the party checking the proof needs the evidence and the published rules, not a relationship with the executor. The tradeoff, accepted by design, is that canonical requirements must be strict enough to guarantee cross-system reproducibility.

Fail-closed outcomes and drift classes

Divergence is not smoothed over; it is classified. The standalone verifiers lock negative corpora that fail closed on distinct conditions, including recanonicalization of locked proof bytes, mutation of source event or payload bytes, resigning or key-reference substitution, registry re-admission or append-only record rewrite, verifier-handoff hash drift and attempted repair, revoked or unknown signer state, and provider attestation drift. Each is a separate fail-closed classification rather than a generic error. A verifier result may confirm that an exported bundle is hash-bound and offline-verifiable while still reporting that production is blocked; hash validity and go-live authorization are deliberately different claims, and the verifier never converts one into the other.

Current guarantees and planned extensions

Today the pipeline is deterministic end to end: the same execution yields the same proof hash, the engine has begun signing real proofs, and the offline verifier enforces the properties above. Cross-language consistency is a stated requirement: canonical rules must match across Go, Python, and TypeScript implementations, and an implementation that produces different hashes or serialization is non-conformant. Planned and labeled as such: graded Assurance Levels 1 through 4 will layer stronger evidence and operational guarantees on top of the same replay contract, and an Intentrax Ledger and trust-network are planned as a persistence and discovery layer. Live production traffic and live mutation remain blocked until a separate, independently verified production execution receipt exists.