The Autonomous Execution Proof: A Technical Whitepaper

Intentrax Whitepaper | Execution Assurance
Published 2026-07-09 - Intentrax Labs - www.intentrax.com/labs

The data structure, canonicalization and hashing model, decision logic, and offline verification contract behind the AEP.

Executive summary

The Autonomous Execution Proof (AEP) is the canonical cryptographic artifact produced by the Intentrax Assurance Engine under the AEVN protocol. It is a tamper-evident, replay-verifiable, audit-ready record of a single autonomous execution event. This paper specifies, at an architectural level, how an AEP is derived, why it is deterministic, how integrity is preserved through canonicalization and hashing, how the governing decision is computed, and how an independent verifier confirms the artifact offline. The intended reader is technical: a security engineer, a platform architect, or a technically literate auditor who needs to understand the primitive well enough to trust - or to challenge - it. Machine-readable schemas and golden vectors are the authoritative source; this document is an explanation downstream of them and does not redefine any locked contract.

From internal trace to external proof

Intentrax separates two artifacts by design. The Execution Evidence Object (EEO) is the internal trace of an execution. The AEP is the external proof derived from it. The EEO exists to capture what happened; the AEP exists to let someone who was not present, and who does not trust the executor, establish what happened. Keeping these distinct is a deliberate architectural choice: it adds a normalization layer but yields portable verification that does not leak internal runtime structure and does not require runtime access to check. An AEP lets an independent verifier determine: what execution evidence was normalized; which principal and authority were bound; which policy version governed the event; which decision was produced; how integrity and replay references were preserved; and whether the proof was subsequently altered.

Determinism as a precondition

Replayable verification is only possible if execution is deterministic. The engine follows a deterministic model: no randomness, no dependence on wall-clock time or environment, and no hidden external I/O inside the decision path. The engine core is implemented against the standard library with no external runtime dependencies, which narrows the surface for non-determinism and supply-chain variance. The consequence is a strong invariant: the same execution inputs produce the same proof hash, independent of when or where the derivation runs. This is what makes cross-implementation and cross-language consistency meaningful - a conforming

verifier in another language must arrive at the identical canonical bytes and therefore the identical hash. Determinism is not a performance choice here; it is the property that makes independent replay possible at all.

Canonicalization and the proof hash

Integrity rests on a canonical encoding. Proof content is serialized with ordered keys and a canonical JSON encoding so that semantically identical content always produces identical bytes; the canonical bytes are hashed with SHA-256 to produce the proof hash. Because the encoding is canonical rather than incidental, two systems that agree on the content cannot disagree on the hash. The proof hash is then signed, binding the content to a signing authority and making any post-hoc edit detectable: alter a field, and the recomputed hash no longer matches the signed value. The cost of this approach is a strict, custom canonicalization requirement rather than relying on a language's default JSON serializer - a cost the design accepts in exchange for cross-system determinism. Canonical JSON remains the machine-authoritative form; human-readable companions (certificate, verification report) are required for usability but are explicitly non-authoritative.

The decision model: fail-closed dual gate

The governing decision is computed, not asserted. An action is ALLOWed only if the execution evidence gate and the authority gate both permit it; if either withholds permission, the result is DENY. This dual-gate, fail-closed model biases the system toward denial in ambiguous states, which produces more DENY outcomes than a permissive design but avoids false ALLOWs on incomplete or drifted evidence. Authority is bound explicitly: the proof records the exact policy version that governed the event together with the identity, delegation chain, and constraints under which the action was permitted. Because the decision is re-derivable, a verifier does not read a stored ALLOW/DENY conclusion and take it on faith; it recomputes the decision from the recorded inputs and the referenced policy version, and confirms it matches.

Registry and portable evidence envelope

Recorded proofs are written to an append-only registry with checkpoints. The registry is immutable by design: entries are added, never silently rewritten, which is what makes historical auditability possible at the expense of monotonic storage growth. Each AEP can be packaged into a portable evidence envelope - the canonical AEP plus an Agent bill-of-materials (model, prompt, tool, policy, authority, and delegation references, without requiring raw prompt disclosure) and an Execution bill-of-materials (execution and tenant identifiers, EEO hash, AEP hash, state-machine and step-chain references, and registry references). The envelope is explicitly portable proof only. It carries the human-readable certificate, verification report, and README alongside the machine-authoritative canonical form, and it is designed so a downstream verifier needs no origin-runtime access, no private key material, no network lookup, and no policy re-evaluation service to check it.

Offline verification contract

The standalone verifier defines the trust boundary in operational terms. It uses only repository-local JSON files and SHA-256 recomputation. It does not import the engine's Go packages, call a runtime service, open the network, or require private key material. It recomputes artifact hashes, binds the offline proof-verification report, re-derives the decision, and confirms the evidence bundle is self-consistent - and it fails closed if any artifact is missing, hash-drifted, or misinterpreted as a stronger claim than it makes. The verifier is deliberately conservative about what a pass means. In the current production-beta posture, production-facing evidence bundles resolve to a status that explicitly holds production authority flags - production-ready, production-authorized, runtime-access-required, network-access-required, private-key-material-required - fixed to false. Passing the evidence-bundle check is an integrity verification, not a production go-live authorization. That separation is intentional and is enforced by a negative-vector corpus covering drift, overclaim, and authorization-boundary abuse.

Boundaries, stage, and how to engage

The AEP establishes execution, policy version, bound authority, integrity, and replayability. It does not certify business correctness or regulatory compliance, and this system makes no claim to guarantee compliance, eliminate risk, or provide total security. Ingested external evidence is bound to a proof as an execution passport but never replaces the AEP. AEVN is an open protocol and Intentrax is its reference implementation, not its owner. The engine has recently begun signing real proofs and the offline verifier is operational; Assurance Levels 1-4 and a wider trust network are planned and labeled as such; live production traffic and mutation are currently blocked. To evaluate the primitive directly, run the standalone verifier against a published golden vector, or integrate an adapter with npx @intentrax/sdk init <adapter>. Deeper technical review against the locked schemas and conformance vectors is available to design partners.