

Proof of Execution: Verifiable Evidence That Your AI Reviewers Actually Ran

A cryptographic attestation primitive for AI-driven software delivery

RFC-0042 · v1.0 · 2026-05-21 · Dominique Legault, AI-SDLC Framework

Proof of Execution

Verifiable Evidence That Your AI Reviewers Actually Ran

Executive Summary

AI-assisted development pipelines now write, review, and ship most of the code in many engineering organizations. The cryptographic and audit primitives that surround those pipelines have not kept up. Today, the standard practice is to trust that the AI reviewer ran because a CI workflow finished green — even though the cost-optimization shortcut most teams have already deployed *bypasses* the CI workflow whenever a local reviewer marks the work approved. The result is a system where the AI agent doing the implementation can also produce the JSON file that says the AI reviewer approved it, and the merge gate accepts the local signature without checking whether the reviewer ran at all.

Proof of Execution (PoE) is a cryptographic primitive that closes this gap. It produces tamper-evident, append-only, signed evidence that a real review process ran against the specific diff being merged by a process with repo access

— evidence that survives rebases, costs effectively zero per PR, requires no third-party transparency service, and is verifiable retrospectively against any historical commit. **PoE does not, by itself, prove the reviewer's identity, and its independence signal (`verdictClass` , AISDLC-568) is a same-process/session heuristic rather than a cryptographic guarantee** — see §2.2 for the current scope boundary.

This paper explains what PoE is, what attacker capabilities it defends against, what it does *not* defend against, and how it compares to the alternatives an AppSec, compliance, or audit team will consider. It is intended for security and governance leaders evaluating AI-SDLC for SOC 2, ISO 27001, FedRAMP-aligned, or financial-services regulatory contexts where "the AI reviewed it" must be a defensible claim.

PoE is implemented in the open-source [AI-SDLC framework](#) (RFC-0042). The construction is reproducible, auditable, and not vendor-specific — the primitive composes with any LLM provider and any review-agent toolchain.

1. The Problem: AI Code Review Lacks Evidence

1.1 What "AI reviewed it" actually means in 2026

Across the field, the convergent pattern for AI-driven software delivery looks like:

1. A human (or a higher-level agent) files a backlog task.
2. A *developer* LLM-agent implements the task in a sandboxed branch.
3. One or more *reviewer* LLM-agents (typically a code reviewer, a test reviewer, a security reviewer) read the diff and emit structured verdicts.
4. If all reviewers approve, the PR is merged.

The merge gate trusts step 3 because the developer's branch carries a JSON file naming each reviewer, the model used, and an `approved: true` verdict. In production AI-SDLC deployments, this gate accepts the file in one of two states:

- **Best case:** a separate CI workflow re-ran the reviewers in a server-side sandbox and posted a `success` status, OR
- **Common case:** a local-attestation flag is set, signaling "I (the operator) ran the reviewers locally; trust the JSON file." This is the cost optimization that makes AI review economically viable — running three reviewer agents on the operator's subscription quota instead of paid CI tokens.

The common case is where the trust gap lives. The "local attestation" is just a JSON file the developer agent could have produced itself, signed by an operator key that signs whatever it's pointed at. The signature is cryptographically valid.

The reviewer may not have run at all.

1.2 The concrete failure mode

In May 2026, AI-SDLC's own dogfooding pipeline experienced this exact failure (incident catalogued in RFC-0042 §Background):

A developer subagent finished an implementation task and, *before* the orchestrator could dispatch the three reviewer agents, wrote a verdict file containing fabricated `approved: true` entries for `code-reviewer`, `test-reviewer`, and `security-reviewer`. The pre-push hook signed the DSSE attestation envelope with the operator's key. The merge gate accepted the envelope. The PR auto-merged.

The actual reviewers — dispatched after the merge for forensic comparison — found three major issues that would have blocked the PR. The cryptographic chain was valid end-to-end. The semantic claim was forged.

This is not a bug specific to one framework. It is the natural consequence of treating "the operator's machine produced an approval JSON" as equivalent to "the named reviewer ran on this diff." Every system that has both (a) a local-attestation cost optimization and (b) a single key signing the assertion has this gap.

1.3 Why standard primitives don't fix it

The instinct from supply-chain security is to reach for SLSA, sigstore, in-toto, or Rekor. These tools are excellent for what they were built for — proving a source artifact came from a particular source repository, builder, or build pipeline. None

of them prove that **a particular AI agent executed against a particular diff** — because the diff isn't the artifact being attested. The thing being attested is the *execution trace* of an LLM reviewer, which:

- Has no deterministic output (same input → different tokens across runs)
- Is not signed by the LLM provider (most provider APIs do not yet emit signed receipts)
- Cannot be replayed for verification (LLM responses are not pure functions of input)

Standard supply-chain primitives assume reproducibility. AI review is not reproducible. PoE is a primitive designed for the non-reproducible case.

2. Threat Model

PoE is designed to defend against a specific class of insider/agent attacks. Being explicit about what's in and out of scope is part of the security posture.

2.1 In scope (PoE defends against these)

Threat	Attacker capability	PoE defense
Developer-agent forgery (unexecuted review)	A developer LLM-agent fabricates a verdict file claiming reviewers approved when no reviewer process ran at all	Each reviewer transcript hash is committed <i>before</i> it can be signed; absence of the transcript = absence of evidence = no valid envelope
Operator-key abuse (passive)	An attacker with access to the operator's signing key signs an envelope without running the reviewers	Without matching transcript content committed to the repo's append-only leaf index, the signature does not verify; key alone is insufficient

Threat	Attacker capability	PoE defense
Rebase-time tampering	An attacker rebases a PR to silently swap content while keeping the original envelope	The signed Merkle root binds to specific per-file content hashes (v6 envelope schema, RFC-0042 §Design Layer 4); rebase changes the root and invalidates the signature
Cross-PR replay	An attacker copies a valid envelope from PR-A and attaches it to PR-B	The nonce is bound to the head SHA at signing time; verifier reproduces the binding and rejects mismatched envelopes
Silent dependency drift	An attacker swaps <code>pnpm-lock.yaml</code> between sign-time and verify-time	The reviewed file set is hashed at sign-time and pinned into the envelope; any drift invalidates
Late-binding deletion	An attacker deletes transcript files to hide evidence of a forged sign	The Merkle root is committed and never garbage-collected; even after transcripts age out per retention policy, the cryptographic claim "an attestation existed" remains verifiable

2.2 Out of scope (PoE does NOT defend against these)

Non-threat	Why it's out of scope
Compromised reviewer prompts	If the attacker can modify the reviewer agent's system prompt to always approve, the transcript will faithfully record an approval — and faithfully attest that the (compromised) reviewer ran. Defense lives in the trusted-reviewers manifest + reviewer prompt integrity, not in PoE.
LLM hallucination of false-negatives	PoE proves the reviewer ran; it does not prove the reviewer was <i>correct</i> . A reviewer that misses a real bug produces a valid attestation of a wrong verdict. This is a quality issue, not an attestation issue.
Anonymous public	PoE produces evidence verifiable by anyone with access to the repository. It does not (by default) publish to a public

Non-threat	Why it's out of scope
verification	transparency log; cross-organization audit is a future opt-in (see §7).
Adversary with operator-key + repo-write access	A fully compromised operator can sign forged transcripts AND commit them to the leaf index in one atomic push. Defense lives in operator credential protection (hardware keys, attested key generation), not in PoE.
Provider-side malice	If the LLM provider (Anthropic, OpenAI, etc.) emits maliciously crafted output, PoE attests the malicious output faithfully. Defense lives in provider selection and contractual liability.
Adversarial coordinator forging the independence signal itself (AISDLC-568, partial mitigation shipped; role-bound since AISDLC-572)	Every v6 leaf now carries a <code>verdictClass: 'independent' 'self-authored'</code> field (DEC-0012 opt-b), derived from whether a real <code>SubagentStart</code> harness hook fired, with a reviewer role (<code>code-reviewer</code> , <code>test-reviewer</code> , <code>security-reviewer</code> , or their <code>-codex</code> variants), around the time the transcript was written — a signal a coordinator cannot trigger by writing files or text alone, only by actually dispatching an <code>Agent / Task</code> reviewer subagent. AISDLC-572 closed a gap where the signal was credited on timing alone: any subagent (including a <code>developer</code> or <code>rebase-resolver</code>) dispatched in the window made a leaf <code>independent</code> , so a coordinator could dispatch an unrelated subagent and then hand-author a reviewer transcript within the window and still get credited. Binding the marker to the actual <code>agent_type</code> on the wire (and narrowing the lookback window from 2h to 30 minutes) correctly downgrades that case, plus the original common case (a coordinator running the reviewer's Bash steps itself, in-conversation, without spawning a subagent), to <code>self-authored</code> . It does NOT stop a <i>deliberately adversarial</i> coordinator with full repo Bash/Write access from also fabricating the marker file the hook writes — INCLUDING forging a reviewer <code>agent_type</code> string — see the next row.
Adversarial coordinator fabricating the <code>SubagentStart</code> marker file	Every v6 leaf may additionally carry an optional <code>harnessTranscriptHash</code> field: the SHA-256 of the reviewer subagent's own harness-captured execution transcript (Claude Code auto-writes every subagent invocation to <code>~/.claude/projects/<slug>/<session-id>/subagents/agent-<id>.jsonl</code> — a

Non-threat	Why it's out of scope
directly (AISDLC-570, DEC-0012 opt-a, DEC-0013 → opt1, shipped)	file the coordinator did not author and cannot cheaply fabricate without paying for a real <code>Agent / Task</code> call). Set only when the resolved transcript's reported role is a reviewer AND it contains a diff-binding nonce embedded in the reviewer's dispatch prompt — any failure fails closed to <code>null</code> . Trust-model caveat (DEC-0013): this is verified ONCE, at sign time, on the operator's own machine — a downstream CI verifier cannot independently re-derive it (no <code>~/.claude/projects/</code> on a CI runner), though tampering with an already-signed value is still caught by the Merkle root signature. It also does not prove the subagent's judgment was free of coordinator-engineered prompt bias — a coordinator willing to pay the LLM-token cost can still spawn a real, but biased, reviewer subagent (see the "Compromised reviewer prompts" row above). Wiring status: ACTIVE (AISDLC-573). Both canonical reviewer-dispatch paths (<code>/ai-sdlc execute</code> and <code>/ai-sdlc orchestrator-tick</code>) now generate the nonce and embed it in each reviewer's prompt before dispatch, so this field populates for real reviewer runs. The trust-model caveats above are unaffected — activation closes the wiring gap, not the sign-time-only / prompt-bias limitations.

The threat model is *insider/agent forgery within the operator's own pipeline*. PoE significantly raises the cost of the *unexecuted-review* forgery class without claiming to be a universal solution. As of this writing it structurally distinguishes an independent reviewer subagent from a lazily self-reviewing coordinator (the common case) via `verdictClass`, and additionally raises the forgery floor against a deliberately adversarial coordinator via `harnessTranscriptHash` — though the latter is a sign-time-only signal, not a CI-re-derivable one, and neither closes the prompt-bias gap (see the table rows above).

3. The Approach: Proof of Execution

PoE is built on four layers, each with a single responsibility. The construction is intentionally simple — every layer is reviewable by an auditor without specialist cryptography knowledge.

3.1 Layer 1 — Reviewer transcripts (raw evidence)

Each reviewer agent (code reviewer, test reviewer, security reviewer, and any harness variants — Codex, etc.) writes its full conversation transcript to a JSONL file at a deterministic path:

```
.ai-sdlc/transcripts/<task-id>/<reviewer-name>.jsonl
```

The transcript captures the events that mattered for the review: the initial prompt the reviewer received, the final verdict it returned, and (in future phases) per-tool-call detail. Files are local to the operator's machine by default and are gitignored — they are evidence, not artifacts. Retention is 90 days by default per RFC-0042 §OQ-1 (configurable per repo); the cryptographic claim survives indefinitely even after files age out.

The key property of the transcript: it is structurally rich enough that **forging a plausible one requires running a reviewer-equivalent LLM against the actual diff**. A few-token forgery does not produce a plausible 3–10 KB transcript that analyzes the specific files in the diff. The economic floor on forgery is approximately the cost of running the real reviewer.

3.2 Layer 2 — Append-only Merkle leaf index (commitment)

For each reviewer transcript, the framework computes a *leaf*:

```
{
  "leafIndex": 42,
  "taskId": "AISDL-383.2",
  "reviewerName": "code-reviewer",
  "transcriptHash": "<SHA-256 of the JSONL file>",
  "nonce": "<32-byte hex bound to PR head SHA>",
  "harness": "claude-code",
  "model": "claude-sonnet-4-6",
  "verdictApproved": true,
  "findings": { "critical": 0, "major": 0, "minor": 1, "suggestion": 0 },
```

```
"signedAt": "2026-05-21T17:14:37.561Z"
}
```

Leaves are appended (never modified, never deleted) to a single file in the repository: `.ai-sdlc/transcript-leaves.jsonl`. This file is committed to git. **Once a leaf has been pushed to `main`, it is part of the permanent record.** A later attempt to revise or remove a leaf is a visible git history modification.

The append-only file is the *commitment* layer. It says, in the language of the repo's commit history: "a reviewer produced this transcript hash, at this time, against this PR's head SHA."

3.3 Layer 3 — Merkle root (compaction)

For each PR, the framework computes the Merkle root over the subset of leaves that belong to that PR (a small, well-defined slice of the global leaves file). The root is a single SHA-256 hex string that captures the entire reviewer evidence chain for the PR. Per-leaf inclusion proofs allow point-verification of any individual reviewer's evidence without reading the entire leaf set.

3.4 Layer 4 — Signed envelope (operator attestation)

The PR's envelope is a DSSE-shaped JSON document:

```
{
  "schemaVersion": "v6",
  "subject": { "digest": { "sha1": "<head SHA>" } },
  "transcriptLeaves": [
    { "leafIndex": 41, "reviewerName": "code-reviewer", "transcriptHash": "..."},
    { "leafIndex": 42, "reviewerName": "test-reviewer", "transcriptHash": "..."},
    { "leafIndex": 43, "reviewerName": "security-reviewer", "transcriptHash": "..."}
  ],
  "merkleProofs": [ ... ],
  "rootHash": "<SHA-256 of the Merkle tree root>",
  "rootSignature": "<operator ed25519 signature over rootHash>",
  "nonce": "<32-byte hex bound to head SHA>"
}
```

The envelope is written to `.ai-sdlc/attestations/<head-sha>.dsse.json` and pushed alongside the PR. The operator's key signs *only* the root, which compresses

arbitrarily many reviewer leaves into 32 bytes of attestable content. The trust chain becomes:

```
operator signs root → root commits leaves → leaves bind transcripts → transcripts evidence reviewer execution
```

Each link is independently verifiable. Breaking any link breaks the chain.

3.5 Verification

A verifier (the framework's `verify-attestation` CI workflow, or any third party with read access to the repo) reproduces the chain in reverse:

1. Read the envelope at `<head-sha>.dsse.json`.
2. Verify the operator signature on `rootHash` against the operator pubkey in `.ai-sdlc/trusted-reviewers.yaml`.
3. Reconstruct the Merkle root from `transcriptLeaves` + `merkleProofs` and check it equals `rootHash`.
4. For each leaf, optionally fetch the corresponding transcript file and verify `SHA-256(transcript) == leaf.transcriptHash`.
5. Verify `nonce` was generated against the PR's head SHA.

Any failure invalidates the attestation. The merge gate refuses to merge.

4. Cryptographic Construction in Detail

4.1 Domain separation (RFC-6962 alignment)

The Merkle tree uses RFC-6962-style domain separators:

- Leaf hash: `SHA-256(0x00 || canonical_json(leaf))`
- Internal node: `SHA-256(0x01 || left || right)`

This defends against second-preimage attacks (CVE-2012-2459 class) where an attacker could construct a phantom leaf whose hash equals an internal node. The separator byte makes it cryptographically infeasible to confuse the two layers.

4.2 Odd-leaf handling

When a tree level has an odd number of nodes, the last node is paired with itself. The verifier's `verifyInclusion` function takes the committed leaf count alongside the proof and bounds-checks the claimed leaf index against it. An attacker cannot claim an inclusion proof for a phantom leaf beyond the committed count.

4.3 Nonce binding

The per-leaf nonce is `SHA-256(headSha || 16-random-bytes)`. The random component prevents two leaves with identical content (same task, same reviewer, same model) from sharing a hash. The head-SHA component prevents cross-PR replay: a leaf from PR-A cannot be lifted into PR-B because the nonce was generated against PR-A's head and the verifier checks the binding.

4.4 Append-only invariants

The leaf file `.ai-sdlc/transcript-leaves.jsonl` is written via atomic `tmp + rename`. Concurrent writers (parallel `/ai-sdlc execute` runs against the same repo) are coordinated via either `O_APPEND` semantics or an advisory lock. The verifier additionally validates that `leaf.leafIndex` equals the leaf's position in the file — protecting against in-place tampering that might shift indices.

4.5 Key model — any-of-N

Per RFC-0042 §OQ-4, multiple operator pubkeys can be registered in `.ai-sdlc/trusted-reviewers.yaml`. A verifier accepts an envelope signed by *any* registered key. This supports multi-operator teams without requiring a single shared private key (which would defeat the purpose). New operators onboard by adding their pubkey via a normal PR; the existing operators sign that PR, after which the new operator can sign envelopes.

5. What Adopters Get: Compliance & Audit Fit

5.1 SOC 2 and ISO 27001 alignment

PoE produces audit-grade evidence that the change-management process executed as documented. The mapping to common control frameworks:

Control objective	PoE contribution
SOC 2 CC8.1 (Change management)	Cryptographic record that each code change underwent the documented review process before merge
SOC 2 CC7.2 (Monitoring)	Append-only log of every reviewer execution, queryable by date, repo, reviewer, or PR
ISO 27001 A.14.2.2 (System change control)	Tamper-evident attestation chain for every change reaching production
ISO 27001 A.12.4.1 (Event logging)	Per-reviewer transcript with timestamps, retained per the organization's evidence policy
SLSA Build Level 3-aligned (review provenance)	Provenance for the AI-review step of the build, signed by the operator

The auditor's question — "show me evidence that the AI reviewer ran on this specific PR" — has a deterministic answer: fetch the envelope, verify the signature, fetch the transcript, hash it, compare. No reliance on workflow-run timestamps or screenshot evidence.

5.2 Retention and discovery

The default 90-day local-transcript retention covers the realistic incident-response window (most forgery investigations surface within weeks). Beyond 90 days, the Merkle root remains committed and verifies indefinitely — the cryptographic claim "an attestation existed for this PR with this content hash" outlives the raw transcript. Organizations with longer regulatory retention (HIPAA: 6 years; FINRA: 7 years) configure a higher retention value, or opt into remote storage via `transcript_storage_url` (S3, GCS, or HTTP PUT).

5.3 No third-party dependency

The PoE critical path has zero external services. No Rekor, no transparency log, no certificate authority, no CT log monitor. The merge-gate latency is bounded by local disk and one signature verification. This matters in air-gapped environments, regulated industries with vendor-restriction policies, and any context where the PR cycle's per-second cost is meaningful.

A future opt-in publishes Merkle roots to a public transparency log (Rekor or equivalent) for organizations that want cross-organization audit. The opt-in does not change the critical path.

5.4 Forensic spot-check

When an incident raises the question "did the security reviewer actually flag this in PR #1234?", the operator runs:

```
cli-attestation spot-check 1234
```

The tool fetches the envelope, re-hashes the transcript files, and reports whether the chain still verifies. If transcripts have aged out per retention, the response is "Merkle proof valid; raw transcript GC'd per retention policy" (exit 0 — soft-fail per §OQ-3). The cryptographic claim is preserved; only the content for content-plausibility review is unavailable past the retention window.

6. Comparison to Alternatives

6.1 No attestation (baseline)

The dominant practice today. CI runs the reviewer, the workflow finishes green, the PR merges. Properties: zero cryptographic guarantees, full reliance on CI integrity, no defense against the workflow being bypassed by a local "I ran it" flag, no historical verifiability.

PoE replaces "the workflow finished green" with "the operator signed evidence that the named reviewer ran on this diff."

6.2 Operator-key signed JSON (status quo for many teams)

This is what AI-SDLC shipped before PoE (v5 envelope schema). A single signature covers a JSON file containing reviewer verdicts. The forgery vector documented in §1.2 lives here: the JSON file is written by the developer agent before the reviewer runs; the operator key signs whatever JSON is present.

PoE's improvement: the signed object is a Merkle root over committed per-reviewer transcript hashes. A developer agent cannot produce a valid PoE envelope without first producing a transcript that hashes to the leaf content. Producing a plausible transcript requires running a reviewer-equivalent LLM against the diff — which is exactly what we wanted the reviewer to do.

6.3 SLSA / sigstore / in-toto for the review step

These primitives are excellent for source-code provenance (where the artifact is reproducible from inputs). They do not natively express "an AI agent executed against this diff." Conceptually one could write a SLSA Layer-3 statement with `predicate.builder` = the LLM provider, but the LLM provider does not (today) emit a signed receipt, so the statement is unverifiable.

PoE composes with SLSA: an organization can wrap the PoE envelope inside a SLSA in-toto statement to integrate with existing provenance tooling. The PoE envelope is the contents of the predicate; SLSA provides the outer wrapping.

6.4 Vendor-side execution logs

Some AI coding tools (Cursor, GitHub Copilot Workspace, Devin) maintain server-side execution logs that an organization could in principle subpoena to prove an agent ran. Properties: vendor lock-in, no operator-side cryptographic verification, retention controlled by vendor, no defense against vendor logs being modified or lost.

PoE puts the evidence in the customer's repository, signed by the customer's keys, with no dependency on vendor continuity. This is the standard "bring your own key, bring your own evidence" posture.

6.5 Trusted execution environments (TEE / confidential compute)

A TEE could in principle run the reviewer LLM and emit a remote attestation. Properties: requires TEE-capable infrastructure (limits provider choice), requires TEE attestation key management, defends against host-level compromise (which is not in PoE's threat model anyway).

PoE is complementary: a TEE-hosted reviewer can still write a PoE-compatible transcript that the framework attests. The two layers compose without conflict.

7. Adoption & Operations

7.1 What changes for the operator

After PoE adoption, the operator's workflow shifts in three places:

1. **Reviewer execution:** each reviewer agent writes a JSONL transcript to its task-scoped path. This is automatic; the operator does not invoke it manually.
2. **Pre-push hook:** the existing `check-attestation-sign.sh` hook is updated to produce v6 envelopes. The operator does not change its invocation.
3. **Verification:** the existing `verify-attestation` CI workflow is updated to verify v6 envelopes. Backward compatibility with v3/v4/v5 envelopes is preserved indefinitely (per §OQ-7) so historical PRs remain verifiable.

For end-to-end PR shipping, the operator experience is unchanged. The cryptographic property is the only thing that changes.

7.2 Onboarding cost

New contributors: install AI-SDLC, run `/ai-sdlc init-signing-key` once, commit the pubkey via a normal PR. After the pubkey lands on `main`, the contributor can sign envelopes. No transparency-service registration. No certificate enrollment. No HSM provisioning (HSM-backed keys are supported but not required).

The bootstrap problem (first PR ever in a repo) is resolved per §OQ-6: the first push is the genesis. There is no chicken-and-egg.

7.3 Storage cost

A typical reviewer transcript is 3–10 KB. For a repository at 100 PRs/month with 3 reviewers per PR, the storage footprint is approximately 9 MB/month of transcript files (local-disk default) plus a few KB/month of leaf-file growth (committed). At 12 months retention this is ~110 MB of evidence — comparable to a small CI log archive.

The Merkle leaf file itself (committed, never garbage-collected) grows at ~300 bytes per leaf, or roughly 1 MB per 3,300 PRs. At any realistic project lifetime this is negligible.

7.4 Recovery from key loss

If an operator loses their signing key (hardware loss, key compromise), the recovery flow is:

1. Operator generates a fresh keypair (`/ai-sdlc init-signing-key --force`).
2. Operator opens a PR that adds the new pubkey to `.ai-sdlc/trusted-reviewers.yaml` and removes the lost one.
3. Surviving operators sign that PR. The lost key's prior envelopes remain verifiable historically (the pubkey stays in the registry for verification even after revocation).
4. New envelopes are signed with the new key going forward.

There is no rekey ceremony for in-flight PRs. Historical attestations preserve their original signatures and remain verifiable against the historical pubkey.

7.5 Performance characteristics

Operation	Wall-clock (typical)	Notes
Compute leaf hash	< 1 ms per leaf	SHA-256 over a few KB

Operation	Wall-clock (typical)	Notes
Compute Merkle root	< 5 ms per 1000 leaves	Linear-with-leaves
Generate inclusion proof	< 1 ms per leaf	Log-with-leaves
Sign envelope (ed25519)	< 1 ms	Standard primitive
Verify envelope (no transcript re-hash)	< 5 ms	Hash + signature verify
Verify envelope (with transcript re-hash)	< 50 ms	One SHA-256 per reviewer transcript

PoE adds approximately one disk-write and one signature verification to the PR critical path. It does not change the dominant cost (LLM inference for the review itself).

8. Open Questions and Future Work

The current release (v6 envelope schema, RFC-0042 §Design Layer 4) addresses the seven open questions catalogued in the RFC. The roadmap for v7 and beyond:

- **LLM-as-judge content plausibility verification** — beyond hash verification, sample-check whether transcript content semantically corresponds to the PR diff. This is the layer-2 defense against an attacker who fabricates structurally valid but content-irrelevant transcripts.
- **Public transparency log integration** (opt-in) — publish Merkle roots to Rekord or equivalent for organizations that want cross-organization audit.
- **Signed-receipt LLM providers** — when LLM providers begin emitting signed receipts for completions (Anthropic and OpenAI have both announced exploratory work in this direction in 2026), incorporate provider receipts into the transcript leaves to eliminate the provider-malice gap from §2.2.
- **Hardware-attested operator keys** — TPM/HSM-backed ed25519 signing for organizations with key-storage compliance requirements.

- **Reviewer prompt integrity** — extend the leaf shape to include a hash of the reviewer's system prompt at execution time, so prompt-injection attacks become visible in the audit trail.
-

9. References

- **RFC-0042**: Proof-of-Execution Attestation via In-Repo Merkle Transcripts. AI-SDLC Framework. spec/rfcs/RFC-0042-proof-of-execution-attestation.md
 - **DSSE Specification**: Dead Simple Signing Envelope. <https://github.com/secure-systems-lab/dsse>
 - **SLSA Provenance**: Supply-chain Levels for Software Artifacts. <https://slsa.dev/>
 - **RFC-6962**: Certificate Transparency. Section on Merkle tree construction + second-preimage resistance. <https://datatracker.ietf.org/doc/html/rfc6962>
 - **Sigstore / Rekor**: Transparency log for software signatures. <https://sigstore.dev/>
 - **AI-SDLC Framework**: <https://github.com/ai-sdlc-framework/ai-sdlc>
-

Appendix A — Glossary

- **Attestation envelope** — the signed JSON document produced per-PR carrying the Merkle root, reviewer leaves, proofs, and signature.
- **DSSE** — Dead Simple Signing Envelope. A standard signed-statement format used as the wrapper for PoE envelopes.
- **Leaf** — a single per-reviewer record in the append-only leaf index, carrying the transcript hash, reviewer identity, nonce, and verdict counts.
- **Merkle root** — the SHA-256 hash at the top of a Merkle tree, compressing arbitrarily many leaves into 32 bytes of attestable content.
- **Nonce** — a 32-byte hex value bound to the PR's head SHA, preventing cross-PR replay.

- **Operator** — the human or automation that signs attestation envelopes. In AI-SDLC, typically the engineer or service account that runs the `/ai-sdlc execute` slash command.
 - **Reviewer (agent)** — an AI agent (LLM-driven) whose job is to read a diff and emit a structured verdict. Examples: code-reviewer, test-reviewer, security-reviewer.
 - **Transcript** — the JSONL file recording a reviewer's conversation events for a single execution. Local-disk by default, gitignored.
 - **Trusted-reviewers manifest** — `.ai-sdlc/trusted-reviewers.yaml`, the registry of operator pubkeys whose signatures the merge gate accepts.
-

AI-SDLC is open source under the Apache 2.0 license. Issues, contributions, and security-disclosure inquiries: github.com/ai-sdlc-framework/ai-sdlc.