Proof Packets: A Derive-Don't-Trust Envelope for Accountable Agent Actions

Zain Dana Harper
Seattle, WA | ORCID 0009-0001-7175-5393
https://github.com/HarperZ9

July 2026

Abstract

As autonomous agents take consequential actions, the artifact that records "what happened" becomes load-bearing, and it is usually a log the agent wrote about itself. A log can assert success it did not achieve. This paper describes a small, deterministic verification envelope—a *proof packet*—for a single agent action, whose defining property is that its verdict is *derived from checks, never read from the packet*. A packet carries an objective and its scope, hashed source and context references, a routing and admission record, the action and its arguments hash, a side-effect class, a compensation reference, outputs with digests, and embedded artifact bodies. A pure verifier runs a fixed set of named checks over this structure and folds their results into one verdict from a three-value lattice—MATCH, DRIFT, UNVERIFIABLE, with DRIFT dominating UNVERIFIABLE dominating MATCH. There is no code path that returns MATCH without a passing check set, and a canned MATCH embedded in the packet is itself recorded as DRIFT. Missing evidence (an output digest with no embedded body, a required field absent) is reported as UNVERIFIABLE with the offending JSON path, never as a silent pass; recomputable tamper (a wrong artifact digest, an execution ordered before its admission) is DRIFT. We are deliberately blunt about scope: the contribution is the *envelope and its derive-don't-trust verdict*, not the content of any domain it wraps. We illustrate with fixtures across several domains and state plainly that each fixture is textbook material—for example, the "quantum error correction" fixture is a classical three-qubit repetition code—so that the envelope's generality, not a domain result, is what the illustrations show.

1 Introduction

An autonomous agent that reads a file, calls a tool, or writes to an external system produces, at best, a record of having done so. When that record is the basis for a downstream decision—deploy this, trust this result, pay this invoice—its integrity matters as much as the action's. The default record is a self-report: the agent logs its own success. Nothing in a plain log distinguishes a truthful "completed" from a fabricated one, and nothing forces the log to expose the evidence that would let a third party re-derive the claim.

The response we describe is not a richer log or a trusted execution environment. It is a fixed-shape *packet* for one action, verified by a pure function whose single design commitment is that the verdict is computed from checks over the packet's own materials, so that the packet cannot vouch for itself. This inverts the usual trust direction: instead of reading a status the producer wrote, a verifier re-derives the status the materials support, and any disagreement between the two is itself a finding.

Contributions.

1. **A derive-don't-trust verdict** (Section 3): a pure verifier that folds a fixed set of named checks into a three-value lattice, with the property—enforced, not asserted—that an embedded verdict can never raise the result above what the checks support.
2. **An evidence-gap discipline** (Section 4): the verifier separates *missing evidence* (UNVERIFIABLE, named by JSON path) from *contradicted evidence* (DRIFT), and refuses to let a derivative failure (a hash mismatch) mask a more basic one (an absent field).
3. **A domain-agnostic envelope** (Section 2): the same packet shape wraps a claim in any domain, verified by re-execution of the check set; we demonstrate this across fixtures and are explicit that the fixtures are illustrations, not results.

2 The packet

A proof packet is a canonical JSON object for one action. Its required structure records: an `objective` (claim, scope, and success criterion); `source_refs` and `context_refs`, each carrying a content hash so evidence can be pinned; a `route` (lane, decider, confidence, route reference); an `admission` record (decision, policy, authority, and an ordinal); the `action` (identifiers, tool, kind, an arguments hash, an execution ordinal, an idempotency key); a `side_effect` class and reversibility flag; a `compensation` block; `outputs`, each with a name, a digest, and a reference; the `artifacts` themselves as embedded bytes; and a `final_status`. The packet is content-addressed: a `packet_hash` is computed over the canonical form with no clock or randomness inside the hash scope, so the same materials hash identically on any machine.

3 The verdict

Verification is a pure function from a packet to a set of named check results and one derived verdict. The verdict lattice has three values: MATCH (every check passed), DRIFT (a confirmed contradiction), and UNVERIFIABLE (the check could not be completed). The fold is a fixed precedence: if any check yields DRIFT the verdict is DRIFT; else if any yields UNVERIFIABLE it is UNVERIFIABLE; else MATCH. The load-bearing property is negative: there is no path that returns MATCH without a passing check set. A packet may embed a claimed verdict, but the verifier compares it to the derived one; when they disagree the disagreement is recorded as a failure that inherits the derived severity, so an embedded MATCH over tampered materials stays DRIFT and an embedded MATCH over an incomplete packet stays UNVERIFIABLE. A self-asserted pass cannot win.

4 The checks

The verifier runs a fixed suite; each check is named and its failures carry the offending JSON path. The design distinguishes an *evidence gap* (UNVERIFIABLE) from *tamper* (DRIFT) wherever the gap can be detected before it perturbs the hash scope; a gap that also changes the hashed bytes is reported at the stricter DRIFT level, so it is never a silent pass either way.

- **Required fields, reference completeness.** Each required path, and each source/context/output element field, must be present; a missing one is UNVERIFIABLE named by its path, and marks the packet structurally incomplete.
- **Artifact digests.** Each output digest is recomputed over its embedded artifact bytes. A wrong digest is DRIFT; a digest *claimed* with no embedded body is UNVERIFIABLE, not a silent pass—trusting a claimed digest on faith would let a phantom output reach MATCH.

- **Hash integrity, gated.** The packet hash is recomputed and a mismatch is DRIFT, but the check is gated on structural completeness so that a missing required field, source or context reference field, or output element surfaces as its own UNVERIFIABLE rather than being masked by the derivative hash mismatch that deleting it would cause. A missing artifact *body* is the one gap the gate does not precede: it is reported as DRIFT via the recomputed hash *in addition to* its UNVERIFIABLE digest gap—strictly more severe, so still never a silent pass.
- **Admission before execution.** The action's admission record must match the action, and the execution ordinal must strictly follow the admission ordinal; an action executed before it was admitted is DRIFT.
- **Compensation for external writes.** An external-write side effect without a compensation reference is DRIFT.
- **Resolvable evidence, known states.** An output that cites a source reference not present in the packet is UNVERIFIABLE; an out-of-vocabulary state, side-effect class, or admission decision is DRIFT.

Each check is total and the verifier fails closed: it produces a verdict for any input, and an input it cannot interpret degrades toward UNVERIFIABLE rather than raising.

5 Illustrations, and what they do and do not show

The same envelope wraps a claim in any domain, and we ship fixtures that do so: a causal-inference fixture, an embodied sim-to-real fixture, and a quantum-error-correction fixture, among others. These illustrate one thing only—that a claim plus its evidence and negative controls can be sealed into a packet and re-verified—and we are explicit about the honest size of each. The domain content is textbook: the causal fixture recovers a backdoor adjustment set by brute-force subset enumeration on a small graph; the "quantum error correction" fixture is a *classical* three-qubit repetition code with a syndrome lookup, not a quantum circuit; the embodied fixture uses synthetic, not real, observations. The fixtures also carry *negative controls*: deliberately broken variants that the envelope must reject, so a passing fixture demonstrates the verifier is not vacuous on that shape. None of the fixtures is a domain research result, and framing them as one would be the exact self-vouching the envelope exists to prevent. The contribution is that a single accountable-claim envelope, with a verdict the packet cannot forge, spans these domains unchanged.

6 Related work

Tamper-evident and Merkle-style logging [2] and content-provenance standards [3] give integrity for stored records; a proof packet applies the same content-addressing to a single agent action and adds a verdict that is derived rather than stored. Proof-carrying code [1] moves trust to a re-checkable artifact; a proof packet is a proof-carrying *action record* whose checks are structural and re-executable rather than a logical safety proof. Work on verifiable and constrained agents [4] shifts agent output toward the checkable; the packet is complementary, sitting at the after-the-fact record rather than the pre-action constraint. Reproducibility tooling for computational claims [5] shares the re-execution instinct; our narrow addition is the derive-don't-trust verdict and the explicit gap-versus-tamper separation.

7 Conclusion

A record an agent writes about its own action is worth exactly as much as the checks a third party can re-run against it. A proof packet makes those checks the source of the verdict, so the record cannot vouch for itself: missing evidence is UNVERIFIABLE, contradicted evidence is DRIFT, and MATCH is only ever earned. The envelope is small and domain-agnostic, and its value is precisely that it refuses to let a claim—in any domain—be larger than its evidence.

References

[1] Necula, G. C. *Proof-carrying code.* POPL, 1997.

[2] Merkle, R. C. *A digital signature based on a conventional encryption function.* CRYPTO, 1987.

[3] Coalition for Content Provenance and Authenticity. *C2PA Technical Specification.* https://c2pa.org, 2024.

[4] Chen, B. *Making Failure Safe: A Constrained, Verifiable Agent Framework for Open-Web Data Collection.* arXiv:2607.00035, 2026.

[5] Reproducible Builds Project. *Reproducible Builds: a set of software development practices.* https://reproducible-builds.org, 2026.