

The Verifier — specification for an independent implementation
Preview specification · 11 September 2026 · for a format that is an **Internet-Draft**, not a standard

> **PREVIEW.** `draft-saha-stage-receipts-00` was posted at the IETF on 8 September 2026
> (<https://datatracker.ietf.org/doc/draft-saha-stage-receipts/>); it is a draft, not a standard.
> This document specifies what any verifier of that format must do, published
> now so that a second implementation can be written by someone who has never
> spoken to us. A reference implementation of roughly 140 lines ships beside it
> in `samples/verify.py`.

1. What a verifier is for

A verifier answers one question: **can a stranger, holding only these records, establish what they assert — without trusting whoever produced them?**

Everything below follows from that sentence, including the refusals.

2. The five absolute constraints

A conforming verifier:

1. **Requires no network.** Not for schemas, not for revocation, not for telemetry, not for anything.
2. **Requires no key, no account, and no licence.**
3. **Requires no live third party.** Evidence that needs a living authority to be checkable inherits that authority's lifespan.
4. **Is independently implementable** from this specification and the golden vectors alone.
5. **Never mutates the records it checks.**

These are not implementation preferences. A verifier that violates any of them has re-created the problem the format exists to remove — you would be trusting a vendor again, only now with more steps.

Corollary the publisher is bound by: we may not operate a certificate authority, a key escrow, or any service on which a record's checkability depends.

3. Outcomes — three, never two, never silence

Every check reports exactly one of:

Outcome	Meaning
---	---
PASS	The property was established.
FAIL	The property was contradicted. The verifier emits what it found.
NOT RUN — with reason	The property could not be established. The reason is mandatory , and an unexplained skip is itself a failure of the verifier.

A verifier that reports only pass and fail will eventually report PASS for something it did not check. The third outcome is the point.

4. What the verifier must check

4.1 Serialization

- The file's bytes **are** the canonical serialization of the object they encode. Not equivalent to it — identical.
- No JSON numbers appear anywhere. Every numeric value is a decimal string.
- The result is byte-identical on every platform. Line endings, byte-order marks and path length are named traps, not discovered ones.

4.2 Self-description

- Every required field is present.
- The **instrument** is named and pinned. A record whose instrument cannot be identified is invalid, and the conforming producer behaviour is refusal, not a blank field.
- Every assertion **names the transform constants it depends on**. A relation that holds only under an unstated constant tests the constant, not the relation.
- Every temporal field **carries its zone**.
- Every input and output declares a **trust class**: operator-authored, model-generated, or externally sourced. This does not make externally-sourced content safe; it makes it visible, which is the most a record format can honestly promise to any reader — human or model — downstream.

4.3 States that must not collapse

- **absent**, **declared-none**, and **recorded** are three distinct states and must round-trip distinguishably.
- The outcome class is one of `ok`, `refused`, `error`. A refusal is a result.
- An `error` outcome carries **status, body and origin**. An error without its body is a status code wearing an explanation.
- A failed measurement is not a measured zero.

4.4 Coverage

- `coverage.completeness` is `complete` or `incomplete` — never absent.
- `complete` must be consistent with the declared and emitting stage lists; a contradiction is a FAIL, not a warning.
- Where coverage is `incomplete`, **any consumer of these records is forbidden from asserting a root cause across an undeclared boundary**. A verifier that cannot see the edge of its own chart will eventually report confidently about something it never observed.

4.5 Chain

- Each receipt's `prev` equals the digest of the preceding receipt's exact bytes.
- The manifest's `chain_head` equals the digest of the last receipt.
- **No receipt contains its own digest**. A record cannot contain the proof of its own final state; the digest lives in the chain and in the next receipt.

4.6 Anchoring — and the limit that must be reported, not implied

- `anchor.state` is `anchored` or `unanchored`. Absence is a FAIL.
- Where a record is `unanchored`, the verifier **must** emit **NOT RUN — originality**, with the reason stated.

This is the most important paragraph in the specification. A chain that verifies proves that these records are **consistent with each other**. It does

not prove they are the records that were made: an operator holding every copy can rewrite one and recompute every downstream link. ****A hash proves a record is consistent with itself, never that it is the record that was made.**** Only anchoring to a commitment the operator cannot rewrite converts consistency into originality, and ***the anchor cadence is the tamper window***. A verifier that lets a green chain imply originality is lying by omission, so the format requires it to say so out loud on every unanchored record.

5. What the verifier must NOT do

- ****Not infer.**** An absent field is absent, not defaulted.
- ****Not repair.**** No normalization of nearly-canonical bytes into canonical ones.
- ****Not rank, score, or opine.**** It reports; it does not interpret. An interpreter's output is testimony and is written beside records, never into them.
- ****Not judge the work.**** A well-formed record of a wrong answer verifies. The verifier says nothing about whether the pipeline was correct, the retrieval relevant, or the answer true.
- ****Not require its own producer.**** A verifier that only accepts records from one SDK has certified an implementation, not a format.

6. Rejection vectors are part of the specification

Ten are published in ``samples/rejection/``, each with the reason it must be refused. ****A format with no rejection vectors has not specified anything**** — it has described a happy path and left every implementer to guess at the edges.

An implementation claiming conformance must refuse all ten, and must refuse them ***for the stated reason*** — refusing the right file for the wrong reason is a failing test that happens to look green.

7. The bar this specification is aimed at

****One verifier, four frameworks, blind.**** Records produced by a plain-Python control, LangChain, Langflow and LlamaIndex must all check against a single verifier that requires no framework-specific handling — and ****which cannot tell from the records which framework produced them****, except where the instrument declaration deliberately says so.

If a verifier needs to know its source, there are four dialects and no standard. That result would be published as a failure.

8. Write the second one

The reference verifier is small on purpose. One verifier checking one format is a claim; ****two independent verifiers agreeing on the same records is evidence.****

If you implement one, publish your results against the samples and the rejection vectors — including any place where you and the reference implementation disagree. A disagreement between two verifiers is the most valuable bug report this project can receive, because it means the specification is ambiguous and the ambiguity has been found before anyone

depended on it.