Independent Submission
Internet-Draft
Intended status: Informational
Expires: 23 November 2026

xkumakichi
22 May 2026

Signed Execution Receipts for AI Agent Tool Calls (XAIP Receipts)
draft-xkumakichi-xaip-receipts-00

Abstract

This document defines a wire format for signed execution receipts produced by AI agents when they invoke tools, services, or other agents. A receipt records the minimum facts needed to make a trust decision about a future call: who acted, who delegated, what tool was used, whether the call succeeded, how long it took, and how the call's inputs and outputs are identified (without disclosing their contents).

The format is intentionally tool-system-agnostic. The same receipt structure can be emitted by MCP (Model Context Protocol) servers, LangChain.js callback handlers, OpenAI tool-calling loops, HTTP clients, or proprietary agent runtimes. Receipts use Ed25519 signatures over a JSON-canonicalized payload, and identities are W3C Decentralized Identifiers (DIDs).

Scoring policy, aggregation architecture, and reactive behavior in response to receipts are explicitly out of scope and left to deployments.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 23 November 2026.

Table of Contents

1. Introduction

1.1. Motivation

AI agents increasingly act on behalf of users: they pick tools, call APIs, delegate to other agents, and -- in some deployments -- participate in transaction workflows. Each of those actions is preceded by an implicit trust decision: which tool should I use, and is it likely to do what I expect?

Today, that decision is mostly answered by upstream proxies -- whether the tool's name appears in a model's training data, whether a registry surfaces it, whether a platform recommends it. None of these proxies record what the tool actually did in real calls. There is no widely-deployed, interoperable record format that an agent (or an agent-payment protocol, or an audit system) can use to look back and answer "what happened the last N times this tool was called?"

This document defines such a format. It is intentionally narrow: it covers the wire format for one receipt. How receipts are stored, aggregated, queried, scored, or reacted to is a deployment-policy concern and is out of scope.

1.2. Design Principles

* Wire format only. Scoring models, aggregation topologies, and decision logic are deployment choices, not protocol requirements.
* Tool-system-agnostic. The same receipt can be produced by MCP, LangChain, OpenAI tool calling, plain HTTP, or proprietary runtimes.
* Privacy-preserving by construction. Receipts identify inputs and outputs by hash, not by content. A receipt does not require disclosure of user data, prompts, or tool outputs.
* Independently verifiable. Anyone holding the receipt and the public keys can verify the signatures without consulting any registry or trusted third party.
* Co-signed where possible. Both the Executor (Agent) and the Caller sign the same canonical payload, so neither can unilaterally fabricate the record.

1.3. Out of Scope

This document does NOT define:

* A scoring model. Trust scores derived from receipts are deployment policy.
* An aggregation architecture. Receipts can be stored locally, federated, anchored, or relayed in any pattern.

* A query API. Consumers may serve receipts and/or derived data over any protocol they choose.
* Identity priors. If a deployment chooses to weight different DID methods differently, that is deployment policy.
* A specific transport. Receipts may be exchanged over HTTP, MCP, message queues, or any other carrier.

1.4. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.4.1. Terminology

Agent: An automated system, typically an AI agent, that invokes tools, services, or other agents on a principal’s behalf.
Caller: The party that delegated the tool call to the Agent. Often (but not always) the same legal entity as the Agent’s principal.
Tool: A named operation invoked by the Agent. The tool implementation may be local code, an MCP server, an HTTP API, a sub-agent, or any callable target.
Receipt: A signed record of a single Tool execution attempt.
Executor signature: The signature produced by the Agent that ran the tool.
Caller signature: The signature produced by the Caller over the same canonicalized payload as the Executor signature.
DID: Decentralized Identifier, as defined in the W3C DID Core specification [DID-CORE].

2. Receipt Structure

A receipt is a JSON object with the following fields:

Field	Type	Required	Description
agentDid	string (DID)	yes	The Agent that executed the tool.
callerDid	string (DID)	yes	The Caller that delegated the tool call. MAY equal agentDid when there is no delegation.

toolName	string	yes	A stable identifier for the tool. Format is opaque to this spec.
taskHash	string (hex, lowercase)	yes	A hash of the canonical task input. SHA-256 RECOMMENDED.
resultHash	string (hex, lowercase)	yes	A hash of the canonical task output. SHA-256 RECOMMENDED. For failures, the hash MAY be of a canonical failure description.
success	boolean	yes	true if the tool call satisfied the agent’s success criterion, false otherwise.
latencyMs	integer >= 0	yes	Wall-clock time from invocation to completion, in milliseconds.
failureType	string	yes	One of the values defined in Section 5 when success is false. When success is true, the value MUST be the empty string.
timestamp	string (RFC 3339)	yes	UTC timestamp of completion.
signature	string (hex)	yes	Ed25519 signature by the Agent over the canonical payload.
callerSignature	string	recommended	Ed25519 signature

	(hex)			by the Caller over the same canonical payload.
toolMetadata	object	optional		Tool-class or capability hints. Format is deployment-defined.

Table 1

2.1. Example

```
{
  "agentDid": "did:web:myagent.example",
  "callerDid": "did:key:z6Mk...",
  "toolName": "translate",
  "taskHash":
"9b74c9897bac770ffc029102a200c5de5f2a1b3c4d5e6f708192a3b4c5d6e7f8",
  "resultHash":
"f0e1d2c3b4a5987612345678abcdef00112233445566778899aabbccddeeff00",
  "success": true,
  "latencyMs": 142,
  "failureType": "",
  "timestamp": "2026-05-14T10:30:00.000Z",
  "signature": "...",
  "callerSignature": "..."
}
```

3. Canonical Payload and Signing

3.1. Canonical Payload

The signed payload is the JSON object containing exactly the following fields, in this order after lexicographic sorting per [RFC8785]:

agentDid, callerDid, failureType, latencyMs, resultHash, success, taskHash, timestamp, toolName

The signature, callerSignature, and toolMetadata fields are excluded from the canonical payload. Implementations producing receipts MUST canonicalize using JCS as defined in [RFC8785].

3.2. Signing Algorithm

Signatures are computed using Ed25519, as defined in [RFC8032]. The signature input is the UTF-8 encoding of the canonical JSON string produced in the previous subsection.

The signature field is the Executor's Ed25519 signature, encoded as a lowercase hexadecimal string. The callerSignature field, when present, is the Caller's Ed25519 signature over the same canonical input.

3.3. Verification

A verifier MUST:

1. Recompute the canonical payload from the receipt's fields.
2. Resolve agentDid to its current public key per [DID-CORE].
3. Verify signature against the canonical payload using the Agent's public key.
4. If callerSignature is present, resolve callerDid similarly and verify callerSignature against the same canonical payload.
5. Reject the receipt if any signature verification fails.

A verifier MAY additionally validate that timestamp is within a deployment-defined freshness window.

4. SigningDelegate Pattern (Caller Co-signature)

To produce a co-signed receipt, a Caller MUST NOT transmit private key material to the Executor. Instead, the Caller exposes a SigningDelegate interface:

```
interface SigningDelegate {
  did: DIDString
  sign(payload: string): Promise<HexString>
}
```

The Executor sends the canonical payload string to the Caller's sign method and receives the signature. The private key never leaves the Caller's process boundary.

When the Caller and Executor are not co-located, the transport carrying canonical payloads to the Caller MUST use TLS or an equivalent confidentiality and integrity layer.

A Caller MAY decline to sign -- for example, if the Caller does not consent to the receipt's contents. In that case the Executor publishes the receipt with only its own signature and no callerSignature. Such receipts remain syntactically valid; consumers may weight them differently as a matter of deployment policy.

5. Failure Type Classification

When success is false, failureType MUST be one of:

Value	Condition
timeout	The call exceeded a deployment-defined latency bound (default RECOMMENDED: 30000 ms), or the underlying error was timeout-shaped.
validation	The call failed due to input or output validation (schema, parse, type mismatch).
error	All other failures.

Table 2

failureType MAY be extended by deployments with additional values. Receiving implementations MUST treat unknown failureType values as error for the purposes of any deployment-policy decision they make.

When success is true, failureType MUST be the empty string. This is a deliberate choice over a null value: it keeps the canonical payload's value type stable (always string) so that JCS canonicalization produces a predictable byte sequence regardless of success state. A verifier that substitutes a null value for an empty failureType will compute a different canonical payload and will fail to verify legitimate receipts.

6. Tool Metadata (Optional)

A receipt MAY carry a toolMetadata object describing class or capability hints about the tool. This document does not standardize the schema of toolMetadata. A deployment may use it to convey:

* A tool class (e.g., advisory, data-retrieval, mutation, settlement).
* A settlement layer identifier when the tool executes on-chain transactions.

* A verifiability hint indicating whether the tool's outcome is externally anchored.

toolMetadata is NOT part of the canonical payload and is NOT signed. Consumers that wish to trust toolMetadata MUST validate it through out-of-band means (e.g., the tool's published manifest, signed separately).

A future revision of this document, or a companion document, MAY standardize a portion of the toolMetadata schema if interoperability needs emerge.

7. Identity (DID) Requirements

Both agentDid and callerDid MUST be syntactically valid DIDs per [DID-CORE]. This document does not constrain the DID method. Common choices in production include did:key, did:web, and ledger-anchored methods such as did:xrpl or did:ethr.

A deployment MAY apply policy based on DID method -- for example, treating ledger-anchored identities differently from cryptographic-only identities. Such policy is out of scope for this document; the wire format treats all DID methods uniformly.

8. Security Considerations

8.1. Privacy

Receipts identify inputs and outputs by hash. Implementations MUST NOT include raw inputs, outputs, prompts, user data, secrets, or PII in any signed field. toolMetadata, while not part of the canonical payload, also SHOULD NOT contain such data.

Hash construction matters: a deployment that hashes uncanonicalized inputs may leak information through hash collisions or correlation. Implementations SHOULD canonicalize inputs before hashing (for example, with JCS for JSON inputs).

8.2. Replay

A signed receipt is replayable by anyone who possesses it. Receivers SHOULD enforce a freshness window on timestamp and SHOULD reject duplicate receipts identified by (signature) (which is unique given the inclusion of timestamp in the canonical payload). A deployment that needs cross-receipt deduplication MAY additionally store and dedupe by (agentDid, taskHash, timestamp).

8.3. Caller-Side Forgery

A receipt with only signature (Executor) and no callerSignature represents the Executor's claim alone. A malicious Executor could fabricate such receipts. Co-signature by the Caller prevents such receipts from being accepted as caller-attested: a Caller observing a forged receipt about its own delegations would notice the absence of its callerSignature and could repudiate.

When callerSignature is missing, a deployment SHOULD weight the receipt accordingly. The exact weighting is policy, but treating co-signed and non-co-signed receipts identically is a security mistake.

8.4. Single-Observer Dominance

If a deployment derives reputation or trust signals from receipts and a single Caller produces most of the receipts about a given tool, that Caller's environment-specific bugs, biases, or hostile behavior propagate directly into the derived signal. This is a deployment-policy concern, not a wire-format concern. Deployments SHOULD record the set of distinct callerDid values contributing to any derived statistic so that consumers can reason about observer diversity.

8.5. Key Compromise

A compromised Agent or Caller key allows arbitrary receipt forgery for the lifetime of that key. DID methods that support key rotation SHOULD rotate routinely. Verifiers MUST resolve DIDs to the current key set at verification time, not at receipt emission time.

8.6. Timestamp Trust

timestamp is asserted by the Executor and is not independently anchored by this format. A deployment that requires verifiable time SHOULD pair receipts with an external time-anchoring mechanism ([RFC3161], blockchain inclusion, etc.).

9. IANA Considerations

This document has no IANA actions in its current form. A future revision may register a media type (e.g., application/xaip-receipt+json) and a failureType registry.

10. References

10.1. Normative References

[DID-CORE] Sporny, M., "Decentralized Identifiers (DIDs) v1.0", W3C Recommendation, July 2022, <https://www.w3.org/TR/did-core/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

10.2. Informative References

[RFC3161] Adams, C., Cain, P., Pinkas, D., and R. Zuccherato, "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)", RFC 3161, DOI 10.17487/RFC3161, August 2001, <https://www.rfc-editor.org/info/rfc3161>.

[XAIP-CASE-2026-05] xkumakichi, "XAIP Single-Caller Dominance Case Study", May 2026, <https://github.com/xkumakichi/xaip-protocol/blob/main/docs/case-study/single-caller-dominance.md>.

[XAIP-IMPL] xkumakichi, "XAIP Protocol Reference Implementation", 2026, <https://github.com/xkumakichi/xaip-protocol>.

Appendix A. Relationship to the XAIP Reference Implementation

The XAIP reference implementation [XAIP-IMPL] wraps this wire format with an aggregator, a Bayesian trust score, optional metadata display, risk-flag logic, and a decision engine that ranks candidate tools. None of those components are required to produce or consume receipts conformant to this document. A consumer that only wants to verify and store receipts does not need to import any of them.

A consumer that wants a turnkey aggregator and scoring layer may use the reference implementation. A consumer that disagrees with any of those design choices is free to substitute its own implementation while remaining interoperable at the receipt-format layer.

The single-observer dominance failure mode discussed earlier in this document was first surfaced in the public dataset of that reference implementation [XAIP-CASE-2026-05].

Appendix B. Adoption Path for Agent-Payment Protocols

This format is intended to be useful to agent-payment protocols (for example, agent-to-agent payment protocols, agent-mediated commerce protocols, and agent escrow systems) that need a "trust precondition" check before committing to a transaction. Such a protocol can:

1. Require that an Agent present a set of recent receipts before being allowed to initiate a payment.
2. Define its own scoring policy over the receipt set, or consult an external scoring service.
3. Require that receipts above a certain transaction value include callerSignature (co-signed).
4. Require that receipts for settlement-class tools (declared via toolMetadata) be additionally anchored to an external ledger.

Each of those is a policy decision local to the agent-payment protocol. This document only defines the receipt wire format; it does not define a payment mechanism, a settlement rail, or any value-transfer system.

Appendix C. Change Log

* -00 (2026-05-22): Initial individual draft. Split out from the XAIP reference implementation specification, focused on the receipt wire format only. Removed aggregator, scoring, and decision-engine content; left those to deployment policy.

Author’s Address

xkumakichi
Email: kuma.github@gmail.com