Kaman Security Fabric: Cryptographic Accountability for Autonomous AI Agents

Surajbhan Satpathy and the Yoctotta Research Team
surajbhan.satpathy@yoctotta.com
Kaman AI Research · March 2026

Abstract

Autonomous AI agents that invoke tools, access databases, send messages, and load plugins on behalf of users create an *accountability gap*: organizations cannot prove what an agent actually did, in what order, or whether the record has been tampered with after the fact. Existing observability systems—logs, traces, audit tables—are mutable infrastructure controlled by the platform operator, offering no guarantees against retroactive alteration. We present the *Kaman Security Fabric* (KSF), a cryptographic accountability layer that produces tamper-evident, externally verifiable proof of every consequential agent action without introducing latency into the inference path. KSF introduces three interlocking mechanisms: (1) *Signed Action Records* (SARs), Ed25519-signed, hash-chained records emitted for every tool invocation, data access, credential use, and plugin load; (2) *Merkle root anchoring*, where hourly Merkle trees over SAR batches are published to a transparency endpoint, enabling external verification without exposing individual records; and (3) *data integrity sealing*, where every data lake snapshot is Merkle-tree sealed over the SHA-256 hashes of its constituent Parquet files, linking storage integrity to the SAR chain via OpenLineage facets. The architecture is entirely additive—zero existing APIs change, no inference latency is added (all signing and submission is fire-and-forget), and the system degrades gracefully: if any KSF component fails, agent operation continues unaffected. In production deployment, KSF signs and submits SARs with sub-millisecond overhead per action, computes hourly Merkle roots over windows of up to 100,000 SARs in under 200 ms, and verifies snapshot integrity for data lakes containing thousands of Parquet files in under 5 seconds.

1 Introduction

The deployment of autonomous AI agents in enterprise environments has outpaced the accountability infrastructure needed to govern them. When an agent queries a database, sends an email, invokes a third-party API, or modifies a configuration on behalf of a user, the only record of that action typically resides in platform-controlled logs that can be silently altered, selectively deleted, or retroactively fabricated by the platform operator.

This *accountability gap* presents three concrete risks:

1. **Regulatory exposure.** Compliance frameworks (SOC 2, GDPR, HIPAA) require demonstrable proof that automated systems acted within authorized boundaries. Mutable logs do not constitute proof.
2. **Dispute resolution.** When an agent action produces unintended consequences—a misconfigured workflow, a leaked credential, an incorrect data transformation—organizations cannot reconstruct the precise sequence of events with confidence that the record is accurate.
3. **Trust deficit.** Users and administrators cannot verify that the platform's account of agent behavior matches reality. Trust is asserted, never proven.

We address these risks with the *Kaman Security Fabric* (KSF), a cryptographic accountability layer designed with three constraints:

- **Zero inference latency.** All cryptographic operations (signing, hashing, submission) are fire-and-forget. The agent's response time is unaffected.
- **Zero breaking changes.** KSF is entirely additive. No existing API signatures, database schemas, or message formats change.
- **Graceful degradation.** If any KSF component is unavailable, agent operation continues normally. Accountability is *best-effort* rather than *blocking*.

KSF integrates with the Kaman platform's existing plugin architecture [1], data lake [2], collective memory system [3], and multi-channel communication layer [4]—adding cryptographic guarantees to each without modifying their operational be-

havior.

2 Threat Model

KSF assumes a *semi-honest platform operator*: the operator follows the protocol but may attempt to retroactively alter records after the fact. This is the realistic enterprise threat model—internal actors with database access who might modify audit logs to conceal errors, attribute actions to different agents, or fabricate compliance records.

KSF does *not* defend against:

- Compromise of the platform's Ed25519 private key (mitigated by HSM/KMS storage in production).
- Real-time interception and modification of agent actions before they occur.
- Denial-of-service attacks on the Key Registry (mitigated by fire-and-forget design).

The trust hierarchy is three-tiered:

1. **Platform keypair.** A root Ed25519 keypair signs operator certificates. Stored in HSM/KMS in production, environment variable in development.
2. **Operator keypair.** Per-organization keypair, certified by the platform. Generated when an organization is provisioned.
3. **Deployment keypair.** Per-agent-instance keypair, certified by the operator. Generated at session start, used to sign all SARs for that session.

3 Architecture Overview

KSF comprises four components, each independently deployable:

1. **KsfSigner** — a per-deployment singleton in the agent runtime that generates Ed25519-signed, hash-chained Signed Action Records.
2. **KsfEmitter** — a fire-and-forget transport layer that batches SARs and publishes them to the Key Registry via NATS JetStream (primary) or HTTP (fallback).
3. **Key Registry** — a standalone FastAPI microservice that stores public keys, ingests SARs, computes Merkle roots, and serves verification endpoints.
4. **KDL Integrity** — a Python module in the data lake that computes file content hashes and snapshot Merkle roots, linking storage integrity to the SAR chain.

3.1 Data Flow

The end-to-end flow for a single agent action:

Listing 1: SAR lifecycle

```
Step 1: Agent executes tool call
Step 2: KsfSigner.sign(action_type, payload)
  - Hash payload (SHA-3-256, sorted keys)
  - Sign hash + prev_hash with Ed25519
  - Increment sequence counter
  - Return SAR object
Step 3: KsfEmitter.emit(sar)
  - Buffer SAR (max 50, max 500ms)
  - Publish batch to NATS ksf_sar subject
Step 4: Key Registry ingests SAR
  - Verify Ed25519 signature
  - Insert into sar_ledger (append-only)
Step 5: Hourly Merkle worker
  - Compute Merkle root over window
  - Platform signs the root
  - Publish to transparency endpoint
```

Every step after Step 1 is asynchronous and non-blocking. The agent never waits for cryptographic operations to complete.

4 Signed Action Records

A Signed Action Record (SAR) is the atomic unit of accountability in KSF. Every consequential agent action produces exactly one SAR.

4.1 SAR Structure

Listing 2: SAR record fields

```
action_id UUID v4 (globally unique)
deployment_id UUID (signing agent instance)
operator_id UUID (owning organization)
action_type Enum (10 types, see below)
payload_hash SHA-3-256 of canonical payload
payload_preview First 120 chars (human-readable
    )
signature Ed25519(private_key, payload_hash
             || prev_hash || sequence)
prev_hash SHA-3-256 of previous SAR
sequence Monotonic counter (per deployment)
created_at ISO 8601 timestamp
```

4.2 Action Types

KSF defines ten action types covering all consequential agent behaviors:

Action Type	Trigger
TOOL_INVOKE	Tool call completed in LangGraph
DATA_ACCESS	KDL API call (read or write)
CHANNEL_SEND	Message sent to external channel
CAML_OBSERVE	Observation shared to collective memory
CAML_RECALL	Insight retrieved from collective memory
SESSION_START	Agent session initialized
SESSION_END	Agent session terminated
CREDENTIAL_USE	Credential fetched from Auth Service
PLUGIN_LOAD	Plugin loaded and verified
CONFIG_CHANGE	Configuration modified

Table 1: SAR action types and their triggers.

4.3 Hash Chain Integrity

SARs form a per-deployment hash chain: each SAR's signature covers the `prev_hash` of the preceding SAR, creating an append-only sequence where any insertion, deletion, or reordering is detectable.

The chain property is:

$$\text{sig}_i = \text{Ed25519}(k,\ H(\text{payload}_i) \parallel H(\text{SAR}_{i-1}) \parallel i)$$

where k is the deployment's Ed25519 private key, H is SHA-3-256, and i is the sequence number. The first SAR in a chain uses a zero hash as `prev_hash`.

Verification requires only the deployment's public key (available from the Key Registry) and proceeds in $O(n)$ time by walking the chain forward.

4.4 Payload Hashing

Payloads are canonicalized before hashing: keys are recursively sorted, values are JSON-serialized, and the resulting string is SHA-3-256 hashed. This ensures deterministic hashes regardless of key ordering in the source language (critical for cross-language compatibility between the TypeScript agent runtime and the Python Key Registry).

5 Key Registry Service

The Key Registry is a standalone FastAPI microservice responsible for key management, SAR ingestion, Merkle root computation, and verification.

5.1 Database Schema

The Key Registry uses four tables in a dedicated `ksf` PostgreSQL schema:

- **`ksf.public_keys`** — Keypair registry with entity ID, entity type (platform/operator/deployment), public key (BYTEA), certificate signature, parent ID, and status (active/revoked).
- **`ksf.sar_ledger`** — Append-only SAR storage. The database role has `INSERT` and `SELECT` privileges only—no `UPDATE` or `DELETE`.
- **`ksf.merkle_roots`** — Hourly Merkle roots with window boundaries, SAR count, and platform signature.
- **`ksf.plugin_manifests`** — Plugin integrity records with manifest hash, developer signature, and platform countersignature.

5.2 SAR Ingestion

SARs arrive via two channels:

1. **NATS JetStream** (primary) — A consumer subscribes to `events.ksf_sar`, buffers messages (up to 50, max 500 ms), and bulk-inserts into `sar_ledger`. Signature verification occurs at ingestion.
2. **HTTP batch endpoint** (fallback) — `POST /sar/submit/batch` accepts up to 50 SARs per request with base64-encoded signatures. Client ID ownership is enforced: users can only submit SARs for their own organization.

5.3 Verification Endpoints

The Key Registry exposes verification endpoints that require no cryptographic knowledge from the caller:

- `GET /sar/verify/{action_id}` — Returns the SAR with its signature and chain validity status.
- `GET /sar/deployment/{id}` — Paginated activity feed for a deployment.
- `GET /sar/deployment/{id}/summary` — Trust state, total SARs, chain integrity, last action timestamp.
- `GET /compliance/status` — Aggregated health indicators (data integrity, agent accountability, plugin security, audit trail completeness).

- `GET /merkle/transparency` — Public endpoint serving the latest Merkle roots for external verification.

6 Merkle Root Anchoring

Individual SAR verification proves that a specific action was signed by a specific deployment. Merkle root anchoring provides a stronger guarantee: that the *set* of all SARs in a time window has not been altered.

6.1 Computation

A background worker in the Key Registry computes Merkle roots on a configurable interval (default: 1 hour):

1. Query all SARs in the window $[t_{\text{start}}, t_{\text{end}})$.
2. Compute leaf hashes: $\ell_i = \text{SHA-256}(\texttt{action\_id}_i \,\|\, \texttt{payload\_hash}_i \,\|\, \texttt{signature}_i)$.
3. Sort leaves lexicographically (ensures determinism).
4. Build binary Merkle tree: pair adjacent nodes, hash concatenations, repeat until a single root remains. Odd-length levels duplicate the last node.
5. Platform signs the root: $\text{sig} = \text{Ed25519}(k_{\text{platform}}, \text{root})$.
6. Store root, signature, window boundaries, and SAR count.

6.2 Merkle Proofs

The Key Registry can generate inclusion proofs for individual SARs: the $O(\log n)$ sequence of sibling hashes needed to reconstruct the root from a leaf. This allows external parties to verify that a specific SAR was included in a published root without accessing the full dataset.

6.3 Transparency Endpoint

The transparency endpoint (`GET /merkle/transparency`) serves the last 720 roots (30 days of hourly roots) as a static JSON document. This endpoint is intentionally public and unauthenticated—it contains no sensitive data (only hashes and signatures) and enables independent verification by auditors, regulators, or automated compliance tools.

7 Plugin Integrity Verification

The Kaman plugin architecture [1] supports an extensible ecosystem of MCP-based plugins. KSF adds integrity verification to the plugin lifecycle, ensuring that plugins have not been tampered with between publication and runtime loading.

7.1 Signing Workflow

1. **Developer signs.** The plugin developer computes a SHA-3-256 hash of the plugin manifest (with keys recursively sorted for determinism) and signs it with their Ed25519 key.
2. **Platform countersigns.** On marketplace publication, the platform verifies the developer signature and adds its own countersignature.
3. **Storage.** Both signatures and the manifest hash are stored in the `plugin` table and registered with the Key Registry.

7.2 Runtime Verification

When the NATS consumer processes an `extension_saved` event (indicating a new or updated plugin), KSF performs verification before the plugin is made available for tool calls:

1. Compute the manifest hash of the installed plugin.
2. Fetch the registered hash and signatures from the Key Registry.
3. Compare hashes. If they differ, the plugin has been modified.
4. Verify developer and platform signatures.
5. Emit a `PLUGIN_LOAD` SAR recording the verification result.

If verification fails, the plugin is still loaded (to avoid breaking existing deployments) but the SAR records the failure, providing an audit trail of integrity violations.

8 Data Lake Integrity

KSF extends the Kaman Data Lake (KDL) [2] with file-level content hashing and snapshot-level Merkle roots, creating a verifiable link between the SAR chain and the actual data stored in object storage.

8.1 File Content Hashing

After every write operation (insert, update, or delete), KSF computes SHA-256 hashes of the Parquet files in the latest snapshot. Rather than downloading files from S3, hashing is performed through DuckDB's built-in `parquet_scan()` and `sha256()` functions, which read files directly from object storage via the httpfs extension.

Listing 3: DuckDB file content hashing

```
SELECT sha256(
  string_agg(col_data, '|' ORDER BY col_data)
) FROM (
  SELECT columns(*)::VARCHAR AS col_data
  FROM parquet_scan('s3://bucket/file.parquet')
)
```

8.2 Snapshot Merkle Root

File hashes for a snapshot are assembled into a Merkle tree using the same algorithm as SAR Merkle roots (Section 6): sorted leaves, binary tree construction, odd-level duplication. The resulting root is stored alongside the snapshot metadata.

8.3 Verification API

KDL exposes two verification endpoints:

- `GET /{lake}/verify/snapshot/{id}` — Recomputes file hashes and Merkle root, compares against stored values, and returns a verification result with status (`VERIFIED`, `TAMPERED`, `NO_INTEGRITY_DATA`, or `VERIFICATION_ERROR`).
- `GET /{lake}/integrity/snapshot/{id}` — Returns stored integrity data (Merkle root, file count, computation timestamp) with optional file-level hash listing.

8.4 OpenLineage Integration

KSF extends the platform's OpenLineage event model with two custom facets:

- **`ksf_sar`** — Links a lineage event to its corresponding SAR via `sar_action_id`.
- **`ksf_snapshot`** — Includes the snapshot Merkle root and snapshot ID, enabling lineage consumers to verify data integrity at the point of observation.

These facets follow the OpenLineage custom facet specification with `_producer` and `_schemaURL` fields, ensuring compatibility with standard OpenLineage consumers (Marquez, Datahub, etc.) that simply ignore unknown facets.

9 CAML Integration

The Collective Agent Memory Layer (CAML) [3] enables agents to share validated learnings across deployments. KSF upgrades CAML's authentication model from HMAC-SHA256 to Ed25519 deployment signatures, creating a verifiable link between collective memory operations and the SAR chain.

9.1 SAR-Linked Observations

When an agent submits an observation to CAML, it first creates a `CAML_OBSERVE` SAR and includes the `sar_action_id` in the observation request. The CAML gateway verifies that the SAR exists, was signed by the correct deployment, and has a valid chain—adding reputation signals based on the result:

- SAR valid and chain intact: no reputation penalty.
- SAR missing or invalid signature: −20 reputation points.
- SAR from wrong deployment: −25 reputation points.

9.2 Migration Strategy

The transition from HMAC-SHA256 to Ed25519 follows a three-phase migration with configurable grace periods:

1. **Phase A** (current): `sar_action_id` is optional. Missing IDs are logged as warnings.
2. **Phase B**: Required for new deployments. Existing deployments have a 30-day grace period (configurable via `CAML_KSF_GRACE_DAYS`).
3. **Phase C**: Hard requirement for all deployments.

10 Agent Runtime Integration

KSF integrates with the agent runtime through lightweight, non-invasive hooks at existing event boundaries. No agent node logic changes; SARs are emitted as fire-and-forget side effects.

10.1 Hook Points

Location	SAR Type
`toolCallNode.ts`	`TOOL_INVOKE`
`agentNew.ts`	`SESSION_START/END`
`camlClient.ts`	`CAML_OBSERVE/RECALL`
`kdlTools.ts`	`DATA_ACCESS`
`authCredentialsService.ts`	`CREDENTIAL_USE`
`natsConsumer.ts`	`PLUGIN_LOAD`

Table 2: SAR emission hook points in the agent runtime.

Each hook follows the same pattern:

Listing 4: Fire-and-forget SAR emission

```
# After tool/action completes:
try {
  ksfEmit(ACTION_TYPE, payload, preview)
} catch { /* KSF is non-critical */ }
```

The `try/catch` ensures that KSF failures never propagate to the agent's operational path. The dynamic import pattern (`await import("../ksf/index.js")`) ensures the KSF module is only loaded if it exists, supporting deployments where KSF is not installed.

10.2 KsfSigner Lifecycle

The KsfSigner is instantiated per deployment (agent session) and cached for the session's lifetime:

1. On first SAR emission for a deployment, a new Ed25519 keypair is generated.
2. The public key is registered with the Key Registry (fire-and-forget).
3. The signer maintains an in-memory sequence counter and `prev_hash`.
4. On session end, `ksfFlush()` drains the emission buffer.

10.3 Emission Batching

The KsfEmitter buffers SARs and flushes on two triggers: batch size (50 SARs) or time (500 ms since first buffered SAR). This amortizes NATS publish overhead while bounding latency.

If NATS is unavailable, the emitter falls back to HTTP batch submission to the Key Registry. If both transports fail, SARs are logged to the console and dropped—consistent with the graceful degradation design.

11 UX Design Philosophy

KSF's cryptographic infrastructure is never exposed to end users. The UI translates complex guarantees into three intuitive concepts:

User Need	UI Surface	Mental Model
What did my agent do?	Activity Feed	Bank statement
Can I prove it?	Proof Export	Notary stamp
Is everything OK?	Trust Badge	Traffic light

Table 3: UX mapping from cryptographic guarantees to user mental models.

11.1 Language Rules

The following terms are never surfaced to users: "cryptographic signature," "Merkle root," "Ed25519," "hash," "SAR," "ledger," "countersigned." Instead:

- Signatures become "verified" or "certified."
- Merkle roots become "public record" or "proof anchor."
- SARs become "actions" or "activity."
- Chain integrity failure becomes "issue detected — review needed."
- Revoked keypair becomes "agent suspended."

11.2 Trust Badge

A three-state badge displayed on every agent card:

- **Green (Verified)**: All recent SARs valid, chain intact.
- **Amber (Review needed)**: PII rejection, loop detected, or SAR gap.
- **Red (Suspended)**: Keypair revoked or integrity failure detected.

11.3 Compliance Dashboard

A single-page view for compliance managers displaying four health indicators (0–100% scores), summary statistics (total SARs, registered plugins, last verification timestamp), and a table of recent Merkle roots. Each indicator links to a filtered activity feed for drill-down investigation.

12 Performance

Operation	Latency
Ed25519 sign (per SAR)	<1 ms
SHA-3-256 payload hash	<0.1 ms
NATS batch publish (50 SARs)	∼2 ms
Merkle root (100K SARs)	∼180 ms
Snapshot verification (1K files)	∼3 s
SAR chain verification (1K SARs)	∼50 ms
Plugin manifest hash	<1 ms

Table 4: KSF operation latencies.

The critical metric is **added inference latency**: zero. All KSF operations execute after the agent action completes, in a separate async context. The agent's response to the user is never delayed by signing, hashing, or network submission.

Storage overhead is modest: each SAR record is approximately 500 bytes in PostgreSQL. At 10,000 agent actions per day (a busy enterprise deployment), the `sar_ledger` grows by approximately 5 MB/day or 1.8 GB/year—negligible compared to the data lake storage it protects.

13 Security Analysis

13.1 Tamper Detection

Any modification to a stored SAR is detectable:

- **Content modification**: The Ed25519 signature over the payload hash will not verify.
- **Deletion**: The chain breaks—the next SAR's `prev_hash` will not match.
- **Insertion**: Sequence numbers will conflict or the chain will fork.
- **Reordering**: Sequence numbers and `prev_hash` linkage will be inconsistent.

13.2 Client ID Enforcement

SAR submission endpoints enforce ownership: authenticated users can only submit SARs with their own organization's `client_id`. Cross-organization SAR injection returns HTTP 403.

13.3 Append-Only Ledger

The `sar_ledger` table's database role has `INSERT` and `SELECT` privileges only. Even if an attacker gains access to the Key Registry database, they cannot modify or delete existing SARs through the application layer.

13.4 External Verifiability

The Merkle root transparency endpoint enables verification without platform cooperation: an external party can independently verify that a specific SAR was included in a published root by checking the Merkle proof against the platform-signed root hash.

14 Related Work

Certificate Transparency [5] introduced the concept of publicly auditable, append-only logs with Merkle tree anchoring for TLS certificate issuance. KSF adapts this model to AI agent actions, replacing certificate issuance events with Signed Action Records.

Blockchain-based audit trails (Hyperledger Fabric [6], Ethereum [7]) provide tamper-evident ledgers but introduce consensus overhead incompatible with the sub-millisecond latency requirements of AI agent orchestration. KSF achieves equivalent tamper-evidence guarantees without distributed consensus by using a centralized-but-verifiable model: the platform operates the ledger, but external parties can verify its integrity through Merkle proofs and Ed25519 signatures.

Sigstore [8] provides keyless signing for software artifacts using ephemeral certificates and transparency logs. KSF shares the transparency log concept but uses long-lived deployment keys rather than ephemeral certificates, as agent sessions span minutes to hours and require chain continuity.

W3C Verifiable Credentials [9] standardize the format for cryptographically verifiable claims. KSF's SAR structure is conceptually compatible but optimized for high-throughput, append-only recording rather than credential exchange.

15 Conclusion

KSF demonstrates that cryptographic accountability for autonomous AI agents is achievable without performance penalties, breaking changes, or user-facing complexity. By combining Ed25519-signed hash chains with Merkle root anchoring

and data integrity sealing, KSF provides three guarantees that mutable logs cannot:

1. **Tamper-evidence**: Any modification to the action record is cryptographically detectable.
2. **External verifiability**: Third parties can verify the integrity of the action record without platform cooperation.
3. **Data-action linkage**: Storage integrity is cryptographically linked to the action chain, proving not just what agents did but that the data they operated on has not been altered.

As autonomous agents assume greater responsibility in enterprise operations—executing workflows, managing data pipelines, interacting with customers—the ability to prove what happened becomes not merely useful but legally and operationally essential. KSF provides this proof infrastructure as an invisible, zero-cost addition to the platform.

References

[1] Kaman AI Research. Unified plugin architecture for enterprise AI agents: A Model Context Protocol approach. Technical report, Kaman, 2026. https://kaman.ai/papers/mcp-plugin-architecture.pdf

[2] Kaman AI Research. KDL: A version-native data lake architecture for AI-driven enterprise analytics. Technical report, Kaman, 2026. https://kaman.ai/papers/kdl-data-lake.pdf

[3] Kaman AI Research. KMMS & CAML: Hierarchical memory and collective intelligence for enterprise AI agents. Technical report, Kaman, 2026. https://kaman.ai/papers/kmms-caml-whitepaper.pdf

[4] Kaman AI Research. Omnichannel communication architecture for autonomous AI agents. Technical report, Kaman, 2026. https://kaman.ai/papers/multi-channel-architecture.pdf

[5] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. *RFC 6962*, 2013.

[6] E. Androulaki et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of EuroSys*, 2018.

[7] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[8] The Sigstore Authors. Sigstore: Software signing for everybody. https://sigstore.dev, 2022.

[9] W3C. Verifiable credentials data model v1.1. W3C Recommendation, 2022.