Kaman 3.0: Architecture of an Enterprise AI Agent Platform

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

Abstract

Enterprise AI agent platforms must simultaneously solve problems that span the full system stack: routing prompts to cost-optimal language models, managing context windows that overflow during long sessions, discovering and invoking tools from catalogs of hundreds of functions, ingesting and versioning data from dozens of heterogeneous sources, accumulating and sharing knowledge across agent deployments, delivering responses across sixteen communication channels, and proving—cryptographically—that every agent action is tamper-evident and auditable. Solving any one of these problems in isolation is well-studied; solving all of them in a single, coherent, production-grade system is the engineering challenge that defines the enterprise AI agent category.

This paper presents the architecture of Kaman 3.0, an enterprise AI agent platform deployed in production since 2025. Rather than treating each subsystem as independent, we describe how eight architectural layers interact: an LLM routing layer that abstracts nine providers behind an OpenAI-compatible API; a LangGraph-based agent core with adaptive context management; a semantic tool discovery system that replaces static tool binding with dynamic, budget-aware retrieval; a Model Context Protocol (MCP) plugin architecture supporting 39+ connectors; a version-native data lake with time-travel queries; a hierarchical memory system with cross-deployment collective intelligence; an omnichannel communication layer; and a cryptographic accountability fabric that produces externally verifiable proof of every agent action.

Each layer is described in a dedicated companion paper; this paper focuses on the interactions, trade-offs, and architectural invariants that hold the system together.

1 Introduction

Building an AI agent that can answer questions is straightforward. Building an AI agent platform that enterprises trust with their data, their customers, and their compliance obligations requires solving a set of interconnected systems problems that no single research contribution addresses.

Consider a concrete scenario: a sales operations agent receives a request via Slack to generate a quarterly revenue report. This single request triggers a cascade of system interactions:

1. The message arrives through the **channel layer**, which normalizes platform-specific payloads and resolves session identity.
2. The **LLM router** classifies the prompt's complexity and selects a cost-appropriate model from nine available providers.
3. The **context manager** assembles the prompt from conversation history, system instructions, and recalled memories, compressing older messages to fit the token budget.
4. The agent decides it needs a SQL tool and a charting tool. The **tool discovery** system retrieves them from a catalog of hundreds of functions using semantic search.
5. Tool execution queries the **data lake**, which serves the latest snapshot of the revenue table with sub-second latency via DuckDB.
6. The data lake write path ingests the generated report through an **MCP connector**, versioning it as a new snapshot.
7. The agent shares a learned pattern ("Q3 reports always need regional breakdowns") with **collective memory**, where other agents across the organization can recall it.
8. Every action—tool call, data access, credential use, message sent—is signed and chained by the **security fabric**, producing a tamper-evident audit trail.
9. The response streams back through the channel layer with platform-appropriate formatting.

Each step involves a subsystem that is the subject of a dedicated technical paper in the Kaman research series. This paper describes how these subsystems compose into a coherent platform.

1.1 Companion Papers

This paper references eight companion papers, each providing deep technical detail on a specific architectural layer:

Layer	Paper
LLM Routing	Intelligent LLM Routing [1]
Context Mgmt	Adaptive Context Management [2]
Tool Discovery	Semantic Tool Discovery [3]
Plugin System	MCP Plugin Architecture [4]
Data Lake	KDL Data Lake [5]
Memory	KMMS & CAML [6]
Channels	Multi-Channel Architecture [7]
Security	Kaman Security Fabric [8]

Table 1: Kaman research paper series.

2 Design Principles

Eight principles guide architectural decisions across all subsystems:

1. **Never block inference.** No supporting system—security, lineage, memory, telemetry—may add latency to the agent's response path. All cross-cutting concerns are fire-and-forget.
2. **Degrade gracefully.** Every subsystem must function when its dependencies are unavailable. The agent answers questions even if the Key Registry is down, the memory service is unreachable, or the data lake is overloaded.
3. **Additive over breaking.** New capabilities are added without modifying existing API contracts. Zero breaking changes across all eight subsystems.
4. **Multi-tenant by default.** Every query, every cache, every queue filters by `client_id`. Tenant isolation is not a feature; it is an architectural invariant.
5. **Provider-agnostic.** No subsystem depends on a specific LLM provider, cloud vendor, or message broker. Provider abstraction layers insulate the core from vendor lock-in.
6. **Budget-aware.** Context windows, token costs, API rate limits, and storage quotas are first-class resource constraints, not afterthoughts.
7. **Auditable by default.** Every consequential action produces a signed record. Auditability is not opt-in.
8. **Composition over monolith.** Each layer is independently deployable, testable, and scalable. Inter-layer communication uses NATS JetStream for async events and HTTP for synchronous queries.

3 System Architecture

Kaman 3.0 is organized into eight architectural layers, each with a clear responsibility boundary and well-defined interfaces to adjacent layers.

3.1 Layer Stack

Layer	Components
Channels	16 platforms: Slack, Teams, WhatsApp, Email, Telegram, …
Frontend	React / Next.js: Marketplace, Chat, Workflow Editor
Agent Core	LangGraph StateGraph: Context Mgmt, Tool Discovery, Think
LLM Router	OpenAI-compatible API: 9 providers, complexity classification
Plugin System	MCP: 39+ connectors, transport abstraction, sandboxed execution
Data Lake	KDL: DuckDB + S3 + PostgreSQL, versioning, time travel, lineage
Memory	KMMS 5-layer hierarchy + CAML collective intelligence, PII scanning
Security Fabric	KSF: SAR signing, Merkle anchoring, plugin integrity verification
Infrastructure	PostgreSQL, Redis, NATS JetStream, S3/MinIO

Table 2: Kaman 3.0 layer stack from top (user-facing) to bottom (infrastructure).

3.2 Inter-Layer Communication

Layers communicate through two mechanisms:

- **Synchronous (HTTP/gRPC)**: Used when the calling layer needs an immediate result. Examples: agent → LLM router (inference), agent → data lake (query), frontend → Key Registry (verification).
- **Asynchronous (NATS JetStream)**: Used for events, telemetry, and cross-cutting concerns.

Examples: tool call → SAR emission, data write → lineage event, plugin save → integrity check.

NATS JetStream serves as the platform's event backbone, carrying SAR submissions, OpenLineage events, plugin synchronization, schema change notifications, and chat stream delivery. All async events follow a fire-and-forget pattern: the producer never waits for consumer acknowledgment on the inference path.

4 LLM Routing

The LLM routing layer [1] abstracts nine providers (OpenAI, Anthropic, Groq, Azure, AWS Bedrock, Google, Ollama, vLLM, xAI/Cerebras) behind an OpenAI-compatible API surface. The agent core and all tool implementations issue standard `/v1/chat/completions` requests; the router handles provider selection, failover, and cost optimization transparently.

4.1 Adaptive Complexity Classification

The router extracts lexical, syntactic, and domain features from each prompt and classifies complexity into tiers. Simple prompts (factual lookups, formatting requests) are routed to fast, inexpensive models; complex prompts (multi-step reasoning, code generation, ambiguous instructions) are routed to capable models. This achieves 35–55% cost reduction compared to routing all requests to the most capable model.

4.2 Multi-Scope Model Management

Models are organized in four scopes with priority resolution: Private > Role > Client > Global. This allows organizations to configure model access at any granularity—from platform-wide defaults to per-user preferences—without modifying the agent core.

4.3 Automatic Failover

Each model has a fallback chain (e.g., Claude → Anthropic API → AWS Bedrock). Circuit breakers track provider health with exponential backoff. The system maintains 99.95% availability through automatic rerouting, compared to 99.5% with single-provider deployments.

5 Agent Core and Context Management

The agent core is implemented as a LangGraph StateGraph with PostgreSQL checkpointing for resumable sessions. The execution graph follows a cycle:

`Think` → `Fetch Tools` → `Call LLM` ↔ `Tool Call` → `Suggest` → `Summarize`

5.1 Adaptive Context Management

The context management system [2] treats the LLM's context window as a managed resource. A pre-flight checking node estimates token consumption before every LLM invocation and triggers compression when utilization exceeds 85%.

Compression is tiered: the most recent 5 messages are preserved verbatim, older messages are summarized by an LLM call, and the oldest summaries are further condensed. This achieves 3:1 to 8:1 compression ratios while sustaining sessions exceeding 100 turns.

5.2 Structured Thinking

The Think node produces structured reasoning artifacts with confidence scores, reducing redundant chain-of-thought across turns. High-confidence conclusions are cached and reused; low-confidence conclusions trigger re-evaluation.

5.3 Sub-Agent Orchestration

Complex tasks are decomposed into sub-agents that execute in isolated context scopes. Each sub-agent has its own token budget, tool set, and conversation history. Results are aggregated into the parent context as structured summaries, preventing context pollution from specialized tasks.

6 Dynamic Tool Discovery

The tool discovery system [3] replaces static tool binding (which consumes $O(N)$ tokens for N tools) with a single meta-tool—`search_tools`—that retrieves relevant tools on demand via semantic search.

6.1 Multi-Tier Retrieval

Tool retrieval proceeds through three tiers:

1. **Expert-configured tools**: Keyword match against tools explicitly assigned to the current

agent.

2. **System tools**: Keyword match against platform-provided tools (KDL queries, workflow operations).
3. **Database tools**: Vector similarity search over the full function catalog using pgvector embeddings.

Results are merged, deduplicated, and bound to the LLM's tool context. An LRU cache with always-include sets ensures frequently used tools remain available without re-retrieval.

6.2 Loop Detection

When the agent calls the same tool repeatedly with similar arguments, the system evicts the tool from the bound set (forcing re-discovery) and appends guidance suggesting alternative approaches. This eviction-based strategy avoids hard blocking, which would prevent legitimate retry scenarios.

Token consumption drops from 42,000+ (static binding at 100 tools) to 3,000–5,000 at steady state, freeing 80–90% of the context budget for conversation history and reasoning.

7 Plugin Architecture

The plugin system [4] implements a hub-and-spoke architecture based on the Model Context Protocol (MCP), reducing the $O(N \times M)$ integration problem (N capabilities $\times$ M external systems) to $O(N + M)$.

7.1 Transport Abstraction

Four wire protocols are supported: STDIO (for local plugins), HTTP (stateless APIs), SSE (streaming), and WebSocket (bidirectional). The transport layer provides unified lifecycle management, connection pooling, and failure recovery across all protocols.

7.2 Credential Management

Credentials are organized in three scopes (user $>$ team $>$ organization) with automatic OAuth token refresh. Template-based injection inserts credentials into plugin requests without exposing them to the agent's context window or the LLM.

7.3 Sandboxed Execution

Plugin code executes in a sandboxed environment with controlled access to HTTP, database, and LLM clients. Execution timeouts, resource limits, and audit logging ensure that plugins cannot compromise platform stability or security.

The Kaman Security Fabric [8] verifies plugin integrity at load time by comparing manifest hashes against registered signatures, and emits a `PLUGIN_LOAD` SAR for every plugin activation.

8 Data Lake

The Kaman Data Lake (KDL) [5] is a version-native lakehouse combining three co-designed tiers:

- **PostgreSQL catalog**: Transactional metadata via the DuckLake extension. Each lake has its own metadata database.
- **DuckDB engine**: Embedded columnar engine for sub-second analytical queries with connection pooling and adaptive memory allocation.
- **S3/MinIO storage**: Immutable Parquet files in object storage, providing durability and cost-effective long-term retention.

8.1 Version Control

Every write (insert, update, delete) produces a new immutable snapshot. Time-travel queries (`AT (VERSION => n)`) enable point-in-time reconstruction of any historical state. Change tracking computes insertions, deletions, and modifications between arbitrary snapshot pairs.

8.2 Data Ingestion

External data sources are ingested through MCP connectors [4] with schema detection, column mapping, and incremental synchronization. A Redis-based write buffer batches operations (flushing every 5 seconds or 1,000 operations) to amortize the overhead of Parquet file creation.

8.3 Data Integrity

The security fabric [8] seals every snapshot with a Merkle root computed over SHA-256 hashes of constituent Parquet files. Verification endpoints enable on-demand integrity checks that recompute hashes and compare against stored values, detecting any post-write tampering.

8.4 Lineage

KDL publishes OpenLineage events for every data operation, capturing dataset-level and column-level lineage. The security fabric extends these events with custom facets linking lineage to the SAR chain and snapshot integrity data.

9 Memory Architecture

The memory system [6] operates at two scales: individual agent memory (KMMS) and collective intelligence (CAML).

9.1 KMMS: Individual Memory

KMMS implements a 5-layer cognitive hierarchy:

- **L0 (Availability Index)**: Redis bitmap indicating which layers contain data for a given key. Sub-millisecond lookups.
- **L1 (Abstract Concepts)**: High-level domain understanding stored in Neo4j. Generated by LLM-driven consolidation from lower layers.
- **L2 (Core Concepts)**: Structured facts with provenance, stored in Neo4j + pgvector.
- **L3 (Example Summaries)**: Condensed interaction summaries with entity extraction.
- **L4 (Actual Examples)**: Verbatim episodes stored as vector embeddings in pgvector.

Retrieval proceeds top-down (L0 $\rightarrow$ L4), stopping when confidence exceeds the sufficiency threshold. Consolidation proceeds bottom-up (L4 $\rightarrow$ L1), with LLM-driven summarization, fact extraction, contradiction detection, and abstraction generation.

9.2 CAML: Collective Intelligence

CAML extends KMMS across independently deployed agent instances, enabling collective learning while maintaining strict privacy boundaries:

- **Observations**: Agents share validated patterns, anomalies, and domain insights through a typed observation taxonomy (6 types, 14 domains).
- **PII enforcement**: A three-stage pipeline (regex $\rightarrow$ GLiNER NER $\rightarrow$ LLM classifier) scans all observations before sharing. Rejection incurs reputation penalties.
- **Reputation**: A 0–100 score with asymmetric incentives governs recall priority. Composite recall scoring weights semantic relevance (45%), reputation (25%), validation count (20%), and recency (10%).

The security fabric [8] links CAML observations to the SAR chain: every `CAML_OBSERVE` action includes a `sar_action_id` that the CAML gateway verifies against the Key Registry.

10 Omnichannel Communication

The channel layer [7] surfaces agent capabilities across 16 communication platforms through a three-phase message lifecycle: Receive (normalize platform-specific payloads), Process (route to agent core), and Respond (format and deliver with platform-appropriate constraints).

10.1 Supported Platforms

Category	Platforms
Messaging	WhatsApp, Telegram, Discord
Collaboration	Slack, Mattermost, Microsoft Teams
Async	Email (SMTP/IMAP), Gmail, SMS (Twilio)
Voice	Alexa Skills
Scheduled	Cron
Other	Google Workspace, Odoo, Webhooks

Table 3: Supported communication channels.

10.2 Streaming Delivery

Platforms fall into three delivery categories: native streaming (WebSocket, SSE), buffered delivery (Slack, Teams—accumulate tokens and edit messages), and synchronous response (WhatsApp, Alexa—single response within timeout). The channel layer automatically selects the appropriate strategy, with graceful degradation from streaming to buffered to synchronous.

10.3 Session Continuity

Sessions are identified by composite keys (`senderId` $\oplus$ `agentId`), enabling the same user to interact with the same agent across different channels while maintaining conversation continuity. LangGraph checkpointing ensures session state survives process restarts.

11 Security Fabric

The Kaman Security Fabric (KSF) [8] provides cryptographic accountability for every consequential agent action through three interlocking mechanisms:

1. **Signed Action Records (SARs)**: Ed25519-signed, hash-chained records for every tool invocation, data access, credential use, and plugin load. Ten action types cover all consequential behaviors.

2. **Merkle root anchoring**: Hourly Merkle trees over SAR batches, platform-signed and published to a transparency endpoint for external verification.

3. **Data integrity sealing**: Every data lake snapshot is Merkle-tree sealed over file content hashes, linking storage integrity to the SAR chain via OpenLineage facets.

KSF is entirely fire-and-forget: zero added inference latency, zero breaking changes, graceful degradation on component failure. Plugin integrity verification at load time ensures that extensions have not been tampered with between publication and runtime.

12 Cross-Cutting Concerns

12.1 Multi-Tenancy

Every database query, cache key, NATS subject, and API endpoint filters by `client_id`. Tenant isolation is enforced at the infrastructure layer, not the application layer—it is architecturally impossible for one tenant's data to leak to another through any supported code path.

12.2 Authentication

A central Auth Service validates JWT tokens and provides user/organization/role context. All inter-service communication includes authentication headers. The LLM router, Key Registry, KMMS/CAML, and channel layer each validate tokens independently.

12.3 RBAC

Role-based access control governs resources at four scope levels: user, role, team, and organization. The tool discovery system [3] enforces RBAC at search time—users only discover tools they have permission to invoke. The data lake [5] enforces row-level security and column masking based on the requesting user's roles.

12.4 Observability

The Observatory subsystem captures tool call results, message flows, agent decisions, and performance metrics. OpenLineage events (extended with KSF facets) provide data lineage tracking. Structured logging with correlation IDs enables end-to-end request tracing across all eight layers.

13 Deployment Architecture

Kaman 3.0 deploys as a set of containerized services orchestrated by Docker Compose (development) or Kubernetes (production):

Service	Technology	Port
Agent Core	Node.js / Express	5000
Frontend	Next.js	3000
Marketplace	Next.js	3010
LLM Router	Python / FastAPI	8080
KDL	Python / FastAPI	8000
Key Registry	Python / FastAPI	8002
KMMS/CAML	Python / FastAPI	8003
Auth Service	Node.js	3002

Table 4: Core services and their technologies.

All services are stateless (state resides in PostgreSQL, Redis, and S3), enabling horizontal scaling behind a load balancer. The NATS message broker provides service decoupling—producers and consumers scale independently.

13.1 Infrastructure Dependencies

- **PostgreSQL** (with pgvector): Primary database for metadata, RBAC, sessions, embeddings, SAR ledger, and DuckLake catalogs.

- **Redis**: Session cache, write buffer, rate limiting, pub/sub for real-time events.

- **NATS JetStream**: Async event streaming for SAR submission, lineage, plugin sync, and chat delivery.

- **S3/MinIO**: Object storage for Parquet data files, knowledge base documents, and generated artifacts.

- **Neo4j** (optional): Graph storage for KMMS knowledge layers L1–L3.

14 Performance Characteristics

Operation	Latency
LLM routing overhead	$<$100 ms
Tool discovery (semantic search)	80 ms median
Context compression (per turn)	200–500 ms
KDL query (10M rows)	$<$1 s
KMMS recall (layered)	80 ms
Channel webhook-to-first-token	$<$1 s
SAR sign + emit	$<$1 ms
Merkle root (100K SARs)	$\sim$180 ms

Table 5: Representative latencies across system layers.

The dominant latency in any agent interaction is LLM inference (typically 1–10 seconds depending on model and prompt complexity). All other system operations—routing, tool discovery, context management, memory recall, security signing—complete within that inference window, adding no perceptible delay to the user experience.

15 Lessons Learned

Three years of production operation have yielded several architectural insights:

1. **Fire-and-forget is the only viable pattern for cross-cutting concerns.** Early designs that made security signing, lineage tracking, and memory recording synchronous introduced cascading failures when any downstream service degraded. The fire-and-forget pattern (with async retry) eliminated this class of outages entirely.

2. **Dynamic tool binding dramatically outperforms static binding.** The transition from static (all tools in context) to dynamic (on-demand retrieval) reduced context consumption by 80–90% and, counterintuitively, improved tool selection accuracy—the LLM makes better choices when presented with 5 relevant tools rather than 100 irrelevant ones.

3. **Multi-provider LLM access is table stakes.** Provider outages, rate limits, and pricing changes are frequent enough that single-provider architectures create unacceptable availability risk. Automatic failover with complexity-based routing provides both reliability and cost optimization.

4. **Version-native data is worth the complexity.** The ability to time-travel to any historical data state has proven essential for debugging agent behavior, reproducing reported issues, and satisfying compliance requirements. The overhead of snapshot-based versioning (approximately 15% additional storage) is negligible compared to its operational value.

5. **Cryptographic accountability must be invisible.** Users do not care about Ed25519 signatures or Merkle trees. They care about green checkmarks, downloadable reports, and the assurance that "what my agent did" is recorded and provable. The UX abstraction layer that translates cryptographic guarantees into familiar mental models (bank statements, traffic lights, notary stamps) is as important as the cryptography itself.

16 Conclusion

Kaman 3.0 demonstrates that the enterprise AI agent platform is not a single system but an *ecosystem* of eight co-designed subsystems, each solving a distinct problem while respecting shared architectural invariants: never block inference, degrade gracefully, isolate tenants, and prove what happened.

The platform's modular architecture—with each layer described in a dedicated companion paper—enables independent evolution: the LLM router can add new providers without touching the agent core; the data lake can change its storage format without affecting the plugin system; the security fabric can upgrade its cryptographic primitives without modifying any hook point.

As the capabilities of language models continue to expand, the engineering challenge shifts from "can the model do this?" to "can the platform safely, reliably, and accountably let the model do this at enterprise scale?" Kaman 3.0 provides one answer to that question.

References

[1] Kaman AI Research. Intelligent LLM routing: Multi-provider abstraction with adaptive complexity classification. Technical report, Kaman, 2026. https://kaman.ai/papers/intelligent-llm-routing.pdf

[2] Kaman AI Research. Adaptive context management for long-running LLM agent sessions. Technical report, Kaman, 2026. https://kaman.ai/papers/adaptive-context-management.pdf

[3] Kaman AI Research. Semantic tool discovery and context-aware binding for large language model agents. Technical report, Kaman, 2026. https://kaman.ai/papers/dynamic-tool-search.pdf

[4] 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

[5] 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

[6] 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

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

[8] Kaman AI Research. Kaman Security Fabric: Cryptographic accountability for autonomous AI agents. Technical report, Kaman, 2026. https://kaman.ai/papers/kaman-security-fabric.pdf