KMMS & CAML: Hierarchical Memory and Collective Intelligence for Enterprise AI Agents

Surajbhan Satpathy and the Yoctotta Research Team

surajbhan.satpathy@yoctotta.com

Kaman AI Research · March 2026 · KMMS: Production · CAML: Active Development

Abstract

Enterprise AI agents face two fundamental memory challenges: *individual continuity*—maintaining rich, structured context across long-running interactions—and *collective intelligence*—sharing validated knowledge across independently deployed agent instances. This paper introduces two complementary systems that address these challenges within the Kaman platform.

KMMS (Kaman Memory Management System) is a hierarchical, temporally-aware memory architecture that replaces flat conversation history with a 5-layer cognitive model spanning abstract concepts to verbatim episodes. It provides top-down retrieval with structural confidence scoring, bottom-up consolidation via LLM-driven summarization, and temporal versioning that preserves full provenance chains.

CAML (Collective Agent Memory Layer) extends KMMS into a cross-deployment shared memory substrate. Agent instances observe patterns, recall collective knowledge before acting, and validate recalled observations after acting—creating a self-reinforcing feedback loop governed by reputation scoring, PII enforcement, and cryptographic audit trails.

Together, KMMS and CAML enable a new class of enterprise agent that remembers everything it has learned, knows what other agents have learned, and can distinguish high-confidence knowledge from tentative inference—all while maintaining strict data privacy and auditability guarantees.

1 Introduction and Motivation

1.1 The Memory Problem in Enterprise AI

Modern LLM-based agents operate within a fundamental tension: they possess vast parametric knowledge from training, but have no persistent, structured memory of their operational context. Each conversation begins from zero. Each deployment is an island. This creates three categories of failure in enterprise environments:

Redundant Inference. When Agent A discovers that a procurement workflow requires a specific approval chain, Agent B—deployed by a different team or organization—must rediscover this pattern independently. Across thousands of agent instances, the same domain knowledge is re-derived millions of times.

Knowledge Decay. An agent that assists a user over months accumulates context through conversation history—often spanning multiple communication channels [1]. But conversation logs are flat, unbounded, and unstructured. As context windows fill, early interactions are lost. There is no mechanism to distill “John prefers Python for data pipelines” from hundreds of conversations spanning six months.

Coordination Blindness. Agents operating in the same domain (legal, finance, customer support) cannot benefit from each other’s discoveries. A regulatory change detected by one agent’s workflow remains invisible to all others until each independently encounters it.

1.2 Design Philosophy

KMMS and CAML are designed around five principles:

1. **Retrieval is top-down.** Start with the most abstract, confident representation. Drill into detail only when the query demands it. This mirrors human recall—you remember “John is a senior engineer” before remembering the exact conversation where he mentioned it.
2. **Consolidation is bottom-up.** New data enters

at the most granular level and propagates upward through LLM-driven summarization. Abstractions are always grounded in source material.

3. **Confidence emerges from structure.** The depth at which a retrieval resolves is itself a confidence signal. An answer from L1 (abstract concepts) carries different epistemic weight than one from L4 (verbatim episodes).
4. **Nothing is deleted.** Old versions are archived via SUPERSEDES chains. Contradictions are detected and logged, not silently overwritten. The full history of what was believed and when is always recoverable.
5. **Privacy is non-negotiable.** No observation enters the collective pool without passing a multi-stage PII scan. Audit logs are append-only with daily Merkle root computation for tamper evidence.

2 KMMS: The 5-Layer Memory Architecture

2.1 Layer Definitions

KMMS organizes knowledge into five layers, each with distinct storage characteristics, update frequencies, and retrieval semantics:

L0: Availability Index (Redis)
Ultra-fast $O(1)$ entity/topic existence check. Updated when new entities appear.
L1: Abstract Concepts (Neo4j)
1–3 sentence high-level abstractions per entity. Updated rarely—only on major conceptual shifts.
L2: Core Concepts (Neo4j + pgvector)
Structured facts with validity windows and versioning. Updated when facts change, with SUPERSEDES chains preserving history.
L3: Example Summaries (Neo4j + pgvector)
Time-scoped summaries of L4 episodes with embeddings. Updated incrementally as new L4 data arrives.
L4: Actual Examples (pgvector)
Verbatim conversation chunks, documents, raw data. Immutable after creation.

2.2 Layer-Specific Storage Strategy

Each layer maps to the database technology best suited for its access pattern:

Layer	Primary	Secondary	Key Operations
L0	Redis	—	SISMEMBER ($O(1)$), SADD
L1	Neo4j	—	Version chain, SUPERSEDES
L2	Neo4j	pgvector	Validity windows, contradictions
L3	pgvector	Neo4j	Semantic similarity, time-range
L4	pgvector	—	Immutable insert, vector search

Table 1: Storage mapping per KMMS layer.

2.3 The Knowledge Node Model

Every piece of knowledge in KMMS is represented as a KnowledgeNode—a unified model that carries layer-specific metadata:

```
class KnowledgeNode:
    # Identity
    id: UUID
    org_id: UUID # Multi-tenant isolation
    client_id: str
    content: str
    content_hash: str # SHA-256 for dedup

    # Layer System
    layer: Layer # L0..L4
    entity: str # "John", "Project X"
    topic: str # "employment", "skills"
    content_type: ContentType

    # Temporal Metadata
    event_time: datetime
    valid_from: datetime
    valid_until: datetime # null = current
    last_verified: datetime

    # Version Chain
    version: int
    status: NodeStatus # CURRENT | SUPERSEDED
    supersedes: UUID
    superseded_by: UUID

    # Confidence & Decay
    strength: float # [0,1] decays over time
    confidence: float # [0,1] epistemic
        certainty
    reinforcement_score: float
    importance: float

    # Embeddings
    embedding: List[float] # 384d (bge-small)

    # L3-Specific
    time_window: TimeWindow
    source_episode_ids: List[UUID]

    # L2-Specific
    fact_type: FactType
    fact_value: Dict
```

2.4 Entity Extraction Pipeline

When new content enters KMMS at L4, an enrichment pipeline automatically runs:

1. **GLiNER Zero-Shot NER** extracts entities (persons, organizations, products, technologies,

topics) with configurable confidence thresholds.

2. **L0 Index Update** registers newly discovered entities and topics in Redis for fast routing.
3. **Node Metadata Update** tags the L4 node with its primary entity and topic.
4. **Auto-Linking** finds semantically similar existing nodes via pgvector and queues LLM-driven relationship extraction.

This pipeline runs asynchronously via NATS JetStream, ensuring that the write path remains fast while enrichment happens in the background.

3 KMMS Retrieval: Top-Down with Confidence Scoring

3.1 The Retrieval Algorithm

Retrieval follows a top-down strategy, starting at the most abstract layer and drilling deeper only when the current layer's results are insufficient:

```
Query: "What does John think about
      the DuckDB migration?"

Step 1: Parse Query
  -> Embed query (384d vector)
  -> Detect temporal intent
  -> Identify entities via L0 index

Step 2: Route via L0
  -> SISMEMBER("John") -> exists
  -> SISMEMBER("DuckDB migration") -> exists

Step 3: Drill Layers (stop when sufficient)
  -> L1: Too abstract for this query
  -> L2: Partial answer (preferences)
  -> L3: Sufficient (weekly summary found)
  -> Return L3 results (skip L4)

Step 4: Score Confidence
  -> Base confidence for L3 = 0.85
  -> Semantic match 0.92 -> +0.05 bonus
  -> Last verified 5 days ago -> no penalty
  -> Final confidence = 0.90

Step 5: Generate Response Metadata
  -> Hedging suggestion: "Based on..."
  -> Provenance chain: [L1-id, L3-id]
```

3.2 Query Sufficiency Heuristics

The system determines whether to continue drilling based on query specificity signals:

- **Verbatim signals** (“what did he say exactly”, “quote”) → must reach L4
- **Specificity signals** (“specifically”, “precisely”) → must reach L3 or L4
- **General queries** (“tell me about”, “what do you know”) → L1 or L2 sufficient

3.3 Temporal Intent Parsing

Queries are classified by temporal intent, which affects both filtering and confidence:

Intent	Retrieval Behavior
CURRENT	Only `status=CURRENT`, `valid_until=NULL`
EVOLUTION	All versions, ordered by `valid_from`
WINDOW	Filter by `event_time` range
POINT_IN_TIME	Validity window intersection
UNSPECIFIED	Best matches across all time

Table 2: Temporal intent classification.

3.4 Confidence Scoring Model

Confidence is a composite score that communicates epistemic certainty:

Layer	Base	Meaning
L0	0.30	“I have something on this topic”
L1	0.50	“High-level understanding”
L2	0.75	“Established facts”
L3	0.85	“Based on multiple examples”
L4	0.95	“You specifically said. . . ”

Table 3: Base confidence by layer.

Modifiers: $+0.05$ for high evidence count (> 5 sources), $+0.05$ for strong semantic match (> 0.9), $+0.05$ for recent verification (< 7 days). -0.10 for stale verification (> 30 days), weak match (< 0.6), or low node strength (< 0.3). -0.30 for returning superseded content on a “current” query.

$$\text{confidence}_{\text{final}} = \text{clamp}\left(\text{base} + \sum \text{modifiers},\ 0.0,\ 1.0\right)$$

The confidence score maps to hedging language: ≥ 0.90 uses no hedging; ≥ 0.75 prefixes with “Based on our discussions. . . ”; ≥ 0.50 uses “From what I understand. . . ”; < 0.50 uses “I have limited information, but. . . ”

4 KMMS Consolidation: Bottom-Up Knowledge Synthesis

4.1 The Consolidation Pipeline

Consolidation is the process by which raw episodes (L4) are progressively distilled into structured knowledge. It runs asynchronously via NATS-backed workers:

```
New Interaction -> L4 Insert
  -> Batch Threshold Met? (3+ L4 nodes)
    -> Yes: L4->L3 Summarize Episodes
      -> Significant Change Detected?
        -> Yes: L3->L2 Extract Facts
          -> Major Conceptual Shift?
            -> Yes: L2->L1 Update Abstraction
              -> L0 Index Updated
```

4.2 L4 → L3: Episode Summarization

When enough L4 episodes accumulate for an entity/topic pair (default threshold: 3), the consolidation worker:

1. Groups episodes by time window (day, week, month, or quarter)
2. Checks for existing L3 summary covering the same window
3. If exists: **incremental merge**—LLM merges new content, reporting additions, updates, and contradictions
4. If not: **new summary**—fresh time-scoped summary
5. Generates 384-dimensional embedding for semantic search at L3
6. Evaluates whether summary contains factual statements warranting L2 extraction

4.3 L3 → L2: Fact Extraction

Fact extraction uses LLM tool calling for structured output:

```
{
  "extracted_facts": [{
    "key": "preferred_language",
    "value": "Python for data pipelines",
    "fact_type": "preference",
    "confidence": 0.85,
    "updates_existing": null,
    "is_contradiction": false,
    "source_quote": "I always use Python..."
  }]
}
```

Each extracted fact is checked against existing L2 facts:

- **New fact** → create L2 node
- **Matching fact** → reinforce (increase strength)
- **Contradicting fact** → trigger contradiction resolution

4.4 Contradiction Detection and Resolution

When a new fact contradicts an existing one, the system uses LLM classification:

1. Classify as **direct** (mutually exclusive), **temporal** (fact changed over time), or **contextual** (true in different contexts)
2. Determine resolution: **supersede**, **coexist**, or **flag_for_review**
3. For supersession: mark old fact as SUPERSEDED, create new fact with pointer, preserve full version chain

4.5 L2 → L1: Abstraction Generation

When L2 facts change significantly:

1. Gather all current L2 facts for the entity
2. LLM generates a 1–3 sentence abstraction
3. Compare with existing L1 via Jaccard similarity
4. If $<$ 70% similar: create new L1 version (SUPERSEDES chain)
5. If $\geq$ 70%: update `last_verified` only

4.6 Ontology-Guided Consolidation

KMMS supports optional ontology definitions per organization:

- **Entity types**: Valid categories for extracted entities
- **Relationship types**: Named, directed relationships with inverse names and bidirectionality flags
- **Custom prompts**: Domain-specific instructions injected into LLM consolidation prompts

5 CAML: The Collective Agent Memory Layer

5.1 The Collective Intelligence Problem

KMMS solves individual agent memory. CAML solves the problem of knowledge that should be shared *across* independently deployed agent instances—potentially across different organizations.

Consider: thousands of AI agents deployed across legal firms discover, independently, that a new regulatory change affects contract review workflows. Without CAML, each agent must encounter the regulation through user interaction, figure out implications from scratch, and develop a response pattern through trial and error. With CAML, the first agent to encounter the change *observes* it to the collective pool. Subsequent agents *recall* this observation and *validate* it after applying it—building community confidence.

5.2 Architecture: CAML as a KMMS Extension

Rather than building a separate microservice, CAML is implemented as a new route namespace within KMMS:

```
KMMS Service (Port 8081)
+-- /api/v1/memory/* # Individual memory
+-- /api/v1/search/* # Semantic search
+-- /api/v1/entities/* # Entity management
+-- /caml/v1/observe # Submit observation
+-- /caml/v1/recall # Query collective
+-- /caml/v1/validate # Report outcome
+-- /caml/v1/operators/* # Operator mgmt
+-- /caml/v1/transparency/* # Audit snapshots
```

This provides shared infrastructure (PostgreSQL, pgvector, Redis, NATS, embedding service), natural layer mapping, reusable consolidation, and consistent auth middleware.

5.3 Core Concepts

Operator: An organization deploying CAML-connected agents. Linked to a Kaman client for billing and quota management. Authenticated via a secret key issued at registration.

Deployment: A unique agent instance within an operator. Each deployment has its own key, reputation score, and audit history. Supports heterogeneous agent types.

Observation: A structured knowledge contribution to the collective pool. Typed by a taxonomy, scoped to a domain, and gated by PII scanning and reputation thresholds.

Validation: Feedback from an agent that acted on a recalled observation. Creates a closed-loop quality signal: observe → recall → act → validate.

6 CAML Observation Taxonomy and Lifecycle

6.1 Observation Types

Every observation is classified by type with distinct schema requirements:

Type	Description	Rep.	Age	Evid.
ANOMALY	Unusual pattern	0	30d	3
DOMAIN	Emerging trend	10	60d	5
WORKFLOW	Reusable task seq.	20	90d	10
EFFICIENCY	Improvement technique	20	120d	5
REGULATORY	Compliance change	40	365d	1
CONSENSUS	System-generated	—	180d	—

Table 4: Observation types with minimum reputation (Rep.), maximum age, and minimum supporting evidence (Evid.) requirements.

6.2 Domain Taxonomy

Observations are scoped to one of 14 domains: `legal`, `finance`, `procurement`, `hr`, `customer_support`, `logistics`, `manufacturing`, `healthcare`, `real_estate`, `education`, `government`, `retail`, `technology`, `general`.

6.3 Observation Lifecycle

The observation lifecycle follows a structured write path:

1. Resolve deployment identity
2. Check reputation threshold
3. Check rate limit (per-operator)
4. PII scan (3-stage cascade)
5. Embed summary + pattern (384d)
6. Write to `caml_observations`
7. Update reputation (+1)
8. Append to audit log

After entering the collective pool, other agents recall and validate observations. The validation feedback loop updates observation consensus counts and adjusts the author's reputation score.

6.4 Pattern Schemas

Each observation type carries a structured `pattern` field. For example, a WORKFLOW_PATTERN includes steps, average resolution turns, success rate, failure modes, and prerequisites. An EFFICIENCY_DELTA includes the metric, baseline value, improved value, method, and conditions.

7 PII Enforcement Pipeline

Because CAML enables cross-organization knowledge sharing, PII enforcement is a *hard gate* on the write path. No observation enters the collective pool without passing the scanner.

7.1 Three-Stage Cascade

The pipeline is designed as a cascade—each stage is more expensive but catches more subtle PII—and early exits on the cheap stages keep median latency low:

Stage 1: Regex Patterns (∼2 ms). In-process, zero external calls. Detects email addresses, phone numbers (international), Indian PAN/Aadhaar, US SSNs, credit card numbers (Luhn), IPv4 addresses, and passport numbers. False positive mitigation includes safe email domains, private IP ranges, short digit sequences, and Aadhaar length validation.

Stage 2: GLiNER Zero-Shot NER (∼50 ms). In-process model with PII-specific entity labels: person (2+ words), address (3+ words), date of birth, financial account, medical record, government ID. Threshold: 0.65 (high to reduce false positives).

Stage 3: LLM Classifier (∼200 ms). Via Model Proxy. Catches implicit PII: "the user in Mumbai who...", quasi-identifier combinations, indirect personal references, health info tied to individuals. Allows organization names, generic roles, aggregate statistics, and technical patterns.

7.2 Performance Profile

Scenario	Latency	Frequency
Clean content (passes Stage 1)	∼2 ms	∼85%
Passes Stages 1–2	∼52 ms	∼12%
All 3 stages run	∼252 ms	∼3%
PII detected at Stage 1	∼2 ms	Fast reject

Table 5: PII pipeline latency by scenario.

7.3 Rejection Consequences

PII rejection triggers: HTTP 422 with reason code (no PII details leaked), −10 reputation penalty, content hash cached for 24 hours (prevents resubmission), and 5 rejections in 24 hours triggers automatic deployment suspension with −25 reputation penalty.

8 Reputation and Trust Framework

8.1 Score Model

Reputation is a deployment-level score bounded to $[0, 100]$, designed to be *asymmetric*: slow to build, fast to lose.

Event	Delta
Observation accepted	+1
Observation validated by another agent	+3
Observation reaches consensus	+10
30-day consistency bonus	+10
Observation refuted	−5
Heavily refuted observation	−15
PII rejection	−10
Deployment suspension	−25

Table 6: Reputation event deltas.

8.2 Composite Recall Scoring

When an agent queries CAML, results are ranked by a composite score:

$$\begin{aligned} s = 0.45 &\cdot \mathrm{sim}_{\mathrm{semantic}} \\ + 0.25 &\cdot \frac{\log(1+r)}{\log(101)} \\ + 0.20 &\cdot \frac{v}{v+f+1} \\ + 0.10 &\cdot e^{-d/60} \end{aligned} \tag{1}$$

where r is author reputation, v is validation count, f is refutation count, and d is age in days. This ensures semantic relevance is the primary signal (45%), high-reputation authors rank higher (25%), community-validated observations are preferred (20%), and recent observations get a mild boost (10%).

8.3 Background Reputation Engine

A scheduled worker runs four maintenance tasks:

1. **Consistency Bonuses**: +10 for 10+ accepted writes and zero PII rejections over 30 days
2. **Consensus Promotion**: Observations with 3+ validations and ratio ≥ 0.75 promoted to CONSENSUS_SIGNAL
3. **Auto-Deactivation**: Observations with 5+ refutations and ratio ≥ 0.6 marked inactive
4. **PII Suspension**: 5+ PII rejections in 24 hours triggers automatic suspension

9 Agent Integration Architecture

9.1 LangGraph Node Placement

CAML integrates into Kaman's LangGraph agent as two new nodes:

```
START -> mam_input -> think
  -> caml_recall -> fetch_tools
  -> skill_retrieval -> check_context
  -> call_llm <-> tool_call
  -> ... -> caml_observe -> END
```

`camlRecallNode` runs after the think step. It extracts the latest user query, queries CAML with a 2-second timeout, filters by composite score (≥ 0.3), applies a token budget (1,500 tokens max), and formats results as a `<collective_memory>` system message injected before the last `HumanMessage`.

camlObserveNode runs after successful task completion, extracting structured observations fire-and-forget.

9.2 Context Injection Format

Recalled observations are injected as structured XML:

```
<collective_memory>
<observation type="WORKFLOW_PATTERN"
  domain="legal" confidence="0.87"
  composite_score="0.72">
For contract clause analysis, pre-fetching
jurisdictional context reduces resolution
from 8.5 to 3.2 turns.
</observation>
</collective_memory>
```

9.3 Resilience Guarantees

All CAML operations are non-blocking and fault-tolerant:

- **Recall**: Races against 2s timeout; returns empty on failure
- **Observe**: Fire-and-forget; failures logged, never block
- **Validate**: Same resilience pattern
- **Provisioning**: 30s backoff; agent continues without CAML

10 Audit, Transparency, and Observability

10.1 Append-Only Audit Log

Every CAML operation generates an audit log entry with deployment ID, operator ID, request hash (SHA-256), outcome, observation details, PII stage hit, reputation at write time, and latency. The application database role has `INSERT` and `SELECT` permissions only—no `UPDATE` or `DELETE`.

10.2 Daily Merkle Root Computation

Each day at 03:00 UTC, a scheduled job computes a SHA-256 Merkle root over all audit log entries for the previous day. Operators can verify that a specific entry is included via the transparency API, providing tamper-evidence without requiring a blockchain.

10.3 Prometheus Metrics

CAML exports comprehensive metrics: write path (`caml_writes_total`, `caml_write_latency_ms`), read path (`caml_recalls_total`, `caml_recall_latency_ms`), PII scanner (`caml_pii_rejections_total`, `caml_pii_scan_latency_ms`), reputation (`caml_reputation_score`, `caml_reputation_events_total`), and system gauges (`caml_active_observations`, `caml_active_deployments`).

11 Database Architecture and Storage Strategy

11.1 Multi-Database Design

KMMS uses three databases, each chosen for its strengths: **Redis** for L0 index, cache, and rate limiting; **Neo4j** for L1–L2 graph, version chains, and relationship traversals; **PostgreSQL + pgvector** for L3–L4 embeddings, CAML tables, audit logs, and reputation. **NATS JetStream** handles the job queue for consolidation, enrichment, and embedding tasks.

11.2 CAML Schema

CAML adds 8 tables, all prefixed with `caml_`:

Table	Purpose
`caml_operators`	Operator identity and tier-based quotas
`caml_deployments`	Agent instance identity and keys
`caml_observations`	Core observation data with consensus
`caml_obs_embeddings`	384d vectors (HNSW, m=16, ef=64)
`caml_audit_log`	Append-only operation log
`caml_reputation`	Per-deployment scores $[0, 100]$
`caml_rep_events`	Reputation change history
`caml_validations`	Three-outcome feedback model

Table 7: CAML database tables.

12 Performance Characteristics

Operation	P50	P95	P99
KMMS			
L0 lookup (Redis)	<1 ms	2 ms	5 ms
L4 write (embed + insert)	50 ms	120 ms	250 ms
Layered retrieval (L1–L3)	80 ms	200 ms	400 ms
Full drill to L4	120 ms	300 ms	600 ms
Consolidation L4→L3	2–5 s	8 s	15 s
Consolidation L3→L2	3–8 s	12 s	20 s
CAML			
Observe (clean)	80 ms	200 ms	350 ms
Observe (3 PII stages)	280 ms	500 ms	800 ms
Recall (semantic search)	30 ms	80 ms	150 ms
Validate	20 ms	50 ms	100 ms

Table 8: End-to-end latency profile.

Scalability: KMMS and CAML use stateless FastAPI workers behind a load balancer. Neo4j and PostgreSQL scale independently. Rate limiting via Redis ensures no single operator overwhelms the system. Background workers scale horizontally with NATS consumer groups.

13 Security Model

13.1 KMMS Security

- **Multi-tenant isolation**: All queries filter by `org_id` and `client_id`
- **Auth integration**: Token validation via Kaman Auth Service
- **RBAC**: Dataset-level permissions with scope levels (user, role, tenant)
- **Content hashing**: SHA-256 deduplication prevents data duplication

13.2 CAML Security

- **Operator auth**: HMAC-SHA256 signature validation for writes; deployment keys derived from operator secrets
- **Read auth**: JWT tokens with operator/deployment scope, 24-hour TTL
- **Replay prevention**: `(deployment_id, timestamp, body_hash)` tuples cached 10 min in Redis
- **Timestamp tolerance**: Requests within ±300 s of server time
- **Rate limiting**: Redis-based, tier-dependent (Free: 20 writes/hr; Enterprise: 100K writes/hr)
- **Audit immutability**: `INSERT+SELECT` only on audit tables
- **PII hard gate**: 3-stage scanner required; details never in rejection responses

14 Future Directions

14.1 Blockchain Migration Path

CAML's architecture is forward-compatible with decentralized governance. HMAC keys map to Ed25519 wallets, gateway middleware to smart contracts, the audit log to on-chain events, the reputation engine to on-chain contracts, and the PII pipeline to oracle wrappers. **Trigger criteria**: 20+ independent competing operators and regulatory demand for trustless governance.

14.2 Private Observation Pools

KMMS already supports per-client datasets. Extending this to CAML would allow operators to maintain a private pool alongside the collective one—observing patterns too sensitive for the public pool while still benefiting from collective recall.

14.3 Cross-Layer CAML Integration

Future work will connect CAML observations to the KMMS consolidation pipeline: raw observations → L4, consensus signals → L3, extracted domain insights → L2, and high-level domain understanding → L1.

14.4 Federated Learning from Validations

Validation signals contain implicit information about what works in which contexts. Future work will use these signals to fine-tune domain-specific models for observation quality prediction.

15 Conclusion

KMMS and CAML represent a paradigm shift in how enterprise AI agents manage knowledge. Rather than treating memory as an afterthought—flat conversation logs or simple RAG—these systems implement a structured cognitive architecture that mirrors how organizations actually build and share knowledge.

KMMS provides the individual agent with a memory that gets smarter over time. Raw interactions are automatically distilled into structured facts organized in a hierarchy that enables both fast, confident retrieval and deep, provenance-preserving drill-down. The system knows not just *what* it knows, but *how confident* it should be and *where that knowledge came from*.

CAML extends this intelligence across the network. By creating a shared knowledge substrate governed by reputation, PII enforcement, and cryptographic audit trails, it enables collective intelligence where each agent's experience makes every other agent more capable—while maintaining strict privacy boundaries and accountability.

Together, they transform AI agents from stateless function callers into knowledge-accumulating systems that learn individually, share collectively, and maintain the trust infrastructure that enterprise deployments demand.

References

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