International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(23s) 2026 pp. 1520-1534
https://doi.org/10.70917/ijcisim-2026-5709
Article

Memory Architecture for Managed Cloud Agents: A Memory-Plane Approach to Hydration, Persistence, and Forgetting

Naveen Kumar Vedurupaka

Independent Researcher, USA

Abstract: Memory" has become a catch-all term in applied agent engineering: it covers saved user facts, retained conversation history, workspace knowledge, transient session state, and the retrieval indices used to search all of it. Production systems routinely collapse this variety into a single shorthand, treating memory as retrieval-augmented generation over a vector database. This article argues that the shorthand is an architectural error and proposes an alternative account of agent memory as a memory plane: a brokered, policy-aware subsystem that hydrates prompt context on the read path and performs selective, governed persistence on the write path. Drawing on the memory subsystems exposed by managed-agent products, open-source agent frameworks, and cloud data-management primitives, the article separates a brokered control-and-data plane from backend stores, defines a five-store logical taxonomy together with an authority rule that treats the vector index as a derived structure rather than a source of truth, and specifies a broker interface built around a tombstone-cascade-verify protocol for verifiable forgetting. It frames consistency and durability as per-memory-class decisions mapped to managed services across three major cloud platforms, and it proposes an operational treatment covering engineering service-level objectives, and a low-risk migration path. The latency and service-level figures presented throughout should be read as engineering design targets rather than as measurements or vendor guarantees. The article compares the memory plane against six competing architectures and closes with a set of problems that appear to remain open, among them autonomous consolidation quality, contradiction repair at scale, and cross-agent shared memory.
Keywords: Agent Memory; Memory Plane; Retrieval-Augmented Generation; Vector Search; Distributed Systems; Consistency Models; Data Governance; Cloud Architecture; Right To Erasure

1. Introduction

1.1 An overloaded word

Ask five production teams what "agent memory" means and the answers rarely match. One team means the durable facts and preferences a system stores about a user. Another means the retained transcript of past conversations. A third points to project or workspace knowledge an organisation exposes to an agent, while a fourth describes the transient state of the current session: open tickets, partial plans, recent tool calls. A fifth, more technically minded, answers with retrieval indices, the machinery that makes any of the above searchable. These are not five phrasings of one idea; they are five distinct engineering concerns that keep surfacing separately across independent vendors. OpenAI separates saved memories, conversation history, and project memory (OpenAI, 2026a). Anthropic keeps chat search and memory summaries apart from project-scoped knowledge retrieval and from Claude Code's file-backed memory directory (Anthropic, 2026). Google's Gemini distinguishes personalisation from past chats, saved information, and connected Workspace sources (Google, 2026). When independent vendors converge on the same decomposition without coordinating, that convergence is itself a form of evidence: the underlying problem appears layered rather than monolithic (Vytla, 2024).

Yet a shorthand persists that flattens all of it into one sentence: memory is retrieval-augmented generation over a vector database. This shorthand looks like an architectural error rather than a simplification. Retrieval-augmented

generation grounds a model's output in retrieved text; agent memory is the broader problem of managing accumulated experience over the life of an agent, capturing it, consolidating it, retrieving and reusing it, correcting it, governing who may read it, and forgetting it on request. Collapsing the two treats a derived retrieval accelerator as if it were authoritative, governed state, and the practical cost shows up quickly. A fact that exists only as an embedding cannot be versioned when it changes, cannot be traced for provenance, and cannot be reliably deleted. Open-source agent frameworks draw comparable distinctions, though most leave governance, consistency, and deletion to whoever integrates them. Mem0 exposes a layered capture, promote, and retrieve pipeline (Chhikara et al., 2025). LangChain and LangGraph separate profile and collection memories from semantic search and long-term memory stored as namespaced documents (LangChain, 2026). LlamaIndex keeps a short-term buffer apart from longer-term memory blocks (LlamaIndex, 2026). AutoGen defines an explicit memory protocol (AutoGen, 2026), and Zep exposes a dedicated memory service rather than a bare index (Zep, 2026). None of these treats a vector store as the whole of memory (Ankam, 2024).

1.2 Thesis: memory as a brokered plane

This article starts from that separation and turns it into an engineering position rather than a taxonomy exercise. A managed cloud agent needs a memory plane: a brokered, policy-aware subsystem that sits between the agent's reasoning components and its storage backends, hydrating prompt context on the read path and performing selective, governed persistence on the write path. The framing lines up with the Cognitive Architectures for Language Agents account, which describes language agents as compositions of working and long-term memories together with decision procedures (Sumers et al., 2024), and with runtime designs that products and frameworks already ship in narrower form, including Claude Code's file-backed memory directory, LangGraph's namespaced store abstraction, and Mem0's capture-and-retrieve pipeline (Anthropic, 2026; Chhikara et al., 2025; LangChain, 2026). What this article adds is a treatment of the plane as a distributed-systems artefact: its stores, its interfaces, its consistency and deletion semantics, and its failure modes, specified with something close to the rigour an engineer would bring to any other stateful subsystem (Ankam, 2022).

The plane organises around five logical store roles. A working or cache store holds hot-path scratch state. A short-term or session store keeps durable per-thread checkpoints. A long-term or episodic store holds the append-oriented record of what happened. A semantic or knowledge store keeps curated, versioned facts and profiles. A vector or embedding store supports similarity retrieval. The rule that binds them is one of authority: facts, episodes, consent, provenance, and deletion state live in authoritative stores that can be versioned, audited, backed up, and reconciled, while the vector index is a derived projection, rebuildable from those stores and never itself the source of truth. These are logical roles rather than a mandated count of databases. A cost-conscious deployment might collapse several roles into one relational database with a vector extension; a high-scale deployment might split them across a globally consistent transactional store, a hot cache, a dedicated vector service, and an object archive. Agent frameworks already support this separation of memory semantics from storage mechanism (Chhikara et al., 2025; LangChain, 2026; LlamaIndex, 2026; AutoGen, 2026).

1.3 Why "managed cloud" changes the problem

The qualifier managed cloud is doing real work in this article's scope. By a managed cloud agent, we mean one whose compute and memory plane run inside cloud-managed networking and storage boundaries, typically with private connectivity, cloud identity and access management, managed secrets, managed observability, and managed data services. This environment changes the shape of the problem in two ways. First, memory becomes a governed data subsystem in which every read, write, and deletion is subject to tenant isolation, access policy, encryption, and audit, enforced through managed-cloud primitives rather than left to application code. Second, memory is a distributed-systems problem even within a single region, because the plane is polyglot: a cache, a session store, an authoritative fact store, a vector index, and an object archive, each running as a different managed service with different consistency and durability characteristics.

These classes do not need identical guarantees. A session checkpoint needs read-your-writes so the agent does not lose the thread between turns; a deletion needs to propagate from the authoritative record through every derived summary, vector, and cache entry, or a supposedly forgotten fact survives somewhere; a vector index can lag by seconds and simply be rebuilt. Treating memory as one problem with one setting either over-pays for guarantees the vector index does not need, or under-protects the session and deletion paths that do.

1.4 Contributions and structure

This article develops five engineering contributions. First, it separates the memory plane into a brokered control-and-data plane distinct from backend stores and distinguishes read-path hydration from write-path persistence. Second, it defines the five-store logical taxonomy with canonical record models and an authority rule that designates the vector index as derived. Third, it specifies a memory-broker interface, including a tombstone-cascade-verify protocol for verifiable forgetting. Fourth, it frames consistency, durability, latency, and cost as per-memory-class decisions and maps each class to managed services across AWS, Google Cloud, and Azure. Fifth, it offers an operational treatment: engineering service-level objectives, and a low-risk migration path. None of the individual mechanisms drawn on here originates with this article; the contribution is their composition into one governable plane, together with the design rules, separation of authority from acceleration, per-class consistency, and verifiable forgetting, that appear to make the composition sound (Vytla, 2026).

The remainder of the article proceeds as follows. Section 2 reviews related products, frameworks, and research. Section 3 presents the proposed framework: the five-store taxonomy, the brokered reference architecture, and the forget protocol. Section 4 develops the comparative analysis: the per-class trade-offs, the cloud service map, and the competing-architecture comparison. Section 5 discusses limitations, operational considerations, and open problems. Section 6 concludes.

2. Related Work

2.1 Products

The managed-agent products make the multi-layer view of memory visible in production settings. OpenAI separates saved memories, conversation history, and project memory (OpenAI, 2026a). Its more recent Dreaming update is worth dwelling on: a background process now synthesises memory across many conversations and revises it as circumstances change, rewriting "going to Singapore in July" into "went to Singapore" once the trip has passed, which reads as asynchronous consolidation and temporal supersession shipped as a product feature rather than as a research idea (OpenAI, 2026b). Anthropic's Claude separates chat search and memory summaries, project-scoped knowledge with retrieval, and Claude Code's file-backed memory directory (Anthropic, 2026). Google's Gemini separates personalisation from past chats, saved information and instructions, and connected Workspace sources (Google, 2026). AWS Bedrock AgentCore Memory may be the most explicit instance of the read and write asymmetry this article builds on: it exposes short-term memory as raw interaction events scoped by actor and session, and long-term memory as structured insights, semantic facts, summaries, and preferences, extracted asynchronously and retrieved by semantic search across sessions (Amazon Web Services, 2026a). The common thread across these four products is the one this article generalises: transient session state, durable saved facts, and retrieval indices behave as distinct layers rather than a single store.

2.2 Frameworks

Open-source frameworks supply the abstraction boundary between memory semantics and storage mechanism. Mem0 implements a layered capture, promote, and retrieve pipeline with single, batch, and filter-based deletion (Chhikara et al., 2025; Mem0, 2026). LangChain and LangGraph distinguish profile from collection memories, separate hot-path from background writes, and model long-term memory as namespaced documents behind a store and checkpointer abstraction (LangChain, 2026). LlamaIndex separates a short-term buffer from longer-term memory blocks (LlamaIndex, 2026). AutoGen defines a memory protocol of query, add, update-context, and clear operations

(AutoGen, 2026). Zep exposes a dedicated memory application programming interface rather than a bare index, built on a temporally aware knowledge graph (Zep, 2026; Rasmussen et al., 2025). Letta, the productised form of MemGPT's virtual-context idea, exposes editable memory blocks and a searchable archival tier (Letta, 2026; Packer et al., 2023). What these frameworks provide is the interface and the layering; what most leave to the integrator is the governance plane, per-class consistency, tenant isolation enforced at a broker, and verifiable forgetting, which is the gap this article tries to close.

2.3 Research and benchmarks

Several research threads inform the engineering position taken here. Cognitive Architectures for Language Agents supplies the cognitive-architecture account that the plane operationalises, describing language agents as working and long-term memories paired with decision procedures (Sumers et al., 2024). Generative Agents demonstrates memory as accumulated experience: a memory stream of observations, periodic reflection that distils higher-level insights, and relevance-and-recency retrieval into context (Park et al., 2023). Reflexion shows externalised learning without weight updates, in which an agent stores verbal self-critiques in episodic memory and reuses them on a later attempt (Shinn et al., 2023). Voyager contributes procedural memory: a library of reusable, executable skills that grows as an embodied agent succeeds at new tasks (Wang et al., 2023). MemGPT frames memory management as virtual-context paging between a small in-context tier and larger external tiers (Packer et al., 2023), an idea Letta's productisation ships as editable memory blocks and archival memory. Rasmussen et al. (2025) contribute the temporally aware fact model behind Zep, whose bitemporal edges resemble the validity-interval design used for the semantic store in Section 3.

On evaluation, LongMemEval tests information extraction, multi-session reasoning, temporal reasoning, knowledge updates, and abstention (Wu et al., 2025). Its successor shifts toward workflow knowledge, dynamic state tracking, and environment-specific edge cases that may better reflect long-lived, multi-tool agents rather than single-session chat assistants (Wu et al., 2026). Maharana et al. (2024) probe very-long-term conversational memory with the LoCoMo benchmark, and Li et al. (2026) extend this line toward implicit, beyond-factual constraint recall that plain string-matching metrics tend to miss. The correspondence between these category sets and the design concerns this article addresses is fairly close: knowledge updates read as the benchmark form of stale and contradictory facts, abstention reads as the benchmark form of memory-induced hallucination, and temporal reasoning is what the validity-interval model in the semantic store exists to support. A caveat is worth carrying forward from this literature: the evidence base looks considerably stronger for chat-assistant memory, user-profile recall across sessions, than for long-lived, multi-tool, multi-agent production systems, where accumulated environment experience rather than simple preference recall is what appears to matter most. That gap remains one of the open problems discussed in Section 5.

2.4 Competing architectures

The memory plane is not the only way to give an agent memory, and the case for it is comparative rather than purely aesthetic. Six alternative designs are worth naming briefly, because each is a pattern some teams actually ship. Vector-only retrieval-augmented generation chunks, embeds, and retrieves everything by similarity, with the vector index effectively serving as the system of record. Full-context replay consolidates nothing and replays the entire conversation history into the prompt on every turn. Fine-tuning into weights periodically trains past interactions into the model, so memory lives in the parameters rather than in external state. Workflow or session state alone keeps per-task checkpoints for the current run but holds no cross-session long-term memory. Graph-first designs hold all memory in a single temporal knowledge graph, queried by traversal. Event-sourced-only designs keep an append-only log of raw events as the sole store, projecting every view, facts, summaries, vectors, from the log on read. Section 4 compares these six alternatives against the memory plane proposed here across deletion, provenance, versioning, serving latency, cost, auditability, and rebuildability.

3. Proposed Framework: The Memory Plane

3.1 Operating model and assumptions

The framework assumes a specific deployment environment. The agent runtime is stateless in the sense that matters for memory: nothing durable is held in process, so all durable memory lives in external managed stores reached over private network paths. Identity comes from cloud identity and access management, and every call into the memory plane carries a scope, at minimum a tenant and usually a user and project, that the plane uses for isolation and policy. Keys and secrets are managed, data is encrypted in transit and at rest, and observability combines provider-native control-plane audit logging with application-level telemetry emitted by the broker itself. The deployment is single-region and multi-tenant, with isolation enforced in the broker rather than left to each backend query; cross-region replication of memory sits outside this article's scope.

Two clarifications matter here. First, stateless refers to durability rather than to within-session execution: a stateful, session-aware runtime such as AWS Bedrock AgentCore Runtime can keep a dedicated per-session microVM warm across invocations, preserving in-process variables and filesystem state for the session's lifetime, but that state is session-scoped and ephemeral and does not change the authority, consistency, or forgetting rules developed below (Amazon Web Services, 2026a). Second, the five-store decomposition adopted here is best read as a convention rather than a claim of theoretical completeness; it is adopted because current products and frameworks already partition memory this way in practice, across OpenAI, Anthropic, and Google's products and across Mem0, LangChain, LlamaIndex, and AutoGen (OpenAI, 2026a; Anthropic, 2026; Google, 2026; Chhikara et al., 2025; LangChain, 2026; LlamaIndex, 2026; AutoGen, 2026).

A methodological caveat applies to every product cited in this article. Vendor documentation is evidence of product behaviour, the surfaces a user sees and the controls a user is given, not proof of internal architecture. A product that exposes separate saved-memory, chat-history, and project surfaces demonstrates that the vendor finds the distinction worth exposing; it does not reveal how those layers are stored, replicated, or deleted behind the interface. This article therefore uses product behaviour to motivate its design rather than to prove it, and benchmark numbers published by framework vendors are treated as indicative rather than neutral evidence, since they are the authors' own reported results.

3.2 Six design goals

Six goals shape the memory plane, each stated with a criterion a design can be checked against. Separation of authority from acceleration requires that every memory item have exactly one authoritative store, with derived structures, vectors, summaries, caches, treated strictly as projections; the test is whether every derived index could be dropped tomorrow and rebuilt from the authoritative stores with no fact lost. Per-class consistency and durability requires each memory class to declare its own consistency and durability target rather than inherit one platform default; a reviewer should be able to name the consistency level of each store and justify why it is neither stronger nor weaker than the class requires. A policy-aware read path requires that context assembly be mediated by a broker that enforces tenant, user, project, and sensitivity scope before any backend is queried, not as a filter applied afterward.

A selective, reproducible write path requires that writes be explicit and tiered: raw turns and tool outcomes persist synchronously, while facts, summaries, embeddings, and graph updates are produced asynchronously by inspectable jobs whose provenance can be named and replayed. Verifiable forgetting requires that deletion be a control flow that tombstones the authoritative record, cascades to every derived summary, vector, and cache entry, verifies that nothing survived, and returns an audit receipt, treating time-to-live expiry as a lifecycle convenience rather than a substitute for the protocol. Observability and rebuildability requires that every memory read, write, deletion, and reindex be measurable, and that every derived projection be reconstructable deterministically from the canonical stores; the test is whether memory latency, staleness, and deletion compliance are live metrics, and whether a full projection rebuild fits the platform's recovery-time objective.

3.3 The five-store taxonomy

The plane organises memory into five logical store roles, summarised in Table 1. Authority is the property that distinguishes them: three of the five hold truth (session, episodic, semantic), one holds disposable live state (working

or cache), and one holds a derived projection of the others (vector). The working or cache tier is ephemeral, hot-path state, tool outputs, partial plans, locks, and recently retrieved context bundles, written synchronously and read directly by the orchestrator; it should be treated as time-to-live-first and safe to reconstruct, never as a shadow system of record. The short-term or session tier is the durable per-thread checkpoint that lets a conversation resume where it left off, message order, tool state, planner state, and a version token for compare-and-set; it needs read-your-writes within a thread, or the agent loses continuity between turns.

The long-term or episodic tier is the append-oriented record of interactions, actions, observations, and outcomes; order and time matter here more than globally strong reads, because an episode is a fact about a moment that already passed rather than a mutable current value. The semantic or knowledge tier holds curated facts, profiles, and rules whose value lies in being true and current; a fact is modelled as a versioned assertion with a validity interval, so a new assertion closes an old fact's validity window rather than silently overwriting it, which gives the tier first-class contradiction handling. The vector or embedding tier is a derived retrieval structure over chunks, episodes, facts, and documents, optimised for similarity and hybrid search; every vector record carries a back-pointer to its canonical source object, which is what makes deletion a cascade rather than a guess and reindexing deterministic when an embedding model changes.

TABLE 1. The five logical memory stores.

Store	Definition	Canonical record (key fields)	Write path	Read path	Retention / deletion
Working / cache	Ephemeral hot-path scratch state: tool outputs, partial plans, locks, retrieval bundles	cache_key, namespace, value/ref, ttl_ms, lease_owner, created_at	Synchronous, on the hot path	Direct key read by the orchestrator	Seconds to minutes; TTL-first; reconstructable
Short-term / session	Durable per-thread checkpoint: messages, tool and planner state, policy context	tenant_id, agent_id, thread_id, turn_no, messages_ref, tool_state, policy_tags, etag, expires_at	Synchronous checkpoint per turn	Hydrated at turn start	Minutes to days; TTL plus archive or compaction
Long-term / episodic	Append-oriented record of interactions, actions, observations, outcomes	episode_id, scope, kind, summary, raw_ref, observed_at, valid_at, provenance, confidence	Append event, later compact	Recent or similar-episode retrieval	Weeks to years; archive raw, summarise old
Semantic / knowledge	Curated, versioned facts, profiles, and rules, valued for being true and current	fact_id, subject, predicate, object, source_ids, valid_from, valid_to, confidence, supersedes_id, visibility_scope	Selective extraction; explicit upsert	Filtered lookup; profile hydration	Until superseded, expired, or deleted; versioned

Vector / embedding	Derived ANN / hybrid index over the above; for similarity, not authority	vector_id, embedding_model, dims, source_object_id, content_hash, vector, metadata, index_version	Async embed / index after source write	Similarity or hybrid search, then rerank	Rebuildable; delete cascades from canonical tombstone

The rule that binds all five stores is stated as a single authority invariant in four parts. Every memory object has exactly one authoritative representation, held in exactly one store. Every derived record, a vector, a summary, a cache entry, carries a reference back to the authoritative object it was produced from. A deletion is applied first to the authoritative object and only then cascades to derived records; it never begins at a derived copy. Every derived layer can be reconstructed from the authoritative stores, and nothing of value is lost if it is. Concretely, a user's stated preference is authoritative in the semantic store, the episode that produced it is authoritative in the episodic store, the embedding that makes it searchable is a derived copy in the vector index, and any cached hydration of it is disposable. Designating the wrong store as authoritative, most commonly treating the vector index as the durable home of a fact, appears to be what makes deletion, versioning, and audit impossible later.

3.4 The brokered reference architecture

A single broker fronts the memory plane's data plane of stores; the agent's control plane, gateway, orchestrator, planner, policy engine, model router, and retriever, calls instead of reaching the stores directly, as summarised in Table 2. The broker is deliberately the only synchronous entry point from the agent runtime; background consolidation and indexing workers are internal memory-plane components driven by a change stream. This centralisation is not merely a stylistic preference. Four of the six design goals are properties that can only be guaranteed at a single chokepoint: tenant and sensitivity scope must be applied before any backend query or some call site will eventually forget to apply it; authority must be enforced by routing each write to its one canonical store; forgetting must fan a tombstone out across every store and verify it; and observability must see every memory operation in order to measure it. Spread across the control-plane components, each of these becomes a convention every engineer must remember individually; concentrated in the broker, they become invariants.

TABLE 2. Memory-plane reference architecture (components and connections).

Component	Role	Connects to
Agent control plane (gateway, orchestrator, planner, policy engine, model router)	Client-facing reasoning components; issues memory calls but holds no store credentials	Memory broker (the single synchronous entry point)
Memory broker	Policy gate, request router, and audit point for every memory operation	Working cache, session store, episodic store, semantic store, vector index, security and operations layer
Working cache	Ephemeral hot-path scratch state	Broker (synchronous read and write)
Session store	Durable per-thread checkpoints	Broker (synchronous read and write)
Episodic store	Append-oriented event and outcome log	Broker (synchronous write); async projection pipeline (read)
Semantic store	Curated, versioned facts and profiles	Async projection pipeline (write); broker (read)

Vector / hybrid index	Derived retrieval structure over episodes, facts, and documents	Async projection pipeline (write); broker (read)
Async projection pipeline	Change-stream consumers: consolidation and indexing workers	Episodic and session change stream as input; semantic store and vector index as output
Object archive	Backups, raw events, export artefacts, audit receipts	All stores, for recovery and compliance evidence
Security and operations	Identity and access management, tenant isolation, telemetry	Every broker operation, as a cross-cutting control

The read path is policy-aware context assembly under a token budget rather than an instruction to load memory. A policy engine classifies the incoming request by tenant, user, project, and sensitivity, then the broker assembles context in layers: it restores the session checkpoint, fetches explicit semantic facts in scope, retrieves recent and relevant episodes, runs hybrid or vector search filtered by the same policy tags, and reranks the merged candidates into a single bundle before the prompt is hydrated. This just-in-time assembly is close to how current systems already retrieve only what is relevant rather than loading an entire history into context (Anthropic, 2026; Chhikara et al., 2025; LangChain, 2026). Two constraints turn this into an engineering problem rather than a database query. A token budget means retrieved memory competes with reasoning for the context window, so the broker ranks and truncates rather than returning everything it finds. Provenance means every item in the bundle carries its source object and confidence, so the prompt can distinguish authoritative from merely similar content, letting the agent abstain when confidence is low rather than treat a retrieved string as settled fact, which functions as one guard against memory-induced hallucination.

The write path follows one rule: write raw turns and tool outcomes synchronously to the session and episodic stores only, and produce facts, summaries, embeddings, and graph updates asynchronously. The synchronous half is cheap and authoritative; everything requiring interpretation, deciding which durable facts a conversation established, summarising a thread, embedding new content, happens off the hot path, driven by change-data-capture streams such as DynamoDB Streams, Spanner change streams, or the Cosmos DB change feed (Amazon Web Services, 2026c; Google Cloud, 2026b; Microsoft Azure, 2026a). A consolidation worker reads new episodes, extracts candidate facts with their provenance, and upserts them into the semantic store under compare-and-set with supersession, closing a contradicted fact's validity interval rather than overwriting it. What gets promoted is governed by a write policy scoring salience, source trust, task outcome, and deduplication against existing facts, deliberately conservative because a missed fact can be recovered from a retained episode while a wrongly asserted one has already polluted the prompt. Because the derived layers are produced by replaying the change stream, they can be re-derived deterministically from canonical state, which serves consolidation, disaster recovery, and migration backfill with the same underlying machinery.

3.5 The broker interface and the forget contract

The broker exposes a small, fixed set of operations that the control plane programs against: hydrateContext and searchSimilar for reads, appendTurn and upsertFacts for writes, consolidate for background maintenance, and forget and exportMemory for governance. hydrateContext returns not just text but a bundle carrying per-item provenance, the policy decisions applied, and a session token the write path later uses to checkpoint safely. upsertFacts takes a compare-and-set precondition so a fact update either applies to the version it expected or fails loudly, which is what makes supersession safe under concurrency. searchSimilar returns source object identifiers rather than free-floating chunks, honouring the authority rule and keeping deletion tractable.

Deletion is the operation that must be stricter than an ordinary write, because a single fact tends to exist in more places than the code that wrote it remembers: the canonical record, session residue, derived summaries, vector embeddings, hot caches, and the object archive. A naive delete of the canonical row leaves every derived copy behind, and the agent can still surface the "forgotten" fact through the vector index or a stale summary. Products acknowledge

this indirectly: OpenAI keeps saved memories separate from conversation history, so removing one need not remove the other; Gemini warns that deleting a saved memory may still require deleting the underlying conversation or disconnecting a connected source; and Mem0 exposes single, batch, and filter-based deletion precisely because deletion is rarely a single key (OpenAI, 2026a; Google, 2026; Mem0, 2026).

The broker therefore implements forget as a five-step tombstone-cascade-verify protocol, summarised in Table 3. First, the canonical semantic and episodic records in scope are tombstoned, recording the delete intent durably before anything else happens. Second, working-cache entries and session checkpoints in scope are expired, so no live state re-materialises the fact mid-conversation. Third, derived copies are removed: the vectors that reference the tombstoned objects, located through the source-object back-pointer, and the summaries that incorporate them. Fourth, the index pipeline is drained, ensuring the change stream has processed the tombstone so that no in-flight reindex re-creates a vector for an object being deleted. Fifth, the plane verifies by reconciliation: it scans every store for surviving copies in scope, confirms none remain within the deletion service-level objective, and returns a receipt recorded as an immutable audit event — what was deleted, from which stores, on whose authority, and when.

TABLE 3. The tombstone-cascade-verify forgetting protocol.

Step	Action	Purpose
1. Tombstone	Mark in-scope semantic facts and episodic events as deleted in their authoritative stores, recording the delete intent durably	Gives the cascade a durable driver and preserves an audit trail of what is being erased before anything else happens
2. Expire session residue	Drop or expire working-cache entries and session checkpoints in scope	Prevents live state from re-materialising the fact mid-conversation
3. Delete derived copies	Remove vectors that reference the tombstoned objects, located through the source-object back-pointer, and the summaries that incorporate them	Removes the copies that a naive canonical-row delete would otherwise leave behind
4. Drain the index pipeline	Confirm the change stream has processed the tombstone	Stops an in-flight reindex from recreating a vector for an object that is being deleted
5. Verify by reconciliation	Scan every store for surviving copies in scope, confirm none remain within the deletion service-level objective, and return a receipt	Turns forgetting into evidence: what was deleted, from which stores, and when

Time-to-live mechanisms such as DynamoDB time-to-live, Spanner row-deletion policies, or Cosmos DB time-to-live are lifecycle conveniences that expire ephemeral rows on a schedule; they do not cascade to derived vectors and summaries and do not produce the verification evidence an erasure request demands, so they can support the working and session tiers but should not substitute for the forget protocol above.

4. Design Trade-offs and Comparative Analysis

4.1 Authority before mechanism

The headline question in a memory plane is not which vector database performs best, but which store is the source of truth for each memory class. Only once authority is assigned does the rest of the stack, caches in front, indices alongside, graph overlays on top, become a set of separable choices rather than one overloaded decision. The managed-service ecosystem rewards this separation: managed caches optimise for latency, managed transactional databases optimise for durability and consistency, and managed vector services optimise for retrieval, so asking one

of them to do all three jobs tends to produce the two classic mistakes this section returns to: making a globally strong database the home of every memory class, which over-pays, or making the vector index the only durable copy of a fact, which under-protects. The authority assignment follows directly from the taxonomy in Section 3: the session store is authoritative for live thread state, the episodic store for what happened, and the semantic store for what is currently true, while the working cache and the vector index are never authoritative.

4.2 Consistency chosen per class

Consistency is selected per memory class rather than set once for the platform. Session state needs read-your-writes within a thread, or the agent loses continuity when a turn's checkpoint is not visible to the next turn. Vector indices and summaries can usually tolerate eventual consistency, since they are derived and rebuildable, and a few seconds of staleness costs relevance rather than correctness; paying for strong consistency here looks like waste. Policy-critical facts may need stronger semantics, since acting on a stale entitlement or consent flag is a security failure rather than a relevance miss. The cloud platforms expose these choices explicitly. Azure Cosmos DB offers five consistency levels with a documented cost: strong and bounded-staleness levels use more replicas and can measurably reduce read throughput relative to weaker levels, with session consistency serving as the common default for the read-your-writes case (Microsoft Azure, 2026a). DynamoDB distinguishes eventually consistent from strongly consistent reads on a per-request basis (Amazon Web Services, 2026c). Spanner provides external consistency for the specific facts that warrant it (Google Cloud, 2026b). The Redis-family caches that serve the working tier rely on asynchronous replication, so a failover can lose the most recent writes, which is one reason the cache should never hold authority (Amazon Web Services, 2026d).

4.3 A cloud service map and a default stack

The reasoning by tier is fairly direct. The working tier wants a managed Redis-family cache for low latency and time-to-live behaviour: ElastiCache, Memorystore, or Azure Cache for Redis depending on platform (Amazon Web Services, 2026d; Google Cloud, 2026c; Microsoft Azure, 2026c). The session tier wants fast keyed access with expiry and per-thread read-your-writes, served by DynamoDB fronted by a cache, AlloyDB or Spanner where stronger semantics help, or Cosmos DB with session consistency (Amazon Web Services, 2026c; Google Cloud, 2026b; Microsoft Azure, 2026a). The episodic and semantic tiers are well served by managed PostgreSQL, Aurora, AlloyDB, or Azure Database for PostgreSQL, whose JSONB support, transaction logs, and point-in-time recovery fit versioned facts and append-oriented episodes reasonably well (Amazon Web Services, 2026e; Google Cloud, 2026b; Microsoft Azure, 2026b). The vector tier is where a dedicated service tends to earn its cost at scale: OpenSearch, Google Cloud Vector Search, and Azure AI Search provide managed approximate-nearest-neighbour and hybrid retrieval, while pgvector co-located in the same PostgreSQL instance wins when update simplicity and single-database governance matter more than corpus scale (Amazon Web Services, 2026e; Google Cloud, 2026c; Microsoft Azure, 2026c; pgvector, 2026). A cross-cutting object-storage archive, S3, Cloud Storage, or Azure Blob Storage, each with versioning and immutability controls, holds what the operational stores should not: raw events, export artefacts, and compliance evidence (Amazon Web Services, 2026e; Google Cloud, 2026c; Microsoft Azure, 2026c).

A single vendor-neutral recommendation follows from this reasoning: start with PostgreSQL, Redis, a managed vector service, and object storage. PostgreSQL holds the session, episodic, and semantic roles in collapsed form; Redis serves the working cache; a managed vector service, or pgvector in the same database, provides the index; object storage is the archive. This appears to be the smallest stack that still honours the authority rule, and it should scale a long way before anything else is needed. Two failure modes bracket the design space. The over-design mistake applies uniform strong or serialisable consistency to every class, buying write-latency and read-throughput cost for guarantees that the working, vector, and most semantic facts do not need. The under-design mistake makes the vector index the only durable copy of a fact, which is ungovernable, undeletable, and unversionable. The right number of databases is probably the smallest one that lets each class obtain the authority, consistency, and durability it needs, and no more (Gaggar, 2025).

4.4 Operational targets

A memory plane should probably be operated with the same discipline given to any store of customer data: encryption in transit and at rest, private connectivity, tenant-isolated namespaces, explicit retention classes, and point-in-time recovery or an immutable archive, paired with provider-native audit logs and application-level telemetry the broker emits for every read, write, deletion, and reindex (Amazon Web Services, 2026b; Google Cloud, 2026a; Microsoft Azure, 2026b). Measured against this baseline, a small set of engineering targets can make memory failures detectable rather than latent: hydration latency in the low hundreds of milliseconds at the ninety-fifth percentile, retrieved-memory tokens held under roughly fifteen percent of the prompt budget, unauthorised-retrieval rate held at zero under red-team and shadow testing, and deletion-compliance latency bounded and proven with a receipt rather than assumed. These are starting points to calibrate per product rather than universal constants, and two of them, unauthorised retrieval and deletion compliance, function as hard objectives rather than tunable ones, since a cross-tenant leak or an incomplete erasure is a compliance failure regardless of how good retrieval otherwise performs.

4.5 Comparison against competing architectures

The case for the memory plane is comparative rather than absolute. Table 4 measures it against six alternative designs, vector-only retrieval-augmented generation, full-context replay, fine-tuning into weights, workflow or session state alone, a graph-first design, and an event-sourced-only design, across deletion, provenance, versioning, serving latency, cost, auditability, and rebuildability.

TABLE 4. Competing memory architectures (qualitative ratings; serving latency and cost as low, medium, or high).

Architecture	**Deletion**	**Provenance**	**Versioning**	**Serving latency**	**Cost**	**Auditability**	**Rebuildable**
Vector-only (RAG over a vector database)	Weak	Weak	None	Low	Low-medium	Weak	n/a (it is the store)
Full-context replay	Easy but uncurated	Raw only	None	High (prompt grows)	High (tokens)	Raw only	n/a
Fine-tuning into weights	None	None	None	Low	High (retrain)	None	n/a
Workflow / session state only	Easy (TTL)	Partial	None	Low	Low	Partial	n/a
Graph-first (temporal graph as sole store)	Strong	Strong	Strong	Medium	Medium-high	Strong	Partial
Event-sourced only (raw log, project on read)	Hard (immutable log)	Strong	Strong	Poor (projected on read)	Medium	Strong	Strong
Memory plane (this article)	Strong	Strong	Strong	Low	Medium	Strong	Strong

Vector-only retrieval and fine-tuning fail the governance dimensions outright: neither can delete, version, or audit a specific fact, which is why the plane treats the vector index as derived and never bakes memory into model weights. Full-context replay and workflow-state-only sidestep governance debt by not accumulating durable memory at all, at the cost of unbounded prompt growth or no cross-session experience. The two comparatively strong alternatives are graph-first and event-sourced designs, and the memory plane borrows from both, temporal validity from the graph branch and a replayable change-stream spine from the event-sourced branch, without adopting either wholesale: a temporal graph is more machinery than most versioned-value facts require as a sole store, and a pure immutable log makes the one operation that must be cheap and verifiable, deletion, the hardest to implement well. The plane appears to be the only row strong across deletion, provenance, versioning, auditability, and rebuildability while keeping serving latency low; its clearest concession is cost, since a polyglot stack has more moving parts than a single store, a cost the collapsed default stack in Section 4.3 bounds until scale forces the roles apart (Vytla, 2024).

5. Discussion

5.1 A low-risk migration path

Few teams build a memory plane greenfield; most already have an agent that stuffs history into the prompt or treats a single vector store as memory. A four-stage migration can limit exposure at each step. Shadow read stands up the plane to assemble context in parallel with the existing strategy without feeding its output into the live prompt, using the shadow to measure relevance and hydration latency against real traffic. Dual write begins writing new conversations into the canonical session and episodic stores alongside the existing path, so the authoritative log accumulates without disrupting serving. Async backfill populates facts, summaries, and vectors from change-data-capture and historical exports, re-deriving projections exactly as steady-state consolidation would. Guarded cutover lets the broker begin hydrating live prompts only once relevance, latency, and the deletion workflow meet the operational targets, behind feature flags reversible per tenant. Service-native change streams give the backfill an ordered, replayable source, and point-in-time recovery on the authoritative stores gives every stage a rollback, so the migration functions, in effect, as a rebuild drill run forward rather than for recovery (Vytla, 2026).

5.2 Limitations and open problems

Several problems remain genuinely open, and none of them is a reason to delay building a memory plane; they seem more like reasons to build one that is observable and rebuildable enough to absorb future improvement rather than requiring another rebuild. Autonomous consolidation quality still looks like more craft than science: the write path defers fact extraction and summarisation to background workers, but how well those workers decide what to promote, precision without losing important facts, is tuned empirically rather than derived from a settled theory. Temporal contradiction repair at scale is unsolved beyond the single-entity case: the validity-interval model handles supersession in principle, but doing so reliably across millions of evolving facts, with conflicts detected and resolved automatically, remains an open engineering problem (Vytla, 2025).

Cross-agent shared memory with safe access control was scoped out of this article deliberately; sharing memory across agents while preserving tenant isolation and provenance looks like a governance problem as much as an engineering one (Gaggar, 2025). Evaluation of environment-derived experience lags the systems it is meant to measure: the evidence base is considerably stronger for chat-assistant preference recall than for the accumulated environment experience that long-lived, multi-tool agents actually depend on, even as newer benchmarks such as LongMemEval-V2 and LoCoMo-Plus begin to narrow the gap (Wu et al., 2026; Li et al., 2026). Safe learned and procedural memory raises a related question this article's write policy does not fully answer: as agents promote not only facts but reflections and reusable skills, deciding which learned behaviours are safe to keep, reuse, or retire is only lightly addressed by salience and trust scoring alone. Trust and explainability round out the list: memory that surfaces something a user did not expect can erode trust even when retrieval is accurate, and giving users a legible account of what is remembered and why a particular memory was used remains unsolved at product scale beyond the provenance the plane already records (Ankam, 2022).

5.3 A decision checklist

The soundness of a memory design can be checked fairly quickly against a short set of contrapositives, each tracing back to one of the goals in Section 3. If an engineer cannot explain which store is authoritative for a given fact, the design is not yet sound (Ankam, 2024). If deleting a memory does not also delete its derived summaries and vectors, the design is not yet compliant. If the vector index is the only durable copy of a fact, the design is not yet governable. If every memory class uses the same consistency level, the design is probably paying too much or risking too much. If the system cannot distinguish session state from long-term facts, it will likely forget too much or remember too much. If memory writes are automatic but not observable, an operator cannot tell whether the system is learning or being poisoned. If memory projections cannot be rebuilt from canonical stores, disaster recovery and audit will probably be fragile exactly when they are needed most.

6. Conclusion

This article has argued one position throughout: agent memory in production is neither a single database nor retrieval-augmented generation by itself, but a memory plane, a brokered, policy-aware subsystem that hydrates context on the read path and persists selectively on the write path. Four claims follow from that framing. Memory reads as a brokered plane rather than a vector database. Authority should be separated from acceleration, with facts, episodes, consent, and provenance living in versioned authoritative stores while the vector index remains a derived, rebuildable projection. Consistency is chosen per memory class rather than set once for the platform. Forgetting is a control flow, tombstone, cascade, verify, rather than a single delete statement. The five-store taxonomy, the brokered architecture, the interface and forget contracts, the per-class trade-offs, and the operational treatment developed across this article follow from these four claims rather than standing as independent design choices.

Underneath all of them sits a single shift in stance: memory is not storage to be queried but experience to be managed, captured, consolidated, corrected, governed, and forgotten over the life of an agent, and the memory plane is offered here as the engineering substrate that makes that stance operational rather than aspirational. None of the mechanisms surveyed, the products, the frameworks, or the research threads, originates with this article; the contribution offered is their composition into a single governable plane and the explicit articulation of the design rules that make the composition sound rather than incidental. As managed-agent products converge further on layered memory, and as evaluation benchmarks extend from chat-assistant recall toward the environment experience that long-lived agents actually accumulate, the design pressure on production teams will likely shift from whether to build a memory plane toward how rigorously to specify one. Teams that treat memory as a regulated data subsystem from the outset, with explicit authority, per-class consistency, and verifiable forgetting, should find that rigour pays for itself well before regulatory or incident pressure forces the question. The six problems left open in Section 5 mark where the next engineering effort belongs, not where this account falls short of its own goals.

REFERENCES

1. Amazon Web Services. (2026a). Add memory to your Amazon Bedrock AgentCore agent: Memory types and long-term memory. AWS Documentation. https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/memory.html
2. Amazon Web Services. (2026b). Amazon Virtual Private Cloud user guide; AWS CloudTrail user guide. AWS Documentation. https://docs.aws.amazon.com/vpc/latest/userguide/
3. Amazon Web Services. (2026c). Amazon DynamoDB developer guide: Streams, time to live, and read consistency. AWS Documentation. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
4. Amazon Web Services. (2026d). Amazon ElastiCache documentation: Replication and high availability for Redis and Valkey. AWS Documentation. https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/Replication.html
5. Amazon Web Services. (2026e). Amazon Aurora user guide: Backup and restore; Amazon S3 user guide: Object Lock; Amazon OpenSearch Service developer guide: Vector and k-NN search. AWS Documentation. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Managing.Backups.html
6. Anthropic. (2026). Use Claude's chat search and memory to build on previous context. Claude Help Center. https://support.claude.com/en/articles/11817273-use-claude-s-chat-search-and-memory-to-build-on-previous-context
7. AutoGen. (2026). Memory and RAG. AutoGen AgentChat Documentation, Microsoft. https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/memory.html

8. Chhikara, P., Khant, D., Aryan, S., Singh, T., & Yadav, D. (2025). Mem0: Building production-ready AI agents with scalable long-term memory. arXiv. https://doi.org/10.48550/arXiv.2504.19413
9. Google Cloud. (2026a). Private Service Connect; Cloud Audit Logs overview. Google Cloud Documentation. https://cloud.google.com/vpc/docs/private-service-connect
10. Google Cloud. (2026b). Spanner documentation: TrueTime, external consistency, and change streams; AlloyDB for PostgreSQL overview. Google Cloud Documentation. https://cloud.google.com/spanner/docs/true-time-external-consistency
11. Google Cloud. (2026c). Memorystore for Redis documentation; Cloud Storage object versioning and bucket lock; Vector Search documentation. Google Cloud Documentation. https://cloud.google.com/memorystore/docs/redis
12. Google. (2026). Get personalization in Gemini Apps. Gemini Apps Help. https://support.google.com/gemini/answer/16598623
13. H. Gaggar, (2025). "Policy-Driven Access Control for Secure Deployment of Autonomous AI Agents in Enterprise Environments." International Journal of Computational and Experimental Science and Engineering, 11(4). [Online]. Available: https://doi.org/10.22399/ijcesen.5497
14. J. Ankam, (2024). "Contracts Across Modalities: Detecting Embedding Staleness and Governance Drift in Multimodal Lakehouse Architectures." International Journal of Computational and Experimental Science and Engineering, 10(1). [Online]. Available: https://doi.org/10.22399/ijcesen.5469
15. J. Ankam, (2022). "Benchmarking Autonomy: A Taxonomy of Human-in-the-Loop Checkpoints for AI-Generated Transformation Code." International Journal of Computational and Experimental Science and Engineering, 8(3). [Online]. Available: https://doi.org/10.22399/ijcesen.5462
16. LangChain. (2026). Memory overview and persistence. LangChain / LangGraph Documentation. https://docs.langchain.com/oss/python/concepts/memory
17. Letta. (2026). Memory blocks (core memory) and archival memory. Letta Docs. https://docs.letta.com/guides/core-concepts/memory/memory-blocks
18. Li, Y., Guo, W., Zhang, L., Xu, R., Huang, M., Liu, H., Xu, L., Xu, Y., & Liu, J. (2026). LoCoMo-Plus: Beyond-factual cognitive memory evaluation framework for LLM agents. arXiv. https://doi.org/10.48550/arXiv.2602.10715
19. LlamaIndex. (2026). Memory in LlamaIndex. LlamaIndex Developer Documentation. https://developers.llamaindex.ai/python/framework/module_guides/deploying/agents/memory/
20. Maharana, A., Lee, D.-H., Tulyakov, S., Bansal, M., Barbieri, F., & Fang, Y. (2024). Evaluating very long-term conversational memory of LLM agents. arXiv. https://doi.org/10.48550/arXiv.2402.17753
21. Mem0. (2026). Memory types, add memory, search memory, and delete memory. Mem0 Documentation. https://docs.mem0.ai/core-concepts/memory-types
22. Microsoft Azure. (2026a). Consistency levels in Azure Cosmos DB; working with the change feed. Microsoft Learn. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels
23. Microsoft Azure. (2026b). Azure Private Link and private endpoints; Azure Monitor overview; Azure Database for PostgreSQL backup and restore. Microsoft Learn. https://learn.microsoft.com/en-us/azure/private-link/
24. Microsoft Azure. (2026c). Azure Cache for Redis documentation; Blob versioning and immutable storage; vector search in Azure AI Search. Microsoft Learn. https://learn.microsoft.com/en-us/azure/azure-cache-for-redis/
25. OpenAI. (2026a). Memory FAQ. OpenAI Help Center. https://help.openai.com/en/articles/8590148-memory-faq
26. OpenAI. (2026b). Dreaming: Better memory for a more helpful ChatGPT. https://openai.com/index/chatgpt-memory-dreaming/
27. Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & Gonzalez, J. E. (2023). MemGPT: Towards LLMs as operating systems. arXiv. https://doi.org/10.48550/arXiv.2310.08560
28. Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. arXiv. https://doi.org/10.48550/arXiv.2304.03442
29. pgvector. (2026). pgvector: Open-source vector similarity search for Postgres [Computer software]. GitHub. https://github.com/pgvector/pgvector
30. Rasmussen, P., Paliychuk, P., Beauvais, T., Ryan, J., & Chalef, D. (2025). Zep: A temporal knowledge graph architecture for agent memory. arXiv. https://doi.org/10.48550/arXiv.2501.13956
31. S. K. Vytla, (2025). "DEVOPS-GRID: DevOps-driven reliability and digital-twin operations for intelligent power systems." Gongcheng Kexue Yu Jishu/Advanced Engineering Science, 57(3).
32. S. K. Vytla, (2026). "GOVERN-CTRL: A Governance Control Plane Architecture for Production-Scale Data and Machine Learning Pipelines." International Journal of Computational and Experimental Science and Engineering, 12(3). [Online]. Available: https://doi.org/10.22399/ijcesen.5394
33. S. K. Vytla, (2024). "POLICY-ML: Policy-as-Code Enforcement for Compliance, Auditability, and Trust Across Data and Machine Learning Pipelines." International Journal of Computational and Experimental Science and Engineering, 10(4). [Online]. Available: https://doi.org/10.22399/ijcesen.5393
34. Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. arXiv. https://doi.org/10.48550/arXiv.2303.11366
35. Sumers, T. R., Yao, S., Narasimhan, K., & Griffiths, T. L. (2024). Cognitive architectures for language agents. Transactions on Machine Learning Research. https://doi.org/10.48550/arXiv.2309.02427

36. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). Voyager: An open-ended embodied agent with large language models. arXiv. https://doi.org/10.48550/arXiv.2305.16291
37. Wu, D., Wang, H., Yu, W., Zhang, Y., Chang, K.-W., & Yu, D. (2025). LongMemEval: Benchmarking chat assistants on long-term interactive memory. Proceedings of the International Conference on Learning Representations (ICLR 2025). https://doi.org/10.48550/arXiv.2410.10813
38. Wu, D., Wang, H., Yu, W., Zhang, Y., Chang, K.-W., & Yu, D. (2026). LongMemEval-V2: Evaluating long-term agent memory toward experienced colleagues. arXiv. https://doi.org/10.48550/arXiv.2605.12493
39. Zep. (2026). Memory and sessions. Zep Documentation. https://help.getzep.com/v2/memory