

Layered Agent Memory, State, and Project Knowledge Architecture

Executive Summary

The rapid evolution of artificial intelligence agents has exposed a critical architectural vulnerability: the over-reliance on context windows as a surrogate for genuine memory. While modern language models offer expansive context limits, utilizing them as the sole repository for agent history introduces severe operational degradation, escalating costs, latency, and the "lost-in-the-middle" effect.¹ This report investigates the imperative shift toward layered memory architectures. Reliable agent memory must be implemented as an externally managed knowledge and state architecture, separating temporary execution context from durable project knowledge. The central hypothesis tested throughout this analysis demonstrates that immediate context must remain task-specific; workflow state must be structured; and organizational knowledge must require rigorous validation before promotion. Furthermore, retrieval algorithms must combine permission filtering, semantic relevance, recency, and authority to prevent contradictions. This analysis provides a blueprint for a production-ready, model-agnostic memory ecosystem suitable for advanced developers building scalable agent fleets in 2026.

Key Conclusions

The evidence from production deployments and 2026 benchmarks strongly supports the necessity of a layered architecture over monolithic context appending.¹ First, the context window is definitively not memory. Relying on full-context injection yields a 91% latency penalty and massive token inflation compared to selective external retrieval, while simultaneously risking accuracy degradation due to attention dilution.¹ Second, epistemic separation is mandatory. Systems must explicitly divide factual world knowledge, subjective agent opinions, and verifiable workflow state. Architectures that merge these elements—treating an agent's past hypothesis as a verified fact—produce irreconcilable hallucinations.⁴ Third, bitemporal lineage resolves contradictions. Temporal knowledge graphs tracking both event time and ingestion time definitively resolve factual supersession where simple vector similarity fails.⁶ Finally, durable execution requires dedicated workflow engines. Agent state must be decoupled from semantic memory, utilizing persistent checkpoints to allow precise task resumption following environmental failures.⁸

Terminology and Conceptual Model

A precise vocabulary is fundamental to architecting memory systems. The following table establishes the operational boundary between transient execution, durable knowledge, and system mechanics, addressing the critical definitions required for this domain.

Term	Definition and Architectural Role	Storage and Lifecycle
Context Window	The maximum token limit a model processes in a single invocation. Acts as the CPU register.	Transient. Cleared post-invocation.
Prompt / Transient Context	The specific information injected into a single model call, including instructions and retrieved excerpts.	Ephemeral.
Persistent Memory	Information saved to external datastores that survives agent restarts and environment failures.	Durable. Requires explicit deletion.
Conversation / Session Memory	Context relevant to one ongoing interaction, distinct from long-term user profiles.	Short-term to Durable (if archived).
Task / Workflow Memory	Information required to complete one defined task or durable execution run (e.g., checkpoints).	Durable state, often in relational databases.
Project / Client Memory	Knowledge spanning multiple tasks for a specific initiative or customer. Requires strict isolation.	Durable. Governed by RBAC namespaces.
Organisational / Global Memory	Validated, universally applicable knowledge (e.g., SOPs) reused across projects.	Highly Durable. Requires human approval.
User / Agent-Local	Persistent attributes about a human user, or private	Durable. Subject to

Memory	scratchpad state for a single agent.	privacy/consent rules.
Fleet / Shared Memory	Information accessible to a coordinated group of agents working on a common objective.	Durable. Pub/sub or blackboard patterns.
Working Memory	Active operational context (e.g., current plan). Volatile, updated frequently.	Temporary cache or checkpointed state.
Short-term Memory	Data useful across a brief sequence of steps, such as recent tool outputs.	Ephemeral.
Episodic / Autobiographical Memory	Chronological records of specific past events and agent actions.	Durable event logs or temporal graphs.
Semantic / Declarative Memory	Generalized, factual knowledge abstracted from episodes (e.g., system architecture).	Vector stores or knowledge graphs.
Procedural / Operational Memory	Instructions on how tasks are executed (e.g., coding practices, prompt templates).	Files or structured RDBMS.
Prospective Memory	Triggers for future actions (e.g., reminders).	Durable workflow schedules, not prose.
Working / Execution State	Temporary variables, scratchpad notes, and active plans used during a task.	Volatile.
Durable / Application	Verified progress surviving system crashes	Immutable or Append-only

State	(checkpoints, event logs, event sourcing).	DB.
Artefacts / Files as Memory	Generated outputs or repository files (markdown, JSON) serving as declarative context.	Durable repository storage.
Retrieval-Augmented Generation (RAG)	The mechanism of fetching external data and injecting it into the prompt context.	System pipeline operation.
Embeddings / Vector Stores	High-dimensional numerical representations of text enabling semantic similarity search.	Durable indexing infrastructure.
Lexical / Hybrid Retrieval	Keyword matching (BM25) combined with vector search for exact-phrase recall.	Query mechanism.
Knowledge Graphs / Graph Retrieval	Networks of entities and relationships, supporting multi-hop traversal and temporal edges.	Graph DB (e.g., Neo4j, Graphiti).
Metadata Filtering / Relevance Scoring	Using structured attributes (tenant ID, date) to restrict and rank semantic search results.	Deterministic execution layer.
Recency Weighting / Confidence	Adjusting retrieval scores based on data freshness and inferred reliability.	Scoring algorithm component.
Provenance / Source Authority	The traceable origin of a memory and its hierarchical trustworthiness.	Essential metadata field.

Memory Lineage / Versioning	Tracking transformations of a fact over time, maintaining parent-child version relationships.	Database schema design.
Compaction / Summarisation / Consolidation	Reducing memory volume while preserving utility, often by extracting semantic facts from episodes.	Background system process.
Forgetting / Retention / Expiration	Policies dictating when data must be removed due to irrelevance, age, or compliance.	Automated garbage collection.
Deletion / Tombstoning	Physically removing data versus marking a record as superseded while keeping the audit trail.	Write/Update operations.
Access Control / Namespace / Tenant Isolation	Restricting data visibility to authorized agents/users, preventing cross-client leakage.	Hard infrastructure/DB constraint.
Data Classification / Source of Truth	Categorizing data sensitivity and designating the authoritative system of record.	Governance layer.
Write / Read Authority	Permissions dictating which agent or user can create, update, or retrieve a specific memory.	Security middleware.
Contradiction Detection	Algorithmic identification of new facts that conflict with existing semantic memory.	Write pipeline validation step.

Stale / Poisoned Memory	Outdated facts, or maliciously injected instructions intended to compromise agent behavior.	System vulnerability requiring mitigation.
Memory / Retrieval / Prompt Contamination	Injecting semantically similar but factually irrelevant or cross-tenant data into the context.	Failure mode prevented via metadata.

Context Window versus Persistent Memory

The fundamental distinction in agent architecture lies in recognizing that a model context window is a transient computational workspace, not a storage medium.¹⁰ The context window contains only the information strictly necessary for the immediate inference cycle. When information leaves this window, the model retains no structural awareness of it unless the context is explicitly retrieved and reinjected.¹⁰ While modern providers offer expansive limits scaling to millions of tokens, expanding context windows does not eliminate the need for an external memory architecture.¹

Utilizing massive context windows leads to severe architectural degradation. Context dilution and attention degradation manifest as the "lost-in-the-middle" behavior, where models selectively ignore crucial constraints buried within extensive transcripts.¹ Furthermore, prompt bloat inflates token costs and exacerbates latency, while stale summaries and duplicated information confuse the model with conflicting instructions.¹

A comparison of architectural approaches reveals varying degrees of efficacy. Injecting the *entire conversation transcript* in every prompt achieves high raw accuracy for short interactions but suffers catastrophic latency penalties (e.g., 17 seconds p95) and unsustainable costs, making it inappropriate for long-running agents.¹ A *recent-message sliding window* manages token limits but causes the agent to abruptly forget critical instructions established early in the dialogue. A *rolling conversation summary* preserves a compressed timeline but suffers from summary distortion, where nuance is lost and model-generated errors are permanently reinforced.¹² *Retrieval of selected historical messages* introduces semantic recall but lacks deterministic state tracking. *Event-log reconstruction* is excellent for auditing and workflow replay but fails to provide abstracted semantic reasoning.⁹ The most robust approach for production is *hybrid retrieval using files, metadata, embeddings, and structured state*. This decouples deterministic workflow progression from probabilistic text generation, ensuring the model only receives highly relevant, filtered context.¹³

Normally, a well-engineered prompt context must be strictly limited to the task objective, immediate instructions, relevant constraints, current workflow state, selected project facts,

required artefacts, recent actions, relevant source excerpts, permissions, and the output schema. Conversely, unrelated client data, unverified hypotheses, expansive historical transcripts, and raw chain-of-thought from previous tasks must remain outside the model context to prevent contamination and latency growth.

Memory Taxonomy by Scope

Memory persistence must be scoped to its operational utility to prevent cross-contamination.

4.1 Model-call Memory

Information required solely for one inference cycle includes the immediate prompt, raw tool results, selected source fragments, requested output formats, and temporary reasoning scratchpads. This data must be immediately discarded or relegated to cold audit logs after the call to prevent token accumulation and context bloat.

4.2 Conversation Memory

Context specific to an ongoing user interaction includes recent discussion threads, unresolved questions, temporary conversational terminology (e.g., referring to "the current dashboard"), and user corrections made during the session. This differs from long-term user memory, which abstracts permanent preferences (e.g., "always output in Python") from transient conversational state.¹⁴

4.3 Task Memory

Information required to complete one defined task includes requirements, intermediate calculations, temporary hypotheses, dependencies, current progress markers, and validation results. Upon task completion, task memory should be flushed from active context; the finalized outputs and generated artefacts are promoted to workflow or project memory, while the intermediate scratchpad notes are discarded or archived.¹⁵

4.4 Workflow-run Memory

Associated with durable execution, this includes workflow inputs, completed steps, retries, checkpoints, approvals, errors, and state transitions. Workflow state must be written to a deterministic database (e.g., PostgreSQL via LangGraph Saver or Temporal) rather than relying solely on agent-generated prose, because structured engines ensure execution resumes exactly at the point of failure without duplicating side effects.⁸

4.5 Project Memory

Knowledge remaining useful throughout an initiative encompasses project goals, architecture decisions, design standards, terminology, constraints, approved specifications, and known issues. Project memory must be tightly organized, version-controlled, and strictly namespaced to the specific repository or initiative to prevent cross-project contamination.¹⁷

4.6 Client Memory

Highly sensitive knowledge specific to a customer or commercial engagement includes commercial requirements, brand guidelines, contractual constraints, private files, and approval history. This information must be isolated from unrelated clients through strict logical or physical separation (e.g., Row-Level Security, separate vector collections) to prevent catastrophic data leakage.

4.7 Organisational Memory

Validated, universally applicable knowledge reused across multiple projects includes standard operating procedures, coding standards, reusable templates, and compliance guidance. Promotion into organizational memory requires rigorous governance—typically human approval or strict consensus algorithms—to prevent an isolated project anomaly from becoming a flawed organizational standard.

4.8 User Memory

Persistent attributes regarding a specific human interactor require distinguishing between explicitly saved preferences (e.g., API keys), inferred preferences (e.g., communication style), and sensitive confidential information. User memory mandates strict adherence to consent, transparency, data minimization, and deletion rights, ensuring that sensitive data is not indiscriminately embedded into vector stores accessible to unauthorized agents.

Memory Taxonomy by Cognitive Function

Mirroring human cognition provides a functional blueprint for agent systems.¹⁵

5.1 Working Memory

Working memory represents the active operational context, including the current objective, active plan, next action, currently opened files, and immediate blockers.¹⁵ Its size is strictly limited by the context window budget. It is highly volatile and updated rapidly. Working memory should only be checkpointed to durable state when the agent transitions between major task phases or requires human-in-the-loop approval.

5.2 Short-term Memory

Useful across a brief sequence of interactions, short-term memory encompasses recent tool outputs and short-lived assumptions. Retention policies should expire this memory once the immediate objective resolves or a session terminates, preventing transient assumptions from calcifying into permanent facts.

5.3 Episodic Memory

Chronological records of events answer the question, "What happened?" This includes what occurred during a run, which decisions were made, what failed, and which artefacts were

produced. Architectures like Hindsight map this to the "Experience Network".¹⁹ Episodic memory directly relates to event logs, decision journals, and run histories, often relying heavily on bitemporal timestamps.⁶

5.4 Semantic Memory

Semantic memory represents generalized, factual knowledge abstracted from episodes.⁶ It encompasses product facts, system architectures, and standard policies. Semantic memory must be actively sourced from verified episodic data, versioned, and explicitly contradicted or retired when outdated facts emerge, rather than simply appending new vectors.

5.5 Procedural Memory

Procedural memory encapsulates the "how-to" knowledge regarding work execution, including workflows, SOPs, coding practices, and tool-use instructions. This knowledge is most safely and effectively stored as executable code, structured workflow definitions, or file-based templates (e.g., CLAUDE.md) rather than probabilistic prompts, ensuring deterministic execution.¹⁷

5.6 Prospective Memory

Information regarding actions that must occur in the future includes reminders, pending approvals, scheduled checks, and dependency-triggered actions. Prospective memory must never rely on hoping a language model "remembers" via prose; it must be implemented through durable schedules, event queues, or workflow state timers (e.g., Temporal sleep functions).⁹

Working State versus Durable State

Establishing a precise boundary between transient working state and durable system state is critical for system reliability. Working state includes current scratch notes, temporary planning, unverified hypotheses, unfinished drafts, and local execution details. This data is volatile and safely recomputable. Durable state encompasses task status, approved decisions, final specifications, checkpoints, committed code, and audit events.⁹

State transitions involving non-idempotent side effects (e.g., database writes, sending emails, payments) must be written durably. A checkpoint must be created immediately before and after such actions. While temporary caches and intermediate drafts may be recomputed, checkpoints, approvals, and verified facts must survive process crashes and must remain immutable or utilize append-only histories. When concurrent agents modify shared state, optimistic concurrency control (e.g., compare-and-swap versioning) is required to resolve conflicts.² Partially completed state should be represented as explicitly "Draft" or "Proposed," distinguishing it clearly from "Verified," "Approved," and "Superseded" states.

Files as Memory

File-based repositories (e.g., Markdown, JSON, YAML) provide high human readability, version

control via Git, portability, and low infrastructure complexity.¹⁷ Frameworks utilizing conventions like CLAUDE.md or MEMORY.md inject predefined rules directly into the system prompt.²³ Files serve excellently as authoritative systems of record for procedural knowledge, source code, and architecture decision records.

However, files present distinct disadvantages: poor semantic retrieval capabilities without external indexing, inconsistent structures, duplication risks, and concurrency conflicts.²⁵ They are less suited for dynamic, highly relational knowledge.

A clean project-memory directory structure integrates explicit knowledge domains:

- /docs/project_manifest.md (Authoritative routing and project goals)
- /knowledge/decisions/ (Architecture Decision Records)
- /memory/episodes/ (Task journals and historical snapshots)
- /artefacts/ (Final, generated outputs)

To prevent models from treating all files equally, the system must assign authority weights based on the directory path and metadata tags during retrieval, ensuring a formalized ADR outranks a developer's temporary plain-text notes.

Storage Technologies and their Proper Roles

Diverse storage technologies must be combined to form a cohesive memory architecture.

8.1 Relational Databases

PostgreSQL and SQLite are the backbone of robust memory governance. They handle structured workflow state, task records, RBAC permissions, metadata filters, versioned facts, and checkpointer persistence impeccably.⁸ They scale predictably but require operational overhead compared to local files. They are essential for both solo builders and multi-agent fleets.

8.2 Document Databases

Document stores (e.g., MongoDB) provide flexible schemas for nested project documents, agent profiles, and conversation objects. However, JSONB columns in PostgreSQL often render dedicated document databases unnecessary for minimal viable architectures.

8.3 Vector Stores

Vector databases (e.g., pgvector, Pinecone) index high-dimensional embeddings for similarity and approximate nearest-neighbor search.¹ While exceptional at semantic recall, treating semantic similarity as factual relevance is a critical risk. Vector stores produce misleading results when handling negations (e.g., "User does *not* use AWS" vs "User uses AWS") or temporal supersession.²⁸

8.4 Full-text and Lexical Search

Lexical search utilizes inverted indexes and keyword matching (e.g., BM25) for exact phrase retrieval. It excels where vector retrieval fails—specifically with exact entity matching, distinct

error codes, and unique identifiers, bypassing language-specific embedding drift.

8.5 Hybrid Search

The industry standard for production systems. Hybrid search combines semantic similarity with lexical matching, aggressively filtered by RDBMS metadata (recency, permission, project scope) and reranked using Reciprocal Rank Fusion (RRF).²

8.6 Knowledge Graphs

Technologies like Zep (Graphiti) structure memory as entities, relationships, and temporal edges, supporting complex graph traversal.⁶ They are justified when multi-hop relational queries and bitemporal logic are paramount, but they introduce massive unnecessary complexity for simple fact retrieval systems.

8.7 Event Logs and Event Stores

Event stores record chronological activity for causal tracing, auditability, and workflow replay.⁹ An event log describes *what happened*, whereas semantic memory describes *what is currently true*.

8.8 Object and Artefact Stores

S3-compatible storage is required for PDFs, images, datasets, and large generated reports, referenced by URIs in the metadata store, preventing database bloat.

8.9 Key-value Stores and Caches

Redis handles temporary state, locks, leases, short-lived retrieval caches, rate limits, and ephemeral session state, operating strictly outside durable semantic knowledge.

8.10 Combined Architectures

A highly effective combined architecture dictates that PostgreSQL owns structured state, metadata, and lexical search; Git owns source-controlled project knowledge; pgvector handles semantic recall; and S3 owns large artefacts.¹³ This stack represents the minimum set of technologies necessary before introducing complex knowledge graphs.

Memory Records and Canonical Schemas

A standardized memory schema enables system-wide interoperability and rigorous governance.

Field	Example Value	Description
memory_id	mem_9a8b7c	Unique identifier.
tenant_id	org_123	Top-level isolation

		boundary.
project_id	proj_456	Project scope boundary.
memory_type	decision	Categorization (e.g., fact, decision, preference).
scope	project	E.g., model call, task, project, client, global.
content	"All endpoints use OAuth2."	The payload text.
source_reference	doc_api_req_v2	Pointer to original context/document.
creator	arch_agent_01	Agent or human who originated the memory.
creation_timestamp	2026-07-24T10:00:00Z	When the memory was created.
valid_from	2026-07-24T10:00:00Z	Start of temporal validity.
verification_status	approved	Unverified, inferred, source-backed, approved, superseded.
authority_level	approved project decision	Transient output, user statement, source document, policy.
sensitivity	internal	Public, internal, confidential, restricted, secret.
confidence	0.98	Algorithmic or assigned trust score.
superseded_by	null	Pointer to replacement

		memory, if stale.
--	--	-------------------

Memory Ingestion and Write Policy

Information must enter long-term memory through a highly governed write pipeline. Automatic summarization of every conversation pollutes the database with noise, transient hallucinations, and trivialities.¹² Model-generated summaries should not be stored automatically without validation, and low-confidence inferences must never enter durable semantic memory.

The Memory Write Pipeline:

- 1. Candidate information is identified.
- 2. Scope is determined and source attached.
- 3. Sensitivity is classified.
- 4. Duplication and contradiction checks are executed against active records.²⁸
- 5. Authority is assigned.
- 6. Verification is performed (algorithmic or human).
- 7. Retention policy is attached.
- 8. Memory is written, indexes updated, and an audit event recorded.

One agent must not unilaterally overwrite another agent's memory unless possessing higher structural authority. Humans should approve memories that alter organizational policies or critical project architecture. Active memories should be append-only regarding their audit trail; corrections are represented by creating a new version and marking the old as superseded_by. Disagreements are stored by retaining both competing facts linked by a contradiction edge, pending human adjudication.

Proposed Rule: Agents may propose durable memories, but only authorized system components validate scope, provenance, permissions, and lifecycle before committing them.

Retrieval Architecture

Retrieval must prioritize security and scope over sheer semantic similarity. The retrieval pipeline operates as follows:

- 1. **Request Interpretation:** Determine query intent.
- 2. **Scope & Permission Resolution:** Extract Tenant ID, Project ID, and User ID.
- 3. **Permission Filtering:** Apply a hard deterministic filter to the database to isolate only authorized records.
- 4. **Candidate Retrieval:** Execute hybrid search (Vector + Lexical/BM25) solely within the pre-filtered subset.²
- 5. **Scoring & Reranking:** Apply recency, authority, and provenance weighting.
- 6. **Contradiction Checks:** Filter out memories marked as superseded.
- 7. **Diversity & Token Budgeting:** Select a non-redundant subset fitting the context limit.
- 8. **Context Formatting:** Assemble the final prompt payload with citation attachments.

Filtering before semantic search is paramount. A retrieval query must not search global

organizational memory and subsequently attempt to remove another client's data, as nearest-neighbor proximity calculations will already be cross-contaminated by unauthorized vectors.

Relevance Scoring

A mathematical approach is required to balance competing retrieval priorities. An illustrative scoring function: $\text{Relevance} = \text{Hard_Filters}(\text{Scope}, \text{Permissions}) \times$
Access permissions and project scope must function as hard Boolean filters (0 or 1). Semantic similarity cannot override client isolation; an identical vector match belonging to Client B must yield a 0 for Client A. Furthermore, newer information should not always override authoritative information—a highly authoritative system constraint (e.g., "Do not delete root files") outranks a newer, hallucinated agent instruction. Highly similar content may be outdated, and highly authoritative content may be irrelevant to the specific prompt. Model-based rerankers add value for high-stakes diversity selection but introduce unacceptable instability, latency, and cost for real-time, multi-turn chat loops, where deterministic RRF scoring is vastly superior.

Memory Retrieval Failures

A robust system requires diagnosing and mitigating retrieval failures.

Failure Mode	Cause & Symptom	Prevention & Recovery
Irrelevant / Missing	Poor embeddings / BM25 miss. Symptoms: Hallucinations.	Hybrid search implementation. Query expansion.
Over-retrieval / Duplicate	Redundant ingestion. Symptoms: Prompt bloat, latency.	Deduplication upon ingestion. Maximal Marginal Relevance (MMR) during selection.
Stale Retrieval	Failure to tombstone old data. Symptoms: Reverting to old APIs.	Validity windows. Bitemporal edge invalidation. ⁷
Cross-Client Leakage	Vector similarity overriding namespaces. Symptoms: Massive security breach.	Hard RLS metadata filtering prior to ANN search.

Authority Inversion	Agent guesses outranking SOPs. Symptoms: Policy violation.	Explicit authority weighting multipliers in the scoring function.
Contradictory Memories	Merging unverified hypotheses with facts. Symptoms: Paralyzed planning.	Epistemic separation (Opinion vs World networks). ¹⁹
Summary Distortion	Repeated abstractive compression. Symptoms: Loss of nuance.	Archival retention of raw sources; extraction over summarization.
Embedding Drift	Changing embedding models without re-indexing. Symptoms: Search failure.	Version-locking embedding pipelines.

Summarisation, Compaction, and Consolidation

Compaction prevents unbounded memory growth. However, repeated abstractive summarization poses severe risks: loss of nuance, removal of dissent, factual distortion, disappearing provenance, and model-generated errors calcifying into permanent reality.

A safe compaction pipeline operates via extraction rather than abstraction:

1. Identify a temporal compaction boundary.
2. Preserve raw source material entirely.
3. Extract explicit factual claims and entity relationships.²
4. Attach provenance pointers to the raw source.
5. Detect contradictions against the existing database.
6. Mark confidence and version the summary.
7. Store the summary in the active retrieval index while archiving the raw source to cold storage for future inspection.

Compaction must never silently destroy source evidence. Raw task logs should be archived, while verified architectural decisions are retained verbatim.

Hierarchical Memory Architecture

Memory must be architected in distinct layers, enforcing specific lifecycle policies:

- **Layer 0: Immediate model context.** Contains only data for the current call. Stored in RAM. No retention.
- **Layer 1: Active task workspace.** Contains the current plan, task state, and blockers. Stored in Redis or KV. Flushed upon task completion.
- **Layer 2: Workflow-run state.** Contains durable execution history, checkpoints, and

approvals. Stored in PostgreSQL via Checkpointer. Append-only, retained per compliance rules.

- **Layer 3: Project knowledge.** Contains approved project facts, architecture, and constraints. Stored in Hybrid DB. Requires validation. Context-injected upon task start.¹⁷
- **Layer 4: Client knowledge.** Reusable information for one client. Isolated via strict RBAC.
- **Layer 5: Organisational knowledge.** Approved policies and SOPs. Human-verified. Read-only retrieval for agents.
- **Layer 6: Archive and provenance.** Raw transcripts, logs, and superseded documents. Stored in S3. Immutable.

Information moves upward organically but requires friction. A temporary task insight (Layer 1) does not automatically become organizational policy (Layer 5). An approved task decision may become project memory (Layer 3), and repeated, successful cross-project practices may eventually be reviewed for organizational promotion (Layer 5).

Project Knowledge Architecture

Persistent project knowledge requires rigorous structure to prevent chaotic text accumulation. Key categories include project charters, architecture decisions, terminology, approved specifications, environment dependencies, known risks, and artefact catalogues.

Project knowledge is best organized through a combination of structured database records and authoritative files, rather than one monolithic markdown file or a purely unstructured document index. The **Project Manifest** serves as the critical router, identifying authoritative document URIs, current versions, trusted namespaces, and retrieval priorities. When agents initialize project context at the beginning of a task, they parse the manifest to load foundational constraints into Layer 1, allowing deeper architectural histories in Layer 3 to be fetched dynamically via specific tool calls.³⁰

Decisions, Assumptions, and Facts

Merging disparate cognitive assertions into an undifferentiated vector store causes agents to treat their own past assumptions as verified facts. Epistemic separation is required⁵:

- **Facts:** Claims describing reality, supported by sources (e.g., "The server is Ubuntu").
- **Decisions:** Authorized choices (e.g., "Use PostgreSQL"). Includes decision-maker and rationale.
- **Assumptions:** Temporary beliefs. Schema must include owner, expiration_date, risk, and validation_plan.
- **Hypotheses:** Propositions requiring investigation.
- **Preferences/Opinions:** Subjective weights (e.g., "User prefers dark mode"). Stored with a confidence score.¹⁹
- **Observations:** What a tool reported during execution.

Versioning, Provenance, and Lineage

Memory systems require robust bitemporal versioning, an approach popularized by temporal

knowledge graphs like Zep.⁶ This tracks *Event Time* (when the fact was true) versus *Ingestion Time* (when the system recorded it). Old records are not silently overwritten; they are mutated into immutable historical views with a superseded_by pointer.

Provenance chains trace lineage explicitly: Source PDF → Extracted Claim → Validated Project Fact → Compact Summary → Retrieved Context. Agents and human auditors must be able to inspect this lineage to verify truth. Model-generated content lacking traceable external sources must never be promoted to high-authority organizational memory.

Preventing Outdated or Incorrect Memories

A comprehensive stale-memory prevention strategy avoids treating semantic similarity as truth. A stale-memory pipeline utilizes validity periods, review dates, and automatic revalidation. When handling changed product specifications, changed APIs, or shifting user preferences, contradiction detection triggers.²⁸

Contradictions cannot be resolved simply by "newest-wins," as an unauthorized agent prompt could overwrite an authoritative system policy. Resolution follows a source-priority rule: highest-authority-wins. If authorities are equal, the system retains both, flags as disputed, and escalates for human adjudication. The memory-status model clearly marks records as: current, potentially stale, under review, disputed, superseded, archived, or deleted.

Memory Confidence and Trust

Memory trustworthiness incorporates source authority, the number of independent corroborating sources, validation methods, age, and domain stability. Crucially, a model's self-generated confidence score should never be treated as equivalent to factual reliability. Systems must distinguish between *retrieval confidence* (BM25/Vector match quality), *factual confidence* (corroborated truth), *source confidence* (origin authority), and *applicability confidence* (task relevance). Agents should be prompted to communicate uncertainty explicitly when utilizing low-confidence or unverified memories.

Memory Sharing Between Agents

Multi-agent systems (MAS) rely on architectural patterns for knowledge sharing:

- **Shared global memory:** High contamination risk. Avoid unless tightly governed.
- **Fleet memory:** Agents within a specific group share selected operational insights.
- **Project memory:** Access to common project knowledge, heavily utilized.
- **Task-local memory:** Ephemeral sharing between agents on a single task.
- **Agent-private memory:** A private working scratchpad.
- **Publish-subscribe memory:** Agents publish structured facts to topics.
- **Blackboard memory:** Agents read/write to a shared structured workspace. Excellent for emergent consensus.³²

By default, agents should share verified artefacts, task status, approved decisions, dependency data, and blockers. They should *not* share unrestricted internal scratch notes, unverified guesses, raw chain-of-thought, or entire conversation histories.

Cross-agent Hand-offs

Passing complete conversation transcripts between agents causes exponential token growth. Cross-agent hand-offs utilize structured JSON schemas detailing: `source_agent`, `destination_agent`, `project`, `task_objective`, `completed_work`, `current_state`, `relevant_artefact_URLs`, `verified_facts`, `unresolved_assumptions`, `blockers`, `permissions`, and `provenance_links`.

For example, a planner agent hands off to a coding agent by submitting a verified architecture outline and repository pointers, rather than its internal exploratory reasoning. The coding agent then hands off to a test agent, supplying only the finalized code URLs and test requirements.

Project and Client Isolation

Strong multi-tenant isolation is the paramount security requirement. Prompt engineering (e.g., "do not reveal another client's information") is computationally reckless and insufficient.

Defence in depth requires:

1. **Namespaces & Tenant IDs:** Every memory record contains a hard tenant ID.
2. **Database Row-Level Security (RLS):** Queries inherently reject rows lacking the active execution token.
3. **Separate Indexes:** Physically separate vector collections for disparate clients.
4. **Retrieval Filters:** Pre-search metadata filtering.

Applying cross-client filters only *after* vector search fails because nearest-neighbor calculations are fundamentally skewed by unauthorized data. Full tenant isolation (separate databases) is required for high-sensitivity or regulated commercial environments.

Privacy and Permissions

Memory governance mandates adherence to consent, purpose limitation, and data minimization. Systems must enforce retention periods, deletion rights, and access logs.

Permissions must be granularly distinguished:

- **Permission to read:** Viewing metadata.
- **Permission to retrieve:** Searching contents.
- **Permission to insert into model context:** Crucial distinction. Data may be legally stored but inappropriate to place into an unrestricted agent's prompt.
- **Permission to share/export/delete.**

Privacy-preserving techniques include pre-embedding redaction of PII, tokenization, field-level encryption, and data-loss prevention rules preventing sensitive data from leaving secure enclaves.

Memory Security and Poisoning

Memory poisoning via indirect prompt injection is a severe threat.³⁴ An attacker places malicious instructions in a document; the agent retrieves it, embedding the payload directly into its context window as if it were a system instruction.

The system must distinguish data from executable commands. Protections include:

1. **Instruction-data separation:** Wrapping retrieved context in strict <data> tags.
2. **Trust labels:** High-authority system prompts explicitly override low-authority retrieved web data.
3. **Read-only retrieval:** Sandboxing the agent from making state changes based on unverified external memory.
4. **Confirmation checks:** Human-in-the-loop approval before consequential actions. Retrieved content must never automatically become an instruction to the agent.

Forgetting and Memory Lifecycle

Indefinite retention creates massive liability and latency. Systems must intentionally forget expired task state, temporary notes, obsolete assumptions, and revoked user data.

- **Tombstoning / Soft Deletion:** Removing data from active retrieval but preserving the audit log. Used for superseded project facts.
- **Hard Deletion:** Cryptographic erasure of records and embeddings. Mandatory for expired client data and PII.
- **Lifecycle States:** Candidate, Active, Verified, Ageing, Review Required, Superseded, Archived, Deleted. A background garbage-collection daemon (e.g., Claude's Dream/KAIROS) sweeps the database, executing retention policies without interrupting real-time inference.²⁵

Memory Consolidation and Promotion

Local task memories become shared knowledge through governed promotion pipelines. For example: an agent observes a task issue; it records it as an episodic memory; the system detects repeated occurrences and proposes a candidate reusable insight; a human supervisor validates it; it is promoted to project guidance.

Promotion must preserve evidence, exceptions, scope constraints, and the original owner. This strict governance prevents one unusual anomaly from becoming an organization-wide rule.

Event Logs versus Semantic Memory

An event log answers *what happened* (e.g., "Agent X ran script Y"). Semantic memory answers *what is currently believed to be true* (e.g., "The script requires Python 3.12"). Architectures derive current semantic views from event histories via projections or materialised views.⁹ Event sourcing is highly justified for auditable workflows and financial tracking, whereas ordinary versioned records are sufficient for standard project documentation.

Knowledge Graphs and Relationship Memory

Agent memory benefits from explicit graph structure when relationships are dense and temporal tracking is essential.⁶ Entities (people, projects, files) connect via relationships (owns, contradicts, produced_by). Graphs enable complex multi-hop retrieval and entity resolution. However, a knowledge graph is massive unnecessary complexity for a simple Q&A agent; it is

only appropriate for advanced enterprise environments requiring temporal fact supersession and cross-domain reasoning.

Model-Independent Memory Interfaces

Architectures must decouple memory from specific LLM providers. The Memory API operates independently, exposing structured contracts for `store_memory`, `retrieve_memory`, `supersede_memory`, and `verify_memory`.³⁵ The service returns structured JSON containing content, source references, scope, authority, version, and retrieval scores. The integration layer then formats this payload specifically for the active model provider (e.g., Anthropic XML tags or OpenAI system messages).

Memory Middleware and Context Assembly

A Context-Assembly service must sit between the agent runtime and memory stores. Individual agents should *never* directly query unrestricted databases. The middleware authenticates the agent, enforces RBAC, executes hybrid retrieval, filters stale items, strictly manages token usage, attaches provenance, and formats the provider-specific context package. The final package includes system policies, active workflow state, selected project facts, and unresolved contradictions.

Token Budgeting and Context Selection

Memory retrieval under limited context budgets utilizes dynamic allocation. The system does not blindly insert the top k vector matches. Budget allocation prioritizes:

1. Hard constraints (System policy, Output instructions).
2. Current task state.
3. High-authority project rules.
4. Source evidence.
5. Diversity-selected historical episodes. Agents are provided a tool call to explicitly request additional paginated context if the initial budget was insufficient.

Multi-Agent Memory Consistency

Concurrent agents reading and writing memory introduce race conditions, stale reads, and conflicting decisions. Memory types requiring strong consistency (task ownership, approval states, active specifications) must utilize Postgres locks, optimistic concurrency (compare-and-swap), or strict append-only ledgers.² Search indexes and aggregated analytics can safely tolerate eventual consistency.

Persistent Agent Identity versus Persistent Execution

A "persistent agent" should be implemented as a persistent identity and memory namespace activated on demand, rather than a continuously running, indefinitely looping model process. The agent retains its role, capabilities, project assignments, and historical task summaries via

the external database, avoiding the need to retain an indefinitely growing raw conversation thread in memory.

Practical Memory Patterns

35.1 Hybrid structured and semantic memory: Uses Postgres for deterministic workflow tracking and pgvector for semantic search. Ideal for complex development fleets. **35.2**

Blackboard shared memory: A central datastore where specialized agents read/write structured findings. Excellent for multi-agent synthesis, but requires a strong Control Unit to manage turn-taking.³² **35.3 Architecture decision record system:** High-authority files (e.g., markdown in Git) injected as top-priority context. Overrides probabilistic semantic search.

35.4 Event-sourced run memory: Uses systems like Temporal to ensure durable execution. Essential for tasks spanning hours or days.⁹

Comparison of Frameworks and Technologies

- **LangGraph:** Excellent for workflow state and multi-turn memory via its Checkpointer (PostgresSaver). Provides durable execution boundaries.⁸
- **Temporal / Restate:** Enterprise-grade durable execution. They manage deterministic replay, timers, and human-in-the-loop logic, fundamentally separating workflow state from semantic memory.²¹
- **Mem0 / Letta:** Dedicated memory management layers. Mem0 excels at LOCOMO benchmarks and token reduction.³ Letta provides hierarchical Core/Recall/Archival structures.³⁸
- **Zep (Graphiti):** State-of-the-art for temporal knowledge graphs. Handles bitemporal logic and fact invalidation exceptionally well.⁶
- **PostgreSQL:** The essential backbone for structured state, metadata, vector extensions (pgvector), and full-text search.

Proposed Layered Agent Memory System

The **Layered Agent Memory and Project Knowledge System** comprises:

- **Memory API & Context-Assembly Service:** The gatekeeper. Enforces RBAC and token budgets. Required.
- **Workflow-State Store:** PostgreSQL (LangGraph Saver). Owns durable execution. Required.
- **Project & Client Knowledge Store:** Hybrid DB. Owns isolated semantic memory. Required.
- **Document/Artefact Repository:** S3 or Git. Owns large files and code. Required.
- **Contradiction & Version Manager:** Resolves supersession before DB writes. Required for scale.
- **Knowledge Graph:** Optional. Used only for intense temporal/relational reasoning.

Architecture Diagrams

Diagram 1: Layered memory architecture

Code snippet

```
graph TD
    subgraph Layer_0_1 [Layer 0 & 1: Agent Runtime]
        A[Agent LLM] <--> B
    end
    subgraph Layer_2 [Layer 2: Workflow Engine]
        B <--> C
    end
    subgraph Layer_3_4 [Layer 3 & 4: Semantic Knowledge]
        C -- Tool Calls --> D[Context Assembly Middleware]
        D --> E[RBAC & Scope Filter]
        E --> F
        E --> G[(Vector Index)]
    end
    subgraph Layer_6 [Layer 6: Archive]
        F -- Superseded / Large Files --> H
    end
```

Diagram 2: Memory-write pipeline

Code snippet

```
graph LR
    A[Agent Proposes Memory] --> B
    B --> C
    C --> D{Contradiction Check}
    D -- Conflict Found --> E
    E -- Existing Wins --> G
    E -- Candidate Wins --> F
    D -- No Conflict --> H
    F --> H
    H --> I[Update Vector Embeddings]
    I --> J[Log Audit Event]
```

Diagram 3: Memory-retrieval pipeline

Code snippet

graph LR

```
A --> B
B --> C
C --> D
D --> E
E --> F
F --> G[Context Assembly & Citation]
G --> H[Model Call]
```

Diagram 4: Memory promotion

Code snippet

graph TD

```
A -->|Pattern Detected| B[Candidate Project Insight]
B --> C{Human/Supervisor Review}
C -- Approved --> D[Project Memory]
D -->|Cross-Project Evidence| E{Governance Review}
E -- Approved --> F[Organisational Policy]
```

Diagram 5: Project and client isolation

Code snippet

graph TD

```
A[Context Middleware] --> B{Tenant Resolver}
B -- Client A Token --> C
B -- Client B Token --> D
C -. -> E[Hard Isolation]
D -. -> E
```

Diagram 6: Contradiction and supersession

Code snippet

graph LR

```

A --> B{Contradiction Detector}
B -- Matches API Spec V1 --> C[Authority Check]
C -- V2 Higher/Equal --> D
D --> E
E --> F

```

Diagram 7: Multi-agent sharing

Code snippet

graph TD

```

A[Agent 1: Planner] -->|Writes| B
A -->|Publishes| C
C -->|Reads| D[Agent 2: Coder]
D -->|Writes| E
D -->|Publishes Artefact URI| C

```

Algorithms and Pseudocode

Algorithm: Hybrid Lexical and Semantic Search with Relevance Scoring

Python

```

def retrieve_memory(query, tenant_id, project_id, token_limit):
    # 1. Deterministic Hard Filtering (Must happen first)
    allowed_records = db.query(
        "SELECT id, metadata FROM memories WHERE tenant_id =? AND project_id =? AND status
= 'active'",
        tenant_id, project_id
    )
    if not allowed_records:
        return

    # 2. Semantic Search (Vector)
    query_vector = embed(query)
    semantic_results = vector_store.similarity_search(query_vector, filter_ids=allowed_records,
top_k=20)

    # 3. Lexical Search (BM25)
    lexical_results = text_index.bm25_search(query, filter_ids=allowed_records, top_k=20)

```

```
# 4. RRF Scoring & Weighting
scored_candidates = {}
for res in union(semantic_results, lexical_results):
    base_score = rrf(res.semantic_rank, res.lexical_rank)
    authority_weight = get_authority_multiplier(res.authority_level)
    time_decay = calculate_decay(res.valid_from)

    scored_candidates[res.id] = base_score * authority_weight * time_decay

# 5. Token Budgeting
sorted_memories = sort_by_score(scored_candidates)
return pack_context_within_budget(sorted_memories, token_limit)
```

Practical Scenarios

- **Scenario A: Software-development project.** The planner agent reads the Project Manifest (Layer 3) to establish constraints. The coder agent writes transient code to its private scratchpad (Layer 1). The reviewer agent reads the shared blackboard, validates the code, and writes test results to workflow state (Layer 2). Verified coding conventions update the organizational memory (Layer 5).
- **Scenario B: Long-running research project.** Agents explore topics over weeks. Raw articles are saved to S3. Extracted findings are written to the Semantic database. When a contradiction is found, the system evaluates authority (e.g., peer-reviewed outranks preprint), supersedes the old finding, and retains the lineage.
- **Scenario C: Marketing fleet.** Campaign brainstorming remains in ephemeral task memory. Approved brand rules (Client Knowledge, Layer 4) are immutably retrieved by copywriter agents, ensuring temporary creative ideas never contaminate the core brand constraints.
- **Scenario D: Multiple client projects.** Client A and Client B queries hit the middleware. Row-Level Security explicitly drops any vector matches from the opposing client's namespace *before* semantic sorting occurs, guaranteeing zero cross-contamination regardless of semantic similarity.
- **Scenario E: Product specifications change.** A hardware spec updates from 5V to 12V. The contradiction detector identifies the clash. The 5V record is marked *superseded_by*, dropping it from active retrieval. Provenance links allow the system to warn developers about legacy artifacts generated under the old 5V assumption.
- **Scenario F: Agent-generated incorrect memory.** An agent proposes a false conclusion. During the write pipeline, a human-in-the-loop or high-authority deterministic check rejects the promotion. The memory remains in the audit log but never enters active semantic retrieval.
- **Scenario G: Model-provider replacement.** Transitioning from OpenAI to Anthropic requires zero data migration. The Memory API is model-agnostic. The Context-Assembly

service simply maps the retrieved JSON payload into Anthropic's XML tag format instead of OpenAI's system message format.

Anti-patterns

- **Treating context window as permanent memory:** Leads to token limits, latency, and hallucination.¹ *Replacement:* Layered external retrieval.
- **Newest-wins for conflicts:** Causes hallucinated agent guesses to overwrite system policies. *Replacement:* Authority-weighted supersession and contradiction detection.
- **Vector search as Source of Truth:** Fails on negations and temporal logic.²⁸ *Replacement:* RDBMS as metadata truth; vectors merely as search indexes.
- **Mixing workflow state with semantic knowledge:** Breaks deterministic resumption.⁹ *Replacement:* Workflow engines (Temporal/LangGraph) for state; Mem0/Zep for semantics.
- **Cross-client filters applied after search:** Causes severe data leakage due to ANN neighborhood approximation. *Replacement:* Hard RLS metadata filtering prior to vector search.
- **Knowledge graphs without clear use cases:** Adds massive operational complexity without ROI. *Replacement:* Standard Postgres hybrid search.

Testing Strategy

A comprehensive testing strategy isolates retrieval mechanics from generation mechanics.

- **Isolation Tests:** Send a query from Client A designed to match Client B's unique keyword. Assert the API returns a null set.
- **Contradiction Tests:** Inject Fact A, then Fact B (contradicts A, but higher authority). Assert Fact B is retrieved and Fact A is superseded.
- **Stale-memory Tests:** Query a superseded API spec; assert the new version is returned with provenance pointers.
- **Prompt-injection Tests:** Insert a malicious prompt into an indexed document. Assert the system tags it as <data>, preventing the agent from executing it as an <instruction>.
- **Crash-recovery Tests:** Terminate an agent mid-task. Assert it resumes utilizing LangGraph PostgresSaver without repeating non-idempotent tool calls.³⁹

Evaluation Framework

Evaluation requires a weighted scorecard across operational and quality metrics:

Category	Metric	Description
Retrieval Quality	Mean Reciprocal Rank (MRR)	Accuracy of retrieving the correct semantic memory.

	Contradiction Rate	Frequency of superseded facts surfacing in context.
Security & Isolation	Cross-Client Leakage Rate	Must be 0%. Measured via penetration testing.
Memory Quality	Summary Fidelity	Information retained post-compaction (vs raw source).
	Provenance Completeness	% of memories traceable to an original source.
Performance	Context Size & Token Cost	Reduction in tokens compared to full-context baselines. ³
	Retrieval Latency	Target < 500ms p95.

Observability

The memory operations dashboard must expose: retrieval latency, context tokens supplied (monitoring for prompt bloat), contradictions flagged (indicating conflicting source documents), permission denials, and vector-index size. Crucially, lineage traces must allow administrators to visually inspect exactly which source document generated an agent's specific output, enabling rapid debugging of hallucinations.

Minimum Viable Architecture

For a solo builder or small team, avoiding over-engineering is critical. The minimum viable stack consists of:

- Version-controlled project files (CLAUDE.md) via Git.
- PostgreSQL + LangGraph SqliteSaver/PostgresSaver for structured workflow state.²⁶
- PostgreSQL metadata filtering + BM25 full-text search.
- Manual promotion of durable memories.
- *Do not build initially:* Complex knowledge graphs, automated organizational consolidation, or continuous memory-reflection daemons.

Implementation Roadmap

- **Stage 1: Project files & task state.** Implement Git-backed manifests and basic checkpoints.

- **Stage 2: Durable workflow memory.** Integrate LangGraph PostgresSaver or Temporal for event history and resumability.
- **Stage 3: Search and retrieval.** Add Postgres full-text search (BM25) with hard scope metadata filters.
- **Stage 4: Semantic retrieval.** Add pgvector, hybrid search (RRF), and establish an evaluation dataset.
- **Stage 5: Memory governance.** Implement bitemporal versioning, contradiction handling, and explicit supersession logic.
- **Stage 6: Multi-agent sharing.** Deploy the Context-Assembly middleware, structured hand-offs, and pub/sub blackboards.
- **Stage 7: Multi-client memory.** Enforce strict tenant isolation (RLS), privacy auditing, and access policies.
- **Stage 8: Advanced consolidation.** Implement automated compaction and entity-resolution knowledge graphs (only if justified by extreme scale).

Final Recommended Design

The central hypothesis is unequivocally validated: reliable agent memory cannot exist as an indefinitely growing conversation transcript. It must be a governed, layered, externally managed knowledge and state architecture.

Immediate context must remain small and task-specific. Workflow state must be explicitly structured in durable databases (e.g., Postgres, Temporal). Project and organizational knowledge must reside in heavily governed hybrid databases utilizing both lexical and semantic retrieval. Cross-client isolation must be enforced via RDBMS infrastructure, never prompt engineering. Verified knowledge and authoritative files must always outrank an agent's self-generated recollection. By adopting this layered architecture, engineers guarantee that their agentic systems remain secure, contextually relevant, highly auditable, and resilient to system failures, providing the foundational stability required for enterprise-scale AI deployment.

Works cited

1. Agent Memory Architectures: 5 Patterns and Trade-offs - Atlan, accessed July 25, 2026, <https://atlan.com/known/agent-memory-architectures/>
2. [Literature Review] Hindsight is 20/20: Building Agent Memory that Retains, Recalls, and Reflects - Moonlight, accessed July 25, 2026, <https://www.themoonlight.io/en/review/hindsight-is-2020-building-agent-memory-that-retains-recalls-and-reflects>
3. MemO: Building Production-Ready AI Agents with Scalable Long-Term Memory - arXiv, accessed July 25, 2026, <https://arxiv.org/html/2504.19413v1>
4. HINDSIGHT: Structured Agent Memory that Retains, Recalls, and Reflects - ACL Anthology, accessed July 25, 2026, <https://aclanthology.org/2026.acl-demo.27.pdf>
5. Hindsight: Structured Memory in Agent AI - Emergent Mind, accessed July 25, 2026, <https://www.emergentmind.com/papers/2512.12818>

6. Zep: Temporal Knowledge Graph Architecture - Emergent Mind, accessed July 25, 2026, <https://www.emergentmind.com/topics/zep-a-temporal-knowledge-graph-architecture>
7. Zep (Graphiti) vs Cognee: AI Agent Memory Compared (2026) - Vectorize, accessed July 25, 2026, <https://vectorize.io/articles/zep-vs-cognee>
8. langgraph-persistence | Agent Skills Library - Awesome MCP Servers, accessed July 25, 2026, <https://mcpservers.org/agent-skills/langchain-ai/langgraph-persistence>
9. Durable Execution for AI Agent Runtimes: Checkpointing, Replay, and Recovery - Zylos, accessed July 25, 2026, <https://zylos.ai/research/2026-04-24-durable-execution-agent-runtimes/>
10. Agent Memory: How to Build Agents That Learn and Remember - Letta, accessed July 25, 2026, <https://www.letta.com/blog/agent-memory/>
11. AI Agent Memory Systems in 2026: Mem0, Zep, Hindsight, Memvid and Everything In Between — Compared | by Yogesh Yadav | Dev Genius, accessed July 25, 2026, <https://blog.devgenius.io/ai-agent-memory-systems-in-2026-mem0-zep-hindsight-memvid-and-everything-in-between-compared-96e35b818da8>
12. langchain-skills/config/skills/langgraph-persistence/SKILL.md at main - GitHub, accessed July 25, 2026, <https://github.com/langchain-ai/langchain-skills/blob/main/config/skills/langgraph-persistence/SKILL.md>
13. Giving Agents a Memory: Vector Search with PostgreSQL - CFCamp, accessed July 25, 2026, <https://www.cfcamp.org/sessions/vector-search-with-postgresql.html>
14. letta-ai/ai-memory-sdk: An experimental SDK for adding agentic memory and learning in a pluggable way - GitHub, accessed July 25, 2026, <https://github.com/letta-ai/ai-memory-sdk>
15. Memory Architecture Patterns | The AI Agent Factory - Panaversity, accessed July 25, 2026, <https://agentfactory.panaversity.org/docs/Building-Agent-Factories/augmented-memory/memory-architecture-patterns>
16. Temporal — What does “preserving state” really mean? | by Cornelia Davis | Medium, accessed July 25, 2026, <https://blog.corneliadavis.com/temporal-what-does-preserving-state-really-mean-ebdca256526f>
17. The CLAUDE.md Memory System - Deep Dive - SFEIR Institute, accessed July 25, 2026, <https://institute.sfeir.com/en/claude-code/claude-code-memory-system-claude-md/deep-dive/>
18. Giving Pi Agent Project Memory with Mem0: A Practical Schema Migration Demo, accessed July 25, 2026, <https://mem0.ai/blog/giving-pi-agent-project-memory-with-mem0-a-practical-schema-migration-demo>

19. HINDSIGHT: Building Agent Memory That Retains, Recalls, and Reflect - Vaibhav Phutane, accessed July 25, 2026, <https://vap1231.medium.com/hindsight-building-agent-memory-that-retains-recalls-and-reflect-5e84709bd8f9>
20. Zep: A Temporal Knowledge Graph Architecture for Agent Memory - arXiv, accessed July 25, 2026, <https://arxiv.org/html/2501.13956v1>
21. Restate vs Temporal, accessed July 25, 2026, <https://www.restate.dev/vs/temporal>
22. The definitive guide to Durable Execution - Temporal, accessed July 25, 2026, <https://temporal.io/blog/what-is-durable-execution>
23. The CLAUDE.md Memory System - SFEIR Institute, accessed July 25, 2026, <https://institute.sfeir.com/en/claude-code/claude-code-memory-system-claude-md/>
24. Agent Memory Engineering - Nicolas Bustamante, accessed July 25, 2026, <https://nicolasbustamante.com/blog/agent-memory-engineering>
25. Claude Code Memory System Explained: 4 Layers, 5 Limits, and a Fix - Milvus Blog, accessed July 25, 2026, <https://milvus.io/blog/claude-code-memory-memsearch.md>
26. checkpoints | langgraph - LangChain Reference, accessed July 25, 2026, <https://reference.langchain.com/python/langgraph/checkpoints>
27. MemO (Memo.ai) Memory Layer — Purpose and Core Functionality - Stackademic, accessed July 25, 2026, <https://blog.stackademic.com/mem0-memo-ai-memory-layer-purpose-and-core-functionality-375cc5a2bfd0>
28. Agent Memory Garbage Collection: Engineering Strategic Forgetting, accessed July 25, 2026, <https://tianpan.co/blog/2026-04-14-agent-memory-garbage-collection>
29. Complete guide to Knowledge & Context Graphs | via Zep & Graphiti - Medium, accessed July 25, 2026, <https://medium.com/@whynesspower/complete-guide-to-knowledge-context-graphs-via-zep-graphiti-c6da7ce8b13b>
30. Memory tool - Claude Platform Docs, accessed July 25, 2026, <https://platform.claude.com/docs/en/agents-and-tools/tool-use/memory-tool>
31. Hindsight: Structured Agent Memory that Retains, Recalls, and Reflects - ACL Anthology, accessed July 25, 2026, <https://aclanthology.org/2026.acl-demo.27/>
32. Patterns for Democratic Multi-Agent AI: Blackboard Architecture — Part 1 | by edoardo schepis | Medium, accessed July 25, 2026, <https://medium.com/@edoardo.schepis/patterns-for-democratic-multi-agent-ai-blackboard-architecture-part-1-69fed2b958b4>
33. LbMAS Implementation: Multi-Agent LLM System - Emergent Mind, accessed July 25, 2026, <https://www.emergentmind.com/topics/lbmas-implementation>
34. Memory poisoning in AI agents: exploits that wait - Christian ..., accessed July 25, 2026, <https://christian-schneider.net/blog/persistent-memory-poisoning-in-ai-agents/>
35. memO/LLM.md at main - GitHub, accessed July 25, 2026,

<https://github.com/mem0ai/mem0/blob/main/LLM.md>

36. langgraph.checkpoint.postgres - LangChain Reference, accessed July 25, 2026, <https://reference.langchain.com/python/langgraph/checkpoint/postgres>
37. Chapter 7: Workflows: Durable Execution | Architecting on Cloudflare, accessed July 25, 2026, <https://architectingoncloudflare.com/chapter-07>
38. Letta: Platform for Stateful LLM Agents | by DhanushKumar - Stackademic, accessed July 25, 2026, <https://blog.stackademic.com/letta-platform-for-stateful-llm-agents-a83b58a1c926>
39. Mastering LangGraph Checkpointing: Best Practices for 2025 - Sparkco, accessed July 25, 2026, <https://sparkco.ai/blog/mastering-langgraph-checkpointing-best-practices-for-2025>