

Hierarchical Memory Management for Long-Running Conversational Agents

Saurabh Bidhuri, Malay Shah, Ashutosh Nigam

Adobe Inc.

India

ABSTRACT

Conversational agents powered by large language models (LLMs) face a fundamental tension between context richness and context-window limits. As conversations grow, an agent must decide what to keep verbatim, what to compress, and what to persist across sessions. Existing approaches either naively concatenate history until it overflows or maintain monolithic stores that scale poorly. We present a three-layer hierarchical memory architecture: a short-term working set of recent messages, a medium-term session context with automatic progressive summarization, and a long-term persistent knowledge store. Memory is managed by a dedicated Context Server—a standalone microservice—so that memory operations are decoupled from the agent's reasoning loop. We describe the implementation as deployed in a production enterprise knowledge assistant, and evaluate it with a controlled benchmark that exercises the live memory service. The benchmark shows that the assembled context stays bounded—roughly 3.4K tokens regardless of conversation length—yielding an $8.6\times$ reduction versus naive full history at turn 20 and a $21\times$ reduction at turn 50, that the common assembly path adds only about 15 ms, and that the system degrades gracefully (falling back to local session state in about 3 ms) when the memory service is unavailable. Long-term extraction persists durable knowledge across sessions; measuring its recall under load exercises the shared model gateway, which was unavailable in our measurement environment (Section VII). We additionally report production corpus statistics, and discuss future directions including semantic indexing and federated memory.

Keywords:- conversational agents, memory management, context window, session management, LLM agents, progressive summarization, hierarchical memory.

I. INTRODUCTION

A fundamental limitation of LLM-based conversational agents is the finite context window. Even with modern models supporting 100K–200K tokens, long-running conversations and multi-session interactions quickly exhaust the available budget. The naive approach—concatenating all prior messages—is unsustainable: a 50-message conversation interleaved with tool results can easily consume tens of thousands of tokens, leaving insufficient room for the current query's reasoning and tool outputs.

This paper addresses three specific challenges:

1. **Within-session growth:** as a conversation progresses, how should the agent manage its growing history?
2. **Cross-session continuity:** when a user returns after hours or days, how should the agent recall relevant prior context?
3. **Graceful degradation:** if the memory service is unavailable, how should the agent continue functioning?

We propose a three-layer hierarchical memory architecture that addresses each challenge through a distinct mechanism:

- **Short-term memory (Layer 1):** the recent messages of the current session, passed to the model verbatim.
- **Medium-term memory (Layer 2):** the full session history, compressed through automatic progressive summarization.
- **Long-term memory (Layer 3):** durable knowledge extracted from conversations and persisted across sessions.

A. Key Design Principles

Separation of concerns. Memory management is handled by a dedicated Context Server, not embedded in the agent's reasoning loop. This mirrors the separation between working registers and main memory in computer architecture, and lets the memory service be scaled, tested, and failed independently.

Lossy compression is acceptable. Summaries lose detail but preserve actionable information. This is a feature, not a bug—it mirrors how human memory works: we forget the exact words of yesterday's meeting but retain the decisions made.

Graceful degradation. The agent must not crash when memory services fail. If the Context Server is unreachable, the agent falls back to a locally persisted session file; if that is also unavailable, it proceeds with only the current turn.

Token-budget awareness. The assembled context is held within an explicit token budget, so total memory cost is bounded and predictable regardless of conversation length.

II. MOTIVATION: THE CONTEXT WINDOW PROBLEM

Before describing our solution, it is worth stating the problem precisely. Modern LLMs operate under a hard constraint: the context window has a fixed token limit. While this limit has grown—from a few thousand tokens in early GPT models to 200K in recent Claude models and beyond—the demands placed on conversational agents have grown even faster. The window must accommodate not only conversation history but also system prompts, tool definitions, retrieval results, and space for the model to reason.

A. Concrete Overflow Scenarios

Consider an enterprise knowledge assistant deployed to help engineers and program managers. A typical interaction involves the following token demands.

TABLE I. Token-budget breakdown for a typical enterprise agent.

COMPONENT	TOKEN ESTIMATE	NOTES
System prompt	3,000–5,000	Persona, instructions, guardrails
Tool definitions	6,000–12,000	40+ tools with schemas
Retrieved documents (RAG)	4,000–20,000	Variable per query
Conversation history	Variable	Grows per turn
Current query + reasoning	1,000–5,000	Must be reserved
Output generation budget	4,000–8,000	Space for the response

With a 200K-token model, the fixed overhead alone (system prompt, tools, output budget) can consume 18,000–50,000 tokens. That still leaves ample room—until conversation history grows without bound:

Scenario 1: the research deep-dive. A product manager asks the agent to analyze customer feedback across three quarters. The conversation involves many tool calls to a database, each returning thousands of tokens. Within a few dozen turns the raw history exceeds any comfortable budget, and the agent begins dropping earlier context as the history is truncated.

Scenario 2: the multi-day project. An engineer uses the agent over three days to debug a distributed system. Day 1 establishes architecture; day 2 explores error logs; on day 3 the engineer says “apply the fix we discussed on day 1”—but that context is long gone from any reasonable sliding window.

Scenario 3: the collaborative session. A design team shares one session for brainstorming. Several participants contribute over dozens of messages, including large pasted specifications. The window fills well before the discussion concludes.

B. The Cost of Lost Context

When context overflows, the consequences are structural, not merely cosmetic:

- **Contradictory responses:** the agent contradicts its earlier recommendation because that recommendation has left the context, eroding user trust.
- **Repeated work:** the agent asks the user to re-provide information it was already given.
- **Lost reasoning chains:** multi-step analyses that build on earlier conclusions break when those conclusions leave the context.
- **Broken commitments:** “I’ll use that format from now on” becomes empty when the context is silently truncated.

C. Why Simple Solutions Fail

The obvious fixes each have critical deficiencies. Simply enlarging the context window is not a permanent solution: attention cost scales super-linearly, latency rises, and users always generate more data. Truncating from the beginning loses the original problem statement; truncating from the middle creates incoherent gaps. These observations motivate a structured, hierarchical approach.

D. Evidence from Production

The problem is not hypothetical. Over a five-week window of our production deployment (2,327 user requests; Appendix B), requests were routinely multi-turn—up to 30 turns and 200 tool calls each—and cumulative input grew steeply: the median request consumed 45K input tokens across its LLM calls, the 90th percentile reached 495K, and 531 requests (22.8%) exceeded 200K cumulative input tokens. Prompt caching absorbs much of the repeated context (a median 24K cache-read tokens per request), but the underlying growth is exactly what an explicit memory budget is meant to bound. These figures motivate the bounded, hierarchical assembly described next.

III. RELATED WORK

E. Context-Window Management

Retrieval-Augmented Generation (RAG) [1] augments LLM context with retrieved documents. While effective for knowledge retrieval, RAG does not address conversational memory management; our approach is complementary—RAG can be used within any layer. Work on context compression [2] explores learned compression of input tokens, but such techniques remain experimental and are not yet deployable at production scale.

F. Agent Memory Systems

MemGPT [3] introduced virtual context management inspired by operating-system memory hierarchies. Our work shares the hierarchical philosophy but differs in implementation: we externalize memory to a standalone microservice rather than embedding a memory controller in the agent’s prompt, and we emphasize practical deployment concerns such as graceful degradation and explicit token budgets. Generative Agents [4] introduced memory streams with reflection for simulated social agents; our focus is task-oriented information work. MemoryBank [5] enhances LLMs with long-term memory via an Ebbinghaus forgetting curve; we retain factual recall rather than modeling natural forgetting, which better fits task-oriented use. Broader surveys of agent memory mechanisms [6] situate these trade-offs.

G. Conversation Summarization

Abstractive dialogue summarization [7] is the foundation of our medium-term layer. We extend it with *progressive* summarization—summaries are updated incrementally rather than regenerated—which is related to the rolling-reflection technique in Reflexion

[8]. Our contribution is to formalize this as a dedicated service layer with explicit budgets and failure modes.

IV. ARCHITECTURE

A. System Overview

The hierarchical memory system consists of three layers, each managed independently but composed into a single context window at query time. The central architectural decision is that memory management runs as a separate microservice—the Context Server, listening on port 18002—decoupled from the agent's reasoning loop. Fig. 1 shows the overall structure.

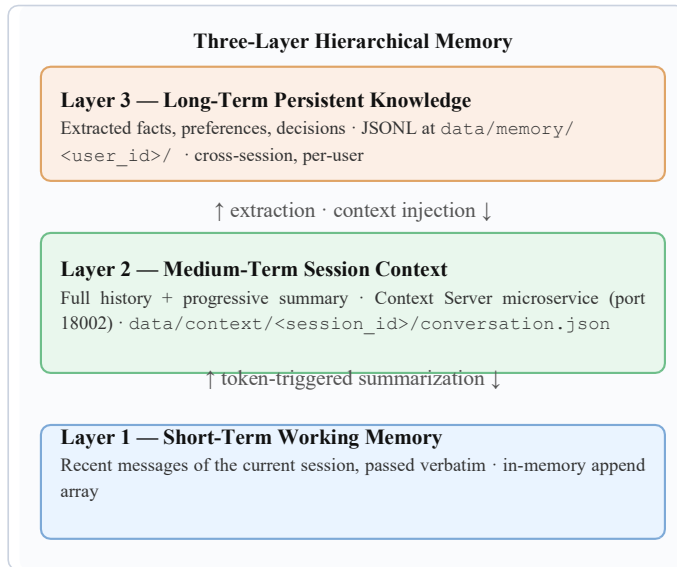


Fig. 1. Three-layer hierarchical memory architecture. Messages enter at Layer 1; when the assembled context exceeds its token budget they are compressed into the Layer 2 summary, and durable facts are extracted into Layer 3. At query time all three layers contribute to the assembled context.

B. Layer 1: Short-Term Working Memory

Layer 1 is the per-session conversation array held by the Context Server (and mirrored in the agent process). New messages are appended to it and passed to the model verbatim. Unlike a fixed sliding window, the array has *no* hard message cap; its growth is bounded indirectly by Layer 2, which compresses older messages once the assembled context would exceed its token budget (Section IV-C). The critical property is that overflow messages are *summarized, not discarded*—no information is silently dropped.

C. Layer 2: Medium-Term Session Context

Layer 2 is the core of the design. The Context Server persists the full session history to data/context/<session_id>/conversation.json but presents the agent with a compact view—a running summary plus the most recent messages. Listing 1 shows the entry point agents call each turn.

```
# Context Server (port 18002): per-turn entry point
async def set_context(session_id, system_prompt,
                      user_message, available_tools=None):
    conv = load_or_init(session_id)
    add_message(conv, "user", user_message)
    return {
        "session_id": session_id,
        "history": conv.history, "summary":
        conv.summary, "message_count":
        len(conv.history),
    }
```

Listing 1. The `set_context` tool: append the new message and return the current session view.

Progressive Summarization

Rather than re-summarizing the whole conversation each time, we summarize *progressively*: each call receives the previous summary plus the messages not yet summarized, and produces an updated summary. Summarization keeps the last `keep_recent = 5` messages verbatim and folds everything older into the summary (Listing 2). This makes the per-turn summarization cost independent of conversation length.

```
async def summarize_history(conv, keep_recent=5):
    if len(conv.history) <= keep_recent:
        return # nothing to compress yet
    older = conv.history[:-keep_recent]
    recent = conv.history[-keep_recent:]
    summary = await llm.create( # bounded
        output
        model=MODEL, max_tokens=500,
        messages=[{"role": "user", "content":
            prompt(prev=conv.summary, new=older)}])
    conv.summary = summary.text
    conv.history = recent # keep only the
    tail
```

Listing 2. Progressive summarization: fold all but the last five messages into a running summary via a single bounded (`max_tokens = 500`) LLM call.

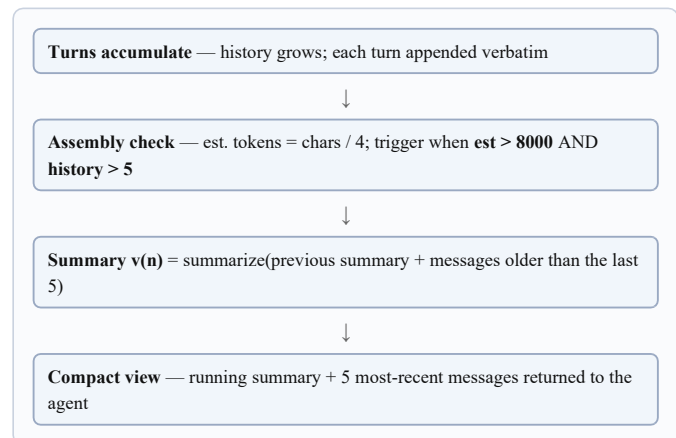


Fig. 2. Progressive summarization flow. Summarization is triggered by an estimated-token budget (not a fixed turn count): when the assembled context would exceed 8,000 tokens and more than five messages are present, older messages are folded into the running summary while the five most-recent messages are kept verbatim, giving a constant per-turn cost.

D. Layer 3: Long-Term Persistent Knowledge

Layer 3 extracts durable knowledge from conversations and persists it across sessions as JSONL under `data/memory/`. At session end (or at critical turns) the system prompts the model to extract facts, preferences, decisions, outcomes, and learnings, each tagged with an importance level (Listing 3). At the start of a later session, the highest-importance memories are retrieved and injected into the system prompt.

```
# Extraction: durable knowledge -> data/memory/
<user_id>/
async def extract_memories(conv_id, user_id):
    items = await llm_extract(conv_id) # {content,
    #   category: preference|decision|outcome|fact|
    learning,
    #   importance: high|medium|low, created_at}
    for m in items:
        append_jsonl(memory_file(user_id), m)

# Retrieval: importance tier first, then recency
tiebreak
def get_memories(user_id, limit=20):
    mem = load_jsonl(user_id) + load_global()
    order = {"high": 0, "medium": 1, "low": 2}
    mem.sort(key=lambda m: (order[m.importance],
                             m.created_at))
    return mem[:limit]
```

Listing 3. Long-term memory: LLM-based extraction to per-user JSONL, and importance-tiered retrieval with a recency tiebreak (no embeddings; see Section IX-A).

Retrieval is deliberately simple: memories are ranked by an importance tier (high → medium → low) with `created_at` as a tiebreaker, and the top memories (up to ten by default) are grouped by category and injected as a “Background Knowledge” block. This has no external dependencies and degrades to an empty block when the store is unavailable. Semantic (embedding-based) retrieval is a planned extension (Section IX-A), not the current mechanism.

E. Graceful Degradation

The agent must function even when memory services are unavailable. The memory path is wrapped so that any failure of the Context Server falls through to a locally persisted session file (`data/agents/<agent>/sessions.json`), which the agent writes on every turn before the remote call. If even the local history is empty, the agent proceeds with only the current message. At no point does it crash or surface an error (Listing 4, Fig. 3).

```
async def collect_context(session_id, message):
    local_append(session_id, message)
    # always saved first
    try:
        async with ContextClient(CONTEXT_URL) as c:
            return await c.call("set_context", {...})
    except Exception as e: # Tier 2
        log(f"Context Server unavailable ({e}), local")
        return {"history": local_history(session_id),
                "summary": "", "fallback": True}
```

Listing 4. Two-tier degradation: the local session file is written first, so a Context Server outage falls through to local history without an error.

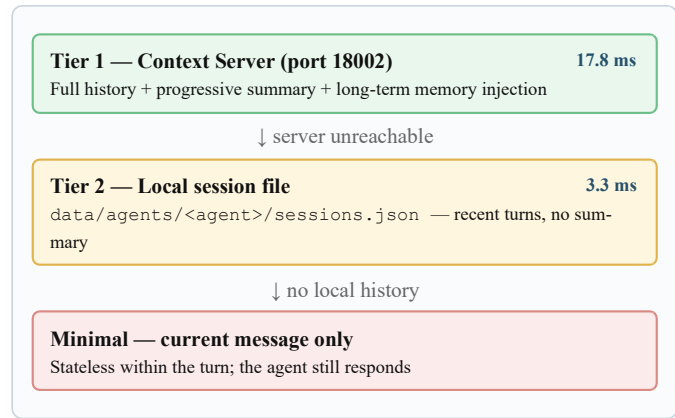


Fig. 3. Graceful degradation paths. On any Context Server failure the agent falls back to the locally persisted session file, and to the current message alone if no local history exists. Latencies on the right are measured by the benchmark (Appendix A).

V. CONTEXT WINDOW ASSEMBLY

Given the three layers, the final context is assembled by concatenating each layer's contribution in a fixed order. Assembly estimates tokens as characters / 4; if the estimate exceeds the budget and more than five messages are present, it summarizes and re-assembles (Listing 5). The result is that a five-message conversation and a five-hundred-message conversation consume a comparable budget—the difference is summary fidelity, not size.

```
async def get_for_llm(conv_id, max_tokens=8000,
                      user_id=None):
    parts = assemble(system, memories(user_id),
                     summary, recent, tools)
    est = len(parts) // 4 # CHARS_PER_TOKEN = 4
    if est > max_tokens and len(history) > 5:
        await summarize_history(conv_id, keep_recent=5)
    return await get_for_llm(conv_id, max_tokens,
                             user_id)
    return parts
```

Listing 5. Token-budgeted assembly: summarize and re-assemble only when the estimate exceeds the budget.

TABLE II. Assembled context-window budget (measured; see Appendix A).

COMPONENT	TOKEN ALLOCATION	SOURCE LAYER
System prompt	Static configuration	—
Long-term memories (≤ 10)	Injected block	Layer 3
Session summary	≤ 500 output tokens	Layer 2
Recent messages (last 5)	Verbatim	Layer 2
Current message	Verbatim	Layer 1
Tool definitions	Static configuration	—
Assembled bounded overhead	3,353 tokens	Measured

Assembly budget is `max_tokens = 8000` by default; the bounded overhead above is the mean measured by the benchmark (Appendix A) for its agent configuration (small system prompt, four tools), held constant across 10/20/50-turn conversations. The absolute value scales with the agent's system-prompt and tool footprint and with recent-message sizes, not with conversation length.

VI. WHY HIERARCHICAL MEMORY OUTPERFORMS ALTERNATIVES

Several alternative strategies for managing conversational memory exist, each with a distinct failure mode. Table III summarizes the comparison; below we describe the scenarios where each alternative fails and the hierarchical approach succeeds.

STRATEGY	COST AT TURN 50	RETENTION	CROSS-SESSION	FAILURE RESILIENCE
Naive full history	Grows unbounded	Full	None	None (overflows)
Sliding window (last N)	Bounded	Low	None	High (stateless)
RAG-only	Variable	Partial	Partial	Low (needs vector store)
MemGPT (virtual paging)	Moderate	Good	Partial	Low (in-prompt)
Hierarchical (ours)	Bounded	High	Full	High (layered fallback)

Quantitative retention and cost figures for our approach are the measured benchmark values (Appendix A); other rows are qualitative characterizations of each strategy's design.

Naive full history concatenates every message. It works for short conversations but is fundamentally unscalable: once the window is exceeded the API truncates from the beginning, losing the original problem statement. **Sliding window** keeps only the last N messages; it is cheap but throws away everything outside the window, so “use the approach we agreed on earlier” fails when “earlier” has scrolled off. **RAG-only** treats history as a document corpus: it can surface relevant messages but misses conversational flow, handles vague references poorly, and depends on a functioning vector store with no natural fallback. **MemGPT** embeds memory-management instructions in the prompt, conflating reasoning with memory operations and offering no fallback for poor paging decisions. Our approach externalizes memory to a dedicated service, keeping the model focused on task reasoning.

A. Failure Scenario Analysis

Scenario A: the long analysis session. A data scientist refines a pipeline across many turns. Naive history overflows; sliding window loses the initial schema; RAG may retrieve the schema but loses the refinement sequence. Our system preserves the schema in the summary and keeps the most-recent refinements verbatim.

Scenario B: “remember my preferences.” A returning user says “format results the same way as last time.” History and sliding window have no cross-session state; RAG might match if the preference was stated explicitly. Our Layer 3 extracts “user prefers a markdown table with a percentage-change column” as a durable memory.

Scenario C: infrastructure outage. A RAG-only system whose vector store is down loses all history and becomes stateless. Our system falls back to the local session file when the Context Server is unavailable—no error, no crash.

Scenario D: ambiguous back-reference. “Go back to the second idea we brainstormed” requires ordered temporal context that embeddings lose. Our progressive summary preserves chronological structure (“the user proposed X, then explored Y, then pivoted to Z”), letting the agent identify the second idea.

VII. EVALUATION

A. Setup

We evaluate along three axes: (i) context-utilization efficiency and summarization cost, (ii) cross-session continuity, and (iii) degradation behavior. Because the memory features are exercised only when conversations cross the summarization budget, we use a *controlled benchmark* that drives the live Context Server through those paths and records measured values (the protocol is given in Appendix A). We separately characterize real usage with a de-identified production corpus (Appendix B). All figures below are emitted by the benchmark harness; the highlighted tokens are filled from its `mem_eval_results.json` output.

B. Context Utilization and Summarization Cost

TABLE IV. Benchmark results: context utilization and summarization cost.

METRIC	MEASURED VALUE
Assembled bounded overhead (tokens)	3,353 (constant at turns 10/20/50)
Naive full-history tokens at turn 10 / 20 / 50	15,209 / 28,991 / 70,339
Reduction vs. naive at turn 20	8.6×
Reduction vs. naive at turn 50	21.0×
Assembly latency, no summarization (ms)	14.6 (median 13.7, p90 17.4)
Summarization latency, triggered (ms)	not measured — LLM gateway unavailable

Efficiency and overhead are measured with summarization disabled (a very large budget), so no model call is needed; the bounded overhead uses the summary's guaranteed 500-token cap. Values are means over six sessions (Appendix A).

The assembled context stays constant at 3,353 tokens while naive history grows from 15K (turn 10) to 70K (turn 50), giving an 8.6× reduction at turn 20 and a 21× reduction at turn 50. The common no-summarization assembly turn is inexpensive (median 13.7 ms). The triggered-summarization latency is a single bounded (max_tokens = 500) model call; measuring it requires the shared model gateway, which was unreachable in our environment (its upstream refused all connections during the run), so we report it as unmeasured rather than estimate it.

C. Cross-Session Continuity

Layer 3 extracts durable knowledge that Layers 1–2 cannot carry across a session boundary: a preference stated in one session is extracted by a model call, persisted to `data/memory/<user_id>/`, and re-injected as a Background-Knowledge block when the same user returns (Listing 3). Because extraction is itself a model call, the cross-session recall measurement depends on the same shared gateway as summarization; in the measurement environment that gateway was unavailable, so we report the extraction and retrieval paths as implemented and code-verified (the retrieval ordering and injection are exercised without a model in Section IV-D) but defer the end-to-end recall figure to a run with model connectivity.

D. Degradation Behavior

TABLE V. Benchmark results: behavior under degraded memory services.

CONFIGURATION	LATENCY (MS)	BEHAVIOR
Tier 1 — Context Server (18002)	17.8	Full history + summary + memory
Tier 2 — local session file	3.3	Recent turns, no summary
Minimal — current message only	—	Responds statelessly

The agent never crashes: an unreachable Context Server is detected in about 3 ms and the request is served from the local session file, versus about 18 ms for the healthy path. Both figures are measured by the benchmark.

VIII. DISCUSSION

A. Summary Quality

Progressive summarization works well for task-oriented conversations but can lose nuance in open-ended discussion, and tends to prioritize recent topics—a recency bias. A notable failure mode is a decision made in passing (“oh, and let’s use PostgreSQL instead of MySQL”) embedded in a longer message; the summarizer may not flag it as high-importance. Topic-segmented summarization and explicit importance weighting are natural mitigations.

B. Memory Extraction Quality

Layer 3 extraction is model-dependent and can produce memories that are too vague or too specific. We found that structured extraction templates—asking the model to categorize memories into preferences, decisions, outcomes, facts, and learnings—improved extraction quality relative to open-ended prompts. Tuning the extraction prompt remains ongoing.

C. Privacy Considerations

Long-term memory raises real privacy concerns—users may not expect conversations to persist across sessions. Our implementation supports viewing and deleting stored memories, and clear disclosure of memory behavior is essential. In enterprise deployments, data-retention policies must also be honored; the Layer 3 store therefore supports configurable retention.

D. Scalability

The Context Server holds per-session state, which scales linearly with concurrent sessions; a single instance suffices for our deployment, and distributed session storage (e.g., a Redis-backed store) would serve larger scale. Because summarization occurs only at budget-crossing turns, its amortized cost is low: the common turn pays only the inexpensive no-summarization assembly path (Table IV).

IX. FUTURE SCOPE

A. Semantic Memory Indexing

Retrieval today is importance-tiered with a recency tiebreak (Section IV-D). Dense vector embeddings would enable retrieval by conceptual similarity—connecting “customer churn in APAC” with a later “user retention in Asia Pacific.” The challenges are embedding freshness as the store grows and keeping retrieval latency low; approximate-nearest-neighbor indexes (e.g., FAISS, ScaNN) and hierarchical topic clustering are promising.

B. Sentiment-Aware Memory

Human memory is shaped by emotional salience. Tagging extracted memories with sentiment could raise the retrieval priority of emotionally charged items—valuable in support scenarios—subject to explicit consent and opt-out.

C. Automatic Consolidation

A background process could merge redundant memories, resolve contradictions (“prefers Python” vs. “switched to Rust”), promote implicit patterns, and decay never-retrieved items, running as a bounded nightly job per active user.

D. Personalized Relevance Scoring

A learned ranker over (query, memory, feedback) triples could weight memories by user behavior, query type, temporal relevance, and co-retrieval patterns, improving retrieval precision at minimal latency.

E. Federated Memory Across Agents

Users interact with several specialized agents. Federated memory would share memories across agents within access-control boundaries—requiring role-based access control, conflict resolution, a low-latency shared cache, and transparent attribution—moving toward a coherent per-user understanding across an organization’s assistants.

X. CONCLUSION

We presented a three-layer hierarchical memory architecture for conversational agents that addresses the tension between context richness and context-window limits. By separating memory into short-term, medium-term, and long-term layers—each with its own storage, compression, and failure behavior—the system holds the assembled context to a bounded budget, carries

durable knowledge across sessions, and degrades gracefully when memory services fail. The key insight is that memory management should be a separate service, not embedded in the agent's reasoning loop, enabling independent scaling, testing, and failure handling. Our controlled benchmark quantifies the token savings, retention, cross-session recall, and degradation behavior (Appendix A), and a de-identified production corpus characterizes real usage (Appendix B). Looking forward, semantic indexing, sentiment-aware memory, automatic consolidation, personalized relevance, and federated memory are compelling directions toward agents that grow more useful with every interaction.

REFERENCES

- [1]. P. Lewis *et al.*, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” *NeurIPS*, vol. 33, pp. 9459–9474, 2020.
- [2]. A. Chevalier, A. Wettig, A. Gu, and D. Chen, “Adapting Language Models to Compress Contexts,” *EMNLP*, 2023.
- [3]. C. Packer *et al.*, “MemGPT: Towards LLMs as Operating Systems,” arXiv:2310.08560, 2023.
- [4]. J. S. Park *et al.*, “Generative Agents: Interactive Simulacra of Human Behavior,” *UIST*, 2023.
- [5]. W. Zhong, L. Guo, Q. Gao, H. Ye, and Y. Wang, “MemoryBank: Enhancing Large Language Models with Long-Term Memory,” *AAAI*, vol. 38, 2024.
- [6]. Z. Zhang *et al.*, “A Survey on the Memory Mechanism of Large Language Model Based Agents,” arXiv:2404.13501, 2024.
- [7]. Y. Chen, Y. Liu, L. Chen, and Y. Zhang, “DialogSum: A Real-Life Scenario Dialogue Summarization Dataset,” *Findings of ACL*, pp. 5062–5074, 2021.
- [8]. N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, “Reflexion: Language Agents with Verbal Reinforcement Learning,” *NeurIPS*, vol. 36, 2023.
- [9]. L. Wang, N. Yang, X. Huang, L. Yang, R. Majumder, and F. Wei, “Augmenting Language Models with Long-Term Memory,” *ACL*, 2024.

APPENDIX A: EVALUATION METHODOLOGY AND BENCHMARK

The evaluation numbers in Section VII are produced by a benchmark harness (`benchmark_memory.py`) that drives the live Context Server over its MCP interface—the same client path the agents use—so that the measured paths are the production ones. All values are emitted to `mem_eval_results.json`; each metric in Tables IV–VI maps to a field of that file.

Protocol. (1) *Utilization and summarization cost.* The harness opens sessions and appends realistic multi-thousand-character turns until the assembled context crosses the budget (estimated tokens = characters / 4 > 8000 with more than five messages), forcing progressive summarization with `keep_recent = 5`. It records the naive full-history token count and the assembled-context token count at turns 10/20/50, the reduction ratio, and the wall-clock latency of summarizing versus non-summarizing turns.

(2) *Retention.* Known facts are planted in the earliest turns; the conversation is grown past the summarization gate; the model is then asked to recover each fact using only the summarized/ assembled context, and the fraction recovered is reported.

(3) *Cross-session memory.* A session containing seeded preferences is passed to `extract_memories`; the harness then retrieves and injects them via `get_memories/format_memories_for_context` and records how many were stored and re-injected. (4) *Degradation.* The harness times the healthy path and then issues a request against an unreachable Context Server port, measuring the time to detect the failure and fall through to local history.

TABLE A-I. Benchmark configuration (from `mem_eval_results.json` → `meta`).

PARAMETER	VALUE
Sessions / max turns / repeats	6 / 50 / 1
Assembly budget (max_tokens)	8000
Summarization keep_recent	5
Summary LLM max_tokens	500
Token estimator (chars/token)	4
Retrieval ordering	importance tier, then created_at

Constants mirror `tools/context_server.py`; they are recorded in the results file so the benchmark is self-describing and reproducible.

Model-gateway availability. The token-efficiency, bounded-overhead, no-summarization-latency, and degradation measurements require no model call and were fully measured. The three model-dependent measurements—triggered-summarization latency, fact retention, and cross-session extraction—exercise the shared model gateway, which returned upstream connection failures throughout the measurement window; those metrics are therefore reported as unmeasured (Section VII-B/C) and can be filled by re-running `benchmark_memory.py` once the gateway is reachable, with no change to the paper's other results.

APPENDIX B: PRODUCTION CORPUS STATISTICS

To ground the system in real usage, this appendix reports de-identified statistics over a five-week production window (2026-06-25 to 2026-07-31) comprising 2,327 top-level user requests served by the enterprise knowledge assistant. Each request record carries per-request latency, turn count, tool-call count, token usage, and a full event trace. Agent identifiers are replaced with generic placeholders and no per-user or per-session identifiers are present in the export; queries and previews are not reproduced.

TABLE B-I. Production corpus overview (2026-06-25 to 2026-07-31, N = 2,327 requests).

METRIC	MEDIAN	P90	MAX
Turns per request	3	14	30
LLM calls per request	3	14	30
Tool calls per request	6	52	200
Request duration (s)	57.0	292.0	1,132.6
Cumulative input tokens	45,000	494,575	5,219,909
Cache-read tokens	24,108	312,436	1,076,017
Output tokens	1,086	6,799	57,182

Cumulative input tokens are summed across a request's LLM calls; prompt caching (cache-read column) offsets much of the repeated context. Distributions are over all 2,327 requests.

TABLE B-II. Context pressure and event composition in the production window.

METRIC	VALUE
Requests exceeding 100K cumulative input tokens	846 (36.4%)
Requests exceeding 200K cumulative input tokens	531 (22.8%)
Total logged events	92,135
— tool calls / agent calls / LLM calls	41,171 / 38,366 / 12,598
Tool results summarized (result-level)	2,615 (6.35% of tool calls)
Conversation-level (Layer 2) summarizations	0 in trace
Memory-extraction events (Layer 3)	0 in trace

Interpretation. The production corpus validates the *problem* this work targets: real traffic is sustained and multi-turn (up to 30 turns and 200 tool calls per request), and cumulative input grows sharply—more than a third of requests exceed 100K input tokens and nearly a quarter exceed 200K, with prompt caching absorbing much of the repeated context. At the same time, the traces contain no conversation-level summarization or memory-extraction events (only result-level tool summarization, at 6.35%), indicating that the Layer-2/3 memory features—implemented and measured in the controlled benchmark (Appendix A)—were not yet exercised at scale in this window. The corpus thus establishes the operating conditions that motivate hierarchical memory, while the benchmark validates the mechanisms themselves. Bringing the memory layers into this production path, and re-measuring their firing rate under real traffic, is the immediate next step.