MemoryArchitect: Adaptive Policy-Controlled Memory Governance for RAG and LLM Systems

I. Introduction

1. Aim and Motivation

Project Aim: *MemoryArchitect* is proposed as a memory governance layer for agentic AI systems, particularly large language model (LLM) based agents. Its purpose is to manage what an AI agent remembers, forgets, and retrieves according to explicit policies. By doing so, the agent can retain important information over prolonged multi-session interactions while minimizing unnecessary token usage, avoiding knowledge confusion (hallucinations or contradictions), and respecting privacy constraints (through controlled forgetting of sensitive data).

Motivation: Long-term memory management has emerged as a critical challenge for AI agents that engage in extended conversations or tasks. Modern LLMs are *stateless* beyond their fixed context window – they "forget everything outside their prompt". Once an interaction falls out of the model's context (often a few thousand tokens), it is gone, forcing repetition or leading to loss of relevant details. Simply increasing the context window (e.g. GPT-4's 128K context or more) only delays the problem and incurs steep computational costs [7]. Even architectures that enable *streaming* long contexts (up to millions of tokens) face efficiency and memory issues[26]. Retrieval-Augmented Generation (RAG) offers a partial remedy by fetching external information on demand, but if done naïvely (retrieving a fixed number of passages regardless of necessity or relevance), it can introduce irrelevant context and confuse the model[11]. Poor memory management can thus manifest in several ways:

- **Forgetting Important Facts:** An agent without persistent memory will eventually forget earlier user-provided facts or context once those fall outside the window. For example, a user might mention a dietary restriction in Session 1, and the agent, having forgotten it by Session 5, gives a contradictory suggestion. Empirical studies show that outdated or irrelevant stored experiences can negatively influence current tasks, causing error propagation and inconsistent behavior[27].
- **Hallucinations and Confabulations:** If the agent cannot recall a relevant fact because it was not retrieved or has been dropped, it may guess or *hallucinate* an answer. Large models tend to produce an answer rather than admit not knowing, so missing context can yield plausible-sounding but incorrect information. Ensuring vital facts are remembered and fetched at query time reduces the chance for such hallucinations.[27]
- **Overloaded Contexts:** If an agent tries to carry too much history, it risks *context overflow*. Every additional token in the prompt increases latency and cost, and attention on very old tokens may degrade. Without compaction, an ever-growing history is infeasible to include. There is a need to **summarize or forget** less important details to keep context size bounded. Indeed, even state-of-the-art models struggle with truly long-range coherence – recent benchmarks indicate that models like GPT-4 fail on episodic recall when conversations span even 10k–100k tokens[6][9].
- **Privacy and Retention Risks:** Long-term logs of interactions can accumulate sensitive personal data. Without governance, an AI might retain personally identifiable information indefinitely, raising compliance issues. Data protection principles (such as GDPR's "right to be forgotten") demand mechanisms for data minimization and deletion. In practice, most current LLM memory systems do not implement timely expiration or selective retention controls. This gap means an agent could unintentionally expose or misuse old sensitive data. Ensuring that an agent's memory complies with privacy policies (e.g., auto-deleting certain data after X days) is therefore essential.

In summary, there is a pressing need for a smarter memory layer that **keeps an AI's knowledge relevant, compact, and compliant** over time. The MemoryArchitect project addresses this need by introducing a policy-controlled memory system that actively decides what to remember, for how long, and what to forget, all while retrieving information intelligently to support the agent's current task.

B. Background and Context

Agentic AI and Memory: *Agentic AI* refers to AI systems (often powered by LLMs) that exhibit autonomous, goal-driven behavior – they can perceive inputs, take actions (e.g. calling tools or APIs), and adapt based on feedback. Examples include conversational assistants, autonomous researchers, and multi-modal interactive agents. Memory is a foundational component for such agents, enabling them to carry information across steps and across sessions. In humans, long-term memory allows us to maintain continuity in dialogue and accumulate knowledge; analogously, AI agents require persistent memory to avoid "starting from scratch" each session.

Modern LLM-based agents typically implement memory in one of two ways: (1) **Extended Context Windows**, wherein a compressed summary of the conversation or a sliding window of recent messages is included in the prompt, and (2) **External Knowledge Stores**, such as vector databases, where past interactions or facts are embedded and stored, to be retrieved when needed[17]. The latter approach is commonly known as retrieval-augmented generation (RAG), where an external memory (documents, embeddings, etc.) is queried to provide the LLM with relevant information beyond its prompt. RAG has proven effective in boosting factual accuracy and letting models use information not contained in their parameters. However, using an external memory raises new questions: how to decide which pieces of knowledge to add to memory, how to organize and index them, and how to remove or update them over time.

Memory Mechanisms in LLM Agents: Conceptually, an LLM agent's memory module has three parts: **(1) Memory Writing**, which captures and stores new information (this could be verbatim transcripts of interactions, or distilled summaries/extractions of key facts); **(2) Memory Management**, which organizes and maintains the stored data (merging similar entries, compressing older ones, deleting low-value items, etc. to prevent overload); and **(3) Memory Reading/Retrieval**, which fetches relevant pieces of memory when the agent needs context for generating a response or making a decision. For example, if a user asks a follow-up question referring to something from last week, the agent's memory system should retrieve the pertinent detail from the long-term store and present it to the LLM. A well-designed memory ensures important details are retained and accessible when needed, while unneeded data is pruned to save space and avoid distraction.

Common Memory Challenges: Despite various implementations, current long-term memory solutions for LLMs face several challenges (as also noted in prior research and industry analyses[3], [22]:

- **Token Limitations:** LLMs have a fixed context length, so appending the entire dialogue history will eventually exceed the limit or incur prohibitive cost. Important information can get pushed out unless summarized or stored externally. Even very long-context models (100k tokens or more) see degraded performance on very old tokens and are not a panacea[26].
- **Knowledge Dilution & Inconsistency:** If the memory is not properly prioritized, crucial facts can be lost among trivial details. The agent might then answer incorrectly or inconsistently, especially in dialogues with many semantically similar statements (risking confusion). Without persistent memory of key facts, an agent may contradict itself (e.g., forgetting a user's name or preferences mentioned earlier).[27], [18]
- **Uncontrolled Growth:** An unmanaged memory store can grow without bound as the conversation continues. This not only consumes more storage and increases retrieval latency but also may start surfacing stale or irrelevant information if not cleaned up.[17], [4]
- **Privacy Concerns:** Storing entire conversation logs indefinitely is problematic when those logs contain personal or sensitive data. Agents need mechanisms for *data retention policies* – for instance, to automatically purge or anonymize information after it's no longer needed. Features like time-to-live (TTL) for memory entries and explicit deletion rules are seldom built-in today but are necessary for compliance and for user trust (e.g., forgetting a user's private story after the task is done)[5], [8].

These challenges motivate the need for a memory governance layer that can introduce structure and policy to the agent's use of memory. Rather than treating memory as an ever-growing transcript or a black-box vector store, MemoryArchitect treats it as a **managed knowledge base** with rules about what to keep, what to retrieve, and what to discard.

C. Related Work and Limitations

A number of research efforts have recently explored long-term memory for LLM-based agents. Each offers useful ideas but also has limitations that our approach aims to overcome. We briefly review some representative systems and findings:

1. **MemGPT (Packer et al., 2023):** MemGPT introduced the concept of using an LLM as an *Operating System* for memory, with a **multi-tier memory hierarchy**[17]. In MemGPT, the main LLM can call a smaller "memory manager" LLM (the OS) that decides when to swap information between the active context (analogous to RAM) and a long-term vector store (analogous to disk). This design let the agent "remember, reflect, and evolve" over long interactions beyond the normal context window. However, MemGPT's approach relies on the LLM itself to manage memory via special tool calls, which is powerful but complex and computationally expensive. It focused on extending context and maintaining coherence, rather than on policy or selective forgetting – MemGPT did not implement mechanisms like timed deletion or privacy tagging. In short, MemGPT demonstrated the potential of LLM-managed memory scheduling, but it **lacked a governance layer** for applying domain-specific retention rules (no concept of expiring data or auditing what was removed, for instance).
2. **Mem0 (Chhikara et al., 2025):** Mem0 is a system that achieved state-of-the-art results on long-dialogue benchmarks by using a *memory-centric architecture*. An LLM in Mem0 distills each dialogue turn into key factual snippets and stores them in a persistent knowledge base[24]. The memory is updated in two phases: extraction of new facts from the latest interaction, and an integration phase that merges or deletes facts to keep the memory consistent. Mem0 showed that this approach can significantly improve accuracy on long conversation tasks (e.g., ~26% higher accuracy on the LoCoMo benchmark) while using far fewer tokens than a full history[4]. However, Mem0's memory management is largely *LLM-driven* and optimized for accuracy – it doesn't expose controls for user-defined policies. For example, Mem0 has no notion of time-based expiration; it retains extracted facts indefinitely unless they are contradicted later. It also doesn't distinguish sensitive information – everything extracted is stored in the same way. Thus, Mem0 excels at **what to remember** for task performance, but lacks configurability in **how long to remember** or **when to forget** from a governance or privacy standpoint.
3. **Reflective Memory Management (RMM, Shinn et al., 2023):** RMM (often referenced as "In Prospect and Retrospect") from Google Research introduced a framework for an agent to **reflect on its memory usage** and refine it. The system logs which retrieved memories were actually useful to the agent's response and which led to errors, and uses this feedback to adjust the memory store[24]. For instance, if a retrieved memory was irrelevant or caused a contradiction in the agent's answer, the system can down-rank or remove that memory entry. Over time, this reflective loop helps *mitigate error propagation* by pruning misleading content and reinforcing useful facts. RMM demonstrated improved long-term dialog consistency by combining selective addition of new information with deletion of harmful or outdated information[27]. This aligns with the idea that an agent should not blindly accumulate experiences, but rather curate them – a principle we adopt in MemoryArchitect's reflection and adaptation component. A limitation of RMM, however, is that it mainly focuses on the *feedback loop* for accuracy, and less on predefined policy rules or guarantees (e.g., it doesn't explicitly ensure data privacy rules are followed, unless those happen to align with the agent's error feedback).
4. **A-MEM (Xu et al., 2025):** A-MEM (Agentic Memory) is a research prototype emphasizing *knowledge organization*. Instead of storing memories as independent text chunks only, A-MEM organizes them into an interconnected graph or network of semantic nodes[28]. This approach, inspired by the Zettelkasten method of notetaking, means that when a new memory is added, the system attaches metadata (context descriptors, tags) and links it to related existing memories. Over time, the memory forms a graph of concepts and facts, which can help with multi-hop retrieval or inference. A-MEM's structured approach can improve an agent's ability to reason over its past knowledge. However, **by itself A-MEM does not enforce forgetting or prioritization policies** – once information is in the memory network, there is no built-in mechanism for aging out old nodes or focusing on the most relevant subgraph. It addresses *how to represent and connect* memories, more than *when to delete or summarize* them. Thus, A-MEM could potentially be combined with a policy layer like MemoryArchitect to add lifecycle

management to its rich knowledge graphs.

5. **MemoryBank (Zhong et al., 2024):** MemoryBank is an approach that explicitly incorporates ideas from cognitive psychology – notably the **Ebbinghaus forgetting curve** – into AI memory. In MemoryBank, each memory item's "strength" decays over time if not used, simulating human forgetting[31]. Frequently accessed memories remain strong (and thus persist in the agent's active recall), whereas seldom-used ones gradually fade and eventually drop out. This ensures the agent's memory doesn't keep growing indefinitely with stale data – old memories naturally expire unless reinforced. MemoryBank's heuristic proved effective in improving long-term assistant consistency, and it even demonstrated integration with both closed-source and open-source LLMs in an AI companion setting. The limitation is that a purely frequency-based decay can remove information that might be important but hasn't been referenced recently (for example, a critical fact that is infrequently needed could disappear right when it's needed the most)[2]. Also, MemoryBank, like others, did not incorporate policy exceptions (e.g., "never forget *this* item") or user-driven preferences about what to retain. It's a one-size-fits-all forgetting mechanism. MemoryArchitect draws inspiration from the idea of *gradual forgetting* (we implement a reflection score that decays without use, akin to MemoryBank's decay schedule, and thus aligned with human memory models[31], [2], [[23]), but we combine it with explicit rules and thresholds to ensure important data is kept as long as needed.

Summary of Gaps: Across these works, no single solution offers a comprehensive, *policy-driven* memory governance system. Techniques like MemGPT and RMM treat memory as a dynamic resource to extend context or improve consistency, but they don't give developers or users transparent control over retention policies (e.g., automatic expiry or privacy filters). High-performing systems like Mem0 and MemoryBank optimize what to store for task success and efficiency, but they lack mechanisms for *customizable forgetting rules*, sensitive data handling, or auditability. For example, none of the above systems provide an easy way to answer: *"What does the agent currently remember about topic X, and why?"* or *"Show me all data older than 30 days that was deleted."*

Our Approach: *MemoryArchitect* is designed to fill this gap by marrying the strengths of prior approaches (efficient RAG, reflection/feedback loops, structured memory, etc.) with an explicit **policy layer** for governance[16]. We treat memory management not just as a technical caching problem, but as a set of configurable policies—much like an IT system's data retention policy—enforced by the memory controller. MemoryArchitect acts as an intermediary between the agent and its memory store, intercepting memory read/write operations to apply rules for lifespan, relevance, and privacy. In the following sections, we detail the architecture of MemoryArchitect, its core components and algorithms, and how it adapts over time. We then illustrate the system's behavior with an extended example and discuss implementation considerations and evaluation plans.

II. System Design: MemoryArchitect Architecture

A. System Overview

At a high level, MemoryArchitect consists of **five core components** working in tandem to provide an intelligent, self-managing memory layer for LLM agents. These components align with the memory operations discussed earlier (writing, management, retrieval) and integrate policy decisions at each step. The five main subsystems are:

- **Memory Chunk Structure:** A standardized representation for each unit of knowledge or history in the memory store, enriched with metadata that drives policy-based decisions.[18], [30]
- **Intelligent Retrieval Pipeline:** A mechanism to fetch the most relevant memory chunks for a given query by filtering and ranking based on both semantic similarity and policy-driven priorities (importance, freshness, etc.), including enforcement of expirations.[30], [1]
- **Reflection and Adaptation Loop:** A feedback process that updates the memory after each use, reinforcing useful items (increasing their score or priority) and pruning or downgrading items that are stale or not useful. This is how the system "learns" which memories matter and should be retained.[18], [1]

- **Dynamic Context Management:** A strategy to manage the limited context window of the LLM by deciding which retrieved or stored information to include in the prompt. It ensures that the highest-priority content fits in the prompt, using tactics like eviction of low-value context and on-the-fly summarization to compress information.[11], [8]
- **Server Load and Memory Optimization:** Background processes and storage strategies to keep the memory store efficient in the long run – handling the expiration of outdated entries, compression of data, multi-tier storage, and other optimizations to prevent unbounded growth or slowdowns.[8], [25], [14]

These components together enable **adaptive memory governance**. MemoryArchitect continuously balances competing objectives: retaining critical long-term knowledge vs. forgetting clutter, leveraging rich context vs. minimizing token usage, and personalizing memory for each user/session vs. ensuring privacy isolation. Each component is implemented with policy hooks so that behaviors can be tuned per application requirements. For example, one can configure how aggressive the TTL expirations are, or what counts as a "high reflection score" threshold for retention.

When the agent needs to retrieve context, MemoryArchitect's retrieval pipeline controls what memory is searched, and which results are returned to the agent. Throughout, the policy engine monitors and enforces rules (like "ephemeral data expires after 24 hours" or "never retrieve private notes unless explicitly asked"). An **audit log** records key events (additions, evictions, retrievals) for transparency.[29]

In the following subsections, we describe each component in detail, including data structures, algorithms, and how they relate to prior art. We also indicate which policies influence each part of the system. The components are presented in a logical order (from how data is stored, to how it's retrieved, updated, and trimmed, and finally how it is presented to the LLM).

B. Memory Chunk Structure

All information in the MemoryArchitect system is stored as **memory chunks** in a vector database (such as ChromaDB or similar). A chunk is the atomic unit of memory – it could represent a piece of factual knowledge, a user utterance, an assistant response, an extracted summary of a past conversation, etc. Crucially, each chunk is not just raw text; it carries a rich set of **metadata** [21]that describes its origin, usage, and governance attributes. This metadata is pivotal for policy enforcement and intelligent retrieval. The standard metadata fields for a memory chunk include:

- **Source Type:** The origin or format of the content. For example: `doc` (a document or knowledge base snippet), `chat_user` (a user's utterance in a dialogue), `chat_assistant` (the assistant's reply), `api_result` (data retrieved from an API call), etc. Knowing the source type can inform retrieval strategies. For instance, when answering a factual question, the system might prioritize `doc` chunks (verified knowledge) over raw `chat` history. Source type can also correlate with reliability or longevity (e.g., `doc` might be considered more canonical than a user's speculative chat message).
- **Policy Classification:** A label indicating the *intended lifespan and importance* of the chunk. We define policy classes such as: `canonical` (core facts or ground truth information that rarely changes), `factual` (useful facts or knowledge specific to the user's context), `intent-bound` (information tied to the user's current goal or query, likely short-term relevant), `ephemeral` (short-lived context or intermediate information), and `private` (sensitive user-specific data that might require special handling). This classification guides how the chunk is treated. For example, **canonical** chunks are kept indefinitely and shared across sessions when relevant (they form a knowledge base the agent can always rely on), whereas **ephemeral** chunks are meant to expire quickly (they have value only in the immediate conversation). Prior work has noted that not all information is equally valuable and longevity can be tiered[[5]. High-priority data (e.g. a key fact about the user's profile or a critical decision made) should be preserved, while low-priority data (temporary details unlikely to be needed again) can be dropped or summarized soon after use[5].
- **Time-to-Live (TTL) / Expiry Timestamp:** A metadata field that specifies how long the chunk should persist. This can be a duration (e.g. 24 hours for ephemeral data) or an absolute timestamp after which the chunk is considered expired. TTL provides an automatic aging mechanism to enforce

retention policies. For example, a chunk classified as `ephemeral` might be given a TTL of a few hours or a day, ensuring it is purged once it's no longer fresh[25]. A `private` chunk containing sensitive data might have a short TTL by policy (e.g. delete after one use or one day). In contrast, `canonical` chunks might have no expiry (they remain until explicitly removed or updated). Automatic expiration is critical for containing memory bloat and for compliance – it mimics the human tendency to eventually forget details that aren't reinforced[19], [13]. By the time an expired chunk reaches its TTL, the system will either delete it or archive it (move to cold storage) so it no longer appears in active retrievals. Enforcing TTL in retrieval means any chunk past its expiry is ignored, preventing outdated or policy-violating data from influencing answers.

- **Reflection Score:** A dynamic **usefulness metric** (e.g., scaled 0 to 100) that represents how valuable or frequently needed this chunk has been in the past. This score is initially set to a default (e.g., 50 for new chunks) and then updated by the Reflection & Adaptation Loop (Component C) after each interaction. If a chunk is retrieved and actually contributes to the assistant's answer (detected via usage analysis), its reflection score increases. If it is retrieved but not used, or not retrieved at all for a long time, the score may decrease. A high reflection score means the chunk has proved its worth repeatedly (the system "remembers" that this memory was helpful), whereas a low score means the chunk has little demonstrated utility or might be obsolete. This metric enables the system to **reinforce or forget memories in a human-like way** – important memories stay strong, unimportant ones fade[27] [18]. This idea is analogous to the *"importance"* or *"salience"* values assigned in cognitive-inspired agent memories (e.g., Generative Agents assign an importance score to each memory based on its significance). Our reflection score serves as an internal heuristic for memory priority: for example, when the memory store grows too large, chunks below a certain score threshold might be pruned first, and retrieval ranking considers this score alongside similarity. It also interacts with TTL (e.g., an important chunk might get its TTL extended or ignored).
- **User Intent Tags:** Semantic tags or keywords indicating what topics or intents the chunk is related to. These tags can be derived automatically (e.g., via an LLM that labels the content) or provided contextually[20]. For instance, a chunk about a project deadline could carry tags like `project_management, deadline`, or a support chat chunk about a refund policy might have `customer_service, refunds`. Intent tags allow context-aware filtering: when a new user query comes in, the system can identify the query's topic or intent (such as "gardening" or "travel planning") and then restrict retrieval to chunks with matching tags. This prevents crosstalk between unrelated topics and improves precision. It effectively groups memories by context, so that when the conversation shifts, irrelevant past memories (with different tags) are less likely to be retrievel. Intent tags serve as a high-level index into the memory, complementary to vector similarity. They are especially useful in multi-session scenarios to avoid pulling in irrelevant old context from past sessions that have no bearing on the current discussion.
- **Session/User Metadata:** In a multi-user or multi-session environment, each chunk can also be tagged with the `user_id` it belongs to and optionally a `session_id`[21]. This ensures **data isolation** between users and sessions. By filtering on user_id, MemoryArchitect guarantees that one user's private memories never leak into another user's query results. Essentially, each user gets a personal memory store namespace. Even for the same user, if needed, session identifiers can separate distinct conversations (unless we intentionally want cross-session recall). Our prototype assumes a per-user memory space (so all sessions of a user share memory, enabling continuity), but sensitive or context-specific info can be tagged as session-bound if we want it forgotten after the session[8]. This design follows best practices for multi-tenancy in vector databases.

All these metadata fields are stored alongside the chunk's vector embedding in the database. Modern vector stores natively support metadata and allow filtering queries based on metadata conditions[25]. MemoryArchitect leverages this to perform *pre-retrieval pruning* (e.g., only search within `user_id = X` and `intent_tag = Y` and non-expired chunks). The metadata schema is crucial for implementing policies: for example, the TTL is a policy for expiration, the classification is a policy for how to handle data (ephemeral vs canonical), and the reflection score embodies an adaptive policy of reinforcing useful data. By structuring each memory entry this way, we transform the memory from an amorphous log of text into a **managed knowledge base** where each entry carries context about its own governance.

Rationale and Related Work: This approach draws on ideas from databases and cognitive science. In databases, adding indices and metadata enables fine-grained queries and access control – here we index memories by attributes like source, topic, and user. In cognitive terms, one can liken this to how human memory attributes context to memories (time, place, relevance) and tends to forget the "where and when" for trivial details over time. Systems like generative agent simulations have used metadata tagging (e.g., importance, timestamps) to decide what an agent remembers or reflects on[18]. By designing our memory chunk structure with rich descriptors, we set the stage for the subsequent components (retrieval, reflection) to operate efficiently and transparently.

C. Intelligent Retrieval Pipeline

When the agent (LLM) needs context to answer a user query or perform a task, MemoryArchitect's retrieval component kicks in to fetch the most relevant memory chunks. The goal is to provide the LLM with *just the information it needs* – no more (to save tokens and avoid confusion) and no less (to ensure important facts are on hand). Our retrieval pipeline is **intelligent** in that it doesn't rely solely on raw vector similarity. It consists of multiple stages that incorporate metadata filtering and policy rules before and after the similarity search. The pipeline can be summarized in four main stages[17], [8], [28], [14], [19]:

1. **Metadata Pre-Filtering:** Before any expensive embedding similarity search, MemoryArchitect first narrows down the candidate pool using metadata criteria. This drastically reduces noise and computational load by focusing only on the subset of memory that could be relevant *a priori*. Common filters include:
2. **User scope:** Only consider chunks belonging to the active user (e.g., `user_id = current_user`). This ensures multi-user isolation (each user's queries only retrieve their own memories)[[20]]. If the system has global chunks (like shared facts available to all users), those can be included separately, but no cross-user personal data is allowed.
3. **Session or Topic scope:** If the use-case is such that each session is distinct, we might filter by `session_id`. In our design we assume cross-session recall is allowed (to enable continuity), so instead we filter by **intent tags**. We infer the user's current query topic or intent (this can be done via a simple classification or by examining keywords). For example, if the user's current question is about gardening, we apply a filter like `intent_tags CONTAINS "gardening"[[21]]`. This means we only search within memory chunks relevant to gardening, ignoring chunks tagged with unrelated topics (e.g., past conversations about travel). This yields a targeted search and avoids retrieval of irrelevant context.
4. **Recency or Time filters:** We may choose to filter out very old chunks except if they are of certain types. For instance, we might ignore ephemeral chunks older than a week (they likely expired), or only consider chunks within the last N interactions for certain query types. Recency filtering can work in tandem with the reflection score (e.g., ignore chunks that haven't been accessed in a very long time and also have low scores).
5. **Source or Policy filters:** Depending on the query, we might restrict sources. If the user query is asking for a factual definition, we might only search `canonical` or `factual` chunks (which come from knowledge base documents) and not search `chat` history at all. Conversely, if the user asks "What did I tell you last week about X?", we might specifically search their `chat_user` chunks from the relevant time frame.

By combining these filters, we transform the retrieval into a *focused query*. For example: *Retrieve from vector store where* `user_id = Alice` *AND* `intent_tag IN {gardening, general}` *AND* `policy_class != ephemeral`*. Such filtering is supported by vector databases via metadata conditions[10]. This step implements the policy that one user's data should remain isolated and that only relevant topical memories should be considered, improving both privacy and precision. Research on structured RAG has shown that segmenting the vector search space (e.g., by data type or topic) can significantly enhance retrieval accuracy[22] – our design leverages this insight by using metadata as a "pre-search index."

1. **Hybrid Similarity Search and Ranking:** After filtering, the system performs a semantic similarity search on the embeddings of the remaining candidate chunks. We use the user's current query (or the LLM's current task description) to generate an embedding and ask the vector store for the top-k nearest

chunks. However, instead of trusting the raw similarity scores blindly, MemoryArchitect immediately adjusts the ranking using policy-informed weighting. Specifically, we compute a composite relevance score for each retrieved chunk as a function of:

2. **Semantic similarity:** how well the chunk's content matches the query content.
3. **Reflection score (usefulness):** how "important" that chunk has proved over time.
4. **Policy priority:** based on class – e.g., if something is `canonical` or `critical`, we might boost it; if something is `ephemeral`, we might down-weight it unless the query context strongly calls for ephemeral context.

We can imagine a weighted scoring function such as: Score(chunk) = w_s.sim (query, chunk) + w_r.f{reflection_score} + w_c.policyBoost(class) where f might normalize the reflection score to [0,1] and policyBoost gives a small bonus or penalty for certain classes. For example, we might set \$$w_s$ = 0.7, w_r = 0.2, w_c = 0.1 in each configuration. This means similarity drives most of the ranking, but an item with a much higher reflection score can outrank a slightly more similar item because we trust it more (it has been useful before). Conversely, an ephemeral item with a low score might be ranked below a factual item even if the ephemeral one had equal similarity, because the factual chunk is deemed more reliable. This hybrid ranking approach ensures that **important, proven memories surface to the top**, addressing the experience-following behavior observed in agents[27] (the agent's outputs follow what's retrieved; by retrieving high-quality, high-value items, we steer the agent towards better outputs).

The use of reflection score in retrieval is akin to a learning-to-rank system that personalizes results based on past usefulness – like how search engines use click-through data to promote useful pages. In academic terms, our approach relates to *iterative/feedback-enhanced retrieval*: e.g., the SELF-RAG framework, which has the model critique retrieved passages and adapt its retrieval behavior, effectively improves relevance[[1]]. Our system does this adaptation over multiple turns via reflection scores rather than in one query, but the spirit is similar: retrieval is not static cosine similarity; it evolves as the system learns which memory items truly matter.

1. **TTL Enforcement and Staleness Check:** After obtaining a ranked list of candidate chunks, the pipeline performs a final policy check: it filters out any chunks that should **not be used due to expiration or other policy flags**. Even though we might have filtered out most expired items in the metadata step, it's possible that a chunk just on the verge of expiry made it through, or the query timeframe might be slightly beyond a time-based filter. Concretely, for each candidate chunk we check:
2. If the current time is past the chunk's `expiry_time` (TTL), then exclude that chunk. (It will likely be pruned from the store soon as well.)
3. If the chunk is marked `private` or otherwise sensitive and the agent's current mode disallows using such data (for instance, maybe certain highly sensitive info should not be automatically used), then exclude unless the user explicitly requested it.
4. If there are multiple chunks that contain redundant information (e.g., overlapping content due to how data was split), we might also cull duplicates at this stage for brevity.

TTL enforcement ensures the agent **does not consider information that it is supposed to have "forgotten."** This is important not only for policy compliance but also for relevance – an expired chunk by definition was deemed not to persist, so using it could reintroduce irrelevant or outdated context. For example, if an ephemeral chunk from yesterday's session was set to expire after 24h and the user comes back a week later with a related query, that chunk won't even be considered in retrieval[31]. This keeps the active context focused and the user's privacy protected (e.g., a one-time password shared an hour ago that expired should not be remembered by the agent later).

The outcome of this stage is a final set of top-\$k\$ memory chunks that are both relevant and policy-compliant to serve to the LLM.

1. **Context Assembly:** Although not a "retrieval" step per se, it's worth noting how the retrieved chunks are used. The selected memory chunks are formatted (often as a brief bullet list of facts or appended as a "Memory" section in the prompt) for the LLM's consumption. The system may include identifiers or citations in the prompt to trace the source of each memory if needed (for auditing or for allowing

the LLM to reason about multiple sources). For instance, the prompt might say: *"Previously, you learned: (1) Alice's favorite color is blue. (2) Alice's meeting is on Friday 3 PM."* before the user's new question. By providing only the most relevant info, we keep the prompt concise. In our design, typically only a handful of chunks (maybe 1–5) are included, depending on their length and the context budget. This contrasts with dumping an entire conversation history, which could be hundreds of messages. By focusing on the distilled memory, we achieve much better efficiency. (In the example scenario later, we show that each prompt can be just a few hundred tokens instead of many thousands, thanks to selective retrieval – yielding faster and cheaper responses.)

Relation to Prior Work: Our retrieval pipeline can be viewed as a combination of RAG best practices and new adaptive elements. Traditional RAG would do a vector similarity search and return the top results, which is effective but can return irrelevant info if the query embedding isn't precise. Researchers have improved on this with reranking strategies and filters. For instance, hierarchical retrieval methods for structured data apply filters much like our metadata step. The importance of *iterative retrieval* or reranking was underscored by Zeng et al. (2024)[1] they found that combining different retrieval methods and using rerankers consistently outperforms a single-step retrieval. Our system's reflection score essentially functions as a built-in reranker that favors chunks shown to be useful across iterations. Moreover, by enforcing TTL, we incorporate a temporal dimension often missing in retrieval models – ensuring *freshness* of the context (somewhat analogous to web search engines promoting fresher content for time-sensitive queries).

D. Reflection and Adaptation Loop

MemoryArchitect includes a continual learning mechanism whereby the system *updates and adapts its memory store after each interaction*. This Reflection & Adaptation Loop allows the memory to improve over time, focusing more on what's useful and shedding what's not. The process works as follows after the LLM agent produces a response (and before the next user query arrives):

1. **Usage Tracking:** The system examines which memory chunks were retrieved for the query and how they were used in the LLM's output. If the LLM's answer clearly utilized a chunk (for example, it repeated a fact from that chunk or was influenced by it), we mark that chunk as "used." This can be detected by simple string overlap (did the model mention a unique keyword from the memory?) or more sophisticated methods (embedding similarity between the model's answer and the memory chunk, or instrumentation of the LLM to see attention weights). If a chunk was retrieved but the LLM's answer did *not* use it (e.g., the model ignored it or went a different direction), we mark it as "unused." Additionally, if the model's answer contradicts a retrieved chunk – say the chunk says *"User's birthday is Jan 1"* but the model output *"Your birthday is Feb 2"* – we mark that as a **negative usage** (a sign of a possible error either in memory or model)[24]. These signals (used, unused, contradicted) form the basis of reflection.
2. **Feedback Reflection:** Given the usage info, MemoryArchitect "reflects" on the quality of its memory retrieval:
3. For each **used** chunk: we interpret this as the chunk being helpful and relevant. We *increase its reflection score*. For instance, a chunk might go from score 50 to 60 after one successful usage. Repeated usage over multiple conversations can raise it further (with a cap or diminishing returns perhaps). This echoes the idea in cognitive psychology that recalling a memory strengthens it.
4. For each **unused** chunk that was retrieved: this suggests the chunk might not have been relevant or necessary. We *decrease its reflection score* slightly (e.g., 50 down to 45). If a chunk consistently gets retrieved but never actually helps, its score will decay, indicating it might not be worth retrieving in the future. This addresses the "false positive" retrievals by learning to not surface them as often.
5. For **contradicted or misleading** chunks: this is a stronger negative signal. If a memory item led the model to make an error or conflict, it might be wrong or outdated. The system could drastically lower its score or even flag it for removal or human review. (In practice, contradictions might also arise from model error, but at least it flags that memory as contentious.)
6. Additionally, **unused memory** overall decays: any chunk that hasn't been retrieved or referenced in a long time can have a slow downward adjustment of reflection score (simulating forgetting curve). This

happens passively over time intervals.

This reflection step is analogous to the "experience replay" analysis in Xiong et al.'s study: they found that selectively adding and deleting experiences yields better performance[27]. Our approach implements selective deletion via the scoring mechanism – by lowering scores for unhelpful experiences, we prepare them to be deleted soon.

1. **Memory Pruning and Consolidation:** After updating scores, MemoryArchitect checks if any memory maintenance actions should be taken:
2. If some chunks now have a reflection score below a certain threshold (say the score fell below 20 out of 100), and they are not critical by policy, the system can **prune** them[4]. The assumption is that over many interactions, if a piece of information proved essentially useless (and especially if it's old or was ephemeral), it can be removed to free up space. This is akin to a forgetting threshold. The threshold can be tuned, but in principle ensures the vector store doesn't hold on to junk indefinitely.
3. The system also checks TTLs: any chunk that reached its expiry is removed now if it wasn't already. This might involve actually deleting the vector entry or moving it to an archive.
4. **Summarization or merging:** If multiple related chunks exist and some are low-value, the system might consolidate them. For instance, if over a session the user gave several updates that are now low-score, we might merge them into one summary chunk (this requires the dynamic context module, but can be triggered as part of cleanup).
5. **Conflict resolution:** In the reflection step, if contradictions were found, the system may decide to keep one version (e.g., the latest or highest-scored fact) and remove or mark the other as deprecated. This ensures consistency in the knowledge base.

The result of pruning is that the memory store remains lean. Only frequently used or high-scoring items persist in long-term memory, along with any canonical knowledge that was never meant to be removed. Low-scoring items effectively "fade out." This design is directly inspired by how human memory fades for non-reinforced details and by systems like MemoryBank that mimic such decay[31][[32]. It also incorporates the benefits observed in reflective systems: Google's RMM work noted improved accuracy by refining memory selection over time[24] – our system achieves a similar effect by continuously curating the memory to favor useful content.

1. **Policy Adaptation:** Reflection is also a chance to adjust policy parameters if needed. For example, if we notice the system is consistently retrieving too many ephemeral chunks that turn out unused, we might tighten the filter (only retrieve ephemeral if very high similarity or within shorter time window). This kind of meta-adaptation can be handled through offline analysis or an automated policy tuner. In a sense, MemoryArchitect can learn not just about specific facts, but about how to better apply its own policies through experience (though this aspect is future work and not fully implemented in the prototype).

Through this loop, MemoryArchitect's memory becomes **adaptive**. Over the long run, the user should experience an assistant that "remembers" the useful things (even across sessions separated by days or weeks) but conveniently forgets the clutter. Importantly, this adaptation is achieved without retraining the base LLM – all updates happen in the external memory system, which means it's fast and safe to do at runtime. We also log these changes: if a chunk was pruned, the audit log notes that it was removed due to low score, etc., allowing developers to inspect or override if something valuable was mistakenly dropped.

Comparison: The reflection loop in MemoryArchitect bears resemblance to *self-healing knowledge base* ideas in LLM agents. For example, the SELF-RAG framework trains a model to decide when to retrieve and even critique the retrieved info[1] – while our reflection is not done by the LLM itself, it achieves a similar outcome by iterative adjustment. Generative Agents (Park et al.) had agents periodically *r*eflect on their observations to form higher-level conclusions[18]; in our case, while we don't generate free-form reflections, we do raise or lower the importance of memories which is analogous to strengthening certain recollections. In Reinforcement Learning terms, one could see memory retrieval as an action and reflection score updates as a reward signal – encouraging the system to select certain memories more in the future (positive reward) or less (negative reward). Though we are not explicitly using RL here, the heuristic tuning of scores works

effectively as noted in prior personalization systems.

By continually aligning the memory contents with what truly helps the agent's performance, MemoryArchitect aims to **improve the agent's long-term effectiveness and consistency**. Over many interactions, the user should notice that the assistant becomes more accurate and context-aware (since the important facts are always remembered), and also safer (since erroneous or misleading info gets pruned away). This adaptive behavior addresses what Xiong et al. called "experience-following" — our agent's future outputs are shaped by a curated set of past experiences[27], hence we ensure those experiences are the right ones.

E. Dynamic Context Management

Even after retrieving relevant memory chunks, we must fit them into the LLM's input context which has a finite size. *Dynamic Context Management* is the component responsible for deciding **which pieces of information to include in the prompt for the current turn** and how to format them. It works closely with retrieval and reflection, but whereas retrieval figures out *what is relevant in general*, context management figures out *what is necessary right now given the token budget*. Key aspects of this component include:

- **Token Budgeting:** The system knows the maximum context length of the model (e.g., 4096 tokens, or 8192, etc.). It allocates space for the user's query, the system prompt (which might include instructions or persona), and the memory. For example, if the model context is 4096 tokens and the current user query plus system instructions take 500, we have ~3596 tokens left for memory and any other auxiliary info. We set a budget, say we aim to use at most 3000 tokens for memory to leave headroom. If the retrieved chunks combined are within this budget, we include them all. If not, we need to prioritize.
- **Prioritization and Eviction:** We rank the retrieved chunks by importance (this ranking is already available from the retrieval stage). We start including from the top until we hit the token budget. If adding the next chunk would overflow, we stop. In practice, our retrieval is tuned to return only a handful of results, so often all can be included. But in a case where, say, 10 chunks were somewhat relevant but together they are too long, only the top N will be used. The lower-ranked ones are effectively *evicted* from this prompt. They still remain in long-term memory; they're just not included this time. Eviction strategy ensures that **only the highest-value content stays in the working memory** (analogous to how humans might only recall the most salient facts when summarizing a complex event).
- **Summarization of Context:** If the content to include is too large or verbose, MemoryArchitect can invoke a summarization step. For example, if a relevant memory chunk is itself long (maybe a paragraph), but only a high-level point from it is needed, we can replace it with a shorter summary sentence. Summarization is done by an LLM or a heuristic and aims to preserve the key facts while cutting down length[20]. This is especially useful for **dialogue history**. Instead of including a word-for-word transcript of a past conversation, the system might include a summary: e.g., "(Earlier conversation: user discussed starting a garden and got advice on watering and sunlight.)" This gives the model the necessary context ("the user was talking about gardening and got some advice") without dumping the entire dialogue. Summarization can thus dramatically reduce tokens while retaining essential info[]Packer2024MemGPT].
- **Recency vs. Long-term mix:** Dynamic context management also balances recent conversation turns with long-term memory. Typically, for coherence, one will include the last few turns of the ongoing conversation verbatim (especially if it's a live session) in addition to long-term memory snippets. Our system can include, say, the last 1-2 user questions and assistant answers (ephemeral short-term memory) plus the retrieved long-term chunks. If even that threatens to overflow, we might start summarizing or dropping the oldest turns among those recent ones. Essentially, there is a *sliding window* for immediate dialogue and a *selective recall* for older knowledge. This approach is very much like MemGPT's idea of swapping: we keep recent interactions in context and rely on the vector store for older ones[17] except we automate it via rules rather than via the LLM's own decision.
- **Human-Inspired Mechanism:** One can draw an analogy to human memory: we have short-term memory (the last thing the user said, which the AI should directly address) and long-term memory (facts learned earlier). Humans can typically hold only a few recent items in mind (our "working memory"), and we abstract older events. Our dynamic context mechanism is designed to mimic this:

the LLM's prompt contains the immediate conversation (to maintain dialogue flow) and a selection of distilled long-term memories. We specifically drop or compress details that are no longer needed in full detail. For example, earlier small talk or intermediary steps that have since been resolved can be omitted or replaced with a note. This ensures the model's attention isn't wasted on irrelevant details and prevents important info from being diluted among a sea of tokens[7], [3], [23].

In practice, implementing this requires careful prompt orchestration. We might maintain a running *summary memory* for each session that is updated as the conversation progresses (as done in some chatbot implementations: a moving summary plus recent turns). MemoryArchitect can leverage its reflection loop here: if certain dialogue turns have been reflected into long-term memory (e.g., key facts extracted), we can feel safer dropping the verbatim text of those turns from the prompt now. They've been "remembered" in abstraction, so we don't need to carry the raw text.

Benefits: Dynamic context management yields efficiency gains. By not carrying the full conversation, we reduce token usage significantly. As an illustrative calculation, if a naive approach would have included 50 turns of dialogue (which could be say 5000 tokens) versus our approach which includes a summary and a few facts (maybe 500 tokens), the prompt is an order of magnitude smaller. This translates to faster inference and lower cost. We expect, as the system scales, that the average prompt size will stabilize at a manageable level (e.g., a few hundred to a thousand tokens) rather than growing linearly with conversation length. Prior research, such as in the Generative Agents paper, shows that agents can operate with a summarized memory of thousands of events and still behave consistently. Also,[7] noted that summarization combined with retrieval can maintain performance while keeping context size bounded – our design mirrors that philosophy.

Dynamic Contradiction Handling: Another aspect of context management is handling conflicting information in the prompt. If two retrieved chunks might conflict (say one says X is true, another says Y is true, and they can't both be true), the system has a choice: include both and let the model figure it out (not ideal, it might confuse the model), or decide which one to include (based on recency, reflection score, or external logic). We lean toward resolving such conflicts outside the prompt when possible. For example, if two facts conflict, often one is older and perhaps updated by the other – we would include only the latest or highest-scoring fact. This prevents the model from seeing contradictory cues. This kind of curation is again informed by the reflection loop which would have caught that conflict earlier and perhaps pruned one. If both facts must be included (maybe the user specifically asked to verify consistency), we would then at least label them or include context so the model can handle it. Generally, though, MemoryArchitect tries to present a *consistent set of context* to the LLM.

Overall, dynamic context management ensures that MemoryArchitect works within the LLM's limitations elegantly. It treats the context window as a precious resource to be allocated optimally. By doing so, it helps achieve one of our core goals: **minimize token overhead while preserving relevant knowledge**. This addresses the inefficiency of naive long contexts and is supported by evidence from systems like MemGPT (which essentially did a manual form of this through an LLM scheduler) and other efficient long-context solutions.

F. Server Load Optimization and Scalability

Finally, MemoryArchitect includes mechanisms that operate mostly behind-the-scenes to keep the memory system scalable and efficient as it grows with potentially many users and long periods of usage:

- **Periodic Expiry and Cleanup:** In addition to the on-the-fly TTL enforcement, the system can run a background job (say every hour or day) to permanently delete expired chunks from the vector store. This ensures the database is not cluttered with data that will never be retrieved. Over time, this keeps the index size manageable.
- **Pruning Thresholds:** We mentioned a reflection score threshold for pruning. This can be dynamically adjusted based on memory size constraints. For example, we might aim to keep at most N chunks per user. If the store exceeds, say, 10,000 chunks for a user, we may increase the threshold to prune more aggressively (drop the lowest-scoring until we're under the limit)[4]. This guarantees an upper bound on memory size per user. Such thresholds prevent any single user's data from consuming excessive resources, which is important in multi-tenant deployments.

- **Metadata Compression:** As memory entries age or decrease in importance, we can also compress them instead of full deletion. For instance, for very low-score items, we might strip the stored text and only keep an embedding (to save space) or keep a very short summary of it. We then won't ever use them to answer directly, but we might still use them for analytical purposes or not at all. This is similar to "archiving." Another technique is to compress groups of memories. Suppose a user had 100 ephemeral chunks about a task that's now completed; we could replace all of them with one summary note and drop the rest. This not only frees space but also removes clutter that could accidentally influence retrieval. Prior approaches like Generative Agents effectively did this by consolidating experiences into higher-level reflections[18]. We follow suit by ensuring no proliferation of redundant data.
- **Multi-tier Storage:** For extreme scalability, the system can employ a multi-tier approach: a fast vector store for recent and high-value memory, and a slower, cheaper storage for older or low-value memory (or even just backups on disk). In real deployments, one might use an in-memory store for active data and move cold data to disk or cloud storage periodically. Our design allows for this by tagging data with last-access times and scores; one could export chunks below certain thresholds to cold storage. This way, if a user disappears for a year and comes back, we can still retrieve some archived info if needed, but it doesn't live in the fast index meanwhile.
- **Caching Layer:** Another optimization is caching recent retrieval results or embeddings. For instance, if the user often asks the same question, the system can cache the answer or the retrieved memory to speed up response (though careful to invalidate if memory changes). On a broader scale, if many users ask similar queries that hit the same canonical chunks, those chunks might be cached in memory (in the computational sense) to avoid re-searching the database each time. This is more of a system optimization, but it leverages the fact that our memory chunks are discrete and identifiable.
- **Load Shedding Policies:** In heavy load scenarios, MemoryArchitect can degrade gracefully by, for example, reducing the number of chunks it retrieves or turning off reflection updates temporarily. This ensures that under high concurrent usage, the system prioritizes core functionality (delivering relevant info) over nice-to-have (like perfect reflection score updates every single turn).

All these optimizations aim to maintain **throughput and responsiveness** of the memory system at scale. Without them, a long-running agent might accumulate so much data that queries slow down (vector search over a huge index) and storage costs balloon. By continuously trimming and optimizing, MemoryArchitect keeps the memory "lean and fit." This is analogous to regular garbage collection in software systems.

From a research perspective, this component aligns with observations in works like MemoryBank and others: it's beneficial to remove stale data not just for accuracy but for speed. Also, at scale, engineering efforts like those in LangChain or Chroma have discussed sharding and multi-tenancy strategies[25] – our approach is consistent with their recommendations (we isolate by user, and we could shard by user if needed across databases). Techniques like the *attention sinks for streaming LLMs[26]* are more on the model side, but similarly aim to handle long sequences efficiently. Our focus here is external memory efficiency.

In summary, through server-side optimizations, MemoryArchitect ensures that its benefits (better long-term memory) do not come at the cost of system performance. It strives to be *practical for deployment*: maintaining quick retrieval times, bounded memory growth, and predictable behavior even as usage scales to many users with long histories.

III. Methodology and Evaluation

With the design of MemoryArchitect laid out, we now turn to the methodology for implementing and evaluating this system. We describe the prototype implementation, walk through an **extended example scenario** demonstrating how the system works in practice, and outline our evaluation plan including benchmarks and metrics.

A. Prototype Implementation Strategy

Our prototype of MemoryArchitect is built using a modular approach, integrating with existing LLM and vector database tools: - **LLM Backend:** For language generation, we can use OpenAI's GPT-4 via API or

an open-source model (like Llama 2 70B) hosted locally. The memory system is model-agnostic as long as we can intercept the model's inputs and outputs. We use the LangChain framework to orchestrate prompts, which allows easy insertion of our memory retrieval step before each LLM call. - **Vector Store:** We use ChromaDB as the vector database to store memory chunks. Chroma was chosen for its support of metadata filtering and ease of integration with LangChain[12]. Each memory chunk is stored with an embedding (we use OpenAI's text-embedding-ada model or a similar embedding model) and the metadata fields described (source, class, TTL, score, tags, user_id, etc.). - **Memory Controller:** We implement MemoryArchitect's logic as a Python class within the LangChain agent loop. This controller has methods `add_memory(chunk)`, `retrieve_memories(query)`, `reflect_and_update (used_chunks, unused_chunks)`, etc., corresponding to the components described. Policies (like TTL durations per class, reflection score thresholds, etc.) are configurable via a YAML config. - **Summarization and NLP utilities:** We leverage the LLM itself for tasks like tagging intent (classifying the topic of a query), summarizing long texts, and possibly detecting usage of chunks in answers (e.g., by asking the LLM explicitly which provided facts were used – although for now simpler heuristics are used to avoid extra calls). - **Audit Logging:** The system logs events to a structured log (e.g., a JSON file) recording what chunks were retrieved for each query, which were used, any chunks pruned, etc. This will be valuable for debugging and for evaluation.

Integration into Agent Workflow: In a typical loop, when a user message comes in: 1. The agent calls MemoryArchitect.retrieve_memories(query) to get relevant context. 2. MemoryArchitect does metadata filtering (user_id, intent from a quick classification of query), similarity search, ranking, TTL check, and returns a list of top memory chunks. 3. The agent inserts these into the prompt and the LLM generates a response. 4. After receiving the LLM's output, the agent calls MemoryArchitect.reflect_and_update(query, retrieved_chunks, LLM_output) which triggers the reflection loop. This updates scores and prunes if needed. 5. The cycle repeats for the next user query.

By wrapping these calls around the LLM, we ensure every answer benefits from the memory layer and every interaction updates the memory.

Policy Configuration: For our prototype, we set default policies as follows: *canonical* chunks have no TTL (persist indefinitely), *factual* chunks have a long TTL (e.g., 30 days) and may be updated rather than removed, *intent-bound* and *ephemeral* chunks have short TTLs (a few hours to a day), *private* chunks have medium TTL (maybe a week) but are only retrievable for the same user and might be filtered from certain outputs. Reflection score starts at 50; we prune chunks that drop below 20 and are older than some age (to avoid premature removal of new chunks). These values can be tuned, and part of our evaluation is to see how sensitive the system is to these parameters.

Ensuring Alignment and Safety: We incorporate some simple safety checks in the memory system: for example, *private* class data is never returned to queries coming from different users (ensured by user_id filter). Additionally, if certain data should never be exposed without user re-confirmation (like a very sensitive personal detail), we could tag it and require an explicit user query to retrieve (the system would otherwise act as if it's not there). While the primary focus of MemoryArchitect is memory efficacy, these governance features inherently improve safety by reducing unintended information leakage.

The implementation is designed to be extensible. For instance, one could plug in a different vector store or add a knowledge graph on the side if needed (similar to A-MEM's approach) for specialized queries. But the core prototype uses straightforward components to validate the concept.

B. Example Workflow in Action

To illustrate how MemoryArchitect functions end-to-end, consider the following **hypothetical multi-session scenario** with a user. We will step through four sessions, highlighting how each component of the system comes into play. This scenario will demonstrate MemoryArchitect's ability to preserve important information across sessions, use it to answer questions, and adapt the memory over time. (For clarity, we label memory chunks and track their metadata through the example.)

Initial Setup: A new user **Alice** begins using the assistant. MemoryArchitect creates a memory space for Alice (assigns `user_id = Alice`). Initially, the memory is empty or only contains some globally shared

canonical knowledge (in this case, assume nonspecific pre-loaded for simplicity).

Session 1: The user asks: "I'm starting a garden. What should I know about growing tomatoes?"

Retrieval: The query embedding for "growing tomatoes" is generated. Metadata filters: `user_id = Alice` (only her data, which currently is empty of any personal knowledge), and an inferred `intent_tag = gardening` (we classify the query as gardening topic). The system also knows this is a general factual question, so it might allow searching `source_type = doc` (looking into a gardening knowledge base) and perhaps some global canonical knowledge about tomatoes. It finds a few relevant chunks in the global knowledge base (or a pre-loaded gardening corpus): 1. *Chunk A:* "Tomatoes need 6-8 hours of sun daily" (class: canonical or factual, source: doc, reflection score default 50). Similarity to query is high. 2. *Chunk B:* "Water tomatoes deeply 2-3 times per week rather than daily" (class: factual, from a gardening tips doc, score 50). 3. *Chunk C:* "Ideal soil temperature for tomato germination is above 18°C (65°F)" (class: factual, score 50). Suppose these are the top 3 with similarity scores 0.9, 0.85, 0.7 respectively. Metadata filtering already limited to gardening domain content. All three chunks are factual and under Alice's user or global. None are expired (they might not have TTL as canonical info). The system ranks them: likely A > B > C by relevance. All have equal reflection initially. It will include as many as fit the budget. Say each is one sentence (~15 tokens each), we can include all three easily. - **Prompt Assembly:** The assistant's prompt now includes these facts in a "Memory" section, for example: - Memory: (1) *Tomatoes need 6–8 hours of sun daily.* (2) *Tomatoes prefer deep watering a few times a week rather than daily light watering.* (3) *They germinate best when soil is warm (above 18°C).* Then the conversation (system message, user question) follows. This provides the LLM with relevant knowledge about tomato growing. - **LLM Response:** Using this memory, the LLM answers Alice's question, likely explaining how much sun tomatoes need, how to water them properly, and maybe mentioning soil temperature, etc. For instance, it might say: "To grow tomatoes, make sure they get 6-8 hours of sunlight each day. Water them deeply about 2-3 times a week rather than a little every day, so the roots grow strong. Also, ensure the soil is warm (at least 65°F) for good germination..." The model has thus utilized chunks A and B for sure, and possibly C. - **Reflection:** After output: - The system checks usage. The answer explicitly mentioned sunlight and watering (chunks A and B content), but did it mention soil temp? If not, chunk C was not used. So mark A and B as *used*, C as *unused*. No contradictions are present (the answer didn't conflict with the facts, it echoed them). - Update scores: A's score 50 -> 60, B: 50 -> 60 (they were useful). C: 50 -> maybe 45 (slight downgrade for being retrieved but unused). - No chunk falls below pruning threshold (45 is still fine). - **Memory writing:** Now, the conversation itself can be stored. The user's question "I'm starting a garden... growing tomatoes?" and the assistant's answer can be stored as well, if we think they might be useful later. We classify those as `chat_user` and `chat_assistant` chunks respectively, likely `ephemeral` or `intent-bound` (since they are specifically about the gardening query). We tag them with `gardening`. We might give them a TTL of, say, 7 days (so that if Alice comes back soon, we remember what was discussed, but if months pass, this specific Q&A might expire). Their reflection score might start lower (say 30) because a specific QA context is less broadly useful than a general fact. So we store: - *Chunk U1:* ("User asked about growing tomatoes..."), type: chat_user, tags: gardening, TTL: 7d, score: 30. - *Chunk A1:* ("Assistant answered with advice on sun, watering..."), type: chat_assistant, tags: gardening, TTL: 7d, score: 30. Now Alice's memory store has: chunks A, B, C (factual knowledge from docs) and U1, A1 (conversation history).

Session 2 (next day): Alice returns and asks: "Remind me how often I should water my tomatoes."

Retrieval: Now `user_id = Alice` filter includes her existing memory from Session 1. The query is classified as gardening again. The system will search Alice's memory. Relevant chunks might be: - The factual chunk about watering (Chunk B: "water 2-3 times a week"), which now has reflection score 60 and tag gardening. - The assistant's prior answer (Chunk A1) which likely contained the watering advice in prose (score 30, ephemeral chat). - Perhaps the user's question chunk (U1) but that just contains the question, not useful for answer. - Maybe the sunlight chunk (A) could match slightly (since watering is mentioned with sunlight sometimes, but likely chunk A is about sun, not directly matching "water" query).

The system filters by `gardening` tag. It retrieves chunk B with very high similarity (the query explicitly about watering matches chunk B almost verbatim). It also might retrieve A1 (because A1 contains the assistant's full answer text which includes watering advice, though similarity might be moderate). Ranking:

Chunk B (factual, score 60) vs A1 (ephemeral, score 30). Even if A1 had some overlap in content, chunk B is clearly higher priority (both because of higher similarity and higher reflection score). TTL check: All are within TTL (B had none or long, A1 has 7d and it's just the next day, so still valid). The system decides to include chunk B for sure. It might include chunk A1 if space allows, but since chunk B already contains the needed info succinctly, including A1 (which is a verbose answer from yesterday) is redundant. Let's say it includes only B for brevity. - **Prompt Assembly:** Memory provided: *"Previously noted: Tomatoes should be watered deeply 2–3 times per week rather than every day."* followed by the user's new question. - **LLM Response:** The model responds: "You should water your tomatoes deeply about two to three times a week, rather than daily." This is essentially chunk B repeated, showing the assistant "remembered" the advice. - **Reflection:** The watering chunk B was used again. Its reflection score goes from 60 -> 70 (even more solid now). The assistant's old answer chunk A1, if it was retrieved and not used (suppose we did retrieve it but model preferred the factual phrasing), might get a slight negative (30 -> 28). If it wasn't even retrieved, no change. Either way, ephemeral chat chunks like A1 might approach expiry in a few days. Let's say at this point 24h passed, if A1's TTL was 1 day, it might actually expire now. So the system may mark chunk A1 (and U1) as expired and remove them. That's fine, because the important info from that exchange (watering advice) is preserved in chunk B which persists as factual knowledge. This demonstrates how ephemeral conversation detail doesn't clutter long-term memory beyond its useful window.

After Session 2, Alice sees that the assistant could recall the specific watering schedule without her repeating it – a success of the memory system fetching chunk B.

Session 3 (a week later): Alice asks: "My tomatoes have yellow leaves. What could be wrong?"
This is a troubleshooting question, not explicitly connected to the previous Q&A, but there might be relevant info from before (overwatering can cause yellow leaves). Also, new info might need to be fetched. - **Retrieval:** Filter by `user_id = Alice`, tag maybe `gardening` (likely yes). The system will consider: - Past chunks: sunlight (A: sun hours, maybe relevant if lack of sun can yellow leaves), watering (B: how often watering, relevant because overwatering is a common cause of yellow leaves). - Reflection scores: A is 60, B is 70 now, both tagged gardening. C (soil temp) is 45, also gardening but less relevant to yellow leaves. - Also, the system may search the global knowledge for "yellow leaves tomatoes" and find a new chunk: - *Chunk D:* "Yellow leaves on tomato plants are often a sign of overwatering or nutrient deficiency." (class: factual, from a gardening troubleshooting guide, score 50 initially). It retrieves chunk D (high similarity to "yellow leaves" query). It also retrieves chunk B (the user's watering habit info) because even though the query didn't mention watering, the system might see a semantic link: chunk B's text about watering might have some similarity to "watering" concept in D, or we might proactively include the user's known context (since we tag chunk B with gardening, it might come up anyway as moderately similar due to overlapping domain terms). Chunk A (sunlight) might also be considered if under-watering or lack of sun are known causes; similarity might be lower, so perhaps not retrieved if we limit to top 3. Let's say the top results are: - D: "Yellow leaves => overwatering or nutrient deficiency" (sim score 0.9, new factual, score 50). - B: "Water 2-3 times/week" (sim ~0.7 because the word "water" appears, score 70). - A: "Needs 6-8h sun" (sim 0.5, score 60, might be borderline). Ranking with reflection scores might actually put B with its high score close to D. But D is directly addressing yellow leaves, so likely D remains top, B second. We might include both D and B (they likely fit in prompt). C and others are not included. TTL check: B and A are non-expiring or long (still fine), D is new, no TTL. - **Prompt Assembly:** Provide memory: e.g. 1. *"You have been watering tomatoes 2–3 times per week (your last noted schedule)."* (This reminds the model of the user's specific context from chunk B.) 2. *"According to gardening tips: Yellow leaves on tomatoes can indicate overwatering or a nutrient deficiency."* (from chunk D). User's query about yellow leaves follows. - **LLM Response:** The model uses this info to form an answer: "Yellow leaves could be a sign of overwatering – even though watering 2-3 times a week is usually okay, make sure the soil isn't waterlogged. It can also indicate a lack of nutrients like nitrogen, so consider fertilizing. Ensure they are getting enough sunlight as well." The answer combined the general knowledge (overwatering or nutrients) with the user-specific memory (how often Alice waters). Notably, the assistant contextualized the advice by recalling that Alice waters 2-3 times/week, suggesting overwatering *might* still be an issue if perhaps the soil drainage is bad. This shows the power of combining personal and general knowledge. - **Reflection:** - Chunk D was used (the answer talked about overwatering and nutrients) – raise D's score (50 -> 60). - Chunk B was referenced (the answer explicitly mentioned the watering frequency that matches chunk B) – raise B's score (70 -> 75). - Chunk A was not used or not included – maybe

lower A a bit (60 -> 55) if it was retrieved and ignored. - No contradictions, and no immediate pruning needed (all are high-score). - We might tag the situation of "tomato yellow leaves troubleshooting" as an `intent-bound` memory if we store this Q&A. Perhaps we store: - *Chunk U2:* user's question about yellow leaves, tag gardening, TTL moderate (since it might be relevant in context of this ongoing gardening project). - *Chunk A2:* assistant's answer about yellow leaves, tag gardening, TTL moderate, score 30 initial. But these may or may not be needed later. They might be kept for continuity within this gardening topic.

At this point, the system has accumulated a set of factual chunks (A: sun, B: watering, C: soil temp, D: troubleshooting leaves) that represent a **knowledge base about Alice's gardening**. It also has some ephemeral interaction records (which will expire or be summarized eventually). The reflection scores have ensured that the frequently referenced pieces (watering, now troubleshooting) are recognized as important. The assistant has demonstrated memory of past interactions (knowing how often Alice waters, etc.) without the user repeating it.

Session 4 (same day, continuing the conversation): Alice follows up in the same session: "I think I might be overwatering then. How can I improve drainage?"

Here, the conversation is ongoing (Session 3's context is still fresh). This question is directly building on the previous answer. - **Dynamic Context in play:** Since this is same session and a direct follow-up, MemoryArchitect will include the immediate last Q&A turn in prompt (for coherence). So the last user question about yellow leaves and assistant's answer are still within the short-term context. They haven't been forgotten yet (we typically keep, say, the last 2-3 exchanges verbatim in the prompt to maintain conversational flow). - **Retrieval:** We still filter `gardening`. Now the query is about improving drainage. Possibly relevant: - Perhaps a global tip about drainage (e.g., *"To improve soil drainage, use well-draining soil, add perlite, ensure pots have holes"* could be a new chunk from knowledge base). - The system will search docs for "improve soil drainage tomatoes" and likely find one. Call it *Chunk E:* "Improving drainage: use well-draining potting mix, ensure containers have drainage holes, avoid overwatering." Score 50. - It also considers user's own context: chunk B (watering) again might come up since drainage is linked to watering frequency. But the query explicitly says she acknowledges overwatering, now asks how to improve drainage, so chunk E is more relevant. - We include chunk E. - We also keep the recent turn's content in context (assistant's previous advice which mentioned waterlogged soil). - **Prompt:** Memory might include: "(Recall from earlier: Overwatering can cause yellow leaves. To improve drainage, make sure to use proper soil and pots with holes.)" – some of that might be in the recent assistant answer already. Possibly we rely on the assistant's last answer (already in context) which said "ensure proper drainage". If that was already said, the user's question is basically asking for elaboration. The assistant can answer from general knowledge. Whether we inject chunk E or let the model answer from its own knowledge is optional; but adding chunk E ensures accurate details. - **LLM Response:** The model advises on improving drainage: e.g., "You can improve drainage by mixing sand or perlite into your soil, planting in raised beds or containers with drainage holes, and by not watering too frequently. Ensuring excess water can escape will prevent waterlogging." - **Reflection:** The chunk E if used will get a score bump. Others unchanged. The conversation continues...

This scenario shows MemoryArchitect maintaining continuity across multiple sessions (Session 1 to 3 were separate days) and also within a session (Session 3 to 4 follow-up). The user did not have to repeat earlier information – the assistant "remembered" it and used it appropriately. At the same time, the system did not keep irrelevant details around. Notice that by Session 3, the exact wording of the Session 1 answer wasn't needed – only the key facts were retained. Ephemeral data (like the full dialogue of Session 1) was cleared after serving its purpose. The long-term memory now contains distilled knowledge that the user and assistant co-constructed (some from the knowledge base, some from their specific Q&A). This is exactly how MemoryArchitect is intended to function.

A few points demonstrated: - **Policy-controlled retention:** Ephemeral chat chunks from Session 1 expired and did not pollute Session 3, but factual chunks persisted. - **Metadata filtering:** Only gardening-tagged info was pulled for gardening questions; if Alice had a completely separate conversation about, say, cooking in another session, those would be tagged differently and would not interfere. - **Adaptive retrieval:** The watering chunk's high reflection score kept it high in rankings even for semi-related query (it was included in Session 3 retrieval because the system recognized it as important context that might matter – indeed it did).

Had the user's watering habit not been relevant, worst case it's not used and eventually its score would drop, but in this case it was relevant. - **Summarization:** We didn't explicitly show a big summarization, but imagine if the conversation had 20 turns of detailed troubleshooting, the system would have summarized earlier parts once they became less needed. For instance, after solving the yellow leaves issue, if a month later the user comes with a new gardening question, the system might just keep a note "(User previously dealt with overwatering issues)" rather than the full transcript. - **Scalability:** Each query, we only searched within Alice's ~4 relevant chunks plus some global knowledge. Even as global knowledge might be large, metadata and intent filters narrow it (only gardening domain). This makes retrieval fast. The memory store for Alice is still small (a few chunks). Over years, it may grow, but with pruning and summarization, it remains manageable.

C. Evaluation Plan

To rigorously evaluate MemoryArchitect's effectiveness, we propose a two-pronged evaluation: (1) **Standard Benchmarks** for long-term memory in dialogs, and (2) **Custom Scenarios** that test specific features like policy compliance and efficiency.

Benchmarks: A primary benchmark is the **LoCoMo Long Conversation Memory** benchmark (Maharana et al. 2024)[15], which consists of multi-session dialogue tasks (up to 35 sessions, hundreds of turns) requiring the agent to recall facts and maintain consistency over time. We will compare an LLM agent with MemoryArchitect against baseline approaches (e.g., an agent with no long-term memory, an agent with a simpler vector-store retrieval without our governance layer, etc.). Key metrics: - *Accuracy on recall questions:* how often does the agent correctly answer questions about earlier facts? (LoCoMo provides such queries at various dialogue stages.) - *Consistency:* measures whether the agent's answers remain consistent (not contradicting itself) across sessions. - *Token usage:* how many tokens on average does the agent use per turn (prompt length), as a measure of efficiency. In preliminary expectations, MemoryArchitect should significantly improve recall accuracy (since important facts are retained and retrieved) – for instance, Mem0 achieved ~26% improvement on similar tasks[4], we aim for similar or better. Consistency should improve due to policy-driven pruning of contradictory info. Token usage should be much lower than a full-history baseline (we will quantify the reduction; e.g., using 90% fewer tokens as Mem0 did or maintaining a flat usage vs. growing with conversation length).

We will also test on **custom QA tasks** inspired by Generative Agents[18] and others: e.g., an agent given a series of interactions to simulate "life events" and then asked questions that require long-term memory. We can create a synthetic test where certain facts are mentioned and should be remembered or forgotten per policy (for example, the agent learns a user's secret that should be forgotten after one use – we test if it indeed doesn't recall it later unless allowed).

Custom Scenario Evaluation: We will specifically test MemoryArchitect's unique features: - *Policy enforcement:* We conduct experiments where certain data points are tagged with different policies and verify the behavior. For instance, feed the agent a piece of sensitive info tagged `private` with TTL = 1 day. Query the agent after TTL expiry whether it still knows that info – the expected outcome is that it does **not** (to confirm deletion). We also check that one user's info does not appear in another user's session (testing multi-user isolation). - *Adaptive memory benefits:* We simulate a scenario with some misleading facts injected. For example, the agent is given a false piece of info at some point which leads it to an error, and we observe if the reflection loop downranks or removes that piece, and if performance improves in subsequent turns (similar to RMM's analysis of error propagation). - *Efficiency:* We measure system response time and prompt size over long dialogues. The expectation is that with MemoryArchitect, prompt size grows sublinearly with conversation length – ideally plateauing after a point because old stuff gets summarized or removed. This can be compared to a baseline of sliding window (where prompt grows until it hits the window size hard limit, then truncates context) or full context (which is infeasible beyond a certain point). We might see, for example, that after 100 turns, the baseline might be at max context (e.g., 4096 tokens used every time, losing older info), whereas MemoryArchitect might only use, say, 1000 tokens on average with all relevant info packed in. This would confirm the efficiency advantage. We will log average tokens per turn and compute the reduction.

D. Expected Outcomes

We expect MemoryArchitect to demonstrate the following improvements over baseline memory-less or naive-memory agents: - **Improved long-term recall** of user-specific facts (the agent will answer follow-up questions correctly more often, even across sessions, without user reminder). - **Reduced contradictions and hallucinations**, thanks to consistent retrieval of relevant facts and the pruning of stale misinformation (aligning with findings that better memory management yields more robust performance. - **Lower average token usage** per response, making the system more cost-effective. The dynamic context management should dramatically reduce prompt size on long conversations (similar to the 90% reduction reported by others). - **Enhanced user trust and compliance:** The agent should not retain information beyond policy, addressing privacy concerns. If tested, we expect to show that MemoryArchitect can comply with retention limits (e.g., deleting data after X time) whereas a standard vector-store agent might unknowingly surface something from long ago. - **Scalability:** The system should handle many sessions/users without performance degradation, due to the optimizations. As evidence, we'll track memory store size per user and retrieval latency.

Our evaluation will also consider any failure modes. For instance, if the reflection loop aggressively prunes something that later became relevant, the agent might forget something it shouldn't. We will analyze such cases from logs and potentially adjust parameters (this is part of the research: finding the right balance in the policies). Another potential issue is if the tagging/classification fails (intent tags wrong), the retrieval might miss relevant info – we need to measure the robustness of the intent classification. However, since we also rely on semantic similarity, the system has a fallback to still retrieve by content.

In conclusion, the methodology combines both controlled experiments and real-world simulation to cover both functional performance and qualitative behavior. We will use the results to iterate on the policy configurations. Ultimately, we aim to show through these evaluations that MemoryArchitect provides a **significant advancement in long-term memory capability for LLM-based agents**, enabling them to be more **contextually aware, efficient, and trustworthy** over prolonged interactions.

IV. Conclusion

In this work, we introduced *MemoryArchitect*, an adaptive policy-controlled memory governance layer for retrieval-augmented LLM systems. By integrating a prototype design for an efficient RAG memory system with multi-session support, we demonstrated how an LLM-based agent can achieve persistent yet controlled memory. The architecture combines *structured memory chunks* (with rich metadata and policy tags), an *intelligent retrieval pipeline* (metadata filtering, hybrid ranking, TTL enforcement), a *reflection loop* to adapt over time, *dynamic context management* for prompt efficiency, and *server-side optimizations* for scalability. Together, these components enable an AI assistant that remembers important information about the user and past dialogues, while staying within operational bounds (token limits, privacy rules).

Our design draws on and extends ideas from state-of-the-art research: it retrieves like a RAG system, "forgets" like MemoryBank's forgetting curve (through decay of reflection scores)[31], reflects on its performance similar to Google's RMM[24], and manages context akin to MemGPT's swapping strategy but implemented via explicit rules rather than latent decisions We mapped these influences to our system and showed that by unifying them, MemoryArchitect achieves capabilities that individual prior systems lacked (no previous single system implemented *all* of policy-driven retention, reflection, summarization, etc. in one).

The extended example scenario illustrated MemoryArchitect in action: the agent retained key facts (like a watering schedule) across sessions and leveraged them to provide a personalized, coherent experience, all while automatically expiring the transient details. This kind of long-term consistency with minimal prompt bloat is a clear improvement over baseline approaches.

In summary, MemoryArchitect shows that by adding a well-designed memory layer, we can transform an LLM agent from a stateless oracle into a continuity-preserving assistant that **adapts, learns, and abides by explicit memory policies**. This not only improves the user experience, but also enhances reliability and safety (the agent is less likely to hallucinate or divulge old sensitive info improperly). We view MemoryArchitect as a step towards AI systems that possess a more **human-like memory**: one that forgets the trivial, remembers the important, and can explain its own recollections. Such systems could form

the backbone of next-generation personal AI assistants, long-running autonomous agents, and any LLM application where long-term knowledge is key.

Through comprehensive evaluation and iterative refinement, we will validate MemoryArchitect's contributions and pave the way for robust long-term memory management in AI.

References

[1] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. Technical report, github, 2024.

[2] American Psychological Association. Supplemental material for a new look at memory retention and forgetting, 2022.

[3] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

[4] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

[5] Dan Denis, Dimitrios Mylonas, Craig Poskanzer, Verda Bursal, Jessica D. Payne, and Robert Stickgold. Sleep spindles preferentially consolidate weakly encoded memories. *Journal of Neuroscience*, 41(18):4088–4099, 2021.

[6] Cody V. Dong, Qihong Lu, Kenneth A. Norman, and Sebastian Michelmann. Towards large language models with human-like episodic memory. *Trends in Cognitive Sciences*, 2025.

[7] Moulik Gupta. Can ai truly develop a memory that adapts like ours? Towards Data Science, 2025.

[8] Chuanyang Hong and Qingyun He. Enhancing memory retrieval in generative agents through llm-trained cross attention networks. *Frontiers in Psychology*, 2025.

[9] Alexis Huet, Zied Ben Houidi, and Dario Rossi. Episodic memories generation and evaluation benchmark for large language models. Technical report, Telecom Paris, 2025.

[10] Satish Jayanthi. The secret to more accurate, intelligent llms isn't data — it's metadata. Coalesce Blog, 2025.

[11] Hailey Joren, Jianyi Zhang, Chun-Sung Ferng, Da-Cheng Juan, Ankur Taly, and Cyrus Rashtchian. Sufficient context: A new lens on retrieval-augmented generation systems. Technical report, Google Research, 2025.

[12] LangChain Team. A long-term memory agent. LangChain Documentation, 2025.

[13] Zhiyu Li, Shichao Song, Chenyang Xi, Hanyu Wang, Chen Tang, et al. Memos: A memory operating system for ai systems. *arXiv preprint arXiv:2507.03724*, 2025.

[14] Geng Liu, Fei Zhu, Rong Feng, Zhiqiang Yi, Shiqi Wang, Gaofeng Meng, and Zhaoxiang Zhang. Semi-parametric memory consolidation: Towards brain-like deep continual learning. *arXiv preprint arXiv:2504.14727*, 2025.

[15] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. *arXiv preprint arXiv:2402.17753*, 2024.

[16] MATFFO. Policy-based control in ai agents. Matoffo Blog, 2025.

[17] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2024.

[18] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th ACM Symposium on User Interface Software and Technology (UIST 2023)*, 2023.

[19] Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

[20] Francesco Salvi, Manoel Horta Ribeiro, Riccardo Gallotti, and Robert West. On the conversational persuasiveness of gpt-4. *Nature Human Behaviour*, 2025.

[21] Drishti Shah. Using metadata for better llm observability and debugging. Portkey Blog, 2025.

[22] Minchae Song. Enhancing rag performance by representing hierarchical nodes in headers for tabular data. *IEEE Access*, 2025.

[23] Timothy Tadros, Giri P. Krishnan, Ramyaa Ramyaa, and Maxim Bazhenov. Sleep-like unsupervised replay reduces catastrophic forgetting in anns. *Nature Communications*, 2022.

[24] Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long T. Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, Anand Iyer, Tianlong Chen, Huan Liu, Chen-Yu Lee, and Tomas Pfister. In prospect and retrospect: Reflective memory management for long-term dialogue agents. *arXiv preprint arXiv:2503.08026*, 2025.

[25] Liyuan Wang, Xingxing Zhang, Qian Li, Mingtian Zhang, Hang Su, Jun Zhu, and Yi Zhong. Incorporating neuro-inspired adaptability for continual learning in ai. *Nature Machine Intelligence*, 2023.

[26] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. Technical report, MIT Han Lab, 2025.

[27] Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts llm agents: An empirical study of experience-following behavior. *arXiv preprint arXiv:2505.16067*, 2025.

[28] Wujiang Xu, Kai Mei, Hang Gao, Juntao Tan, Zujie Liang, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

[29] Zaoyang. Audit logs in llm pipelines: Key practices. Newline Blog, 2025.

[30] Ruihong Zeng, Jinyuan Fang, Siwei Liu, and Zaiqiao Meng. On the structural memory of llm agents. *arXiv preprint arXiv:2412.15266*, 2024.

[31] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing llms with long-term memory. *arXiv preprint arXiv:2305.10250*, 2023.

[32] Weishun Zhong, Tankut Can, Antonis Georgiou, Ilya Shnayderman, Mikhail Katkov, and Misha Tsodyks. Random tree model of meaningful memory. *Physical Review Letters*, 2025.