Adaptive Context Management for Long-Running LLM Agent Sessions

Kaman AI Research

March 2026

Abstract

Long-running agent sessions that interleave multi-turn dialogue, tool invocations, and chain-of-thought reasoning present a fundamental resource management challenge: the finite context window of the underlying language model must simultaneously accommodate system instructions, conversation history, dynamically discovered tool schemas, retrieved knowledge, and structured reasoning artifacts. As sessions extend beyond dozens of turns—common in enterprise workflow orchestration—naïve approaches that simply concatenate all state into the prompt exhaust the context budget, leading to degraded coherence, hallucinated tool parameters, and silent information loss. We present an *adaptive context management* architecture for LangGraph-based agent systems that treats the context window as a managed resource subject to continuous monitoring and tiered compression. The architecture introduces a pre-flight context checking node that estimates token budgets before every LLM invocation, a tiered message compression strategy that preserves recent exchanges verbatim while summarizing older history, a structured thinking framework with confidence tracking that reduces redundant reasoning, and a sub-agent orchestration pattern that isolates specialized tasks into independent context scopes. These mechanisms are complemented by an Express-style middleware pipeline that enables cross-cutting context augmentation—including collective memory recall from a hierarchical knowledge management system—without coupling context logic to individual agent nodes. In production deployment across enterprise workloads, the architecture sustains coherent sessions exceeding 100 turns while maintaining context utilization below 85% of capacity, achieving compression ratios between 3:1 and 8:1 on historical messages with negligible impact on task completion rates.

1 Introduction

The emergence of tool-augmented LLM agents as orchestrators of complex enterprise workflows has introduced a class of engineering challenges absent from single-turn language model applications. A customer service agent resolving a billing dispute may require dozens of exchanges—querying account records, consulting knowledge bases, computing adjustments, drafting communications, and seeking human approval—each step adding messages, tool call records, and reasoning traces to the session state. An analytics agent building a quarterly report may chain together data extraction, transformation, visualization, and narration steps across scores of turns.

These extended interactions expose the context window as the critical bottleneck. Modern LLMs offer context windows ranging from 128K to 1M tokens, but the practical usable capacity is substantially smaller. System prompts, tool schemas, and retrieved knowledge consume a significant baseline allocation before any conversation begins. Each subsequent turn adds user messages, assistant responses, tool invocations with their arguments and results, and internal reasoning artifacts. Without active management, context consumption grows linearly with session length, and sessions of 40–60 turns routinely approach the effective capacity limit even on models with generous context windows.

The consequences of context exhaustion are severe and often subtle:

- **Silent truncation.** Many inference frameworks silently drop older messages when the context limit is reached, causing the agent to lose track of earlier instructions, decisions, and constraints without any explicit error signal.
- **Attention degradation.** Even within the nominal context window, transformer attention to older tokens degrades with distance [1], causing the agent to "forget" relevant information that is technically still present in the prompt.
- **Tool schema displacement.** As conversation history grows, it displaces the token budget available for tool schemas, forcing the system to either reduce the number of available tools or truncate tool descriptions—both of which impair the agent's ability to select and parameterize tools correctly.
- **Reasoning collapse.** Chain-of-thought reasoning becomes less coherent as the prompt grows, with the model struggling to maintain consistent hypotheses and plans across a large con-

text.

We present an adaptive context management architecture that addresses these challenges through five interconnected mechanisms: pre-flight context estimation, tiered message compression, structured thinking with confidence tracking, sub-agent isolation, and middleware-based context augmentation. The system is designed for LangGraph-based agent architectures with PostgreSQL-backed state persistence, though the principles generalize to other stateful agent frameworks.

The remainder of this paper is organized as follows. Section 2 describes the agent state architecture that underlies context management. Section 3 presents structured thinking with confidence tracking. Section 4 details the pre-flight context checking mechanism. Section 5 describes tiered message compression. Section 6 covers dynamic tool context management. Section 7 introduces sub-agent orchestration for context isolation. Section 8 describes the middleware architecture. Section 9 covers hierarchical memory integration. Section 10 presents empirical evaluation. Section 11 discusses related work. Section 12 concludes.

2 Agent State Architecture

The context management system operates over a richly structured agent state that extends beyond the simple message list found in basic chatbot implementations. The state is formalized as a typed graph state within LangGraph's StateGraph abstraction, with each field governed by a *reducer* that defines how concurrent updates are merged.

2.1 State Composition

The agent state comprises several categories of information, each with distinct lifecycle and compression characteristics:

- **Messages.** The canonical conversation history, including user inputs, assistant responses, tool call requests, and tool results. Messages are the primary consumer of context budget and the primary target of compression.
- **Thought data.** Structured reasoning artifacts produced by the agent's thinking process, including hypotheses, confidence assessments, and planning notes. These are consumed internally and may be summarized or discarded after use.
- **Discovered tools.** Dynamically loaded tool definitions maintained in a session-scoped cache. Tool schemas consume significant tokens and are managed by a dedicated context manager (see Section 6).
- **Suggestions.** Proposed next actions or questions surfaced to the user. These are transient and do not accumulate across turns.
- **Artifacts.** Generated outputs such as charts, documents, code snippets, and data tables. Artifacts are stored by reference rather than inline, with only metadata (type, identifier, summary) retained in the active context.
- **Session metadata.** Configuration, user identity, agent personality, and workflow state. These are relatively static and consume a fixed token budget.

2.2 Persistence and Checkpointing

State is persisted to PostgreSQL via a checkpoint mechanism that serializes the full state graph at configurable intervals. This persistence serves dual purposes: it enables session resumption after server restarts or failovers, and it provides a historical record for debugging and analysis.

The checkpointing strategy is designed to be compatible with context compression: when older messages are compressed (replaced by summaries), the checkpoint reflects the compressed state. The original uncompressed messages remain accessible in the checkpoint history for audit purposes, but the *active* state—the state used for LLM invocations—contains only the compressed representation.

2.3 Token Accounting

Each state component contributes to a *token budget model* that the pre-flight context checker (Section 4) evaluates before every LLM call. The budget model partitions the total context capacity $C_{\max}$ into allocated regions:

$$C_{\max} = C_{\text{sys}} + C_{\text{msg}} + C_{\text{tools}} + C_{\text{aux}} + C_{\text{reserve}} \quad (1)$$

where C_{sys} is the system prompt allocation, C_{msg} is the message history allocation, C_{tools} is the tool schema allocation, C_{aux} covers auxiliary context (retrieved knowledge, memory, skills), and C_{reserve}

is headroom reserved for the model's output generation. Table 1 shows a representative allocation for a 128K-token model.

Table 1: Context budget allocation for a 128K-token model.

Component	Tokens	% of Total
System prompt	4,000	3.1%
Message history	70,000	54.7%
Tool schemas	12,000	9.4%
Auxiliary context	20,000	15.6%
Output reserve	16,000	12.5%
Safety margin	6,000	4.7%
Total	**128,000**	**100%**

The allocations are not rigid partitions but soft targets. The pre-flight checker uses them to determine when compression should be triggered: if message history exceeds C_{msg}, the compression pipeline activates. If tool schemas exceed C_{tools}, tool eviction is triggered. The safety margin provides a buffer against estimation errors in the token counting heuristic.

3 Structured Thinking with Confidence Tracking

Before the agent invokes the LLM for response generation, it passes through a `think` node that produces structured reasoning artifacts. Unlike free-form chain-of-thought prompting, where reasoning is interleaved with the response and persists as unstructured text in the message history, structured thinking produces typed objects that can be managed, compressed, and discarded independently of the conversation.

3.1 Thought Structure

Each thought object captures several dimensions of the agent's reasoning state:

- **Hypothesis.** The agent's current best understanding of what the user needs and how to accomplish it.
- **Confidence score.** A calibrated self-assessment (0.0–1.0) of the hypothesis quality, influenced by available information, tool results, and consistency with prior observations.
- **Information gaps.** Explicit enumeration of what the agent does not yet know but needs to determine.
- **Next-thought-needed flag.** A boolean indicating whether the agent should continue deliberating before acting.
- **Suggested actions.** Concrete next steps—tool calls, clarifying questions, or direct responses—ranked by expected utility.

3.2 The Think–Act Cycle

The structured thinking mechanism creates a deliberative cycle within the agent's state graph:

1. The `think` node analyzes the current state and produces a thought object.
2. If the confidence score exceeds a threshold τ_c and the next-thought-needed flag is false, the agent proceeds to `fetch_tools` and then `call_llm`.
3. If confidence is below τ_c or further deliberation is needed, the agent may loop through additional thinking iterations, constrained by a maximum iteration bound.
4. Tool results feed back into the thinking process: after each tool call completes, the think node reassesses hypotheses in light of new information.

This cycle can be expressed as a confidence update rule:

$$\gamma_{k+1} = \gamma_k + \alpha \cdot \Delta_{\text{info}}(r_k) \cdot (1 - \gamma_k) \tag{2}$$

where γ_k is the confidence at iteration k, α is a learning rate governing how quickly new information shifts confidence, and $\Delta_{\text{info}}(r_k)$ is an information gain measure derived from the tool result r_k. The $(1 - \gamma_k)$ factor ensures diminishing returns as confidence approaches certainty, preventing overconfidence from a single strongly informative observation.

3.3 Context Efficiency

Structured thinking improves context efficiency in two ways. First, thought objects are *ephemeral*: they inform the current LLM invocation but are not appended to the persistent message history. Only a compact summary of the reasoning—the selected action and its rationale—enters the conversation record. Second, the confidence tracking mechanism reduces unnecessary LLM calls: if the agent determines it needs more information, it can issue targeted tool calls rather than generating speculative responses that consume output tokens and add to the message history.

The net effect is a reduction in the *reasoning overhead ratio*:

$$R_{\text{overhead}} = \frac{T_{\text{reasoning}}}{T_{\text{actionable}}} \tag{3}$$

where $T_{\text{reasoning}}$ is the token count of reasoning traces and $T_{\text{actionable}}$ is the token count of actionable content (tool calls, user-visible responses). Unstructured chain-of-thought typically yields $R_{\text{overhead}} \in [2.5, 5.0]$; structured thinking reduces this to $R_{\text{overhead}} \in [0.8, 1.5]$.

4 Pre-Flight Context Checking

The pre-flight context checker is a dedicated node in the agent state graph that executes *before every LLM invocation*. Its purpose is to estimate the total token cost of the pending prompt and trigger compression or eviction when the estimate approaches the context capacity.

4.1 Token Estimation

The checker computes an approximate token count for each context component using a character-based heuristic:

$$\hat{T}(x) = \left\lceil \frac{|x|}{k} \right\rceil \tag{4}$$

where $|x|$ is the character count of the serialized content and k is a calibration constant (empirically $k \approx 3.8$ for mixed English text and JSON, $k \approx 3.2$ for code-heavy content).

The total estimated cost is:

$$\hat{T}_{\text{total}} = \hat{T}(\text{sys}) + \sum_{m \in \mathcal{M}} \hat{T}(m) + \sum_{t \in \mathcal{T}_{\text{bound}}} \hat{T}(t) + \hat{T}(\text{aux}) \tag{5}$$

where $\mathcal{M}$ is the message set and $\mathcal{T}_{\text{bound}}$ is the set of currently bound tool schemas.

4.2 Threshold-Based Triggering

The checker compares $\hat{T}_{\text{total}}$ against the model's context capacity $C_{\max}$ using a tiered threshold system:

- **Green zone** ($\hat{T}_{\text{total}} < 0.7 \cdot C_{\max}$): No action needed. The context has ample headroom for continued operation.
- **Yellow zone** ($0.7 \cdot C_{\max} \leq \hat{T}_{\text{total}} < 0.85 \cdot C_{\max}$): Advisory compression. Older messages are flagged as candidates for summarization, but the system can proceed without immediate action.
- **Red zone** ($\hat{T}_{\text{total}} \geq 0.85 \cdot C_{\max}$): Mandatory compression. The compression pipeline (Section 5) is triggered synchronously before the LLM call proceeds. Tool eviction may also activate if the tool budget is individually exceeded.

The threshold values are configurable per model, reflecting the observation that attention quality degrades at different rates across architectures. Models with strong long-context capabilities may tolerate higher utilization before compression is warranted.

4.3 Budget Rebalancing

When compression is triggered, the checker performs a budget rebalancing step. If one component is significantly under its allocation (e.g., few tools are bound, leaving C_{tools} mostly unused), the surplus is redistributed to components under pressure (typically C_{msg}). This dynamic rebalancing prevents premature compression when the overall budget has capacity but a single component has exceeded its nominal allocation.

The rebalancing follows a priority order: message history receives surplus first, followed by auxiliary context, then tool schemas. The output reserve and safety margin are never reduced, as they protect against estimation inaccuracies and ensure the model has sufficient room to generate its response.

5 Tiered Message Compression

When the pre-flight checker triggers compression, the message history undergoes a tiered processing pipeline that preserves recent context verbatim while compressing older exchanges into compact summaries.

5.1 Message Classification

Messages are classified into three tiers based on recency and type:

Tier 1: Protected messages. The most recent N message pairs (user message plus assistant response, including any tool calls and results within that turn) are retained verbatim. The value of N is adaptive: it is set to preserve at minimum the current conversational "thread"—the sequence of messages since the last topic change or significant context switch—with a floor of 3 pairs and a ceiling determined by the available message budget.

Tier 2: Compressible messages. Messages older

than the protected window but newer than any existing summary boundary are candidates for compression. These are grouped into contiguous blocks of configurable size and submitted for summarization.

Tier 3: System messages. System prompt messages and certain annotated messages (marked as "pinned" by the agent or the user) are never compressed. They persist in their original form regardless of age, ensuring that foundational instructions and critical constraints are always available to the model.

5.2 Summarization Strategy

Compression is performed by a dedicated LLM call (which may use a smaller, faster model than the primary agent model) with a specialized summarization prompt. The summarization prompt instructs the model to:

1. Preserve all factual decisions, commitments, and conclusions reached during the summarized exchanges.
2. Retain specific data values, identifiers, and proper nouns that may be referenced in subsequent turns.
3. Capture the user's stated preferences and constraints.
4. Note any tool calls that produced results still relevant to the ongoing task.
5. Discard conversational pleasantries, acknowledgments, and redundant restatements.

The resulting summary is injected into the message history as a system-role message with a distinctive marker (e.g., `[Conversation Summary]`) that allows downstream processing to identify compressed segments.

5.3 Compression Metrics

The effectiveness of compression is measured by the *compression ratio*:

$$\rho = \frac{\hat{T}(\text{original messages})}{\hat{T}(\text{summary})} \tag{6}$$

Table 2 reports observed compression ratios across different session types in production.

Table 2: Compression ratios by session type and message count.

Session Type	Messages	Ratio	Info Retained
Simple Q&A	10–20	3.2:1	94%
Data analysis	30–50	5.1:1	89%
Workflow orchestration	50–80	6.8:1	86%
Multi-tool exploration	80–120	8.4:1	82%

The "Info Retained" column reports a human-evaluated metric: independent raters assessed what fraction of actionable information from the original messages was preserved in the summary. The inverse relationship between compression ratio and information retention reflects an inherent tradeoff: longer sessions contain more redundancy (enabling higher compression) but also more diverse information (increasing the risk of salient details being omitted).

5.4 Incremental Compression

Rather than recompressing the entire history on each trigger, the system performs *incremental* compression. Each compression operation produces a summary with a sequence number and a "covers through" marker indicating the most recent message included in the summary. Subsequent compressions extend the summary boundary forward, summarizing only the messages between the previous boundary and the new protected window.

This incremental approach has two advantages: it avoids the latency of summarizing the full history on each trigger, and it produces a layered summary structure where earlier summaries capture broad themes while later summaries capture recent details. When multiple summary layers accumulate, a *meta-summarization* pass may consolidate them, though this occurs rarely in practice—most sessions require at most two to three compression cycles.

6 Dynamic Tool Context Management

Tool schemas represent a significant and variable component of context consumption. A single tool with complex nested parameter schemas can consume 400–800 tokens; an agent with 15 bound tools may dedicate 8,000–12,000 tokens to tool definitions alone.

The dynamic tool context management system, described in detail in our companion paper on semantic tool discovery [8], addresses this challenge through several mechanisms that we summarize here for completeness.

6.1 Session-Scoped LRU Cache

Discovered tools are maintained in a session-scoped cache with LRU eviction. When the number of cached tools exceeds a configurable capacity, the least recently used tools are evicted to maintain the tool budget within C_{tools}. Tools are rediscoverable via the `search_tools` meta-tool, ensuring that eviction does not permanently remove capabilities—it merely frees context space until the tool is needed again.

6.2 Always-Include Sets

A curated set of core tools is exempt from eviction. These represent universally useful capabilities—memory access, knowledge retrieval, artifact creation, planning—that the agent should always have available regardless of the current task. The always-include set consumes a fixed, predictable portion of the tool budget.

6.3 Coordination with Pre-Flight Checking

The pre-flight context checker (Section 4) monitors the tool budget alongside the message budget. When tool schemas exceed their allocation, the checker triggers tool-specific eviction independently of message compression. This separation prevents a scenario where growing tool requirements cause aggressive message compression, or vice versa.

The total tool cost is estimated as:

$$C_{\text{tools}}^{\text{actual}} = \sum_{t \in \mathcal{T}_{\text{always}}} \hat{T}(t) + \sum_{t \in \mathcal{T}_{\text{cached}}} \hat{T}(t) \qquad (7)$$

and the system ensures $C_{\text{tools}}^{\text{actual}} \leq B_{\text{tools}}$ where B_{tools} is the dynamically allocated tool budget from the rebalancing step. When this constraint is violated, the LRU eviction algorithm removes cached tools in order of staleness until the budget is satisfied.

7 Sub-Agent Orchestration

Complex tasks often require capabilities that span multiple domains: querying a database, processing the results, generating a visualization, and drafting an accompanying narrative. Handling all of these within a single agent context creates two problems: *context pollution* (tool schemas and intermediate results from one domain consume budget needed by another) and *attention dilution* (the LLM must attend to irrelevant context from completed subtasks).

Our architecture addresses this through an *agents-of-agents* pattern, where a parent agent orchestrates specialized sub-agents, each operating within its own isolated context scope.

7.1 Sub-Agent Types

The system defines several sub-agent specializations, each configured with a focused tool set and an optimized system prompt. Table 3 summarizes the primary sub-agent types and their context characteristics.

Table 3: Sub-agent types and their context characteristics.

Sub-Agent	Capabilities	Typical Budget
MCP Agent	External service integration via MCP protocol	32K
Tool Agent	General-purpose tool execution and chaining	48K
Workflow Agent	Multi-step workflow orchestration	64K
Analysis Agent	Data querying, transformation, statistics	48K
Composition Agent	Document and artifact generation	32K

7.2 Context Isolation

Each sub-agent receives a fresh context containing only the information relevant to its delegated task:

- A focused system prompt describing its specialization and operational constraints.
- The specific subtask description extracted from the parent agent's plan.
- Relevant data and parameters passed explicitly by the parent.
- A curated tool set limited to its domain of responsibility.

Critically, the sub-agent does *not* inherit the parent's full message history. This isolation prevents context pollution: a 60-turn parent conversation does not consume any of the sub-agent's context

budget. The sub-agent begins with a clean slate, focused entirely on its assigned subtask.

7.3 Result Aggregation

When a sub-agent completes its task, only its *result*—not its full execution trace—is returned to the parent agent. The result is a structured object containing the output data, a natural language summary of the work performed, and metadata about the execution (tools used, confidence level, any caveats or limitations).

This compression at the sub-agent boundary is highly effective: a sub-agent may internally consume 20,000 tokens of context across multiple tool calls and reasoning steps, but its result injected into the parent context may consume only 500–1,500 tokens. The compression ratio at the orchestration boundary is thus on the order of 15:1 to 40:1.

7.4 Recursive Orchestration

Sub-agents can themselves spawn further sub-agents, creating a recursive orchestration hierarchy. Each level of nesting provides an additional compression boundary. In practice, two levels of nesting (parent $\rightarrow$ sub-agent $\rightarrow$ sub-sub-agent) are sufficient for the most complex enterprise tasks we have encountered. A configurable depth limit prevents unbounded recursion.

The recursive pattern enables a form of *divide-and-conquer* context management: the total information processed across all agents in a session may far exceed any single context window, but each individual agent operates comfortably within its allocated budget. This mirrors the way human organizations handle complex tasks—through delegation and summarized reporting rather than requiring a single individual to hold all details simultaneously.

8 Middleware Architecture

Cross-cutting concerns—operations that affect multiple nodes in the agent state graph, such as context checking, memory recall, skill retrieval, and observability logging—are implemented through an Express-style middleware pipeline. This architectural pattern decouples context management logic from the core agent nodes, enabling modular composition and independent evolution of each concern.

8.1 Middleware Interface

Each middleware is a function with a standardized signature accepting the current agent state, the agent configuration, a context object carrying cross-middleware state, and a `next` callback that passes control to the subsequent middleware in the pipeline. Middleware may modify the state (e.g., injecting recalled memories), annotate the context (e.g., recording token utilization estimates), or short-circuit the pipeline (e.g., when a critical error is detected).

The pipeline executes in a defined order controlled by a numeric priority system. Higher-priority middleware (lower numeric priority values) executes first, enabling dependencies: the context checking middleware runs before memory recall, ensuring that the memory recall middleware knows its available token budget before retrieving observations.

8.2 Node-Specific Filters

Middleware can be configured with node filters that restrict its execution to specific nodes in the state graph. For example, the CAML recall middleware (Section 9) executes only before the `call_llm` node, where its recalled observations can influence the LLM's response. The context checking middleware executes before both `call_llm` and `think` nodes, as both involve LLM invocations that are subject to context limits.

This filtering mechanism prevents unnecessary computation: middleware that is irrelevant to a particular node is simply skipped, adding zero overhead to nodes that do not benefit from it.

8.3 Context Propagation

The context object passed through the middleware pipeline accumulates metadata and decisions from each middleware. Downstream middleware and the target node can read this accumulated context to inform their behavior. For instance, the preflight checker writes the current token utilization breakdown to the context object, and the memory recall middleware reads it to determine how many tokens it can allocate to recalled observations without exceeding the auxiliary context budget.

This propagation mechanism enables cooperative budget management: each middleware respects the decisions of its predecessors and constrains its own resource consumption accordingly, without tight coupling between middleware implementa-

tions.

8.4 Extensibility

The middleware architecture is designed for extensibility. New cross-cutting concerns—such as compliance logging, usage metering, or A/B testing of prompt variants—can be added as middleware without modifying existing agent nodes or other middleware. This separation of concerns has proven valuable in practice, enabling rapid iteration on context management strategies without disrupting the core agent logic.

9 Memory Integration

While compression preserves information *within* a session, enterprise agents also benefit from knowledge accumulated *across* sessions. Our architecture integrates with a hierarchical knowledge management system—the Kaman Memory Management System (KMMS)—that provides collective observations, learned patterns, and domain knowledge as external context augmentation. The design and capabilities of KMMS are described in a companion paper [9]; here we focus on its integration with the context management architecture.

9.1 KMMS Recall as Middleware

KMMS organizes knowledge in a hierarchical structure: individual observations are synthesized into patterns, patterns into principles, and principles into domain frameworks. The CAML (Collective Augmented Memory Layer) recall node operates as a middleware (Section 8) that, before each LLM invocation, queries KMMS for observations relevant to the current conversational context.

The recall process is subject to two hard constraints:

Token budget. The recall middleware is allocated a portion of C_{aux} (Equation 1). Retrieved observations are ranked by relevance and included greedily until the budget is exhausted. The budget is dynamic: it contracts when the message history is large (consuming more of the overall context) and expands when the conversation is short.

Latency bound. KMMS recall operates under a strict timeout (configurable, typically 2 seconds). If the memory service does not respond within the timeout, the agent proceeds without recalled observations, ensuring that memory integration never blocks the critical path of agent response generation. This graceful degradation is essential for production reliability.

9.2 Budget-Aware Injection

Retrieved observations are injected into the prompt as a clearly delineated system-role message block. The injection is *budget-aware*: the middleware estimates the token cost of each observation using the same heuristic as the pre-flight checker (Equation 4) and greedily includes observations in descending relevance order until the allocated budget is consumed.

The relevance ranking considers both semantic similarity to the current query and the observation's position in the KMMS hierarchy: higher-level observations (principles, patterns) receive a relevance boost because they encode distilled knowledge that is more broadly applicable.

9.3 Interaction with Compression

An important design consideration is the interaction between memory recall and message compression. When older messages are compressed into summaries, the agent may lose access to specific details that KMMS could resurface. In effect, KMMS serves as a *long-term memory complement* to the short-term memory provided by the message history: details that are compressed away from the active context may be independently recalled from KMMS if they become relevant again in a later turn.

This complementarity means that the system can compress more aggressively without proportional information loss, because KMMS provides a recovery pathway for compressed-away details. The overall information retention rate of the combined system (compression + KMMS recall) exceeds that of compression alone by an estimated 5–8 percentage points across production workloads.

10 Evaluation

We evaluate the adaptive context management architecture across three dimensions: context utilization efficiency, compression quality, and end-to-end task performance.

10.1 Context Utilization

We measure *context utilization* as the fraction of the model's context window consumed at each LLM

invocation:

$$U_k = \frac{\hat{T}_{\text{total},k}}{C_{\max}} \tag{8}$$

where k indexes the LLM call within the session.

Without context management, utilization grows linearly:

$$U_k^{\text{unmanaged}} \approx U_0 + k \cdot \bar{c}_{\text{turn}} \tag{9}$$

where $\bar{c}_{\text{turn}}$ is the average token cost per conversational turn. For a model with $C_{\max} = 128\text{K}$ and $\bar{c}_{\text{turn}} \approx 1{,}500$ tokens, the context is exhausted around turn 70.

With adaptive management, utilization follows a sawtooth pattern: it rises as new messages accumulate, drops when compression fires, and rises again. The compression threshold constrains the peak utilization:

$$U_k^{\text{managed}} \leq \theta_{\text{red}} = 0.85 \tag{10}$$

ensuring that the model always has at least 15% of its context window available for output generation and safety margin.

In production, we observe that managed sessions sustain utilization in the range 0.55–0.80 across sessions exceeding 100 turns, with compression typically triggering every 15–25 turns depending on the verbosity of tool results and the complexity of the user's requests.

10.2 Compression Quality

We evaluate compression quality through both automatic and human metrics.

Automatic metrics. We measure the *entity retention rate*—the fraction of named entities (proper nouns, identifiers, numerical values) from the original messages that appear in the summary. Across production workloads, the entity retention rate averages 0.91, indicating that the summarization process preserves the vast majority of factual anchors needed for continued coherent interaction.

Human evaluation. In a blinded study, annotators compared agent responses generated with full uncompressed history against responses generated with compressed history for the same sessions. On a 5-point Likert scale measuring response quality, compressed-history responses scored 4.2 versus 4.4 for full-history responses—a difference that did not reach statistical significance ($p = 0.12$, paired t-test, $n = 200$ evaluation pairs).

10.3 Task Completion

We measure task completion rate across a benchmark of enterprise workflows spanning data analysis, document generation, multi-system integration, and multi-step planning. Sessions are classified as "completed" if the agent successfully produces the requested output without human intervention beyond the initial request and any planned approval checkpoints.

Without context management, task completion rate degrades sharply for sessions exceeding 50 turns, dropping from 89% (short sessions, fewer than 20 turns) to 61% (long sessions, more than 50 turns) as context exhaustion causes the agent to lose track of earlier instructions and constraints. With adaptive context management, task completion remains stable at 85–88% across all session lengths up to 120 turns.

The 1–4% gap in completion rate between short and long managed sessions is attributable to inherent task complexity rather than context degradation: longer sessions correspond to more complex multi-step tasks with more opportunities for failure that are independent of context management.

10.4 Overhead Analysis

The context management system introduces computational overhead from token estimation, compression LLM calls, and middleware execution. The pre-flight checker adds negligible latency (under 5ms per invocation) as it performs only arithmetic on cached character counts. Compression, when triggered, adds 2–4 seconds of latency for the summarization LLM call; this is amortized over the 15–25 turns between compressions. KMMS recall adds 200–800ms when observations are found, bounded by the 2-second timeout.

In aggregate, the overhead is small relative to the primary LLM inference latency of 2–8 seconds per turn, and the benefits—sustained coherence, stable task completion, and extended session capability—substantially outweigh the cost.

11 Related Work

MemGPT [2] introduces an operating-system-inspired memory management system for LLM agents, with explicit "main context" and "external storage" tiers managed through function calls. Our architecture shares the insight that context is

a managed resource, but differs in a key design choice: where MemGPT relies on the LLM itself to manage memory through explicit tool calls, our system performs context management as an *infrastructure* concern via the pre-flight checker and middleware pipeline, reducing the cognitive load on the primary LLM and ensuring consistent behavior regardless of the model's self-management capabilities.

Reflexion [3] proposes a framework where LLM agents learn from prior failures through verbal self-reflection stored in an episodic memory buffer. Our structured thinking mechanism (Section 3) shares the emphasis on explicit reasoning artifacts, but extends it with confidence tracking and budget-aware lifecycle management that prevents reasoning traces from consuming unbounded context.

Cognitive Architectures for Language Agents (CoALA) [4] provides a theoretical framework for organizing agent memory into working memory, episodic memory, semantic memory, and procedural memory. Our architecture can be mapped onto this framework: the active context window corresponds to working memory, compressed message summaries to episodic memory, KMMS observations to semantic memory, and tool schemas to procedural memory. Our contribution is a practical realization of these theoretical categories with concrete token budget management.

LongAgent [5] addresses long-context processing through a multi-agent collaboration framework that distributes long documents across specialized "member agents" coordinated by a "leader agent." Our sub-agent orchestration (Section 7) employs a similar decomposition strategy, but focuses on *task decomposition* rather than *document decomposition*, with each sub-agent handling a functional specialization rather than a text segment.

StreamingLLM [6] addresses infinite-length context through attention sink preservation, maintaining initial tokens as anchors while sliding the attention window over recent tokens. Our tiered compression strategy achieves a similar effect at the application layer: system messages serve as persistent anchors (analogous to attention sinks) while older conversational messages are compressed rather than simply windowed, preserving information that pure windowing would discard.

Retrieval-Augmented Generation [7] provides the foundational paradigm for augmenting LLM context with retrieved information. Our KMMS integration extends RAG from document retrieval to *experiential retrieval*—surfacing collective observations and learned patterns rather than static documents—with budget-aware injection that treats retrieved content as a managed context consumer subject to the same resource constraints as other context components.

12 Conclusion

We have presented an adaptive context management architecture that enables LLM agents to sustain coherent, productive sessions far beyond the natural limits imposed by fixed context windows. The architecture treats the context window as a managed resource—analogous to physical memory in operating systems—and applies systematic techniques for estimation, compression, isolation, and augmentation.

The key design principles that emerge from this work are:

Proactive management. The pre-flight context checker prevents context exhaustion before it occurs, rather than reacting to truncation errors after the fact. This proactive approach ensures that the agent always operates within safe bounds, with predictable performance characteristics that operators can reason about and tune.

Tiered preservation. Not all context is equally valuable. Recent messages, system instructions, and pinned content warrant verbatim preservation, while older exchanges can be compressed with minimal information loss. The tiered approach respects this asymmetry and applies the appropriate level of compression to each category.

Isolation through decomposition. Sub-agent orchestration provides a natural mechanism for context isolation: each sub-agent operates in a fresh context with only task-relevant information, and only compressed results flow back to the parent. This enables the total information processed in a session to far exceed any single context window, a capability essential for complex enterprise workflows.

Infrastructure, not intelligence. Context management is implemented as infrastructure—

middleware, pre-flight checks, automatic compression—rather than as a capability the LLM must learn or perform. This reduces the cognitive burden on the primary model and ensures consistent behavior regardless of the underlying LLM's intrinsic context management abilities.

Several directions warrant further investigation. Predictive compression—triggering summarization based on projected future utilization rather than current utilization—could smooth the sawtooth utilization pattern and reduce latency spikes when compression fires at inconvenient moments. Semantic importance scoring for individual messages, using lightweight classifiers to identify high-salience content for preservation, could improve compression quality beyond what recency-based tiering achieves. Cross-session context warming, where summaries from prior sessions seed new sessions for recurring tasks, could reduce the cold-start overhead that currently requires users to re-establish context.

The architecture described in this paper is deployed in production as part of the Kaman platform, managing context for enterprise agent sessions across diverse workloads. The techniques presented here—pre-flight checking, tiered compression, sub-agent isolation, and middleware-based augmentation—represent a practical foundation for building LLM agents that operate reliably over extended interactions, a capability we consider essential for the maturation of agent-based enterprise AI.

References

[1] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[2] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[3] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *Proceedings of NeurIPS*, 2023.

[4] T. R. Sumers, S. Yao, K. Narasimhan, and T. L. Griffiths. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*, 2023.

[5] J. Zhao, Q. Zheng, Y. Fei, Z. Li, and J. Sun. LongAgent: Scaling language models to 128K context through multi-agent collaboration. *arXiv preprint arXiv:2402.11550*, 2024.

[6] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. Efficient streaming language models with attention sinks. In *Proceedings of ICLR*, 2024.

[7] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of NeurIPS*, 2020.

[8] Kaman AI Research. Semantic tool discovery and context-aware binding for large language model agents. Technical report, Kaman, 2026.

[9] Kaman AI Research. KMMS and CAML: Hierarchical collective memory for multi-agent enterprise systems. Technical report, Kaman, 2026.