Persistent Memory Architecture for Large Language Models:

Solving Context Drowning & Fine-tuning Without Modifying Weights

James Thomas
Independent Researcher
2026

Abstract

Large language models are stateless by design. Every new session begins from zero — no memory of the user, no understanding of ongoing goals, no behavioral continuity. The industry response has been to grow context windows. This paper argues that larger context windows do not solve the problem; they delay it and, in doing so, introduce a new failure mode: context drowning, the degradation of model reasoning and coherence as the active context becomes saturated with irrelevant or redundant information.

This paper presents a four-tier persistent memory architecture — Semantic, Procedural, Episodic, and Session memory — implemented in Kai, a local LLM assistant platform I am creating. The architecture is inspired by established models of human memory and is designed around one core principle: inject only what is relevant, when it is relevant. Through semantic routing, episodic compression, high-quality shutdown re-embedding, and a session continuity mechanism called the Good Morning Note, the system enables behavioral and contextual continuity across sessions without modifying model weights.

The central claim of this paper is that persistent, architecturally-routed memory is a more correct solution to the stateless problem than context window expansion, and that the pattern described here constitutes a form of non-parametric behavioral adaptation — the model learns from a user over time without any fine-tuning.

---

1. Introduction

When Tony Stark talks to JARVIS in the early Iron Man films, the interaction feels real not because JARVIS is intelligent in the abstract, but because JARVIS knows Tony. He knows his history, his preferences, his ongoing projects, and what they discussed yesterday. The magic isn't general intelligence — it's relational continuity.

Current large language models, despite their capabilities, lack this quality entirely. Every session is a clean slate. A user building a complex software system, a researcher working through a months-long project, an agent executing a multi-day workflow — all of them start over every time. The workaround most people reach for is the system prompt: hand-write a paragraph of context and paste it in at the start. This is functional, but it is manual, it doesn't scale, and it places the burden of memory on the user.

The mainstream technical response to this has been to increase context window size. Models now support windows of 128K, 200K, even 1M tokens. The assumption is that if you can fit everything in the context, you don't need memory architecture. This assumption is wrong.

Context windows are not free. They have computational cost, latency cost, and — most importantly — a coherence cost. When a model is given a 200K token context, it is not equally attending to all of it. Attention is distributed, relevance degrades with distance, and the model's effective reasoning window is significantly smaller than its technical limit. Flooding the context with everything produces a phenomenon I call context drowning: the model loses the thread, contradicts itself, misses what matters, and produces responses that feel disconnected from the actual goal. Bigger context is not better context.

The solution is not to expand the container. The solution is to be selective about what goes in it. Human memory does this naturally — sensory input is filtered, encoded by significance, and retrieved by relevance. We do not recall everything at once. We recall what the current moment calls for. A well-designed memory architecture for LLMs should work the same way.

This paper describes a four-tier memory system built for Kai, a local LLM assistant. The system stores information in four distinct layers — semantic facts, procedural rules, episodic summaries, and volatile session data — and injects only relevant subsets of that information into the model's context window on each turn. It introduces a session continuity mechanism (the Good Morning Note) that orients a fresh model instance at startup, creating the subjective experience of an unbroken conversation across restarts.

The broader argument is this: persistent memory architecture, not larger context windows, is the path toward agentic LLM systems that actually work. And as a direct consequence, it enables a form of behavioral adaptation — the model becoming more tailored to a specific user over time — without any modification to the model's weights.

2. Background and Motivation

2.1 The Stateless Problem

A transformer-based language model has no persistent state between inference calls. Each call receives a context window — a sequence of tokens — and produces a completion. When the call ends, nothing is retained. The model does not remember. From an engineering perspective, this is a feature: statelessness enables horizontal scaling and simplifies inference infrastructure. From a user experience and agent capability perspective, it is a fundamental limitation.

The standard mitigation is to prepend conversation history to each new call, effectively giving the model access to prior turns within a single session. This works until the session history exceeds the context window, at which point the oldest turns are dropped. Continuity across sessions — even sequential ones — requires external scaffolding that the LLM ecosystem has largely not standardized.

2.2 Context Window Growth Is Not the Answer

The prevailing response from model developers has been to increase maximum context length. This addresses the within-session truncation problem but introduces its own failure modes. Research on long-context transformers has consistently shown degradation in retrieval accuracy and reasoning quality as context fills — particularly for information buried in the middle of a long context (the "lost in the middle" effect, Liu et al., 2023). Beyond a certain threshold, more tokens in context actively hurts performance on tasks requiring precise recall or multi-step reasoning.

There is also the computational reality: processing a 200K token context costs orders of magnitude more than processing a 4K token context. For latency-sensitive applications and for local deployments on consumer hardware, this is not a viable scaling strategy. A system that carefully selects a 3K token context block will consistently outperform one that dumps 50K tokens of loosely relevant history.

2.3 Human Memory as a Design Model

Cognitive science has long distinguished between types of memory that operate on different timescales and serve different functions. The Atkinson-Shiffrin model and its descendants identify sensory, short-term, and long-term memory as distinct systems. Tulving's subsequent work distinguishes semantic memory (general facts and knowledge), episodic memory (autobiographical events in time), and procedural memory (how to do things). Working memory theory, developed by Baddeley and Hitch, describes a limited-capacity system that temporarily holds and manipulates information relevant to the current task.

These distinctions map naturally onto the problem of LLM memory design. What does the model need to know that is stable and factual? What behavioral patterns should persist? What past events are relevant to the current query? What is only relevant for the duration of this session? Building memory layers around these distinct functions is a more principled approach than treating all information as equivalent and scaling the container.

2.4 Existing Approaches and Their Limits

Current memory implementations for LLMs fall broadly into three categories: full conversation history (linear, hits token limits, no selectivity), retrieval-augmented generation (RAG, designed for documents rather than conversational memory, no behavioral layer), and fine-tuning (modifies weights, expensive, requires data curation, cannot adapt in real time). None of these alone produces the kind of continuous, adaptive, multi-session memory that enables genuinely relational AI assistance. The architecture described in this paper attempts to fill that gap.

---

3. Architecture

The Kai memory system is organized into four tiers, each storing a different kind of information with a different persistence model, retrieval mechanism, and injection strategy. A fifth cross-cutting component — the Memory Router — manages which tiers are queried and what subset of their content is injected into the context window on each turn.

3.1 Semantic Memory — The Stable Identity Layer

Semantic memory stores factual, stable information about the user: their name, role, location, hardware specifications, stated preferences, tools they use, and similar identity-grounding facts. These are key-value pairs with associated confidence scores and source attribution.

Facts are extracted automatically from user messages via pattern matching (e.g., "my name is X" → user_name, "I prefer X" → preference_N) and can also be written explicitly. Each fact records whether it was user-stated, inferred from a message, or observed from system context. When the same key is written again, it is upserted rather than duplicated, so facts remain current without accumulating stale duplicates.

The confidence score is meaningful: facts stated explicitly by the user carry 1.0 confidence; inferred facts carry lower scores. This metadata is available for routing and filtering decisions. Source attribution enables transparency — the system knows the difference between "the user told me this" and "I inferred this from context."

Critically, semantic memory is not injected wholesale. With hundreds of facts potentially stored, dumping all of them into every prompt would itself cause context bloat. The Memory Router (Section 3.5) filters semantic facts to only those relevant to the current query domain before

injection.

3.2 Procedural Memory — The Behavioral Layer

Procedural memory stores behavioral rules: how the model should communicate, at what length, with what tone, and what its defaults are for initiative and system actions. These rules persist across sessions and constitute the personality and operating style of the assistant.

Default procedural rules include tone (*direct, honest, a bit of edge*), response length (*brief by default, detailed when needed*), initiative (*suggest things proactively*), and system action policy (*report first, act second, always confirm*). These can be overwritten by the user at any time and persist.

Unlike semantic memory, procedural rules are always injected into every context block — they are never filtered by the router. The reasoning is that behavioral rules are not query-dependent. They are the foundation of the assistant's character and should be available on every turn regardless of what is being asked. In this sense, procedural memory functions like a persistent system prompt enhancement that the model itself helps maintain.

This layer is the closest analog to what fine-tuning accomplishes for behavioral adaptation. Fine-tuning modifies weights to make a model respond in a particular way to particular inputs. Procedural memory achieves a functionally similar result by making behavioral rules explicit, persistent, and always present in context. The model does not need to have learned the behavior — it is instructed to apply it, and those instructions survive every session reset.

3.3 Episodic Memory — The History Layer

Episodic memory is the most architecturally complex tier. It stores timestamped records of past interactions — not as raw conversation transcripts (which are expensive), but as compressed summaries and extracted knowledge fragments. The design goal is to preserve the meaning of past interactions while minimizing token cost.

3.3.1 Entry Types

The episodic store uses four entry types:

- turn — raw conversation turns, stored but not embedded
- archive — compressed summaries of past sessions, embedded for vector search
- learned — knowledge fragments extracted from conversations, embedded for vector search
- milestone — user-defined significant events

Raw turn entries are not embedded. Embedding every turn would create significant Ollama queue pressure and provide marginal benefit, since the raw turns are eventually compressed. Only archive and learned entries — the distilled, semantically rich content — are embedded and indexed for retrieval.

3.3.2 Compression

When session history exceeds a character threshold (approximately 12,000 characters / 3,000 tokens by default), a compression process fires. The system identifies the oldest turns in the session history, passes them to the language model with a summarization prompt, and stores the resulting summary as an archive entry with a vector embedding. The raw turns are then deleted from the episodic database, keeping the store lean. Full verbatim transcripts are preserved separately in an archive table, available for exact recall but not included in routine context injection.

This design mirrors the way human memory consolidates experience over time. Raw sensory detail is not preserved indefinitely; meaning is extracted and retained in more compressed, generalized form. The transcript table plays the role of the hippocampus in long-term storage — the full record exists, but routine cognition works from compressed representations.

3.3.3 Knowledge Extraction

Every third conversation turn, the system sends the recent exchange to the language model with a knowledge extraction prompt, asking it to identify anything worth remembering as a standalone fact. The result is stored as a learned entry with an embedding. This process is rate-limited and non-blocking — it runs in a background thread after the user's response has already been delivered. Pre-filtering skips trivial exchanges to avoid noise.

This is the mechanism by which the system develops a progressively richer model of the user and their work over time. Each extracted knowledge fragment is a small permanent deposit into the episodic store.

3.4 Session Memory — The Volatile Layer

Session memory is intentionally not persisted. It holds volatile, instantaneous state that is relevant within a session but meaningless across sessions: current CPU and RAM utilization, GPU temperature, disk usage, active process counts. This data is extracted from model responses via pattern matching and held in an in-memory dictionary for the duration of the session.

The explicit non-persistence of session state is an architectural decision with meaningful consequences. It prevents pollution of the long-term stores with data that changes every minute. It also ensures a clean separation between what the model knows enduringly versus what it knows right now. A system that stored CPU utilization to disk would, over time, accumulate thousands of stale readings that would either clutter context injection or require ongoing cleanup. By design, session state evaporates on restart.

3.5 The Memory Router — Selective Context Injection

The Memory Router is the component that makes the rest of the architecture practical. With hundreds of semantic facts and dozens of episodic entries potentially in the store, injecting

everything on every turn would recreate the context bloat problem the architecture is designed to solve.

The router defines seven semantic domains:

Domain	Contents
identity	Name, role, location
preferences	Likes, dislikes, hobbies, stated interests
hardware	PC specs, RAM, GPU, CPU, disk
documents	Uploaded files, RAG content
history	Past conversations, episodic entries
notes	Reminders, things to remember
campaign	D&D campaign context (DM mode)

At startup, each domain description is embedded with a lightweight CPU-based model (Xenova/bge-small-en-v1.5, 384 dimensions, ~5ms per embedding). On each user turn, the query is embedded and its cosine similarity to each domain vector is computed. The top-K domains above a relevance threshold are activated. Only semantic facts and episodic entries belonging to activated domains are retrieved and injected.

This means a question about code gets *hardware* and *history* domains, not *preferences* or *campaign*. A question about music gets *preferences* but not *hardware*. The context block shrinks to what the current query actually needs.

A memory directory — a short summary listing what data exists in each store — is always injected regardless of routing. This gives the model awareness of what is available without loading the actual data. The model can request additional context if needed; the directory is the index, not the content.

3.6 Two-Tier Embedding Strategy

The system uses two embedding models with different performance profiles:

- Fast model (CPU): Xenova/bge-small-en-v1.5, 384 dimensions, ~5ms per query, no VRAM usage. Used for all live operations: query routing, real-time episodic search, document chunking during upload.
- High-quality model (GPU): qwen3-embedding:4b via Ollama, 2560 dimensions. Used exclusively at shutdown, when the chat model has been unloaded and GPU VRAM is free.

At shutdown, the system re-embeds all non-turn episodic entries and all RAG chunks with the high-quality model, storing results in shadow tables. Future searches use these higher-

dimensional vectors when available, improving semantic precision.

This strategy eliminates VRAM contention entirely. On 8GB consumer cards — the most common local deployment target — running a 7B chat model alongside a 4B embedding model simultaneously is not feasible. By offloading high-quality embedding to shutdown, the system gets the benefit of large embeddings without competing for resources during live inference.

3.7 Context Block Assembly

The final context block injected into the system prompt on each turn is assembled in a defined order:

- [IDENTITY] — Kai's persona (always)
- [MEMORY DIRECTORY] — Summary of available stores (always, ~200 chars)
- [PROCEDURAL] — Behavioral rules (always)
- [SEMANTIC] — Filtered facts from routed domains
- [EPISODIC] — Archived summaries (only if history domain active)
- [SESSION] — Volatile runtime stats (always, minimal)
- [UPLOADED FILES] — Document inventory (always)
- [RAG CHUNKS] — Relevant document excerpts (only if documents domain active)
- [CAMPAIGN] — Campaign context (DM mode only)

Episodic, RAG, and campaign retrieval run in parallel via a thread pool. If the assembled block exceeds a character budget (10,000 chars standard, 14,000 chars in DM mode), the system trims oldest episodic entries first, then RAG chunks if still over limit. Procedural rules and the memory directory are never trimmed.

---

4. Session Continuity: The Good Morning Note

Even with a fully populated memory store, a fresh model instance starts cold. The semantic and episodic data exists, but the model has no sense of the current moment — no orientation to where things were left off, no awareness of what is most pressing.

The solution is a mechanism called the Good Morning Note. At the end of each session, before the model is unloaded, the system prompts it to write a note to its next instance. The note is approximately 1,000 characters and covers:

- What was most important about this session
- Where the current project or conversation thread stands
- What should be loaded or considered first on next startup
- Any context the next instance should know immediately

On the next startup, this note is injected immediately after the system prompt, before any user message. The model wakes up oriented. It knows what it was doing, what matters, and where to pick up. The user does not have to re-explain themselves.

The Good Morning Note solves a specific problem that structured memory retrieval alone cannot. Memory retrieval is reactive — it responds to queries. The Good Morning Note is proactive — it primes the model with continuity before any query arrives. It is the difference between a colleague who reads the notes before the meeting and one who walks in cold.

The token cost is minimal. One thousand characters is roughly 250 tokens — a trivial overhead relative to a 10,000-character context block. The continuity benefit is significant. In practice, sessions following a Good Morning Note show consistent orientation from the first response, with the model referencing the prior session's state naturally and without being asked.

The mechanism is also self-improving. As the model writes more Good Morning Notes over time, it develops a better implicit model of what information is worth preserving across sessions. This is a form of meta-cognitive learning — the model learns to summarize for its future self — that emerges from the pattern without any explicit training.

---

5. Non-Parametric Behavioral Adaptation

Fine-tuning is the standard answer to the question of making a model behave differently for a specific user or use case. You collect examples, you train, you adjust weights. This works, but it has real costs: compute, time, expertise, and inflexibility — a fine-tuned model is a static artifact that cannot adapt in real time.

The architecture described in this paper achieves a functionally similar result through a different mechanism. Over the course of interactions, the system:

- Accumulates stable facts about the user (semantic memory)
- Develops and refines behavioral rules (procedural memory)
- Builds a record of relevant past interactions (episodic memory)
- Extracts and retains knowledge fragments from every conversation (knowledge extraction)
- Writes forward-facing continuity notes across sessions (Good Morning Note)

The cumulative effect is that the model, given this context block, behaves in a way that is increasingly calibrated to this specific user. It knows their name, their role, their preferences, their ongoing projects, their communication style expectations, and their history. It does not need to re-learn any of this — it is injected.

This is non-parametric behavioral adaptation: the adaptation lives in the data layer, not the weight layer. The base model is unchanged. Any model that can follow the context block will

exhibit the adaptive behavior. This has several practical advantages:

- Model upgrades are non-destructive — swapping in a new base model preserves all accumulated memory
- No training infrastructure required — the adaptation happens at inference time
- Real-time adaptation — new facts are stored immediately and available on the next turn
- Transparent and editable — the memory store can be inspected, corrected, and curated

The limitation is honest: this is not the same as fine-tuning. Fine-tuning changes the model's priors — its default behavior even without supporting context. This architecture changes the model's behavior given context. A model with no memory context will not behave in the adapted way. The adaptation is context-dependent. For applications where the memory system is always present, this distinction is academic. For applications requiring adapted behavior on arbitrary cold prompts, fine-tuning remains necessary.

For the majority of agentic and assistant use cases, however, the context is always present. The memory system is always injected. The distinction dissolves in practice.

---

6. Context Drowning and Its Solution

Context drowning is what happens when a model's active context is saturated beyond its effective reasoning capacity. It is not a hard failure — the model does not crash. It degrades. Responses become less coherent, contradict earlier statements, miss key details, or answer the wrong question. The model is overwhelmed by the volume of what it has been asked to hold.

This is not a theoretical problem. Anyone who has worked with long-context models on complex, ongoing tasks has experienced it. The model loses the thread. The continuity that was present at the start of a long conversation is gone by turn forty.

Growing context windows delays context drowning but does not solve it. If the context window doubles, the drowning point doubles too. The actual problem — that the model is being given more information than it can effectively reason over — is not addressed by giving it more room to drown in.

The architecture described in this paper addresses context drowning directly through three mechanisms:

6.1 Semantic Routing

By activating only relevant memory domains per query, the system ensures that the context block contains information the current turn actually needs. A 3,000-token context of highly relevant information produces better responses than a 30,000-token context of mixed-relevance content. Routing is the primary defense against drowning.

6.2 Episodic Compression

Past interactions are summarized, not replayed verbatim. A 10,000-word conversation compresses to a 200-word summary that captures its meaning. The model gets the knowledge without the tokens. Compression allows the episodic store to grow indefinitely without increasing context injection cost.

6.3 Budget-Aware Trimming

The context assembly process respects a hard character budget. If assembled content exceeds the limit, oldest episodic entries are dropped first, then RAG chunks. Procedural rules, the memory directory, and the Good Morning Note are protected. The system guarantees a context block of bounded size regardless of how large the memory stores become.

Together, these mechanisms mean that context size stays roughly constant over time even as the accumulated knowledge grows. A user with two years of interaction history injects approximately the same number of tokens per turn as a user with two weeks of history. The knowledge is there when needed; it does not burden every turn.

---

7. Applications and Implications

7.1 Agentic Systems

Agents deployed to accomplish long-horizon goals are exactly the systems most harmed by statelessness and context drowning. An agent working on a multi-day research task, a software project, or a business process needs persistent awareness of what has been done, what decisions have been made, and what comes next. The architecture described here provides this directly. The Good Morning Note is particularly relevant for agentic systems: it is a structured handoff between agent instances, carrying forward the thread of work without requiring the receiving instance to reconstruct context from scratch.

7.2 Minecraft Bot Agents (Mindcraft Context)

A concrete application of this architecture is in game-based AI agents such as those developed in the Mindcraft project, where language models control Minecraft bots. Such agents currently face statelessness between runs and context saturation during long play sessions. Applying this memory architecture would allow a bot to accumulate knowledge about the game world (semantic memory), develop consistent behavioral tendencies (procedural memory), remember significant past events and decisions (episodic memory), and orient quickly on respawn (Good Morning Note). The result would be an agent that develops, over multiple sessions, an increasingly sophisticated and contextually aware approach to the game — without any model retraining.

7.3 Personal AI Assistants

The motivating use case for this architecture is the personal assistant. A user who interacts with an assistant daily accumulates shared history, shared vocabulary, shared goals. The memory system allows that accumulation to be real — not simulated by repeating context on every turn, but genuinely persistent and selectively retrievable. The assistant that remembers what you were working on last Tuesday, knows you prefer brief responses, and picks up mid-thought is qualitatively different from the one that starts fresh every time.

7.4 Implications for the Field

The broader implication of this architecture is that the framing of the stateless problem as a context window problem is incorrect and has been leading the field toward a locally optimal but globally suboptimal solution. Context window scaling is engineering. Memory architecture is epistemology — a question of what the model should know, when, and how that knowledge should be organized.

The cognitive science foundations of this architecture are not new. The insight that different kinds of information should be stored and retrieved differently has been understood for decades. The application of these principles to LLM system design is underexplored, and the results, as demonstrated in Kai, are significant.

---

8. Limitations and Future Work

This architecture, as implemented, has several limitations worth acknowledging.

The knowledge extraction and compression steps depend on the language model's summarization quality. A model that summarizes poorly will store poor episodic entries, degrading retrieval quality over time. The system is only as good as the model's ability to distill meaning from conversation.

The memory router's domain classification is based on cosine similarity between a query embedding and static domain description embeddings. This is effective for broad classification but can fail on ambiguous or cross-domain queries. More sophisticated routing — including multi-label classification or learned domain boundaries — is a clear area for improvement.

The Good Morning Note is currently written by the model at shutdown. Its quality depends on the model's ability to introspect on what was most important in the session. Structured prompting improves this, but the note is not guaranteed to be optimal. Future work could explore structured note formats that constrain the model's output to high-signal fields.

Finally, this paper presents the architecture and its reasoning, but does not include controlled benchmarks comparing it against baseline (no memory) and competing approaches (full context injection, RAG-only). Quantitative evaluation of coherence, retrieval accuracy, and behavioral

consistency across sessions is important future work.

---

9. Conclusion

This paper has presented a four-tier persistent memory architecture for large language models, implemented in the Kai assistant platform. The architecture draws on established cognitive science models of memory to organize information into semantic, procedural, episodic, and session tiers, each with distinct storage, retrieval, and injection characteristics.

The central contributions are: a semantic routing mechanism that selects only relevant memory domains per query, an episodic compression pipeline that preserves meaning without accumulating tokens, a two-tier embedding strategy that maximizes retrieval quality without sacrificing live inference performance, and the Good Morning Note — a session continuity mechanism that orients fresh model instances with minimal context overhead.

Taken together, these mechanisms enable non-parametric behavioral adaptation: the model becomes more tailored to a specific user over time without any modification to its weights. They also constitute a direct solution to context drowning, keeping context blocks bounded and relevant even as accumulated knowledge grows without limit.

The context window will keep growing. That growth will keep being presented as progress. But the real problem is not the size of the container — it is the absence of a principled theory of what should go in it. Memory architecture is that theory. The pattern described in this paper is one version of it. The field needs more.

References

Atkinson, R. C., & Shiffrin, R. M. (1968). Human memory: A proposed system and its control processes. Psychology of Learning and Motivation, 2, 89-195.

Baddeley, A. D., & Hitch, G. (1974). Working memory. Psychology of Learning and Motivation, 8, 47-89.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Hopkins, M., Liang, P., & Manning, C. D. (2023). Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics, 12, 157-173.

Tulving, E. (1972). Episodic and semantic memory. In E. Tulving & W. Donaldson (Eds.),

Organization of Memory (pp. 381-403). Academic Press.

Tulving, E. (1985). Memory and consciousness. Canadian Psychology, 26(1), 1-12.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30.

Wang, L., Chen, X., Deng, X., Wen, H., You, M., Liu, W., Li, Q., & Sun, J. (2024). Prompt engineering for large language models: A survey. arXiv preprint arXiv:2310.14735.