Understanding Is Getting the Context Right

An Operating System for Language Models

Michael Bonsignore — February 2026

The AI industry is in an arms race to build bigger context windows. OpenAI, Google, and Anthropic are all racing toward million-token windows, as if the problem with language models is that they can't hold enough text. But this gets the problem exactly backwards. The issue isn't the size of the context window. It's the quality of what's in it.

Give a model the wrong context and it produces nonsense. Give it the right context and it looks brilliant. The model hasn't changed. The context did. What we've been calling "understanding" is largely a function of context quality. Or to put it more sharply: context selection, not context length, is the dominant scaling law for reasoning quality.

This means the most important problem in language model engineering isn't building a bigger window. It's building a system that keeps the window clean.

The Pollution Problem

Right now, every major language model manages context the same way: it takes the entire conversation history, packs it sequentially into the context window, and hopes for the best. As a conversation grows, the window fills with noise. Failed attempts sit

alongside successful ones. Ten web search results persist even though only one was relevant. Tool call metadata takes up tokens long after the output was consumed. Verbose exchanges from early in the conversation crowd out the precise information needed for the current task.

This is context pollution, and it degrades every long conversation. The model doesn't get dumber over time — it gets buried in noise. Every token of irrelevant context makes the relevant tokens harder to attend to. The signal-to-noise ratio drops, and the model's performance drops with it. Long context without curation is effectively adversarial noise. Transformers don't degrade gracefully with junk — attention spreads probability mass over everything, diluting the signal that matters.

There's a more insidious form of this too. Contradictory information can persist in the window — an earlier statement that was later corrected, a hypothesis that was abandoned, a number that was revised. The model has no way to "unsee" something. Both the wrong answer and the right answer sit there together, and the model has to figure out which one to trust. Sometimes it picks wrong.

The current solution to all of this is to make the window bigger. But a bigger window full of noise is still full of noise. A well-curated 32,000-token window would outperform a polluted 128,000-token window because every token in it would be signal.

The Architecture

The fix is conceptually simple: separate the job of managing context from the job of reasoning. Right now we force one model to think, remember, search, compress, summarize, and reason — all simultaneously. That's absurd from a systems perspective. It's like asking a CPU to manage its own cache hierarchy while executing instructions.

Instead, you use two agents. The first is the reasoning agent, the large foundation model doing the actual work. The second is a curator agent, a smaller model whose only job is managing what goes into the context window.

The curator doesn't need to be a frontier model. Something like Llama 3 running locally would work. Its job is lightweight semantic reasoning — evaluating relevance, classifying threads, tracking references — but it avoids the kind of long-chain

generation that requires a large model. It runs continuously, watching the conversation, evaluating what matters, and assembling the input for each turn of the reasoning agent. In cases where relevance judgments require deeper understanding — recognizing that a hardware note from forty messages ago is relevant to a current training divergence problem, for example — the curator may need to be a medium-sized model rather than a tiny one. But it's still far cheaper than the reasoning agent, and the system is designed so that imperfect curation degrades gracefully rather than catastrophically.

This separation of concerns mirrors well-established patterns in systems engineering. CPU cache hierarchies, database query planners, operating system schedulers — all of these separate the management of information from the processing of information. What this paper proposes is essentially memory virtualization for language models. The transformer is the CPU. The curator is the operating system.

More precisely, the curator is a feedback controller minimizing context error — where context error means the gap between what the reasoning agent needs in its window and what's actually there. It's simultaneously minimizing false inclusion (noise that wastes tokens and dilutes attention), false exclusion (relevant information that's missing), and total token cost. Every user correction tightens the loop. This isn't a heuristic — it's an optimization process.

This means there is no longer a single object called "the context window" that just accumulates. Instead, there's a chat — the complete record of everything said — and a freshly constructed input that the curator assembles for each turn. The reasoning agent never sees the raw chat. It sees what the curator decides it should see.

Threads, Not Logs

The first structural change is that the conversation history is no longer a flat sequential log. It's organized into threads — linked lists of related prompt-response pairs. When you spend three exchanges debugging a function, that's a thread. When you pivot to discussing architecture, that's a new thread. When you come back to the debugging, that continues the first thread.

This is arguably the most important structural insight in the entire architecture. Flat logs are fundamentally wrong because human reasoning doesn't work sequentially. It branches, revisits, suspends topics, and resumes them later. A flat log destroys that structure. What you actually have is a conversation DAG — a directed acyclic graph — not a sequence.

This matters because relevance is contextual. When you ask about architecture, the debugging thread is probably noise. But the linked list structure preserves it intact so it can be pulled back in if needed. You're not retrieving isolated chunks via embedding similarity — you get the actual flow of reasoning within a topic, with all its context preserved. Embedding-based chunk retrieval loses reasoning trajectory. Thread retrieval preserves it. That's a real and significant advantage.

Thread detection is part of the curator's continuous job. It watches the conversation, recognizes when a new prompt continues an existing thread versus starting a new one, and maintains the linked list structures accordingly. This isn't trivial — conversations drift gradually, interleave topics, and make implicit references across threads. The curator will need a combination of semantic similarity, temporal proximity, and its own judgment to make good splits. But it doesn't need to be perfect, because the user interface provides a correction mechanism, and the system learns from those corrections over time.

Two Manifests

The curator maintains two manifest objects that structure every input to the reasoning agent.

The first manifest is a topic index — a compact summary of every thread in the entire conversation history. This should be small, probably under a hundred tokens. It lives at the top of every input along with a brief instruction: these are all the things that have been discussed. You probably don't need all of them, but if you think you need one and it's not below, you can ask for it.

This gives the reasoning agent awareness of the full scope of the conversation without paying the token cost. It knows what exists. It can make informed requests for context it doesn't have. This is the difference between losing your keys and knowing exactly which drawer they're in but choosing not to open it yet.

There is a limitation here worth acknowledging. Language models are notoriously poor at recognizing when they lack information — they tend to proceed with partial context or hallucinate rather than asking for more. The topic index helps significantly by at least making the model aware of what exists, which is a large step up from the current situation where it has no idea what it's missing. But in practice, the system may also benefit from explicit mechanisms like uncertainty estimates or forced retrieval checks at the reasoning agent level to supplement the manifest-based approach.

The second manifest is the active context payload. This is what the curator assembles for each turn. It evaluates the current prompt, determines which threads are relevant, follows those linked lists, and constructs the curated context. This is where the real curation happens — the curator decides what the reasoning agent needs to see right now.

Every turn, the reasoning agent receives the lightweight topic index plus the curated relevant history. Nothing else. No accumulated noise, no stale context, no irrelevant tool outputs. Just signal.

Continuous Evaluation and Metadata

The curator isn't a request-response service that wakes up when a new prompt arrives. It's a continuously running process that watches the conversation and constantly re-evaluates what matters. By the time a new prompt arrives, the curator already has an opinion about what's relevant. Assembling the second manifest for a new turn isn't a cold start — it's the latest snapshot of an ongoing evaluation.

Part of this continuous work is maintaining metadata on everything in the chat. Right now, language models treat user claims, web search results, hallucinated inferences, and tool outputs as identical tokens. That's epistemically incoherent. A user-stated fact and an unverified web search snippet should not carry the same weight, but in a flat context window, they do.

Provenance tracking fixes this — knowing that a claim came from a web search versus something the user stated versus something the reasoning agent inferred versus output from a tool call. The reasoning agent can weigh information differently based on where it came from. And the curator can use provenance to make eviction decisions — tool call metadata is probably more disposable than a user-stated requirement.

Other useful metadata includes confidence levels, how many times a thread has been referenced, whether something was corrected or superseded later, and timestamps for recency weighting. This moves the system from treating context as a bag of text to treating it as structured knowledge — closer to how cognition actually works.

The Global Repository

Without persistent storage, the curator is still bounded by a single session. The whole architecture falls apart if the conversation history disappears when the chat ends.

The solution is a global repository where everything lives permanently — every thread, every manifest, all the metadata and provenance. The repository is the ground truth. A chat session is just a temporary working window into the repository.

When you start a new conversation and mention something you worked on last week, the curator already has those threads in the repository. It pulls in the relevant linked lists, assembles the second manifest, and the reasoning agent picks up where you left off without you having to re-explain anything.

This reframes what "memory" means for a language model system. Current approaches to memory extract little factoids — the user lives in a certain place, the user has a certain preference. That's lossy to the point of being almost useless for real work. The repository preserves actual reasoning chains, full threads with context, the provenance of where ideas came from. That's real memory.

The Knowledge Graph

The repository isn't just storage. Given enough time and use, it becomes a knowledge graph — not because anyone designed it that way, but because the structure makes it inevitable.

The curator is already tracking when threads reference each other. When a conversation about attention mechanisms connects to an earlier conversation about convolution, that's an edge. When a debugging session reveals something relevant to architecture decisions, that's an edge. When a user marks two threads as related or adds metadata linking them, that's an edge. Over time, these connections accumulate into a rich graph that represents the shape of the user's thinking across every conversation they've ever had.

A tool like Obsidian is practically built for this. Each thread becomes a note, edges are bidirectional links between notes, and the user gets a navigable, visual map of their entire body of work with an LLM. Right now, every major LLM interface gives users a flat list of chats sorted by date. Hundreds of hours of deep work, interconnected ideas, evolving understanding — reduced to a chronological list with no structure. That's a staggering waste. The user builds an enormous body of interconnected knowledge through conversation, and the interface gives them no way to see it, navigate it, or build on it.

The knowledge graph also makes the curator significantly smarter. Instead of evaluating thread relevance in isolation, the curator can traverse the graph. When the user mentions attention mechanisms, the curator doesn't just pull that one thread — it follows edges to related threads about geometric interpretations, about MLP replacements, about the original paper that started the line of inquiry. Graph traversal gives the curator a richer model of relevance than any flat index or embedding similarity search could provide.

And this is where the value compounds over time. A new user gets basic curation. A user with six months of graph-connected history gets something qualitatively different — a curator that understands not just what topics exist but how they relate, which lines of inquiry are still active, and what background context enriches the current conversation. The knowledge graph turns the system from a tool with memory into something closer to a research partner.

Local First

It's worth emphasizing that this entire architecture can and should run locally. The curator model, the repository, the knowledge graph, the metadata — none of this needs to live on someone else's server.

A consumer-grade machine with a decent GPU — or even an M1 Mac Mini running Ollama — can handle a model like Llama 3 comfortably. The curator's job is lightweight inference: short evaluations of relevance, classification, metadata updates. It doesn't need a data center. The repository is just structured data on disk. The knowledge graph is a folder of linked files. The user interface is a local application.

The only component that might reasonably be remote is the reasoning agent itself, if the user wants to work with a frontier model like Claude or GPT-4. But even that is optional — users running larger models locally through Ollama or similar tools can keep the entire pipeline on their own hardware.

This matters for practical reasons. Local inference is fast, free after hardware cost, and has no rate limits. But it also matters for deeper reasons. The knowledge graph that builds up over months of use represents something genuinely personal — the structure of how someone thinks, what they've explored, how their ideas connect. That should belong to the user, live on the user's machine, and persist regardless of what happens to any particular AI company's service. The current model, where all of your conversation history lives on a corporate server and you get to keep a flat export at best, is not good enough for a system that's meant to serve as a long-term intellectual partner.

The User Interface

Everything described so far — the threads, the manifests, the metadata, the curator — depends on one thing that doesn't currently exist: a user interface that exposes context as a first-class object you can see and manipulate.

Right now context is invisible. You can't see what's in the window. You can't remove something that's polluting it. You can't pin something important. You can't annotate, tag, or organize. You just talk and hope the system handles it well. That's like writing code without being able to see the file system.

The system needs a user-facing screen that shows all the threads in the conversation. The user can see what's there, see what the curator is planning to include in the next turn, and directly manipulate any of it. Remove this thread, keep that one, mark this statement as important, flag that exchange for decay.

Crucially, the default behavior must be opt-out, not opt-in. The curator should be aggressive — hiding noise, evicting stale context, pruning tool outputs automatically. The user interface is there for when you want to override, not for routine gardening. If users have to spend a significant portion of their time managing their context graph, the system has failed at its primary job. The curator handles it. The user intervenes only when it gets something wrong.

But the interface goes beyond simple include and exclude controls. The user should be able to add any metadata they want. Tag a thread with a project name. Annotate a statement with a note about why it matters. Add a custom label, a priority level, a relationship to another thread, a caveat, a status. The metadata schema should be open-ended because users understand their own work in ways no automated system can anticipate. The curator benefits from every piece of metadata the user adds, because it all enriches the relevance model.

This isn't a convenience feature layered on top of the architecture. It's the enabling layer that makes the entire system intelligent. Without the interface, the curator has no training signal. With it, every user action is data.

Every time the user marks something for inclusion, that's a signal that the curator underestimated its relevance. Every time the user excludes something, the curator overestimated it. Over time, these corrections become training data. The curator learns the user's patterns and converges toward autonomous management.

The patterns are remarkably consistent and learnable. Tool calls are the obvious example: you invoke a tool, use the result, and the invocation metadata is just noise afterward. The curator watches you discard tool metadata ten times and learns the pattern. Same with web search — you pull back ten results, keep one, and discard the other nine. Same with file reads where only three lines mattered, API responses where one field was relevant, error messages from debugging where only the final fix was worth keeping.

All of these follow the same retrieve-select-discard pattern. The user starts out doing manual curation. The curator watches and learns. Gradually, the user intervenes less and less. The manual controls never go away — they're always there as override and correction — but they get used less frequently as the system gets smarter.

The Curator as Active Participant

The curator shouldn't be limited to silent background work. It should have its own communication channel with the user, separate from the main conversation with the reasoning agent.

Through a sidebar or notification system, the curator can proactively suggest: "You discussed this topic last week — want me to pull that thread in?" The user accepts or dismisses, and the curator adjusts accordingly.

This works in both directions. The curator can suggest inclusions, and it can also flag planned evictions: "This conversation is getting long. I'm planning to drop the early debugging thread — ok?" The user gets a chance to override before eviction rather than discovering the loss after the fact.

So the system has three clean communication channels: the user talking to the reasoning agent through the main chat, the curator talking to the user through the notification sidebar, and the curator assembling input for the reasoning agent through the manifests. Each channel is separate and well-defined.

And of course, the notification channel is yet another learning signal. Every accepted or rejected suggestion makes the curator smarter about when to offer and what to offer.

Current Theory and Decay

A persistent repository needs policies about what stays and what fades. Without them, stale information accumulates, old contradictions persist, and the repository itself becomes a source of pollution — the same problem the system was built to solve. Memory systems fail more from not forgetting than from forgetting.

The solution is two-directional. Information should decay by default. Threads that haven't been referenced in a long time, tool outputs that were consumed and never revisited, exploratory tangents that didn't lead anywhere — these should gradually lose relevance weight and eventually be evicted from active consideration. They remain in the repository as historical record, but the curator stops considering them for manifest inclusion unless something triggers their relevance again.

In practice, this looks something like: every thread carries a relevance score. The score decays over time — an exponential fade based on how long it's been since the thread was last referenced. Each time a thread gets referenced, its score jumps back up. Each time the user manually includes a thread, the boost is larger. Each time the user excludes one, the score drops sharply. Threads above a threshold get considered for the active manifest. Threads below it stay in the repository but fall out of active rotation. The curator learns the right threshold and decay rate from the user's correction patterns. Nothing here is exotic — it's the same kind of scoring that cache eviction policies have used for decades.

In the other direction, the user needs the ability to mark something as current theory. Current theory status means this is what we're working with right now — it's the best understanding we have, treat it as authoritative, protect it from decay. Project requirements, architectural decisions, working hypotheses that anchor ongoing research, user-stated facts — these are current theory. They go on every manifest and persist at full weight.

But unlike a permanent pin, current theory is revisable. When better information arrives, the user updates the current theory and the previous version moves to historical status. It's not dogma — it's the working model, protected from erosion but open to deliberate revision.

Both of these controls live in the same thread interface. The user can mark threads or individual statements as current theory, and the curator treats them accordingly. Over time, the curator learns what kinds of information tend to get this designation and can start suggesting candidates proactively — another instance of the same learning loop that drives the entire system.

The Bigger Point

Everyone in AI is chasing bigger models, bigger context windows, bigger training runs. The assumption is that understanding comes from scale. But understanding isn't a property of the model alone. Understanding emerges from the relationship between the model and its context. Get the context right and a modest model produces remarkable results. Get it wrong and the most powerful model in the world stumbles.

What this paper describes is the heart of a fuzzy operating system for language models. Context management is one half of it — memory management, scheduling, garbage collection, caching, indexing. The other half is protocols: structured instructions that tell the reasoning agent how to approach different kinds of tasks. If the curator decides what the reasoning agent should think about, protocols decide how it should think about it. A protocol might specify how to handle a debugging session, how to write a research paper, how to process a user correction, or how to recover from a failed tool call. Protocols are the instruction set of the operating system. Together with context management, they form a complete runtime environment for language model cognition.

Both halves share the same fundamental character: they're fuzzy, not rigid. The curator doesn't follow fixed eviction rules — it learns from user behavior. Protocols aren't static scripts — they evolve through use, get refined when they fail, and can be proposed, reviewed, and updated as the system encounters new situations. The entire operating system is a learning process that gets smarter through use.

Importantly, either half delivers significant value on its own. A system with only context management — the curator, the manifests, the threaded repository — would dramatically outperform current LLM interfaces just by keeping the window clean. A system with only protocols — structured instructions that guide reasoning behavior — would dramatically improve consistency and reliability even with an unmanaged context window. You don't need both to start getting returns. But implementing both creates something greater than the sum of the parts, because well-managed context makes protocols more effective, and well-designed protocols help the curator understand what kind of context each task requires.

And if you take one more small step — giving the reasoning agent the ability to propose revisions to its own protocols when they fail or prove insufficient — the system becomes a genuine active learner. The curator learns what context to provide. The protocols learn how to guide reasoning. The whole operating system evolves through use, correcting itself, refining itself, getting better at its job without anyone retraining a single weight. That completes the fuzzy operating system.

The pieces exist individually in various forms: retrieval-augmented generation, memory agents, conversation summarization, tool-based planners, agent frameworks. But the synthesis — threads instead of logs, two manifests, a continuous curator acting as a feedback controller, user-in-the-loop corrections with open-ended metadata,

provenance-aware eviction, current theory marking, a global repository, an emergent knowledge graph, and a protocol layer that governs reasoning behavior — is a more coherent and systematic approach to the problem than any of these pieces offer alone.

Scaling parameters is hitting diminishing returns. Better information plumbing is probably higher leverage. Context management isn't a feature. It's the foundation. The industry should be investing as heavily in curation as it is in scale, because a well-curated context window isn't just more efficient — it's the difference between a model that processes text and a system that actually understands.