The most critical topic in the world today is agentic context engineering or how you deal with memory and AI agents. We are overdue for a deep dive on this. So I'm going to go into three key papers that were recently published on aentic context engineering. I'm going to tell you how it works, how people commonly misbuild or mischaracterize their agentic memory systems, and then we're going to talk about the use cases that you can only unlock with aentic context engineering. So let's dive in. First,

people misunderstand memory. When we say context, people often think a giant prompt window. And when we say memory, they often think, well, that has to be a rag or vectorized embeddings in a database. Really, for agents, memory is the system. The prompt is not the agent. The LLM by itself is not the agent. The state, how the agents actions are stored, transformed, filtered, reused, evolved. That's the entire difference between a toy demo and something that handles real work. And we misunderstand

that. The last two years have given us longer context windows and they've given us much much smarter models. But they did not solve the memory problem. In fact, they intensified it. The naive mental model is as contexts get bigger, agents get more capable. But what actually has happened is that attention has become scarce and logs have ballooned and irrelevant history so often drowns out critical signals when we talk about agentic memory. And so because we don't handle our memory correctly, that means performance has

actually often fallen as tasks get longer. And that's not the fault of the LLM, it's the fault of our memory construction. So this is for us to shift. We have to stop trying to stuff everything into a context window and stop assuming everything is a rag and we need to start engineering memory as a first class runtime environment. Google's ADK showed the architectural fix here. This is the one of the papers that I'm going to be talking about. It's a tiered memory system where you have

working context, you have sessions, and you have memory and you have artifacts. So the prompt is dynamically compiled at each step. It's not just accumulated blind. It's the first mainstream articulation where context is truly computed, not just appended. The ACCE paper showed the adaptive fix side of things. How prompts, instructions, and memory must evolve through execution feedback. AC, by the way, is the agentic context engineering paper that Enthropic put together. So, static prompts and

oneshot fine-tunes are not going to survive long horizon tasks. So many people say, "Do I need to fine-tune my agent for this or that?" You don't. Agents need systems that update their strategies without collapsing into vagueness. And the system is what matters. The combination of the prompts, instructions, and memory. Meanwhile, Nanis published a paper that showed a very practical fix. Longunning agents only work when they aggressively reduce context, when they offload their heavy

states into a file system or a virtual machine and isolate sub agent scope very cleanly. So without this long tasks just end up imploding under the bloat of logs or of tool noise or of instruction drift. And that's part of what degrades agent performance. So when you put all these three together, the enthropic

piece, the Google piece, the manis piece, you get the first coherent blueprint for agentic context engineering in late 2025. So if I were to summarize it out, and we're going to talk about each of these, we need to

talk about memory first design, context as effectively a compiler output, retrieval over pinning, we'll get into that, schemadriven summarization, offloading heavy state and what that looks like, how you isolate agent scopes effectively, and evolving playbooks that sharpen over time. So let's jump into it. So the first scaling principle I want to talk is that you should treat context as a compiled view not as a transcript. Every LLM call should be a freshly computed projection against a

durable state. So what's relevant now? What instructions apply now? Which artifacts matter now? And which memories should I surface now? You're computing that at runtime. You're not just assuming a durable state that's always the same. This prevents signal dilution, right? So instead of dragging the last 500 turns of the context window or the conversation into memory for the agent, you rebuild the minimal very relevant slice that preserves task continuity so you don't flood the LLM's attention.

It's the only way to make multi-hour agent loops work. So often people assume that they can get away without this and you can't. Principle number two, you need to build a tiered memory model that separates storage from presentation. So working conduct context is a minimal per call view, right? That's what we talked about. Sessions are much more like structured event logs for the whole trajectory of action. And memory is a durable searchable insight that's extracted across multiple runs.

Meanwhile, artifacts I would define as large objects that are referenced by handle or by tag. They're not just pasted in. So when you define these four correctly and you have a truly tiered memory model across working context, sessions, memory and artifacts, you can separate it out so that context window stays small while state the overall memory system can grow arbitrarily large. It can grow very rich and that mirrors traditional computer architecture to be honest with you. There's we have the idea of a cache, a

RAM and disc drive because the same bottlenecks reappear in LLM agents. And so why reinvent the wheel? Let's just apply it correctly in this context. Principle number three, scope by default. The agent should pull memory when needed, not inherit everything. So default context should contain nearly nothing. I'm going to say it again because almost no one says this. Default context should contain nearly nothing. And retrieval then becomes an active decision. The agent chooses when to recall past steps. It chooses when to

fetch artifacts. It chooses when to load additional details. This keeps attention focused. It avoids context rot and it makes long horizon tasks very feasible because old information is not ever passively carried forward. Principle number four, retrieval beats pinning. So long-term memory needs to be searchable, not pinned and permanent. There's attempts to keep everything in context that tend to fail because attention constraints bite. So if you have a very large context window, like a million

tokens, we'll see even larger ones. It's tempting to just stick it in the context window, but your retrieval accuracy tends to drop. So treat memory as something that the agent will query on demand with very clear relevance ranked and structured information. So the window is always the the context window, right? It's always the result of a search, not the accumulation of history passively. And this is how agents can differentiate between a critical constraint from 5 days ago and noise

from 5 minutes ago. Right? Otherwise, the agent is going to be very recency biased, very confused in the context window. You have to give the agent clear instructions for retrieval and enable it to retrieve in order to get past this idea that you can just sort of throw long-term memory in if you compress it enough. Principle number five, summarization needs to be schema- driven and it needs to be structured and ideally reversible. And people don't talk about that part. So, naive summarization will turn multi-step

reasoning into a very vague sort of overarching glossy soup. It strips away decision structures. it just kind of compresses it all in. But if you compact intentionally, if you compact using schemas, using templates, using event types very intentionally so that you preserve the essential semantics of whatever you have a memory about, then you're going to be dropping surface detail, but you know for a fact that your structure, your schema guarantees that the relevant parts of the memory are preserved. That is what makes long

run context maintainable. It's what makes it debugable because you can inspect not just what was summarized but how it got summarized. Almost no one is talking about this. By the way, this is a really important piece that I don't see in practical agentic conversations. Principle number six, please, please, please offload your heavy state to tools, to file systems, to sandboxes. Do not feed the model raw tool results, especially at scale. You want to write them to disk and pass pointers. You

don't expose if you can 20 overlapping tools, right? Expose a small orthogonal set of tools like a shell or a browser or file operations and then and then let the agent compose its own workflows that keeps the context really lean and it reduces the cognitive burden on the model which unlocks much more complex chains of behavior. You might think if we don't have 20 overlapping tools we can't get to complex chains of behavior but it's actually the opposite. When you have a very clearly orthogonal set of

tools, the agent is more free to understand what's in the box and it can allocate more compute toward those cool workflows. Principle number seven, use sub aents to isolate the state and the scope, not to mimic human organizational charts. So, sub agents should stop context explosion by giving different actors their own working context and responsibilities. planner, executor, verifier are all classic agent types, and they need to have narrow scoped views and communicate through structured artifacts, not just sprawling

transcripts they pass back and forth. This should eliminate a lot of the cross talk, the reasoning drift, the

hallucinated teamwork that plagues a lot of naive multi- aent designs. Notice that planner, executor, verifier are not human job titles. Do not create your agents with human job titles. There is no point. Think of it as agentic task and don't throw your human assumptions onto it. Principle number eight, design the prompt layout around caching and prefix stability. I'll explain what I

mean. So a stable prefix like the identity, the instructions, the static strategy of the agent should rarely change so that the cache can be reused across turns. Only the variable suffix, the current user input, the fresh tool outputs should change. So this transformed multi-step agent loops from a very lengthy 200 milliseconds per step because it's reading this gigantic prompt layout with lots of caching and prefix instability into a very stable thin prefix that can then be read quickly for the suffixes like the

current user input. So the schema is clean, right? It's very narrow. It's very clean and only the output is changing in a way the model can understand. That can drop your latency 10x right from 200 milliseconds to 20 milliseconds or something like it. And that can drive down cost dramatically, but it also enables more reliable outputs. Principle number nine, let the agents own strategies evolve. So static prompts will freeze the agent until you evolve the prompt, right? It freezes it at version one. Whereas anthropics

approach shows that strategies, memories, and instructions should update through execution feedback. Small structured increments that sharpen capabilities instead of overwriting them. And that allows agents that actually learn from doing, not from human tinkering. So fundamentally, if your agent is allowed to update its strategy, if it's allowed to update its memory, it's allowed to update its instructions as it learns, you then unlock the possibility of an agent that learns to do its job better. What are

some common pit pitfalls that break if people do not apply these? And I do mean common. I have done some looking. I have built agents. This often happens. Number one, dumping it all into the prompt, right? That leads to signal dilution. It leads to costs that rise, degraded reasoning. Agents literally become less accurate when you do this. Number two, blind summarization that erases any kind of domain insight. Right? If you just say summarize it and you don't have a schema, you don't have structure, you're

losing constraints, you're losing your edge cases, you're losing any kind of causal relationship there, the things a capable agent needs to do its job, right? And this produces context collapse where the agent becomes increasingly generic rather than doing specific useful work. Pitfall number three, treating long context windows as if they're unlimited RAM. Bigger windows actually increase noise, right? And they increase confusion unless they're paired with relevance filtering. More tokens

does not necessarily mean you're going to get more clarity and it often means more distraction. People will treat context windows like they like the trunk of a car. They can just dump stuff in there. Don't do

that. Pitfall four, using the prompt as an observability sink. People will stick debug logs. They'll sync error lo messages. They'll they'll put giant tool outputs into the prompt and that will just pollute attention. Humans need the observabilities, but the agent just just

drowns in all of that. So, please construct the system for stable agent performance. We will find other ways to get to observability. And actually, a well- constructed memory system is very observable. Pitfall number five, tool bloat. Right? If you give the model many subtly different tool options and a giant tool schema, you might think you're very sophisticated, but all you're doing is increasing error rates and you're slowing your system down. Pitfall number six, anthropomorphizing

the agents, right? If multiple agents have the same transcript and they're all trying to talk and they're trying to assume human roles because you gave them human jobs, you're going to have reasoning drift. You're going to have duplicated efforts. You're going to have compounding hallucinations. You want to make the system less fragile, right? And I think there's probably a lesson for us humans here too, right? We are designing systems where the agents just have to have the context needed to do the job.

Maybe there's a clue for us at work as well into how our roles are evolving. Just a thought. Seventh pitfall, static prompt configurations that never change. People put in like no accumulation of knowledge or sharpening of huristics. They don't design the system to evolve. You essentially rebuild the agent from scratch every run, but you don't give the agent room to adjust its prompting or thinking as it grows in intentional ways. This is a somewhat sophisticated implementation, but good multi-agent

implementations give the system room to learn intentionally. They don't just have static configurations that never change. And as long as you're documenting that learning, you have room to control it, observe it, change it, and manage it safely. Number eight, if you overstructure the harness, the model will feel boxed in. So if a Frontier model produces no improvement when it's swapped in, your architecture is usually the bottleneck. So basically, rigid harnesses can kill emerging capability

and there's a fine line between a useful harness with orthogonal tools and a very clear injectable context prompt that you can configure as needed with schemas that evolve that that gives the model room to demonstrate what it can do. If your harness is so locked down, you probably won't see as much different difference model to model. And you may naively blame the model. But really, your your structure is so narrow and boxed in. There's only one thing the model can do. Pitfall number nine,

ignoring the caching and prefix discipline. Remember I said you have to have caching and preflex discipline. If you don't do that, if you don't have clean prefix discipline in your prompting schemas, you're going to have really unpredictable latency because the model has to rep that prompt a few times to figure out what's going on. It makes it very difficult to scale as tasks get longer. So now we've gotten through the pitfalls. We've gotten through some of the principles for designing agentic

context systems with good context engineering, with good memory. What do you unlock? What can you get done once you're able to design these systems appropriately? Number one, this unlocks really long horizon autonomy. And so if you want to be in the business of multi-hour research, of doing web browsing, of simulating runs over time, say of repo audits, multi-stage code generation where agents stay coherent, you have to be in the business of giving them relevant slices of their state rather than just throwing all the

context at them at once. And so this enables long horizon autonomy. Use case number TR two. This enables true self-improving agents. So agents that get better over time will be able to log and update their strategies, their huristics, their domain knowledge if you construct your memory systems appropriately. And if you scope them right, they will not update those in ways that overcope the agent. You can still constrain a gentic scope, but allow the agent to execute within that scope with increasing intelligence as it

learns from each run. This isn't training with weights, right? We're not changing the weights of the model. This can happen entirely in your memory and instruction layers if your agents are clearly instructed to record and learn from what they did. Use case number three, this enables cross-session personalization that actually scales. So if you want persistent profiles that remember user preferences, that remember constraints, that remember prior outcomes or behavior patterns in your agents, then you don't have to balloon

out the per call context to get that done. If you're constructing the memory state correctly because you just inject the particular slice that matters. Number four, this enables real multi-agent orchestration without cross talk and without drift. So you can have a planner, a researcher, an executor, a validator, a tester all collaborating through structured artifacts and you don't just have chaotic chatter between them that leads to context poisoning. Use case number five, you can enable

deep reasoning over very large bodies of work. So agents can analyze entire repos, data sets, PDFs or logs by treating them as artifacts instead of tokenizing all of them. because you can structure sampling, you can structure retrieval, you can structure clean summarization that decouples reasoning from the raw size of the repo. Number six, you can enable auditable, compliant, enterprise ready agentic systems. And this is an outcome perhaps, but I think it's worth calling out because memory is what stands in the way

of those systems. So you can get full reconstructibility of what the model saw and why it acted. So session logs, compaction events, memory updates would all be traceable. And that's that's all happening at the memory layer. If you don't construct your memory layer correctly, it makes it difficult to trace in ways that are critical for finance, for legal, for medical, or really for production systems. Use case number seven, you need coststable agent operations. And so you need cost growth

that isn't linear. In fact, it should be sublinear. And you can get that if you reuse your caches, if you compact views intelligently, if you keep your working context small. Basically, every token counts if you construct your memory properly. And that gets you to things like always on agent services where you're not just scaling your costs. And that matters if you're trying to really scale agents across the enterprise. Number eight, you can get to domainspecific agent OS environments.

And so finance agents will have long-term risk if their memory isn't constructed correctly. Coding agents can have full workspace history if the memory system is constructed correctly. Medical agents will have durable patient state if the memory system is constructed correctly. Any place where you have persistent workspaces, you can get longerterm intention for agents. And it's not really a function of waiting for LLMs to get smarter. the LLM can understand the long-term history of a project or the long-term risk factors uh

as I called out with finance or the long-term opportunity factors because the memory architecture is there and so when I wrap all of this up what I want to call out is that this is think of this as the trade craft the lessons learned that we have collectively put together at major model makers me in practice building agents others who are building agents in production systems This represents the best of what we know about constructing memory. And it's critical to get memory right. If you want agents that work in production at

scale and so that's why I wanted to do a really deep dive on context engineering for agents. I don't think there's a substitute here. There's no magic bullet. You have to dig in and understand how memory and context engineering actually works. So I did a much longer write up on this on the Substack. I encourage you to dive in. It's something we can't skip if we want agents to really work. Best of luck.