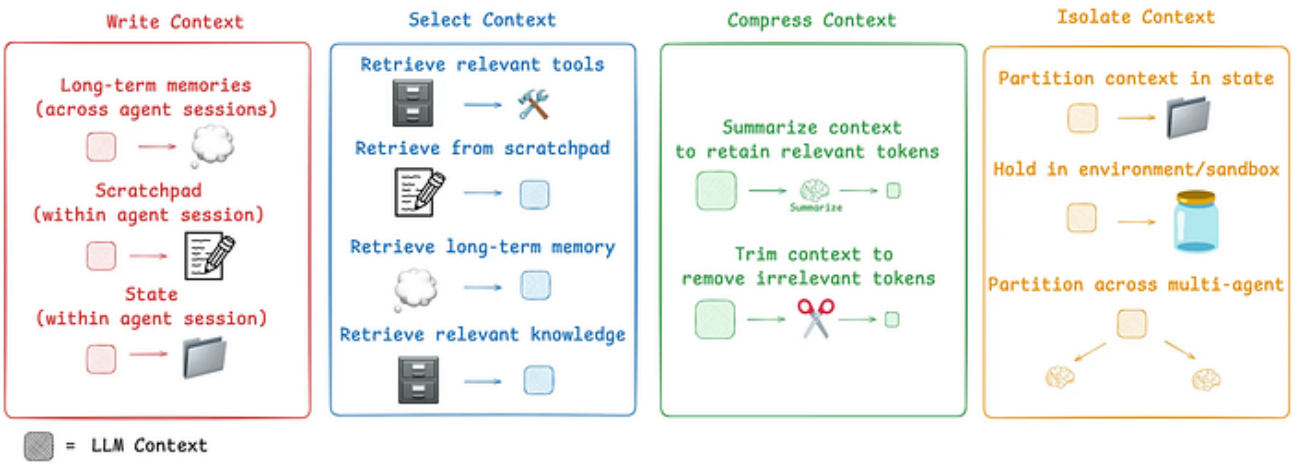
Context Engineering in LLM-Based Agents

Jin Tan Ruan, CSE Computer Science - ML Engineer • 34 min read • 2025-07-23
https://jtanruan.medium.com/context-engineering-in-llm-based-agents-d670d6b439bc



Modern AI agents built on large language models (LLMs) must manage a wealth of information to function effectively. The term context engineering has emerged to describe the “art and science of filling the context window with just the right information for the next step.” In essence, context engineering is about *providing all the relevant context an AI needs to solve a task.* This goes beyond traditional *prompt engineering* - instead of just crafting a single prompt, context engineering considers *everything* the model sees before generating a response. This includes the system instructions, user query, conversation history, long-term knowledge, retrieved documents, tool outputs, and even desired output formats. Effective context engineering is increasingly recognized as critical: as one CEO quipped, “most agent failures are not model failures anymore, they are context failures”. Indeed, context engineering has become “effectively the #1 job” for engineers building AI agents.

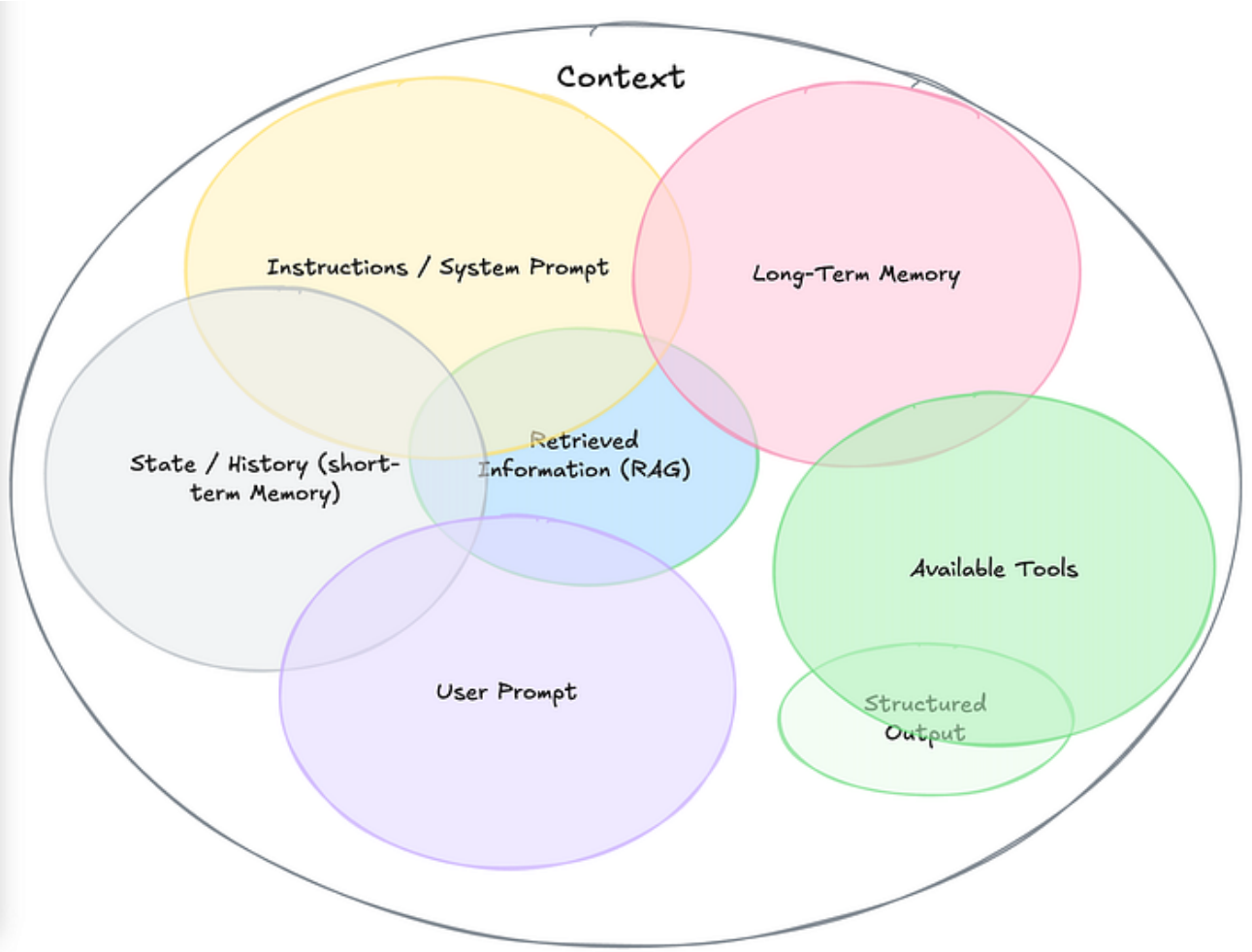
LLMs have finite *context windows* (analogous to a fixed RAM size), so agents cannot simply stuff all information into the prompt. Long-running tasks and tool feedback quickly accumulate, risking context overflow. Feeding too much or irrelevant context can confuse the model or degrade performance. This can lead to issues like *context poisoning, distraction, confusion,* or *clash.* Therefore, managing context is vital for reliability, scalability, and cost-efficiency of agentic systems. Research from Anthropic underscores this: agents in multi-turn research tasks needed careful context management over hundreds of turns to remain coherent. Without it, an agent may exceed the context limit, incur high latency/cost, or make mistakes by focusing on the wrong information. In summary, the quality of context fed to an AI agent largely determines its success. Context engineering thus shifts our focus from finding a “magic prompt” to designing dynamic systems that give the model *the right information, in the right format, at the right time.*

Four categories of context engineering strategies: writing (persisting context), selecting relevant context, compressing context, and isolating context

What is Context Engineering?

Context engineering can be defined as *the discipline of designing and building dynamic systems that provide an AI model with all the information and tools it needs to accomplish a task.* This definition highlights a few key points:

1. System, not just a string - context is not a single static prompt but the output of a system that prepares input for the LLM.
2. Dynamic assembly - the context is constructed on the fly per request, tailored to the task at hand (for one query it might include a user's calendar, for another a relevant document or web search).
3. Right information and tools - the goal is to avoid "garbage in, garbage out" by ensuring the model isn't missing crucial details. This means injecting both knowledge (facts, data) and capabilities (tools/functions) *only* when needed and useful.
4. Format matters - how information is presented is critical. A concise summary of a document is often better than a raw dump of text, and a well-specified JSON schema for output can guide the model better than vague instructions.



Components of AI Agent Context - A visual representation of the overlapping elements that make up the full context provided to a large language model agent

In practice, context engineering encompasses everything the model sees as input. A typical LLM-based agent's context might include:

• System Instructions/Prompt: e.g. role or task definitions, rules, and high-level goals given to

the model (possibly with few-shot examples).

- User Request: the current query or command from the user.
- Short-Term Memory (Conversation History): the running dialogue or recent interactions, which gives the model a sense of the current session's state.
- Long-Term Memory: persistent information stored across sessions - key facts, user preferences, or summaries of past interactions that the agent can retrieve when relevant.
- Retrieved Knowledge (Documents/Database info): any external facts fetched via a *retrieval augmented generation (RAG)* pipeline or API calls, providing up-to-date or domain-specific info the model didn't originally train on.
- Tool Descriptions: definitions of tools or functions the agent can invoke (including how to call them).
- Tool Outputs/Feedback: results returned from tools (e.g. the output of a web search or a code execution), fed back into the model for the next reasoning step.
- Structured Output Templates: any schemas or formats the model is instructed to use for its answer (for instance, an agent might be told to output a JSON with specific fields - this schema itself acts as context guiding the output).

By carefully curating these components at each step, we give the model *everything it needs* to perform robustly. For example, an AI scheduling assistant wouldn't just get the user's question "Are you free tomorrow?" - a context-engineered system would also provide it the user's calendar availability, relevant past emails (to infer tone and intent), the identity/relationship of the requester, and appropriate tools (like a "send_invite" function), all before the model formulates a response. The difference is dramatic: without rich context, the agent's reply may be generic and unhelpful; with rich context, it can produce a personalized, correct answer and take actions (e.g. scheduling a meeting) seamlessly.

Techniques and Tools for Context Engineering

Achieving effective context engineering requires a collection of techniques and tools. Key methods include prompt structuring, retrieval of relevant information, memory management, context compression, intelligent chunking of data, and relevance scoring. We discuss each in turn.

Prompt Design and Structuring

Even though context engineering goes beyond writing a single prompt, prompt structuring remains crucial as the foundation of the context. This involves how we formulate system instructions and user prompts, how we incorporate examples, and how we delineate different parts of the context. Good prompt structuring sets the stage for all subsequent inputs. For instance, clearly separating the roles (system vs user vs assistant) and including high-level instructions can steer the model's behavior. Few-shot examples can be embedded in the prompt to "demo" the desired behavior or format. Prompt design also extends to *calibrating the prompt* to avoid known biases. Research has shown that LLM performance can be brittle to prompt wording and order - e.g. changing formatting, choice of labels, or example selection can yield differing

results. To address this, *prompt calibration* methods have been developed. Zhou *et al.* (2024) systematically analyzed prompt bias and introduced Batch Calibration (BC), a simple zero-shot method to adjust the model's predictions by controlling contextual biases from the prompt format. BC unified several prior calibration techniques and was able to recover lost performance due to prompt idiosyncrasies, achieving state-of-the-art results on a range of language and even image classification tasks. The implication is that careful prompt construction (and adjustment) is an integral part of context engineering - we must ensure the "framing" of the task in the context does not inadvertently confuse the model or skew its outputs. Academic studies on in-context learning also emphasize selecting appropriate exemplars and prompt wording to guide the model effectively. In summary, prompt structuring is about packaging the initial context (instructions + query + examples) in an optimal way, which then sets the stage for adding external information via retrieval or memory.

Retrieval-Augmented Generation (Document Retrieval)

One of the most powerful tools for context engineering is retrieval-augmented generation (RAG), which integrates external knowledge sources into the LLM's context. Pre-trained LLMs have a lot of knowledge in their parameters (their *parametric memory*), but they can't possibly store or recall all specific facts, nor do they stay up-to-date after training. RAG addresses this by equipping the model with a *non-parametric memory*: typically a vector database of documents or a search system, from which relevant information is pulled on the fly. Lewis *et al.* (2020) introduced RAG models that combine a parametric seq2seq model with a dense vector index of Wikipedia; by retrieving passages related to each query, these models outperformed purely parametric models on open-domain QA and produced more factual, specific answers. The key idea is that for knowledge-intensive tasks, the agent should *retrieve documents* (or database entries, etc.) at query time and supply them as part of the LLM's context. This greatly expands the effective knowledge of the agent without needing to retrain the model.

In practice, implementing RAG involves a few steps: document indexing, query embedding, similarity search, and context assembly. A corpus of reference texts (enterprise documents, knowledge base articles, code repositories, etc.) is processed into vector embeddings and stored. When the agent faces a query or a situation that likely requires outside knowledge, it forms an embedding of the query and finds the nearest chunks of text in the vector space (or uses other retrieval methods like keyword search or graph queries, depending on the case). The retrieved text snippets are then inserted into the prompt context (often as a section like "Relevant Information:" before the model is asked to respond). This process effectively *grounds* the model's generation on non-parametric knowledge.

A critical part of retrieval is ensuring the model gets *relevant and sufficient* information. This is where chunking strategies and relevance scoring come into play (discussed more below). If the documents are too large to feed entirely, they must be broken into chunks. The chunks need to be sized and structured such that each chunk is coherent and useful on its own, but not so large as to overflow the context window when several are retrieved. Typically, a balance is struck: "chunks big enough to contain meaningful information, while small enough to enable low-latency, performant retrieval". A good rule of thumb is that a chunk of text should "make sense *out of context* to a human" - if it does, it likely contains a self-contained idea that an LLM can use effectively. Overly small chunks (e.g. sentence fragments) may lose meaning and hurt retrieval

accuracy, whereas overly large chunks (e.g. full chapters) may dilute the relevance.

Relevance scoring is the mechanism by which the agent or retrieval system decides *which* chunks (among thousands of candidates) to include in context. Commonly this is done via vector similarity: the cosine similarity between the query embedding and chunk embedding serves as a relevance score. The top *K* scoring chunks are chosen. However, relying purely on embedding similarity can be insufficient, especially as the corpus grows large or the query has nuanced needs. In production systems, hybrid approaches are often used: e.g. filtering by metadata or keywords, using symbolic search (like regex or SQL) for precise constraints, or employing a reranker model. As an example from software agents, retrieving code by embedding search alone became unreliable as the codebase grew, so developers combined multiple techniques - AST-based chunking, keyword grep, and a *re-ranking step* to order retrieved code snippets by true relevance. In other scenarios, we might incorporate *time recency* into scoring: if freshness matters, rank newer documents higher. The context engineering system can apply custom logic here. For instance, the snippet below illustrates a retrieval helper that filters results by date and then sorts them so that the most recent information is included first:

```python
def retrieve_recent_knowledge(query, cutoff_date):
    nodes = retriever.search(query)
    # Filter retrieved nodes by date, then sort by date (newest first)
    recent_nodes = sorted(
        [n for n in nodes if n.date > cutoff_date],
        key=lambda n: n.date,
        reverse=True
    )
    return "\n----\n".join(n.text for n in recent_nodes[:5])
```

In this example, *retrieve.search* might query a vector index or search API. We then filter the results to those newer than a certain date and sort them descending by date, concatenating the top 5 into a context string. This kind of custom relevance ranking ensures, say, an agent looking up “latest policy updates” actually sees the latest ones.

Once relevant documents are retrieved, they are added to the prompt (often delimited by special tokens or section headings). The LLM then has access to this text and can condition its output on it. RAG has proven highly effective at reducing hallucinations and improving factual accuracy, since the model can cite or draw from actual sources provided in context. However, it introduces its own challenges: the agent might be distracted by irrelevant retrieved text if the retrieval isn’t precise, or it might ignore the context if not prompted to use it. To mitigate this, prompt design often explicitly instructs: *“use the following context to answer”* or even asks the model to cite the provided sources, reinforcing the usage of retrieved content.

Memory Management (Short-term and Long-Term Memory)

Another pillar of context engineering is how an agent manages memory - information that arises during the agent’s operation that may be needed in the future. We can distinguish *short-term memory* (the recent interaction history or working notes within one session) and *long-term memory* (knowledge retained across sessions or tasks). Because an agent’s dialogue or reasoning can span many turns, it cannot literally keep the entire history verbatim in the prompt once it grows beyond the context window. Thus, strategies for memory management are crucial

to avoid losing important information.

Short-Term Memory (Scratchpads & Histories): Within a single session, agents often use a *scratchpad* - a mechanism to record intermediate results, thoughts, or important data outside the immediate LLM context. This idea of a scratchpad is inspired by human problem-solving: we jot down notes or calculations on paper to free our mind's limited working memory. Similarly, AI agents can write important details to an external store (a file, a database, or some state object) instead of trying to pack everything into the prompt. For example, an agent solving a complex task might save its overall plan or partial answers to a scratchpad file via a tool call. By doing so, it ensures that even if the conversational context gets truncated or reset, the key information persists and can be re-injected later. Anthropic's multi-agent research system provides a clear example: the lead agent "saves its plan to Memory to persist the context, since if the context window exceeds 200,000 tokens it will be truncated". In LangChain's framework, this concept can be implemented as part of the agent's *state* that carries over between turns.

Long-Term Memory: Agents that interact continuously with a user (or operate over long periods) benefit from accumulating knowledge about past interactions and world state. This long-term memory is often implemented with a vector store or database that the agent can query when needed. A prominent technique is to extract salient facts from each conversation or session and store them with embeddings (a *memory index*). Later, when relevant, the agent retrieves from this memory index using similarity search (much like document retrieval) to remind itself of past facts. For instance, Shinn *et al.* (2023) introduced Reflexion, a framework where after each trial or turn, the agent *reflects* on what happened, writes a brief "reflection" (in natural language) of mistakes or lessons, and stores this in an episodic memory buffer. In subsequent trials, the agent consults these stored reflections to avoid repeating errors and to improve decision-making. Remarkably, providing an agent with this kind of self-generated memory significantly improved performance on diverse tasks - in coding, Reflexion enabled a pass@1 score of 91% on HumanEval, surpassing even GPT-4's 80% on that benchmark. This underscores how a well-designed memory system can enhance an agent's capabilities without any model fine-tuning, simply by injecting useful context from prior experience.

Another influential work is Generative Agents by Park *et al.* (2023). They built simulated characters that behave in human-like ways over long periods, and their architecture relied on extensive memory and retrieval to drive behavior. Each agent recorded *every observation and interaction* it experienced as natural language entries in a memory stream. These raw memories were periodically *synthesized into higher-level reflections* (summaries that distill the important aspects). When an agent needed to decide on an action (e.g. plan its day or react to a new event), it would dynamically retrieve relevant memories (via embedding search over the memory store) and feed them into the prompt to influence its next move. This design allowed the agents to remember past events (who said what, what their goals were, relationships formed, etc.) and exhibit coherent, evolving behavior over days of simulated time. The researchers demonstrated that these agents produced believable behaviors - e.g. one agent decided to throw a party, and others remembered this and coordinated to attend, driven by the retrieved memory of the invitation. The key takeaway is that LLM-based agents can be extended with a *memory subsystem* that stores a comprehensive record of events (in plain language), summarizes them to avoid overflow, and recalls them when needed to inform future actions. In effect, this is context engineering across a long horizon: since the raw context window can't hold all past interactions,

the system uses an external memory and intelligent retrieval to simulate an indefinitely long context.

Frameworks like LlamaIndex provide ready-made components for memory management. For example, LlamaIndex offers a VectorMemory (which stores chat messages in a vector database for retrieval), a FactExtractionMemory (which extracts and stores factual statements from the conversation), and other specialized memory blocks. Developers can mix and match these to suit their application. Each time the agent is about to generate a response, it can query these memory stores for any entries related to the current query/context and prepend them to the prompt. A crucial design decision is how much memory to retrieve each time - too little and the agent might miss relevant facts; too much and the context window may overflow or irrelevant details may creep in. Often a balance or a threshold on relevance score is used to include only highly pertinent memories.

One challenge with long-term memory is relevance filtering. As an agent's memory grows, we cannot present all of it to the LLM. The agent needs a way to pick which memories are likely useful in the current situation. This is typically done with embedding similarity (treating the conversation state or user query as the retrieval query). However, this can sometimes backfire. The LangChain blog recounts an incident: ChatGPT, which now retains long-term conversation history per user, once fetched a user's location from a past memory and *unexpectedly injected it* into a current response image request. The user was startled - it felt like the model "leaked" personal context that wasn't intended in the prompt. This highlights the need for careful control: memory retrieval should be *relevant* and *appropriate*. If the selection criteria are too loose (e.g. any vaguely similar text is pulled in), the user may feel the AI's response is coming out of left field, or worse, it could breach privacy expectations by recalling information out of scope. Thus, context engineering must include safeguards on memory usage - perhaps tagging memory items with contexts in which they're valid, or requiring a certain confidence to include them.

Context Compression (Summarization and Distillation)

Because of the context window limit, agents often need to compress context - that is, reduce the amount of text while retaining the most important information. Two common approaches are *summarization* and *trimming/pruning*. The idea is to free up space by replacing long transcripts or documents with shorter equivalents that convey the key points needed for the task.

Summarization: Using an LLM (or smaller model) to generate a summary of a chunk of text is a straightforward way to compress. For instance, after a lengthy tool interaction or a long multi-turn dialogue, the agent can produce a synopsis of what has happened so far, and then supply that summary in place of the full text going forward. Anthropic's Claude has an "auto-compact" feature along these lines: if a conversation reaches ~95% of the context window, it will automatically summarize the conversation history to make room. This summary is then used as context for subsequent turns, ensuring the chat can continue beyond the raw window limit. Summaries can also be inserted at specific boundaries in an agent's workflow - for example, summarizing the results of a web search (which might be a few pages of text) before feeding it back into the agent. This way, the agent gets the salient facts without the excess detail. In multi-agent setups, one might even summarize the *interaction between agents* at certain points to hand off a concise state to a new agent, a technique noted by developers to reduce token usage during knowledge hand-off. There are various summarization strategies: *recursive summarization* breaks text into

parts, summarizes each, then summarizes the summaries (useful for very long texts); *hierarchical summarization* might use a structure like chapters, sections, and bullet points. The goal is always to maintain critical info while dropping redundancy.

One must be cautious: summaries are only as good as the model producing them, and important details can occasionally be lost. If the task later requires that exact detail, a too-aggressive summarization could hurt performance. In high-stakes cases, teams use validated or even human-in-the-loop summaries. The Cognition AI team, for example, trained a fine-tuned model specifically to summarize agent interactions, indicating how much effort might go into doing this step reliably for complex applications.

Trimming and Pruning: Not all compression needs a generative summary. Sometimes the solution is to *cut out* the obviously irrelevant or low-value content. This can be done with simple heuristics or advanced methods. A common heuristic is to drop the oldest messages once the conversation gets too long (since earlier context may be less relevant than recent context). Another is to remove verbose content that the model produced (like step-by-step reasoning or tool call logs) once they've served their purpose, keeping only the final results. Developer Drew Breunig refers to this as "pruning" context - akin to snipping away branches that are not bearing fruit for the current query. There's even research into trained *context pruning models*: one paper, Provence, trained a model to identify and remove irrelevant context for QA tasks. Pruning is essentially an information retrieval approach where the system filters out low-relevance parts of the context rather than rewriting the high-relevance parts.

In academic terms, context compression aligns with what Yu *et al.* (2025) identified as one of the three main approaches to long-context LLMs (the others being extending context length and efficient attentions). They note that context compression techniques - whether token-level or via external memory modules - can help handle longer texts but often require careful integration so as not to omit needed info. Their work introduces MemAgent, a system that dynamically compresses context by learning to overwrite a fixed-size memory as it processes a long document. The MemAgent uses reinforcement learning to decide what to keep in a "memory slot" and what to discard at each segment of input, mimicking how a human note-taker might jot down key points from each page of a book and effectively handle an entire book by notes. Thanks to this learned compression strategy, MemAgent demonstrated the ability to scale from training on 8K-token inputs to handling 3.5 *million* token documents with under 5% performance drop - an impressive result that far exceeds typical context windows. The intuition, as they describe, is anthropomorphic: *humans don't try to remember every word, we abstract and focus on key ideas*, discarding redundancy. By applying a similar principle, MemAgent's compressed memory approach achieved near-lossless performance on extremely long text tasks. This is an active research frontier: how to enable LLMs to retain important context indefinitely without blowing up compute or losing accuracy. Until such approaches mature for broad use, most practical systems will rely on summarization and heuristic trimming as described, as part of their context engineering toolkit.

Chunking Strategies and Knowledge Chunking

As hinted earlier, chunking is vital when dealing with large documents or knowledge bases. Chunking refers to splitting text into pieces that can be indexed and retrieved. A well-chosen chunk size and boundaries can drastically improve retrieval relevance and downstream LLM

performance. If chunks are too large, the retrieval system might pull in a lot of extra irrelevant text around the relevant bit ("answer buried in noise"), and those extra tokens waste context space or even distract the model. If chunks are too small, important context might get split up and not retrieved together, or the embeddings might not capture meaning fully.

Key considerations for chunking include: the nature of the data (does it have logical sections like paragraphs, headings, code functions?), the embedding model's input limitations (can it handle long passages, or should input be short?), the expected length of user queries (short queries might only need small focused chunks, whereas very detailed questions might match better with bigger chunks), and the usage scenario (semantic search vs direct QA vs agent reasoning). Generally, for semantic search and RAG, it's advised to chunk along semantic boundaries - e.g. paragraph or sub-section level - rather than fixed token counts that could cut in the middle of an idea. Pinecone's guidance suggests that if a chunk "makes sense to a human on its own, it will make sense to the model". They also warn of the *"lost in the middle"* problem: even if you stuff a huge chunk (say a full document) into the prompt, LLMs have a tendency to pay less attention to the middle parts of very long contexts. Thus, simply increasing chunk size is not a solution; the better approach is to feed the model a set of focused, relevant chunks.

Another aspect is overlapping chunks. To avoid losing context at the boundaries, one strategy is to use sliding windows (e.g. each chunk of N tokens overlaps the previous by M tokens) so that an important sentence that straddles two chunks still appears in at least one chunk entirely. This increases redundancy in the index but can improve the chances of retrieval catching relevant info. There's a trade-off with index size and potential duplication in results.

Finally, after retrieval, if multiple chunks are returned, the agent might consider merging or reordering them. If the context has a logical order (like chronological logs), feeding them in order can help the model understand the sequence of events. Conversely, if each chunk is independent evidence, order might not matter. Some pipelines use a re-ranking or merging step: for example, retrieve 10 small chunks, then have the LLM itself read all 10 and produce a synthesized summary or pick the top 3 to include in final context (i.e. a *two-stage retrieval*). This blurs into the territory of using the LLM to judge relevance after the initial vector similarity cut - a powerful technique but also computationally heavier.

In summary, chunking strategy is a crucial low-level decision in context engineering for RAG systems. It determines how well the system can later *select relevant context*. A thoughtful chunking approach, combined with robust retrieval, ensures the agent isn't starved of needed info nor drowning in extraneous text.

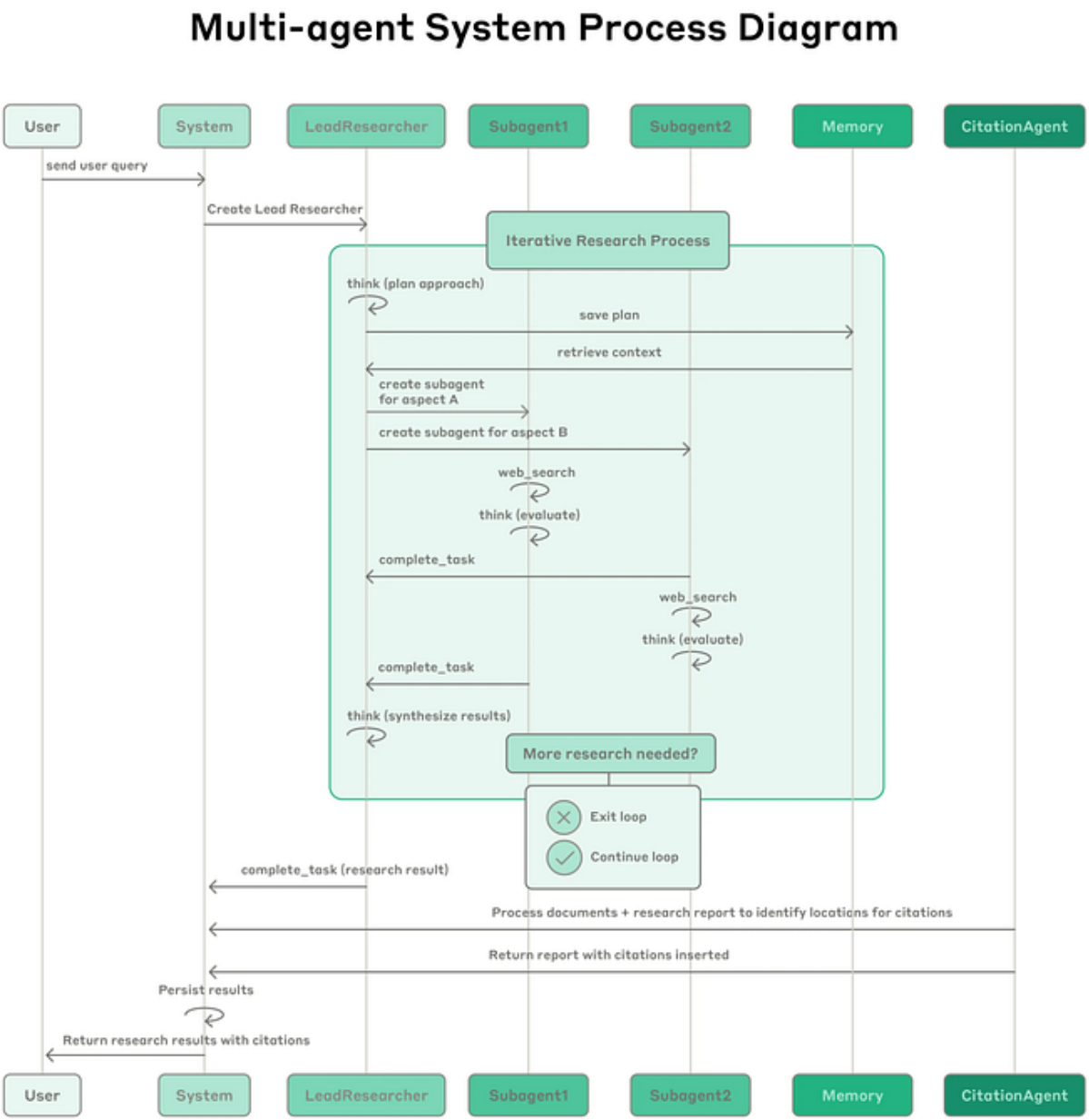
Context Isolation and Multi-Agent Systems

Sometimes the best way to manage context is to isolate it into separate threads or agents. Complex tasks can often be divided among multiple sub-agents, each with its own *narrow context* , rather than one agent trying to juggle a huge combined context. This falls under "Isolate Context" in the earlier strategy diagram. The principle of *separation of concerns* applies: each sub-agent handles a subset of the task with a focused context window, and a coordinator agent integrates the results.

OpenAI's *Swarm* library and others have explored multi-agent orchestration where, for example, one agent might work on parsing a user query, another on gathering facts, another on executing

code, etc., each with their own context and tools. Anthropic's multi-agent researcher system is a real-world case study: it uses a lead agent that spawns specialized worker agents for parallel research subtasks. Each subagent has an isolated context (only the information relevant to its specific search or task) and operates independently, returning a result to the lead agent. Because each agent has its own context window, the overall system gains effectively a *larger combined context* by distributing pieces of the problem to different LLM instances. he lead agent maintains a high-level view (and has a memory of the plan), but it doesn't need to share all intermediate details with every subagent. This architecture proved extremely effective: Anthropic reported that their multi-agent system (Claude orchestrating Claude subagents) solved certain complex queries far better than a single agent could. For a query requiring finding information about many entities in parallel, the multi-agent approach succeeded where a single-agent doing sequential searches failed. Their internal evals showed a ~90% performance boost on a research task using the multi-agent approach. The reason is intuitive - by isolating context, each agent gets to focus deeply on one aspect without being cluttered by others, and the system can cover more ground in parallel. Indeed, "[subagents operate] in parallel with their own context windows, exploring different aspects... simultaneously" and thus can jointly solve a broad task that exceeds any single context window.

However, this strategy comes with costs. Multi-agent systems "burn through tokens fast" - Anthropic observed that their multi-agent runs used ~15× more tokens in total than a single-agent chat on the same task. This is because each agent has its own prompt and they might have overlapping or redundant context among them. There's also the overhead of coordination: the lead agent must communicate with subagents (which itself is context that has to be passed around) and integrate results, which might involve additional LLM calls to summarize or cross-reference findings. If not carefully designed, breaking a task into parts can introduce new failure modes (e.g. subagents diverging or misunderstanding their instructions). The Cognition AI blog "Don't Build Multi-Agents" actually argues that naive multi-agent setups are fragile and that often a well-engineered single agent with good context sharing might be better. They emphasize two principles: share context fully with any sub-task agent (to avoid miscommunication) and recognize that every action/decision by one agent is implicit context for others. In other words, if you do use multiple agents, ensure they have a shared ground truth context or you risk them diverging. This has led some to design multi-agent systems where subagents actually get a copy of the main conversation or plan in their prompt (so they know the overall task) - but that in turn increases token usage. It's a delicate balance.



Process diagram showing the complete workflow of our multi-agent Research system

A compromise approach is using *tool-like subagents* that have narrow output. For example, Hugging Face's *Deep Researcher* agent isolates context by using a scratchpad environment: the agent's chain of thought and tool uses are kept separate from the user-facing conversation, effectively sandboxing the verbose reasoning so the user (and model on the next turn) only sees the distilled outcome. This isolation prevents irrelevant details or errors from one part of the process leaking into another.

In summary, context isolation (via multi-agent or modular design) can be a powerful way to manage complexity by *splitting context into pieces*, but it requires careful engineering to ensure consistency across those pieces. When done right, it extends the reach of LLM systems to larger problems than a single context window could handle, at the cost of higher compute. As LLM-based agents become more complex, finding the right boundaries of context - what each module or agent should know or not know - is part of context engineering as well.

In-Depth Example: “Context Engineering: From Contexts to Agents”

To tie these techniques together, let’s examine insights from recent research and how they illustrate context engineering in action. A theme in cutting-edge work is that *better context management yields more agentic behavior*, effectively turning static LLMs into more autonomous problem-solvers.

One academic example comes from the Reflexion framework (discussed earlier). Here the authors explicitly treat the model’s outputs as *feedback into context*: after each trial, the agent’s self-reflection is appended to an episodic memory store. This transforms a plain LLM into a learning agent without gradient updates - purely by *augmenting its context* on the next trial with those reflections. The results (huge gains in coding task accuracy) demonstrate that context engineering (in this case, adding a loop of generating and re-inserting reflective context) can imbue an agent with iterative improvement capabilities. The implication is that many behaviors we want (learning from mistakes, remembering long-term goals) can be achieved at the context level, without changes to the model’s weights.

Another study to highlight is the Batch Calibration approach by Zhou *et al.* (2024) which, while focused on prompt bias, reinforces the idea that how we engineer context input can fix or enable capabilities that the base model lacks. Their Batch Calibration method takes a batch of prompts with variations (including empty or null prompts to gauge bias) and uses them to adjust the output probabilities of the main prompt. Essentially, it provides extra context in the form of comparative inputs, and in doing so stabilizes the model’s behavior. They achieved state-of-the-art results on 10+ language understanding tasks just by this clever context manipulation - no fine-tuning, just better context feeding. This again underscores the power of treating the LLM as an engine whose performance we shape via its context.

From the perspective of practical agent building, the LangChain and LlamaIndex teams (as seen in their 2025 blogs) have distilled community knowledge into a set of context engineering patterns. They identify the strategies we covered: writing context to external memory, selecting relevant context via retrieval, compressing context by summarization/trimming, and isolating context in multi-agent or tool-specific workflows. Their libraries now provide utilities to implement these. For example, LlamaIndex’s recent *Workflows* feature allows developers to explicitly script a sequence of LLM calls and tool calls, with control over what context goes into each step. This means one can break a complex job into discrete sub-tasks (e.g. first summarize input, then do a database query, then answer question using results) - effectively hand-crafting context transitions at each phase. By structuring the overall process (often called *orchestration*), one can ensure each LLM invocation has exactly the context it needs and nothing more. This reduces the likelihood of context overflow and makes the agent’s behavior more reliable. In a way, this is a higher-level context engineering on the workflow level rather than token level.

One more notable example is Anthropic’s parallel research agents. Their case study (“How we built our multi-agent research system”) vividly shows context engineering trade-offs. Initially, one might think having a single agent with a 100k token window means it could just read everything sequentially. But their results found that splitting the work among subagents (each with their own 100k window, effectively) and then compressing their findings back to the lead agent was far

more effective. It's context engineering in the sense of *allocating attention*: each agent's context is curated for depth in one area, and the lead agent's context is filled with the distilled outputs of those specialists. The performance boost came largely from the system being able to *spend more total tokens* on the problem (since each subagent could consume tokens in parallel). But that only worked because of a careful context hand-off: the lead agent had to have a context that). This example mirrors the "from contexts to agents" idea: by engineering how context is spread and exchanged between multiple LLMs, we created a more powerful *collective agent* that could do things a single context could not.

Future Directions in Context Engineering

The rapid progress in 2023-2025 suggests several exciting directions for context engineering in AI agents. One obvious trajectory is extending native context lengths - e.g., new transformer architectures and training tricks that push context windows to hundreds of thousands or even millions of tokens. Models like GPT-4 (with 128K context) and Claude 4 (with 200K context) already hint at an era where models can directly consume books or multi-hour transcripts. However, as practitioners note, simply having a bigger window doesn't remove the need for context engineering. Issues like the *lost-in-the-middle* effect and the sheer cost of using huge contexts mean we will still need to carefully select what goes into those windows. Nevertheless, research on efficient attention (such as RFA, FlashAttention, etc.) and position encoding extrapolation continues to expand how far we can push contiguous context. In the future, we might commonly have agents with, say, 1 million token context capability - but we will use context engineering to decide which *1 million tokens out of a billion* it actually sees.

Another promising direction is latent space reasoning and iterative refinement, which aim to decouple reasoning from the token-limited visible context. For example, recent research from the University of Maryland introduced a model called *Huginn* (3.5B) that can "think in the latent space" by iteratively processing its hidden state without emitting more tokens. Essentially, it can perform extra computation on a given input by looping internally, rather than unrolling a huge chain-of-thought in the output context. Early results showed it solving tasks with complex reasoning (like tricky math or multi-step logic) better than standard chain-of-thought prompting, all while not consuming additional context window. If these ideas mature, they could mitigate the need to supply the model with huge explicit contexts (since the model can internally organize and remember interim results). It's akin to giving the model scratchpad memory that doesn't count against the token limit. While still at the research stage, such approaches could transform how we think about context limits - the model might effectively have an *unbounded internal context* and only the final answer (or requested details) need to be output.

We also anticipate more sophisticated context ranking and verification methods. Currently, vector similarity is a staple for relevance, but as noted, it can retrieve imperfect results. Future agents may use a combination of sparse (symbolic) and dense retrieval (neural) to double-check context relevance. We might see LLMs themselves used to judge context snippets *before* using them. For example, one could prompt an LLM: "Here's the user question and a candidate piece of context - is this likely to be useful or correct for answering?" and only include it if the LLM agrees. This kind of semantic verification could reduce the inclusion of distracting or wrong context. There is active work on *LLM-based re-ranking* and *context filtering* to minimize context-induced hallucinations.

Tool use will also become more tightly integrated with context engineering. An agent with dozens of tools might automatically do a preliminary step: “figure out which tools are relevant to the query” (using a description of each tool’s capabilities). Indeed, researchers recently applied retrieval techniques to tool selection, fetching only the most relevant tool descriptions to include in the prompt, which improved tool use success by 3× in experiments. So context engineering is not just about knowledge retrieval, but also *tool retrieval*. The lines between retrieving text and retrieving an action blur - in both cases, the agent must choose a small subset from a large set (documents or tools) to bring into focus.

On the engineering side, we’ll likely see more frameworks and libraries that abstract these patterns so developers don’t reinvent the wheel. We already have LangChain, LlamaIndex, Haystack, etc., providing high-level APIs for memory, retrieval, and so forth. These might evolve with more automation (e.g. auto-summarizers that kick in when needed, intelligent context overflow handlers, etc.). There’s also interest in learning to retrieve - training the agent (or a companion model) to pick which documents to retrieve based on success/failure signals, thereby tuning the relevance scoring over time.

Finally, as context engineering gets more sophisticated, evaluation becomes critical. We need to measure not just if an agent can answer one query correctly, but if it remains consistent and effective over long sessions, if it respects which information should be used (or not used) when, and how sensitive it is to errors in its context. New benchmarks that involve long-horizon tasks, or that stress-test an agent’s ability to recall facts given earlier, will drive development of better context strategies.

Conclusion

Context engineering is rapidly emerging as a cornerstone of building intelligent LLM-based systems. While early attention was on prompt engineering - crafting a single perfect prompt - it has become clear that truly *agentic* AI requires a broader approach. Context engineering asks: *What information does the AI need right now, and how do I provide it optimally?* This encompasses retrieving relevant knowledge, remembering past interactions, structuring the conversation and instructions, compressing or expiring extraneous detail, and coordinating multiple contexts when necessary. By treating the LLM’s context window as a precious resource to be managed, we can greatly extend the capability of AI agents without altering their underlying model weights.

The techniques discussed - from RAG to memory buffers, from summarization to multi-agent orchestration - all serve the same goal of empowering the model to perform complex, extended tasks reliably. Academic research has validated many of these: providing iterative reflections improved problem-solving, adding retrieval yields more factual answers, synthesizing memories produces more human-like long-term behavior, and better prompt/context calibration leads to more robust performance. In each case, the differentiator wasn’t a new transformer architecture or more training data - it was a smarter way of feeding context to the model. As one practitioner put it, *“The magic isn’t in a smarter model... It’s in providing the right context for the right task.”*.

For AI researchers and engineers, context engineering opens up a rich design space. It requires thinking about how to break tasks into pieces, how to leverage external knowledge, and how to maintain a coherent thread of information throughout an agent’s operation. In many ways, it is a

multidisciplinary challenge: part prompt design, part information retrieval, part systems engineering. As we push the boundaries of what AI agents can do, mastering context engineering will be key to moving from brittle demos to robust, “magical” AI products. With a solid context foundation, even today’s LLMs can achieve impressive feats - and tomorrow’s models, paired with advanced context strategies, will be even more capable. The adage “garbage in, garbage out” reminds us that an AI’s output is only as good as its input; context engineering ensures that we’re feeding our AI the very best information and guidance we can, thereby unlocking its full potential.

Other mentions by Author

- jtanruan.medium.com | Written by Jin Tan Ruan, CSE Computer Science - ML Engineer