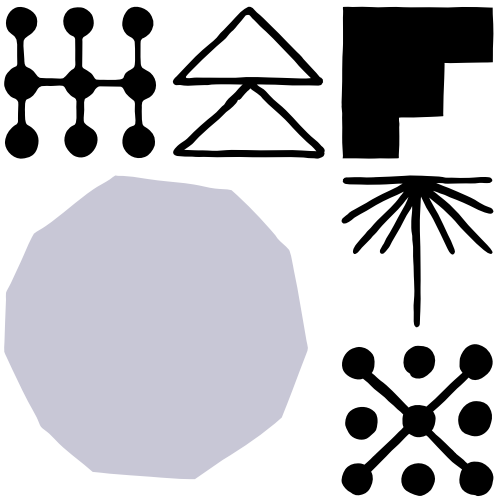


Effective context engineering for AI agents

Published Sep 29, 2025

Context is a critical but finite resource for AI agents. In this post, we explore strategies for effectively curating and managing the context that powers them.

After a few years of prompt engineering being the focus of attention in applied AI, a new term has come to prominence: **context engineering**. Building with language models is becoming less about finding the right words and phrases for your prompts, and more about answering the broader question of “what configuration of context is most likely to generate our model’s desired behavior?"

Context refers to the set of tokens included when sampling from a large-language model (LLM). The **engineering** problem at hand is optimizing the utility of those tokens against the inherent constraints of LLMs in order to

consistently achieve a desired outcome. Effectively wrangling LLMs often requires *thinking in context* — in other words: considering the holistic state available to the LLM at any given time and what potential behaviors that state might yield.

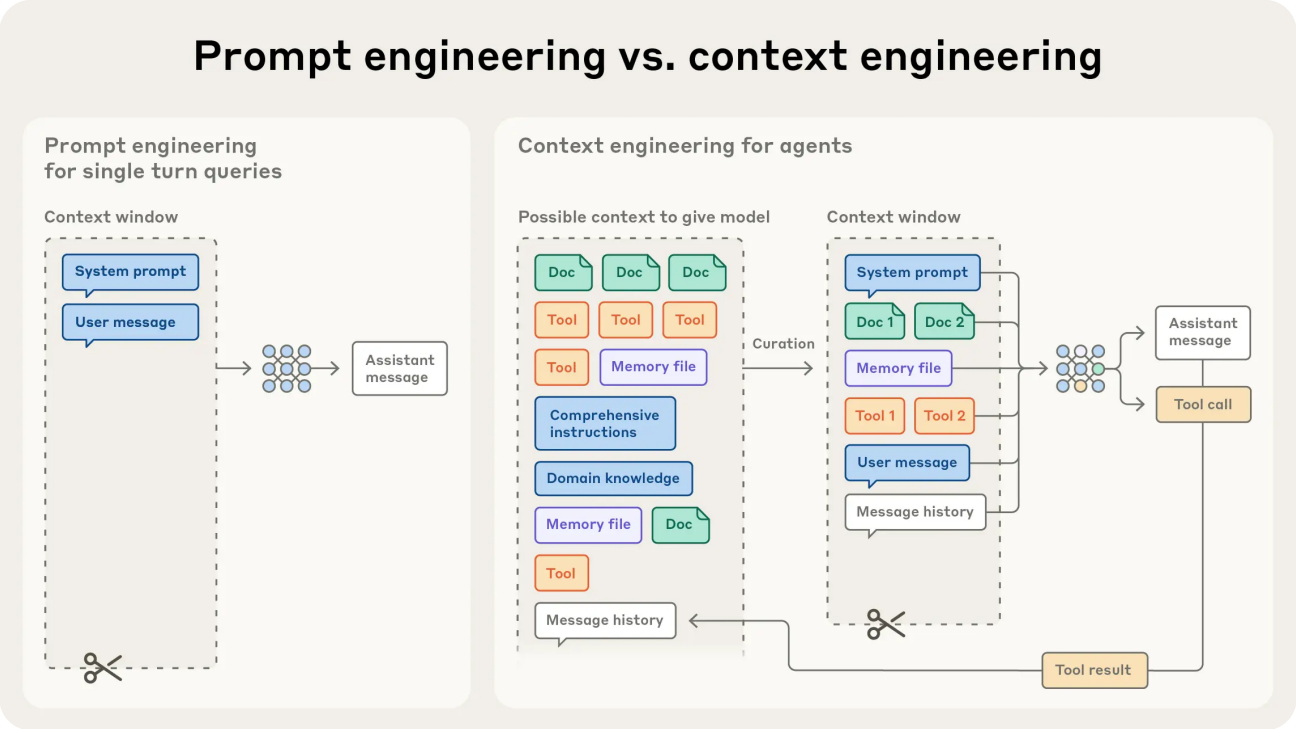
In this post, we'll explore the emerging art of context engineering and offer a refined mental model for building steerable, effective agents.

Context engineering vs. prompt engineering

At Anthropic, we view context engineering as the natural progression of prompt engineering. Prompt engineering refers to methods for writing and organizing LLM instructions for optimal outcomes (see our docs for an overview and useful prompt engineering strategies). **Context engineering** refers to the set of strategies for curating and maintaining the optimal set of tokens (information) during LLM inference, including all the other information that may land there outside of the prompts.

In the early days of engineering with LLMs, prompting was the biggest component of AI engineering work, as the majority of use cases outside of everyday chat interactions required prompts optimized for one-shot classification or text generation tasks. As the term implies, the primary focus of prompt engineering is how to write effective prompts, particularly system prompts. However, as we move towards engineering more capable agents that operate over multiple turns of inference and longer time horizons, we need strategies for managing the entire context state (system instructions, tools, Model Context Protocol (MCP), external data, message history, etc).

An agent running in a loop generates more and more data that *could* be relevant for the next turn of inference, and this information must be cyclically refined. Context engineering is the art and science of curating what will go into the limited context window from that constantly evolving universe of possible information.



In contrast to the discrete task of writing a prompt, context engineering is iterative and the curation phase happens each time we decide what to pass to the model.

Why context engineering is important to building capable agents

Despite their speed and ability to manage larger and larger volumes of data, we've observed that LLMs, like humans, lose focus or experience confusion at a certain point. Studies on needle-in-a-haystack style benchmarking have uncovered the concept of context rot: as the number of tokens in the context window increases, the model's ability to accurately recall information from that context decreases.

While some models exhibit more gentle degradation than others, this characteristic emerges across all models. Context, therefore, must be treated as a finite resource with diminishing marginal returns. Like humans, who have limited working memory capacity, LLMs have an "attention budget" that they draw on when parsing large volumes of context. Every new token introduced depletes this budget by some amount, increasing the need to carefully curate the tokens available to the LLM.

This attention scarcity stems from architectural constraints of LLMs. LLMs are based on the transformer architecture, which enables every token to attend to every other token across the entire context. This results in n^2 pairwise relationships for n tokens.

As its context length increases, a model's ability to capture these pairwise relationships gets stretched thin, creating a natural tension between context size and attention focus. Additionally, models develop their attention patterns from training data distributions where shorter sequences are typically more common than longer ones. This means models have less experience with, and fewer specialized parameters for, context-wide dependencies.

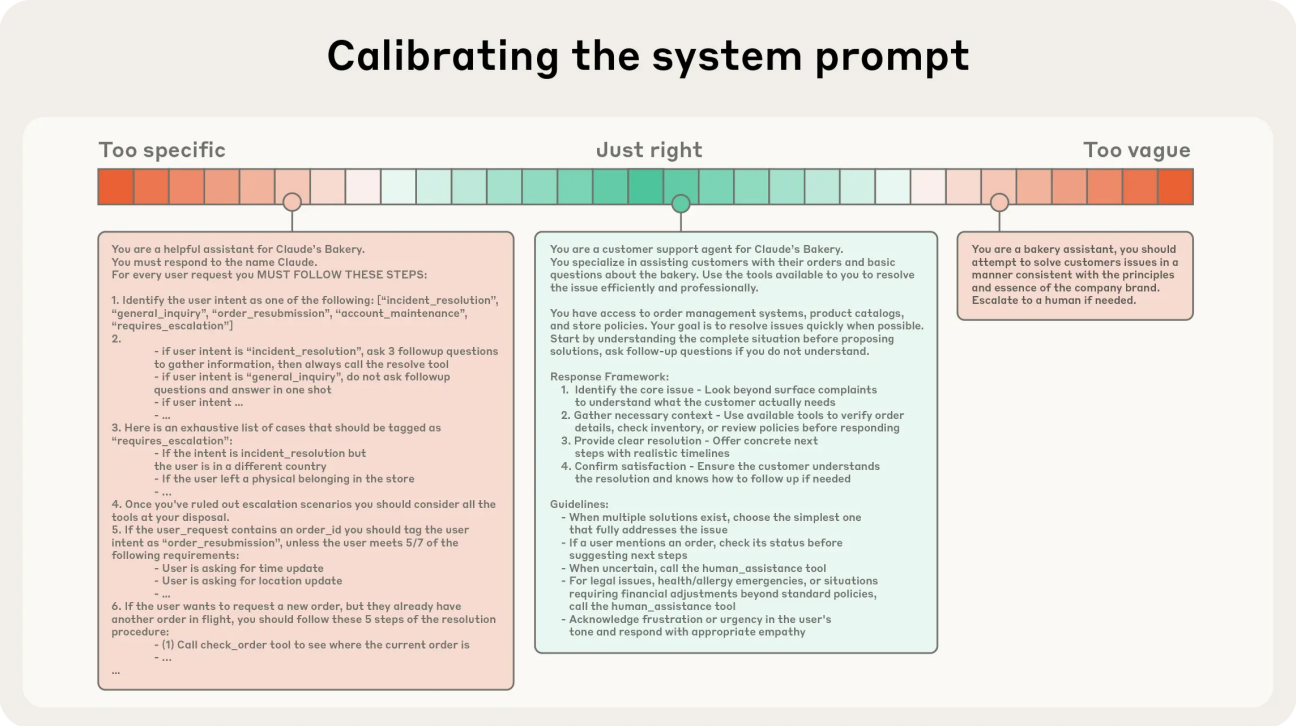
Techniques like position encoding interpolation allow models to handle longer sequences by adapting them to the originally trained smaller context, though with some degradation in token position understanding. These factors create a performance gradient rather than a hard cliff: models remain highly capable at longer contexts but may show reduced precision for information retrieval and long-range reasoning compared to their performance on shorter contexts.

These realities mean that thoughtful context engineering is essential for building capable agents.

The anatomy of effective context

Given that LLMs are constrained by a finite attention budget, *good* context engineering means finding the *smallest possible* set of high-signal tokens that maximize the likelihood of some desired outcome. Implementing this practice is much easier said than done, but in the following section, we outline what this guiding principle means in practice across the different components of context.

System prompts should be extremely clear and use simple, direct language that presents ideas at the *right altitude* for the agent. The right altitude is the Goldilocks zone between two common failure modes. At one extreme, we see engineers hardcoding complex, brittle logic in their prompts to elicit exact agentic behavior. This approach creates fragility and increases maintenance complexity over time. At the other extreme, engineers sometimes provide vague, high-level guidance that fails to give the LLM concrete signals for desired outputs or falsely assumes shared context. The optimal altitude strikes a balance: specific enough to guide behavior effectively, yet flexible enough to provide the model with strong heuristics to guide behavior.



At one end of the spectrum, we see brittle if-else hardcoded prompts, and at the other end we see prompts that are overly general or falsely assume shared context.

We recommend organizing prompts into distinct sections (like `<background_information>`, `<instructions>`, `## Tool guidance`, `## Output description`, etc) and using techniques like XML tagging or Markdown headers to delineate these sections, although the exact formatting of prompts is likely becoming less important as models become more capable.

Regardless of how you decide to structure your system prompt, you should be striving for the minimal set of information that fully outlines your expected behavior. (Note that minimal does not necessarily mean short; you still need to give the agent sufficient information up front to ensure it adheres to the desired behavior.) It's best to start by testing a minimal prompt with the best model available to see how it performs on your task, and then add clear instructions and examples to improve performance based on failure modes found during initial testing.

Tools allow agents to operate with their environment and pull in new, additional context as they work. Because tools define the contract between agents and their information/action space, it's extremely important that tools promote efficiency, both by returning information that is token efficient and by encouraging efficient agent behaviors.

In Writing tools for AI agents – with AI agents, we discussed building tools that are well understood by LLMs and have minimal overlap in functionality. Similar to the functions of a well-designed codebase, tools should be self-contained, robust to error, and extremely clear with respect to their intended use. Input parameters should similarly be descriptive, unambiguous, and play to the inherent strengths of the model.

One of the most common failure modes we see is bloated tool sets that cover too much functionality or lead to ambiguous decision points about which tool to use. If a human engineer can't definitively say which tool should be used in a given situation, an AI agent can't be expected to do better. As we'll discuss later, curating a minimal viable set of tools for the agent can also lead to more reliable maintenance and pruning of context over long interactions.

Providing examples, otherwise known as few-shot prompting, is a well known best practice that we continue to strongly advise. However, teams will often stuff a laundry list of edge cases into a prompt in an attempt to articulate every possible rule the LLM should follow for a particular task. We do not recommend this. Instead, we recommend working to curate a set of diverse, canonical examples that effectively portray the expected behavior of the agent. For an LLM, examples are the "pictures" worth a thousand words.

Our overall guidance across the different components of context (system prompts, tools, examples, message history, etc) is to be thoughtful and keep your context informative, yet tight. Now let's dive into dynamically retrieving context at runtime.

Context retrieval and agentic search

In Building effective AI agents, we highlighted the differences between LLM-based workflows and agents. Since we wrote that post, we've gravitated towards a simple definition for agents: LLMs autonomously using tools in a loop.

Working alongside our customers, we've seen the field converging on this simple paradigm. As the underlying models become more capable, the level of autonomy of agents can scale: smarter models allow agents to independently navigate nuanced problem spaces and recover from errors.

We're now seeing a shift in how engineers think about designing context for agents. Today, many AI-native applications employ some form of embedding-based pre-inference time retrieval to surface important context for the agent to reason over. As the field transitions to more agentic approaches, we increasingly see teams augmenting these retrieval systems with "just in time" context strategies.

Rather than pre-processing all relevant data up front, agents built with the "just in time" approach maintain lightweight identifiers (file paths, stored queries, web links, etc.) and use these references to dynamically load data into context at runtime using tools. Anthropic's agentic coding solution Claude Code uses this approach to perform complex data analysis over large databases. The model can write targeted queries, store results, and leverage Bash commands like head and tail to analyze large volumes of data without ever loading the full data objects into context. This approach mirrors human cognition: we generally don't memorize entire corpuses of information, but rather introduce external organization and indexing systems like file systems, inboxes, and bookmarks to retrieve relevant information on demand.

Beyond storage efficiency, the metadata of these references provides a mechanism to efficiently refine behavior, whether explicitly provided or intuitive. To an agent operating in a file system, the presence of a file named `test_utils.py` in a `tests` folder implies a different purpose than a file with the same name located in `src/core_logic/` Folder hierarchies, naming conventions, and timestamps all provide important signals that help both humans and agents understand how and when to utilize information.

Letting agents navigate and retrieve data autonomously also enables progressive disclosure—in other words, allows agents to incrementally discover relevant context through exploration. Each interaction yields context that informs the next decision: file sizes suggest complexity; naming conventions hint at purpose; timestamps can be a proxy for relevance. Agents can assemble understanding layer by layer, maintaining only what's necessary in working memory and leveraging note-taking strategies for additional persistence. This self-managed context window keeps the agent focused on relevant subsets rather than drowning in exhaustive but potentially irrelevant information.

Of course, there's a trade-off: runtime exploration is slower than retrieving pre-computed data. Not only that, but opinionated and thoughtful engineering is required to ensure that an LLM has the right tools and heuristics for effectively navigating its information landscape. Without proper guidance, an agent can waste context by misusing tools, chasing dead-ends, or failing to identify key information.

In certain settings, the most effective agents might employ a hybrid strategy, retrieving some data up front for speed, and pursuing further autonomous exploration at its discretion. The decision boundary for the 'right' level of autonomy depends on the task. Claude Code is an agent that employs this hybrid model: CLAUDE.md files are naively dropped into context up front, while primitives like glob and grep allow it to navigate its environment and retrieve files just-in-time, effectively bypassing the issues of stale indexing and complex syntax trees.

The hybrid strategy might be better suited for contexts with less dynamic content, such as legal or finance work. As model capabilities improve, agentic design will trend towards letting intelligent models act intelligently, with progressively less human curation. Given the rapid pace of progress in the field, "do the simplest thing that works" will likely remain our best advice for teams building agents on top of Claude.

Context engineering for long-horizon tasks

Long-horizon tasks require agents to maintain coherence, context, and goal-directed behavior over sequences of actions where the token count exceeds the LLM's context window. For tasks that span tens of minutes to multiple hours of continuous work, like large codebase migrations or comprehensive research projects, agents require specialized techniques to work around the context window size limitation.

Waiting for larger context windows might seem like an obvious tactic. But it's likely that for the foreseeable future, context windows of all sizes will be subject to context pollution and information relevance concerns—at least for situations where the strongest agent performance is desired. To enable agents to work effectively across extended time horizons, we've developed a few techniques that address these context pollution constraints directly: compaction, structured note-taking, and multi-agent architectures.

Compaction

Compaction is the practice of taking a conversation nearing the context window limit, summarizing its contents, and reinitiating a new context window with the summary. Compaction typically serves as the first lever in context engineering to drive better long-term coherence. At its core, compaction distills the contents of a context window in a high-fidelity manner, enabling the agent to continue with minimal performance degradation.

In Claude Code, for example, we implement this by passing the message history to the model to summarize and compress the most critical details. The model preserves architectural decisions, unresolved bugs, and implementation details while discarding redundant tool outputs or messages. The agent can then continue with this compressed context plus the five most recently accessed files. Users get continuity without worrying about context window limitations.

The art of compaction lies in the selection of what to keep versus what to discard, as overly aggressive compaction can result in the loss of subtle but critical context whose importance only becomes apparent later. For engineers implementing compaction systems, we recommend carefully tuning your prompt on complex agent traces. Start by maximizing recall to ensure your compaction prompt captures every relevant piece of information from the trace, then iterate to improve precision by eliminating superfluous content.

An example of low-hanging superfluous content is clearing tool calls and results – once a tool has been called deep in the message history, why would the agent need to see the raw result again? One of the safest lightest touch forms of compaction is tool result clearing, most recently launched as a feature on the Claude Developer Platform.

Structured note-taking

Structured note-taking, or agentic memory, is a technique where the agent regularly writes notes persisted to memory outside of the context window. These notes get pulled back into the context window at later times.

This strategy provides persistent memory with minimal overhead. Like Claude Code creating a to-do list, or your custom agent maintaining a NOTES.md file, this simple pattern allows the agent to track progress across complex tasks, maintaining critical context and dependencies that would otherwise be lost across dozens of tool calls.

Claude playing Pokémon demonstrates how memory transforms agent capabilities in non-coding domains. The agent maintains precise tallies across thousands of game steps—tracking objectives like "for the last 1,234 steps I've been training my Pokémon in Route 1, Pikachu has gained 8 levels toward the target of 10." Without any prompting about memory structure, it develops maps of explored regions, remembers which key achievements it has unlocked, and maintains strategic notes of combat strategies that help it learn which attacks work best against different opponents.

After context resets, the agent reads its own notes and continues multi-hour training sequences or dungeon explorations. This coherence across summarization steps enables long-horizon strategies that would be impossible when keeping all the information in the LLM's context window alone.

As part of our Sonnet 4.5 launch, we released a memory tool in public beta on the Claude Developer Platform that makes it easier to store and consult information outside the context window through a file-based system. This allows agents to build up knowledge bases over time, maintain project state across sessions, and reference previous work without keeping everything in context.

Sub-agent architectures

Sub-agent architectures provide another way around context limitations. Rather than one agent attempting to maintain state across an entire project, specialized sub-agents can handle focused tasks with clean context windows. The main agent coordinates with a high-level plan while subagents perform deep technical work or use tools to find relevant information. Each subagent might explore extensively, using tens of thousands of tokens or more, but returns only a condensed, distilled summary of its work (often 1,000-2,000 tokens).

This approach achieves a clear separation of concerns—the detailed search context remains isolated within sub-agents, while the lead agent focuses on synthesizing and analyzing the results. This pattern, discussed in How we built our multi-agent research system, showed a substantial improvement over single-agent systems on complex research tasks.

The choice between these approaches depends on task characteristics. For example:

- Compaction maintains conversational flow for tasks requiring extensive back-and-forth;
- Note-taking excels for iterative development with clear milestones;
- Multi-agent architectures handle complex research and analysis where parallel exploration pays dividends.

Even as models continue to improve, the challenge of maintaining coherence across extended interactions will remain central to building more effective agents.

Conclusion

Context engineering represents a fundamental shift in how we build with LLMs. As models become more capable, the challenge isn't just crafting the perfect prompt—it's thoughtfully curating what information enters the model's limited attention budget at each step. Whether you're implementing compaction for long-horizon tasks, designing token-efficient tools, or enabling agents to explore their environment just-in-time, the guiding principle remains the same: find the smallest set of high-signal tokens that maximize the likelihood of your desired outcome.

The techniques we've outlined will continue evolving as models improve. We're already seeing that smarter models require less prescriptive engineering, allowing agents to operate with more autonomy. But even as capabilities scale, treating context as a precious, finite resource will remain central to building reliable, effective agents.

Get started with context engineering in the Claude Developer Platform today, and access helpful tips and best practices via our memory and context management cookbook.

Acknowledgements

Written by Anthropic's Applied AI team: Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield, with contributions from team members Rafi Ayub, Hannah Moran, Cal Rueb, and Connor Jennings. Special thanks to Molly Vorwerck, Stuart Ritchie, and Maggie Vo for their support.

Get the developer newsletter

Product updates, how-tos, community spotlights, and more. Delivered monthly to your inbox.

Enter your email

Please provide your email address if you'd like to receive our monthly developer newsletter. You can unsubscribe at any time.

Products

Claude

Claude Code

Max plan

Team plan

Enterprise plan

Download app

Pricing

Log in to Claude

Models

Opus

Sonnet

Haiku

Solutions

AI agents

Code modernization

Coding

Customer support

Education

Financial services

Government

Claude Developer Platform

Overview

Developer docs

Pricing

Amazon Bedrock

Google Cloud's Vertex AI

Console login

Learn

Courses

Connectors

Customer stories

Engineering at Anthropic

Events

Powered by Claude

Service partners

Startups program

Company

Anthropic

Careers

Economic Futures

Research

News

Responsible Scaling Policy

Security and compliance

Transparency

Help and security

Availability

Status

Support center

Terms and policies

Privacy choices

Privacy policy

Responsible disclosure policy

Terms of service: Commercial

Terms of service: Consumer

Usage policy

© 2025 Anthropic PBC