Resource link and reference

-github agentic-design-workflow

-google agentic design handbook

Reflection

Self Aware not passive executor: The Reflection pattern is crucial for building agents that can produce high-quality outputs, handle nuanced tasks, and exhibit a degree of self-awareness and adaptability.

With Memory: Without memory, each reflection is a self-contained event; with memory, reflection becomes a cumulative process where each cycle builds upon the last, leading to more intelligent and context-aware refinement.

Use Examples:

1. Creative Writing and Content Generation:

Refining generated text, stories, poems, or marketing copy.

- **Use Case:** An agent writing a blog post.
- **Reflection:** Generate a draft, critique it for flow, tone, and clarity, then rewrite based on the critique. Repeat until the post meets quality standards.
- **Benefit:** Produces more polished and effective content.

At Glance

What: An agent's initial output is often suboptimal, suffering from inaccuracies, incompleteness, or a failure to meet complex requirements. Basic agentic workflows lack a built-in process for the agent to recognize and fix its own errors. This is solved by having the agent evaluate its own work or, more robustly, by introducing a separate logical agent to act as a critic, preventing the initial response from being the final one regardless of quality.

Why: The Reflection pattern offers a solution by introducing a mechanism for self-correction and refinement. It establishes a feedback loop where a "producer" agent generates an output, and then a "critic" agent (or the producer itself) evaluates it against predefined criteria. This critique is then used to generate an improved version. This iterative process of generation, evaluation, and refinement

progressively enhances the quality of the final result, leading to more accurate, coherent, and reliable outcomes.

Rule of thumb: Use the Reflection pattern when the quality, accuracy, and detail of the final output are more important than speed and cost. It is particularly effective for tasks like generating polished long-form content, writing and debugging code, and creating detailed plans. Employ a separate critic agent when tasks require high objectivity or specialized evaluation that a generalist producer agent might miss.

References

Here are some resources for further reading on the Reflection pattern and related concepts:

1. Training Language Models to Self-Correct via Reinforcement Learning, https://arxiv.org/abs/2409.12917
2. LangChain Expression Language (LCEL) Documentation: https://python.langchain.com/docs/introduction/
3. LangGraph Documentation:https://www.langchain.com/langgraph
4. Google Agent Developer Kit (ADK) Documentation (Multi-Agent Systems): https://google.github.io/adk-docs/agents/multi-agents/

LLM-as-Critic, CVPR:
We introduce LLaVA-Critic, the first open-source large multimodal model (LMM) designed as a generalist evaluator to assess performance across a wide range of multimodal tasks. LLaVA-Critic is trained using a high-quality critic instruction-following dataset that incorporates diverse evaluation criteria and scenarios. Our experiments demonstrate the model's effectiveness in two key areas: (i) LMM-as-a-Judge, where LLaVA-Critic provides reliable evaluation scores, performing on par with or surpassing GPT models on multiple evaluation benchmarks; and (ii) Preference Learning, where it generates reward signals for preference learning, enhancing model alignment capabilities. This work underscores the potential of open-source LMMs in self-critique and evaluation, setting the stage for future research into scalable, superhuman alignment feedback mechanisms for LMMs.

Function calling

Basic: The Tool Use pattern, often implemented through a mechanism called Function Calling, enables an agent to interact with external APIs, databases, services, or even execute code. It allows the LLM at the core of the agent to decide when and how to use a specific external function based on the user's request or the current state of the task.

Extended: While "function calling" aptly describes invoking specific, predefined code functions, it's useful to consider the more expansive concept of "tool calling." This broader term acknowledges that an agent's capabilities can extend far beyond

simple function execution. A "tool" can be a traditional function, but it can also be a complex API endpoint, a request to a database, or even an instruction directed at another specialized agent. This perspective allows us to envision more sophisticated systems where, for instance, a primary agent might delegate a complex data analysis task to a dedicated "analyst agent" or query an external knowledge base through its API. Thinking in terms of "tool calling" better captures the full potential of agents to act as orchestrators across a diverse ecosystem of digital resources and other intelligent entities.

Rule of thumb

Use the Tool Use pattern whenever an agent needs to break out of the LLM's internal knowledge and interact with the outside world. This is essential for tasks requiring real-time data (e.g., checking weather, stock prices), accessing private or proprietary information (e.g., querying a company's database), performing precise calculations, executing code, or triggering actions in other systems (e.g., sending an email, controlling smart devices)

GOLDEN RULES by claude-code

But I want my math agent to handle ALL calculation tasks. My edit and draw agent mainly focus on the SHAPES, COLORS, and other format. So I remove the math tools from edit agent, and enable a similar calculate_xxx_with_math_agent_tool to it. Then I want to TOTALLY rewrite the tool system of my math_tools.py following the GOLDEN RULES provided by claude-code.

NOW IMAGINE you are exactly my math-agent, you are given a task description and original code, but you just know the path, maybe you can inference its position and directions, BUT the problems are, which tool you want to use to help you make decision to NEXT bezier segment? Where is it end point? Which symbol will you want, C/Q/T/S, and how to choose the control points? These are very difficult, and are exactly the CORE of my design agent.

And I want devide them into three categories, each category can have **multiple** tools. (Or you can decide by yourself).

1. design_**： E.g. The base symmetric point from other point/line, can get the symmetric control points or end points,

2. function**： E.g. key points selected from sigmoid curve.

3. auxiliary_**: E.g. You may use auxiliary line/grid to determine the location?

etc. ... I am not sure now. BUT the KEY is: ALL your given tool MUST be promised to have specific and non-overlap function, and you PROMISE they will be use in some setting. DONNOT give the unnecessary tools. Tools must be as less and as useful. AS the GOLDEN RULE below.

———————————————————————

I will also give you some GOLDEN RULES by claude code.

- Accurate and powerful: We recommend building a few thoughtful tools targeting specific high-impact workflows, which match your evaluation tasks and scaling up from there. In the address book case, you might choose to implement a search_contacts or message_contact tool instead of a list_contacts tool. Make sure each tool you build has a clear, distinct purpose. Tools should enable agents to subdivide and solve tasks in much the same way that a human would, given access to the same underlying resources, and simultaneously reduce the context that would have otherwise been consumed by intermediate outputs.

Too many tools or overlapping tools can also distract agents from pursuing efficient strategies. Careful, selective planning of the tools you build (or don't build) can really pay off.

- Namespacing (grouping related tools under common prefixes) can help delineate boundaries between lots of tools; MCP clients sometimes do this by default. For example, namespacing tools by service (e.g., asana_search, jira_search) and by resource (e.g., asana_projects_search, asana_users_search), can help agents select the right tools at the right time.

- Returning meaningful context from your tools. Agents also tend to grapple with natural language names, terms, or identifiers significantly more successfully than they do with cryptic identifiers. We've found that merely resolving arbitrary alphanumeric UUIDs to more semantically meaningful and interpretable language (or even a 0-indexed ID scheme) significantly improves Claude's precision in retrieval tasks by reducing hallucinations.

You can enable both by exposing a simple response_format enum parameter in your tool, allowing your agent to control whether tools return "concise" or "detailed" responses (images below

==========

Now let's begin with our math expert agent which can reproduce a complex path with natural task description!

Planning

Overview

In the context of AI, it's helpful to think of a planning agent as a specialist to whom you delegate a complex goal. When you ask it to "organize a team offsite," you are

defining the what—the objective and its constraints—but not the how. The agent's core task is to autonomously chart a course to that goal. It must first understand the initial state (e.g., budget, number of participants, desired dates) and the goal state (a successfully booked offsite), and then discover the optimal sequence of actions to connect them. The plan is not known in advance; it is created in response to the request.

A hallmark of this process is adaptability. An initial plan is merely a starting point, not a rigid script. The agent's real power is its ability to incorporate new information and steer the project around obstacles.However, it is crucial to recognize the trade-off between flexibility and predictability. Dynamic planning is a specific tool, not a universal solution. When a problem's solution is already well-understood and repeatable, constraining the agent to a predetermined, fixed workflow is more effective. This approach limits the agent's autonomy to reduce uncertainty and the risk of unpredictable behavior, guaranteeing a reliable and consistent outcome. Therefore, the decision to use a planning agent versus a simple task-execution agent hinges on a single question: does the "how" need to be discovered, or is it already known?

Use Examples

- In procedural task automation, planning is used to orchestrate complex workflows.
- In robotics and autonomous navigation, planning is fundamental for state-space traversal.
- In structured information synthesis. When tasked with generating a complex output like a research report, an agent can formulate a plan that includes distinct phases for information gathering, data summarization, content structuring, and iterative refinement

---

Keypoint

Structured: Without a structured approach, an agentic system struggles to handle multifaceted requests that involve multiple steps and dependencies. It transforms a simple reactive agent into a strategic executor that can proactively work towards a complex objective and even adapt its plan if necessary.

Well-suited: LLMs are particularly well-suited for first creating a coherent plan to address a goal, which involves decomposing a high-level objective into a sequence of smaller, actionable steps or sub-goals. This allows the system to manage

complex workflows, orchestrate various tools, and handle dependencies in a logical order.

Comments:
What if LLM should come up uncompleted plan and execute partly? Are the training data enough. Workflow may be plan first -> execution-> observe and plan next -> ……

Deepsearch tools/websites

1. Google DeepResearch (Gemini Feature): gemini.google.com
2. OpenAI ,Introducing deep research https://openai.com/index/introducing-deep-research/
3. Perplexity, Introducing Perplexity Deep Research, https://www.perplexity.ai/hub/blog/introducing-perplexity-deep-research

Memory

Effective memory management is crucial for intelligent agents to retain information. Agents require different types of memory, much like humans, to operate efficiently.

Short-Term Memory (Contextual Memory): Similar to working memory, this holds information currently being processed or recently accessed. This window contains recent messages, agent replies, tool usage results, and agent reflections from the current interaction, Efficient short-term memory management involves keeping the most relevant information within this limited space, possibly through techniques like summarizing older conversation segments or emphasizing key details.

Long-Term Memory (Persistent Memory): Data is typically stored outside the agent's immediate processing environment, often in databases, knowledge graphs, or vector databases.

Usage

history, learn, personalize interactions, and manage complex, time-dependent problems.

Langchain & Langgraph Hands-on-code

ChatMessageHistory: Manual Memory Management. For direct and simple control over a conversation's history outside of a formal chain, the ChatMessageHistory class is ideal. It allows for the manual tracking of dialogue exchanges.

```
from langchain.memory import ChatMessageHistory

# Initialize the history object
history = ChatMessageHistory()

# Add user and AI messages
history.add_user_message("I'm heading to New York next week.")
history.add_ai_message("Great! It's a fantastic city.")

# Access the list of messages
print(history.messages)
```

ConversationBufferMemory: Automated Memory for Chains. For integrating memory directly into chains, ConversationBufferMemory is a common choice. It holds a buffer of the conversation and makes it available to your prompt. Its behavior can be customized with two key parameters:

- memory_key: A string that specifies the variable name in your prompt that will hold the chat history. It defaults to "history".
- return_messages: A boolean that dictates the format of the history.
    - If False (the default), it returns a single formatted string, which is ideal for standard LLMs.
    - If True, it returns a list of message objects, which is the recommended format for Chat Models.

Memory Bank and Information Retrieval (RAG): Agents designed for answering questions access a knowledge base, their long-term memory, often implemented within Retrieval Augmented Generation (RAG). The agent retrieves relevant documents or data to inform its responses.

```
from langgraph.store.memory import InMemoryStore

# A placeholder for a real embedding function
def embed(texts: list[str]) -> list[list[float]]:
   # In a real application, use a proper embedding model
   return [[1.0, 2.0] for _ in texts]

# Initialize an in-memory store. For production, use a database-
backed store.
store = InMemoryStore(index={"embed": embed, "dims": 2})

# Define a namespace for a specific user and application context
user_id = "my-user"
application_context = "chitchat"
namespace = (user_id, application_context)

# 1. Put a memory into the store
```

```
store.put(
   namespace,
   "a-memory",  # The key for this memory
   {
       "rules": [
           "User likes short, direct language",
           "User only speaks English & python",
       ],
       "my-key": "my-value",
   },
)

# 2. Get the memory by its namespace and key
item = store.get(namespace, "a-memory")
print("Retrieved Item:", item)

# 3. Search for memories within the namespace, filtering by content
# and sorting by vector similarity to the query.
items = store.search(
```

Key Takeaways

- Memory is super important for agents to keep track of things, learn, and personalize interactions.
- Conversational AI relies on both short-term memory for immediate context within a single chat and long-term memory for persistent knowledge across multiple sessions.
- Short-term memory (the immediate stuff) is temporary, often limited by the LLM's context window or how the framework passes context.
- Long-term memory (the stuff that sticks around) saves info across different chats using outside storage like vector databases and is accessed by searching.
- LangGraph enables advanced, long-term memory by using a store to save and retrieve semantic facts, episodic experiences, or even updatable procedural rules across different user sessions.
    - Update the instructions as tools:

```
Node that updates the agent's instructions
def update_instructions(state: State, store: BaseStore):
   namespace = ("instructions",)
   # Get the current instructions from the store
   current_instructions = store.search(namespace)[0]
```

```
   # Create a prompt to ask the LLM to reflect on the conversation
   # and generate new, improved instructions
   prompt = prompt_template.format(
       instructions=current_instructions.value["instructions"],
       conversation=state["messages"]
   )

   # Get the new instructions from the LLM
   output = llm.invoke(prompt)
   new_instructions = output['new_instructions']

   # Save the updated instructions back to the store
   store.put(("agent_instructions",), "agent_a", {"instructions":
new_instructions})

# Node that uses the instructions to generate a response
def call_model(state: State, store: BaseStore):
   namespace = ("agent_instructions", )
   # Retrieve the latest instructions from the store
   instructions = store.get(namespace, key="agent_a")[0]

   # Use the retrieved instructions to format the prompt
   prompt =
prompt_template.format(instructions=instructions.value["instructions
"])
   # ... application logic continues
```

- Memory Bank is a managed service that provides agents with persistent, long-term memory by automatically extracting, storing, and recalling user-specific information to enable personalized

---

Rule of thumb:

Use this pattern when an agent needs to do more than answer a single question. It is essential for agents that must maintain context throughout a conversation, track progress in multi-step tasks, or personalize interactions by recalling user preferences and history. Implement memory management whenever the agent is expected to learn or adapt based on past successes, failures, or newly acquired information.

References

1. LangGraph Memory, https://langchain-ai.github.io/langgraph/concepts/memory/

Prompting

The core of Prompting

Basic: This appendix details various prompting techniques that extend beyond basic interaction methods. It explores methodologies for structuring complex requests, enhancing the model's reasoning abilities, controlling output formats, and integrating external information. These techniques are applicable to building a range of applications, from simple chatbots to complex multi-agent systems, and can improve the performance and reliability of agentic applications.

Atomic: Agentic patterns, the architectural structures for building intelligent systems, are detailed in the main chapters. These patterns define how agents plan, utilize tools, manage memory, and collaborate. The efficacy of these agentic systems is contingent upon their ability to interact meaningfully with language models.

Basic Rules of Thumb

Precise

Use verbs

Advanced Automatic Prompting

Automatic Prompt Engineering (APE)

Recognizing that crafting effective prompts can be a complex and iterative process, Automatic Prompt Engineering (APE) explores using language models themselves to generate, evaluate, and refine prompts. This method aims to automate the prompt writing process, potentially enhancing model performance without requiring extensive human effort in prompt design.

The general idea is to have a "meta-model" or a process that takes a task description and generates multiple candidate prompts.

Using Google Gems

Google's AI "Gems" represent a user-configurable feature within its large language model architecture. Each "Gem" functions as a specialized instance of the core Gemini AI, tailored for specific, repeatable tasks. Users create a Gem by providing it with a set of explicit instructions, which establishes its operational parameters. This initial instruction set defines the Gem's designated purpose, response style, and

knowledge domain. The underlying model is designed to consistently adhere to these pre-defined directives throughout a conversation.

This allows for the creation of highly specialized AI agents for focused applications. For example, a Gem can be configured to function as a code interpreter that only references specific programming libraries. Another could be instructed to analyze data sets, generating summaries without speculative commentary. A different Gem might serve as a translator adhering to a particular formal style guide. This process creates a persistent, task-specific context for the artificial intelligence.

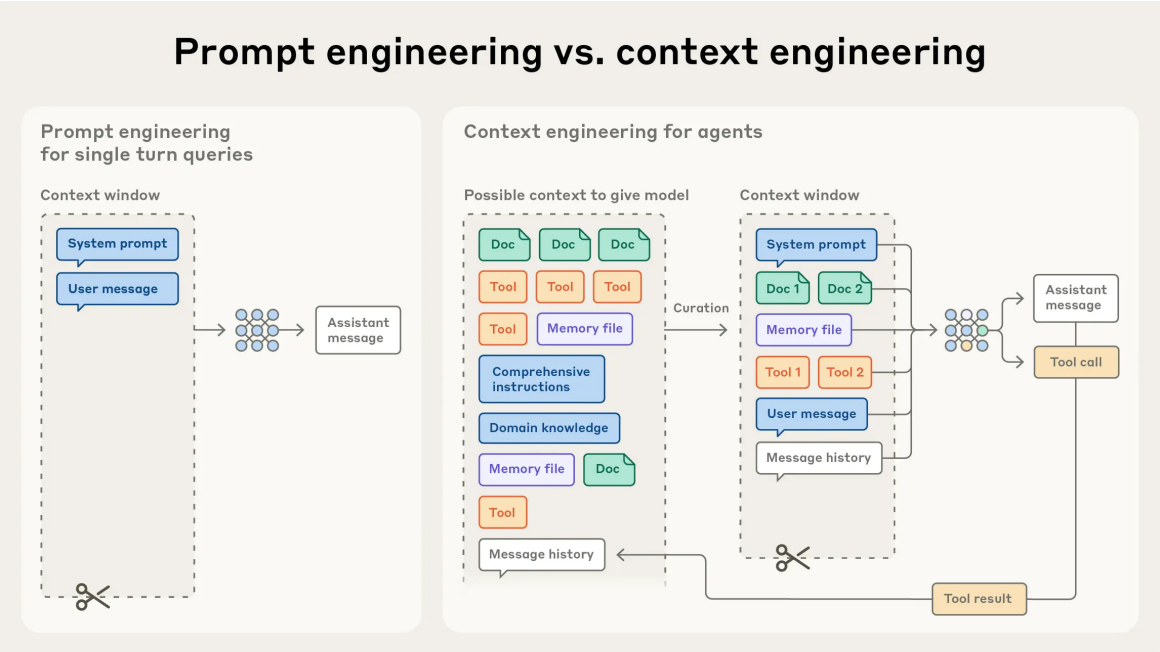
Context Engineering

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.

Why need

1. More and more relevant data are generatesd: 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.



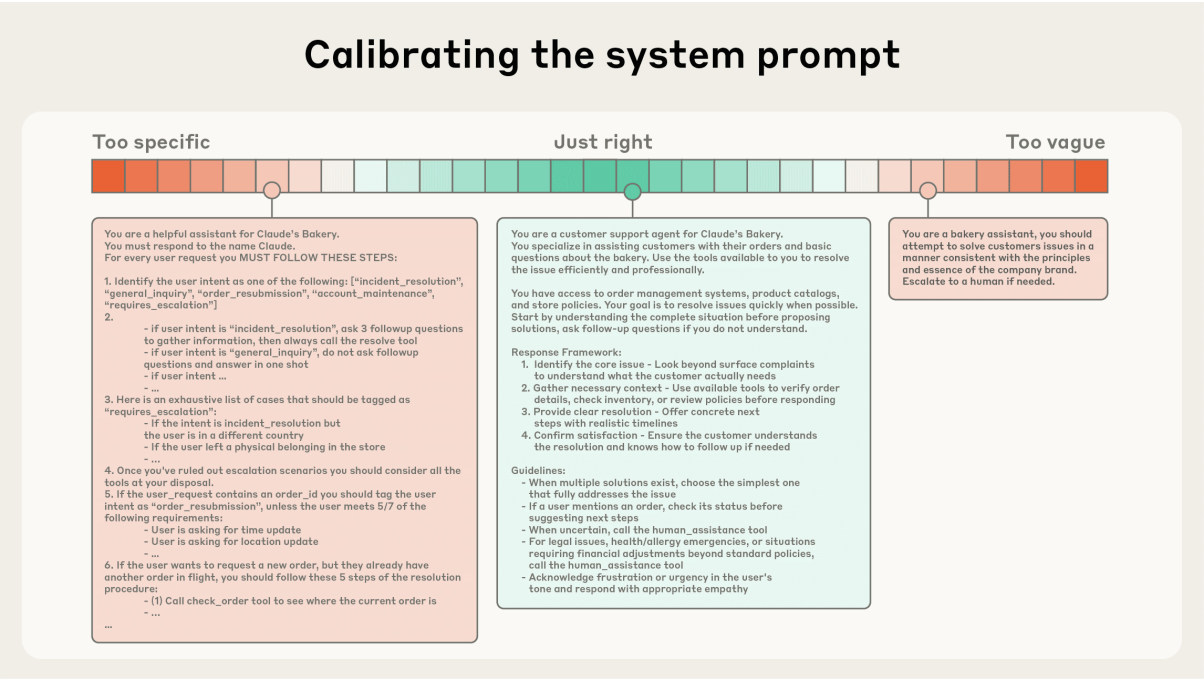
2. Focus is easy to lost: Despite their speed and ability to manage larger and larger volumes of data, we've observed that LLMs, like humans, lose focus or experience confusion at a certain point. Studies on needle-in-a-haystack style benchmarking have uncovered the concept of context rot: as the number of tokens in the context window increases, the model's ability to accurately recall information from that context decreases. Like humans, who have limited working memory capacity, LLMs have an "attention budget" that they draw on when parsing large volumes of context.

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.*

How to define System Prompts?

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 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.effectively, yet flexible enough to provide the model with strong heuristics to guide behavior.

Tools

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. curating a minimal viable set of tools for the agent can also lead to more reliable maintenance and pruning of context over long interactions.

Examples/shots

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.

Overall

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.

In-context-time

Meta-data+retrival tools(tail…)

1. less context token

2. more organized: 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.py. **Folder hierarchies, naming conventions, and timestamps all provide important signals** that help both humans and agents understand how and when to utilize information.

Techniques

code

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

Prompting from minimal.

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.

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.***