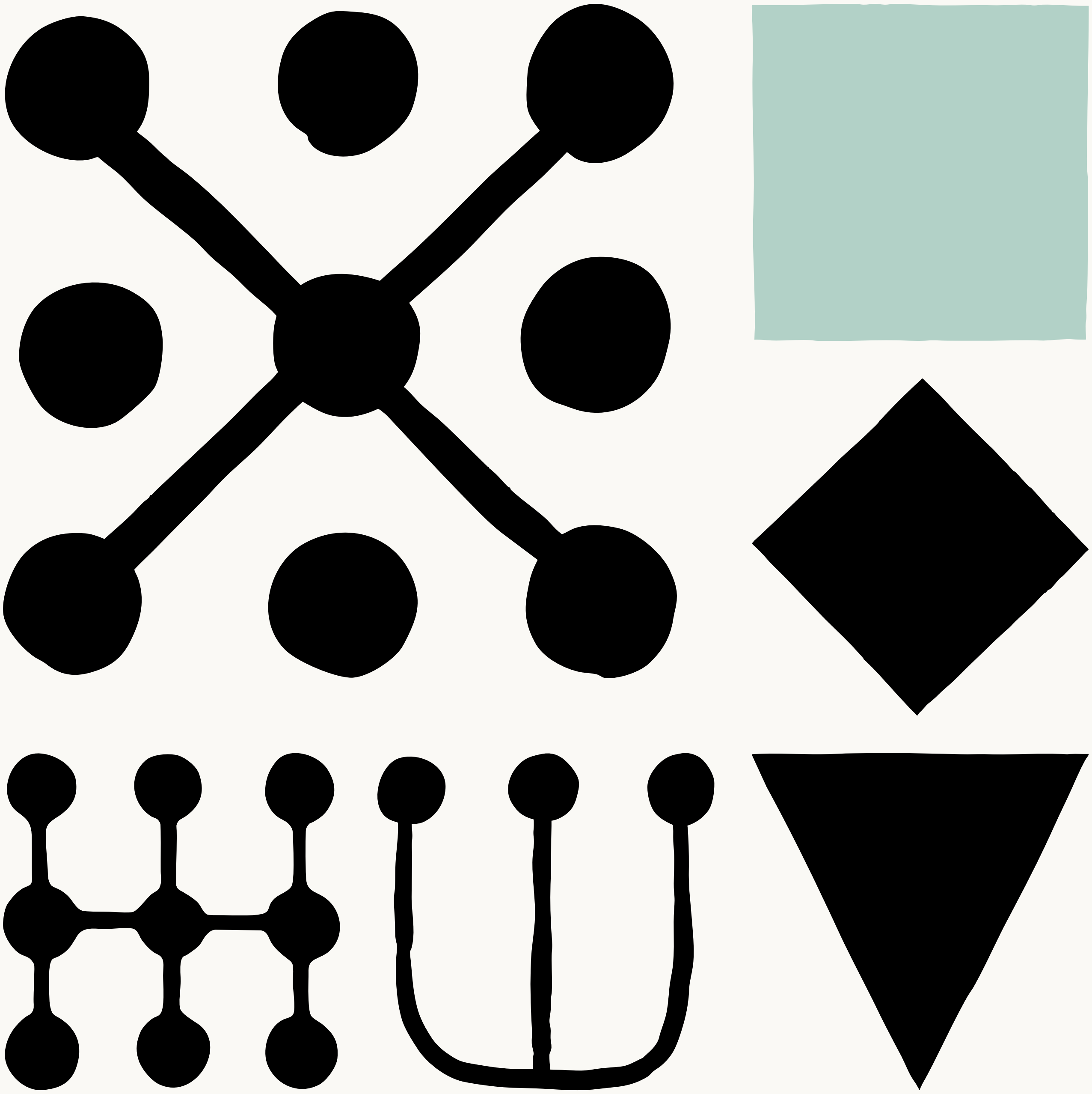
Engineering at Anthropic



How we built our multi-agent research system

Our Research feature uses multiple Claude agents to explore complex topics more effectively. We share the engineering challenges and the lessons we learned from building this system.

Published Jun 13, 2025

Claude now has Research capabilities that allow it to search across the web, Google Workspace, and any integrations to accomplish complex tasks.

The journey of this multi-agent system from prototype to production taught us critical lessons about system architecture, tool design, and prompt engineering. A multi-agent system consists of multiple agents (LLMs autonomously using tools in a loop) working together. Our Research feature involves an agent that plans a research process based on user queries, and then uses tools to create parallel agents that search for information simultaneously. Systems with multiple agents introduce new challenges in agent coordination, evaluation, and reliability.

This post breaks down the principles that worked for us—we hope you'll find them useful to apply when building your own multi-agent systems.

Benefits of a multi-agent system

Research work involves open-ended problems where it's very difficult to predict the required steps in advance. You can't hardcode a fixed path for exploring complex topics, as the process is inherently dynamic and path-dependent. When people conduct research, they tend to continuously update their approach based on discoveries, following leads that emerge during investigation.

This unpredictability makes AI agents particularly well-suited for research tasks. Research demands the flexibility to pivot or explore tangential connections as the investigation unfolds. The model must operate autonomously for many turns, making decisions about which directions to pursue based on intermediate findings. A linear, one-shot pipeline cannot handle these tasks.

The essence of search is compression: distilling insights from a vast corpus. Subagents facilitate compression by operating in parallel with their own context windows, exploring different aspects of the question simultaneously before condensing the most important tokens for the lead research agent. Each subagent also provides separation of concerns—distinct tools, prompts, and exploration trajectories—which reduces path dependency and enables thorough, independent investigations.

Once intelligence reaches a threshold, multi-agent systems become a vital way to scale performance. For instance, although individual humans have become more intelligent in the last 100,000 years, human societies have become *exponentially* more capable in the information age because of our *collective* intelligence and ability to coordinate. Even generally-intelligent agents face limits when operating as individuals; groups of agents can accomplish far more.

Our internal evaluations show that multi-agent research systems excel especially for breadth-first queries that involve pursuing multiple independent directions simultaneously. We found that a multi-agent system with Claude Opus 4 as the lead agent and Claude Sonnet 4 subagents outperformed single-agent Claude Opus 4 by 90.2% on our internal research eval. For example, when asked to identify all the board members of the companies in the Information Technology S&P 500, the multi-agent system found the correct answers by decomposing this into tasks for subagents, while the single agent system failed to find the answer with slow, sequential searches.

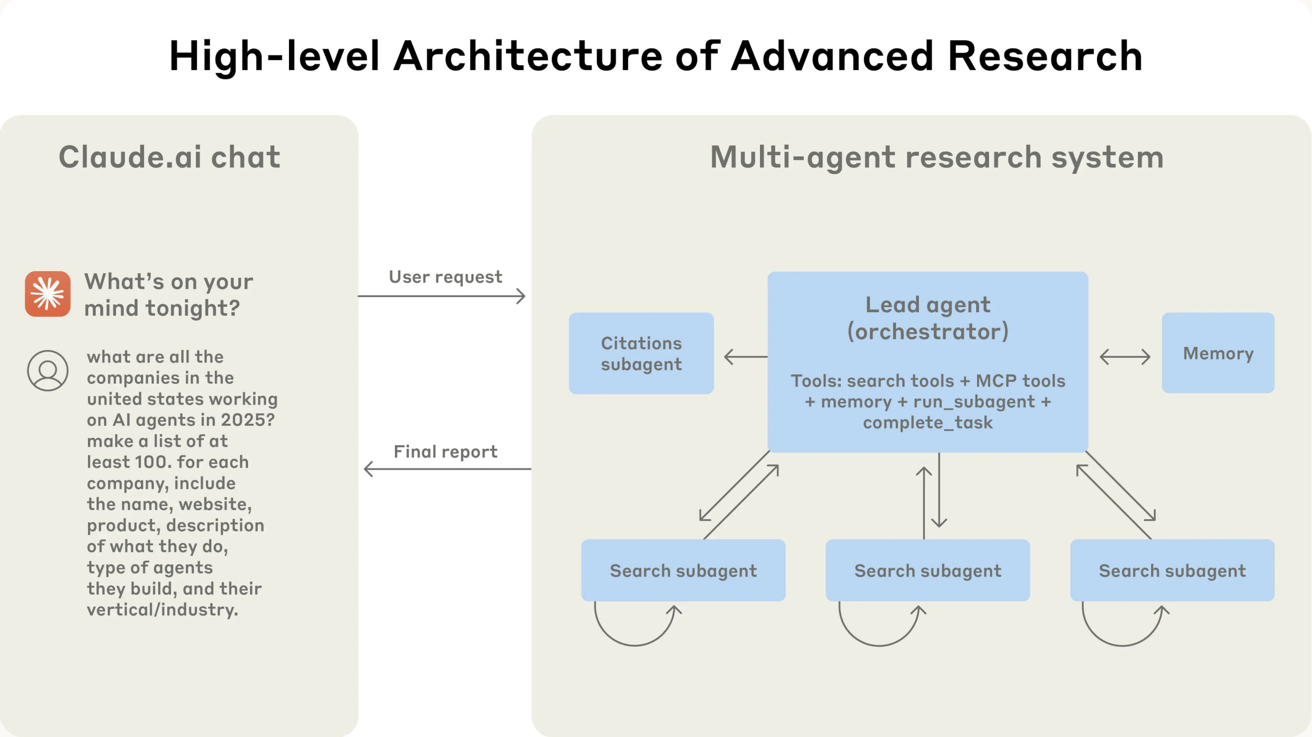
Multi-agent systems work mainly because they help spend enough tokens to solve the problem. In our analysis, three factors explained 95% of the performance variance in the BrowseComp evaluation (which tests the ability of browsing agents to locate hard-to-find information). We found that token usage by itself explains 80% of the variance, with the number of tool calls and the model choice as the two other explanatory factors. This finding validates our architecture that distributes work across agents with separate context windows to add more capacity for parallel reasoning. The latest Claude models act as large efficiency multipliers on token use, as upgrading to Claude Sonnet 4 is a larger performance gain than doubling the token budget on Claude Sonnet 3.7. Multi-agent architectures effectively scale token usage for tasks that exceed the limits of single agents.

There is a downside: in practice, these architectures burn through tokens fast.

In our data, agents typically use about 4× more tokens than chat interactions, and multi-agent systems use about 15× more tokens than chats. For economic viability, multi-agent systems require tasks where the value of the task is high enough to pay for the increased performance. Further, some domains that require all agents to share the same context or involve many dependencies between agents are not a good fit for multi-agent systems today. For instance, most coding tasks involve fewer truly parallelizable tasks than research, and LLM agents are not yet great at coordinating and delegating to other agents in real time. We've found that multi-agent systems excel at valuable tasks that involve heavy parallelization, information that exceeds single context windows, and interfacing with numerous complex tools.

Architecture overview for Research

Our Research system uses a multi-agent architecture with an orchestrator-worker pattern, where a lead agent coordinates the process while delegating to specialized subagents that operate in parallel.


High-level Architecture of Advanced Research
Claude.ai chat
What's on your mind tonight?
User request
Final report
Multi-agent research system
Lead agent (orchestrator)
Tools: search tools + MCP tools + memory + run_subagent + complete_task
Citations subagent
Memory
Search subagent
Search subagent
Search subagent

what are all the companies in the united states working on AI agents in 2025? make a list of at least 100. for each company, include the name, website, product, description of what they do, type of agents they build, and their vertical/industry.

The multi-agent architecture in action: user queries flow through a lead agent that creates specialized subagents to search for different aspects in parallel.

When a user submits a query, the lead agent analyzes it, develops a strategy, and spawns subagents to explore different aspects simultaneously. As shown in the diagram above, the subagents act as intelligent filters by iteratively using search tools to gather information, in this case on AI agent companies in 2025, and then returning a list of companies to the lead agent so it can compile a final answer.

Traditional approaches using Retrieval Augmented Generation (RAG) use static retrieval. That is, they fetch some set of chunks that are most similar to an input query and use these chunks to generate a response. In contrast, our architecture uses a multi-step search that dynamically finds relevant information, adapts to new findings, and analyzes results to formulate high-quality answers.

Process diagram showing the complete workflow of our multi-agent Research system. When a user submits a query, the system creates a LeadResearcher agent that enters an iterative research process. The

LeadResearcher begins by thinking through the approach and saving its plan to Memory to persist the context, since if the context window exceeds 200,000 tokens it will be truncated and it is important to retain the plan. It then creates specialized Subagents (two are shown here, but it can be any number) with specific research tasks. Each Subagent independently performs web searches, evaluates tool results using interleaved thinking, and returns findings to the LeadResearcher. The LeadResearcher synthesizes these results and decides whether more research is needed—if so, it can create additional subagents or refine its strategy. Once sufficient information is gathered, the system exits the research loop and passes all findings to a CitationAgent, which processes the documents and research report to identify specific locations for citations. This ensures all claims are properly attributed to their sources. The final research results, complete with citations, are then returned to the user.

Prompt engineering and evaluations for research agents

Multi-agent systems have key differences from single-agent systems, including a rapid growth in coordination complexity. Early agents made errors like spawning 50 subagents for simple queries, scouring the web endlessly for nonexistent sources, and distracting each other with excessive updates. Since each agent is steered by a prompt, prompt engineering was our primary lever for improving these behaviors. Below are some principles we learned for prompting agents:

1. **Think like your agents.** To iterate on prompts, you must understand their effects. To help us do this, we built simulations using our Console with the exact prompts and tools from our system, then watched agents work step-by-step. This immediately revealed failure modes: agents continuing when they already had sufficient results, using overly verbose search queries, or selecting incorrect tools. Effective prompting relies on developing an accurate mental model of the agent, which can make the most impactful changes obvious.
2. **Teach the orchestrator how to delegate.** In our system, the lead agent decomposes queries into subtasks and describes them to subagents. Each subagent needs an objective, an output format, guidance on the tools and sources to use, and clear task boundaries. Without detailed task descriptions, agents duplicate work, leave gaps, or fail to find necessary information. We started by allowing the lead agent to give simple, short instructions like 'research the semiconductor shortage,' but found these instructions often were vague enough that subagents misinterpreted the task or performed the exact same searches as other agents. For instance, one subagent explored the 2021 automotive chip crisis while 2 others duplicated work investigating current 2025 supply chains, without an effective division of labor.
3. **Scale effort to query complexity.** Agents struggle to judge appropriate effort for different tasks, so we embedded scaling rules in the prompts.

Simple fact-finding requires just 1 agent with 3-10 tool calls, direct comparisons might need 2-4 subagents with 10-15 calls each, and complex research might use more than 10 subagents with clearly divided responsibilities. These explicit guidelines help the lead agent allocate resources efficiently and prevent overinvestment in simple queries, which was a common failure mode in our early versions.

4. **Tool design and selection are critical.** Agent-tool interfaces are as critical as human-computer interfaces. Using the right tool is efficient—often, it's strictly necessary. For instance, an agent searching the web for context that only exists in Slack is doomed from the start. With MCP servers that give the model access to external tools, this problem compounds, as agents encounter unseen tools with descriptions of wildly varying quality. We gave our agents explicit heuristics: for example, examine all available tools first, match tool usage to user intent, search the web for broad external exploration, or prefer specialized tools over generic ones. Bad tool descriptions can send agents down completely wrong paths, so each tool needs a distinct purpose and a clear description.
5. **Let agents improve themselves.** We found that the Claude 4 models can be excellent prompt engineers. When given a prompt and a failure mode, they are able to diagnose why the agent is failing and suggest improvements. We even created a tool-testing agent—when given a flawed MCP tool, it attempts to use the tool and then rewrites the tool description to avoid failures. By testing the tool dozens of times, this agent found key nuances and bugs. This process for improving tool ergonomics resulted in a 40% decrease in task completion time for future agents using the new description, because they were able to avoid most mistakes.
6. **Start wide, then narrow down.** Search strategy should mirror expert human research: explore the landscape before drilling into specifics. Agents often default to overly long, specific queries that return few results. We counteracted this tendency by prompting agents to start with short, broad queries, evaluate what's available, then progressively narrow focus.
7. **Guide the thinking process.** Extended thinking mode, which leads Claude to output additional tokens in a visible thinking process, can serve as a controllable scratchpad. The lead agent uses thinking to plan its approach, assessing which tools fit the task, determining query complexity and subagent count, and defining each subagent's role. Our testing showed that extended thinking improved instruction-following, reasoning, and efficiency. Subagents also plan, then use interleaved thinking after tool results to evaluate quality, identify gaps, and refine their next query. This

makes subagents more effective in adapting to any task.

8. **Parallel tool calling transforms speed and performance.** Complex research tasks naturally involve exploring many sources. Our early agents executed sequential searches, which was painfully slow. For speed, we introduced two kinds of parallelization: (1) the lead agent spins up 3-5 subagents in parallel rather than serially; (2) the subagents use 3+ tools in parallel. These changes cut research time by up to 90% for complex queries, allowing Research to do more work in minutes instead of hours while covering more information than other systems.

Our prompting strategy focuses on instilling good heuristics rather than rigid rules. We studied how skilled humans approach research tasks and encoded these strategies in our prompts—strategies like decomposing difficult questions into smaller tasks, carefully evaluating the quality of sources, adjusting search approaches based on new information, and recognizing when to focus on depth (investigating one topic in detail) vs. breadth (exploring many topics in parallel). We also proactively mitigated unintended side effects by setting explicit guardrails to prevent the agents from spiraling out of control. Finally, we focused on a fast iteration loop with observability and test cases.

Effective evaluation of agents

Good evaluations are essential for building reliable AI applications, and agents are no different. However, evaluating multi-agent systems presents unique challenges. Traditional evaluations often assume that the AI follows the same steps each time: given input X, the system should follow path Y to produce output Z. But multi-agent systems don't work this way. Even with identical starting points, agents might take completely different valid paths to reach their goal. One agent might search three sources while another searches ten, or they might use different tools to find the same answer. Because we don’t always know what the right steps are, we usually can't just check if agents followed the “correct” steps we prescribed in advance. Instead, we need flexible evaluation methods that judge whether agents achieved the right outcomes while also following a reasonable process.

Start evaluating immediately with small samples. In early agent development, changes tend to have dramatic impacts because there is abundant low-hanging fruit. A prompt tweak might boost success rates from 30% to 80%. With effect sizes this large, you can spot changes with just a few test cases. We started with a set of about 20 queries representing real usage patterns. Testing these queries often allowed us to clearly see the impact of changes. We often hear that AI developer teams delay creating evals because they believe that only large evals with hundreds of test cases are useful. However, it's best to start with small-scale testing right away with a few examples, rather than delaying until you can build more thorough evals.

LLM-as-judge evaluation scales when done well. Research outputs are difficult to evaluate programmatically, since they are free-form text and rarely have a single correct answer. LLMs are a natural fit for grading outputs. We used an LLM judge that evaluated each output against criteria in a rubric: factual accuracy (do claims match sources?), citation accuracy (do the cited sources match the claims?), completeness (are all requested aspects covered?), source quality (did it use primary sources over lower-quality secondary sources?), and tool efficiency (did it use the right tools a reasonable number of times?). We experimented with multiple judges to evaluate each component, but found that a single LLM call with a single prompt outputting scores from 0.0-1.0 and a pass-fail grade was the most consistent and aligned with human judgements. This method was especially effective when the eval test cases *did* have a clear answer, and we could use the LLM judge to simply check if the answer was correct (i.e. did it accurately list the pharma companies with the top 3 largest R&D budgets?). Using an LLM as a judge allowed us to scalably evaluate hundreds of outputs.

Human evaluation catches what automation misses. People testing agents find edge cases that evals miss. These include hallucinated answers on unusual queries, system failures, or subtle source selection biases. In our case, human testers noticed that our early agents consistently chose SEO-optimized content farms over authoritative but less highly-ranked sources like academic PDFs or personal blogs. Adding source quality heuristics to our prompts helped resolve this issue. Even in a world of automated evaluations, manual testing remains essential.

Multi-agent systems have emergent behaviors, which arise without specific programming. For instance, small changes to the lead agent can unpredictably change how subagents behave. Success requires understanding interaction patterns, not just individual agent behavior. Therefore, the best prompts for these agents are not just strict instructions, but frameworks for collaboration that define the division of labor, problem-solving approaches, and effort budgets. Getting this right relies on careful prompting and tool design, solid heuristics, observability, and tight feedback loops. See the open-source prompts in our Cookbook for example prompts from our system.

Production reliability and engineering challenges

In traditional software, a bug might break a feature, degrade performance, or cause outages. In agentic systems, minor changes cascade into large behavioral changes, which makes it remarkably difficult to write code for complex agents that must maintain state in a long-running process.

Agents are stateful and errors compound. Agents can run for long periods of time, maintaining state across many tool calls. This means we need to durably execute code and handle errors along the way. Without effective mitigations, minor system failures can be catastrophic for agents. When errors occur, we can't just restart from the beginning: restarts are expensive and frustrating for users. Instead, we built systems that can resume from where the agent was when the errors occurred. We also use the model’s intelligence to handle issues gracefully: for instance, letting the agent know when a tool is failing and letting it adapt works surprisingly well. We combine the adaptability of AI agents built on Claude with deterministic safeguards like retry logic and regular checkpoints.

Debugging benefits from new approaches. Agents make dynamic decisions and are non-deterministic between runs, even with identical prompts. This makes debugging harder. For instance, users would report agents “not finding obvious information,” but we couldn't see why. Were the agents using bad search queries? Choosing poor sources? Hitting tool failures? Adding full production tracing let us diagnose why agents failed and fix issues systematically. Beyond standard observability, we monitor agent decision patterns and interaction structures—all without monitoring the contents of individual conversations, to maintain user privacy. This high-level observability helped us diagnose root causes, discover unexpected behaviors, and fix common failures.

Deployment needs careful coordination. Agent systems are highly stateful webs of prompts, tools, and execution logic that run almost continuously. This means that whenever we deploy updates, agents might be anywhere in their process. We therefore need to prevent our well-meaning code changes from breaking existing agents. We can't update every agent to the new version at the same time. Instead, we use rainbow deployments to avoid disrupting running agents, by gradually shifting traffic from old to new versions while keeping both running simultaneously.

Synchronous execution creates bottlenecks. Currently, our lead agents execute subagents synchronously, waiting for each set of subagents to complete before proceeding. This simplifies coordination, but creates bottlenecks in the information flow between agents. For instance, the lead agent can't steer subagents, subagents can't coordinate, and the entire system can be blocked while waiting for a single subagent to finish searching. Asynchronous execution would enable additional parallelism: agents working concurrently and creating new subagents when needed. But this asynchronicity adds challenges in result coordination, state consistency, and error propagation across the subagents. As models can handle longer and more complex research tasks, we expect the performance gains will justify the complexity.

Conclusion

When building AI agents, the last mile often becomes most of the journey. Codebases that work on developer machines require significant engineering to become reliable production systems. The compound nature of errors in agentic systems means that minor issues for traditional software can derail agents entirely. One step failing can cause agents to explore entirely different trajectories, leading to unpredictable outcomes. For all the reasons described in this post, the gap between prototype and production is often wider than anticipated.

Despite these challenges, multi-agent systems have proven valuable for open-ended research tasks. Users have said that Claude helped them find business opportunities they hadn't considered, navigate complex healthcare options, resolve thorny technical bugs, and save up to days of work by uncovering research connections they wouldn't have found alone. Multi-agent research systems can operate reliably at scale with careful engineering, comprehensive testing, detail-oriented prompt and tool design, robust operational practices, and tight collaboration between research, product, and engineering teams who have a strong understanding of current agent capabilities. We're already seeing these systems transform how people solve complex problems.

A Clio embedding plot showing the most common ways people are using the Research feature today. The top use case categories are developing software systems across specialized domains (10%), develop and optimize professional and technical content (8%), develop business growth and revenue generation strategies (8%), assist with academic research and educational material development (7%), and research and verify information about people, places, or organizations (5%).

Acknowlegements

Written by Jeremy Hadfield, Barry Zhang, Kenneth Lien, Florian Scholz, Jeremy Fox, and Daniel Ford. This work reflects the collective efforts of several teams across Anthropic who made the Research feature possible. Special thanks go to the Anthropic apps engineering team, whose dedication brought this complex multi-agent system to production. We're also grateful to our early users for their excellent feedback.

Appendix

Below are some additional miscellaneous tips for multi-agent systems.

End-state evaluation of agents that mutate state over many turns. Evaluating agents that modify persistent state across multi-turn conversations presents unique challenges. Unlike read-only research tasks, each action can change the environment for subsequent steps, creating dependencies that traditional evaluation methods struggle to handle. We found success focusing on end-state evaluation rather than turn-by-turn analysis. Instead of judging whether the agent followed a specific process, evaluate whether it achieved the correct final state. This approach acknowledges that agents may find alternative paths to the same goal while still ensuring they deliver the intended outcome. For complex workflows, break evaluation into discrete checkpoints where specific state changes should have occurred, rather than attempting to validate every intermediate step.

Long-horizon conversation management. Production agents often engage in conversations spanning hundreds of turns, requiring careful context management strategies. As conversations extend, standard context windows become insufficient, necessitating intelligent compression and memory mechanisms. We implemented patterns where agents summarize completed work phases and store essential information in external memory before proceeding to new tasks. When context limits approach, agents can spawn fresh subagents with clean contexts while maintaining continuity through careful handoffs. Further, they can retrieve stored context like the research plan from their memory rather than losing previous work when reaching the context limit. This distributed approach prevents context overflow while preserving conversation coherence across extended interactions.

Subagent output to a filesystem to minimize the 'game of telephone.' Direct subagent outputs can bypass the main coordinator for certain types of results, improving both fidelity and performance. Rather than requiring subagents to communicate everything through the lead agent, implement artifact systems where specialized agents can create outputs that persist independently. Subagents call tools to store their work in external systems, then pass lightweight references back to the coordinator. This prevents information loss during multi-stage processing and reduces token overhead from copying large outputs through conversation history. The pattern works particularly well for structured outputs like code, reports, or data visualizations where the subagent's specialized prompt produces better results than filtering through a general coordinator.

Product

Claude overview

Claude Code

Claude team plan

Claude enterprise plan

Claude education plan

Download Claude apps

Claude.ai pricing plans

Claude.ai login

API Platform

API overview

Developer docs

Claude in Amazon Bedrock

Claude on Google Cloud's Vertex AI

Pricing

Console login

Research

Research overview

Economic Index

Claude models

Claude Opus 4

Claude Sonnet 4

Claude Haiku 3.5

Commitments

Transparency

Responsible scaling policy

Security and compliance

Solutions

AI agents

Coding

Customer support

Learn

Explore

Anthropic Academy

Customer stories

Engineering at Anthropic

About us

Become a partner

Careers

Events

News

Startups program

Help and security

Status

Availability

Support center

Terms and policies

Privacy choices

Privacy policy

Responsible disclosure policy

Terms of service - consumer

Terms of service - commercial

Usage policy

© 2025 Anthropic PBC