Building the Document Agent

A Word add-in powered by Claude Code SDK

A small side project over 3 weeks
Built independently using only publicly available tools, APIs, and documentation
Views expressed here are my own
Demo Video

Tao Ge (sggetao@gmail.com)

April 4, 2026

Contents

Chapter 1. Overview of the Agent System Built for the Document Agent Project

This chapter describes the agent system I built for the Document Agent project. The goal of the system was to support long-horizon, open-ended document tasks while keeping context growth, execution cost, and system complexity under control. Rather than relying on a single monolithic agent, the system is organized around a main agent, several specialized sub-agents, a set of custom MCP tools, the SDK's built-in tools, and a collection of skills and helper scripts.

At a high level, the system is designed around one central idea: the main agent should remain the top-level coordinator, while heavy work, specialized work, and large artifacts should be pushed outward into sub-agents, tools, files, and reusable scripts.

1.1 High-Level Agent Architecture

The top-level controller is the main agent, which runs on Claude Opus 4.6 with a 1M-token context window. This agent owns the overall task, tracks progress, decides when to delegate, and determines how intermediate artifacts should flow through the system.

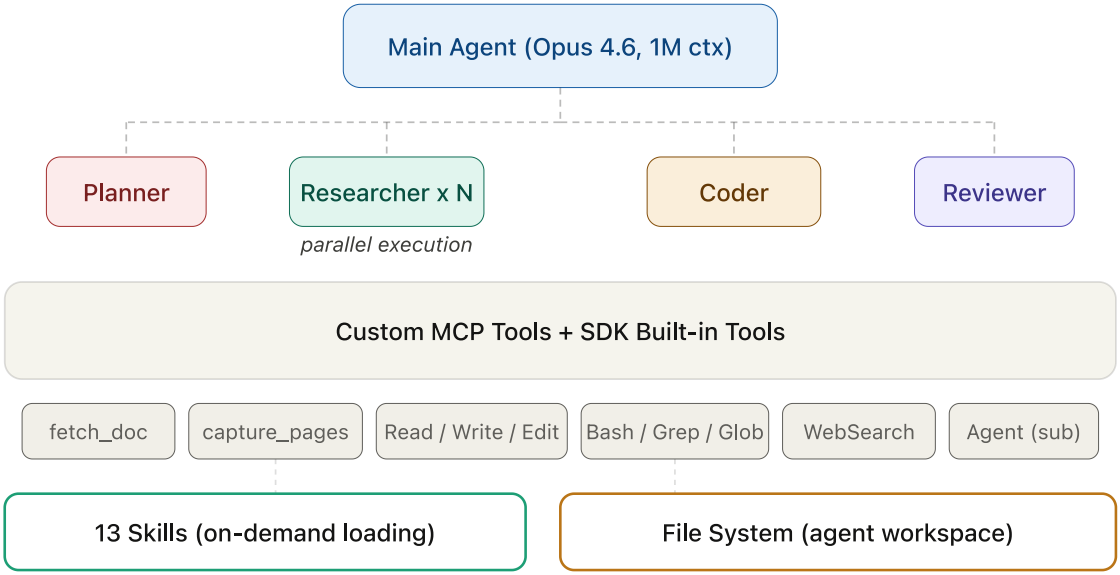

Figure 1: High-level Multi-agent Architecture

This gives the system a clear division of labor. The main agent remains responsible for top-level coordination, while planning, reviewing, research, and large code generation are delegated to specialized roles when needed.

1.2 Parallelism and Execution Model

The system supports limited parallelism through the Researcher agents. Multiple Researcher agents can be launched in parallel within the same response by issuing multiple Agent tool calls. This is useful when the main agent needs to gather information from multiple sources or explore several research threads at once.

At the same time, the current system does not allow `run_in_background`, because doing so interferes with later queries by effectively hijacking subsequent interactions. As a result, the main agent waits

for all parallel sub-agent calls to return before continuing. In other words, the system supports parallel sub-agent execution within a turn, but not fully asynchronous long-running background workflows.

This still provides meaningful speedup for research-heavy tasks without introducing the control and coordination problems that background execution would create.

1.3 Custom MCP Tools

In addition to the SDK's built-in tools, the system relies on a custom MCP server, which provides several Word-specific operations[1]. These tools capture deterministic document operations that would otherwise be wasteful or unreliable to regenerate through pure prompting.

These tools are important because they move deterministic and Word-specific execution out of the agent's free-form reasoning loop and into reusable interfaces.

1.4 SDK Built-in Tools

The system also makes extensive use of the SDK's built-in tools. These provide the general-purpose execution surface that the rest of the architecture builds on top of.

These tools form the generic runtime layer. The custom Word tools then extend that layer for domain-specific document operations.

1.5 Skills as a Domain Knowledge Layer

On top of the runtime and tooling layers, the system uses 13 skills, loaded on demand. These skills serve as reusable domain knowledge modules and keep large specialized guidance out of the main context unless it is actually needed.

They fall into three broad groups:

- **Document-editing skills** — guidance for structural and content editing operations.
- **Visual skills** — guidance for generating and refining visual elements.
- **Grounding skills** — reading external files as data sources for task context.

1.6 Helper Scripts for Deterministic Operations

The system also includes several helper scripts stored inside skill directories. These scripts handle repeated deterministic operations that the model should not have to regenerate from scratch each time. One example provides a rendering theme for SVG output, including Inter font settings and color definitions.

These scripts reduce both token cost and error rate. Instead of asking the model to reproduce known deterministic procedures repeatedly, the system lets the model call or adapt reusable building blocks.

[1] by public office.js APIs: https://learn.microsoft.com/en-us/javascript/api/requirement-sets/word/word-preview-apis?view=word-js-preview

1.7 End-to-End System Flow

Putting everything together, the system operates roughly as follows.

The main agent receives the user request and first decides whether the task can be handled directly or whether part of it should be delegated. If planning is needed, the Planner helps structure the document and, when appropriate, formulate a research plan. If external information is required, one or more Researcher agents gather it, potentially in parallel. If substantial Office.js code needs to be written, the Coder generates it and stores it in files. When the document has been created or modified, the Reviewer checks both content and formatting quality.

Throughout this process, the filesystem acts as the system's external workspace. Large intermediate outputs, generated code, screenshots, extracted document state, and research artifacts are written to disk. The main agent then reads these artifacts selectively instead of carrying everything directly in context.

In this way, the architecture keeps top-level orchestration centralized while distributing heavy work across specialized components, tools, and files.

Chapter 2. Practical Learnings from Building the Document Agent Project

This chapter summarizes the main lessons I learned while building agents for the Document Agent project. These lessons came from practical development rather than theory: repeated issues around context growth, large-file handling, prompt organization, tool design, and agent coordination gradually made certain design principles much clearer. Looking back, the most important learnings can be organized into eight points.

2.1 Start by Defining Clear System Boundaries

One of the most important lessons is that building a good agent system is not just about making the model smarter or adding more agents. It is equally about defining clear boundaries in the system: what should stay in context, what should go to files, what should be handled by tools, and what should be left to the agent.

Without these boundaries, complexity does not disappear. It gradually accumulates inside prompts, context, and ad hoc coordination logic. At first, the system may still appear to work, but over time it becomes harder to debug, harder to extend, and harder to reason about. In practice, many agent systems become fragile not because the model is weak, but because the structure around the model is poorly defined.

A well-designed agent system therefore needs explicit boundaries between reasoning, storage, and execution. Once those boundaries are clear, the whole system becomes much easier to evolve.

2.2 Do Not Over-Constrain the Model Where It Is Already Strong

Another important lesson is that we should not add prompts, skills, or extra rules by default. If the model already performs well in a certain area, trying to control it more aggressively can actually make it worse.

These additions are not free. Besides increasing latency and cost, they also add more material into context, which can distract the model and interfere with its natural decision-making process. In some cases, excessive guidance makes the model more mechanical, less adaptive, and less effective than it would have been with a lighter-touch setup.

The right principle is therefore selective constraint: only add extra prompts, skills, or rules in areas where the model's default behavior is consistently weak, incorrect, or off-target. If the model is already strong, it is often better to let it operate with less interference.

2.3 Move from Single-Agent to Multi-Agent Only When Context Demands It

The move from a single-agent design to a multi-agent design should be driven primarily by context management, not by architectural preference.

A single agent can work well for shorter tasks, but as the task becomes longer and more complex, context grows continuously. That growth increases cost and often hurts quality as well. The model has to carry too much history, too many intermediate states, and too many partially relevant details. Over time, this reduces clarity.

The practical solution is to keep the main agent relatively lightweight and delegate heavier work to sub-agents that operate in separate contexts. This allows the system to isolate large or specialized tasks without permanently bloating the main agent's working memory.

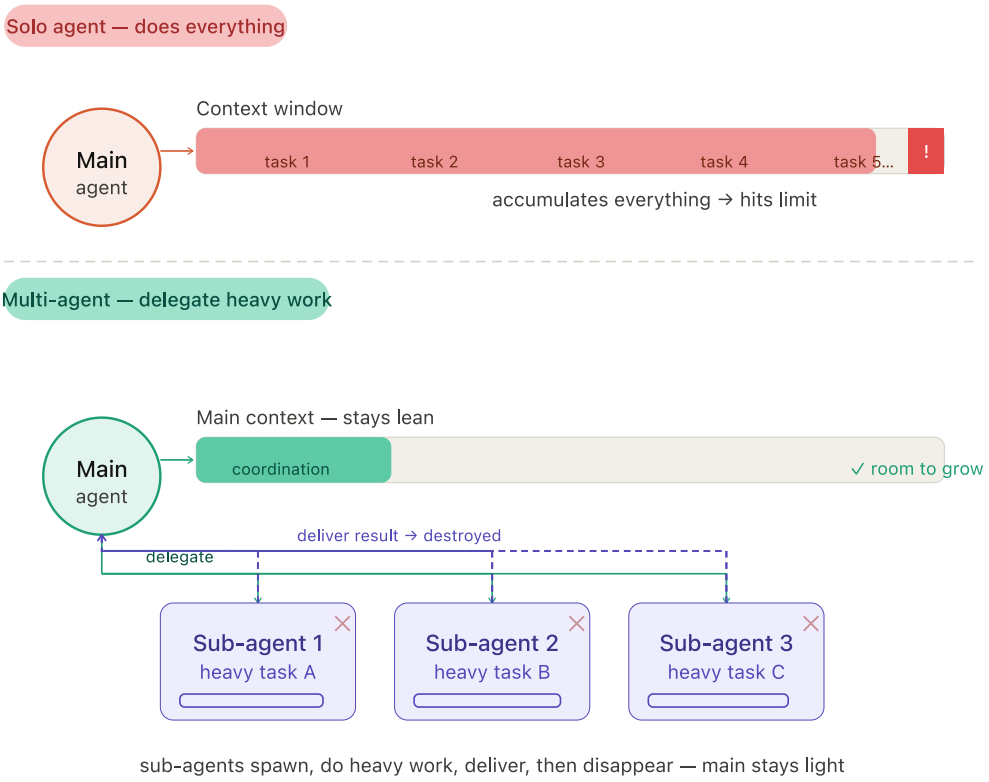

Figure 2: Context Management of a Single **VS** Multi Agents

At the same time, delegation has a cost. Whenever a task is passed to a sub-agent, the system has to communicate enough background information for that sub-agent to work correctly. This means multi-agent design should not be used blindly. In practice, a rough threshold of around 2K/4K/8K tokens proved useful as a rule of thumb: below that, the main agent could often handle the task directly; above that, delegation started to become worthwhile.

The key point is that multi-agent design is fundamentally a context-management strategy.

2.4 Treat the Filesystem as the Agent's Core Workspace

A second major design principle is that the filesystem should be treated as the agent's external memory and primary workspace. **Heavy data should live in files**, while context should contain only lightweight references.

This principle applies across multiple scenarios. When a sub-agent produces a large output, it is usually better to write that output to a file rather than return the full content inside a message. The main agent can then read the file only when needed. If the file is still too large or requires specialized processing, the path can be passed to another sub-agent instead of moving the entire content through context.

The same pattern is useful for code execution. Once code becomes moderately long, it is better to write it to a file and execute it from there. If execution fails, the system can edit the file incrementally rather than regenerate the entire code block from scratch. This makes iteration much cheaper and more reliable.

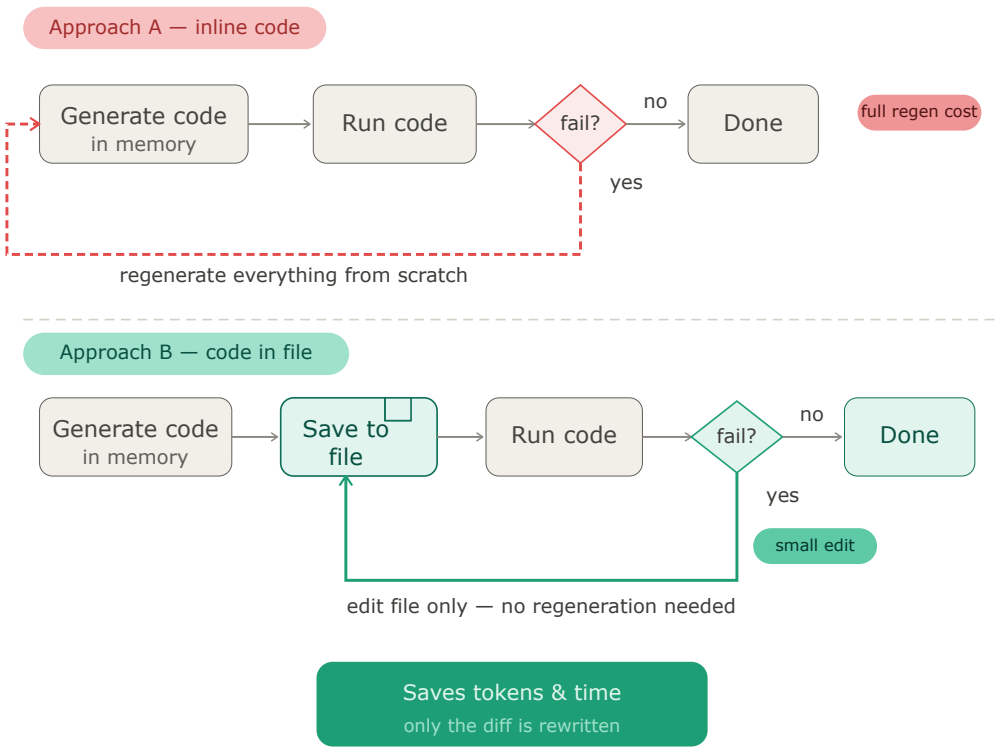

Figure 3: Inline Code **VS** Code in File

Large tool outputs should be governed in the same way. Tool results are themselves part of context, so if they become too large, they should be intercepted, written to disk, and replaced with a short summary plus a file path. Screenshots should also be stored as files rather than embedded inline as bulky payloads.

The broader insight is simple: code, screenshots, intermediate artifacts, research outputs, and other heavy content should live in the filesystem. Context should mainly function as a reference layer that points to those artifacts.

2.5 Read Large Documents Strategically Rather Than Dumping Them into Context

Large documents should not be handled by loading everything into context and hoping the model can sort it out. That approach is expensive, noisy, and often unnecessary.

A much better approach is strategic, on-demand reading. The first step should be to probe the file and understand its size and structure. If the document is large, the system should first inspect an index, table of contents, or any available structural summary. That structure can then guide the model toward the relevant sections, which can be fetched selectively rather than all at once.

If no index is available, a lightweight scan of the beginning of the document can still help establish a rough sense of its shape before going deeper. Another important signal is user selection. If the user has selected a particular part of the document, that selection should be treated as a strong localization clue, allowing the agent to begin from the relevant region instead of approaching the file globally.

A key insight here is that the model often already has strong agentic reading ability. It can decide

where to look first, when to skim, and when to zoom in. System design should take advantage of that capability rather than replacing it with brute-force full-document ingestion.

2.6 Organize Prompting as a Layered System

Prompting works much better when it is treated as a layered system rather than one large undifferentiated instruction block.

A useful structure is to separate prompting into four layers:

- **System Prompt** — dynamic runtime information such as file paths, environment state, and configuration that changes from run to run.
- **CLAUDE.md** — shared static project knowledge that all agents need.
- **Sub-agent prompts** — responsibilities and constraints of individual agents.
- **Skills** — heavier or more specialized domain knowledge that should only be loaded when relevant.

This separation makes the system clearer and easier to maintain. Dynamic information stays in the dynamic layer. Stable shared knowledge stays in the shared layer. Agent-specific behavior stays local to each agent. Heavy domain knowledge stays out of the main context unless needed.

Once these layers are clearly separated, prompt design becomes much more manageable and much less error-prone.

2.7 Be Careful with CLAUDE.md Placement and Resource Scope

One subtle but important practical lesson is that `CLAUDE.md` and related resources only work correctly if their directory placement and scope configuration are handled carefully.

`CLAUDE.md` needs to be placed in the `.claude/` directory under the agent's working directory. If it is not found there, the SDK may search upward through parent directories and load a different `CLAUDE.md` than intended. In practice, this can lead to the wrong shared instructions and the wrong skills being picked up, sometimes from a parent project or even from the home directory.

This kind of error is especially confusing because the system may still appear functional, but it is operating under the wrong project context. The resulting behavior often looks like a prompt bug or a skill bug, when the real problem is simply incorrect resource discovery.

Resource scope settings need similar care. If project-level and personal-level resources are not configured correctly, skills and other expected resources may silently fail to apply. In practice, these files and settings should be treated as part of the runtime configuration, not as incidental details.

2.8 Solidify Deterministic Operations: Tool > Helper Script > Agent

A final major lesson is that the more deterministic a workflow is, the more it should be solidified rather than regenerated by the model every time.

If an operation is fully deterministic, it should usually become a tool. If the pattern is fixed but requires some parameters, it is often better expressed as a helper script. Only tasks that genuinely require judgment, planning, or open-ended reasoning should remain agent responsibilities.

This principle turned out to be very useful in practice. Deterministic document fetching logic is better represented as a tool than as an LLM-driven agent. Repeated structural operations such as

inserting page breaks, adding a table of contents, or initializing styles are often much more reliable as helper scripts than as fresh generations each time. The agent should focus on reasoning-heavy work, not on repeatedly reconstructing procedures that are already known.

This design brings two benefits at once. It reduces token usage because the model is no longer regenerating large amounts of routine code, and it improves reliability because deterministic steps are no longer exposed to unnecessary variation or hallucination.

The practical test is simple: if a process follows the same pattern every time and does not require real judgment, it should move downward toward the tool or helper-script layer.