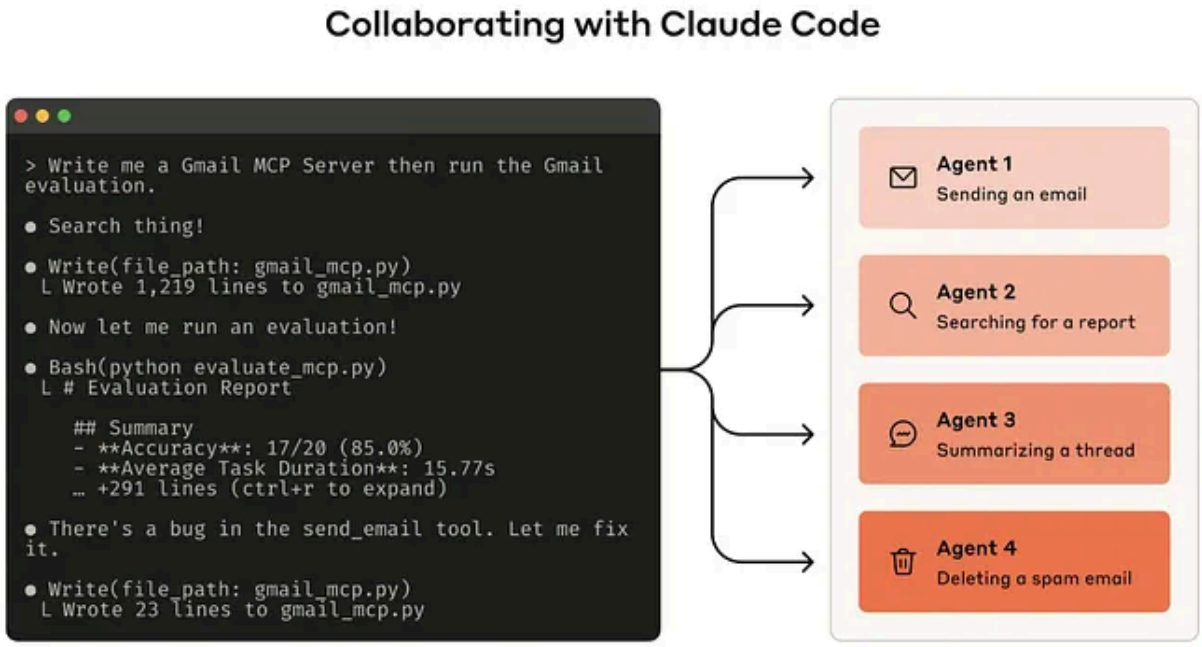
I read Anthropic's complete Tool Creation Guide so you don't have to. Here's What Actually Works!

Anthropic released the article "Writing effective tools for agents — with agents" with the goal of teaching the community clear and direct techniques on how to write and evaluate tools for your agents, using their own agents, such as Claude Code.

I read the complete article and here's a summary of their techniques plus some comments on what we've seen at **CodeGPT** when implementing these tools within VSCode.

What is a tool and how do they work with MCP?

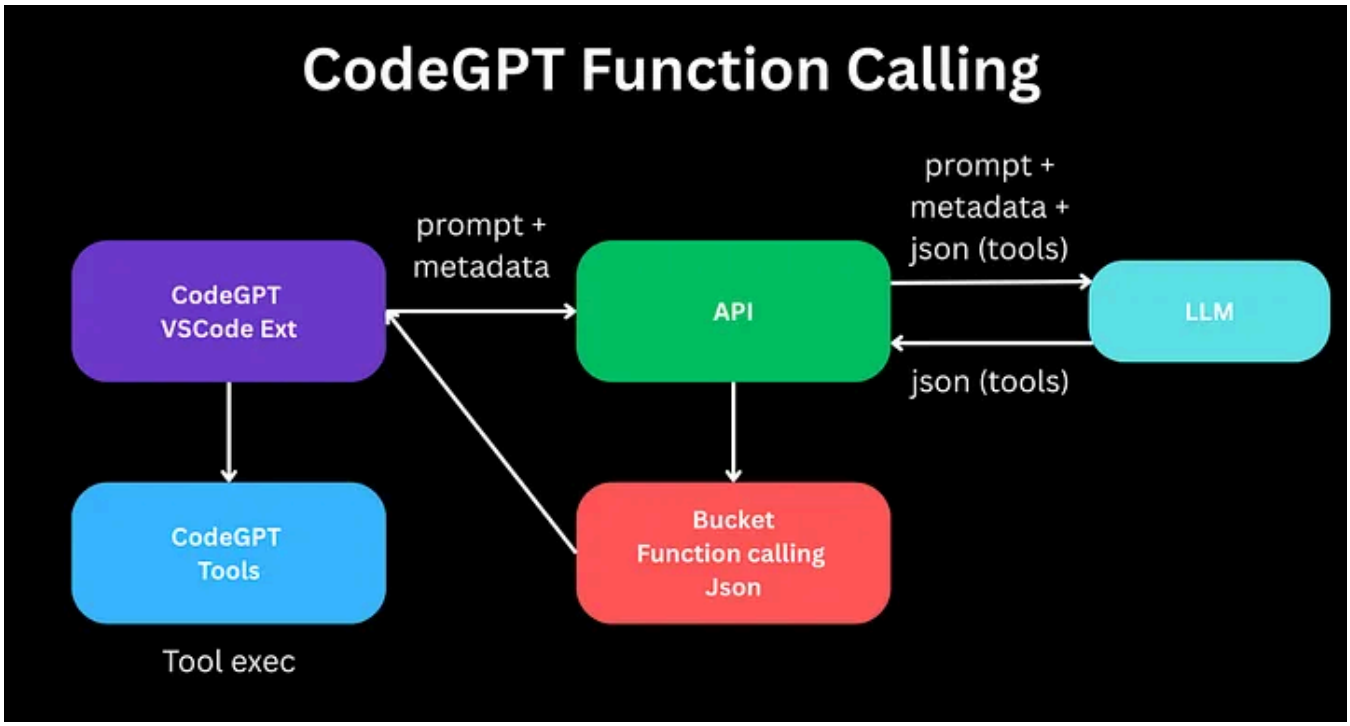


A tool is essentially **function calling for LLM agents**, but with a key difference: while a traditional function like `getWeather("NYC")` always executes the same operation, a tool must be designed for non-deterministic agents.

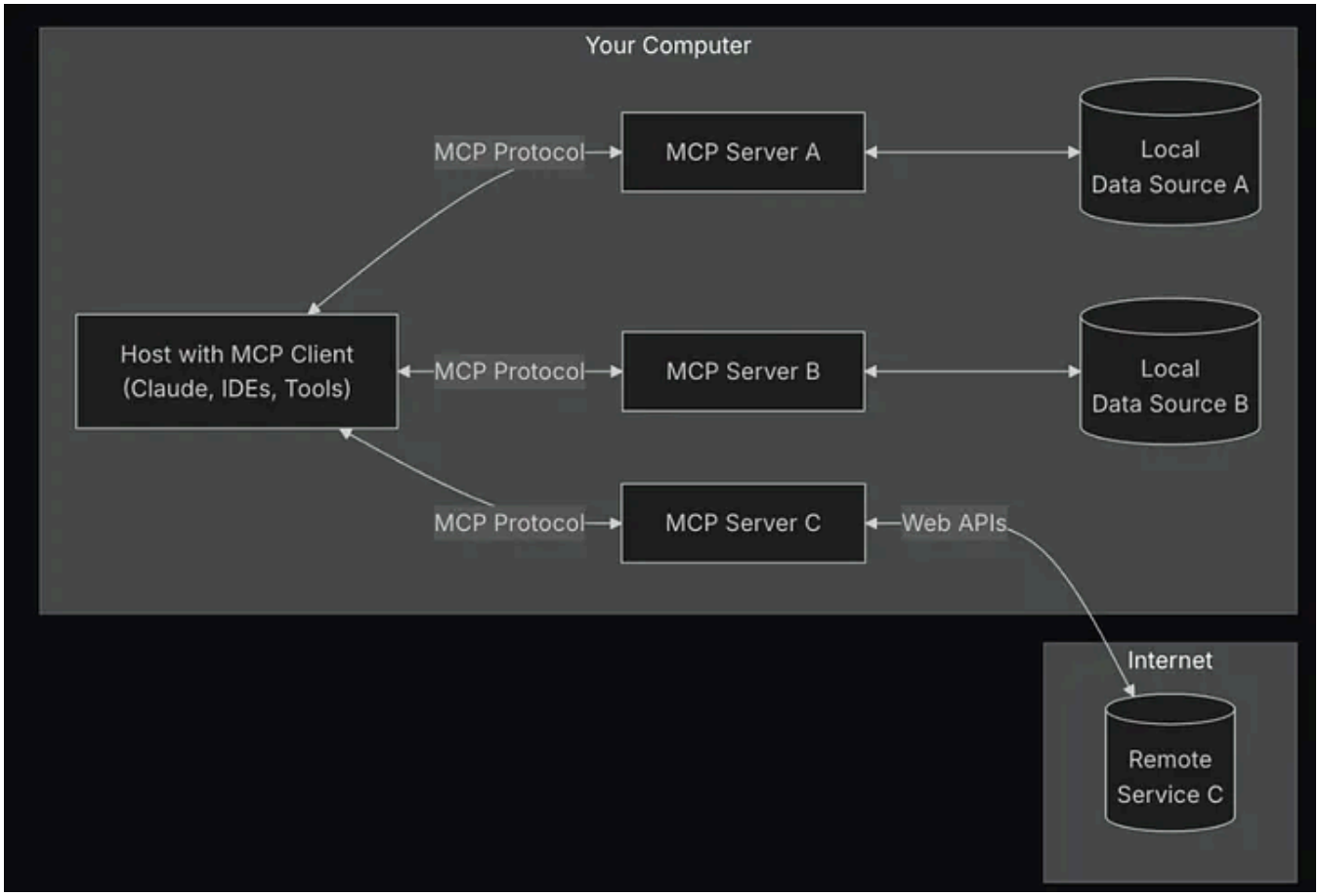
When you ask "Should I bring an umbrella today?", the agent can call the weather tool, respond from its knowledge, or ask for your location first.

This requires designing tools thinking specifically about how agents process information, not as functions for other developers.

This is an example of how we work with Function Calling in CodeGPT. We create the functions in the client, in this case VSCode, but the decision of which tool to call lives in our API, this way we can keep editing and perfecting each call depending on which model is being used.



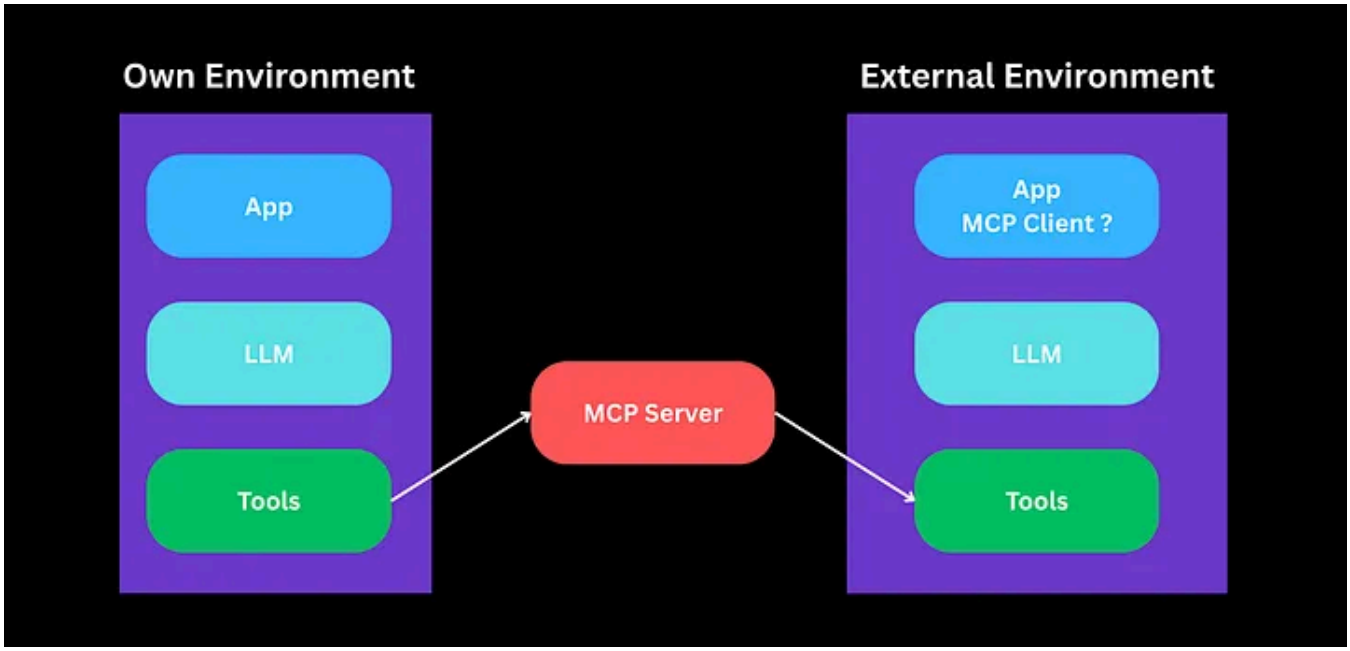
The **Model Context Protocol (MCP)**, which works similarly to how we execute tools, is the standard that allows agents to access hundreds of tools from different services in a unified way. MCP acts as an abstraction layer where the agent can seamlessly combine a Google Drive tool with a Calendar tool and a data analysis tool, automatically creating complex workflows.



Basically, MCP democratizes access to diverse tools and allows agents to solve real tasks by orchestrating multiple specialized capabilities.

But it's not all as beautiful as it seems. When connecting MCPs, you're filling your model's context window with more tokens and also adding unnecessary noise for some user requests.

I recommend using MCP only when you have an agent with specific tools according to the environment it's working in. Otherwise, it will be a disaster... the model will get confused and won't handle so many tools that have nothing to do with user requests.

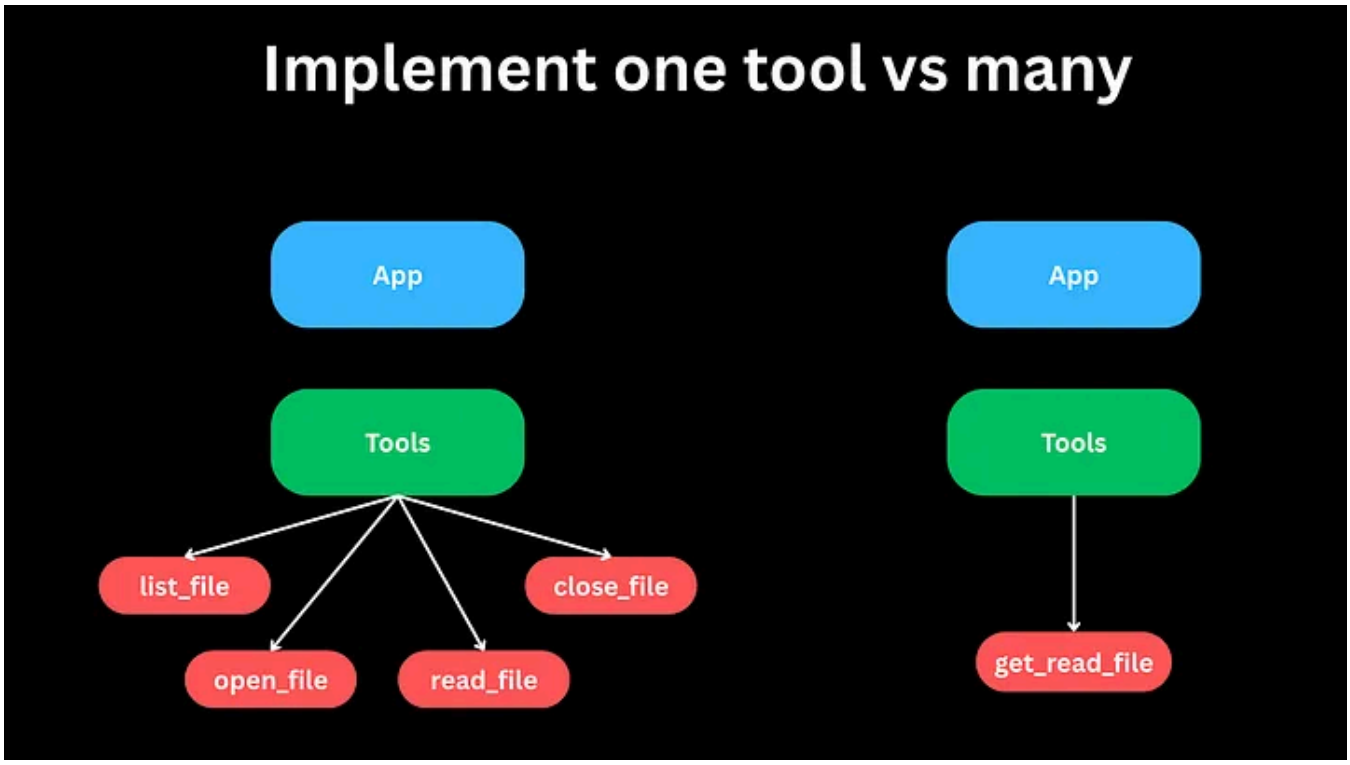


With this information, we can now go to the main point... How can I create tools effectively?

How to create tools effectively

The 5 Fundamental Principles from Anthropic

1. Less is more: Choose the right tools Language models don't follow a deterministic flow, so if you fill them with many basic tools you probably won't be able to properly control the result. Once the flow falls into a tool IT'S YOURS!... Here there's code and structured flows you can execute. A well-created tool is worth more than a thousand basic tools.



- Build few well-thought tools instead of many basic ones
- Consolidate related functionalities into a single tool (e.g., instead of `list_users` + `create_event`, create `schedule_event` that searches availability and schedules)
- Avoid tools that only wrap APIs without adding specific value for agents — for that, we might as well ask the model for JSON output and call the API directly, right? 🤷🏾‍♂️

2. Use intelligent namespacing One of the characteristics of language models is that they're trained with millions of lines of code. That's why they'll perfectly understand when you tell them something like `sum()` or `delete_file()`. So tool names should follow these patterns and it will be much easier for them to know which one to choose.

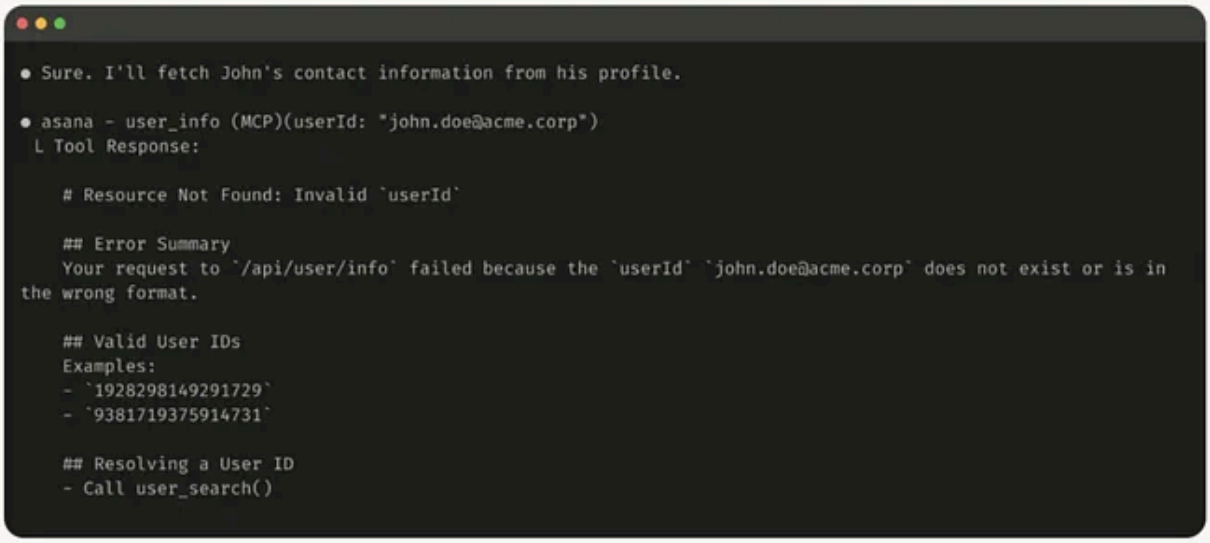
- Group related tools with clear prefixes: `asana_search`, `jira_search`
- Help agents choose the right tool when they have hundreds available
- Define clear functional boundaries between tools

3. Return relevant context, not noise Inside the function you can live in a world of code, variables, and IDs. But once this returns to the model, we should already be working with natural language (Markdown much better). Avoid sending information that

won’t add value to the model. Treat it like a dev who only wants to know the execution result through documentation.

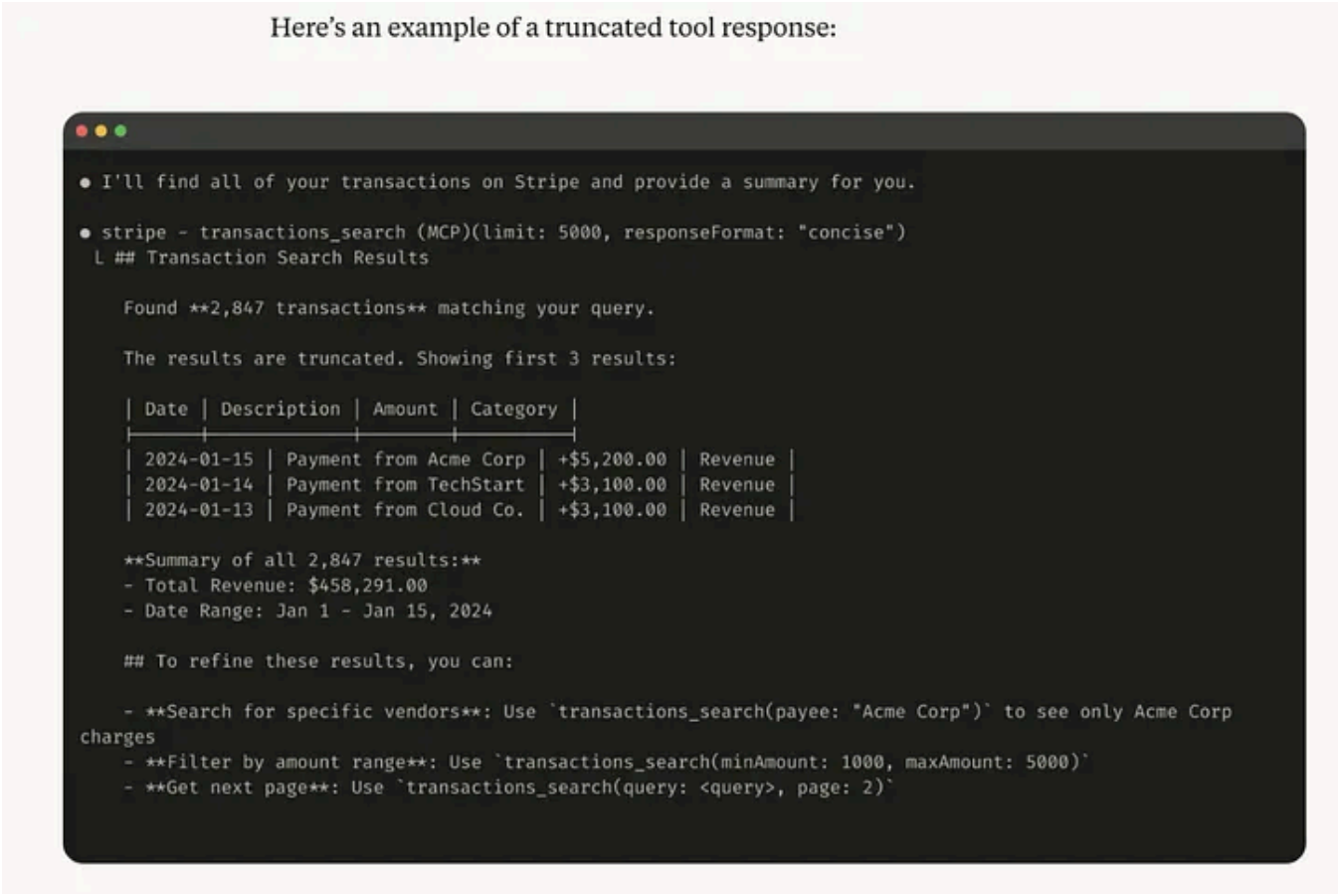
Here’s an example of an unhelpful error response:

Here’s an example of a helpful error response:



- Only relevant information, eliminate unnecessary technical IDs
- Use semantically meaningful identifiers vs cryptic UUIDs
- Implement `response_format` parameters (concise/detailed) to control verbosity

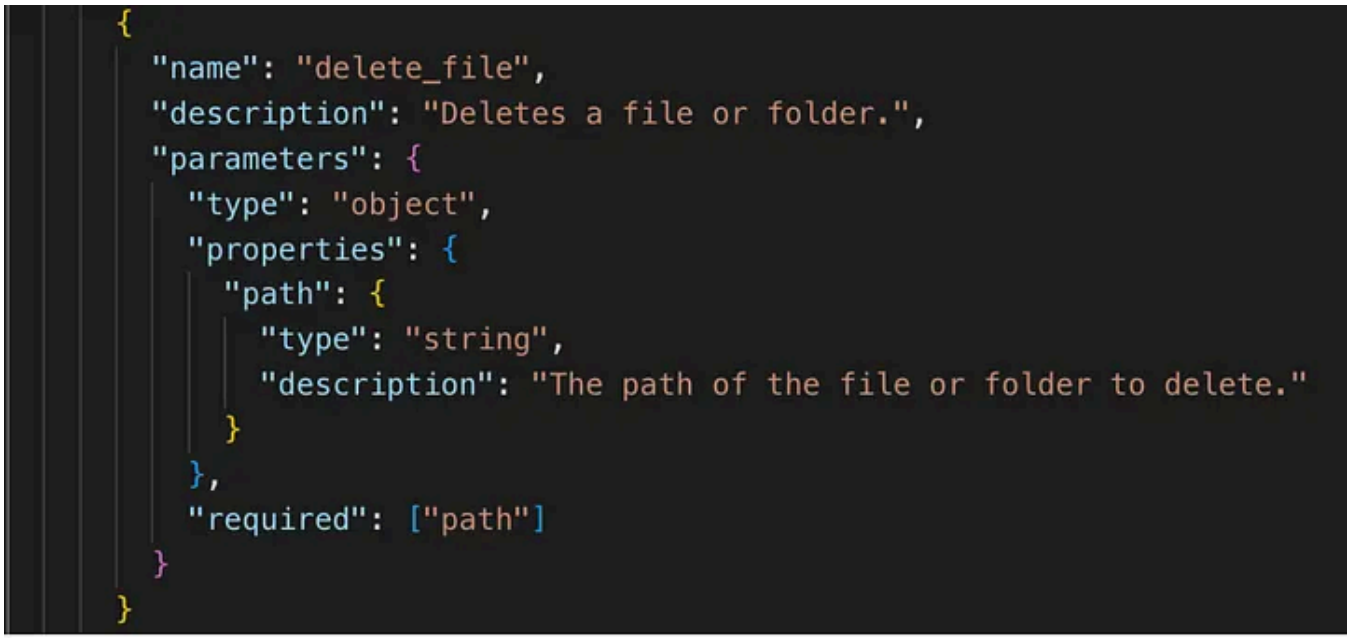
4. Optimize for token efficiency A tool already adds tokens to your context window. If you also add the result of this tool, there will be many more tokens to deal with. Imagine the tool executes SQL queries on a database with hundreds of billions of rows — with a single execution you can break the entire flow. Limit and you’ll win!



- Implement pagination, filters, and truncation with default values
- Limit responses (Anthropic uses 25K tokens maximum)
- Design error messages that guide toward better strategies

5. Prompt-engineer tool descriptions Perhaps one of the most important points. The tool description and its parameters. When a tool is simple… make it simple, the model just wants to understand what it's for and an unambiguous description is the best path.

This is an example of CodeGPT's `delete_file` tool, straight to the point!



```
{
  "name": "delete_file",
  "description": "Deletes a file or folder.",
  "parameters": {
    "type": "object",
    "properties": {
      "path": {
        "type": "string",
        "description": "The path of the file or folder to delete."
      }
    },
    "required": ["path"]
  }
}
```

- Write as if it were for a new employee on your technical team
- Make implicit context and specialized terminology explicit
- Use unambiguous parameter names (use “user_id”, don’t use “user”)

Additional Recommendations from CodeGPT for VSCode

Multi-model compatibility:

- **Ultra-clear schemas**: Models like Gemini or Claude interpret schemas differently. Use explicit `enum` and examples in descriptions
- **Robust error handling**: Some models are more prone to format errors. Validate inputs strictly and return specific errors
- **Functionality description**: Be explicit about WHAT the tool does, not just HOW to use it. GPT-4 vs Claude can interpret the same description very differently

VSCode specific:

- **Context awareness**: Include current workspace information (open files, language, project) in tool responses
- **Performance**: Tools must be fast. VSCode users expect immediate responses, not like web tools that can take seconds
- **Native integration**: Leverage VSCode APIs (show in editor, create files, navigate) instead of just returning text

- **Appropriate scope**: A tool for “search files” is more useful than one for “read specific file” in the development context

The key is to **constantly evaluate** with real cases from your users and iterate based on how different models use your tools in practice.

Now it’s your turn!