

Writing effective tools for agents — with agents

Agents are only as effective as the tools we give them. We share how to write high-quality tools and evaluations, and how you can boost performance by using Claude to optimize its tools for itself.

Published Sep 11, 2025

The Model Context Protocol (MCP) can empower LLM agents with potentially hundreds of tools to solve real-world tasks. But how do we make those tools maximally effective?

In this post, we describe our most effective techniques for improving performance in a variety of agentic AI systems¹.

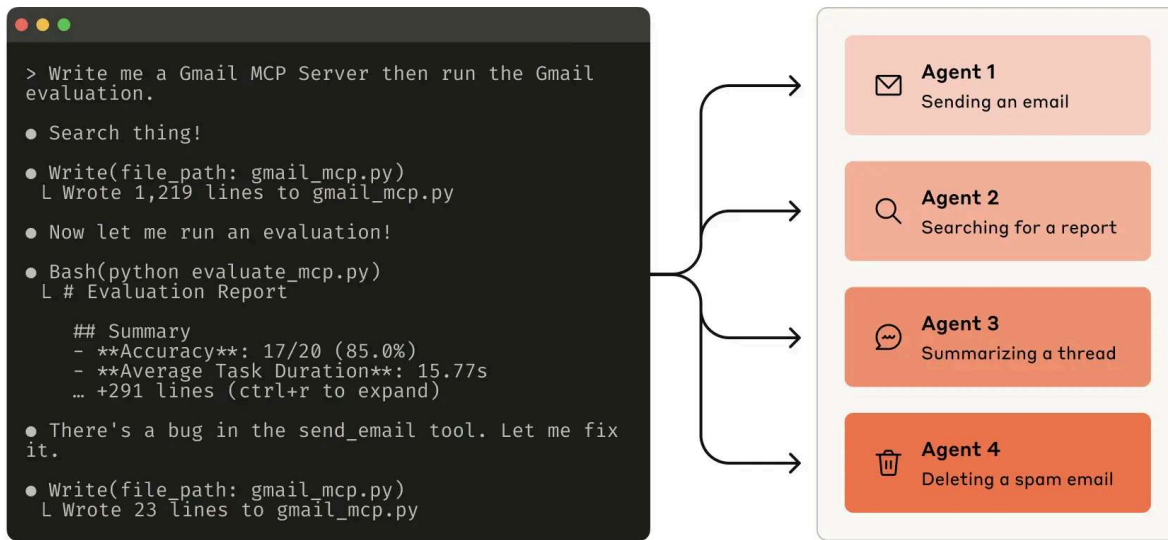
We begin by covering how you can:

- Build and test prototypes of your tools
- Create and run comprehensive evaluations of your tools with agents
- Collaborate with agents like Claude Code to automatically increase the performance of your tools

We conclude with key principles for writing high-quality tools we've identified along the way:

- Choosing the right tools to implement (and not to implement)
- Namespacing tools to define clear boundaries in functionality
- Returning meaningful context from tools back to agents
- Optimizing tool responses for token efficiency
- Prompt-engineering tool descriptions and specs

Collaborating with Claude Code



Building an evaluation allows you to systematically measure the performance of your tools. You can use Claude Code to automatically optimize your tools against this evaluation.

What is a tool?

In computing, deterministic systems produce the same output every time given identical inputs, while *non-deterministic* systems—like agents—can generate varied responses even with the same starting conditions.

When we traditionally write software, we're establishing a contract between deterministic systems. For instance, a function call like `getWeather("NYC")` will always fetch the weather in New York City in the exact same manner every time it is called.

Tools are a new kind of software which reflects a contract between deterministic systems and non-deterministic agents. When a user asks "Should I bring an umbrella today?," an agent might call the weather tool, answer from general knowledge, or even ask a clarifying question about location first. Occasionally, an agent might hallucinate or even fail to grasp how to use a tool.

This means fundamentally rethinking our approach when writing software for agents: instead of writing tools and MCP servers the way we'd write functions and APIs for other developers or systems, we need to design them for agents.

Our goal is to increase the surface area over which agents can be effective in solving a wide range of tasks by using tools to pursue a variety of successful strategies. Fortunately, in our experience, the tools that are most “ergonomic” for agents also end up being surprisingly intuitive to grasp as humans.

How to write tools

In this section, we describe how you can collaborate with agents both to write and to improve the tools you give them. Start by standing up a quick prototype of your tools and testing them locally. Next, run a comprehensive evaluation to measure subsequent changes. Working alongside agents, you can repeat the process of evaluating and improving your tools until your agents achieve strong performance on real-world tasks.

Building a prototype

It can be difficult to anticipate which tools agents will find ergonomic and which tools they won’t without getting hands-on yourself. Start by standing up a quick prototype of your tools. If you’re using [Claude Code](#) to write your tools (potentially in one-shot), it helps to give Claude documentation for any software libraries, APIs, or SDKs (including potentially the [MCP SDK](#)) your tools will rely on. LLM-friendly documentation can commonly be found in flat `llms.txt` files on official documentation sites (here’s our [API’s](#)).

Wrapping your tools in a [local MCP server](#) or [Desktop extension](#) (DXT) will allow you to connect and test your tools in Claude Code or the Claude Desktop app.

To connect your local MCP server to Claude Code, run `claude mcp add <name> <command> [args...]`.

To connect your local MCP server or DXT to the Claude Desktop app, navigate to Settings > Developer Or Settings > Extensions, respectively.

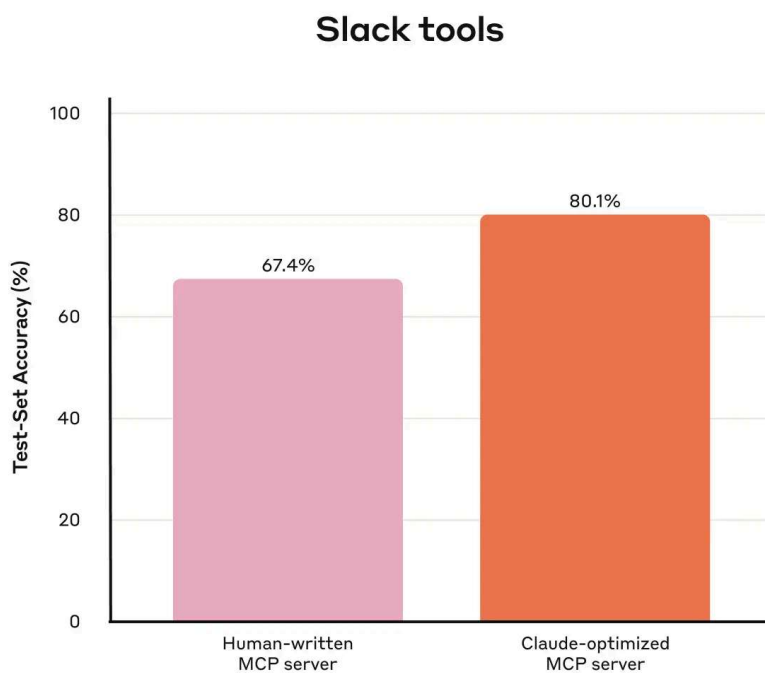
Tools can also be passed directly into [Anthropic API](#) calls for programmatic testing.

Test the tools yourself to identify any rough edges. Collect feedback from your users to build an intuition around the use-cases and prompts you expect your

tools to enable.

Running an evaluation

Next, you need to measure how well Claude uses your tools by running an evaluation. Start by generating lots of evaluation tasks, grounded in real world uses. We recommend collaborating with an agent to help analyze your results and determine how to improve your tools. See this process end-to-end in our [tool evaluation cookbook](#).



Held-out test set performance of our internal Slack tools

Generating evaluation tasks

With your early prototype, Claude Code can quickly explore your tools and create dozens of prompt and response pairs. Prompts should be inspired by real-world uses and be based on realistic data sources and services (for example, internal knowledge bases and microservices). We recommend you avoid overly simplistic or superficial “sandbox” environments that don’t stress-test your tools with sufficient complexity. Strong evaluation tasks might require multiple tool calls—potentially dozens.

Here are some examples of strong tasks:

- Schedule a meeting with Jane next week to discuss our latest Acme Corp project. Attach the notes from our last project planning meeting and reserve a conference room.
- Customer ID 9182 reported that they were charged three times for a single purchase attempt. Find all relevant log entries and determine if any other customers were affected by the same issue.
- Customer Sarah Chen just submitted a cancellation request. Prepare a retention offer. Determine: (1) why they're leaving, (2) what retention offer would be most compelling, and (3) any risk factors we should be aware of before making an offer.

And here are some weaker tasks:

- Schedule a meeting with jane@acme.corp next week.
- Search the payment logs for `purchase_complete` and `customer_id=9182`.
- Find the cancellation request by Customer ID 45892.

Each evaluation prompt should be paired with a verifiable response or outcome. Your verifier can be as simple as an exact string comparison between ground truth and sampled responses, or as advanced as enlisting Claude to judge the response. Avoid overly strict verifiers that reject correct responses due to spurious differences like formatting, punctuation, or valid alternative phrasings.

For each prompt-response pair, you can optionally also specify the tools you expect an agent to call in solving the task, to measure whether or not agents are successful in grasping each tool's purpose during evaluation. However, because there might be multiple valid paths to solving tasks correctly, try to avoid overspecifying or overfitting to strategies.

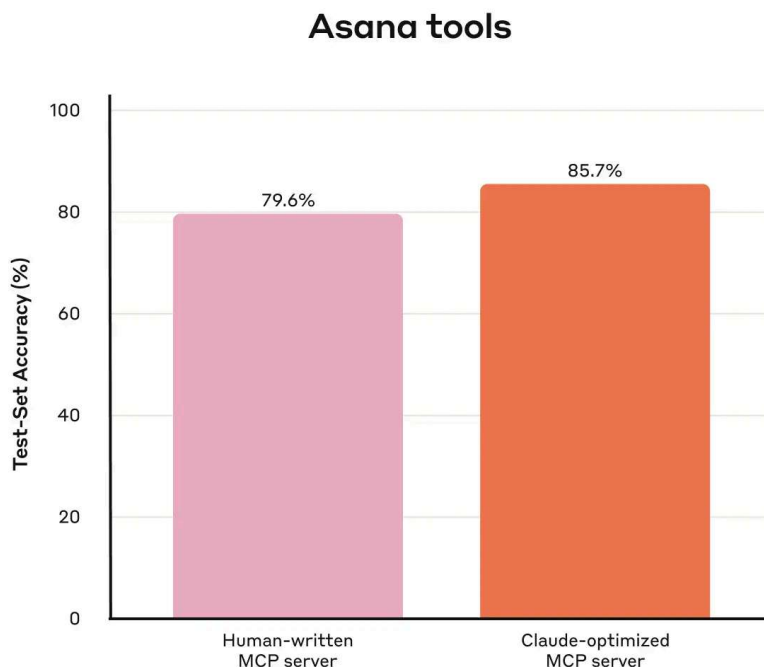
Running the evaluation

We recommend running your evaluation programmatically with direct LLM API calls. Use simple agentic loops (`while`-loops wrapping alternating LLM API and tool calls): one loop for each evaluation task. Each evaluation agent should be given a single task prompt and your tools.

In your evaluation agents' system prompts, we recommend instructing agents to output not just structured response blocks (for verification), but also reasoning and feedback blocks. Instructing agents to output these *before* tool call and response blocks may increase LLMs' effective intelligence by triggering chain-of-thought (CoT) behaviors.

If you're running your evaluation with Claude, you can turn on interleaved thinking for similar functionality "off-the-shelf". This will help you probe why agents do or don't call certain tools and highlight specific areas of improvement in tool descriptions and specs.

As well as top-level accuracy, we recommend collecting other metrics like the total runtime of individual tool calls and tasks, the total number of tool calls, the total token consumption, and tool errors. Tracking tool calls can help reveal common workflows that agents pursue and offer some opportunities for tools to consolidate.



Held-out test set performance of our internal Asana tools

Analyzing results

Agents are your helpful partners in spotting issues and providing feedback on everything from contradictory tool descriptions to inefficient tool implementations and confusing tool schemas. However, keep in mind that what agents omit in their feedback and responses can often be more important than what they include. LLMs don't always say what they mean.

Observe where your agents get stumped or confused. Read through your evaluation agents' reasoning and feedback (or CoT) to identify rough edges. Review the raw transcripts (including tool calls and tool responses) to catch any behavior not explicitly described in the agent's CoT. Read between the lines; remember that your evaluation agents don't necessarily know the correct answers and strategies.

Analyze your tool calling metrics. Lots of redundant tool calls might suggest some rightsizing of pagination or token limit parameters is warranted; lots of tool errors for invalid parameters might suggest tools could use clearer descriptions or better examples. When we launched Claude's web search tool, we identified that Claude was needlessly appending 2025 to the tool's query parameter, biasing search results and degrading performance (we steered Claude in the right direction by improving the tool description).

Collaborating with agents

You can even let agents analyze your results and improve your tools for you. Simply concatenate the transcripts from your evaluation agents and paste them into Claude Code. Claude is an expert at analyzing transcripts and refactoring lots of tools all at once—for example, to ensure tool implementations and descriptions remain self-consistent when new changes are made.

In fact, most of the advice in this post came from repeatedly optimizing our internal tool implementations with Claude Code. Our evaluations were created on top of our internal workspace, mirroring the complexity of our internal workflows, including real projects, documents, and messages.

We relied on held-out test sets to ensure we did not overfit to our “training” evaluations. These test sets revealed that we could extract additional

performance improvements even beyond what we achieved with "expert" tool implementations—whether those tools were manually written by our researchers or generated by Claude itself.

In the next section, we'll share some of what we learned from this process.

Principles for writing effective tools

In this section, we distill our learnings into a few guiding principles for writing effective tools.

Choosing the right tools for agents

More tools don't always lead to better outcomes. A common error we've observed is tools that merely wrap existing software functionality or API endpoints—whether or not the tools are appropriate for agents. This is because agents have distinct “affordances” to traditional software—that is, they have different ways of perceiving the potential actions they can take with those tools

LLM agents have limited "context" (that is, there are limits to how much information they can process at once), whereas computer memory is cheap and abundant. Consider the task of searching for a contact in an address book. Traditional software programs can efficiently store and process a list of contacts one at a time, checking each one before moving on.

However, if an LLM agent uses a tool that returns ALL contacts and then has to read through each one token-by-token, it's wasting its limited context space on irrelevant information (imagine searching for a contact in your address book by reading each page from top-to-bottom—that is, via brute-force search). The better and more natural approach (for agents and humans alike) is to skip to the relevant page first (perhaps finding it alphabetically).

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.

Tools can consolidate functionality, handling potentially *multiple* discrete operations (or API calls) under the hood. For example, tools can enrich tool responses with related metadata or handle frequently chained, multi-step tasks in a single tool call.

Here are some examples:

- Instead of implementing a `list_users`, `list_events`, and `create_event` tools, consider implementing a `schedule_event` tool which finds availability and schedules an event.
- Instead of implementing a `read_logs` tool, consider implementing a `search_logs` tool which only returns relevant log lines and some surrounding context.
- Instead of implementing `get_customer_by_id`, `list_transactions`, and `list_notes` tools, implement a `get_customer_context` tool which compiles all of a customer's recent & relevant information all at once.

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 your tools

Your AI agents will potentially gain access to dozens of MCP servers and hundreds of different tools—including those by other developers. When tools overlap in function or have a vague purpose, agents can get confused about which ones to use.

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.

We have found selecting between prefix- and suffix-based namespacing to have non-trivial effects on our tool-use evaluations. Effects vary by LLM and we encourage you to choose a naming scheme according to your own evaluations.

Agents might call the wrong tools, call the right tools with the wrong parameters, call too few tools, or process tool responses incorrectly. By selectively implementing tools whose names reflect natural subdivisions of tasks, you simultaneously reduce the number of tools and tool descriptions loaded into the agent's context and offload agentic computation from the agent's context back into the tool calls themselves. This reduces an agent's overall risk of making mistakes.

Returning meaningful context from your tools

In the same vein, tool implementations should take care to return only high signal information back to agents. They should prioritize contextual relevance over flexibility, and eschew low-level technical identifiers (for example: `uuid`, `256px_image_url`, `mime_type`). Fields like `name`, `image_url`, and `file_type` are much more likely to directly inform agents' downstream actions and responses.

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.

In some instances, agents may require the flexibility to interact with both natural language and technical identifiers outputs, if only to trigger downstream tool calls (for example, `search_user(name='jane')` → `send_message(id=12345)`). 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).

You can add more formats for even greater flexibility, similar to GraphQL where you can choose exactly which pieces of information you want to receive. Here is an example `ResponseFormat` enum to control tool response verbosity:

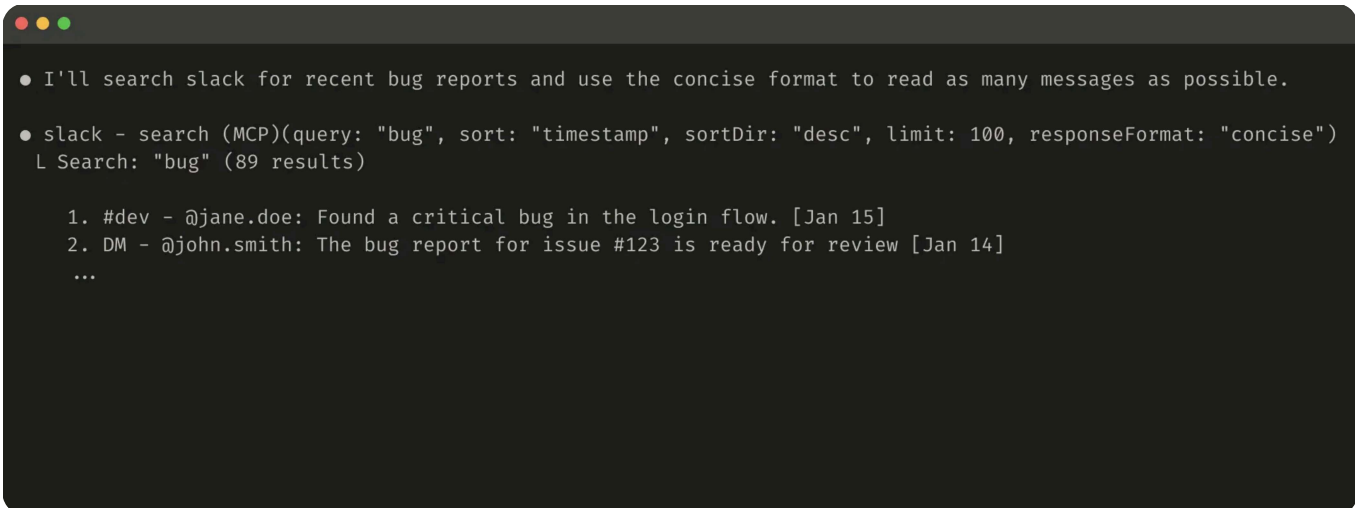
```
enum ResponseFormat {  
    DETAILED = "detailed",  
    CONCISE = "concise"  
}
```

 Copy

Here's an example of a detailed tool response (206 tokens):

```
● I'll search slack for recent bug reports and use the detailed format to see which channel IDs and threads to investigate further.  
  
● slack - search (MCP)(query: "bug", sort: "timestamp", sortDir: "desc", limit: 100, responseFormat: "detailed")  
  L Search results for: "bug"  
  
    ≡ Result 1 of 89 ≡  
    Channel: #dev (C1234567890)  
    From: @jane.doe (U123456789)  
    Time: 2024-01-15 10:30:45 UTC  
    TS: 1705316445.123456  
    Text: Found a critical bug in the login flow.  
  
    ≡ Result 2 of 89 ≡  
    Channel: DM with @john.smith  
    From: @john.smith (U987654321)  
    Time: 2024-01-14 15:22:18 UTC  
    TS: 1705247738.234567  
    Text: The bug report for issue #123 is ready for review  
    Files: bug-report-123.pdf  
    ...
```

Here's an example of a concise tool response (72 tokens):



```
● I'll search slack for recent bug reports and use the concise format to read as many messages as possible.
● slack - search (MCP)(query: "bug", sort: "timestamp", sortDir: "desc", limit: 100, responseFormat: "concise")
  L Search: "bug" (89 results)

  1. #dev - @jane.doe: Found a critical bug in the login flow. [Jan 15]
  2. DM - @john.smith: The bug report for issue #123 is ready for review [Jan 14]
  ...
```

Slack threads and thread replies are identified by unique `thread_ts` which are required to fetch thread replies. `thread_ts` and other IDs (`channel_id`, `user_id`) can be retrieved from a “detailed” tool response to enable further tool calls that require these. “concise” tool responses return only thread content and exclude IDs. In this example, we use $\sim\frac{1}{3}$ of the tokens with “concise” tool responses.

Even your tool response structure—for example XML, JSON, or Markdown—can have an impact on evaluation performance: there is no one-size-fits-all solution. This is because LLMs are trained on next-token prediction and tend to perform better with formats that match their training data. The optimal response structure will vary widely by task and agent. We encourage you to select the best response structure based on your own evaluation.

Optimizing tool responses for token efficiency

Optimizing the quality of context is important. But so is optimizing the *quantity* of context returned back to agents in tool responses.

We suggest implementing some combination of pagination, range selection, filtering, and/or truncation with sensible default parameter values for any tool responses that could use up lots of context. For Claude Code, we restrict tool responses to 25,000 tokens by default. We expect the effective context length of agents to grow over time, but the need for context-efficient tools to remain.

If you choose to truncate responses, be sure to steer agents with helpful instructions. You can directly encourage agents to pursue more token-efficient strategies, like making many small and targeted searches instead of a single, broad search for a knowledge retrieval task. Similarly, if a tool call raises an error (for example, during input validation), you can prompt-engineer your error

responses to clearly communicate specific and actionable improvements, rather than opaque error codes or tracebacks.

Here's an example of a truncated tool response:

```
● I'll find all of your transactions on Stripe and provide a summary for you.
● stripe - transactions_search (MCP)(limit: 5000, responseFormat: "concise")
  L ## Transaction Search Results

  Found **2,847 transactions** matching your query.

  The results are truncated. Showing first 3 results:

  | Date | Description | Amount | Category |
  |-----|-----|-----|-----|
  | 2024-01-15 | Payment from Acme Corp | +$5,200.00 | Revenue |
  | 2024-01-14 | Payment from TechStart | +$3,100.00 | Revenue |
  | 2024-01-13 | Payment from Cloud Co. | +$3,100.00 | Revenue |

  **Summary of all 2,847 results:**
  - Total Revenue: $458,291.00
  - Date Range: Jan 1 - Jan 15, 2024

  ## To refine these results, you can:

  - **Search for specific vendors**: Use `transactions_search(payee: "Acme Corp")` to see only Acme Corp charges
  - **Filter by amount range**: Use `transactions_search(minAmount: 1000, maxAmount: 5000)`
  - **Get next page**: Use `transactions_search(query: <query>, page: 2)`
```

Here's an example of an unhelpful error response:

```
● Sure. I'll fetch John's contact information from his profile.
● asana - user_info (MCP)(userId: "john.doe@acme.corp")
  L {
    "error": {
      "code": "RESOURCE_NOT_FOUND",
      "status": 422,
      "message": "Invalid value",
      "details": {
        "field": "userId",
        "value": "john.doe@acme.corp",
        "type": "invalid_value"
      }
    }
  }
}
```

Here's an example of a helpful error response:

```
• Sure. I'll fetch John's contact information from his profile.

• asana - user_info (MCP)(userId: "john.doe@acme.corp")
  L Tool Response:

    # Resource Not Found: Invalid `userId`

    ## Error Summary
    Your request to `/api/user/info` failed because the `userId` `john.doe@acme.corp` does not exist or is in
the wrong format.

    ## Valid User IDs
    Examples:
    - `1928298149291729`
    - `9381719375914731`

    ## Resolving a User ID
    - Call user_search()
```

Tool truncation and error responses can steer agents towards more token-efficient tool-use behaviors (using filters or pagination) or give examples of correctly formatted tool inputs.

Prompt-engineering your tool descriptions

We now come to one of the most effective methods for improving tools: prompt-engineering your tool descriptions and specs. Because these are loaded into your agents' context, they can collectively steer agents toward effective tool-calling behaviors.

When writing tool descriptions and specs, think of how you would describe your tool to a new hire on your team. Consider the context that you might implicitly bring—specialized query formats, definitions of niche terminology, relationships between underlying resources—and make it explicit. Avoid ambiguity by clearly describing (and enforcing with strict data models) expected inputs and outputs. In particular, input parameters should be unambiguously named: instead of a parameter named `user`, try a parameter named `user_id`.

With your evaluation you can measure the impact of your prompt engineering with greater confidence. Even small refinements to tool descriptions can yield dramatic improvements. Claude Sonnet 3.5 achieved state-of-the-art performance on the [SWE-bench Verified](#) evaluation after we made precise

refinements to tool descriptions, dramatically reducing error rates and improving task completion.

You can find other best practices for tool definitions in our [Developer Guide](#). If you're building tools for Claude, we also recommend reading about how tools are dynamically loaded into Claude's [system prompt](#). Lastly, if you're writing tools for an MCP server, [tool annotations](#) help disclose which tools require open-world access or make destructive changes.

Looking ahead

To build effective tools for agents, we need to re-orient our software development practices from predictable, deterministic patterns to non-deterministic ones.

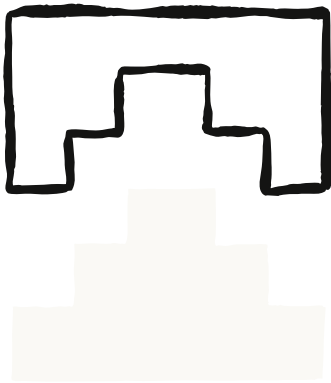
Through the iterative, evaluation-driven process we've described in this post, we've identified consistent patterns in what makes tools successful: Effective tools are intentionally and clearly defined, use agent context judiciously, can be combined together in diverse workflows, and enable agents to intuitively solve real-world tasks.

In the future, we expect the specific mechanisms through which agents interact with the world to evolve—from updates to the MCP protocol to upgrades to the underlying LLMs themselves. With a systematic, evaluation-driven approach to improving tools for agents, we can ensure that as agents become more capable, the tools they use will evolve alongside them.

Acknowledgements

Written by Ken Aizawa with valuable contributions from colleagues across Research (Barry Zhang, Zachary Witten, Daniel Jiang, Sami Al-Sheikh, Matt Bell, Maggie Vo), MCP (Theodora Chu, John Welsh, David Soria Parra, Adam Jones), Product Engineering (Santiago Seira), Marketing (Molly Vorwerck), Design (Drew Roper), and Applied AI (Christian Ryan, Alexander Bricken).

¹Beyond training the underlying LLMs themselves.



Looking to learn more?

Master API development, Model Context Protocol, and Claude Code with courses on Anthropic Academy. Earn certificates upon completion.

[Explore courses](#)

Get the developer newsletter

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

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

