How to implement tool use

Copy page

Choosing a model

We recommend using the latest Claude Sonnet (4.5) or Claude Opus (4.5) model for complex tools and ambiguous queries; they handle multiple tools better and seek clarification when needed.

Use Claude Haiku models for straightforward tools, but note they may infer missing parameters.

If using Claude with tool use and extended thinking, refer to our guide here for more information.

Specifying client tools

Client tools (both Anthropic-defined and user-defined) are specified in the `tools` top-level parameter of the API request. Each tool definition includes:

Parameter	Description
`name`	The name of the tool. Must match the regex `^[a-zA-Z0-9_-]{1,64}$` .
`description`	A detailed plaintext description of what the tool does, when it should be used, and how it behaves.
`input_schema`	A JSON Schema object defining the expected parameters for the tool.
`input_examples`	(Optional, beta) An array of example input objects to help Claude understand how to use the tool. See Providing tool use examples.

Example simple tool definition

Claude Docs

When you call the Claude API with the `tools` parameter, we construct a special system prompt from the tool definitions, tool configuration, and any user-specified system prompt. The constructed prompt is designed to instruct the model to use the specified tool(s) and provide the necessary context for the tool to operate properly:

```
In this environment you have access to a set of tools you can use to answ th
{{ FORMATTING INSTRUCTIONS }}
String and scalar parameters should be specified as is, while lists and object
Here are the functions available in JSONSchema format:
{{ TOOL DEFINITIONS IN JSON SCHEMA }}
{{ USER SYSTEM PROMPT }}
{{ TOOL CONFIGURATION }}
```

Best practices for tool definitions

To get the best performance out of Claude when using tools, follow these guidelines:

- **Provide extremely detailed descriptions.** This is by far the most important factor in tool performance. Your descriptions should explain every detail about the tool, including:
  - What the tool does
  - When it should be used (and when it shouldn't)
  - What each parameter means and how it affects the tool's behavior
  - Any important caveats or limitations, such as what information the tool does not return if the tool name is unclear. The more context you can give Claude about your tools, the better it will be at deciding when and how to use them. Aim for at least 3-4 sentences per tool description, more if the tool is complex.
- **Prioritize descriptions, but consider using `input_examples` for complex tools.** Clear descriptions are most important, but for tools with complex inputs, nested objects, or format-sensitive parameters, you can use the `input_examples` field (beta) to provide schema-validated examples. See Providing tool use examples for details.

> **Example of a good tool description**

> **Example poor tool description**

leaves Claude with many open questions about the tool's behavior and usage.

Providing tool use examples

You can provide concrete examples of valid tool inputs to help Claude understand how to use your tools more effectively. This is particularly useful for complex tools with nested objects, optional parameters, or format-sensitive inputs.

Tool use examples is a beta feature. Include the appropriate beta header for your provider:

Provider	Beta header	Supported models
Claude API, Microsoft Foundry	`advanced-tool-use-2025-11-20`	All models
Vertex AI, Amazon Bedrock	`tool-examples-2025-10-29`	Claude Opus 4.5 only

Basic usage

Add an optional `input_examples` field to your tool definition with an array of example input objects. Each example must be valid according to the tool's `input_schema` :

Python

Claude Docs

```python
client = anthropic.Anthropic()

response = client.messages.create(
    model="claude-sonnet-4-5-20250929",
    max_tokens=1024,
    betas=["advanced-tool-use-2025-11-20"],
    tools=[
        {
            "name": "get_weather",
            "description": "Get the current weather in a given location",
            "input_schema": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "The city and state, e.g. San Francisco
                    },
                    "unit": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"],
                        "description": "The unit of temperature"
                    }
                },
                "required": ["location"]
            },
            "input_examples": [
                {
                    "location": "San Francisco, CA",
                    "unit": "fahrenheit"
                },
                {
                    "location": "Tokyo, Japan",
                    "unit": "celsius"
                },
                {
                    "location": "New York, NY"  # 'unit' is optional
                }
            ]
        }
    ],
    messages=[
        {"role": "user", "content": "What's the weather like in San Francisco?
    ]
)
```

include optional parameters, what formats to use, and how to structure complex inputs.

Requirements and limitations

- **Schema validation** - Each example must be valid according to the tool's `input_schema`. Invalid examples return a 400 error
- **Not supported for server-side tools** - Only user-defined tools can have input examples
- **Token cost** - Examples add to prompt tokens: ~20-50 tokens for simple examples, ~100-200 tokens for complex nested objects

Tool runner (beta)

The tool runner provides an out-of-the-box solution for executing tools with Claude. Instead of manually handling tool calls, tool results, and conversation management, the tool runner automatically:

- Executes tools when Claude calls them
- Handles the request/response cycle
- Manages conversation state
- Provides type safety and validation

We recommend that you use the tool runner for most tool use implementations.

The tool runner is currently in beta and available in the Python, TypeScript, and Ruby SDKs.

Automatic context management with compaction

The tool runner supports automatic compaction, which generates summaries when token usage exceeds a threshold. This allows long-running agentic tasks to continue beyond context window limits.

Basic usage

Define tools using the SDK helpers, then use the tool runner to execute them.

Python TypeScript Ruby

Claude Docs

> If you're using the async client, replace `@beta_tool` with `@beta_async_tool` and define the function with `async def`.

```python
import anthropic
import json
from anthropic import beta_tool

# Initialize client
client = anthropic.Anthropic()

# Define tools using the decorator
@beta_tool
def get_weather(location: str, unit: str = "fahrenheit") -> str:
    """Get the current weather in a given location.

    Args:
        location: The city and state, e.g. San Francisco, CA
        unit: Temperature unit, either 'celsius' or 'fahrenheit'
    """
    # In a full implementation, you'd call a weather API here
    return json.dumps({"temperature": "20°C", "condition": "Sunny"})

@beta_tool
def calculate_sum(a: int, b: int) -> str:
    """Add two numbers together.

    Args:
        a: First number
        b: Second number
    """
    return str(a + b)

# Use the tool runner
runner = client.beta.messages.tool_runner(
    model="claude-sonnet-4-5",
    max_tokens=1024,
    tools=[get_weather, calculate_sum],
    messages=[
        {"role": "user", "content": "What's the weather like in Paris? Also, w
    ]
)
for message in runner:
    print(message.content[0].text)
```

Claude Docs

```
{
  "name": "calculate_sum",
  "description": "Adds two integers together.",
  "input_schema": {
    "additionalProperties": false,
    "properties": {
      "left": {
        "description": "The first integer to add.",
        "title": "Left",
        "type": "integer"
      },
      "right": {
        "description": "The second integer to add.",
        "title": "Right",
        "type": "integer"
      }
    },
    "required": ["left", "right"],
    "type": "object"
  }
}
```

The tool function must return a content block or content block array, including text, images, or document blocks. This allows tools to return rich, multimodal responses. Returned strings will be converted to a text content block. If you want to return a structured JSON object to Claude, encode it to a JSON string before returning it. Numbers, booleans, or other non-string primitives must also be converted to strings.

Iterating over the tool runner

The tool runner is an iterable that yields messages from Claude. This is often referred to as a "tool call loop". Each iteration, the runner checks if Claude requested a tool use. If so, it calls the tool and sends the result back to Claude automatically, then yields the next message from Claude to continue your loop.

You can end the loop at any iteration with a `break` statement. The runner will loop until Claude returns a message without a tool use.

If you don't need intermediate messages, you can get the final message directly:

Python TypeScript Ruby

Claude Docs

```
runner = client.beta.messages.tool_runner(
    model="claude-sonnet-4-5",
    max_tokens=1024,
    tools=[get_weather, calculate_sum],
    messages=[
        {"role": "user", "content": "What's the weather like in Paris? Also, w
    ]
)
final_message = runner.until_done()
print(final_message.content[0].text)
```

Advanced usage

Within the loop, you can fully customize the tool runner's next request to the Messages API. The runner automatically appends tool results to the message history, so you don't need to manually manage them. You can optionally inspect the tool result for logging or debugging, and modify the request parameters before the next API call.

Python TypeScript Ruby

Use `generate_tool_call_response()` to optionally inspect the tool result (the runner appends it automatically). Use `set_messages_params()` and `append_messages()` to modify the request.

```
    max_tokens=1024,
    tools=[get_weather],
    messages=[{"role": "user", "content": "What's the weather in San Francisco
)
for message in runner:
    # Optional: inspect the tool response (automatically appended by the runne
    tool_response = runner.generate_tool_call_response()
    if tool_response:
        print(f"Tool result: {tool_response}")

    # Customize the next request
    runner.set_messages_params(lambda params: {
        **params,
        "max_tokens": 2048  # Increase tokens for next request
    })

    # Or add additional messages
    runner.append_messages(
        {"role": "user", "content": "Please be concise in your response."}
    )
```

Debugging tool execution

When a tool throws an exception, the tool runner catches it and returns the error to Claude as a tool result with `is_error: true`. By default, only the exception message is included, not the full stack trace.

To view full stack traces and debug information, set the `ANTHROPIC_LOG` environment variable:

```
# View info-level logs including tool errors
export ANTHROPIC_LOG=info

# View debug-level logs for more verbose output
export ANTHROPIC_LOG=debug
```

When enabled, the SDK logs full exception details (using Python's `logging` module, the console in TypeScript, or Ruby's logger), including the complete stack trace when a tool fails.

Claude Docs

By default, tool errors are passed back to Claude, which can then respond appropriately. However, you may want to detect errors and handle them differently—for example, to stop execution early or implement custom error handling.

Use the tool response method to intercept tool results and check for errors before they're sent to Claude:

Python TypeScript Ruby

```python
import json

runner = client.beta.messages.tool_runner(
    model="claude-sonnet-4-5",
    max_tokens=1024,
    tools=[my_tool],
    messages=[{"role": "user", "content": "Run the tool"}]
)

for message in runner:
    tool_response = runner.generate_tool_call_response()

    if tool_response:
        # Check if any tool result has an error
        for block in tool_response.content:
            if block.is_error:
                # Option 1: Raise an exception to stop the loop
                raise RuntimeError(f"Tool failed: {json.dumps(block.content)}"

                # Option 2: Log and continue (let Claude handle it)
                # logger.error(f"Tool error: {json.dumps(block.content)}")

    # Process the message normally
    print(message.content)
```

Modifying tool results

You can modify tool results before they're sent back to Claude. This is useful for adding metadata like `cache_control` to enable prompt caching on tool results, or for transforming the tool output.

Claude Docs

Python TypeScript Ruby

```python
runner = client.beta.messages.tool_runner(
    model="claude-sonnet-4-5",
    max_tokens=1024,
    tools=[search_documents],
    messages=[{"role": "user", "content": "Search for information about the cl
)

for message in runner:
    tool_response = runner.generate_tool_call_response()

    if tool_response:
        # Modify the tool result to add cache control
        for block in tool_response.content:
            if block.type == "tool_result":
                # Add cache_control to cache this tool result
                block.cache_control = {"type": "ephemeral"}

        # Append the modified response (this prevents auto-append of original)
        runner.append_messages(message, tool_response)

    print(message.content)
```

Adding `cache_control` to tool results is particularly useful when tools return large amounts of data (like document search results) that you want to cache for subsequent API calls. See Prompt caching for more details on caching strategies.

Streaming

Enable streaming to receive events as they arrive. Each iteration yields a stream object that you can iterate for events.

Python TypeScript Ruby

Set `stream=True` and use `get_final_message()` to get the accumulated message.

```
    max_tokens=1024,
    tools=[calculate_sum],
    messages=[{"role": "user", "content": "What is 15 + 27?"}],
    stream=True
)

# When streaming, the runner returns BetaMessageStream
for message_stream in runner:
    for event in message_stream:
        print('event:', event)
    print('message:', message_stream.get_final_message())

print(runner.until_done())
```

The SDK tool runner is in beta. The rest of this document covers manual tool implementation.

Controlling Claude's output

Forcing tool use

In some cases, you may want Claude to use a specific tool to answer the user's question, even if Claude thinks it can provide an answer without using a tool. You can do this by specifying the tool in the `tool_choice` field like so:

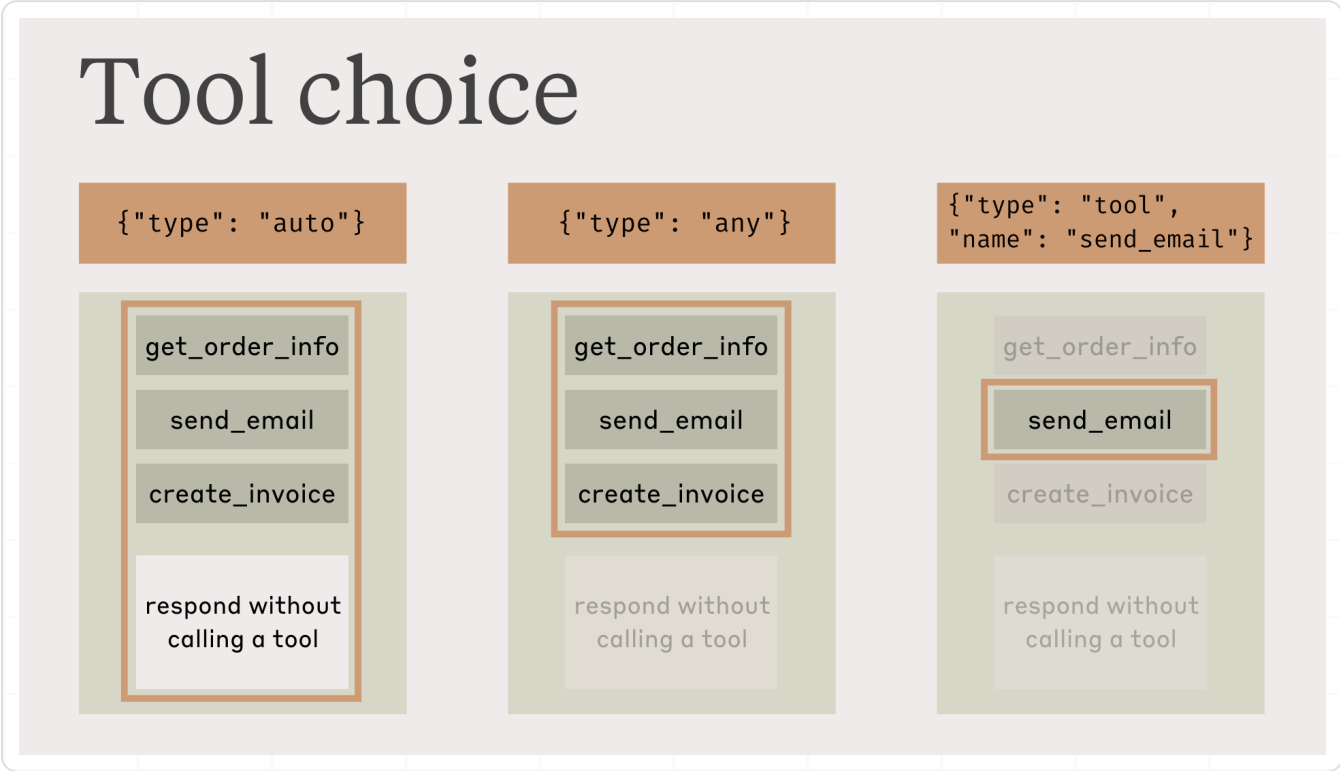
```
tool_choice = {"type": "tool", "name": "get_weather"}
```

When working with the tool_choice parameter, we have four possible options:

- `auto` allows Claude to decide whether to call any provided tools or not. This is the default value when `tools` are provided.
- `any` tells Claude that it must use one of the provided tools, but doesn't force a particular tool.
- `tool` allows us to force Claude to always use a particular tool.
- `none` prevents Claude from using any tools. This is the default value when no `tools` are provided.

content must be reprocessed.

This diagram illustrates how each option works:



Note that when you have `tool_choice` as `any` or `tool`, we will prefill the assistant message to force a tool to be used. This means that the models will not emit a natural language response or explanation before `tool_use` content blocks, even if explicitly asked to do so.

When using extended thinking with tool use, `tool_choice: {"type": "any"}` and `tool_choice: {"type": "tool", "name": "..."}` are not supported and will result in an error. Only `tool_choice: {"type": "auto"}` (the default) and `tool_choice: {"type": "none"}` are compatible with extended thinking.

Our testing has shown that this should not reduce performance. If you would like the model to provide natural language context or explanations while still requesting that the model use a specific tool, you can use `{"type": "auto"}` for `tool_choice` (the default) and add explicit instructions in a `user` message. For example: `What's the weather like in London? Use the get_weather tool in your response.`

Guaranteed tool calls with strict tools

`strict: true` on your tool definitions to enable schema validation.

JSON output

Tools do not necessarily need to be client functions — you can use tools anytime you want the model to return JSON output that follows a provided schema. For example, you might use a `record_summary` tool with a particular schema. See Tool use with Claude for a full working example.

Model responses with tools

When using tools, Claude will often comment on what it's doing or respond naturally to the user before invoking tools.

For example, given the prompt "What's the weather like in San Francisco right now, and what time is it there?", Claude might respond with:

JSON

```
{
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "I'll help you check the current weather and time in San Francis
    },
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": {"location": "San Francisco, CA"}
    }
  ]
}
```

This natural response style helps users understand what Claude is doing and creates a more conversational interaction. You can guide the style and content of these responses through your system prompts and by providing `<examples>` in your prompts.

It's important to note that Claude may use various phrasings and approaches when explaining its actions. Your code should treat these responses like any other assistant-

Claude Docs

Parallel tool use

By default, Claude may use multiple tools to answer a user query. You can disable this behavior by:

- Setting `disable_parallel_tool_use=true` when tool_choice type is `auto`, which ensures that Claude uses **at most one** tool
- Setting `disable_parallel_tool_use=true` when tool_choice type is `any` or `tool`, which ensures that Claude uses **exactly one** tool

> **Complete parallel tool use example**

> **Complete test script for parallel tools**

Maximizing parallel tool use

While Claude 4 models have excellent parallel tool use capabilities by default, you can increase the likelihood of parallel tool execution across all models with targeted prompting:

> **System prompts for parallel tool use**

> **User message prompting**

⚠ **Parallel tool use with Claude Sonnet 3.7**

Claude Sonnet 3.7 may be less likely to make make parallel tool calls in a response, even when you have not set `disable_parallel_tool_use`. We recommend upgrading to Claude 4 models, which have built-in token-efficient tool use and improved parallel tool calling.

If you're still using Claude Sonnet 3.7, you can enable the `token-efficient-tools-2025-02-19` beta header, which helps encourage Claude to use parallel tools. You can also introduce a "batch tool" that can act as a meta-tool to wrap invocations to other tools simultaneously.

See this example in our cookbook for how to use this workaround.

Claude Docs

> **Simpler with Tool runner**: The manual tool handling described in this section is automatically managed by tool runner. Use this section when you need custom control over tool execution.

Claude's response differs based on whether it uses a client or server tool.

Handling results from client tools

The response will have a `stop_reason` of `tool_use` and one or more `tool_use` content blocks that include:

- `id` : A unique identifier for this particular tool use block. This will be used to match up the tool results later.
- `name` : The name of the tool being used.
- `input` : An object containing the input being passed to the tool, conforming to the tool's `input_schema` .

> **Example API response with a `tool_use` content block**

When you receive a tool use response for a client tool, you should:

1. Extract the `name` , `id` , and `input` from the `tool_use` block.
2. Run the actual tool in your codebase corresponding to that tool name, passing in the tool `input` .
3. Continue the conversation by sending a new message with the `role` of `user` , and a `content` block containing the `tool_result` type and the following information:
   - `tool_use_id` : The `id` of the tool use request this is a result for.
   - `content` : The result of the tool, as a string (e.g. `"content": "15 degrees"`), a list of nested content blocks (e.g. `"content": [{"type": "text", "text": "15 degrees"}]`), or a list of document blocks (e.g. `"content": ["type": "document", "source": {"type": "text", "media_type": "text/plain", "data": "15 degrees"}]`). These content blocks can use the `text` , `image` , or `document` types.
   - `is_error` (optional): Set to `true` if the tool execution resulted in an error.

> **Important formatting requirements:**

message and the user's tool result message.

- In the user message containing tool results, the tool_result blocks must come FIRST in the content array. Any text must come AFTER all tool results.

For example, this will cause a 400 error:

```
{"role": "user", "content": [
  {"type": "text", "text": "Here are the results:"},  // ⍰ Text before
  {"type": "tool_result", "tool_use_id": "toolu_01", ...}
]}
```

This is correct:

```
{"role": "user", "content": [
  {"type": "tool_result", "tool_use_id": "toolu_01", ...},
  {"type": "text", "text": "What should I do next?"}  // ⍰ Text after
]}
```

If you receive an error like "tool_use ids were found without tool_result blocks immediately after", check that your tool results are formatted correctly.

Example of successful tool result

Example of tool result with images

Example of empty tool result

Example of tool result with documents

After receiving the tool result, Claude will use that information to continue generating a response to the original user prompt.

Handling results from server tools

Claude executes the tool internally and incorporates the results directly into its response without requiring additional user interaction.

Unlike APIs that separate tool use or use special roles like `tool` or `function`, the Claude API integrates tools directly into the `user` and `assistant` message structure.

Messages contain arrays of `text`, `image`, `tool_use`, and `tool_result` blocks. `user` messages include client content and `tool_result`, while `assistant` messages contain AI-generated content and `tool_use`.

Handling the `max_tokens` stop reason

If Claude's response is cut off due to hitting the `max_tokens` limit, and the truncated response contains an incomplete tool use block, you'll need to retry the request with a higher `max_tokens` value to get the full tool use.

Python

```python
# Check if response was truncated during tool use
if response.stop_reason == "max_tokens":
    # Check if the last content block is an incomplete tool_use
    last_block = response.content[-1]
    if last_block.type == "tool_use":
        # Send the request with higher max_tokens
        response = client.messages.create(
            model="claude-sonnet-4-5",
            max_tokens=4096,  # Increased limit
            messages=messages,
            tools=tools
        )
```

Handling the `pause_turn` stop reason

When using server tools like web search, the API may return a `pause_turn` stop reason, indicating that the API has paused a long-running turn.

Here's how to handle the `pause_turn` stop reason:

Python

Claude Docs

```python
client = anthropic.Anthropic()

# Initial request with web search
response = client.messages.create(
    model="claude-3-7-sonnet-latest",
    max_tokens=1024,
    messages=[
        {
            "role": "user",
            "content": "Search for comprehensive information about quantum com
        }
    ],
    tools=[{
        "type": "web_search_20250305",
        "name": "web_search",
        "max_uses": 10
    }]
)

# Check if the response has pause_turn stop reason
if response.stop_reason == "pause_turn":
    # Continue the conversation with the paused content
    messages = [
        {"role": "user", "content": "Search for comprehensive information abou
        {"role": "assistant", "content": response.content}
    ]

    # Send the continuation request
    continuation = client.messages.create(
        model="claude-3-7-sonnet-latest",
        max_tokens=1024,
        messages=messages,
        tools=[{
            "type": "web_search_20250305",
            "name": "web_search",
            "max_uses": 10
        }]
    )

    print(continuation)
else:
    print(response)
```

When handling `pause_turn`:

- **Modify if needed**: You can optionally modify the content before continuing if you want to interrupt or redirect the conversation
- **Preserve tool state**: Include the same tools in the continuation request to maintain functionality

Troubleshooting errors

> **Built-in Error Handling**: Tool runner provide automatic error handling for most common scenarios. This section covers manual error handling for advanced use cases.

There are a few different types of errors that can occur when using tools with Claude:

- **Tool execution error**
- **Invalid tool name**
- **<search_quality_reflection> tags**
- **Server tool errors**
- **Parallel tool calls not working**

Claude Docs

Financial services

Government

Life sciences

Partners

Amazon Bedrock

Google Cloud's Vertex AI

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

Discord

Terms and policies

Privacy policy

Responsible disclosure policy

Terms of service: Commercial

Terms of service: Consumer

Usage policy