BUILD WITH CLAUDE

Tool use (function calling)

Claude is capable of interacting with external client-side tools and functions, allowing you to equip Claude with your own custom tools to perform a wider variety of tasks.

Learn everything you need to master tool use with Claude via our new comprehensive tool use course! Please continue to share your ideas and suggestions using this form.

Here's an example of how to provide tools to Claude using the Messages API:

```
import anthropic



client = anthropic.Anthropic()



response = client.messages.create(

    model="claude-3-5-sonnet-20240620",

    max_tokens=1024,

    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, CA",
                    }
                },
                "required": ["location"],
            },
        }
    ],
    messages=[{"role": "user", "content": "What's the weather like in San Francisco?"}],
)
print(response)
```

How tool use works

Integrate external tools with Claude in these steps:

1

Provide Claude with tools and a user prompt

Define tools with names, descriptions, and input schemas in your API request.
Include a user prompt that might require these tools, e.g., “What’s the weather in San Francisco?”

2

Claude decides to use a tool

Claude assesses if any tools can help with the user’s query.
If yes, Claude constructs a properly formatted tool use request.
The API response has a `stop_reason` of `tool_use`, signaling Claude’s intent.

3

Extract tool input, run code, and return results

On your end, extract the tool name and input from Claude’s request.
Execute the actual tool code client-side.
Continue the conversation with a new `user` message containing a `tool_result` content block.

4

Claude uses tool result to formulate a response

Claude analyzes the tool results to craft its final response to the original user prompt.

Note: Steps 3 and 4 are optional. For some workflows, Claude’s tool use request (step 2) might be all you need, without sending results back to Claude.

All tools are user-provided

It’s important to note that Claude does not have access to any built-in server-side tools. All tools must be explicitly provided by you, the user, in each API request. This gives you full control and flexibility over the tools Claude can use.

How to implement tool use

Choosing a model

Generally, use Claude 3 Opus for complex tools and ambiguous queries; it handles multiple tools better and seeks clarification when needed.

Use Haiku for straightforward tools, but note it may infer missing parameters.

Specifying tools

Tools 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.

Example Simple Tool Definition

```
{
  "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, CA"
      },
      "unit": {
        "type": "string",
        "enum": ["celsius", "fahrenheit"],
        "description": "The unit of temperature, either 'celsius' or 'fahrenheit'"
      }
    },
    "required": ["location"]
  }
}
```

This tool, named `get_weather`, expects an input object with a required `location` string and an optional `unit` string that must be either “celsius” or “fahrenheit”.

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 over examples. While you can include examples of how to use a tool in its description or in the accompanying prompt, this is less important than having a clear and comprehensive explanation of the tool's purpose and parameters. Only add examples after you've fully fleshed out the description.

Example of a good tool description

```
{
  "name": "get_stock_price",
  "description": "Retrieves the current stock price for a given ticker
symbol. The ticker symbol must be a valid symbol for a publicly traded
company on a major US stock exchange like NYSE or NASDAQ. The tool will
return the latest trade price in USD. It should be used when the user asks
about the current or most recent price of a specific stock. It will not
provide any other information about the stock or company.",
  "input_schema": {
    "type": "object",
    "properties": {
      "ticker": {
        "type": "string",
        "description": "The stock ticker symbol, e.g. AAPL for Apple Inc."
      }
    },
    "required": ["ticker"]
  }
}
```

Example poor tool description

```
{
  "name": "get_stock_price",
  "description": "Gets the stock price for a ticker.",
  "input_schema": {
    "type": "object",
    "properties": {
      "ticker": {
        "type": "string"
      }
    },
    "required": ["ticker"]
  }
}
```

The good description clearly explains what the tool does, when to use it, what data it returns, and what the `ticker` parameter means. The poor description is too brief and leaves Claude with many open questions about the tool's behavior and usage.

Controlling Claude's output

Forcing tool use

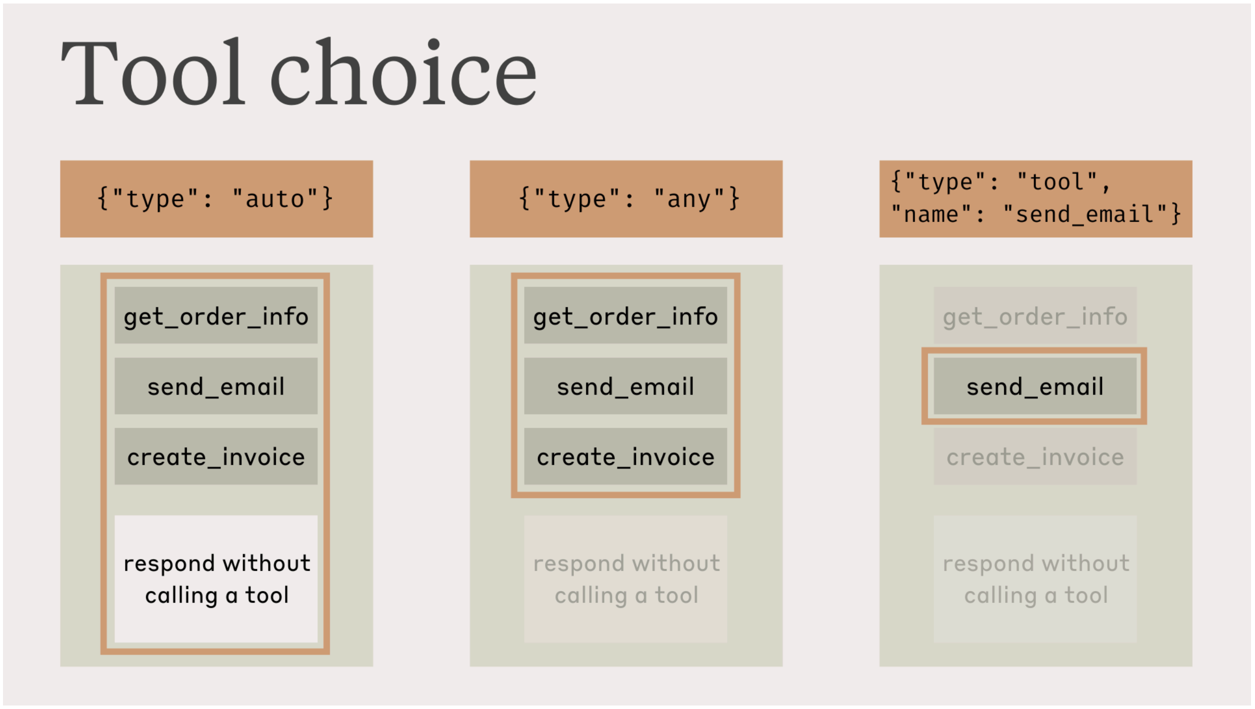
In some cases, you may want Claude to use a specific tool to answer the user's question, even if Claude thinks it can provide an answer without using a tool. You can do this by specifying the tool in the `tool_choice` field like so:

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

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

- `auto` allows Claude to decide whether to call any provided tools or not. This is the default value.
- `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.

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 chain-of-thought `text` content block before `tool_use` content blocks, even if explicitly asked to do so.

Our testing has shown that this should not reduce performance. If you would like to keep chain-of-thought (particularly with Opus) 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.`

JSON output

Tools do not necessarily need to be client-side 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 examples for a full working example.

Chain of thought

When using tools, Claude will often show its "chain of thought", i.e. the step-by-step reasoning it uses to break down the problem and decide which tools to use. The Claude 3 Opus model will do this if `tool_choice` is set to `auto` (this is the default value, see Forcing tool use), and Sonnet and Haiku can be prompted into doing it.

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:

```
{
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "<thinking>To answer this question, I will: 1. Use the get_weather tool to get the current weather in San Francisco. 2. Use the get_time tool to get the current time in the America/Los_Angeles timezone, which covers San Francisco, CA.</thinking>"
    },
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": {"location": "San Francisco, CA"}
    }
  ]
}
```

This chain of thought gives insight into Claude's reasoning process and can help you debug unexpected behavior.

With the Claude 3 Sonnet model, chain of thought is less common by default, but you can prompt Claude to show its reasoning by adding something like `"Before answering, explain your reasoning step-by-step in tags."` to the user message or system prompt.

It's important to note that while the `<thinking>` tags are a common convention Claude uses to denote its chain of thought, the exact format (such as what this XML tag is named) may change over time. Your code should treat the chain of thought like any other assistant-generated text, and not rely on the presence or specific formatting of the `<thinking>` tags.

Handling tool use and tool result content blocks

When Claude decides to use one of the tools you've provided, it will return a response with a `stop_reason` of `tool_use` and one or more `tool_use` content blocks in the API response 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

```
{
  "id": "msg_01Aq9w938a90dw8q",
  "model": "claude-3-5-sonnet-20240620",
  "stop_reason": "tool_use",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "<thinking>I need to use the get_weather, and the user wants SF, which is likely San
Francisco, CA.</thinking>"
    },
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": {"location": "San Francisco, CA", "unit": "celsius"}
    }
  ]
}
```

When you receive a tool use response, 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. [optional] 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"`) or list of nested content blocks (e.g. `"content": [{"type": "text", "text": "15 degrees"}]`). These content blocks can use the `text` or `image` types.
`is_error` (optional): Set to `true` if the tool execution resulted in an error.

Example of a successful tool result

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "15 degrees"
    }
  ]
}
```

Example of a tool result with images

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": [
        {"type": "text", "text": "15 degrees"},
        {
          "type": "image",
          "source": {
            "type": "base64",
            "media_type": "image/jpeg",
            "data": "/9j/4AAQSkZJRg...",
          }
        }
      ]
    }
  ]
}
```

Example of empty tool result

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
    }
  ]
}
```

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

Differences from other APIs

Unlike APIs that separate tool use or use special roles like `tool` or `function`, Anthropic's 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-side content and `tool_result`, while `assistant` messages contain AI-generated content and `tool_use`.

Troubleshooting errors

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

Tool execution error

If the tool itself throws an error during execution (e.g. a network error when fetching weather data), you can return the error message in the `content` along with `"is_error": true`:

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "ConnectionError: the weather service API is not available (HTTP 500)",
      "is_error": true
    }
  ]
}
```

Claude will then incorporate this error into its response to the user, e.g. “I’m sorry, I was unable to retrieve the current weather because the weather service API is not available. Please try again later.”

Max Tokens Exceeded
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.

Invalid Tool Name

If Claude’s attempted use of a tool is invalid (e.g. missing required parameters), it usually means that the there wasn’t enough information for Claude to use the tool correctly. Your best bet during development is to try the request again with more-detailed `description` values in your tool definitions.

However, you can also continue the conversation forward with a `tool_result` that indicates the error, and Claude will try to use the tool again with the missing information filled in:

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "Error: Missing required 'location' parameter",
      "is_error": true
    }
  ]
}
```

If a tool request is invalid or missing parameters, Claude will retry 2-3 times with corrections before apologizing to the user.

<search-quality-reflection>tags
To prevent Claude from reflecting on search quality with <search_quality_reflection> tags, add “Do not reflect on the quality of the returned search results in your response” to your prompt.

Tool use examples

Here are a few code examples demonstrating various tool use patterns and techniques. For brevity's sake, the tools are simple tools, and the tool descriptions are shorter than would be ideal to ensure best performance.

Single Tool Example

```
import anthropic
client = anthropic.Anthropic()

response = client.messages.create(
    model="claude-3-5-sonnet-20240620",
    max_tokens=1024,
    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, CA"
                    },
                    "unit": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"],
                        "description": "The unit of temperature, either \"celsius\" or \"fahrenheit\""
                    }
                },
                "required": ["location"]
            }
        }
    ],
    messages=[{"role": "user", "content": "What is the weather like in San Francisco?"}]
)

print(response)
```

Claude will return a response similar to:

```
{
  "id": "msg_01Aq9w938a90dw8q",
  "model": "claude-3-5-sonnet-20240620",
  "stop_reason": "tool_use",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "<thinking>I need to call the get_weather function, and the user wants SF, which is
likely San Francisco, CA.</thinking>"
    },
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": {"location": "San Francisco, CA", "unit": "celsius"}
    }
  ]
}
```

You would then need to execute the `get_weather` function with the provided input, and return the result in a new `user` message:

```
response = client.messages.create(
    model="claude-3-5-sonnet-20240620",
    max_tokens=1024,
    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, CA"
                    },
                    "unit": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"],
                        "description": "The unit of temperature, either 'celsius' or 'fahrenheit'"
                    }
                },
                "required": ["location"]
            }
        }
    ],
    messages=[
        {
            "role": "user",
            "content": "What's the weather like in San Francisco?"
        },
        {
            "role": "assistant",
            "content": [
                {
                    "type": "text",
                    "text": "<thinking>I need to use get_weather, and the user wants SF, which is likely
San Francisco, CA.</thinking>"
                },
                {
                    "type": "tool_use",
                    "id": "toolu_01A09q90qw90lq917835lq9",
                    "name": "get_weather",
                    "input": {"location": "San Francisco, CA", "unit": "celsius"}
                }
            ]
        },
        {
            "role": "user",
            "content": [
                {
                    "type": "tool_result",
                    "tool_use_id": "toolu_01A09q90qw90lq917835lq9", # from the API response
                    "content": "65 degrees" # from running your tool
                }
            ]
        }
    ]
```

```
)

print(response)
```

This will print Claude’s final response, incorporating the weather data:

```
{
  "id": "msg_01Aq9w938a90dw8q",
  "model": "claude-3-5-sonnet-20240620",
  "stop_reason": "stop_sequence",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "The current weather in San Francisco is 15 degrees Celsius (59 degrees Fahrenheit).
It's a cool day in the city by the bay!"
    }
  ]
}
```

Multiple tool example

You can provide Claude with multiple tools to choose from in a single request. Here’s an example with both a `get_weather` and a `get_time` tool, along with a user query that asks for both.

```
import anthropic
client = anthropic.Anthropic()

response = client.messages.create(
    model="claude-3-5-sonnet-20240620",
    max_tokens=1024,
    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, CA"
                    },
                    "unit": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"],
                        "description": "The unit of temperature, either 'celsius' or 'fahrenheit'"
                    }
                },
                "required": ["location"]
            }
        },
        {
            "name": "get_time",
            "description": "Get the current time in a given time zone",
            "input_schema": {
                "type": "object",
                "properties": {
                    "timezone": {
                        "type": "string",
                        "description": "The IANA time zone name, e.g. America/Los_Angeles"
                    }
                },
                "required": ["timezone"]
            }
        }
    ],
    messages=[
        {
            "role": "user",
            "content": "What is the weather like right now in New York? Also what time is it there?"
        }
    ]
)
print(response)
```

In this case, Claude will most likely try to use two separate tools, one at a time — `get_weather` and then `get_time` — in order to fully answer the user's question. However, it will also occasionally output two `tool_use` blocks at once, particularly if they are not dependent on each other. You would need to execute each tool and return their results in separate `tool_result` blocks within a single `user` message.

Missing information

If the user’s prompt doesn’t include enough information to fill all the required parameters for a tool, Claude 3 Opus is much more likely to recognize that a parameter is missing and ask for it. Claude 3 Sonnet may ask, especially when prompted to think before outputting a tool request. But it may also do its best to infer a reasonable value.

For example, using the `get_weather` tool above, if you ask Claude “What’s the weather?” without specifying a location, Claude, particularly Claude 3 Sonnet, may make a guess about tools inputs:

```
{
  "type": "tool_use",
  "id": "toolu_01A09q90qw90lq917835lq9",
  "name": "get_weather",
  "input": {"location": "New York, NY", "unit": "fahrenheit"}
}
```

This behavior is not guaranteed, especially for more ambiguous prompts and for models less intelligent than Claude 3 Opus. If Claude 3 Opus doesn’t have enough context to fill in the required parameters, it is far more likely respond with a clarifying question instead of making a tool call.

Sequential Tools

Some tasks may require calling multiple tools in sequence, using the output of one tool as the input to another. In such a case, Claude will call one tool at a time. If prompted to call the tools all at once, Claude is likely to guess parameters for tools further downstream if they are dependent on tool results for tools further upstream.

Here’s an example of using a `get_location` tool to get the user’s location, then passing that location to the `get_weather` tool:

```
response = client.messages.create(
    model="claude-3-5-sonnet-20240620",
    max_tokens=1024,
    tools=[
        {
            "name": "get_location",
            "description": "Get the current user location based on their IP address. This tool has no parameters or arguments.",
            "input_schema": {
                "type": "object",
                "properties": {}
            }
        },
        {
            "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, CA"
                    },
                    "unit": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"],
                        "description": "The unit of temperature, either 'celsius' or 'fahrenheit'"
                    }
                },
                "required": ["location"]
            }
        }
    ],
    messages=[{
                        "role": "user",
            "content": "What's the weather like where I am?"
    }]
)
```

In this case, Claude would first call the `get_location` tool to get the user's location. After you return the location in a `tool_result`, Claude would then call `get_weather` with that location to get the final answer.

The full conversation might look like:

Role	**Content**
User	What’s the weather like where I am?
Assista nt	<thinking>To answer this, I first need to determine the user’s location using the get_location tool. Then I can pass that location to the get_weather tool to find the current weather there.</thinking>[Tool use for get_location]
User	[Tool result for get_location with matching id and result of San Francisco, CA]
Assista nt	[Tool use for get_weather with the following input]{ “location”: “San Francisco, CA”, “unit”: “fahrenheit” }
User	[Tool result for get_weather with matching id and result of “59°F (15°C), mostly cloudy”]
Assista nt	Based on your current location in San Francisco, CA, the weather right now is 59°F (15°C) and mostly cloudy. It’s a fairly cool and overcast day in the city. You may want to bring a light jacket if you’re heading outside.

This example demonstrates how Claude can chain together multiple tool calls to answer a question that requires gathering data from different sources. The key steps are:

1. Claude first realizes it needs the user's location to answer the weather question, so it calls the `get_location` tool.
2. The user (i.e. the client code) executes the actual `get_location` function and returns the result "San Francisco, CA" in a `tool_result` block.
3. With the location now known, Claude proceeds to call the `get_weather` tool, passing in "San Francisco, CA" as the `location` parameter (as well as a guessed `unit` parameter, as `unit` is not a required parameter).
4. The user again executes the actual `get_weather` function with the provided arguments and returns the weather data in another `tool_result` block.
5. Finally, Claude incorporates the weather data into a natural language response to the original question.

Chain of thought tool use
By default, Claude 3 Opus is prompted to think before it answers a tool use query to best determine whether a tool is necessary, which tool to use, and the appropriate parameters. Claude 3 Sonnet and Claude 3 Haiku are prompted to try to use tools as much as possible and are more likely to call an unnecessary tool or infer missing parameters. To prompt Sonnet or Haiku to better assess the user query before making tool calls, the following prompt can be used:

Chain of thought prompt

```
Answer the user's request using relevant tools (if they are available).
Before calling a tool, do some analysis within \<thinking>\</thinking>
tags. First, think about which of the provided tools is the relevant tool
to answer the user's request. Second, go through each of the required
parameters of the relevant tool and determine if the user has directly
provided or given enough information to infer a value. When deciding if
the parameter can be inferred, carefully consider all the context to see
if it supports a specific value. If all of the required parameters are
present or can be reasonably inferred, close the thinking tag and proceed
with the tool call. BUT, if one of the values for a required parameter is
missing, DO NOT invoke the function (not even with fillers for the missing
params) and instead, ask the user to provide the missing parameters. DO
NOT ask for more information on optional parameters if it is not provided.
```

JSON Mode

You can use tools to get Claude produce JSON output that follows a schema, even if you don't have any intention of running that output through a tool or function.

When using tools in this way:

You usually want to provide a single tool
You should set `tool_choice` (see Forcing tool use) to instruct the model to explicitly use that tool
Remember that the model will pass the `input` to the tool, so the name of the tool and description should be from the model's perspective.

The following uses a `record_summary` tool to describe an image following a particular format.

```
import base64
import anthropic
import httpx

image_url =
"https://upload.wikimedia.org/wikipedia/commons/a/a7/Camponotus_flavomarginatus_ant.jpg"
image_media_type = "image/jpeg"
image_data = base64.b64encode(httpx.get(image_url).content).decode("utf-8")

message = anthropic.Anthropic().messages.create(
    model="claude-3-5-sonnet-20240620",
    max_tokens=1024,
    tools=[
        {
            "name": "record_summary",
            "description": "Record summary of an image using well-structured JSON.",
            "input_schema": {
                "type": "object",
                "properties": {
                    "key_colors": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "r": {
                                    "type": "number",
                                    "description": "red value [0.0, 1.0]",
                                },
                                "g": {
                                    "type": "number",
                                    "description": "green value [0.0, 1.0]",
                                },
                                "b": {
                                    "type": "number",
                                    "description": "blue value [0.0, 1.0]",
                                },
                                "name": {
                                    "type": "string",
                                    "description": "Human-readable color name in snake_case, e.g.
\"olive_green\" or \"turquoise\""
                                },
                            },
                            "required": ["r", "g", "b", "name"],
                        },
                        "description": "Key colors in the image. Limit to less then four.",
                    },
                    "description": {
                        "type": "string",
                        "description": "Image description. One to two sentences max.",
                    },
                    "estimated_year": {
                        "type": "integer",
                        "description": "Estimated year that the images was taken, if it a photo. Only set
this if the image appears to be non-fictional. Rough estimates are okay!",
                    },
                },
                "required": ["key_colors", "description"],
            },
        }
    ],
    tool_choice={"type": "tool", "name": "record_summary"},
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
                        "media_type": image_media_type,
                        "data": image_data,
                    },
            },
            {"type": "text", "text": "Describe this image."},
          ],
        }
    ],
)
print(message)
```

Pricing

Tool use requests are priced the same as any other Claude API request, based on the total number of input tokens sent to the model (including in the `tools` parameter) and the number of output tokens generated.”

The additional tokens from tool use come from:

The `tools` parameter in API requests (tool names, descriptions, and schemas)
`tool_use` content blocks in API requests and responses
`tool_result` content blocks in API requests

When you use `tools`, we also automatically include a special system prompt for the model which enables tool use. The number of tool use tokens required for each model are listed below (excluding the additional tokens listed above):

Model	Tool choice	Tool use system prompt token count
Claude 3.5 Sonnet	`auto`	294 tokens
	`any`, `tool`	261 tokens

Claude 3 Opus	`auto`	530 tokens
	`any`, `tool`	281 tokens
Claude 3 Sonnet	`auto`	159 tokens
	`any`, `tool`	235 tokens
Claude 3 Haiku	`auto`	264 tokens
	`any`, `tool`	340 tokens

These token counts are added to your normal input and output tokens to calculate the total cost of a request. Refer to our models overview table for current per-model prices.

When you send a tool use prompt, just like any other API request, the response will output both input and output token counts as part of the reported `usage` metrics.