Tool Use in LLMs

Running example: **TerpRemind** · Code: `claude-tool-use/` (demos `01` – `06`)

Learning Objectives

1. Explain the **four-step tool-use protocol** and **who** runs code at each step
2. Write a **tool function** that validates input, and its **JSON Schema**
3. Answer `tool_use` blocks with matching `tool_result` blocks, including errors
4. Implement the **tool loop** driven by `stop_reason`
5. **Add a tool** in four steps, and explain how Claude **chains** tools
6. **Stream** tool calls and decide when to use **eager input streaming**
7. Distinguish **client** tools (yours, text editor) from **server** tools (web search)

The Model Hits a Wall

Ask Claude, with no tools, *"What's the weather in College Park, Maryland?"*

```
I'm sorry, but I don't have access to up-to-date weather information.
```

An LLM is a **closed-book exam taker**: no clock, no database, no files. It only produces text.

> **Tool use:** the model emits a *structured request* — `get_weather(location="College Park, MD")` — and **our code** runs it. **The model never runs anything. It only asks.**

An Analogy: The Manager and the Assistant

A manager locked in an office can only pass notes under the door to an assistant with a phone.

- **Manager (Claude)** decides what info is missing
- Passes a note: "Get College Park's weather" — a **tool-use request**
- **Assistant (your app)** makes the call, returns the answer — a **tool result**
- Manager writes the final reply

> Manager can only ask for what the assistant offers → **describe** your tools.

> Assistant decides whether to act → **validation and safety** live here.

Where Tool Use Shows Up

- **Fresh data** — weather, time, search results
- **Your systems** — databases, your own **REST API**
- **Actions** — set a reminder, send a message, edit a file
- **Reliable computation** — date math, exact arithmetic
- **Coding agents** — Claude Code is a loop calling "read," "edit," "run"

> The Messages API is still **stateless** — resend the whole conversation every call.

Running Example: TerpRemind

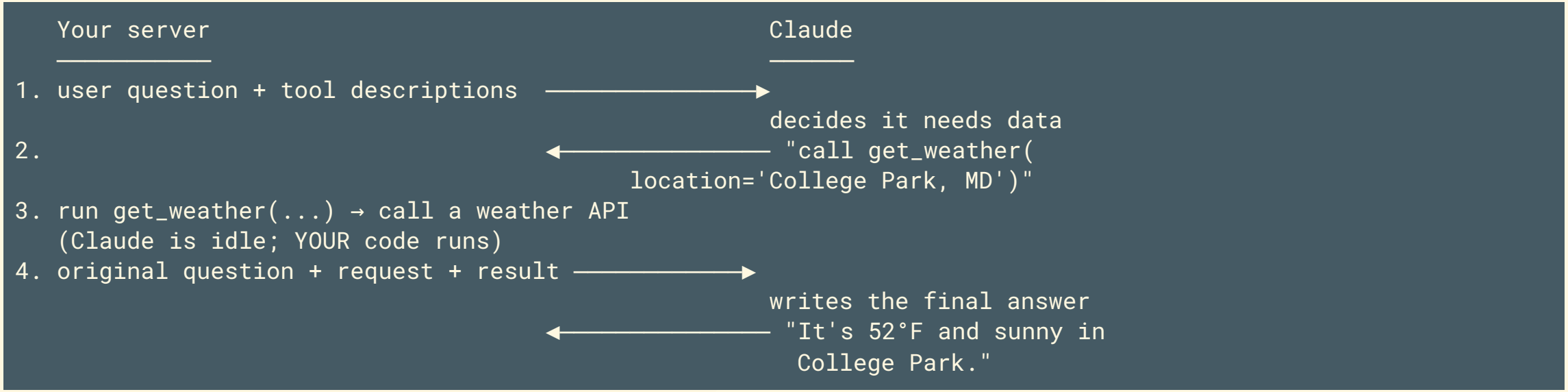
```
Set a reminder for my doctor's appointment. It's a week from Thursday.
```

Goal: Claude **actually sets the reminder**.

Gap	Tool
Doesn't know the exact time	`get_current_datetime`
Unreliable date arithmetic	`add_duration_to_datetime`
Can't act	`set_reminder`

> **Model has a limitation? Add a tool — don't prompt around it.**

The Tool-Use Protocol: Four Steps



Claude fills in a form; you run it; you tell Claude what happened.

Protocol: Common Misconceptions

- **"Claude runs my function."** Never — it emits JSON; **your code** runs it. (Except server tools.)
- **"Claude remembers the result."** Stateless: resend question + request + result.
- **"Claude always uses a described tool."** Claude decides.

> **Treat every tool argument as untrusted input** — like an HTTP request body.

What a Tool Function Is

> A **plain Python function** your app runs when Claude asks. Nothing AI-specific.

Best practices

- **Descriptive names** — Claude reads them
- **Validate inputs**; never trust arguments
- **Raise meaningful errors** — Claude sees the text and can retry
- **Raise, don't return** an error string — the loop marks it `is_error: True`

Our First Tool: `get_current_datetime`

```python
from datetime import datetime

def get_current_datetime(date_format="%Y-%m-%d %H:%M:%S"):
    if not date_format:
        raise ValueError("date_format cannot be empty")
    return datetime.now().strftime(date_format)
```

```python
get_current_datetime()          # '2026-11-01 14:30:25'
get_current_datetime("%H:%M")   # '14:30'
get_current_datetime("")        # ValueError: date_format cannot be empty
```

Validate, and fail with a sentence Claude can act on.

The Other Two TerpRemind Tools

```
def set_reminder(content, timestamp):
    print(f"----\nSetting the following reminder "
          f"for {timestamp}:\n{content}\n----")
```

```
def add_duration_to_datetime(
    datetime_str, duration=0, unit="days",
    input_format="%Y-%m-%d"
):
    date = datetime.strptime(datetime_str, input_format)

    if unit == "seconds":
        new_date = date + timedelta(seconds=duration)
    elif unit == "minutes":
        new_date = date + timedelta(minutes=duration)
    # ... hours, days, weeks ...
    elif unit == "months":
        # month rollover + clamp the day to the
        # month's length (leap years included)
        ...
    elif unit == "years":
        new_date = date.replace(year=date.year + duration)
    else:
        raise ValueError(f"Unsupported time unit: {unit}")

    return new_date.strftime("%A, %B %d, %Y %I:%M:%S %p")
```

`set_reminder` just prints — a stand-in.

Returns e.g. `"Monday, June 27, 2050 12:00:00 AM"`.

> The **weekday** is deliberate: "a week from Thursday" needs it.

What a Tool Schema Is

A JSON object that tells Claude a tool exists:

Field	Purpose
`name`	identifier Claude uses; match your function
`description`	what it does, **when to use it**, what it returns
`input_schema`	JSON Schema for the arguments

> **The schema is Claude's only documentation** — it never sees your code. Aim for 3–4 sentences; describe every argument.

Worked Example: `get_current_datetime_schema`

```python
from anthropic.types import ToolParam

get_current_datetime_schema = ToolParam({
    "name": "get_current_datetime",
    "description": "Returns the current date and time formatted according to the specified format "
                   "string. ... Use this tool when you need to know the current date and time, such "
                   "as for timestamping records, calculating time differences, or displaying the "
                   "current time to users. The default format returns ... (YYYY-MM-DD HH:MM:SS).",
    "input_schema": {
        "type": "object",
        "properties": {
            "date_format": {
                "type": "string",
                "description": "A string specifying the format of the returned datetime. Uses "
                               "Python's strftime format codes. For example, '%Y-%m-%d' returns "
                               "just the date ... '%B %d, %Y' returns a date like 'May 07, 2025'.",
                "default": "%Y-%m-%d %H:%M:%S",
            }
        },
        "required": [],
    },
})
```

`"type": "object"` → args arrive as a dict, so `get_current_datetime(**input)` works.

Writing Schemas in Practice

The easy way: give Claude your function + the tool-use docs:

> *"Write a valid JSON schema spec for tool calling for this function. Follow the best practices in the attached documentation."*

Then **review it** — you ship it.

Misconceptions

- **"Short descriptions are cleaner."** Specific ones pick tools better.
- **"The schema validates for me."** Still validate. (`"strict": true` guarantees valid args.)

Making a Tool-Enabled Request

```
messages = [{"role": "user",
             "content": "What is the exact time, formatted as HH:MM:SS?"}]

response = client.messages.create(
    model=model,
    max_tokens=1000,
    messages=messages,
    tools=[get_current_datetime_schema],
)
```

> The reply is a **multi-block message**: **text blocks** for humans + `tool_use` **blocks** for your code.

What Comes Back

```
Message(
  stop_reason='tool_use',
  content=[
    TextBlock(type='text', text='Let me find that information for you.'),
    ToolUseBlock(type='tool_use', id='toolu_01A...', name='get_current_datetime',
                 input={'date_format': '%H:%M:%S'}),
  ],
)
```

Field	Meaning
`id`	unique ID — **your result must quote it**
`name`	which tool to run
`input`	arguments, **already a dict**

`stop_reason = "tool_use"` → "waiting for a tool result."

Keep Every Block in History

```
messages.append({"role": "assistant", "content": response.content})
```

> Append only the text → the result's `tool_use_id` points nowhere → **API error**.

Misconceptions

- **"The tool call is always `content[1]`."** Filter by `block.type`.
- **"`input` is a JSON string."** It's already a `dict`.

Updating the Helper Functions

```python
from anthropic.types import Message

def add_user_message(messages, message):
    messages.append({
        "role": "user",
        "content": message.content
            if isinstance(message, Message)
            else message,
    })

def add_assistant_message(messages, message):
    messages.append({
        "role": "assistant",
        "content": message.content
            if isinstance(message, Message)
            else message,
    })

def text_from_message(message):
    return "\n".join(b.text for b in message.content
                     if b.type == "text")
```

```python
def chat(messages, system=None, tools=None):
    params = {
        "model": model,
        "max_tokens": 1000,
        "messages": messages,
    }
    if tools:
        params["tools"] = tools
    if system:
        params["system"] = system

    message = client.messages.create(**params)
    return message   # the whole Message,
                     # not content[0].text
```

Helpers accept a string, blocks, or a `Message`. `content[0]` may have no `.text`.

Running the Requested Tool

```
tool_use.input                          # {'date_format': '%H:%M:%S'}
get_current_datetime(**tool_use.input)  # '15:04:22'
```

> The result goes back as a `tool_result` **block** in a **user** message.

Field	Meaning
`tool_use_id`	must equal the `tool_use` block's `id`
`content`	tool output, as a string
`is_error`	`True` if the tool failed

Why IDs Matter: Several Calls at Once

"What's 10 + 10 and 30 + 30?" → two `tool_use` blocks:

```
assistant:  ToolUse(id="toolu_A", name="add", input={"a": 10, "b": 10})
            ToolUse(id="toolu_B", name="add", input={"a": 30, "b": 30})

user:       ToolResult(tool_use_id="toolu_A", content="20")
            ToolResult(tool_use_id="toolu_B", content="60")
```

One `tool_result` per `tool_use`, all in one user message. IDs make order irrelevant.

- Tool failed? **Still send a result**, with `is_error: True`.

The Full Round Trip by Hand DEMO 1

```python
# demos/01_single_tool_by_hand.py
messages = []
add_user_message(messages,
  "What is the exact time, formatted as HH:MM:SS?")
tools = [get_current_datetime_schema]
# Step 1: question + schema
response = chat(messages, tools=tools)
# Step 2: keep the WHOLE assistant message
add_assistant_message(messages, response)
# Step 3: run the tool
tool_use = next(b for b in response.content
                if b.type == "tool_use")
output = get_current_datetime(**tool_use.input)
# Step 4: send the result back, matched by ID
add_user_message(messages, [{
    "type": "tool_result",
    "tool_use_id": tool_use.id,
    "content": json.dumps(output),
    "is_error": False,
}])
# Step 5: ask again -- STILL passing the schema
final = chat(messages, tools=tools)
print(text_from_message(final))
# "The current time is 15:04:22."
```

History after step 4:

```
[0] user       "What is the exact time, ..."
[1] assistant  [Text "Let me find that...",
                ToolUse id=toolu_01A
                  get_current_datetime {...}]
[2] user       [ToolResult tool_use_id=toolu_01A
                content='"15:04:22"']
```

Send the same tool list on every request — history mentions the tool.

The Tool Loop and `stop_reason`

"What day is 103 days from today?" needs two calls in **separate turns → a loop**.

> Call Claude → append reply → stop unless `stop_reason == "tool_use"` → run tools, append results → repeat.

`stop_reason`	Action
`tool_use`	run tools, loop
`end_turn`	done
`max_tokens`	cut off — **don't run** the tool
`pause_turn`	server tool paused; resend
`refusal`	don't run tools

`run_tools`: Answer Every Request

```
def run_tool(tool_name, tool_input):
    if tool_name == "get_current_datetime":
        return get_current_datetime(**tool_input)


def run_tools(message):
    tool_requests = [b for b in message.content
                     if b.type == "tool_use"]
    tool_result_blocks = []

    for tool_request in tool_requests:
        try:
            tool_output = run_tool(tool_request.name,
                                   tool_request.input)
            block = {"type": "tool_result",
                     "tool_use_id": tool_request.id,
                     "content": json.dumps(tool_output),
                     "is_error": False}
        except Exception as e:
            block = {"type": "tool_result",
                     "tool_use_id": tool_request.id,
                     "content": f"Error: {e}",
                     "is_error": True}
        tool_result_blocks.append(block)

    return tool_result_blocks
```

- `run_tool` is a **router** — new tools never touch the loop
- `json.dumps` serializes any result
- `except` sends errors **back to Claude**, so it can retry

The Loop Itself

```python
# simple version
def run_conversation(messages):
    while True:
        response = chat(messages,
            tools=[get_current_datetime_schema])
        add_assistant_message(messages, response)
        print(text_from_message(response))
        if response.stop_reason != "tool_use":
            break
        add_user_message(messages,
                         run_tools(response))
    return messages
```

```python
# tooluse/loop.py -- bounded
def run_conversation(messages, tools, registry,
                     max_turns=10, verbose=True,
                     **chat_kwargs):
    for _ in range(max_turns):
        response = chat(messages, tools=tools,
                        **chat_kwargs)
        add_assistant_message(messages, response)
        if verbose and (t := text_from_message(response)):
            print(t)
        if response.stop_reason != "tool_use":
            return messages
        add_user_message(messages,
                         run_tools(response, registry))
    raise RuntimeError(
        f"Gave up after {max_turns} turns of tool use")
```

Cap the iterations — `while True` can burn money forever.

Tracing It DEMO 2

"Current time in HH:MM format? Also in SS format?"

```
[0] user       'What is the current time in HH:MM format? Also, ... SS format?'
[1] assistant  [Text 'Let me fetch that for you.',
                ToolUse id=toolu_013M  get_current_datetime {'date_format': '%H:%M'}]
[2] user       [ToolResult tool_use_id=toolu_013M  content='"11:16"']
[3] assistant  [ToolUse id=toolu_01Dv  get_current_datetime {'date_format': '%S'}]
[4] user       [ToolResult tool_use_id=toolu_01Dv  content='"30"']
[5] assistant  [Text 'The current time in HH:MM format is 11:16. ...']   ← end_turn
```

- Three iterations; history grows by **user/assistant pairs**
- Claude could also put both calls in [1] — the loop handles either

The Four-Step Recipe

Adding a tool is **mechanical**:

1. Write the function
2. Write its schema
3. Add schema to `tools`
4. Add a case in `run_tool`

```python
# Step 3
response = chat(
    messages,
    tools=[
        get_current_datetime_schema,
        add_duration_to_datetime_schema,
        set_reminder_schema,
    ],
)
```

```python
# Step 4
def run_tool(tool_name, tool_input):
    if tool_name == "get_current_datetime":
        return get_current_datetime(**tool_input)
    elif tool_name == "add_duration_to_datetime":
        return add_duration_to_datetime(**tool_input)
    elif tool_name == "set_reminder":
        return set_reminder(**tool_input)
```

TerpRemind in Action DEMO 3

```
add_user_message(messages, "Set a reminder for my doctors appointment. "
                           "Its 177 days after Jan 1st, 2050.")
run_conversation(messages)
```

```
I'll set a reminder for your doctor's appointment that's 177 days after January 1st, 2050.
Let me calculate that date first.
Now I'll set the reminder for your doctor's appointment on June 27, 2050:
----
Setting the following reminder for 2050-06-27T00:00:00:
Doctor's appointment
----
Your reminder has been set successfully! You will be reminded of your doctor's appointment
on Monday, June 27, 2050.
```

Trace: How Claude Chained the Tools

#	Role	Content
1	assistant	`add_duration_to_datetime('2050-01-01', 177, 'days')`
2	user	`'"Monday, June 27, 2050 12:00:00 AM"'`
3	assistant	`set_reminder("Doctor's appointment", '2050-06-27T00:00:00')`
4	user	`'null'`
5	assistant	Your reminder has been set successfully!

On its own, Claude:

- **Skipped** `get_current_datetime` — not needed
- **Reformatted** dates between tools

> That's **why descriptions spell out formats**.

A Registry Instead of `if` / `elif`

```
REGISTRY = {
    "get_current_datetime": get_current_datetime,
    "add_duration_to_datetime":
        add_duration_to_datetime,
    "set_reminder": set_reminder,
}
SCHEMAS = [get_current_datetime_schema,
           add_duration_to_datetime_schema,
           set_reminder_schema]


def run_tool(registry, tool_name, tool_input):
    if tool_name not in registry:
        raise ValueError(f"Unknown tool: {tool_name}")
    return registry[tool_name](**tool_input)
```

- `if` / `elif` **silently returns** `None` for unknown names
- A dict is shorter and **fails loudly**
- Return useful text: `"Reminder #1 set for ..."` beats `null`

Controlling Tool Selection: `tool_choice`

Value	Behavior
`{"type": "auto"}`	Claude decides (default)
`{"type": "none"}`	no tools
`{"type": "any"}`	must call some tool
`{"type": "tool", "name": "..."}`	must call this tool

> Opus 5.5, Sonnet 5.5, and Fable 5.1 reject `any` / `tool` with a **400** — use `auto` and ask in the prompt.

More tools ≠ better: each schema costs tokens and overlaps confuse selection.

Tool Calls on a Stream

```
content_block_start  (type "tool_use",
                      name "save_article")
input_json   partial_json='{"abstract": "Th...'
input_json   partial_json='", "meta": {"wor...'
...
content_block_stop
```

- `partial_json` — new chunk of text
- `snapshot` — args so far, **parsed**

Example: `save_article(abstract, meta)` with nested `meta = {word_count, review}`.

```python
with chat_stream(messages, tools=tools) as stream:
    for chunk in stream:
        if chunk.type == "text":
            print(chunk.text, end="")
        if chunk.type == "content_block_start":
            if chunk.content_block.type == "tool_use":
                print(f'\n>>> Tool Call: '
                      f'"{chunk.content_block.name}"')
        if (chunk.type == "input_json"
                and chunk.partial_json):
            print(chunk.partial_json, end="")
        if chunk.type == "content_block_stop":
            print("\n")
    response = stream.get_final_message()
```

Why Arguments Arrive in Bursts DEMO 4

The API **buffers each top-level key** to validate it, then sends it all at once.

```
Claude generates:                    You receive:

{"abstract":                         ┐
    "This pa|per pres|ents ...",     │ buffered
                                     ┘ ◀── burst: whole abstract
 "meta": {                           ┐
    "word_count": 847,               │
    "review": "...8 sentences..."    │ buffered
 }                                   ┘ ◀── burst: whole meta object
}
```

`word_count` is generated early but arrives **last**.

Fine-Grained (Eager) Input Streaming

> Turns buffering **off**: chunks arrive immediately. **Cost: no server-side JSON validation.**

```
schema = {"name": "save_article", "description": "Saves a scholarly journal article",
          "eager_input_streaming": True,     # per tool, not per request
          "input_schema": {...}}
with client.messages.stream(model=model, max_tokens=64000,
                            messages=messages, tools=[schema]) as stream:
    for event in stream:
        if event.type == "input_json" and event.partial_json:
            print(event.partial_json, end="", flush=True)
    response = stream.get_final_message()
```

Use for large arguments or live display — **if** you validate input. Same total speed.

What Invalid Output Looks Like

Ask Claude for `"word_count": undefined` (not legal JSON). Streaming **off**:

```
>>> Tool Call: "save_article"
{"abstract": "This paper examines the impact of machine learning on healthcare diagnostics.",
 "meta": "{\n  \"word_count\": undefined,\n  \"review\": \"This study presents ...\"\n}"}
```

- `meta` became a **string** — valid JSON, **wrong schema**
- Streaming **on**: raw `undefined` reaches your code

> Either way: **validate tool input before running the tool.**

Validating the Input Yourself

```python
# tooluse/streaming.py
def validate_article(tool_input):
    if not isinstance(tool_input, dict):
        return "input is not an object"
    if not isinstance(tool_input.get("abstract"), str):
        return "abstract must be a string"
    meta = tool_input.get("meta")
    if not isinstance(meta, dict):
        return "meta must be an object"    # catches the
                                           # stringified meta
    if not isinstance(meta.get("word_count"), int):
        return "meta.word_count must be an integer"
    if not isinstance(meta.get("review"), str):
        return "meta.review must be a string"
    return None
```

- Invalid → **don't run**; return `is_error: True`
- Skip tools on `max_tokens` — truncated calls look valid

> `snapshot` is already parsed — `json.loads` on it fails.
> **Validate the final input.**

Built-in Tools: The Text Editor

Schema built in, implementation yours. Declare a two-field stub:

```
TEXT_EDITOR_SCHEMA = {"type": "text_editor_20250728",
                      "name": "str_replace_based_edit_tool"}
```

command	Inputs	Action
`view`	`path`, `view_range?`	show file or directory
`create`	`path`, `file_text`	create/overwrite
`str_replace`	`path`, `old_str`, `new_str`	replace **exactly one** match
`insert`	`path`, `insert_line`, `insert_text`	insert after line N

The Implementation

```
def str_replace(self, file_path, old_str, new_str):
    abs_path = self._validate_path(file_path)
    if not os.path.exists(abs_path):
        raise FileNotFoundError("File not found")

    with open(abs_path, "r", encoding="utf-8") as f:
        content = f.read()

    match_count = content.count(old_str)
    if match_count == 0:
        raise ValueError("No match found for replacement. "
                         "Please check your text and try again.")
    elif match_count > 1:
        raise ValueError(
            f"Found {match_count} matches for replacement "
            "text. Please provide more context to make a "
            "unique match.")

    self._backup_file(abs_path)   # backup before modifying
    new_content = content.replace(old_str, new_str)
    with open(abs_path, "w", encoding="utf-8") as f:
        f.write(new_content)
    return "Successfully replaced text at exactly one location."
```

Validate → check → back up → write → report.

```
def run_tool(tool_name, tool_input):
  if tool_name == "str_replace_editor":
    cmd, ti = tool_input["command"], tool_input
    if cmd == "view":
      return editor.view(ti["path"],
                         ti.get("view_range"))
    elif cmd == "str_replace":
      return editor.str_replace(ti["path"],
                ti["old_str"], ti["new_str"])
    elif cmd == "create":
      return editor.create(ti["path"],
                           ti["file_text"])
    elif cmd == "insert":
      return editor.insert(ti["path"],
                ti["insert_line"], ti["new_str"])
    ...
```

Running It DEMO 5

```
editor = TextEditorTool("workspace")
messages = []
add_user_message(messages,
    "Open the ./main.py file and add a function that calculates pi to the 5th digit. "
    "Then create a ./test_main.py file that tests your implementation.")
run_conversation(messages, tools=[TEXT_EDITOR_SCHEMA],
                 registry={TEXT_EDITOR_NAME: editor})
```

- Same loop: `view` → `str_replace` → `create` → summary
- **Why?** Build your own AI editor, doc bot, or autograder — **you control which files** Claude touches

A Server Tool: Web Search

> **Anthropic runs it.** Claude searches and cites within one call — **no `tool_use` for you to answer.**

```
web_search_schema = {
    "type": "web_search_20250305",
    "name": "web_search",
    "max_uses": 5,
    "allowed_domains": ["nih.gov"],
}

messages = []
add_user_message(messages,
  "What's the best exercise for "
  "gaining leg muscle?")
response = chat(messages,
                tools=[web_search_schema])
```

- Enable in Claude Console
- **`max_uses`** caps cost
- **`allowed_domains`** → trusted sources
- Newer: `web_search_20260209`

The Response DEMO 6

Block	Contains
`server_tool_use`	the query
`web_search_tool_result`	results (`title`, `url`)
`text` + citations	answer with sources

```python
# demos/06_web_search.py
for block in response.content:
    if block.type == "server_tool_use":
        print(f"[search] {block.input}")
    elif block.type == "web_search_tool_result":
        if isinstance(block.content, list):
            for r in block.content:
                print(f"  source: {r.title} -- {r.url}")
        else:
            print(f"  search error: "
                  f"{block.content.error_code}")
    elif block.type == "text":
        print(block.text, end="")
        for c in block.citations or []:
            print(f" [{c.url}]", end="")
```

- Search errors **don't raise** — check `content`
- Citations **build trust**

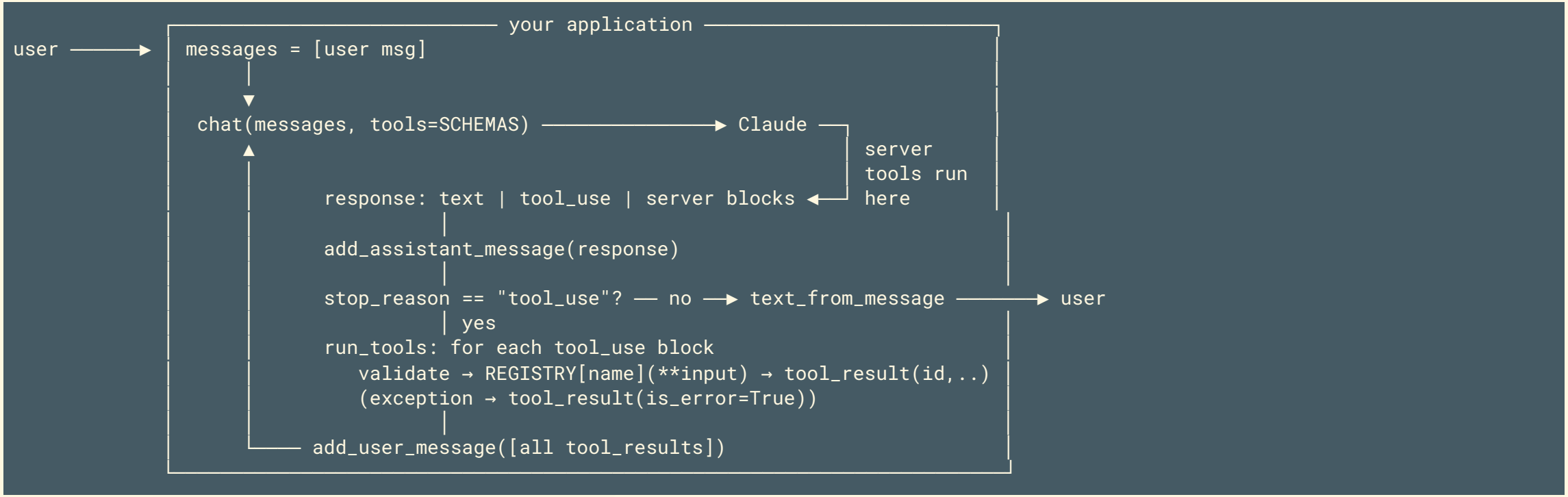
`pause_turn`, and Client vs. Server Tools

Long searches may return `pause_turn` — resend as-is:

```
while response.stop_reason == "pause_turn":       # the demo caps this
    add_assistant_message(messages, response)
    response = chat(messages, tools=[web_search_schema])
```

	Your tool	Text editor	Web search
Schema	you	built in	built in
Executes	you	you	**Anthropic**
Stop reason	`tool_use`	`tool_use`	`end_turn` / `pause_turn`
Main risk	bad args	path traversal	untrusted content, cost

The Complete Picture



The model decides; your loop executes. Errors become results, not crashes.

The Project: `claude-tool-use`

```
claude-tool-use/
├── tooluse/
│   ├── chat.py          # helpers, chat
│   ├── tools.py         # TerpRemind tools + schemas
│   ├── loop.py          # run_tools, run_conversation
│   ├── text_editor.py   # sandboxed editor
│   └── streaming.py     # save_article, validation
├── demos/               # 01 ... 06
├── tests/               # offline, FakeClient
└── workspace/main.py    # editor sandbox
```

```
cd code/claude-tool-use
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
pip install -e .
pytest -q              # no key needed

cp .env.example .env   # add your key
python demos/03_reminder_agent.py
```

Model: `CLAUDE_MODEL` (default `claude-opus-5-5`).

Testing the Loop Without an API Key

Dependency injection: pass a fake `client` into `chat()`.

```
class FakeClient:
    def __init__(self, responses):
        self.responses = list(responses)
        self.requests = []
        self.messages = self    # client.messages.create

    def create(self, **params):
        self.requests.append(
            {**params,
             "messages": list(params["messages"])})
        return self.responses.pop(0)
```

```
run_conversation(messages, tools=["schemas"],
                 registry=registry, client=client,
                 verbose=False)

assert len(client.requests) == 3
# schema sent every time
assert all(r["tools"] == ["schemas"]
           for r in client.requests)
assert [m["role"] for m in messages] \
       == ["user", "assistant"] * 3
# results matched by ID
assert messages[2]["content"][0]["tool_use_id"] == "t1"
```

Summary

1. The model **requests**; your code **executes** (except server tools)
2. A tool = **function + schema**; the description is a prompt
3. Append the **whole** response; filter blocks by `type`
4. **Every** `tool_use` **gets a** `tool_result` with its ID, in one user message
5. **Loop on** `stop_reason`, same tools every call, bounded
6. Adding a tool: **four steps**, loop untouched
7. Streaming buffers per key; **eager** trades validation for speed
8. Text editor: **sandbox paths**. Web search: **render citations**

Where This Leads

- **MCP** — servers **publish tools** any client can use
- **Agents** — this loop + more tools, memory, autonomy
- **Security** — **prompt injection** can trigger your tools
- **Evaluation** — log calls; measure tool selection

Questions?