Programmatic tool calling

Copy page

Programmatic tool calling allows Claude to write code that calls your tools programmatically within a code execution container, rather than requiring round trips through the model for each tool invocation. This reduces latency for multi-tool workflows and decreases token consumption by allowing Claude to filter or process data before it reaches the model's context window.

Programmatic tool calling is currently in public beta.

To use this feature, add the `"advanced-tool-use-2025-11-20"` beta header to your API requests.

This feature requires the code execution tool to be enabled.

Please reach out through our feedback form to share your feedback on this feature.

Model compatibility

Programmatic tool calling is available on the following models:

Model	Tool Version
Claude Opus 4.5 (`claude-opus-4-5-20251101`)	`code_execution_20250825`
Claude Sonnet 4.5 (`claude-sonnet-4-5-20250929`)	`code_execution_20250825`

Programmatic tool calling is available via the Claude API and Microsoft Foundry.

Quick start

Here's a simple example where Claude programmatically queries a data times and aggregates results:

```
curl https://api.anthropic.com/v1/messages \
    --header "x-api-key: $ANTHROPIC_API_KEY" \
    --header "anthropic-version: 2023-06-01" \
    --header "anthropic-beta: advanced-tool-use-2025-11-20" \
    --header "content-type: application/json" \
    --data '{
        "model": "claude-sonnet-4-5",
        "max_tokens": 4096,
        "messages": [
            {
                "role": "user",
                "content": "Query sales data for the West, East, and Central r
            }
        ],
        "tools": [
            {
                "type": "code_execution_20250825",
                "name": "code_execution"
            },
            {
                "name": "query_database",
                "description": "Execute a SQL query against the sales database
                "input_schema": {
                    "type": "object",
                    "properties": {
                        "sql": {
                            "type": "string",
                            "description": "SQL query to execute"
                        }
                    },
                    "required": ["sql"]
                },
                "allowed_callers": ["code_execution_20250825"]
            }
        ]
    }'
```

How programmatic tool calling works

When you configure a tool to be callable from code execution and Claude decides to use that tool:

1. Claude writes Python code that invokes the tool as a function, potentially including multiple tool calls and pre/post-processing logic

`tool_use` block

4. You provide the tool result, and code execution continues (intermediate results are not loaded into Claude's context window)
5. Once all code execution completes, Claude receives the final output and continues working on the task

This approach is particularly useful for:

- **Large data processing**: Filter or aggregate tool results before they reach Claude's context
- **Multi-step workflows**: Save tokens and latency by calling tools serially or in a loop without sampling Claude in-between tool calls
- **Conditional logic**: Make decisions based on intermediate tool results

Custom tools are converted to async Python functions to support parallel tool calling. When Claude writes code that calls your tools, it uses `await` (e.g., `result = await query_database("<sql>")`) and automatically includes the appropriate async wrapper function.

The async wrapper is omitted from code examples in this documentation for clarity.

Core concepts

The `allowed_callers` field

The `allowed_callers` field specifies which contexts can invoke a tool:

```
{
  "name": "query_database",
  "description": "Execute a SQL query against the database",
  "input_schema": {...},
  "allowed_callers": ["code_execution_20250825"]
}
```

Possible values:

- `["direct"]` - Only Claude can call this tool directly (default if omitted)
- `["code_execution_20250825"]` - Only callable from within code execution

We recommend choosing either `["direct"]` or `["code_execution_20250825"]` for each tool rather than enabling both, as this provides clearer guidance to Claude for how best to use the tool.

The `caller` field in responses

Every tool use block includes a `caller` field indicating how it was invoked:

Direct invocation (traditional tool use):

```
{
  "type": "tool_use",
  "id": "toolu_abc123",
  "name": "query_database",
  "input": {"sql": "<sql>"},
  "caller": {"type": "direct"}
}
```

Programmatic invocation:

```
{
  "type": "tool_use",
  "id": "toolu_xyz789",
  "name": "query_database",
  "input": {"sql": "<sql>"},
  "caller": {
    "type": "code_execution_20250825",
    "tool_id": "srvtoolu_abc123"
  }
}
```

The `tool_id` references the code execution tool that made the programmatic call.

Container lifecycle

Programmatic tool calling uses the same containers as code execution:

- **Container creation:** A new container is created for each session unless you reuse an existing one

- **Container ID**: Returned in responses via the `container` field
- **Reuse**: Pass the container ID to maintain state across requests

⚠ When a tool is called programmatically and the container is waiting for your tool result, you must respond before the container expires. Monitor the `expires_at` field. If the container expires, Claude may treat the tool call as timed out and retry it.

Example workflow

Here's how a complete programmatic tool calling flow works:

Step 1: Initial request

Send a request with code execution and a tool that allows programmatic calling. To enable programmatic calling, add the `allowed_callers` field to your tool definition.

ⓘ Provide detailed descriptions of your tool's output format in the tool description. If you specify that the tool returns JSON, Claude will attempt to deserialize and process the result in code. The more detail you provide about the output schema, the better Claude can handle the response programmatically.

Python

```
    betas=["advanced-tool-use-2025-11-20"],
    max_tokens=4096,
    messages=[{
        "role": "user",
        "content": "Query customer purchase history from the last quarter and
    }],
    tools=[
        {
            "type": "code_execution_20250825",
            "name": "code_execution"
        },
        {
            "name": "query_database",
            "description": "Execute a SQL query against the sales database. Re
            "input_schema": {...},
            "allowed_callers": ["code_execution_20250825"]
        }
    ]
)
```

Step 2: API response with tool call

Claude writes code that calls your tool. The API pauses and returns:

```
    "content": [
      {
        "type": "text",
        "text": "I'll query the purchase history and analyze the results."
      },
      {
        "type": "server_tool_use",
        "id": "srvtoolu_abc123",
        "name": "code_execution",
        "input": {
          "code": "results = await query_database('<sql>')\ntop_customers = sort
        }
      },
      {
        "type": "tool_use",
        "id": "toolu_def456",
        "name": "query_database",
        "input": {"sql": "<sql>"},
        "caller": {
          "type": "code_execution_20250825",
          "tool_id": "srvtoolu_abc123"
        }
      }
    ],
    "container": {
      "id": "container_xyz789",
      "expires_at": "2025-01-15T14:30:00Z"
    },
    "stop_reason": "tool_use"
  }
```

Step 3: Provide tool result

Include the full conversation history plus your tool result:

Python

```
    betas=["advanced-tool-use-2025-11-20"],
    max_tokens=4096,
    container="container_xyz789",  # Reuse the container
    messages=[
        {"role": "user", "content": "Query customer purchase history from the
        {
            "role": "assistant",
            "content": [
                {"type": "text", "text": "I'll query the purchase history and
                {
                    "type": "server_tool_use",
                    "id": "srvtoolu_abc123",
                    "name": "code_execution",
                    "input": {"code": "..."}
                },
                {
                    "type": "tool_use",
                    "id": "toolu_def456",
                    "name": "query_database",
                    "input": {"sql": "<sql>"},
                    "caller": {
                        "type": "code_execution_20250825",
                        "tool_id": "srvtoolu_abc123"
                    }
                }
            ]
        },
        {
            "role": "user",
            "content": [
                {
                    "type": "tool_result",
                    "tool_use_id": "toolu_def456",
                    "content": "[{\"customer_id\": \"C1\", \"revenue\": 45000}
                }
            ]
        }
    ],
    tools=[...]
)
```

Step 4: Next tool call or completion

Claude Docs

Step 5: Final response

Once the code execution completes, Claude provides the final response:

```
{
  "content": [
    {
      "type": "code_execution_tool_result",
      "tool_use_id": "srvtoolu_abc123",
      "content": {
        "type": "code_execution_result",
        "stdout": "Top 5 customers by revenue:\n1. Customer C1: $45,000\n2. Cu
        "stderr": "",
        "return_code": 0,
        "content": []
      }
    },
    {
      "type": "text",
      "text": "I've analyzed the purchase history from last quarter. Your top
    }
  ],
  "stop_reason": "end_turn"
}
```

Advanced patterns

Batch processing with loops

Claude can write code that processes multiple items efficiently:

```python
results = {}
for region in regions:
    data = await query_database(f"<sql for {region}>")
    results[region] = sum(row["revenue"] for row in data)

# Process results programmatically
top_region = max(results.items(), key=lambda x: x[1])
print(f"Top region: {top_region[0]} with ${top_region[1]:,} in revenue")
```

This pattern:

- Reduces model round-trips from N (one per region) to 1
- Processes large result sets programmatically before returning to Claude
- Saves tokens by only returning aggregated conclusions instead of raw data

Early termination

Claude can stop processing as soon as success criteria are met:

```python
# async wrapper omitted for clarity
endpoints = ["us-east", "eu-west", "apac"]
for endpoint in endpoints:
    status = await check_health(endpoint)
    if status == "healthy":
        print(f"Found healthy endpoint: {endpoint}")
        break  # Stop early, don't check remaining
```

Conditional tool selection

```python
# async wrapper omitted for clarity
file_info = await get_file_info(path)
if file_info["size"] < 10000:
    content = await read_full_file(path)
else:
    content = await read_file_summary(path)
print(content)
```

```python
# async wrapper omitted for clarity
logs = await fetch_logs(server_id)
errors = [log for log in logs if "ERROR" in log]
print(f"Found {len(errors)} errors")
for error in errors[-10:]:  # Only return last 10 errors
    print(error)
```

Response format

Programmatic tool call

When code execution calls a tool:

```json
{
  "type": "tool_use",
  "id": "toolu_abc123",
  "name": "query_database",
  "input": {"sql": "<sql>"},
  "caller": {
    "type": "code_execution_20250825",
    "tool_id": "srvtoolu_xyz789"
  }
}
```

Tool result handling

Your tool result is passed back to the running code:

```
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_abc123",
      "content": "[{\"customer_id\": \"C1\", \"revenue\": 45000, \"orders\": 2
    }
  ]
}
```

Code execution completion

When all tool calls are satisfied and code completes:

```
{
  "type": "code_execution_tool_result",
  "tool_use_id": "srvtoolu_xyz789",
  "content": {
    "type": "code_execution_result",
    "stdout": "Analysis complete. Top 5 customers identified from 847 total re
    "stderr": "",
    "return_code": 0,
    "content": []
  }
}
```

Error handling

Common errors

Error	Description	Solution
`invalid_tool_input`	Tool input doesn't match schema	Validate your tool's input_schema
`tool_not_allowed`	Tool doesn't allow the requested caller type	Check `allowed_callers` includes the right contexts
`missing_beta_header`	PTC beta header not provided	Add both beta headers to your request

Claude Docs

If your tool takes too long to respond, the code execution will receive a `TimeoutError`. Claude sees this in stderr and will typically retry:

```
{
  "type": "code_execution_tool_result",
  "tool_use_id": "srvtoolu_abc123",
  "content": {
    "type": "code_execution_result",
    "stdout": "",
    "stderr": "TimeoutError: Calling tool ['query_database'] timed out.",
    "return_code": 0,
    "content": []
  }
}
```

To prevent timeouts:

- Monitor the `expires_at` field in responses
- Implement timeouts for your tool execution
- Consider breaking long operations into smaller chunks

Tool execution errors

If your tool returns an error:

```
# Provide error information in the tool result
{
    "type": "tool_result",
    "tool_use_id": "toolu_abc123",
    "content": "Error: Query timeout - table lock exceeded 30 seconds"
}
```

Claude's code will receive this error and can handle it appropriately.

Constraints and limitations

Feature incompatibilities

- **Tool choice**: You cannot force programmatic calling of a specific tool via `tool_choice`
- **Parallel tool use**: `disable_parallel_tool_use: true` is not supported with programmatic calling

Tool restrictions

The following tools cannot currently be called programmatically, but support may be added in future releases:

- Web search
- Web fetch
- Tools provided by an MCP connector

Message formatting restrictions

When responding to programmatic tool calls, there are strict formatting requirements:

Tool result only responses: If there are pending programmatic tool calls waiting for results, your response message must contain **only** `tool_result` blocks. You cannot include any text content, even after the tool results.

```
// ❌ INVALID - Cannot include text when responding to programmatic tool c  s
{
  "role": "user",
  "content": [
    {"type": "tool_result", "tool_use_id": "toolu_01", "content": "[{\"custome
    {"type": "text", "text": "What should I do next?"}  // This will cause an
  ]
}

// ✅ VALID - Only tool results when responding to programmatic tool calls
{
  "role": "user",
  "content": [
    {"type": "tool_result", "tool_use_id": "toolu_01", "content": "[{\"custome
  ]
}
```

This restriction only applies when responding to programmatic (code execution) tool calls. For regular client-side tool calls, you can include text content after tool results.

Programmatic tool calls are subject to the same rate limits as regular tool calls. Each tool call from code execution counts as a separate invocation.

Validate tool results before use

When implementing custom tools that will be called programmatically:

- **Tool results are returned as strings**: They can contain any content, including code snippets or executable commands that may be processed by the execution environment.
- **Validate external tool results**: If your tool returns data from external sources or accepts user input, be aware of code injection risks if the output will be interpreted or executed as code.

Token efficiency

Programmatic tool calling can significantly reduce token consumption:

- **Tool results from programmatic calls are not added to Claude's context** - only the final code output is
- **Intermediate processing happens in code** - filtering, aggregation, etc. don't consume model tokens
- **Multiple tool calls in one code execution** - reduces overhead compared to separate model turns

For example, calling 10 tools directly uses ~10x the tokens of calling them programmatically and returning a summary.

Usage and pricing

Programmatic tool calling uses the same pricing as code execution. See the code execution pricing for details.

Token counting for programmatic tool calls: Tool results from programmatic invocations do not count toward your input/output token usage. Only the final code execution result and Claude's response count.

Best practices

- **Provide detailed output descriptions**: Since Claude deserializes tool results in code, clearly document the format (JSON structure, field types, etc.)
- **Return structured data**: JSON or other easily parseable formats work best for programmatic processing
- **Keep responses concise**: Return only necessary data to minimize processing overhead

When to use programmatic calling

Good use cases:

- Processing large datasets where you only need aggregates or summaries
- Multi-step workflows with 3+ dependent tool calls
- Operations requiring filtering, sorting, or transformation of tool results
- Tasks where intermediate data shouldn't influence Claude's reasoning
- Parallel operations across many items (e.g., checking 50 endpoints)

Less ideal use cases:

- Single tool calls with simple responses
- Tools that need immediate user feedback
- Very fast operations where code execution overhead would outweigh the benefit

Performance optimization

- **Reuse containers** when making multiple related requests to maintain state
- **Batch similar operations** in a single code execution when possible

Troubleshooting

Common issues

"Tool not allowed" error

- Verify your tool definition includes `"allowed_callers": ["code_execution_20250825"]`
- Check that you're using the correct beta headers

- Monitor the `expires_at` field in responses
- Consider implementing faster tool execution

Beta header issues

- You need the header: `"advanced-tool-use-2025-11-20"`

Tool result not parsed correctly

- Ensure your tool returns string data that Claude can deserialize
- Provide clear output format documentation in your tool description

Debugging tips

1. **Log all tool calls and results** to track the flow
2. **Check the `caller` field** to confirm programmatic invocation
3. **Monitor container IDs** to ensure proper reuse
4. **Test tools independently** before enabling programmatic calling

Why programmatic tool calling works

Claude's training includes extensive exposure to code, making it effective at reasoning through and chaining function calls. When tools are presented as callable functions within a code execution environment, Claude can leverage this strength to:

- **Reason naturally about tool composition**: Chain operations and handle dependencies as naturally as writing any Python code
- **Process large results efficiently**: Filter down large tool outputs, extract only relevant data, or write intermediate results to files before returning summaries to the context window
- **Reduce latency significantly**: Eliminate the overhead of re-sampling Claude between each tool call in multi-step workflows

This approach enables workflows that would be impractical with traditional tool use—such as processing files over 1M tokens—by allowing Claude to work with data programmatically rather than loading everything into the conversation context.

Alternative implementations

Client-side direct execution

Provide Claude with a code execution tool and describe what functions are available in that environment. When Claude invokes the tool with code, your application executes it locally where those functions are defined.

Advantages:

- Simple to implement with minimal re-architecting
- Full control over the environment and instructions

Disadvantages:

- Executes untrusted code outside of a sandbox
- Tool invocations can be vectors for code injection

Use when: Your application can safely execute arbitrary code, you want a simple solution, and Anthropic's managed offering doesn't fit your needs.

Self-managed sandboxed execution

Same approach from Claude's perspective, but code runs in a sandboxed container with security restrictions (e.g., no network egress). If your tools require external resources, you'll need a protocol for executing tool calls outside the sandbox.

Advantages:

- Safe programmatic tool calling on your own infrastructure
- Full control over the execution environment

Disadvantages:

- Complex to build and maintain
- Requires managing both infrastructure and inter-process communication

Use when: Security is critical and Anthropic's managed solution doesn't fit your requirements.

Anthropic-managed execution

Anthropic's programmatic tool calling is a managed version of sandboxed execution with an opinionated Python environment tuned for Claude. Anthropic handles container

- Safe and secure by default
- Easy to enable with minimal configuration
- Environment and instructions optimized for Claude

We recommend using Anthropic's managed solution if you're using the Claude API.

Related features

Code Execution Tool

Learn about the underlying code execution capability that powers programmatic tool calling.

Tool Use Overview

Understand the fundamentals of tool use with Claude.

Implement Tool Use

Step-by-step guide for implementing tools.

Claude Docs

Education

Financial services

Government

Life sciences

Partners

Amazon Bedrock

Google Cloud's Vertex AI

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

Discord

Terms and policies

Privacy policy

Responsible disclosure policy

Terms of service: Commercial

Terms of service: Consumer

Usage policy