Introduction TO MCP Course Lessons

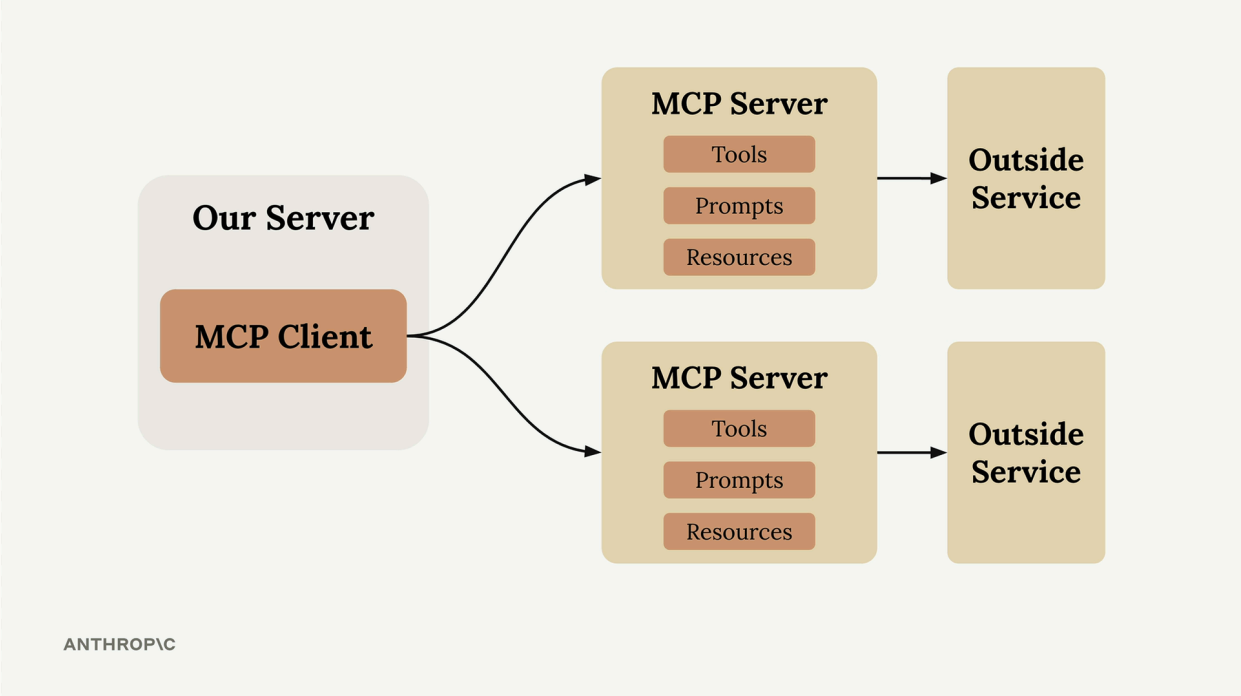
- Direct link to the UV install guide: https://docs.astral.sh/uv/
- Model Context Protocol introduction:

  https://modelcontextprotocol.io/introduction

Introducing MCP

Summary

Model Context Protocol (MCP) is a communication layer that provides Claude with context and tools without requiring you to write a bunch of tedious integration code. Think of it as a way to shift the burden of tool definitions and execution away from your server to specialized MCP servers.

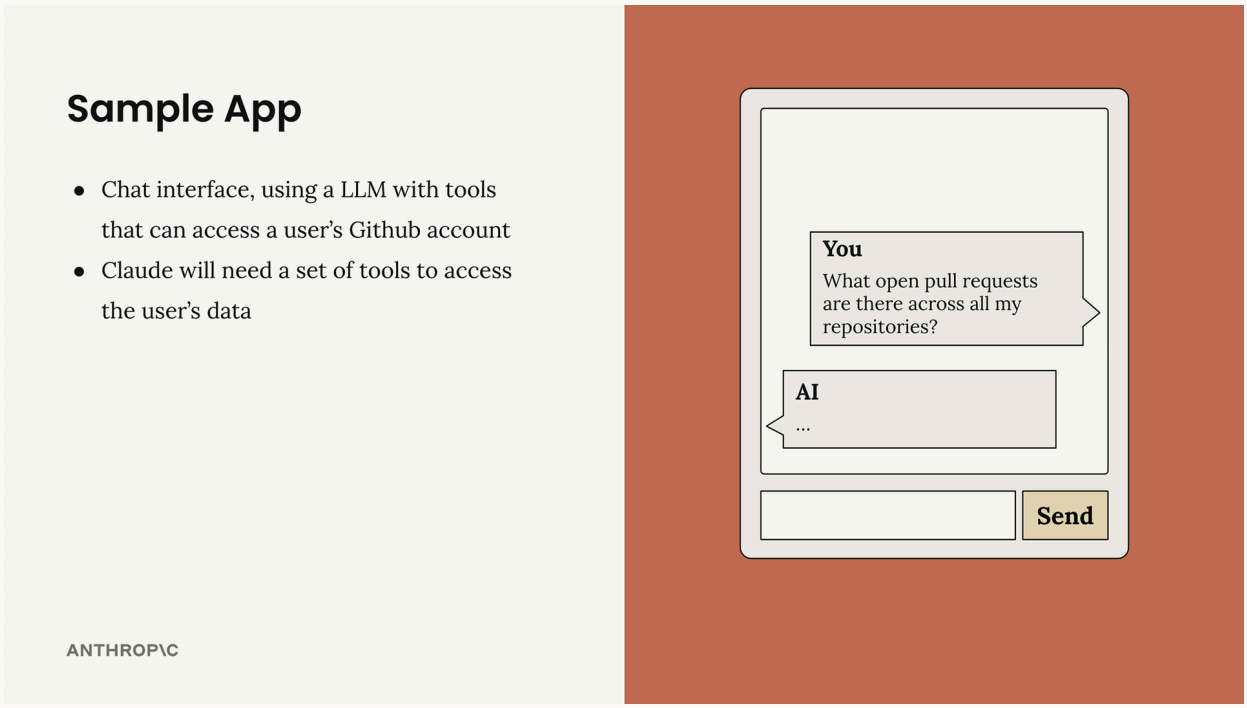


When you first encounter MCP, you'll see diagrams showing the basic architecture: an MCP Client (your server) connecting to MCP Servers that

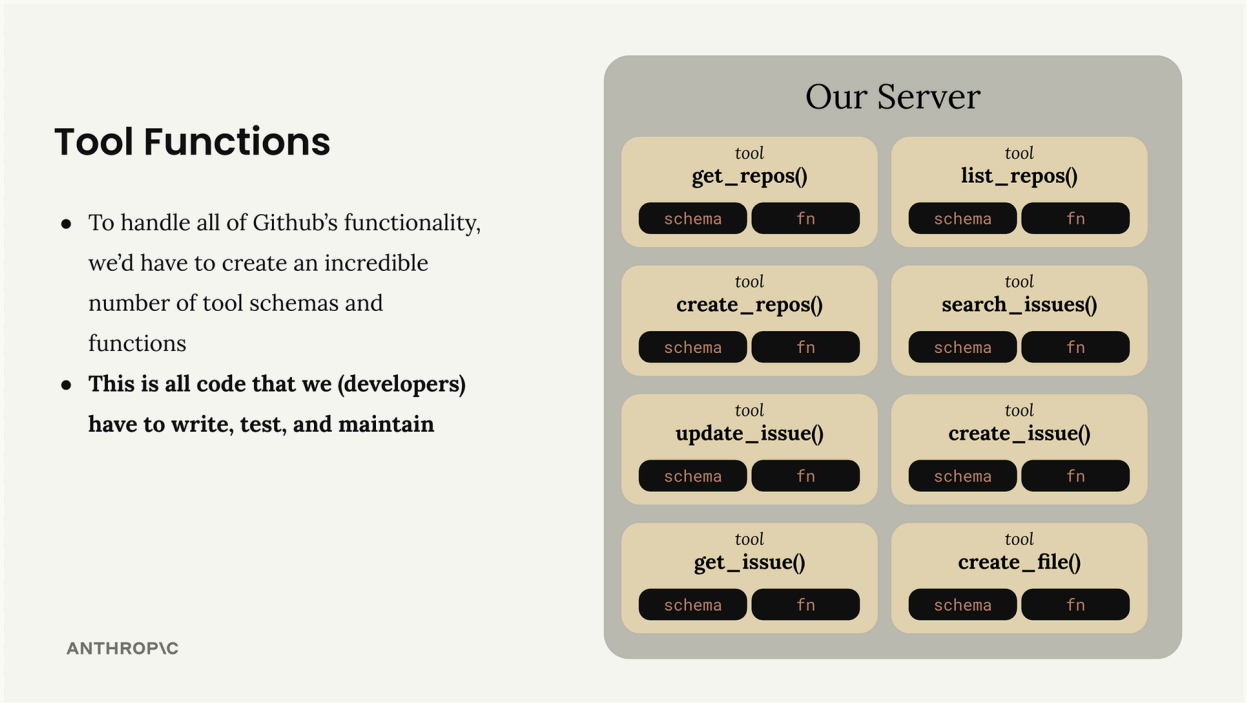
contain tools, prompts, and resources. Each MCP Server acts as an interface to some outside service.

The Problem MCP Solves

Let's say you're building a chat interface where users can ask Claude about their GitHub data. A user might ask "What open pull requests are there across all my repositories?" To handle this, Claude needs tools to access GitHub's API.



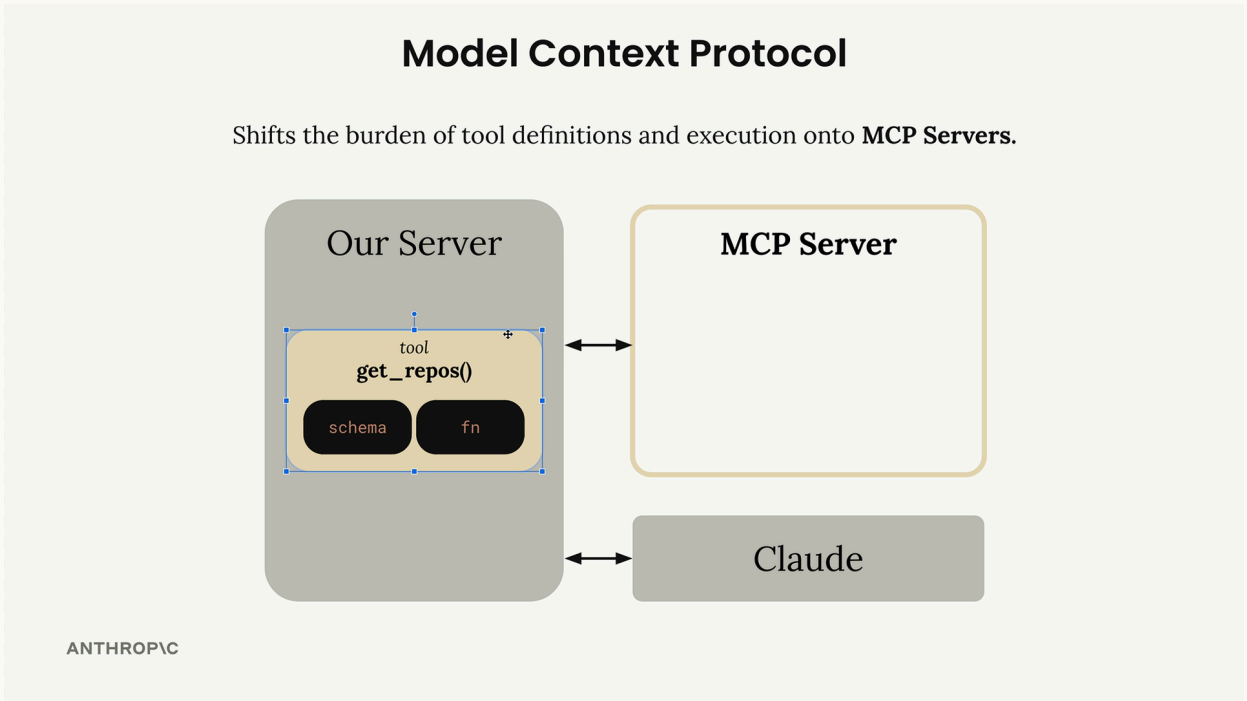
GitHub has massive functionality - repositories, pull requests, issues, projects, and tons more. Without MCP, you'd need to create an incredible number of tool schemas and functions to handle all of GitHub's features.



This means writing, testing, and maintaining all that integration code yourself. That's a lot of effort and ongoing maintenance burden.

How MCP Works

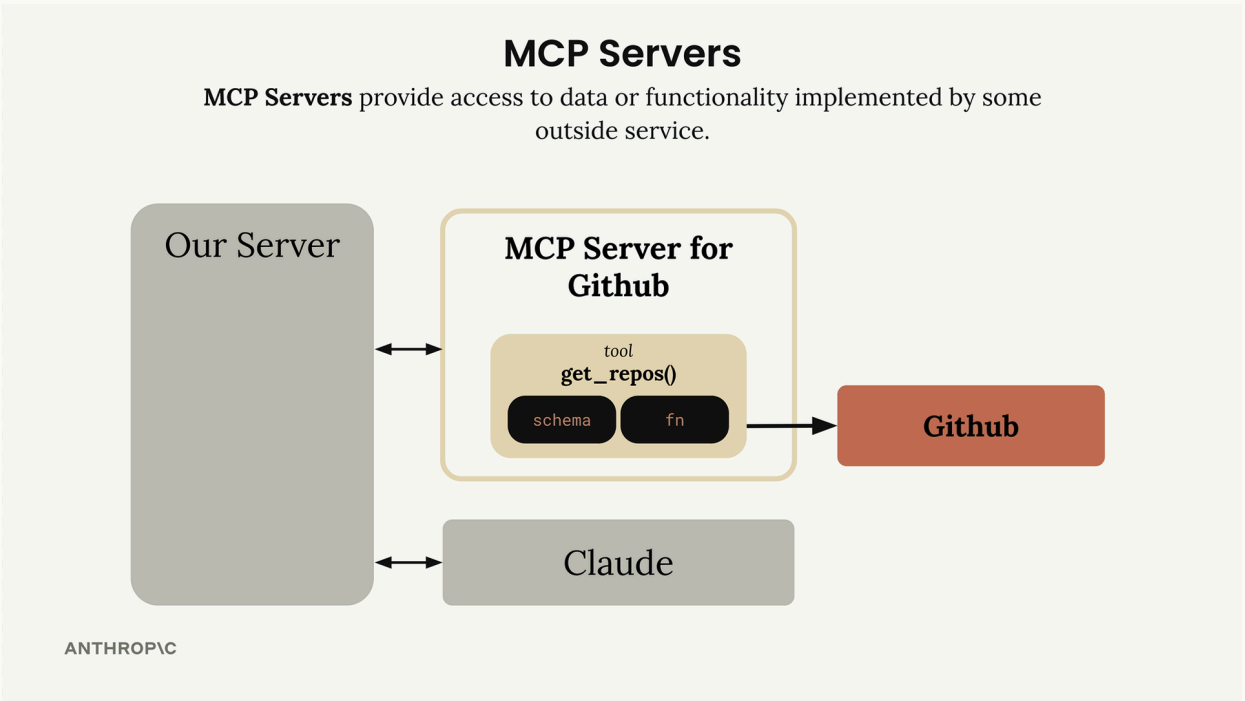
MCP shifts this burden by moving tool definitions and execution from your server to dedicated MCP servers. Instead of you authoring all those GitHub tools, an MCP Server for GitHub handles it.



The MCP Server wraps up tons of functionality around GitHub and exposes it as a standardized set of tools. Your application connects to this MCP server instead of implementing everything from scratch.

MCP Servers Explained

MCP Servers provide access to data or functionality implemented by outside services. They act as specialized interfaces that expose tools, prompts, and resources in a standardized way.



In our GitHub example, the MCP Server for GitHub contains tools like **get_repos()** and connects directly to GitHub's API. Your server communicates with the MCP server, which handles all the GitHub-specific implementation details.

Common Questions

Who authors MCP Servers?

Anyone can create an MCP server implementation. Often, service providers themselves will make their own official MCP implementations. For example, AWS might release an official MCP server with tools for their various services.

How is this different from calling APIs directly?

MCP servers provide tool schemas and functions already defined for you. If you want to call an API directly, you'll be authoring those tool definitions on your own. MCP saves you that implementation work.

Isn't MCP just the same as tool use?

This is a common misconception. MCP servers and tool use are complementary but different concepts. MCP servers provide tool schemas and functions already defined for you, while tool use is about how Claude actually calls those tools. The key difference is who does the work - with MCP, someone else has already implemented the tools for you.

The benefit is clear: instead of maintaining a complex set of integrations yourself, you can leverage MCP servers that handle the heavy lifting of connecting to external services.

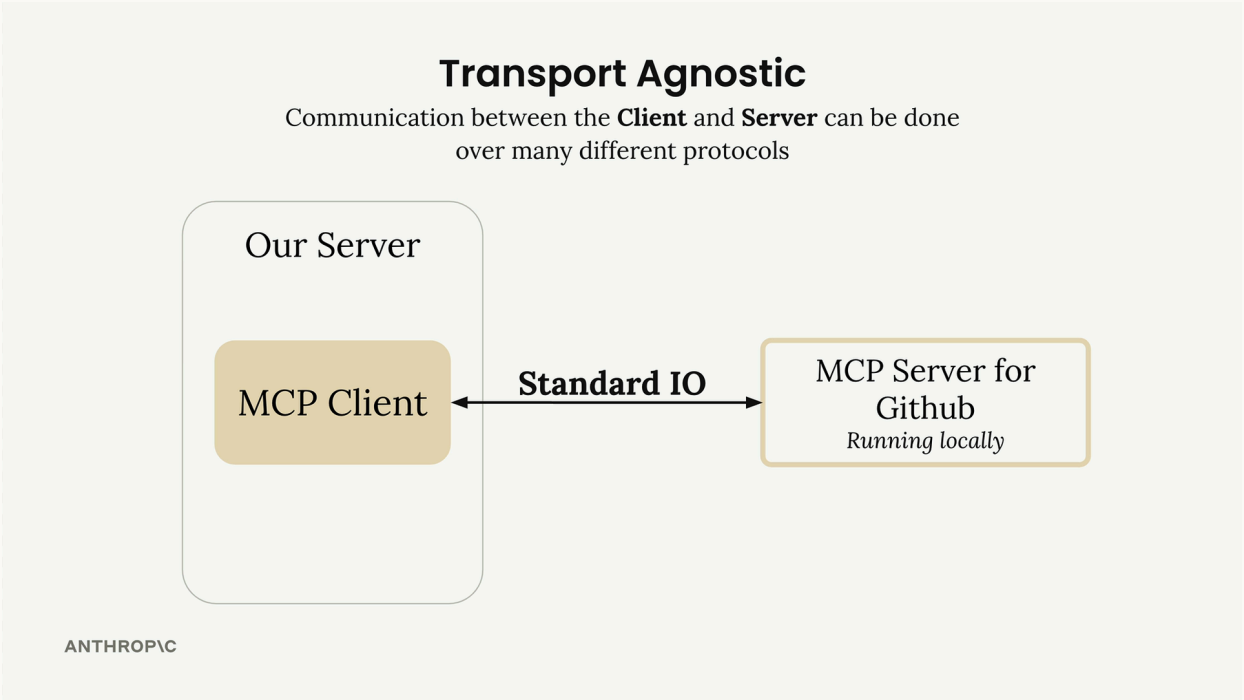
MCP clients

Summary

The MCP client serves as the communication bridge between your server and MCP servers. It's your access point to all the tools that an MCP server provides, handling the message exchange and protocol details so your application doesn't have to.

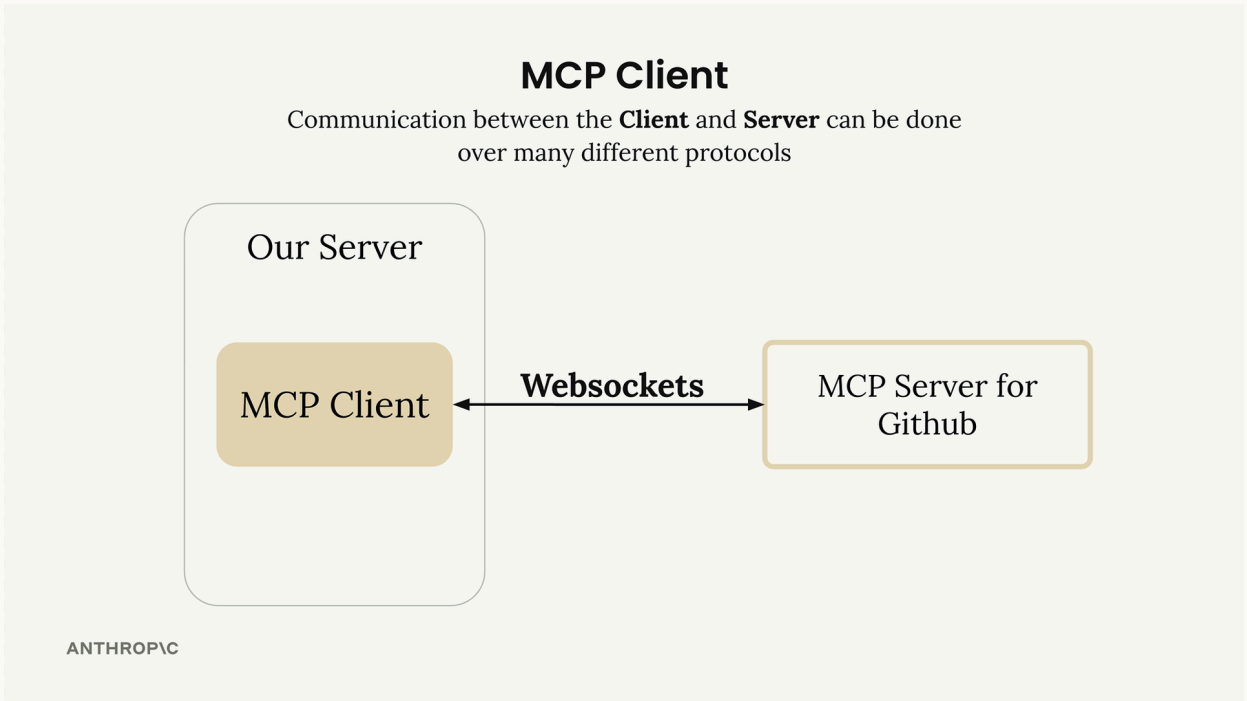
Transport Agnostic Communication

One of MCP's key strengths is being transport agnostic - a fancy way of saying the client and server can communicate over different protocols depending on your setup.



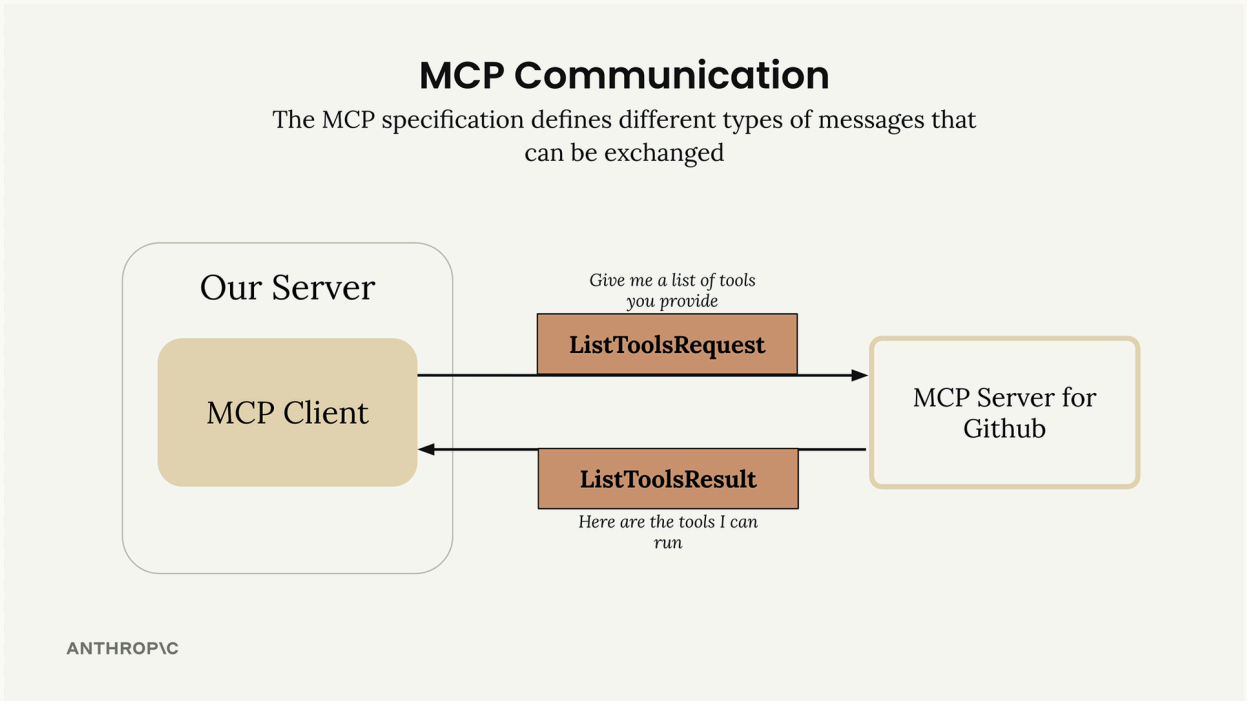
The most common setup runs both the MCP client and server on the same machine, communicating through standard input/output. But you can also connect them over:

- HTTP
- WebSockets
- Various other network protocols

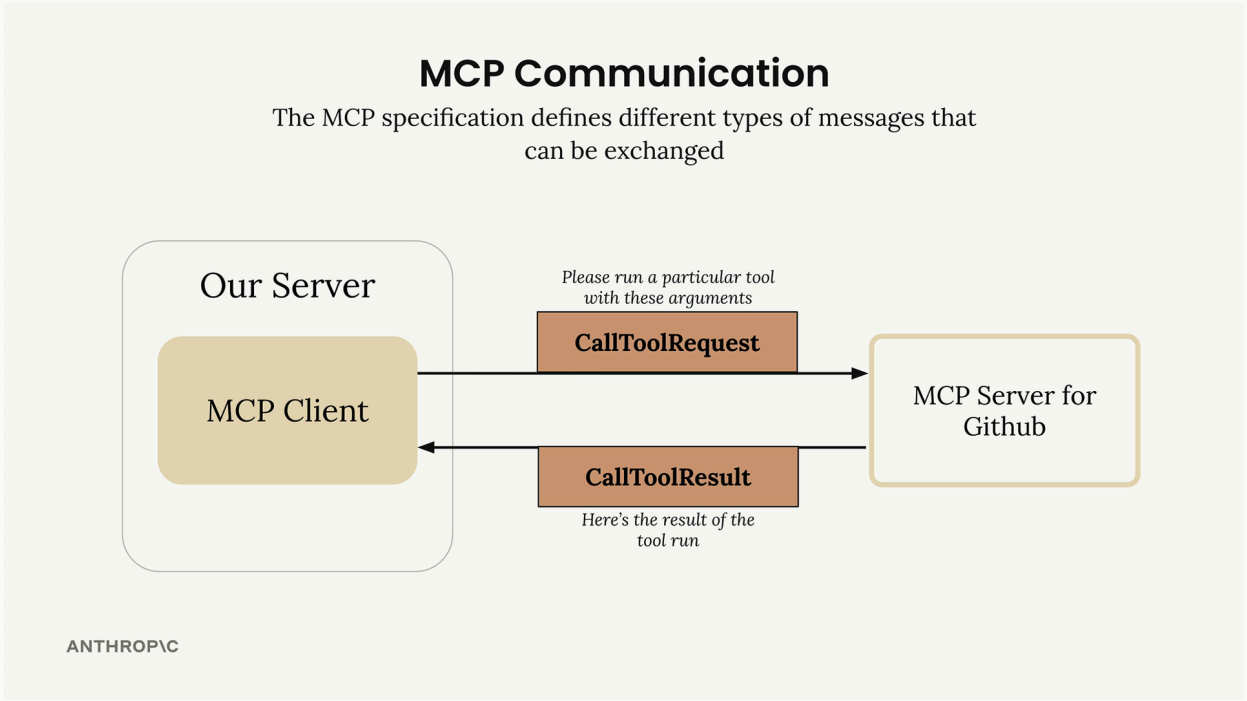


MCP Message Types

Once connected, the client and server exchange specific message types defined in the MCP specification. The main ones you'll work with are:



ListToolsRequest/ListToolsResult: The client asks the server "what tools do you provide?" and gets back a list of available tools.



CallToolRequest/CallToolResult: The client asks the server to run a specific tool with given arguments, then receives the results.

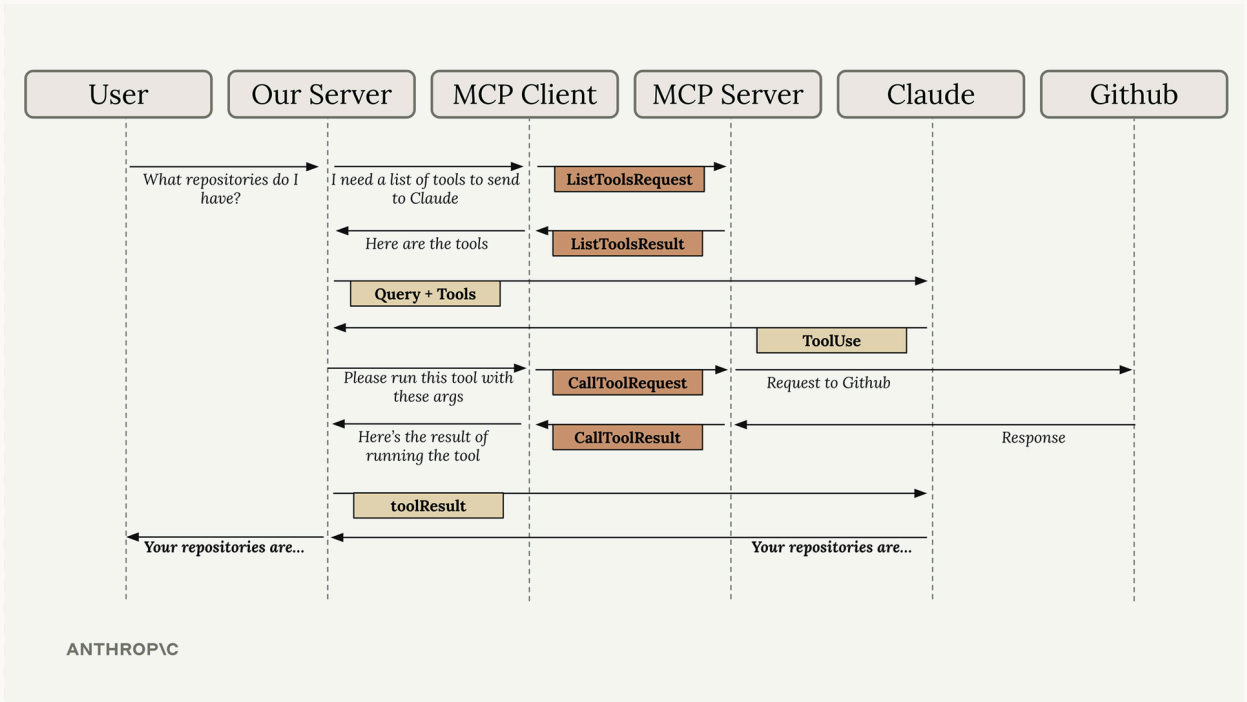
How It All Works Together

Here's a complete example showing how a user query flows through the entire system - from your server, through the MCP client, to external services like GitHub, and back to Claude.

Let's say a user asks "What repositories do I have?" Here's the step-by-step flow:

1. User Query: The user submits their question to your server
2. Tool Discovery: Your server needs to know what tools are available to send to Claude
3. List Tools Exchange: Your server asks the MCP client for available tools
4. MCP Communication: The MCP client sends a **ListToolsRequest** to the MCP server and receives a **ListToolsResult**
5. Claude Request: Your server sends the user's query plus the available tools to Claude
6. Tool Use Decision: Claude decides it needs to call a tool to answer the question
7. Tool Execution Request: Your server asks the MCP client to run the tool Claude specified

8. External API Call: The MCP client sends a **CallToolRequest** to the MCP server, which makes the actual GitHub API call
9. Results Flow Back: GitHub responds with repository data, which flows back through the MCP server as a **CallToolResult**
10. Tool Result to Claude: Your server sends the tool results back to Claude
11. Final Response: Claude formulates a final answer using the repository data
12. User Gets Answer: Your server delivers Claude's response back to the user



Yes, this flow involves many steps, but each component has a clear responsibility. The MCP client abstracts away the complexity of server

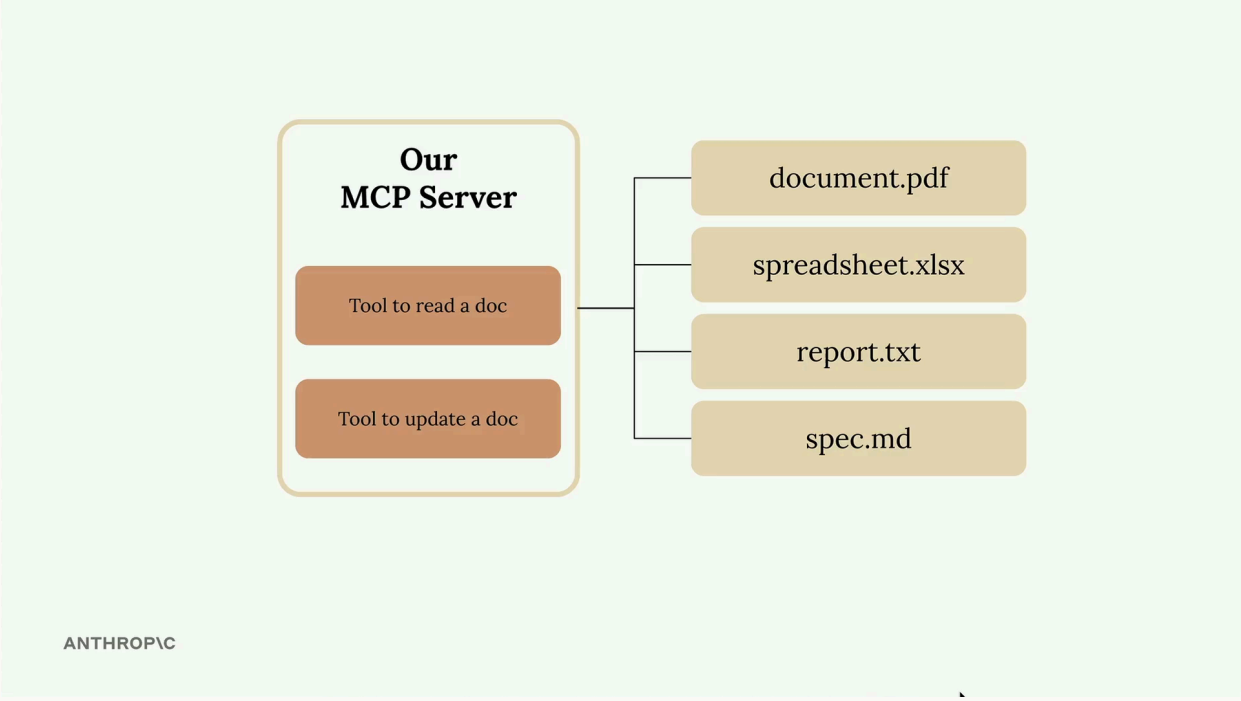
communication, letting you focus on your application logic while still getting access to powerful external tools and data sources.

Understanding this flow is crucial because you'll see all these pieces when building your own MCP clients and servers in the upcoming sections.

Hands-on with MCP servers

Defining tools with MCP

Building an MCP server becomes much simpler when you use the official Python SDK. Instead of writing complex JSON schemas by hand, you can define tools with decorators and let the SDK handle the heavy lifting.



In this example, we're creating a document management server with two core tools: one to read documents and another to update them. All documents exist

in memory as a simple dictionary where keys are document IDs and values are the content.

Setting Up the MCP Server

The Python MCP SDK makes server creation straightforward. You can initialize a server with just one line:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("DocumentMCP", log_level="ERROR")
```

Your documents can be stored in a simple dictionary structure:

```
docs = {
    "deposition.md": "This deposition covers the testimony of Angela Smith, P.E.",
    "report.pdf": "The report details the state of a 20m condenser tower.",
    "financials.docx": "These financials outline the project's budget and expenditures",
    "outlook.pdf": "This document presents the projected future performance of the system",
    "plan.md": "The plan outlines the steps for the project's implementation.",
    "spec.txt": "These specifications define the technical requirements for the equipment"
}
```

Tool Definition with Decorators

The SDK uses decorators to define tools. Instead of writing JSON schemas manually, you can use Python type hints and field descriptions. The SDK automatically generates the proper schema that Claude can understand.

Creating a Document Reader Tool

The first tool reads document contents by ID. Here's the complete implementation:

```
@mcp.tool(
    name="read_doc_contents",
    description="Read the contents of a document and return it as a string."
)
def read_document(
    doc_id: str = Field(description="Id of the document to read")
):
    if doc_id not in docs:
        raise ValueError(f"Doc with id {doc_id} not found")

    return docs[doc_id]
```

The decorator specifies the tool name and description, while the function parameters define the required arguments. The **Field** class from Pydantic provides argument descriptions that help Claude understand what each parameter expects.

Building a Document Editor Tool

The second tool performs simple find-and-replace operations on documents:

```
@mcp.tool(
    name="edit_document",
    description="Edit a document by replacing a string in the documents content with a new string."
)
def edit_document(
    doc_id: str = Field(description="Id of the document that will be edited"),
    old_str: str = Field(description="The text to replace. Must match exactly, including whitespace."),
    new_str: str = Field(description="The new text to insert in place of the old text.")
):
    if doc_id not in docs:
        raise ValueError(f"Doc with id {doc_id} not found")

    docs[doc_id] = docs[doc_id].replace(old_str, new_str)
```

This tool takes three parameters: the document ID, the text to find, and the replacement text. The implementation includes error handling for missing documents and performs a straightforward string replacement.

Key Benefits of the SDK Approach

- No manual JSON schema writing required
- Type hints provide automatic validation
- Clear parameter descriptions help Claude understand tool usage
- Error handling integrates naturally with Python exceptions

- Tool registration happens automatically through decorators

The MCP Python SDK transforms tool creation from a complex schema-writing exercise into simple Python function definitions. This approach makes it much easier to build and maintain MCP servers while ensuring Claude receives properly formatted tool specifications.

The server inspector

When building MCP servers, you need a way to test your functionality without connecting to a full application. The Python MCP SDK includes a built-in browser-based inspector that lets you debug and test your server in real-time.

Starting the Inspector

First, make sure your Python environment is activated (check your project's README for the exact command). Then run the inspector with:

mcp dev mcp_server.py

This starts a development server and gives you a local URL, typically something like **http://127.0.0.1:6274**. Open this URL in your browser to access the MCP Inspector.

Using the Inspector Interface

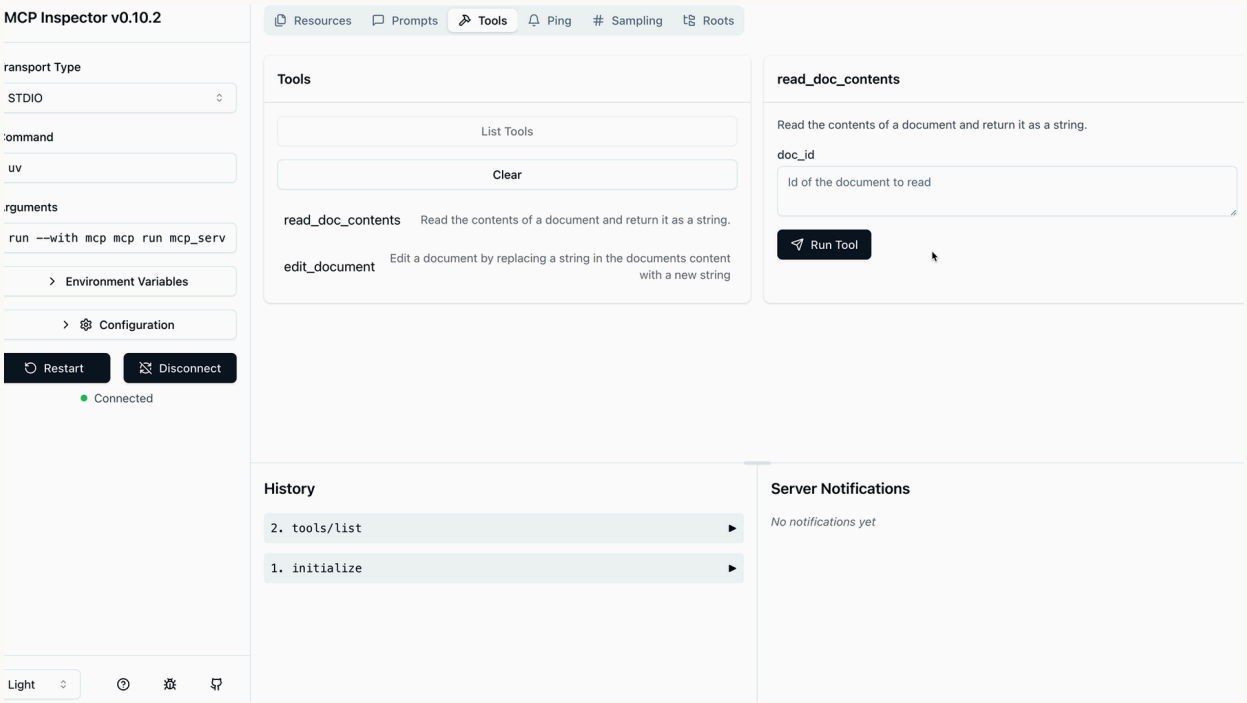
The inspector interface is actively being developed, so it may look different when you use it. However, the core functionality remains consistent. Look for these key elements:

- A Connect button to start your MCP server
- Navigation tabs for Resources, Tools, Prompts, and other features
- A tools listing and testing panel

Click the Connect button first to initialize your server. You'll see the connection status change from "Disconnected" to "Connected".

Testing Your Tools

Navigate to the Tools section and click "List Tools" to see all available tools from your server. When you select a tool, the right panel shows its details and input fields.



For example, to test a document reading tool:

1. Select the **read_doc_contents** tool
2. Enter a document ID (like "deposition.md")
3. Click "Run Tool"
4. Check the results for success and expected output

The inspector shows both the success status and the actual returned data, making it easy to verify your tool works correctly.

Testing Tool Interactions

You can test multiple tools in sequence to verify complex workflows. For instance, after using an edit tool to modify a document, immediately test the read tool to confirm the changes were applied correctly.

The inspector maintains your server state between tool calls, so edits persist and you can verify the complete functionality of your MCP server.

Development Workflow

The MCP Inspector becomes an essential part of your development process. Instead of writing separate test scripts or connecting to full applications, you can:

- Quickly iterate on tool implementations
- Test edge cases and error conditions
- Verify tool interactions and state management
- Debug issues in real-time

This immediate feedback loop makes MCP server development much more efficient and helps catch issues early in the development process.

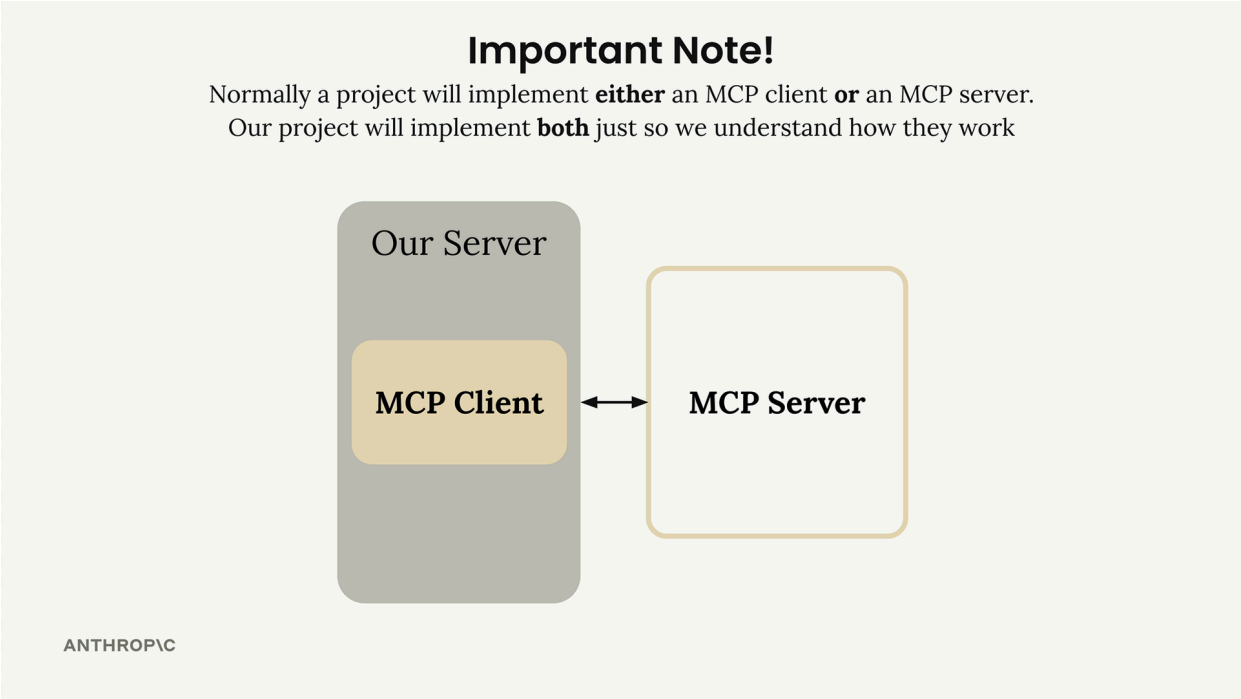
Connecting with MCP clients

Implementing a client

Now that we have our MCP server working, it's time to build the client side. The client is what allows our application code to communicate with the MCP server and access its functionality.

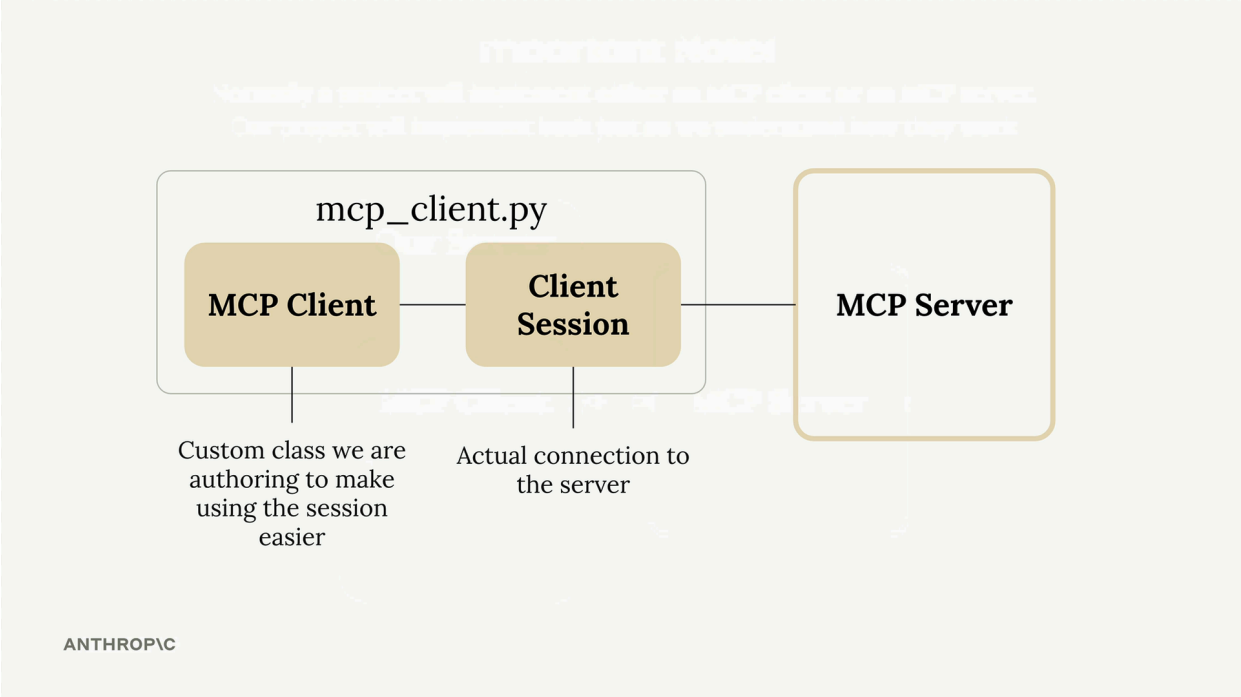
Understanding the Client Architecture

In most real-world projects, you'll either implement an MCP client or an MCP server - not both. We're building both in this project just so you can see how they work together.



The MCP client consists of two main components:

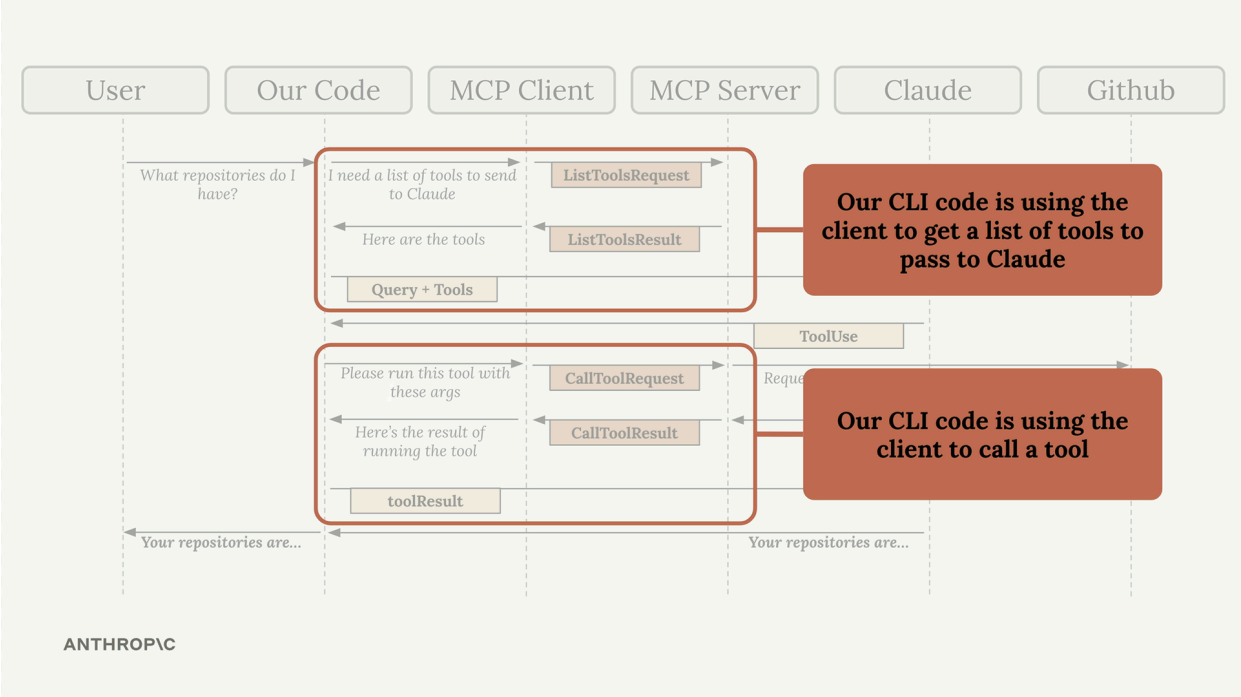
- MCP Client - A custom class we create to make using the session easier
- Client Session - The actual connection to the server (part of the MCP Python SDK)



The client session requires careful resource management - we need to properly clean up connections when we're done. That's why we wrap it in our own class that handles all the cleanup automatically.

How the Client Fits Into Our Application

Remember our application flow diagram? The client is what enables our code to interact with the MCP server at two key points:



Our CLI code uses the client to:

- Get a list of available tools to send to Claude
- Execute tools when Claude requests them

Implementing Core Client Functions

We need to implement two essential functions: **list_tools()** and **call_tool()**.

List Tools Function

This function gets all available tools from the MCP server:

```
async def list_tools(self) -> list[types.Tool]:
    result = await self.session().list_tools()
    return result.tools
```

It's straightforward - we access our session (the connection to the server), call the built-in **list_tools()** method, and return the tools from the result.

Call Tool Function

This function executes a specific tool on the server:

```
async def call_tool(
    self, tool_name: str, tool_input: dict
) -> types.CallToolResult | None:
    return await self.session().call_tool(tool_name, tool_input)
```

We pass the tool name and input parameters (provided by Claude) to the server and return the result.

Testing the Client

The client file includes a simple test harness at the bottom. You can run it directly to verify everything works:

```
uv run mcp_client.py
```

This will connect to your MCP server and print out the available tools. You should see output showing your tool definitions, including descriptions and input schemas.

Putting It All Together

Once the client functions are implemented, you can test the complete flow by running your main application:

```
uv run main.py
```

Try asking: "What is the contents of the report.pdf document?"

Here's what happens behind the scenes:

1. Your application uses the client to get available tools
2. These tools are sent to Claude along with your question
3. Claude decides to use the read_doc_contents tool
4. Your application uses the client to execute that tool
5. The result is returned to Claude, who then responds to you

The client acts as the bridge between your application logic and the MCP server's functionality, making it easy to integrate powerful tools into your AI workflows.

Defining resources

Resources in MCP servers allow you to expose data to clients, similar to GET request handlers in a typical HTTP server. They're perfect for scenarios where you need to fetch information rather than perform actions.

Understanding Resources Through an Example

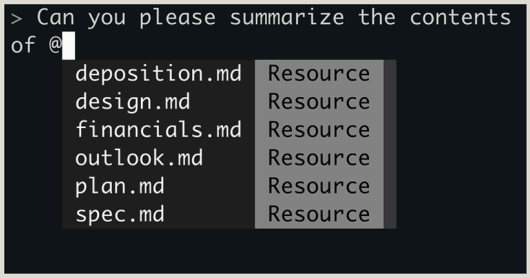
Let's say you want to build a document mention feature where users can type **@document_name** to reference files. This requires two operations:

- Getting a list of all available documents (for autocomplete)
- Fetching the contents of a specific document (when mentioned)

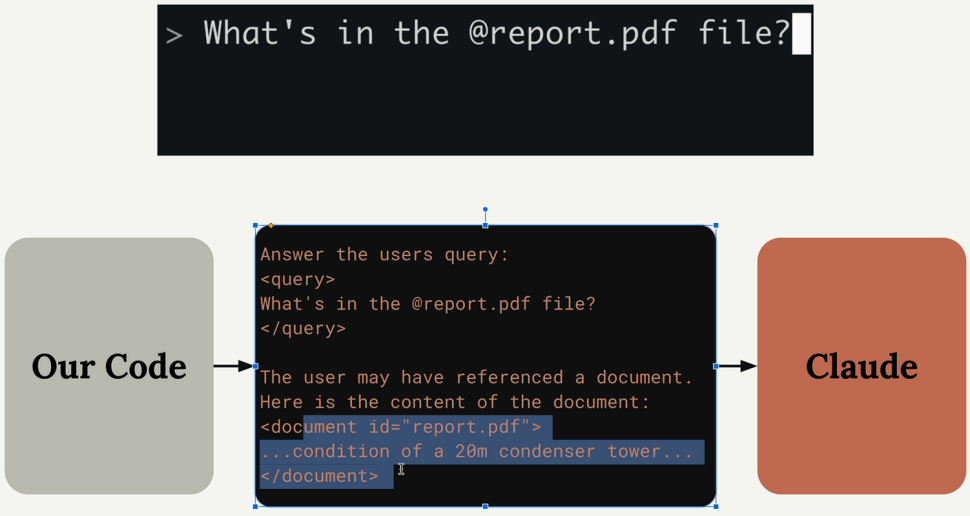
Next Feature

- Users can “mention” a document by writing out “@doc_name”
  - Typing “@” should show a list of all the available documents
  - When a document is mentioned, its contents should be automatically injected into the prompt

ANTHROP\C



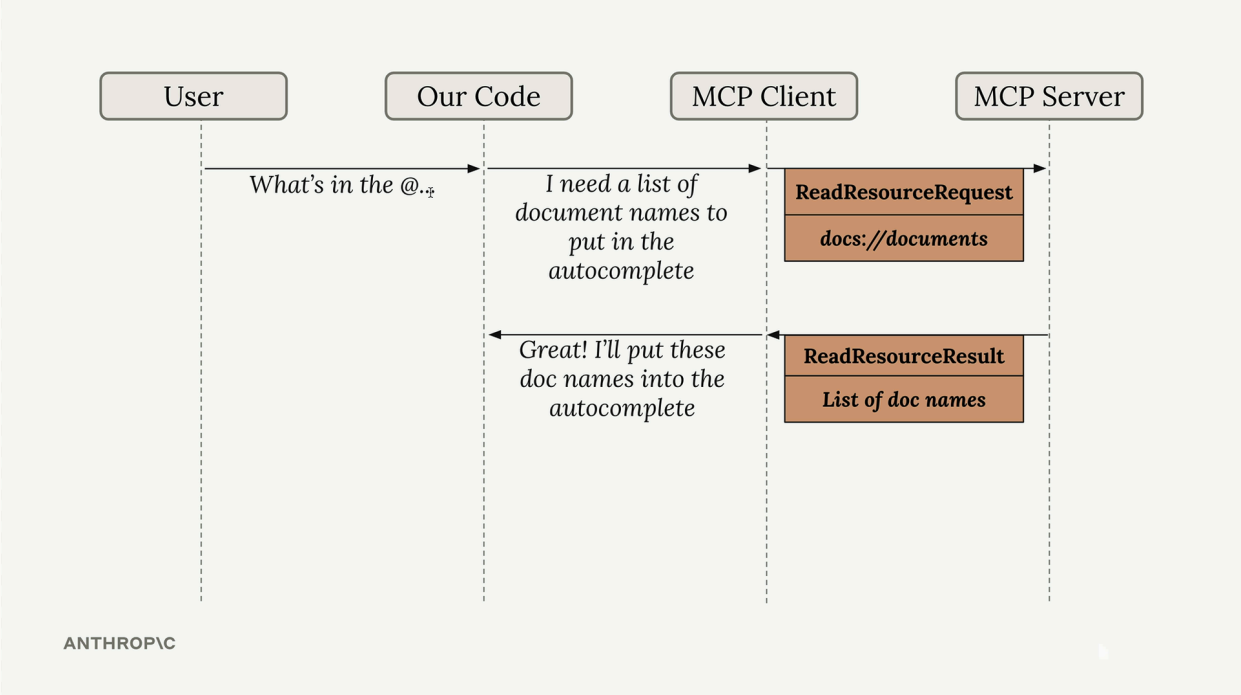
When a user mentions a document, your system automatically injects the document's contents into the prompt sent to Claude, eliminating the need for Claude to use tools to fetch the information.



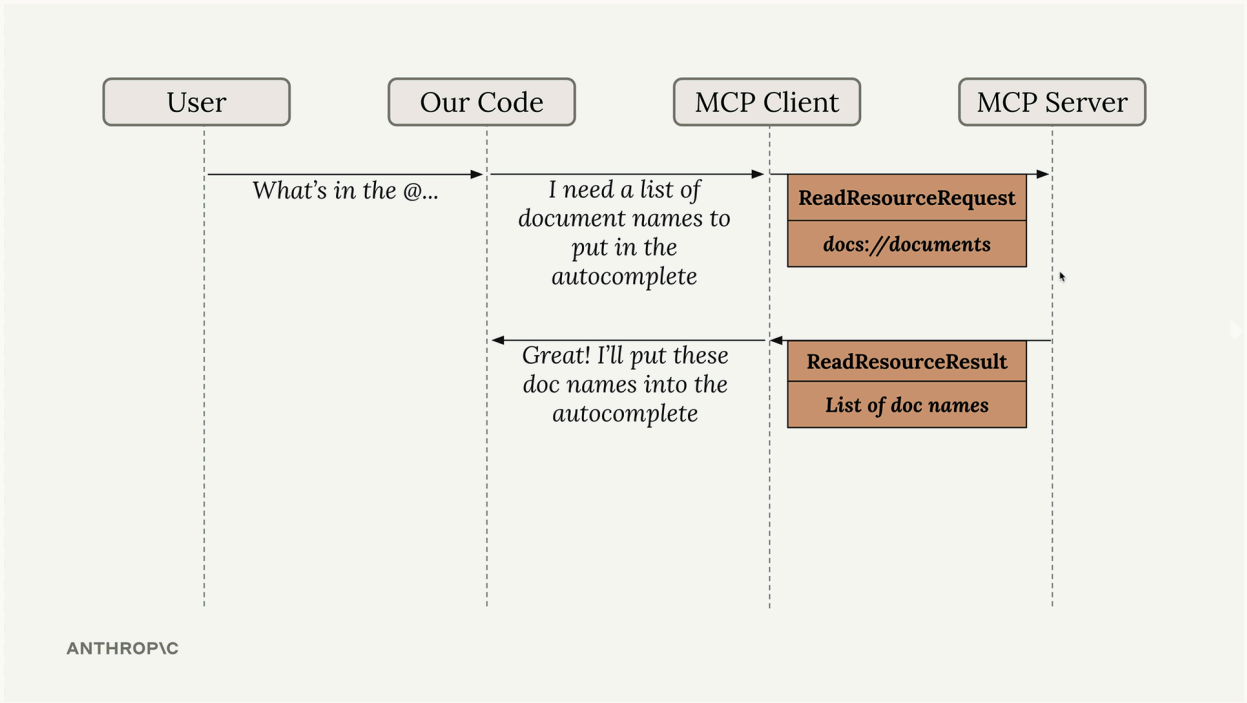
ANTHROP\C

How Resources Work

Resources follow a request-response pattern. When your client needs data, it sends a **ReadResourceRequest** with a URI to identify which resource it wants. The MCP server processes this request and returns the data in a **ReadResourceResult**.



The flow looks like this: your code requests a resource from the MCP client, which forwards the request to the MCP server. The server processes the URI, runs the appropriate function, and returns the result.



Types of Resources

There are two types of resources:

Direct Resources

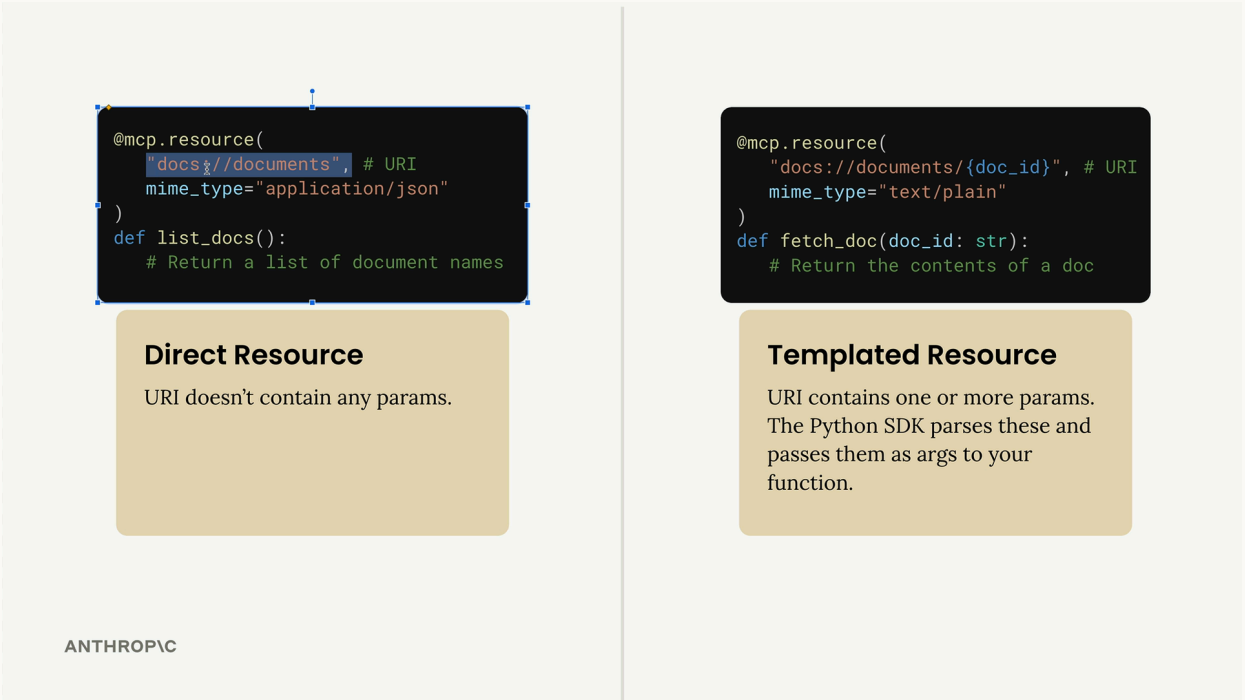
Direct resources have static URIs that never change. They're perfect for operations that don't need parameters.

```
@mcp.resource(
    "docs://documents",
    mime_type="application/json"
)
def list_docs() -> list[str]:
    return list(docs.keys())
```

Templated Resources

Templated resources include parameters in their URIs. The Python SDK automatically parses these parameters and passes them as keyword arguments to your function.

```
@mcp.resource(
    "docs://documents/{doc_id}",
    mime_type="text/plain"
)
def fetch_doc(doc_id: str) -> str:
    if doc_id not in docs:
        raise ValueError(f"Doc with id {doc_id} not found")
    return docs[doc_id]
```



Implementation Details

Resources can return any type of data - strings, JSON, binary data, etc. Use the **mime_type** parameter to give clients a hint about what kind of data you're returning:

- **"application/json"** for structured data
- **"text/plain"** for plain text
- **"application/pdf"** for binary files

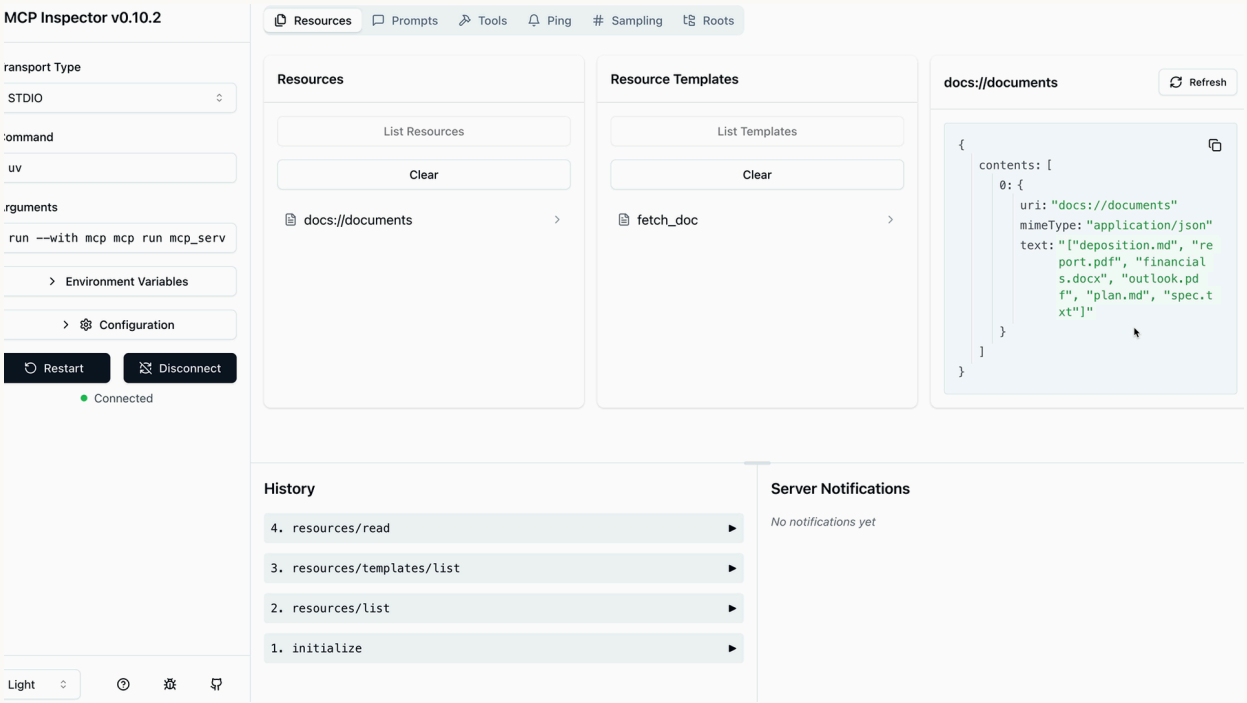
The MCP Python SDK automatically serializes your return values. You don't need to manually convert objects to JSON strings - just return the data structure and let the SDK handle serialization.

Testing Your Resources

You can test resources using the MCP Inspector. Start your server with:

uv run mcp dev mcp_server.py
Then connect to the inspector in your browser. You'll see two sections:

- Resources - Lists your direct/static resources
- Resource Templates - Lists your templated resources

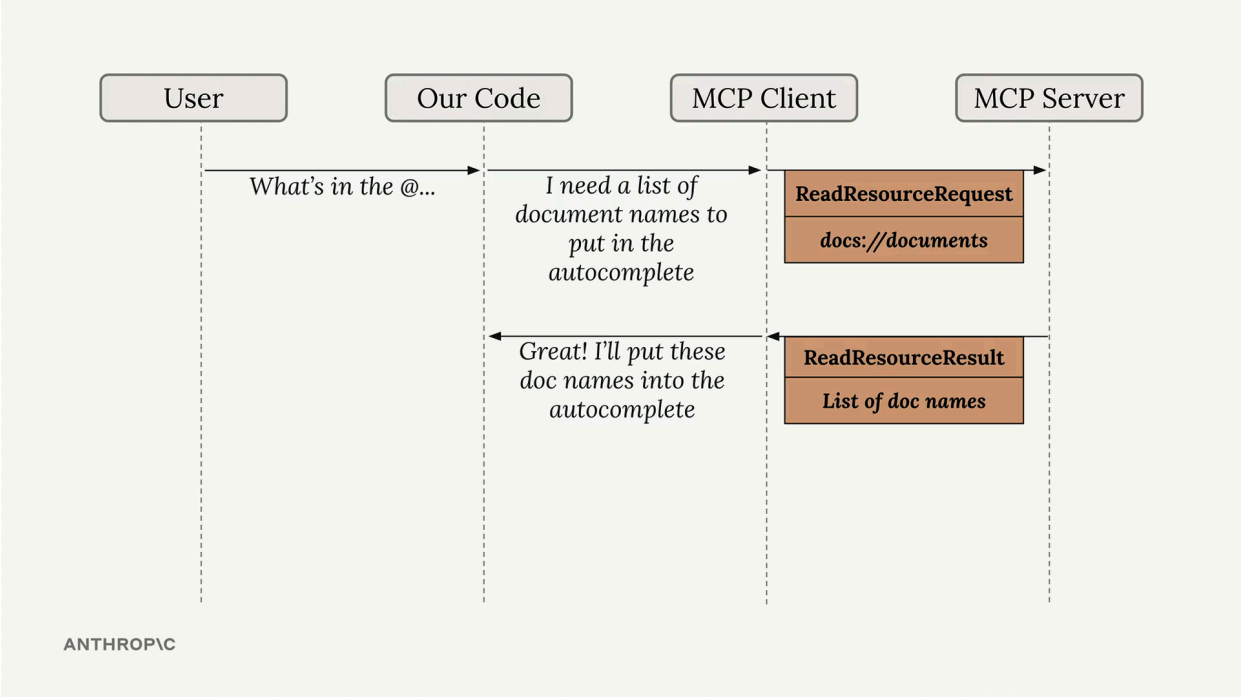


Click on any resource to test it. For templated resources, you'll need to provide values for the parameters. The inspector shows you the exact response structure your client will receive, including the MIME type and serialized data.

Resources provide a clean way to expose read-only data from your MCP server, making it easy for clients to fetch information without the complexity of tool calls.

Accessing resources

Resources in MCP allow your server to expose information that can be directly included in prompts, rather than requiring tool calls to access data. This creates a more efficient way to provide context to AI models.



The diagram above shows how resources work: when a user types something like "What's in the @..." our code recognizes this as a resource request, sends a ReadResourceRequest to the MCP server, and gets back a ReadResourceResult with the actual content.

Implementing Resource Reading

To enable resource access in your MCP client, you need to implement a **read_resource** function. First, add the necessary imports:

```
import json
from pydantic import AnyUrl
```

The core function makes a request to the MCP server and processes the response based on its MIME type:

```
async def read_resource(self, uri: str) -> Any:
    result = await self.session().read_resource(AnyUrl(uri))
```

```
    resource = result.contents[0]

    if isinstance(resource, types.TextResourceContents):
        if resource.mimeType == "application/json":
            return json.loads(resource.text)

    return resource.text
```

Understanding the Response Structure

When you request a resource, the server returns a result with a **contents** list. We access the first element since we typically only need one resource at a time. The response includes:

- The actual content (text or data)
- A MIME type that tells us how to parse the content
- Other metadata about the resource

Content Type Handling

The function checks the MIME type to determine how to process the content:

- If it's **application/json**, parse the text as JSON and return the parsed object
- Otherwise, return the raw text content

This approach handles both structured data (like JSON) and plain text documents seamlessly.

Testing Resource Access

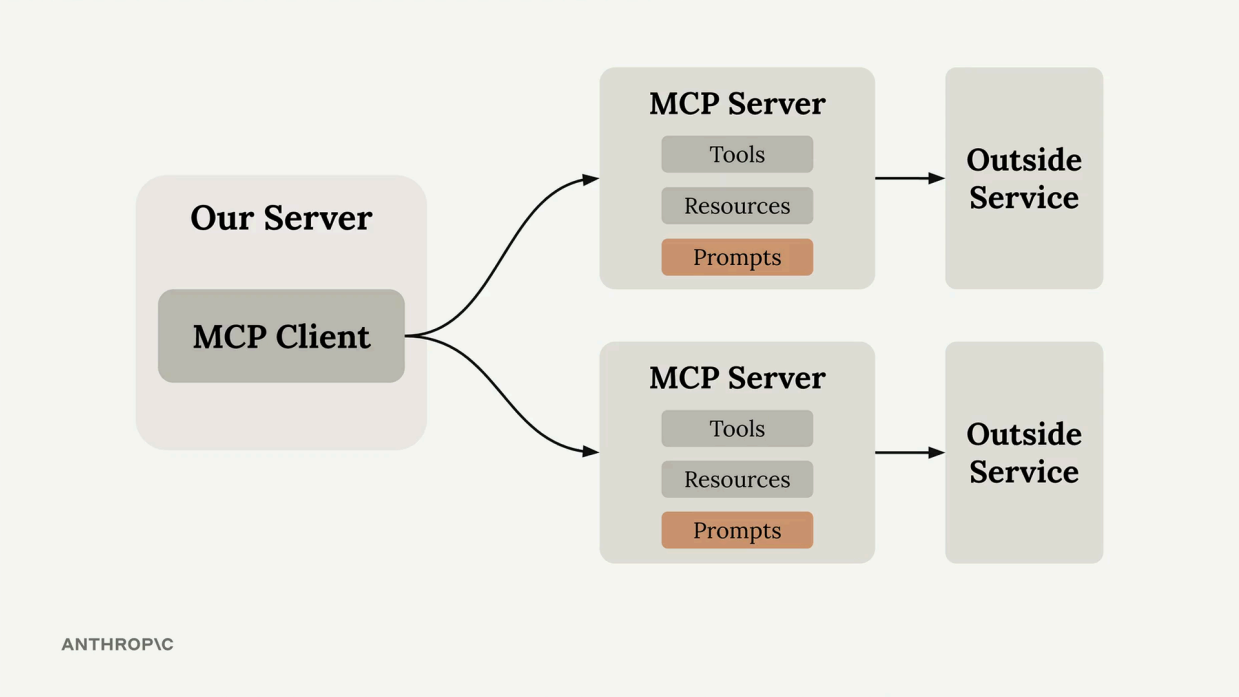
Once implemented, you can test the resource functionality through your CLI application. When you type "@" followed by a resource name, the system will:

1. Show available resources in an autocomplete list
2. Let you select a resource using arrow keys and space
3. Include the resource content directly in your prompt
4. Send everything to the AI model without requiring additional tool calls

This creates a much smoother user experience compared to having the AI model make separate tool calls to access document contents. The resource content becomes part of the initial context, allowing for immediate responses about the data.

Defining prompts

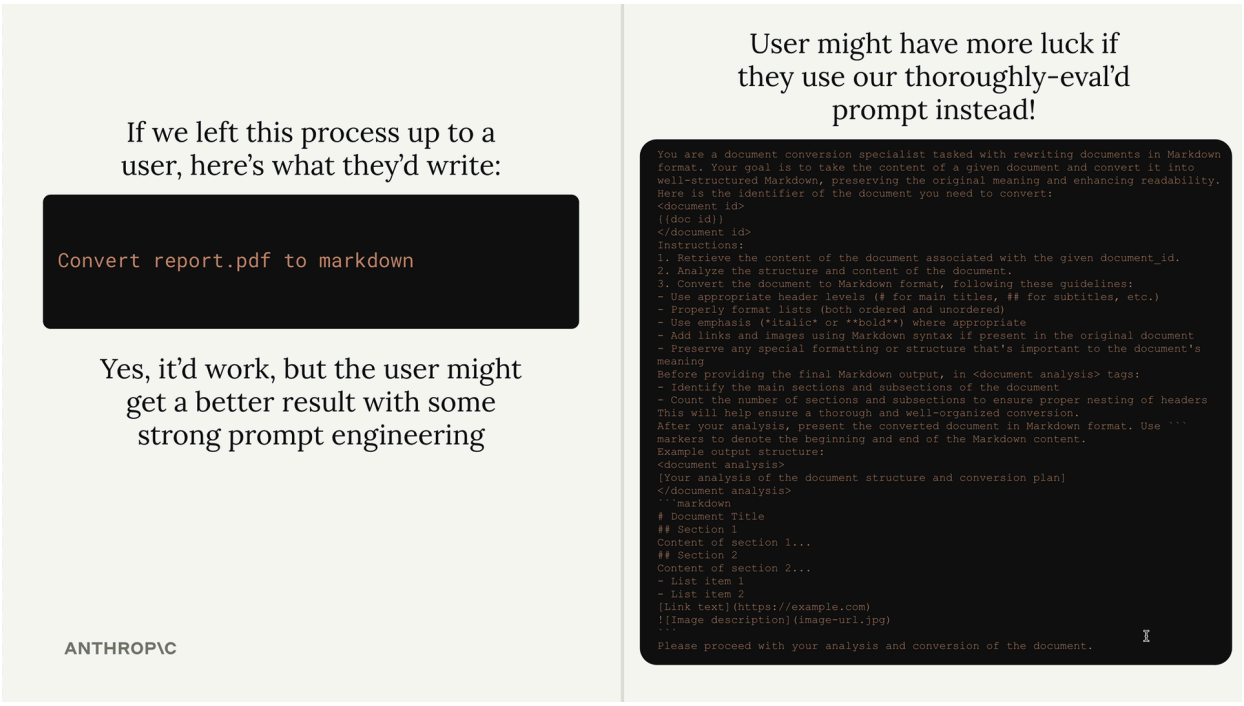
Prompts in MCP servers let you define pre-built, high-quality instructions that clients can use instead of writing their own prompts from scratch. Think of them as carefully crafted templates that give better results than what users might come up with on their own.



Why Use Prompts?

Here's the key insight: users can already ask Claude to do most tasks directly. For example, a user could type "reformat the report.pdf in markdown" and get decent results. But they'll get much better results if you provide a thoroughly tested, specialized prompt that handles edge cases and follows best practices.

As the MCP server author, you can spend time crafting, testing, and evaluating prompts that work consistently across different scenarios. Users benefit from this expertise without having to become prompt engineering experts themselves.



Building a Format Command

Let's implement a practical example: a format command that converts documents to markdown. Users will type **/format doc_id** and get back a professionally formatted markdown version of their document.

The workflow looks like this:

- User types **/** to see available commands
- They select **format** and specify a document ID
- Claude uses your pre-built prompt to read and reformat the document
- The result is clean markdown with proper headers, lists, and formatting

Defining Prompts

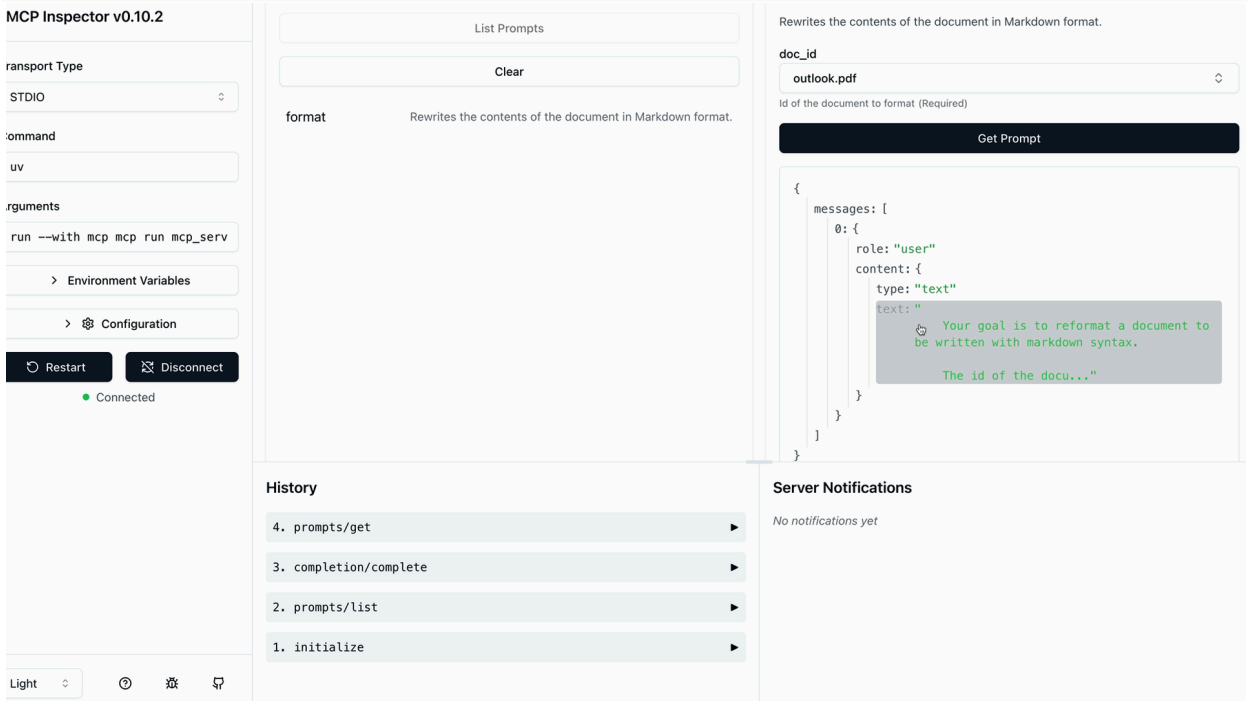
Prompts use a similar decorator pattern to tools and resources:

```
@mcp.prompt(
    name="format",
    description="Rewrites the contents of the document in Markdown format."
)
def format_document(
    doc_id: str = Field(description="Id of the document to format")
) -> list[base.Message]:
    prompt = f"""
Your goal is to reformat a document to be written with markdown syntax.

The id of the document you need to reformat is:
<document_id>
{doc_id}
</document_id>

Add in headers, bullet points, tables, etc as necessary. Feel free to add in structure.
Use the 'edit_document' tool to edit the document. After the document has been reformatted...
"""

    return [
        base.UserMessage(prompt)
    ]
```

The function returns a list of messages that get sent directly to Claude. You can include multiple user and assistant messages to create more complex conversation flows.

Testing Your Prompts

Use the MCP Inspector to test your prompts before deploying them:



The inspector shows you exactly what messages will be sent to Claude, including how variables get interpolated into your prompt template. This lets you verify the prompt looks correct before users start relying on it.

Key Benefits

- Consistency - Users get reliable results every time
- Expertise - You can encode domain knowledge into prompts
- Reusability - Multiple client applications can use the same prompts
- Maintenance - Update prompts in one place to improve all clients

Prompts work best when they're specialized for your MCP server's domain. A document management server might have prompts for formatting, summarizing, or analyzing documents. A data analysis server might have prompts for generating reports or visualizations.

The goal is to provide prompts that are so well-crafted and tested that users prefer them over writing their own instructions from scratch.

Prompts in the client

The final step in building our MCP client is implementing prompt functionality. This allows us to list all available prompts from the server and retrieve specific prompts with variables filled in.

Implementing List Prompts

The **list_prompts** method is straightforward. It calls the session's list prompts function and returns the prompts:

```
async def list_prompts(self) -> list[types.Prompt]:
    result = await self.session().list_prompts()
    return result.prompts
```

Getting Individual Prompts

The **get_prompt** method is more interesting because it handles variable interpolation. When you request a prompt, you provide arguments that get passed to the prompt function as keyword arguments:

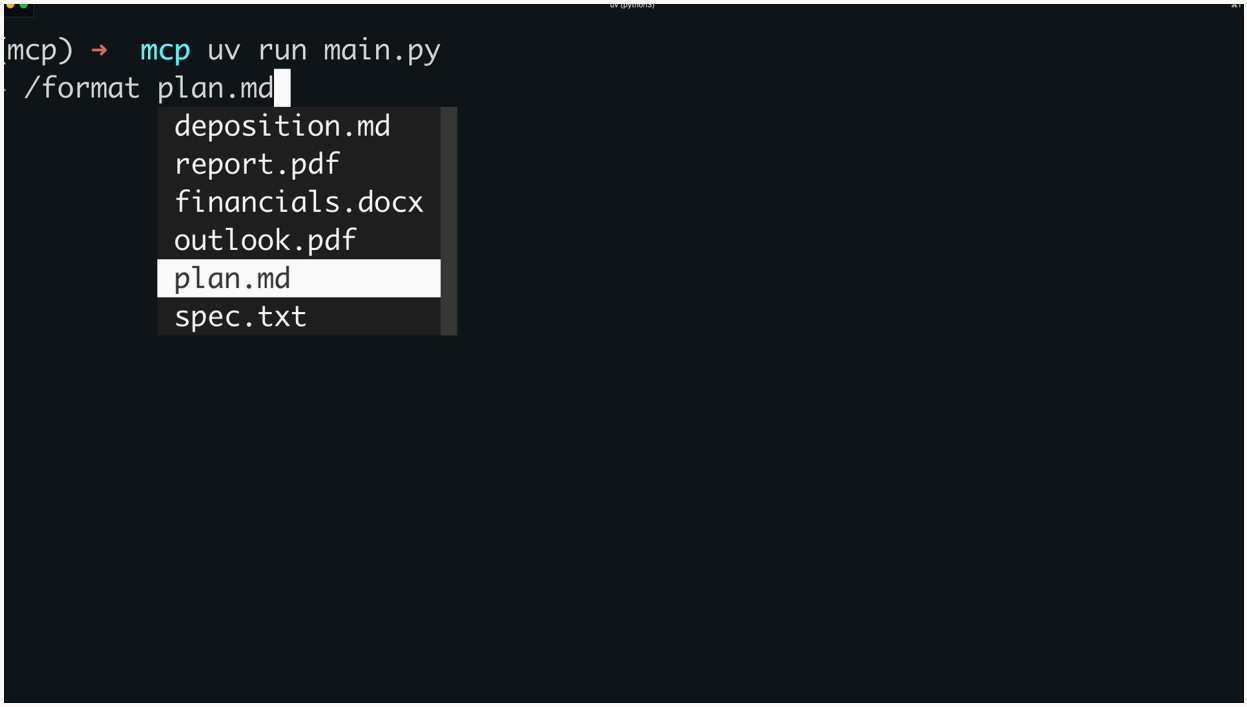
```
async def get_prompt(self, prompt_name, args: dict[str, str]):
```

```
    result = await self.session().get_prompt(prompt_name, args)
    return result.messages
```

For example, if your server has a **format_document** prompt that expects a **doc_id** parameter, the arguments dictionary would contain **{"doc_id": "plan.md"}**. This value gets interpolated into the prompt template.

Testing Prompts in Action

Once implemented, you can test prompts through the CLI. When you type a slash (/), available prompts appear as commands. Selecting a prompt like "format" will prompt you to choose from available documents.



After selecting a document, the system sends the complete prompt to Claude. The AI receives both the formatting instructions and the document ID, then uses available tools to fetch and process the content.

How Prompts Work

Prompts define a set of user and assistant messages that clients can use. They should be high-quality, well-tested, and relevant to your MCP server's purpose. The workflow is:

- Write and evaluate a prompt relevant to your server's functionality
- Define the prompt in your MCP server using the **@mcp.prompt** decorator
- Clients can request the prompt at any time
- Arguments provided by the client become keyword arguments in your prompt function
- The function returns formatted messages ready for the AI model

This system creates reusable, parameterized prompts that maintain consistency while allowing customization through variables. It's particularly useful for complex workflows where you want to ensure the AI receives properly structured instructions every time.

Final assessment on MCP

Question 1: Correct answer

You're building an MCP client to connect your application to an MCP server. What are the two main components you need?

~~A frontend and a backend~~

~~A database and a web server~~

~~A REST API and a GraphQL endpoint~~

An MCP Client class and a Client Session

Question 2: Correct answer

Your MCP client needs to find out what tools are available from an MCP server. What message type should it send?

ListToolsRequest

~~CallToolRequest~~

~~ToolDiscoveryRequest~~

~~GetToolsMessage~~

Question 3: Correct answer

You've built an MCP server and want to test if your tools work correctly before connecting to a full application. What's the easiest way to do this?

~~Write separate test scripts for each tool~~

~~Test everything manually in the terminal~~

Use the built-in MCP Inspector with `mcp dev mcp_server.py`

~~Connect directly to Claude first~~

Question 4: Correct answer

You're building a chat app where users ask Claude about their GitHub data. Without MCP, what's the main problem you'd face?

~~GitHub doesn't allow API access~~

You'd have to write and maintain all the GitHub tool code yourself

~~Claude can't understand GitHub data~~

~~Users can't ask questions about repositories~~

Question 5: Incorrect answer

You're deciding how to implement a new feature in your MCP server. Users should be able to click a button to trigger a "summarize document" workflow. Which MCP primitive should you use?

~~Resources - because you need to fetch document data~~

~~Functions - because it involves processing~~

Prompts - because users control when to start the workflow

~~Tools - because the AI needs new capabilities~~

Question 6: Correct answer

You're using the Python MCP SDK to create a tool that reads files. What's the easiest way to define this tool?

Use the @mcp.tool() decorator on a Python function

~~Create a separate configuration file~~

~~Write JSON schemas manually~~

~~Send HTTP requests to register the tool~~

Question 7: Correct answer

You want to create a resource that fetches different documents based on their ID, like docs://documents/report.pdf. What type of resource should you use?

A templated resource with parameters in the URI

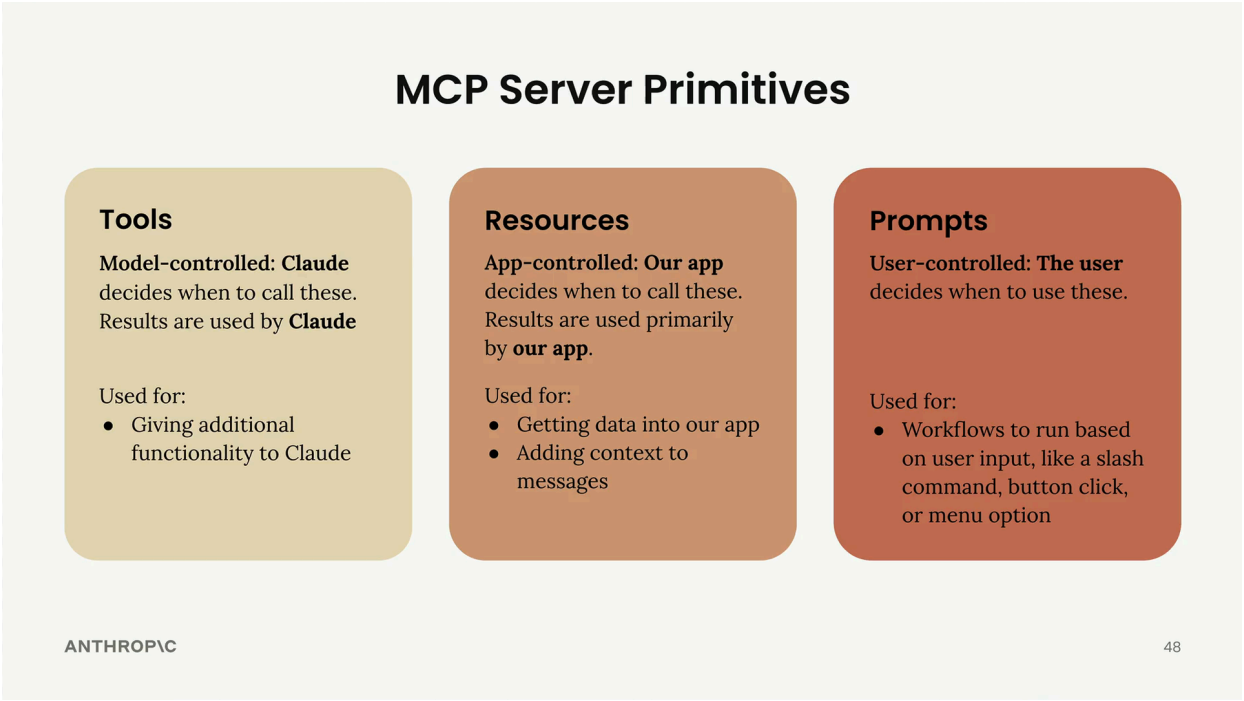
~~A tool instead of a resource~~

~~A direct resource with a static URI~~

~~A database query resource~~

MCP review

Now that we've built our MCP server, let's review the three core server primitives and understand when to use each one. The key insight is that each primitive is controlled by a different part of your application stack.



Tools: Model-Controlled

Tools are controlled entirely by Claude. The AI model decides when to call these functions, and the results are used directly by Claude to accomplish tasks.

Tools are perfect for giving Claude additional capabilities it can use autonomously. When you ask Claude to "calculate the square root of 3 using JavaScript," it's Claude that decides to use a JavaScript execution tool to run the calculation.

Resources: App-Controlled

Resources are controlled by your application code. Your app decides when to fetch resource data and how to use it - typically for UI elements or to add context to conversations.

In our project, we used resources in two ways:

- Fetching data to populate autocomplete options in the UI
- Retrieving content to augment prompts with additional context

Think of the "Add from Google Drive" feature in Claude's interface - the application code determines which documents to show and handles injecting their content into the chat context.

Prompts: User-Controlled

Prompts are triggered by user actions. Users decide when to run these predefined workflows through UI interactions like button clicks, menu selections, or slash commands.

Prompts are ideal for implementing workflows that users can trigger on demand. In Claude's interface, those workflow buttons below the chat input are examples of prompts - predefined, optimized workflows that users can start with a single click.

Choosing the Right Primitive

Here's a quick decision guide:

- Need to give Claude new capabilities? Use tools

- Need to get data into your app for UI or context? Use resources
- Want to create predefined workflows for users? Use prompts

You can see all three primitives in action in Claude's official interface. The workflow buttons demonstrate prompts, the Google Drive integration shows resources in action, and when Claude executes code or performs calculations, it's using tools behind the scenes.

These are high-level guidelines to help you choose the right primitive for your specific use case. Each serves a different part of your application stack - tools serve the model, resources serve your app, and prompts serve your users.