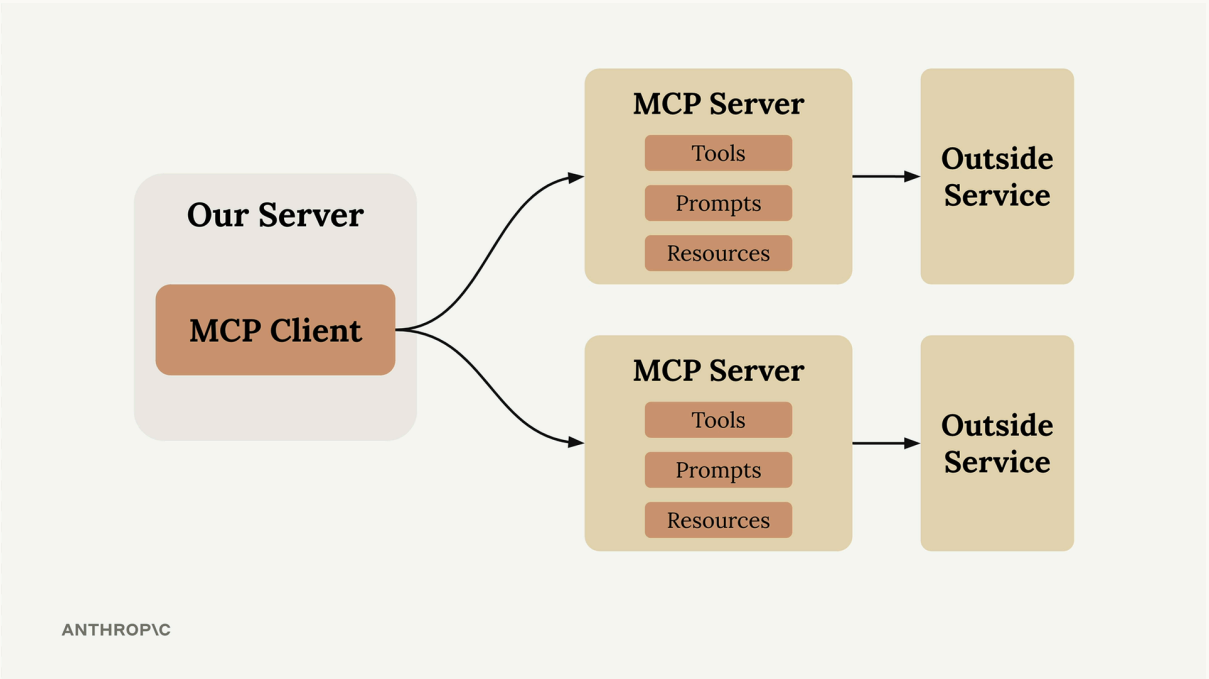
Introducing to MCP

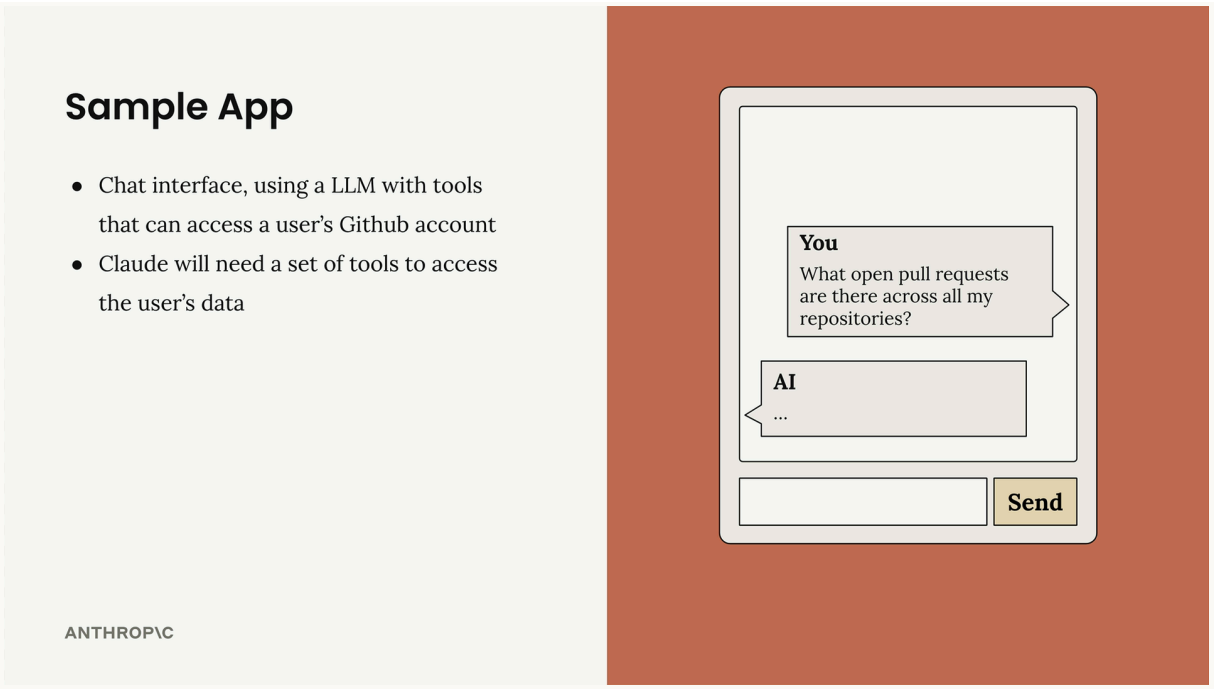
Summary
Model Context Protocol (MCP) is a communication layer that provides Claude with context and tools without requiring you to write a bunch of tedious integration code. Think of it as a way to shift the burden of tool definitions and execution away from your server to specialized MCP servers.



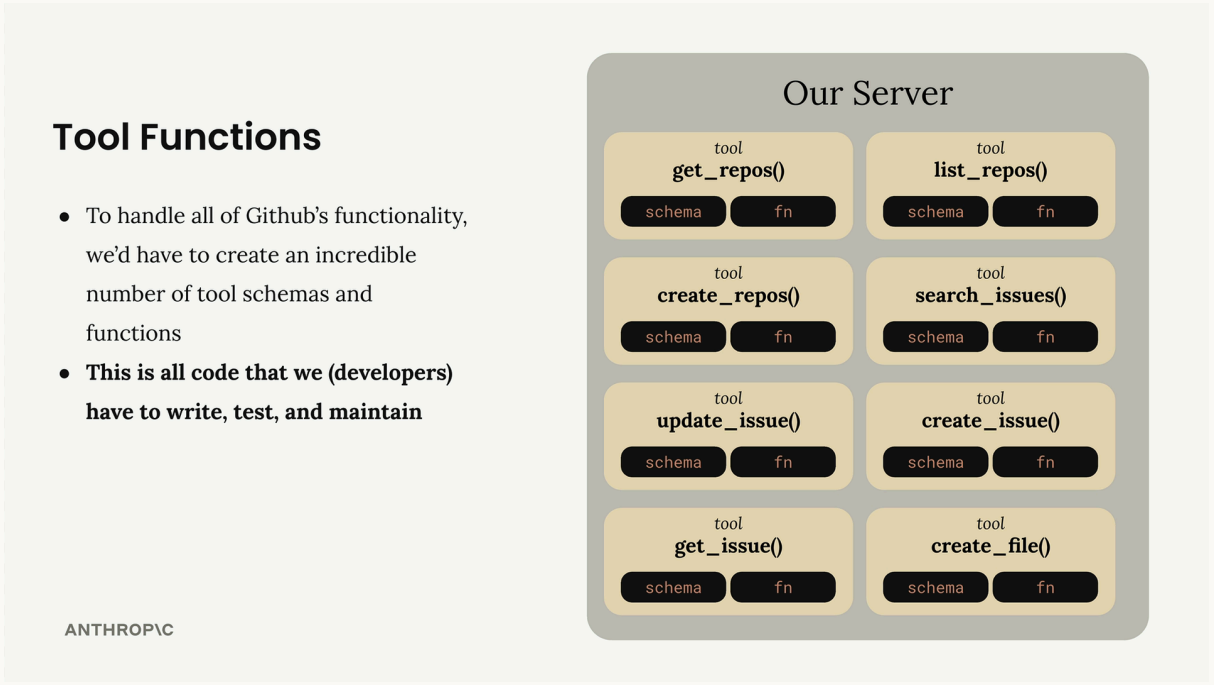
When you first encounter MCP, you'll see diagrams showing the basic architecture: an MCP Client (your server) connecting to MCP Servers that contain tools, prompts, and resources. Each MCP Server acts as an interface to some outside service.

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.



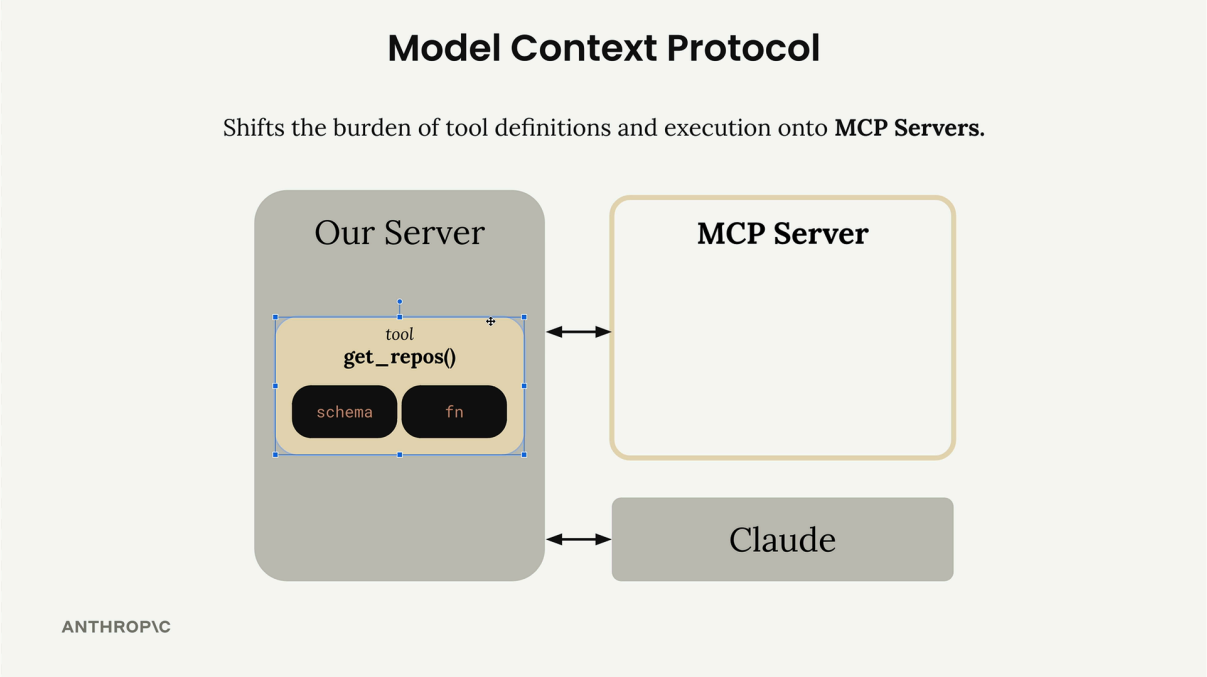
GitHub has massive functionality - repositories, pull requests, issues, projects, and tons more. Without MCP, you'd need to create an incredible number of tool schemas and functions to handle all of GitHub's features.



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

How MCP Works

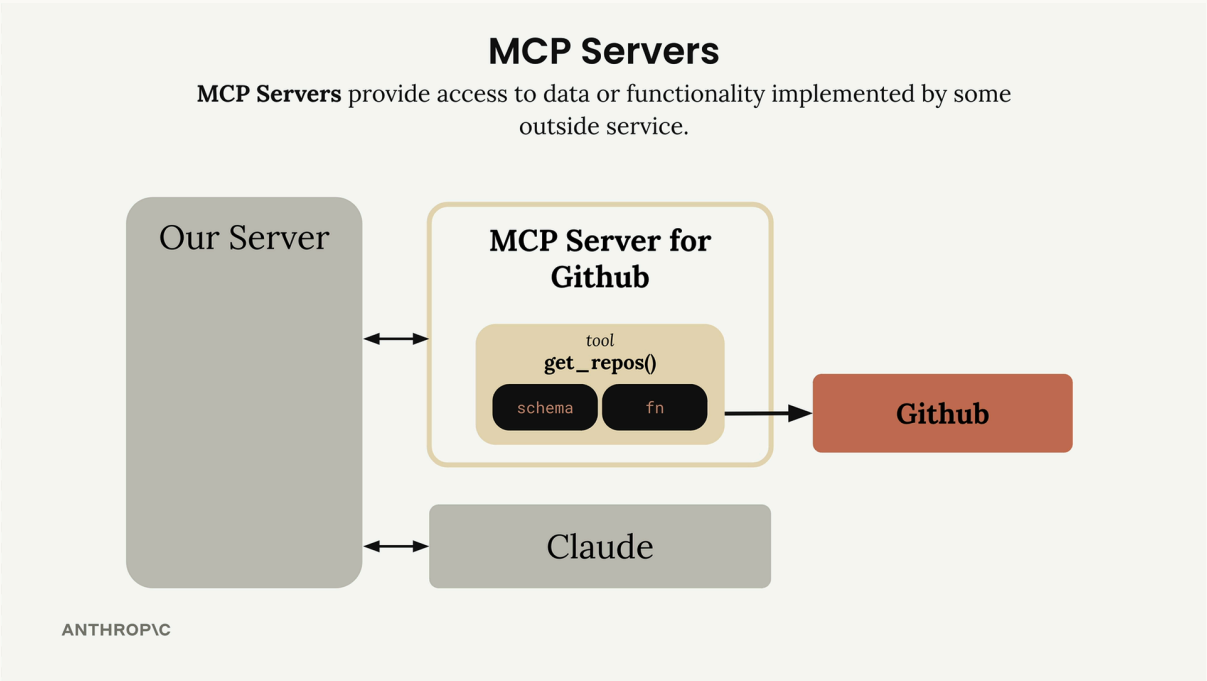
MCP shifts this burden by moving tool definitions and execution from your server to dedicated MCP servers. Instead of you authoring all those GitHub tools, an MCP Server for GitHub handles it.



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.

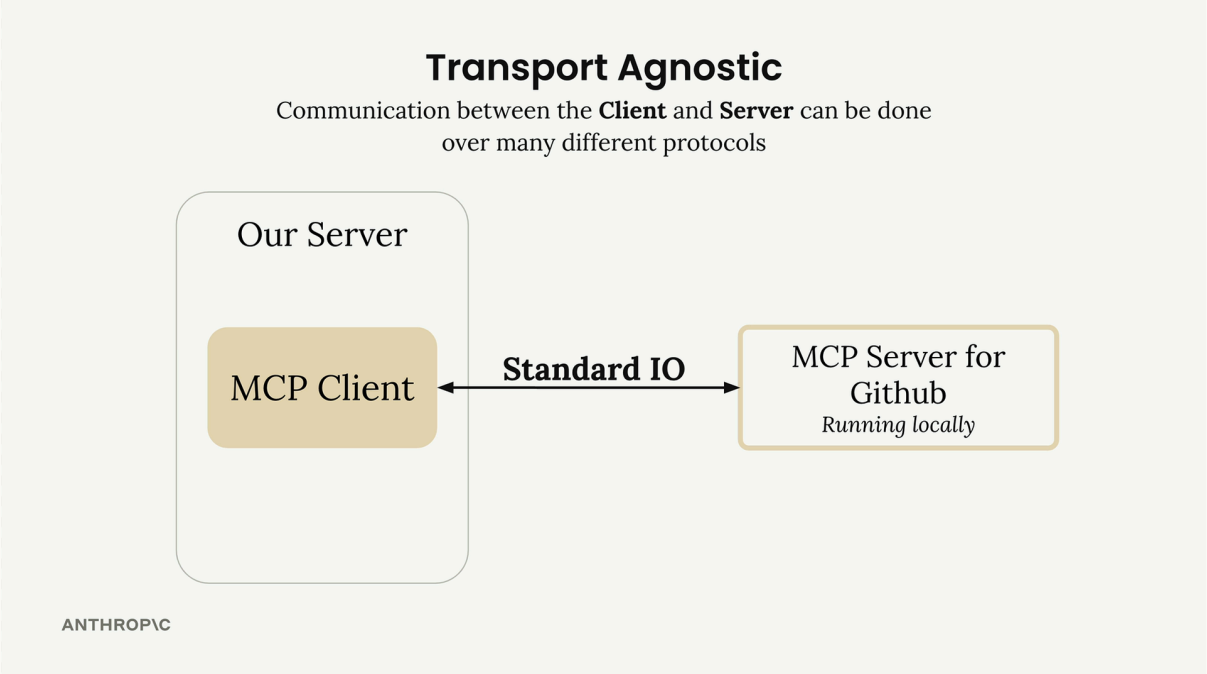
The benefit is clear: instead of maintaining a complex set of integrations yourself, you can leverage MCP servers that handle the heavy lifting of connecting to external services.

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.

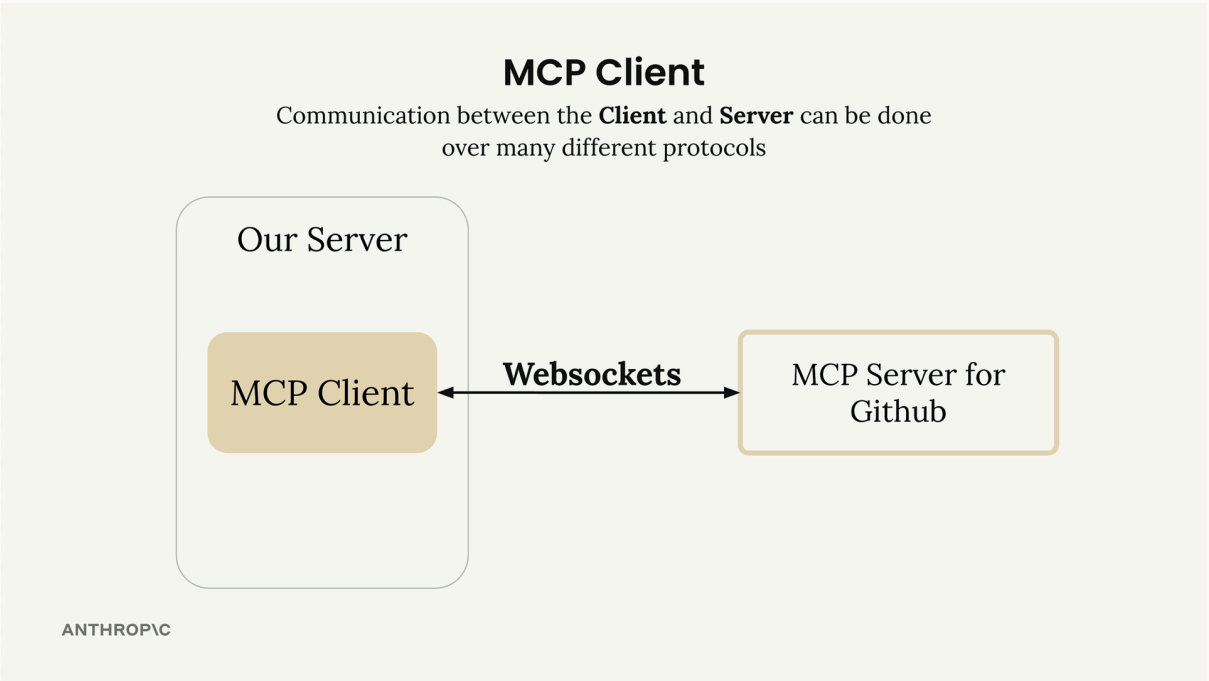
Transport Agnostic Communication

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.



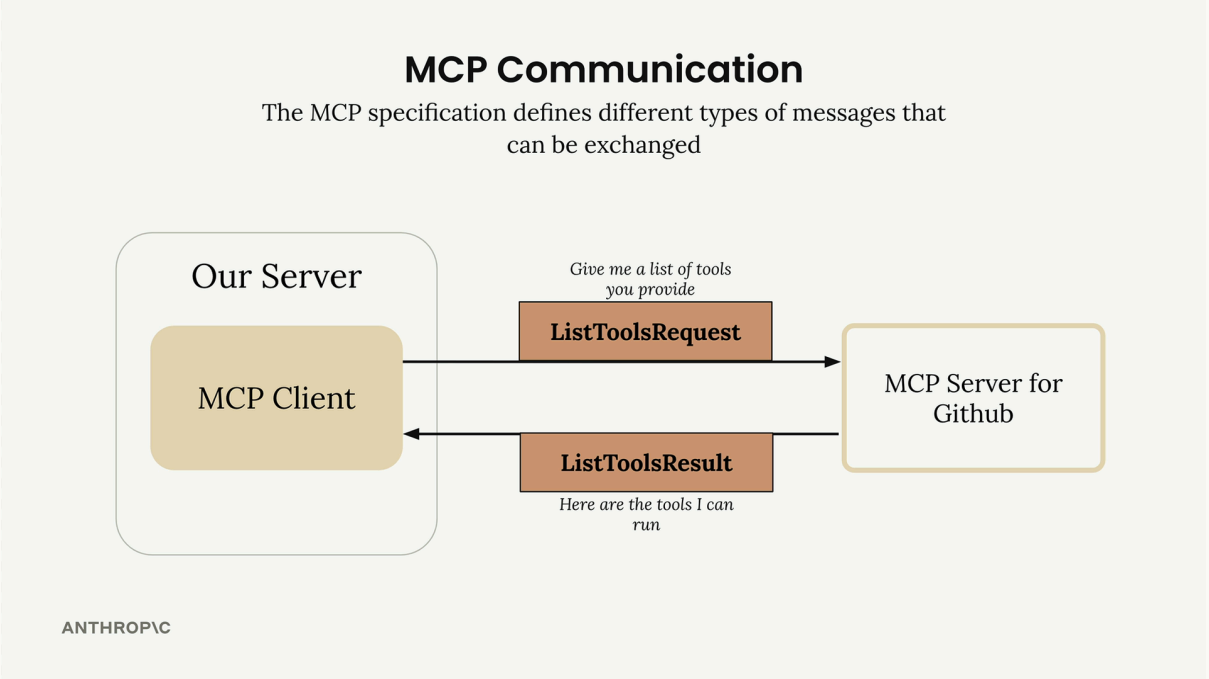
The most common setup runs both the MCP client and server on the same machine, communicating through standard input/output. But you can also connect them over:

- HTTP
- WebSockets
- Various other network protocols

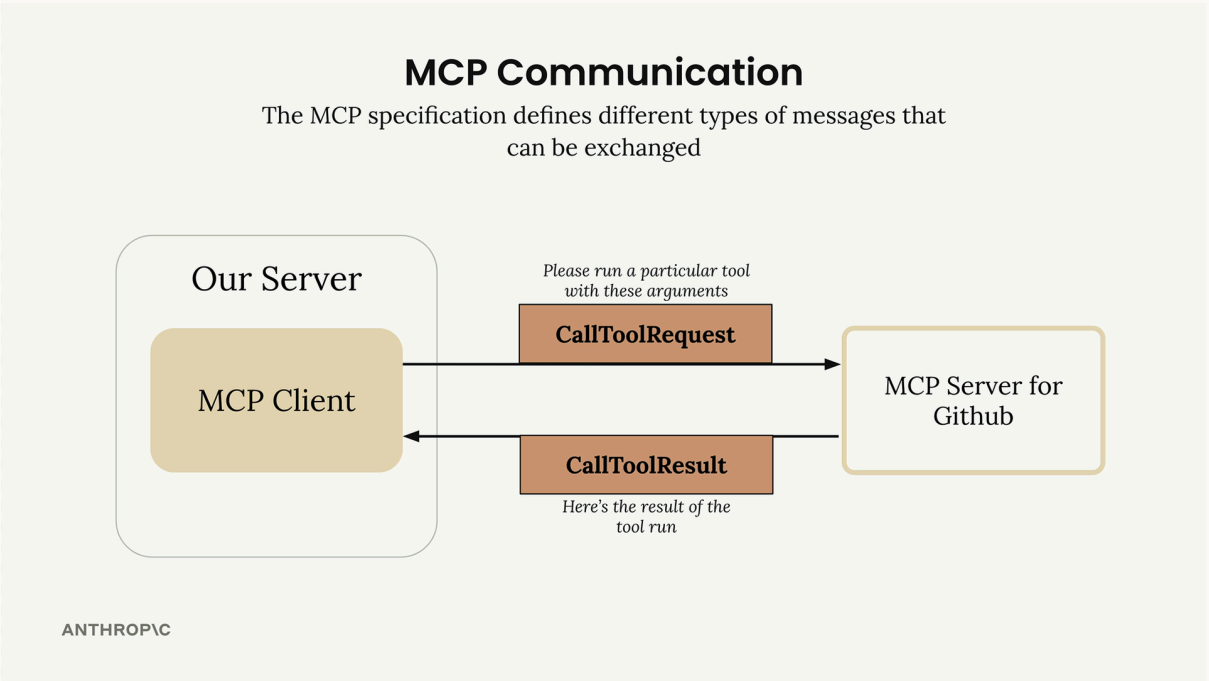


MCP Message Types

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.

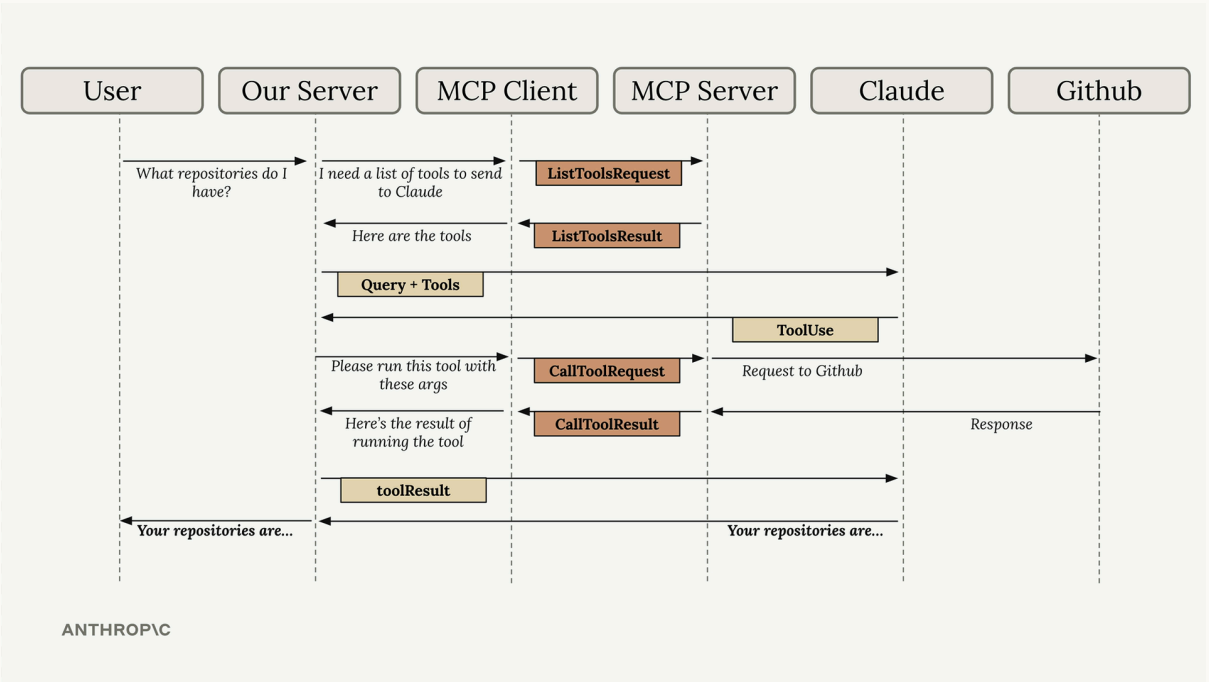
How It All Works Together

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

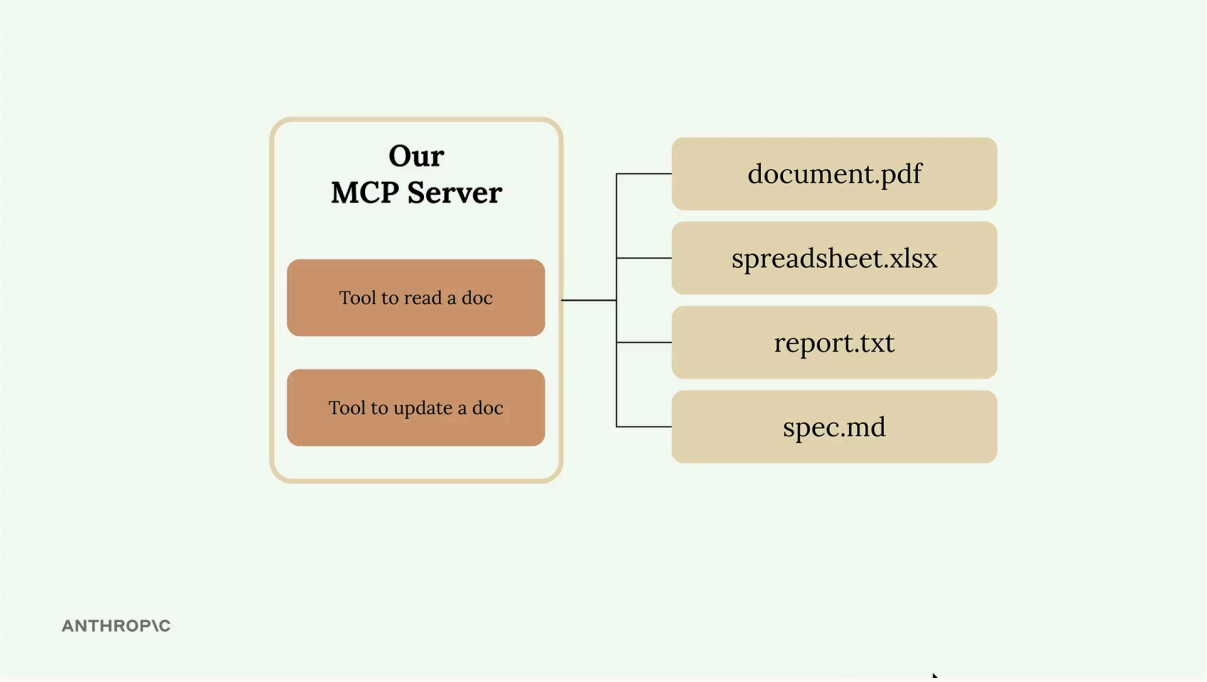
Downloads

cli_project_COMPLETE.zip

Cli_project.zip

Defining tools with MCP

Summary
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

Summary
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

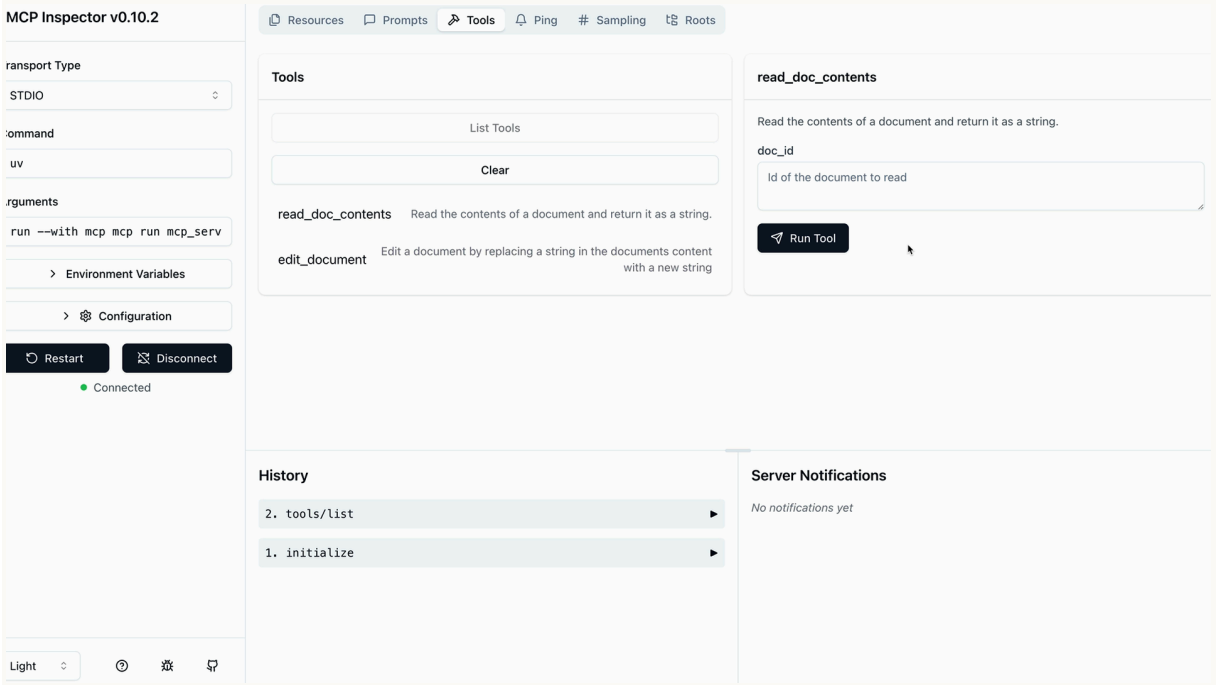
The inspector interface is actively being developed, so it may look different when you use it. However, the core functionality remains consistent. Look for these key elements:

- 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

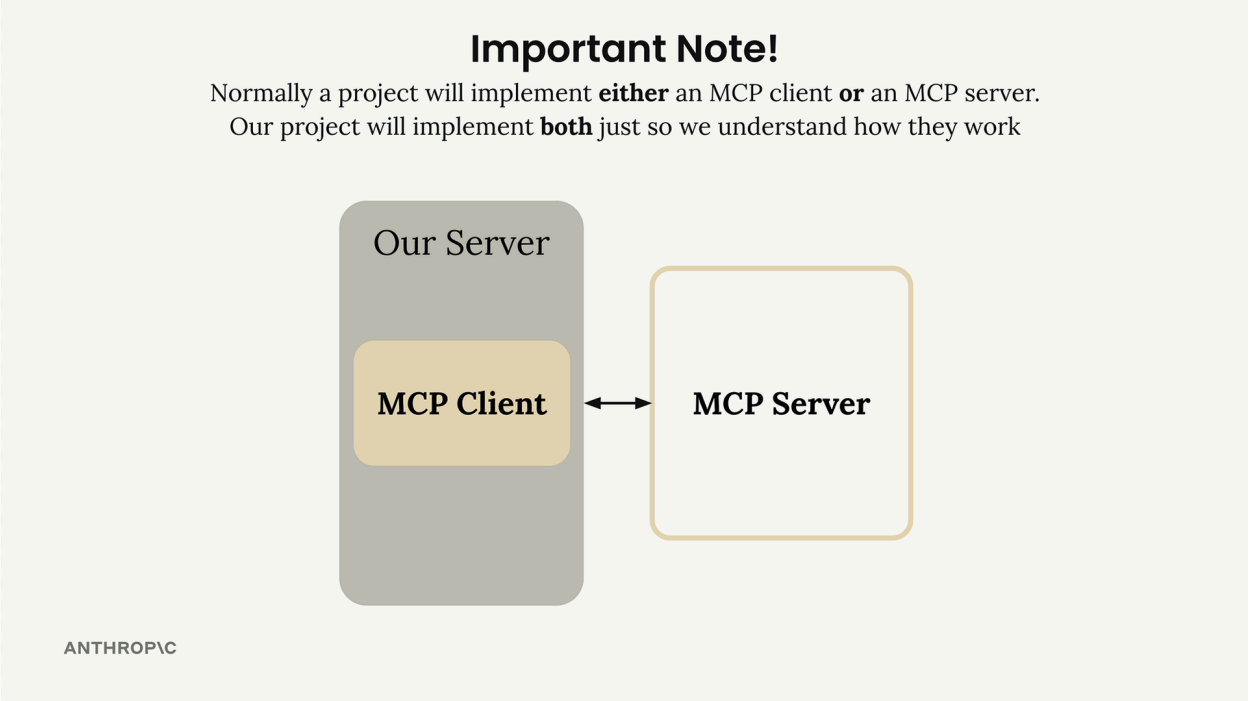
This immediate feedback loop makes MCP server development much more efficient and helps catch issues early in the development process.

Implementing a client

Summary
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.

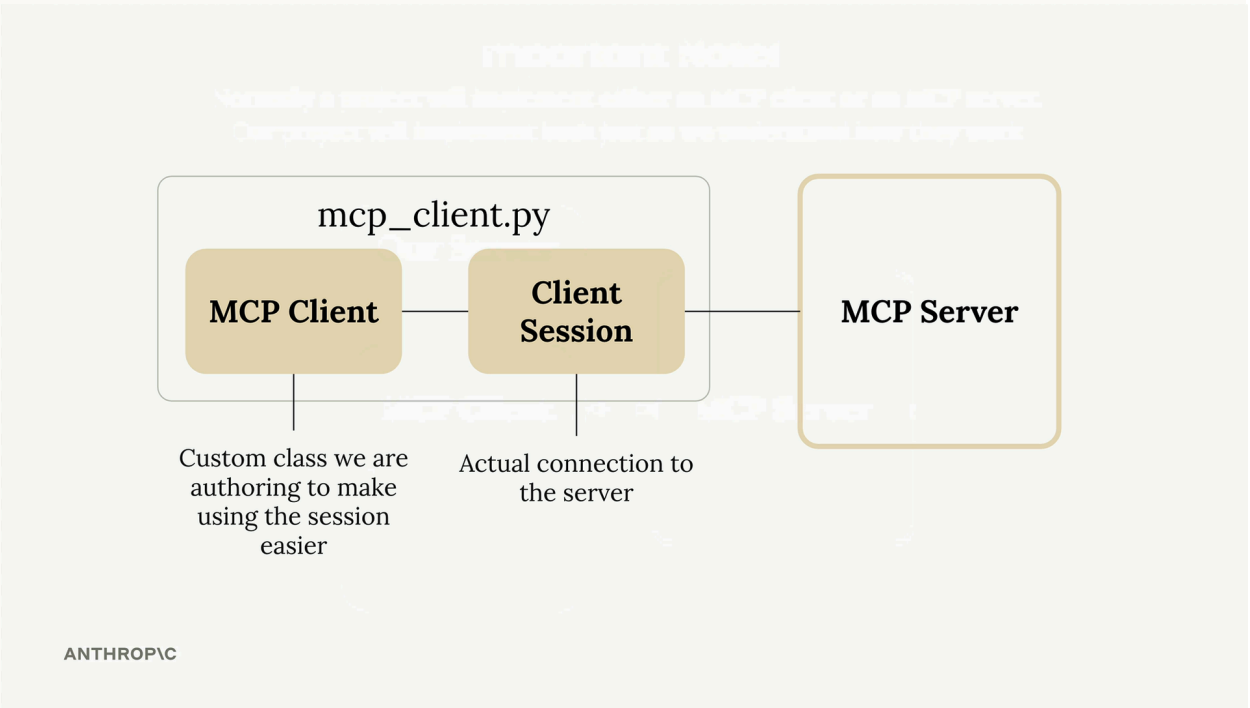
Understanding the Client Architecture

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:

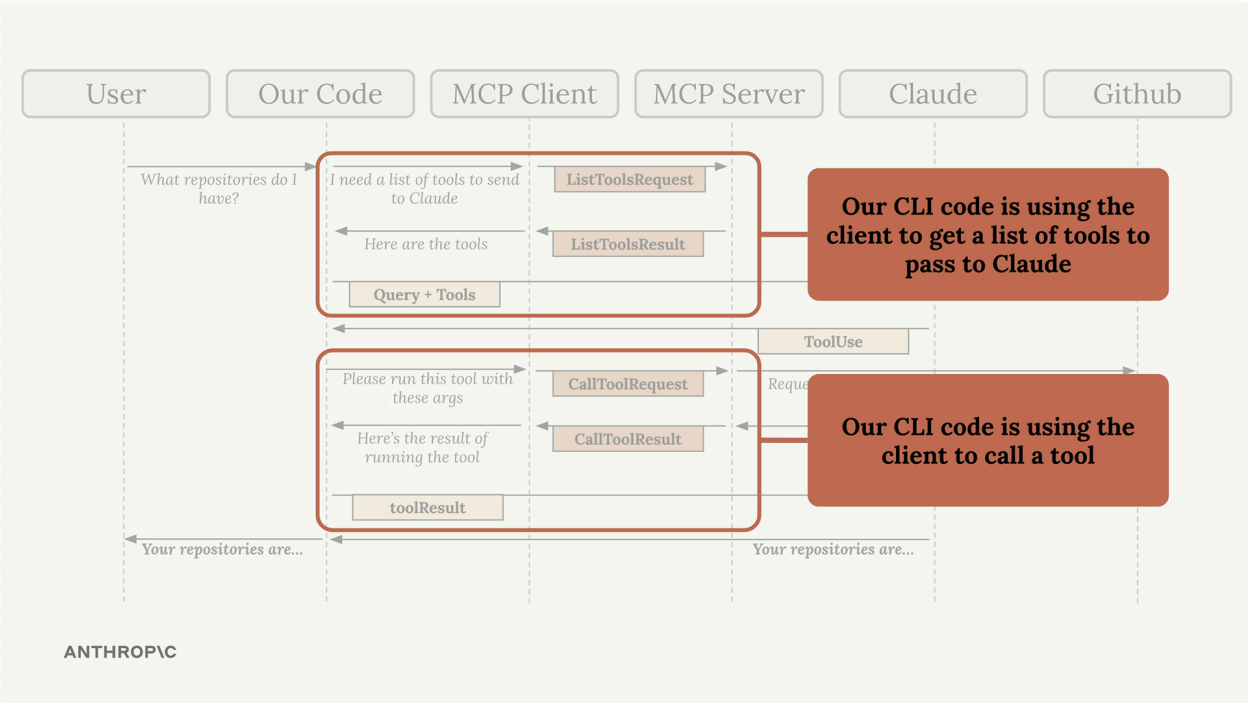
- MCP Client - A custom class we create to make using the session easier
- Client Session - The actual connection to the server (part of the MCP Python SDK)



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

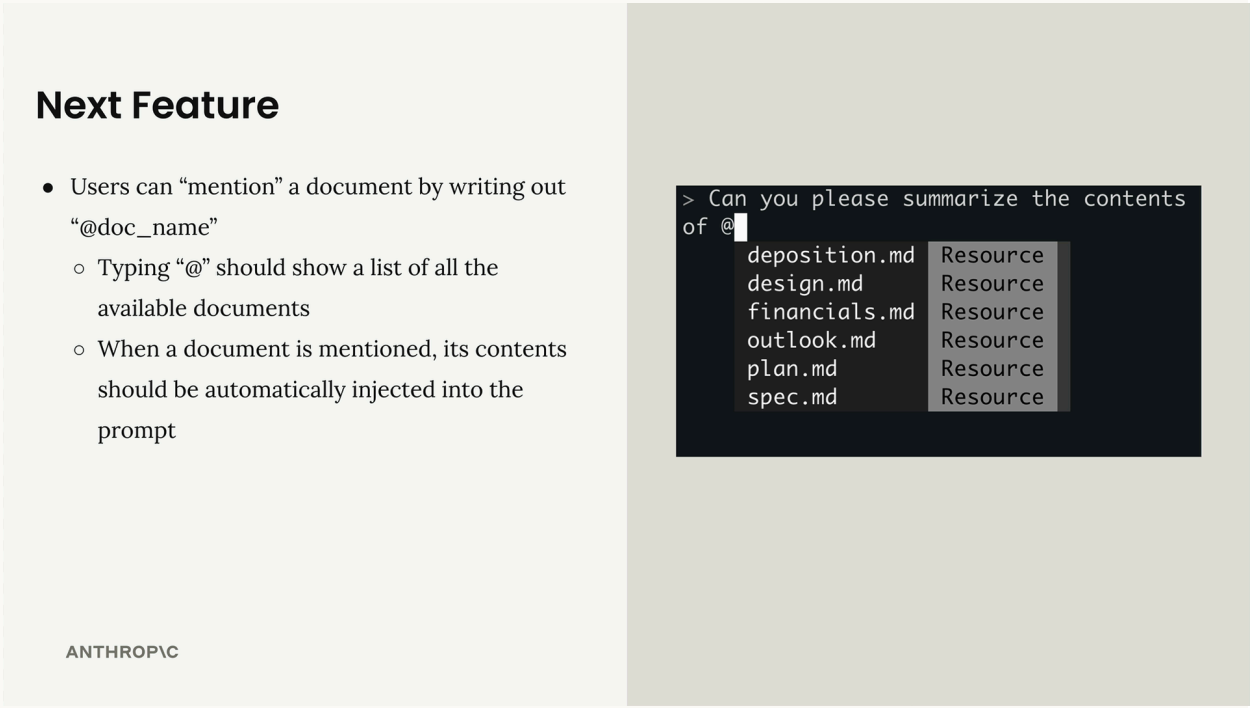
Summary
Resources in MCP servers allow you to expose data to clients, similar to GET request handlers in a typical HTTP server. They're perfect for scenarios where you need to fetch information rather than perform actions.

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)

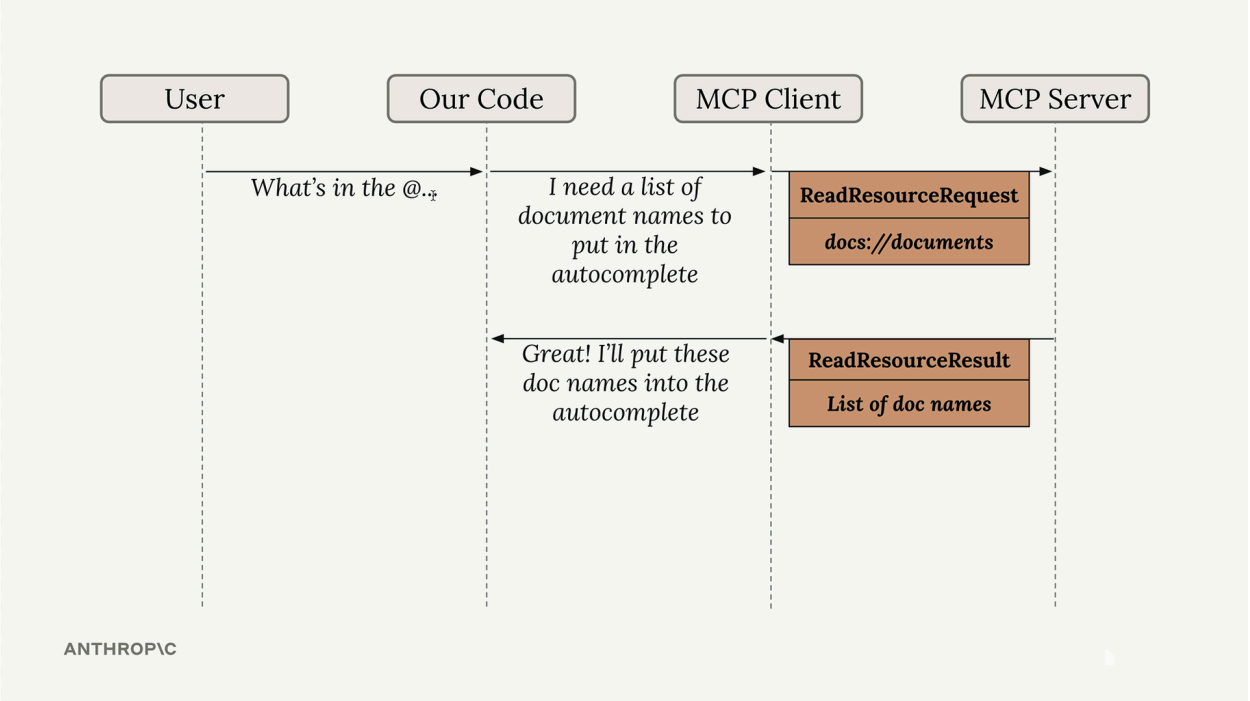
- Fetching the contents of a specific document (when mentioned)



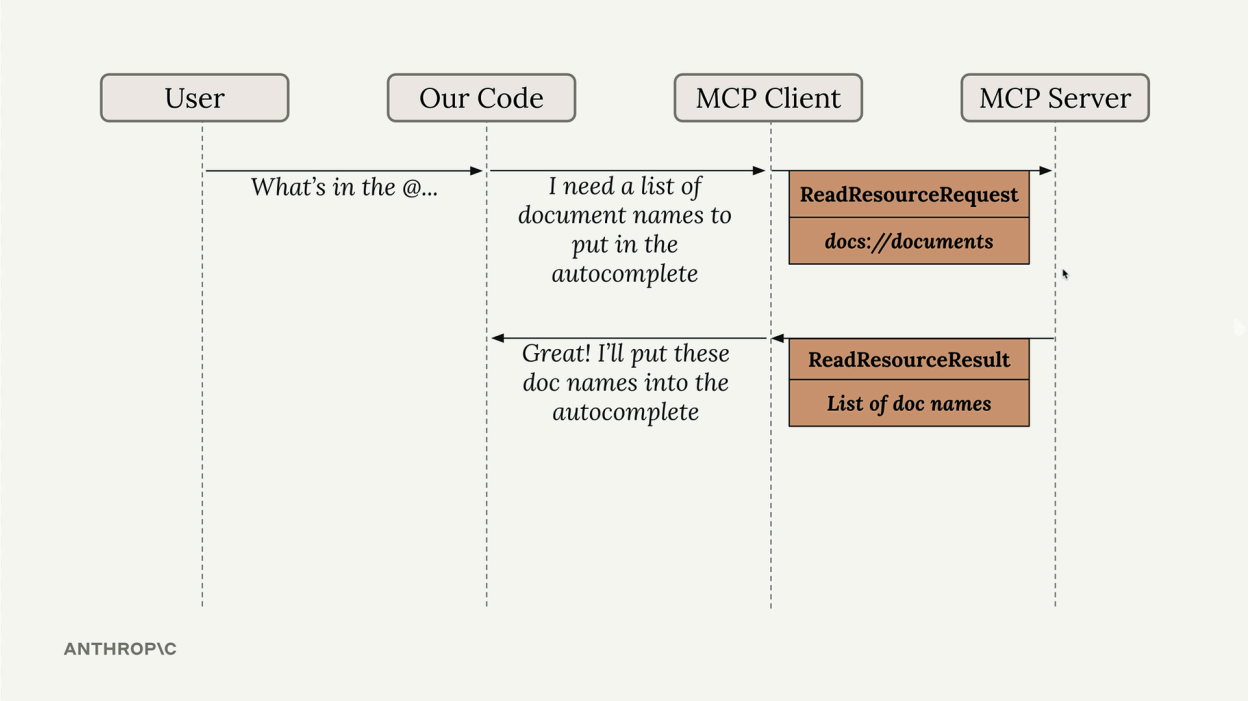
When a user mentions a document, your system automatically injects the document's contents into the prompt sent to Claude, eliminating the need for Claude to use tools to fetch the information.

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

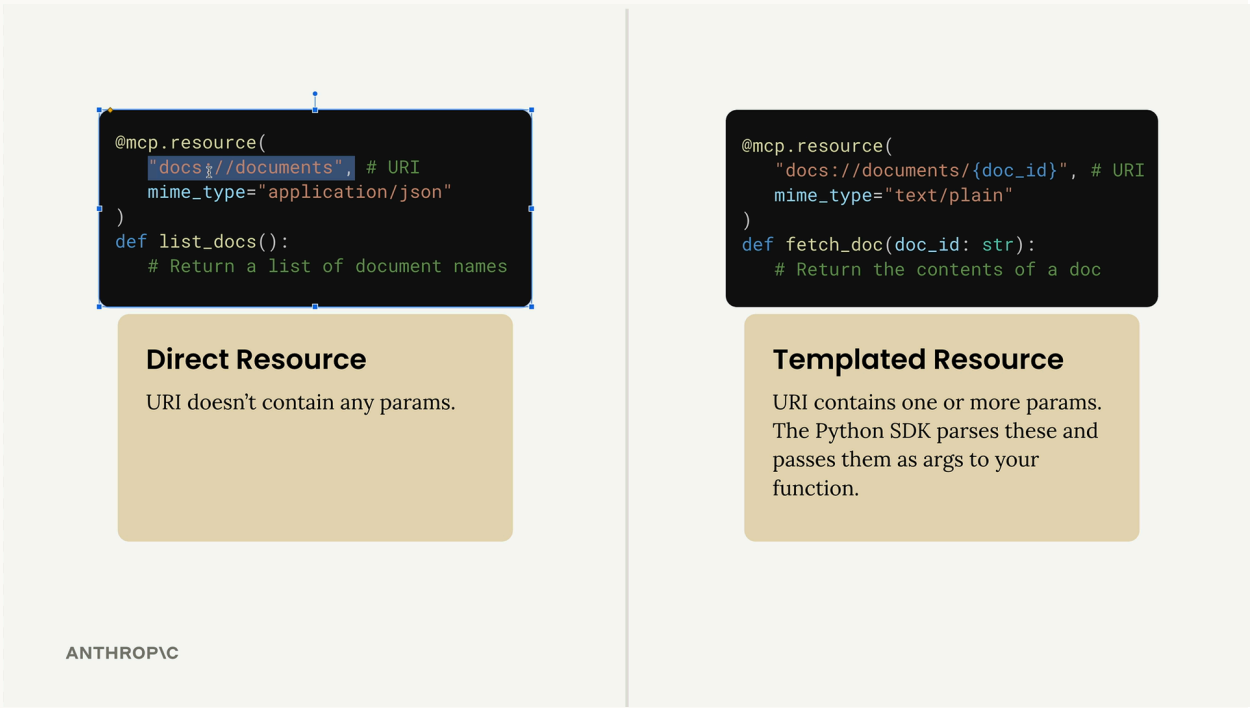
Direct resources have static URIs that never change. They're perfect for operations that don't need parameters.

```
@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

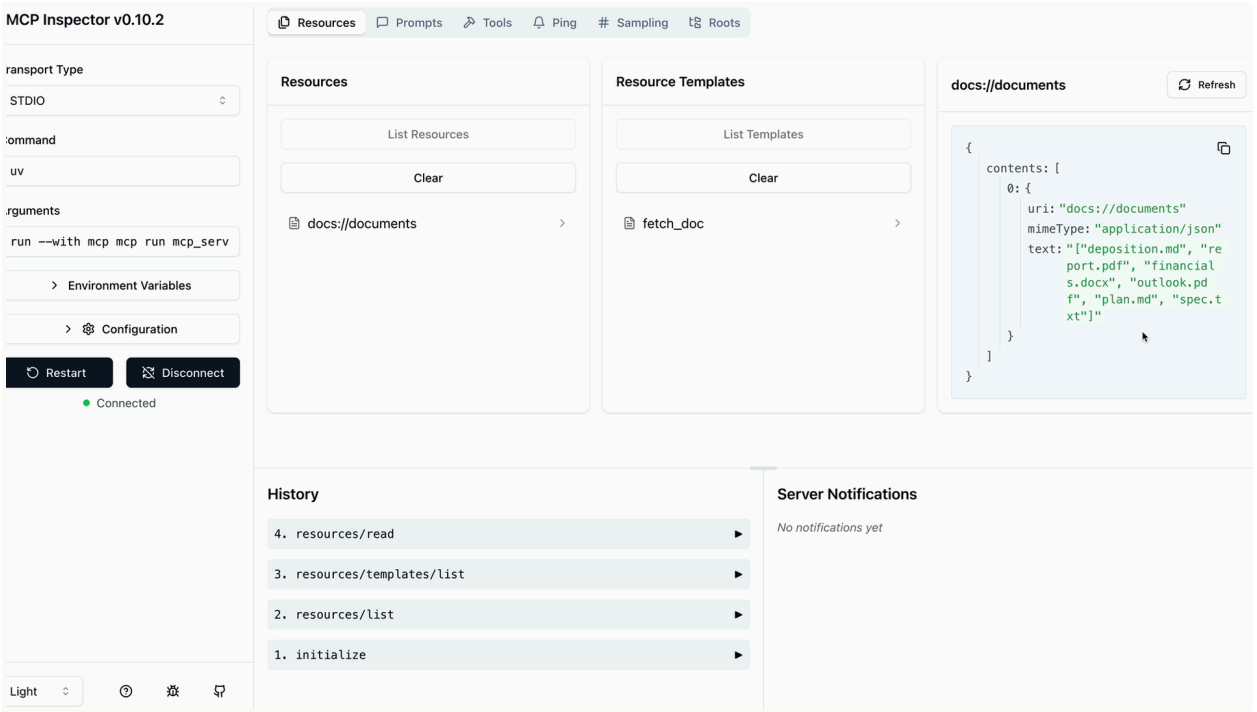
The MCP Python SDK automatically serializes your return values. You don't need to manually convert objects to JSON strings - just return the data structure and let the SDK handle serialization.

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

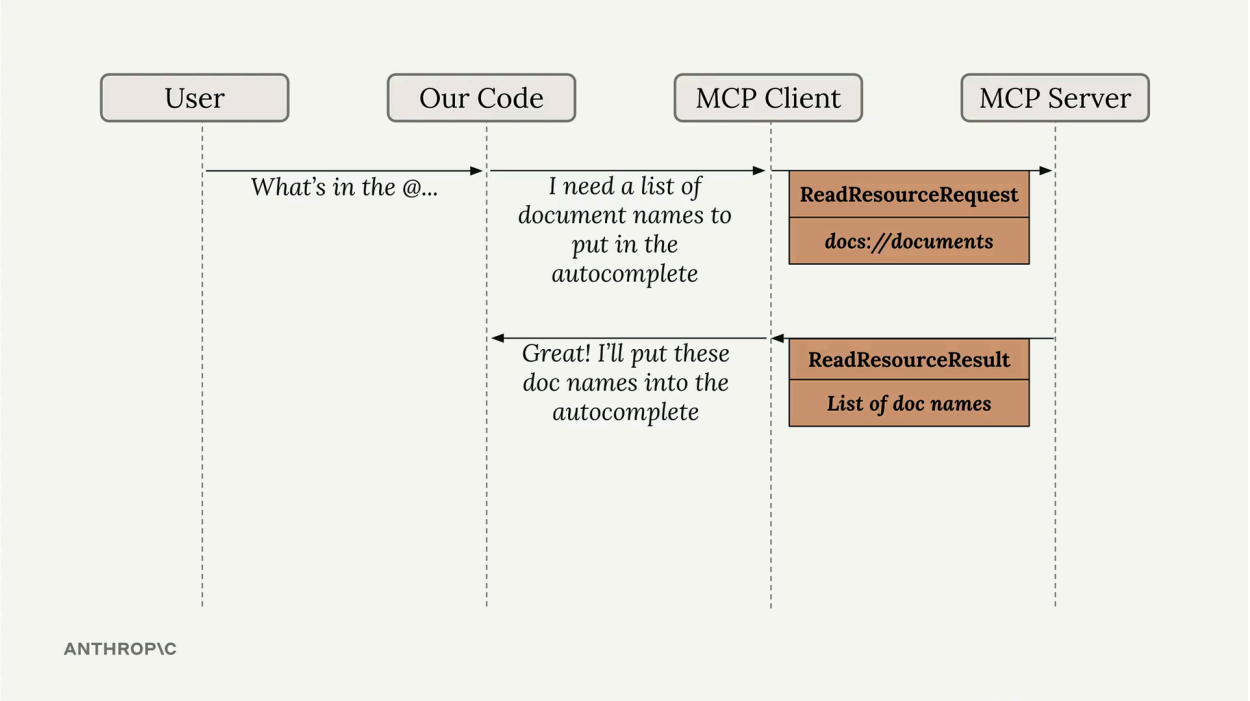


Click on any resource to test it. For templated resources, you'll need to provide values for the parameters. The inspector shows you the exact response structure your client will receive, including the MIME type and serialized data.

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

Summary
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

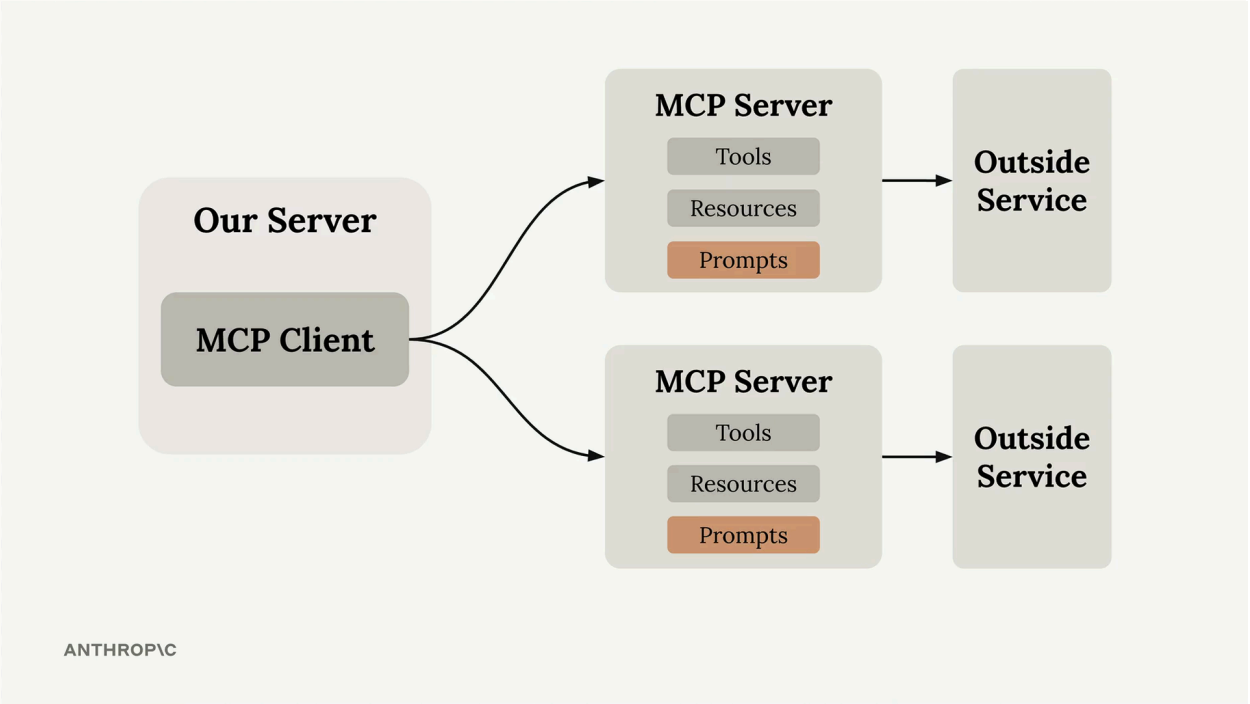
Once implemented, you can test the resource functionality through your CLI application. When you type "@" followed by a resource name, the system will:

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

Summary
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:

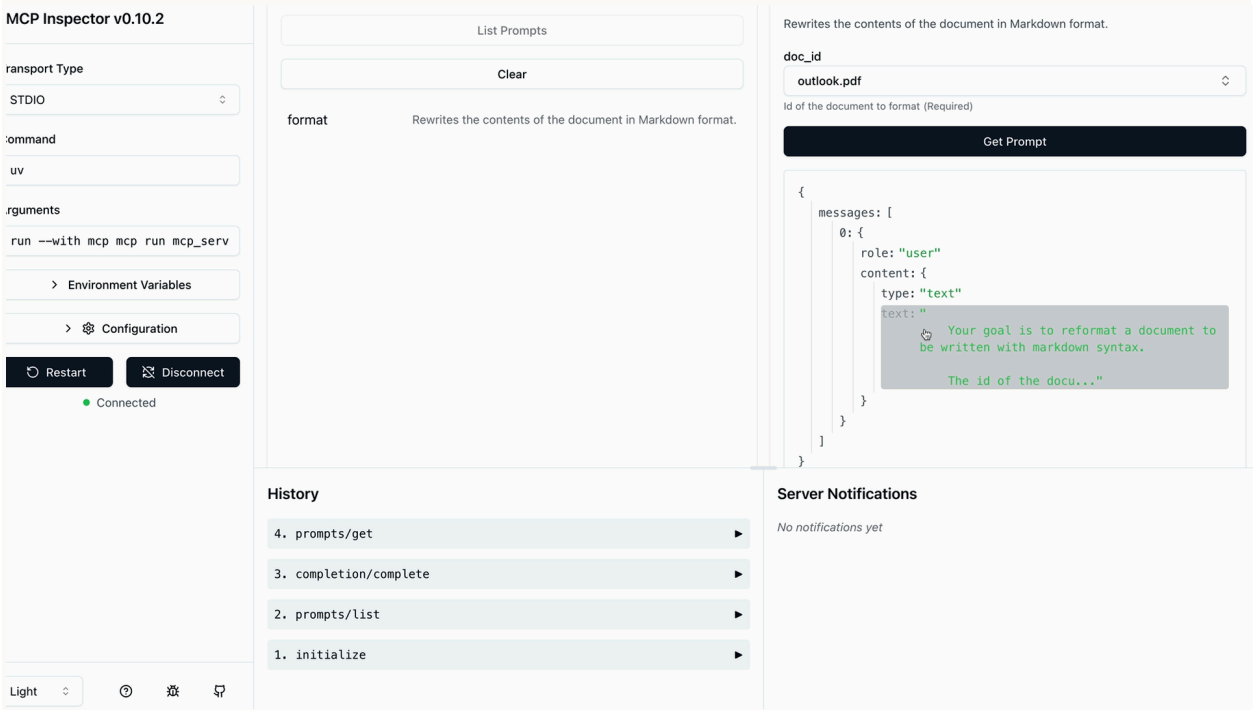
```
@mcp.prompt(
    name="format",
    description="Rewrites the contents of the document in Markdown format."
)
def format_document(
    doc_id: str = Field(description="Id of the document to format")
) -> list[base.Message]:
    prompt = f"""
Your goal is to reformat a document to be written with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

Summary
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

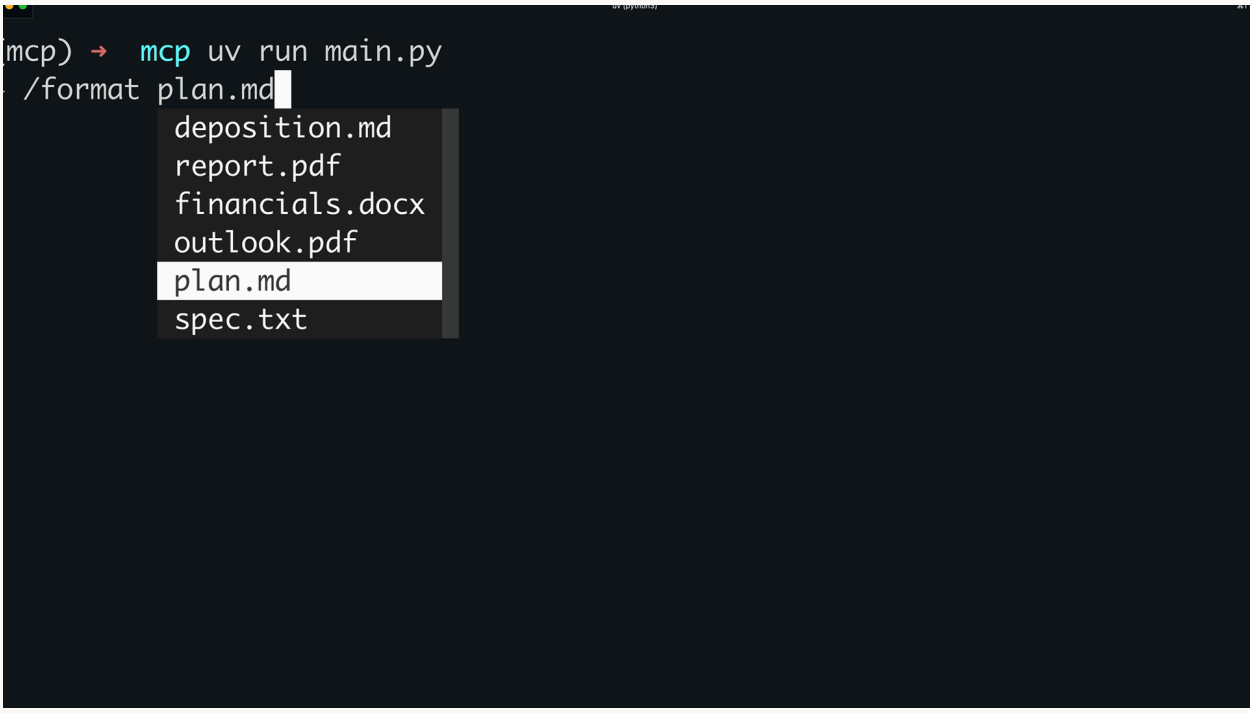
The get_prompt method is more interesting because it handles variable interpolation. When you request a prompt, you provide arguments that get passed to the prompt function as keyword arguments:

```
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:

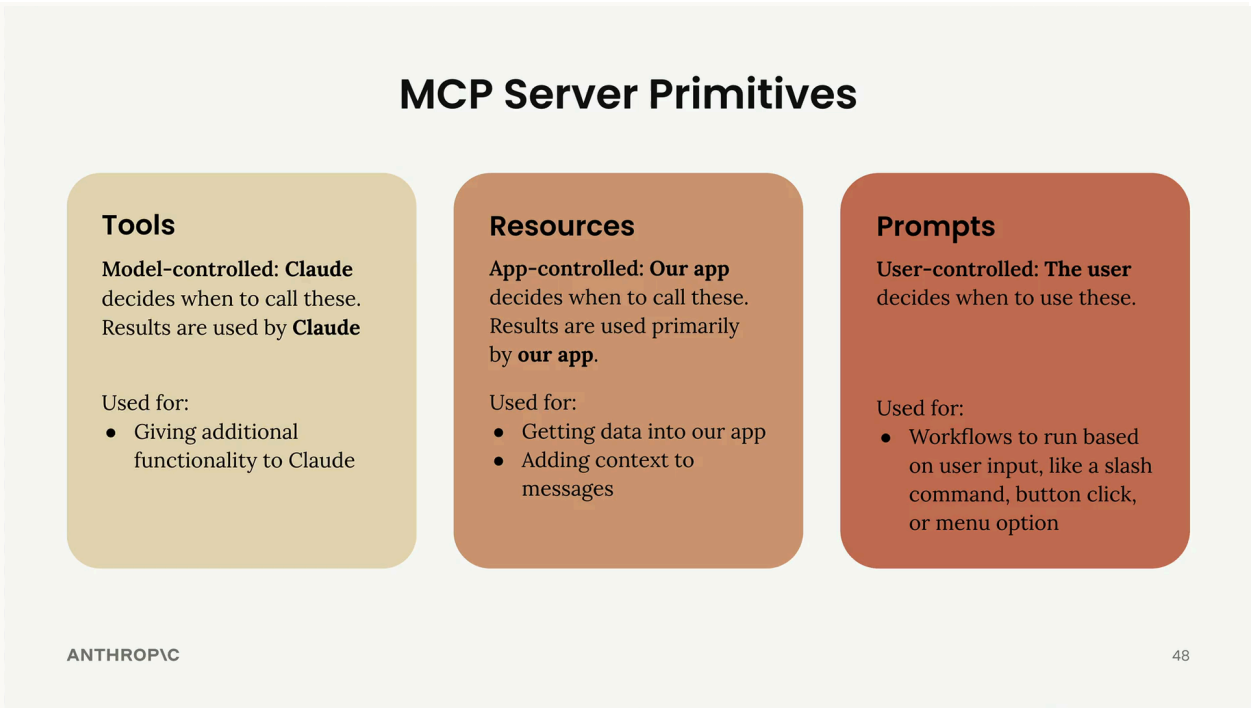
- Write and evaluate a prompt relevant to your server's functionality
- Define the prompt in your MCP server using the @mcp.prompt decorator
- Clients can request the prompt at any time
- Arguments provided by the client become keyword arguments in your prompt function
- The function returns formatted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

MCP Review:

Summary

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.

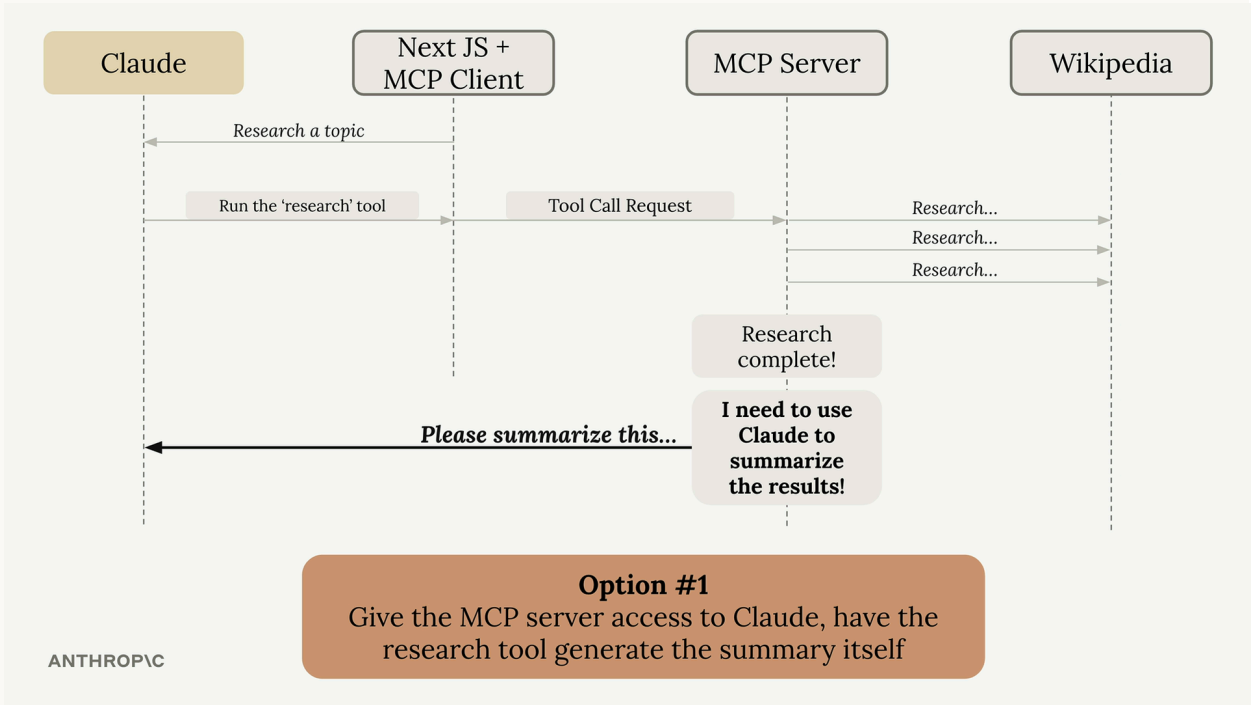
Model Context Protocol:Advanced Topics

Sampling

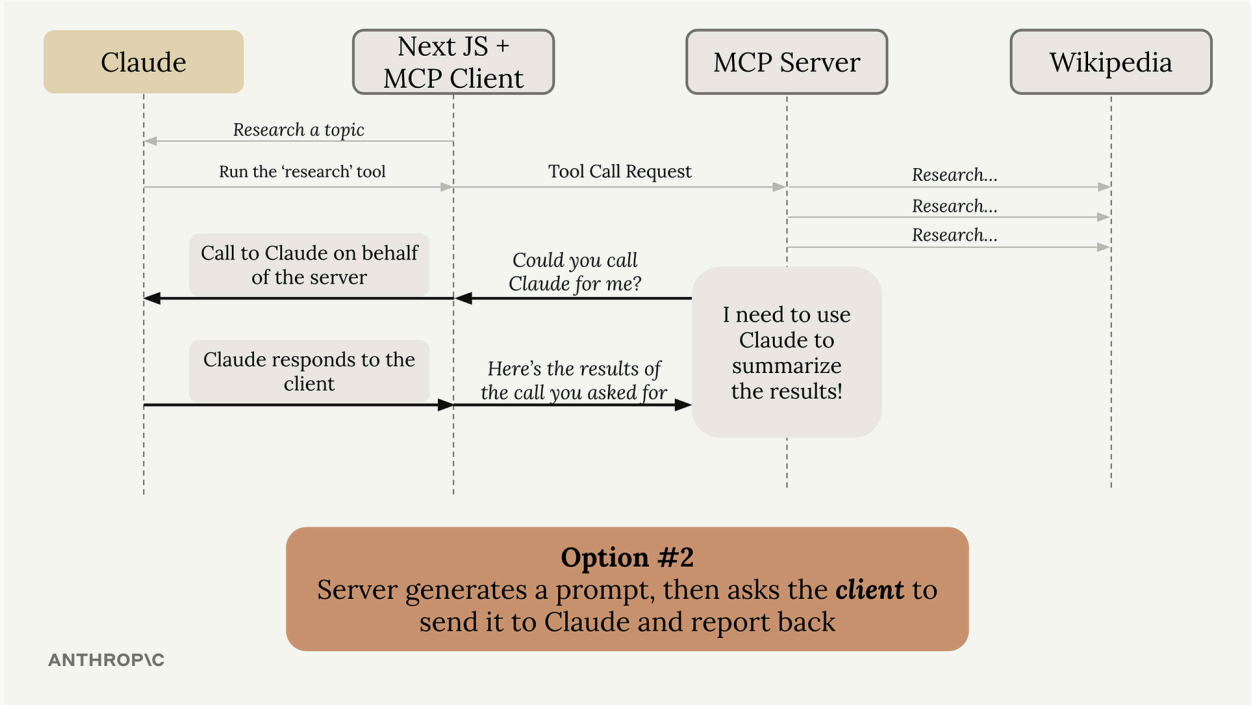
Summary
Sampling allows a server to access a language model like Claude through a connected MCP client. Instead of the server directly calling Claude, it asks the client to make the call on its behalf. This shifts the responsibility and cost of text generation from the server to the client.

The Problem Sampling Solves

Imagine you have an MCP server with a research tool that fetches information from Wikipedia. After gathering all that data, you need to summarize it into a coherent report. You have two options:



Option 1: Give the MCP server direct access to Claude. The server would need its own API key, handle authentication, manage costs, and implement all the Claude integration code. This works but adds significant complexity.



Option 2: Use sampling. The server generates a prompt and asks the client "Could you call Claude for me?" The client, which already has a connection to Claude, makes the call and returns the results.

How Sampling Works

The flow is straightforward:

- Server completes its work (like fetching Wikipedia articles)
- Server creates a prompt asking for text generation
- Server sends a sampling request to the client
- Client calls Claude with the provided prompt
- Client returns the generated text to the server
- Server uses the generated text in its response

Benefits of Sampling

- Reduces server complexity: The server doesn't need to integrate with language models directly
- Shifts cost burden: The client pays for token usage, not the server
- No API keys needed: The server doesn't need credentials for Claude
- Perfect for public servers: You don't want a public server racking up AI costs for every user

Implementation

Setting up sampling requires code on both sides:

Server Side

In your tool function, use the create_message function to request text generation:

```
@mcp.tool()
async def summarize(text_to_summarize: str, ctx: Context):
    prompt = f"""
    Please summarize the following text:
    {text_to_summarize}
    """

    result = await ctx.session.create_message(
        messages=[
            SamplingMessage(
                role="user",
                content=TextContent(
                    type="text",
                    text=prompt
                )
            )
        ],
        max_tokens=4000,
```

```
        system_prompt="You are a helpful research assistant",
    )

    if result.content.type == "text":
        return result.content.text
    else:
        raise ValueError("Sampling failed")
```

Client Side

Create a sampling callback that handles the server's requests:

```
async def sampling_callback(
    context: RequestContext, params: CreateMessageRequestParams
):
    # Call Claude using the Anthropic SDK
    text = await chat(params.messages)

    return CreateMessageResult(
        role="assistant",
        model=model,
        content=TextContent(type="text", text=text),
    )
```

Then pass this callback when initializing your client session:

```
async with ClientSession(
    read,
    write,
    sampling_callback=sampling_callback
) as session:
    await session.initialize()
```

When to Use Sampling

Sampling is most valuable when building publicly accessible MCP servers. You don't want random users generating unlimited text at your expense. By using sampling, each client pays for their own AI usage while still benefiting from your server's functionality.

The technique essentially moves the AI integration complexity from your server to the client, which often already has the necessary connections and credentials in place.

Sampling walkthrough

Downloads
Sampling.zip
Log and progress notifications

Summary

Logging and progress notifications are simple to implement but make a huge difference in user experience when working with MCP servers. They help users understand what's happening during long-running operations instead of wondering if something has broken.

When Claude calls a tool that takes time to complete - like researching a topic or processing data - users typically see nothing until the operation finishes. This can be frustrating because they don't know if the tool is working or has stalled.

With logging and progress notifications enabled, users get real-time feedback showing exactly what's happening behind the scenes. They can see progress bars, status messages, and detailed logs as the operation runs.

How It Works

In the Python MCP SDK, logging and progress notifications work through the Context argument that's automatically provided to your tool functions. This context object gives you methods to communicate back to the client during execution.

```
@mcp.tool(
    name="research",
    description="Research a given topic"
)
async def research(
    topic: str = Field(description="Topic to research"),
    *,
    context: Context
):
    await context.info("About to do research...")
    await context.report_progress(20, 100)
    sources = await do_research(topic)
```

```
    await context.info("Writing report...")

    await context.report_progress(70, 100)

    results = await generate_report(sources)


    return results
```

The key methods you'll use are:

- context.info() - Send log messages to the client
- context.report_progress() - Update progress with current and total values

Client-Side Implementation

On the client side, you need to set up callback functions to handle these notifications. The server emits these messages, but it's up to your client application to decide how to present them to users.

```
async def logging_callback(params: LoggingMessageNotificationParams):

    print(params.data)



async def print_progress_callback(

    progress: float, total: float | None, message: str | None

):

    if total is not None:

        percentage = (progress / total) * 100

        print(f"Progress: {progress}/{total} ({percentage:.1f}%)")

    else:
```

```
    print(f"Progress: {progress}")


async def run():
    async with stdio_client(server_params) as (read, write):
        async with ClientSession(
            read,
            write,
            logging_callback=logging_callback
        ) as session:
            await session.initialize()


            await session.call_tool(
                name="add",
                arguments={"a": 1, "b": 3},
                progress_callback=print_progress_callback,
            )
```

You provide the logging callback when creating the client session, and the progress callback when making individual tool calls. This gives you flexibility to handle different types of notifications appropriately.

Presentation Options

How you present these notifications depends on your application type:

- CLI applications - Simply print messages and progress to the terminal
- Web applications - Use WebSockets, server-sent events, or polling to push updates to the browser

- Desktop applications - Update progress bars and status displays in your UI

Remember that implementing these notifications is entirely optional. You can choose to ignore them completely, show only certain types, or present them however makes sense for your application. They're purely user experience enhancements to help users understand what's happening during long-running operations.

Notifications walkthrough

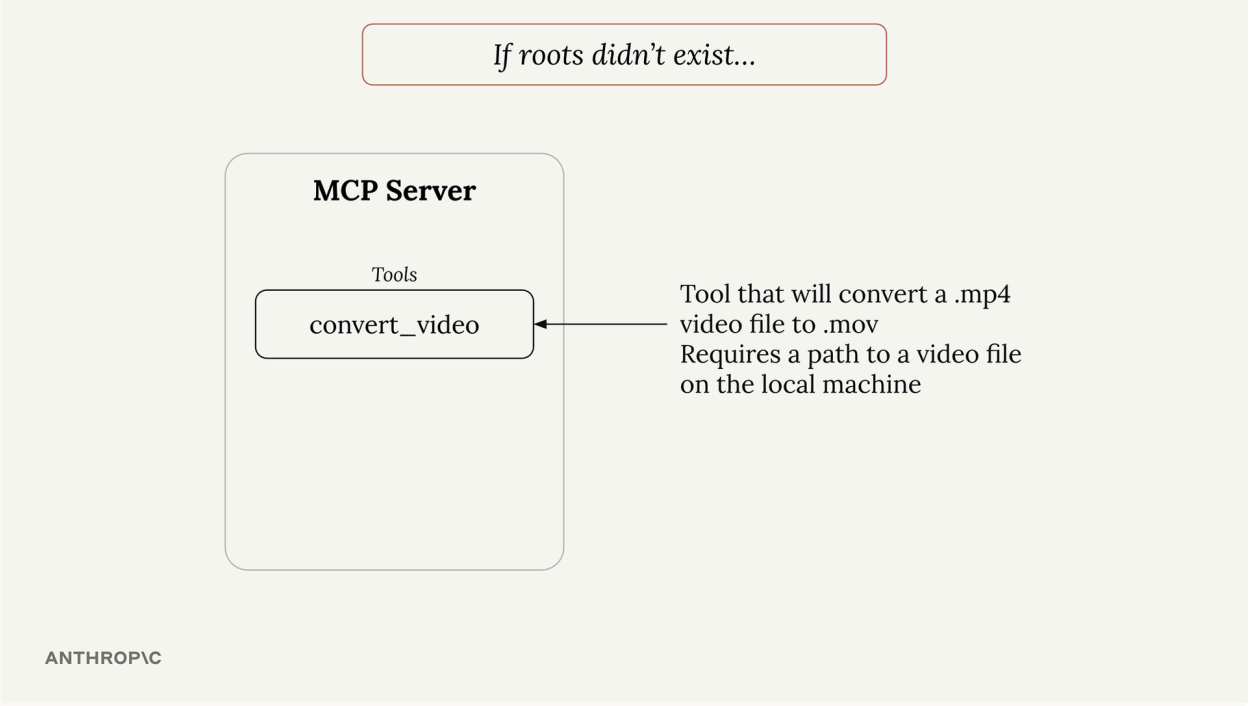
Downloads
Notifications.zip

Roots

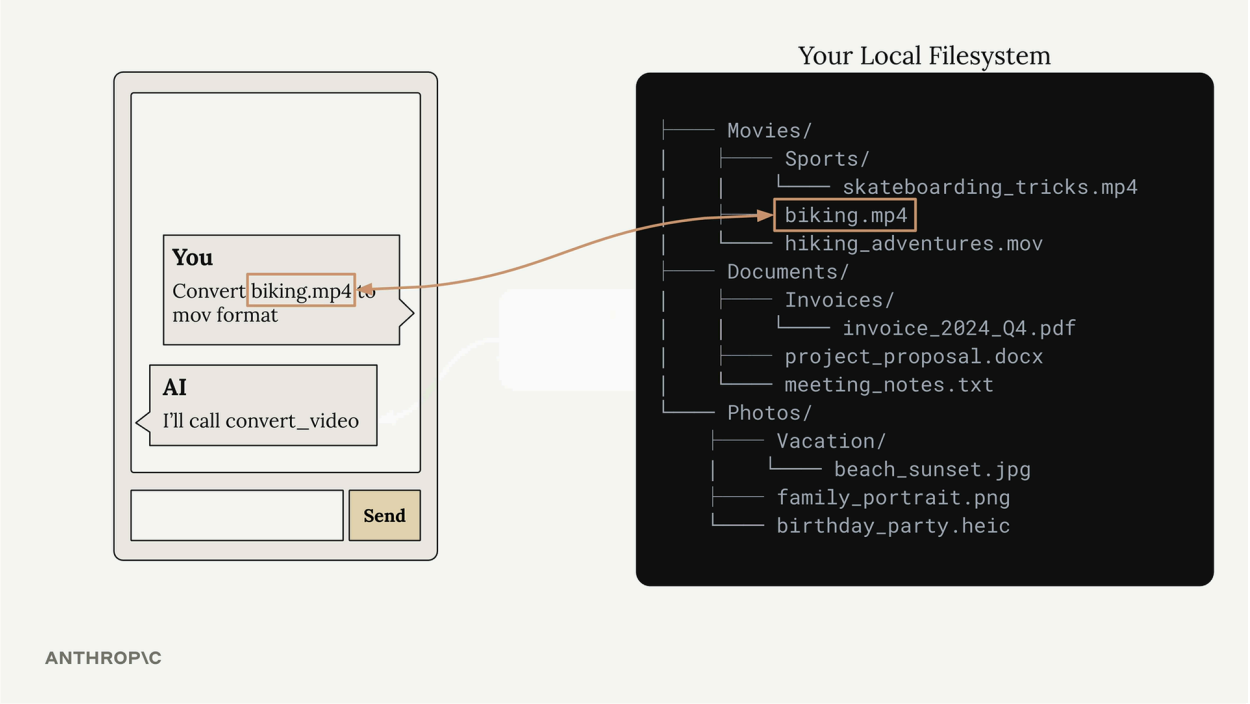
Summary
Roots are a way to grant MCP servers access to specific files and folders on your local machine. Think of them as a permission system that says "Hey, MCP server, you can access these files" - but they do much more than just grant permission.

The Problem Roots Solve

Without roots, you'd run into a common issue. Imagine you have an MCP server with a video conversion tool that takes a file path and converts an MP4 to MOV format.



When a user asks Claude to "convert biking.mp4 to mov format", Claude would call the tool with just the filename. But here's the problem - Claude has no way to search through your entire file system to find where that file actually lives.



Your file system might be complex with files scattered across different directories. The user knows the biking.mp4 file is in their Movies folder, but Claude doesn't have that context.

You could solve this by requiring users to always provide full paths, but that's not very user-friendly. Nobody wants to type out complete file paths every time.

Roots in Action

Here's how the workflow changes with roots:

1. User asks to convert a video file
2. Claude calls list_roots to see what directories it can access
3. Claude calls read_dir on accessible directories to find the file
4. Once found, Claude calls the conversion tool with the full path

This happens automatically - users can still just say "convert biking.mp4" without providing full paths.

Security and Boundaries

Roots also provide security by limiting access. If you only grant access to your Desktop folder, the MCP server cannot access files in other locations like Documents or Downloads.

When Claude tries to access a file outside the approved roots, it gets an error and can inform the user that the file isn't accessible from the current server configuration.

Implementation Details

The MCP SDK doesn't automatically enforce root restrictions - you need to implement this yourself. A typical pattern is to create a helper function like is_path_allowed() that:

- Takes a requested file path
- Gets the list of approved roots
- Checks if the requested path falls within one of those roots
- Returns true/false for access permission

You then call this function in any tool that accesses files or directories before performing the actual file operation.

Key Benefits

- User-friendly - Users don't need to provide full file paths
- Focused search - Claude only looks in approved directories, making file discovery faster
- Security - Prevents accidental access to sensitive files outside approved areas
- Flexibility - You can provide roots through tools or inject them directly into prompts

Roots make MCP servers both more powerful and more secure by giving Claude the context it needs to find files while maintaining clear boundaries around what it can access.

Roots walkthrough

Downloads

Roots.zip

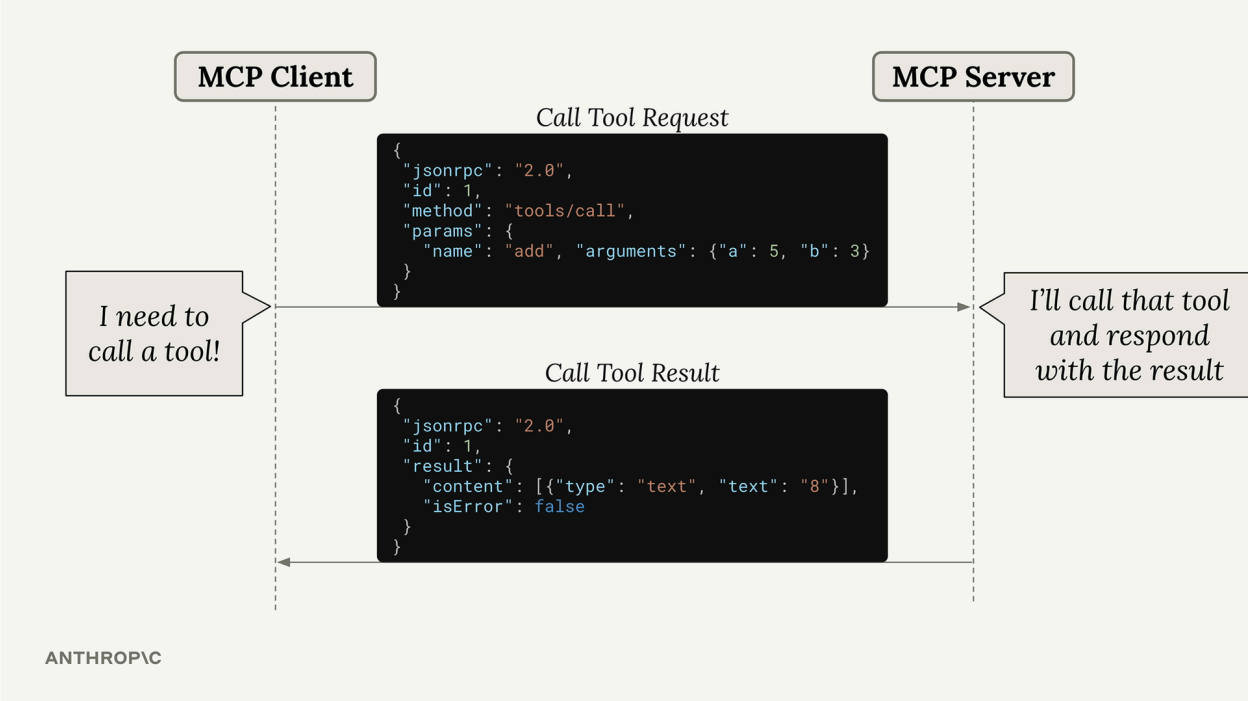
Transports and communication

Summary

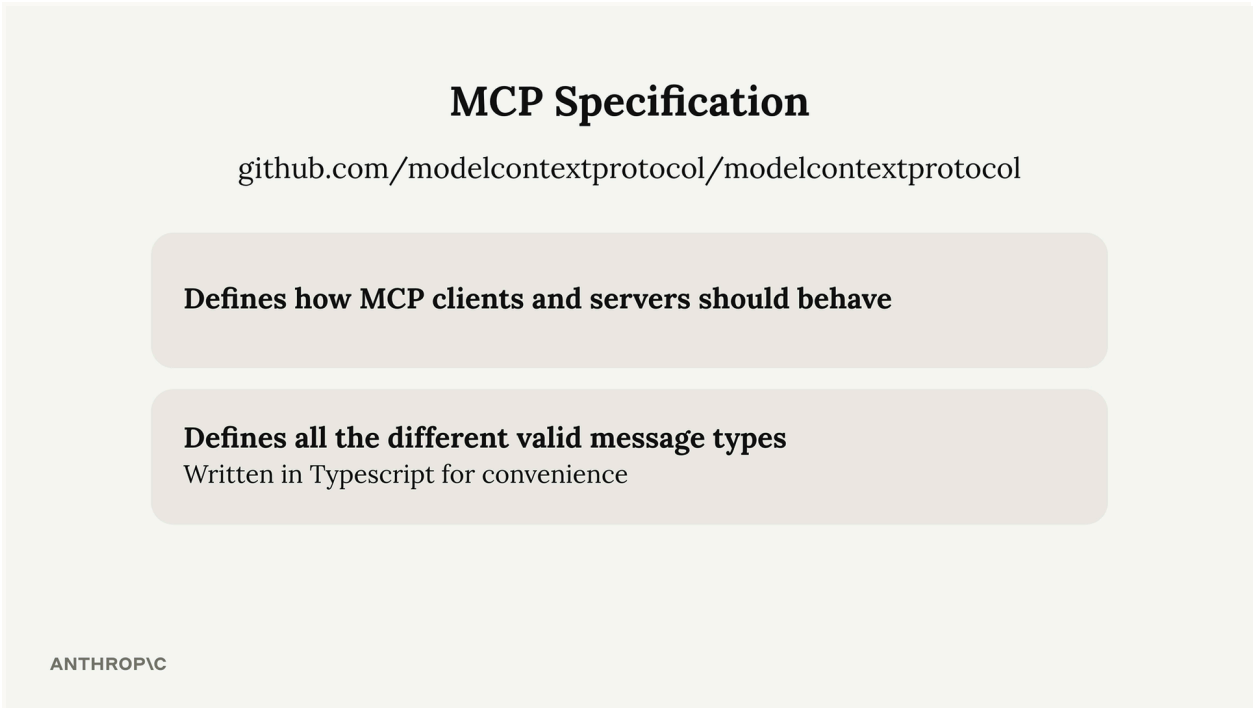
MCP (Model Context Protocol) uses JSON messages to handle communication between clients and servers. Understanding these message types is crucial for working with MCP, especially when dealing with different transport methods like the streamable HTTP transport.

Message Format

All MCP communication happens through JSON messages. Each message type serves a specific purpose - whether it's calling a tool, listing available resources, or sending notifications about system events.



Here's a typical example: when Claude needs to call a tool provided by an MCP server, the client sends a "Call Tool Request" message. The server processes this request, runs the tool, and responds with a "Call Tool Result" message containing the output.



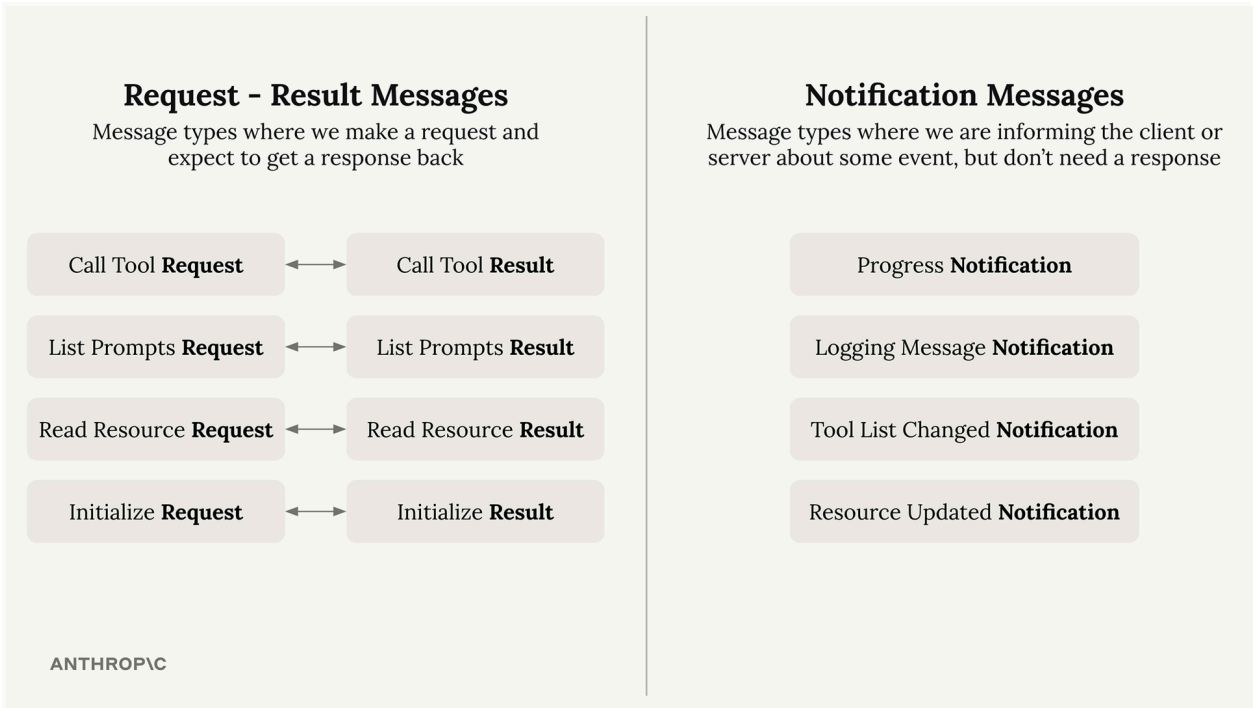
MCP Specification

The complete list of message types is defined in the official MCP specification repository on GitHub. This specification is separate from the various SDK repositories (like Python or TypeScript SDKs) and serves as the authoritative source for how MCP should work.

The message types are written in TypeScript for convenience - not because they're executed as TypeScript code, but because TypeScript provides a clear way to describe data structures and types.

Message Categories

MCP messages fall into two main categories:



Request-Result Messages

These messages always come in pairs. You send a request and expect to get a result back:

- Call Tool Request → Call Tool Result
- List Prompts Request → List Prompts Result
- Read Resource Request → Read Resource Result
- Initialize Request → Initialize Result

Notification Messages

These are one-way messages that inform about events but don't require a response:

- Progress Notification - Updates on long-running operations
- Logging Message Notification - System log messages
- Tool List Changed Notification - When available tools change
- Resource Updated Notification - When resources are modified

Client vs Server Messages

The MCP specification organizes messages by who sends them:

Client messages include requests that clients send to servers (like tool calls) and notifications that clients might send.

Server messages include requests that servers send to clients and notifications that servers broadcast.

Why This Matters

Understanding that servers can send messages to clients is particularly important when working with different transport methods. Some transports, like the streamable HTTP transport, have limitations on which types of messages can flow in which directions.

The key insight is that MCP is designed as a bidirectional protocol - both clients and servers can initiate communication. This becomes crucial when you need to choose the right transport method for your specific use case.

The STDIO transport

Summary
MCP clients and servers communicate by exchanging JSON messages, but how do these messages actually get transmitted? The communication channel used is called a transport, and there are several ways to implement this - from HTTP requests to WebSockets to even writing JSON on a postcard (though that last one isn't recommended for production use).

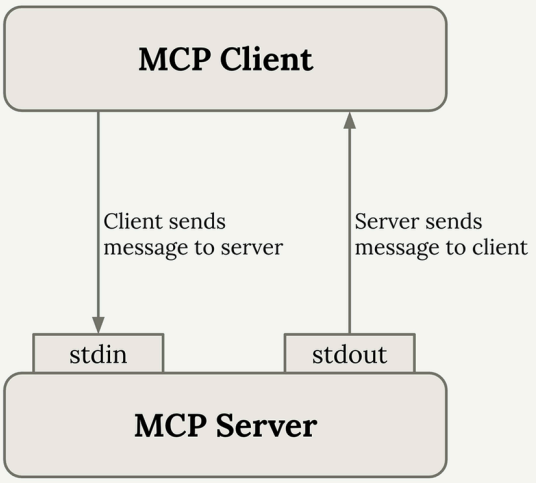
The Stdio Transport

When you're first developing an MCP server or client, the most commonly used transport is the stdio transport. This approach is straightforward: the client launches the MCP server as a subprocess and communicates through standard input and output streams.

‘Stdio’ transport

- Client launches the MCP server as a subprocess
- Client sends messages to the MCP Server using the server’s ‘stdin’
- Server responds by writing to ‘stdout’
- **Critical:** either the server or the client can send a message at any time!
- **Only appropriate when the client and server are running on the same machine**

ANTHROP\C



Here's how it works:

- Client sends messages to the server using the server's stdin
- Server responds by writing to stdout
- Either the server or client can send a message at any time
- Only works when client and server run on the same machine

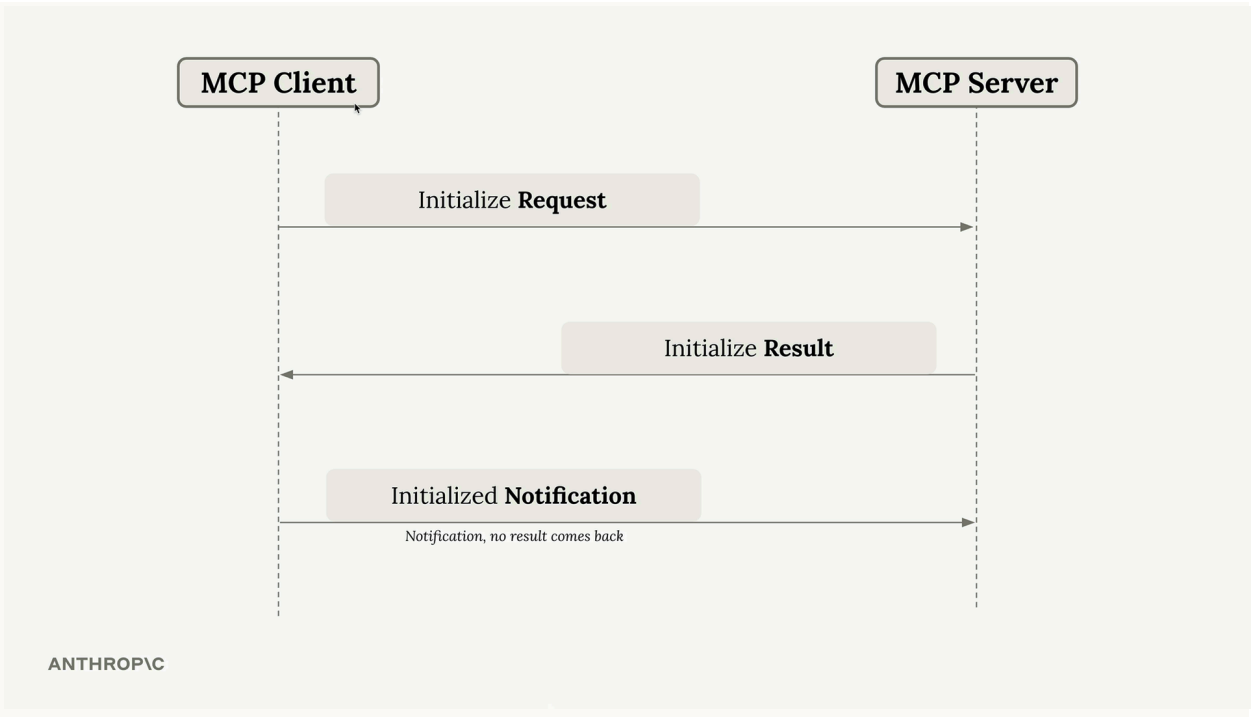
Seeing Stdio in Action

You can actually test an MCP server directly from your terminal without writing a separate client. When you run a server with uv run server.py, it listens to stdin and writes responses to stdout. This means you can paste JSON messages directly into your terminal and see the server's responses immediately.

The terminal output shows the complete message exchange, including example messages for initialization and tool calls.

MCP Connection Sequence

Every MCP connection must start with a specific three-message handshake:

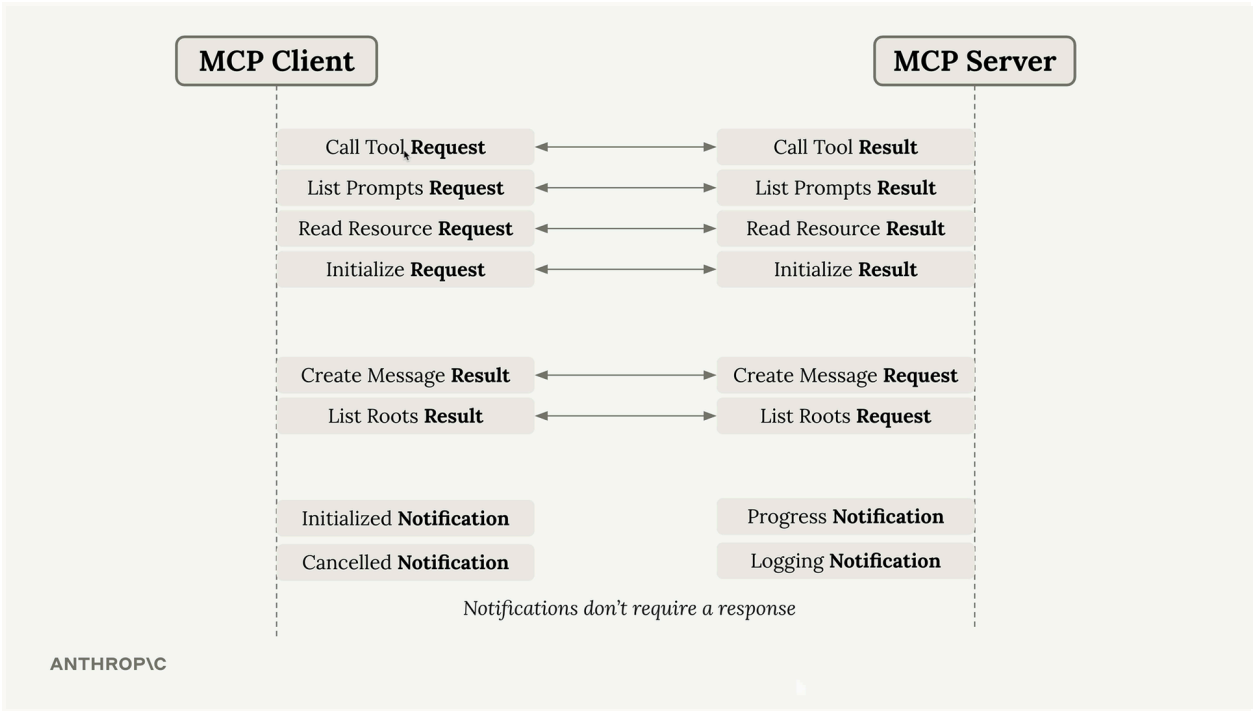


1. Initialize Request - Client sends this first
2. Initialize Result - Server responds with capabilities
3. Initialized Notification - Client confirms (no response expected)

Only after this handshake can you send other requests like tool calls or prompt listings.

Message Types and Flow

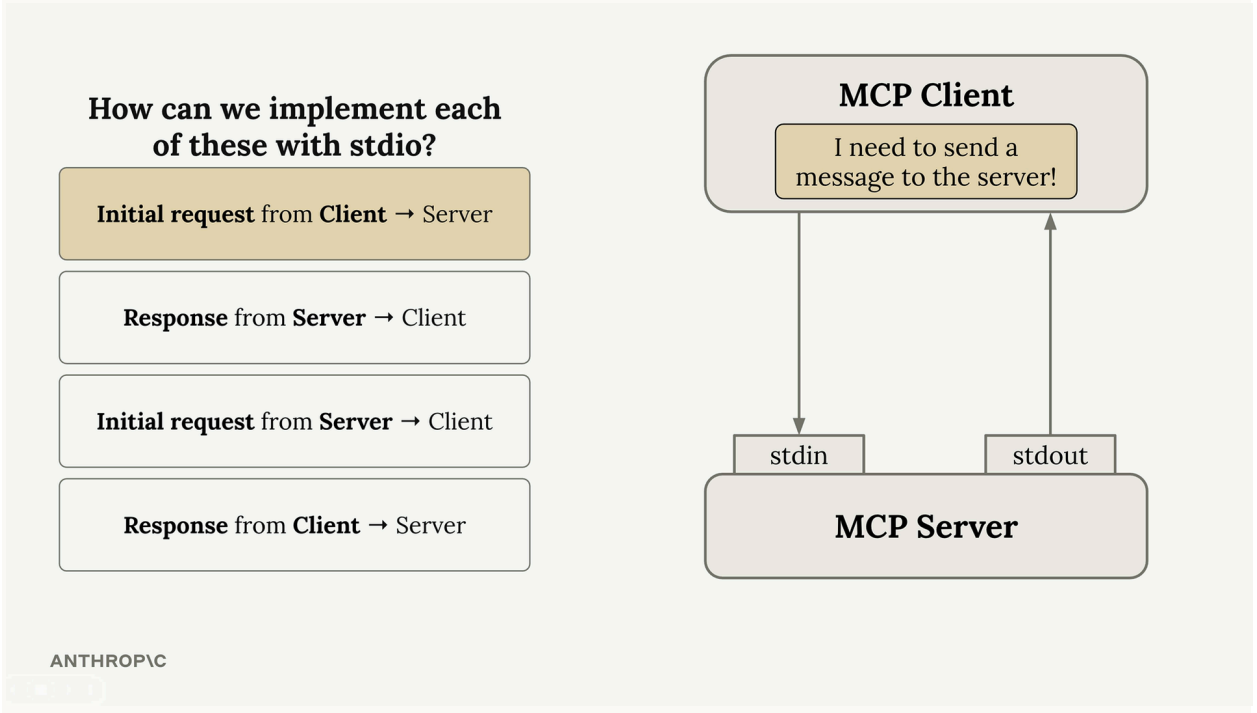
MCP supports various message types that flow in both directions:



The key insight is that some messages require responses (requests → results) while others don't (notifications). Both client and server can initiate communication at any time.

Four Communication Scenarios

With any transport, you need to handle four different communication patterns:



- Client → Server request: Client writes to stdin
- Server → Client response: Server writes to stdout
- Server → Client request: Server writes to stdout
- Client → Server response: Client writes to stdin

The beauty of stdio transport is its simplicity - either party can initiate communication at any time using these two channels.

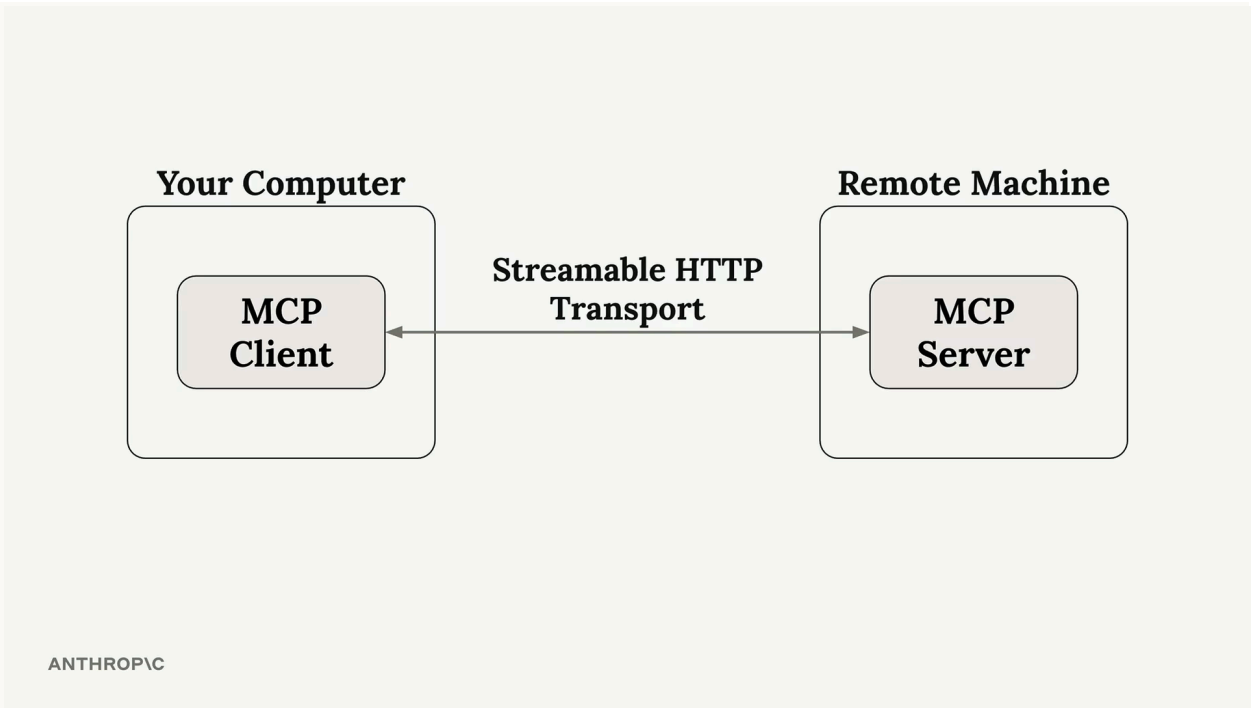
Why This Matters

Understanding stdio transport is crucial because it represents the "ideal" case where bidirectional communication is seamless. When we move to other transports like HTTP, we'll encounter limitations where the server cannot always initiate requests to the client. The stdio transport serves as our baseline for understanding what full MCP communication looks like before we tackle the constraints of other transport methods.

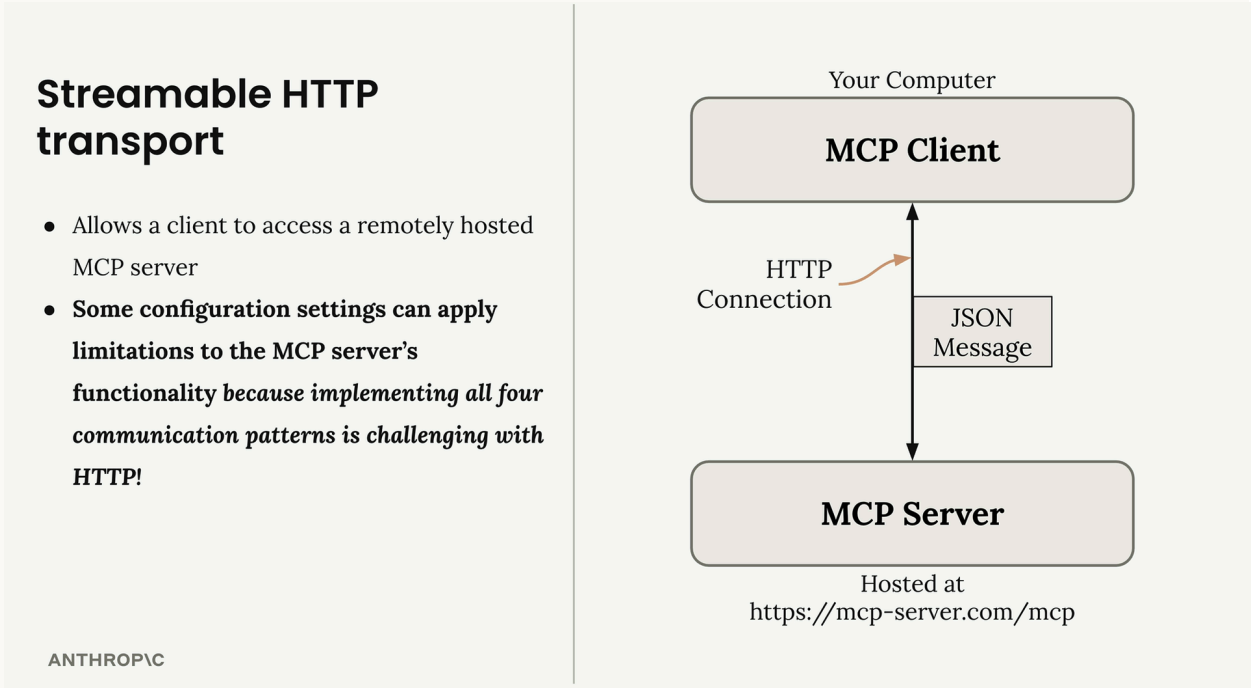
For development and testing, stdio transport is perfect. For production deployments where client and server need to run on different machines, you'll need to consider other transport options with their own trade-offs.

The StreamableHTTP transport

Summary
The streamable HTTP transport enables MCP clients to connect to remotely hosted servers over HTTP connections. Unlike the standard I/O transport that requires both client and server on the same machine, this transport opens up possibilities for public MCP servers that anyone can access.



However, there's an important caveat: some configuration settings can significantly limit your MCP server's functionality. If your application works perfectly with standard I/O transport locally but breaks when deployed with HTTP transport, this is likely the culprit.



Configuration Settings That Matter

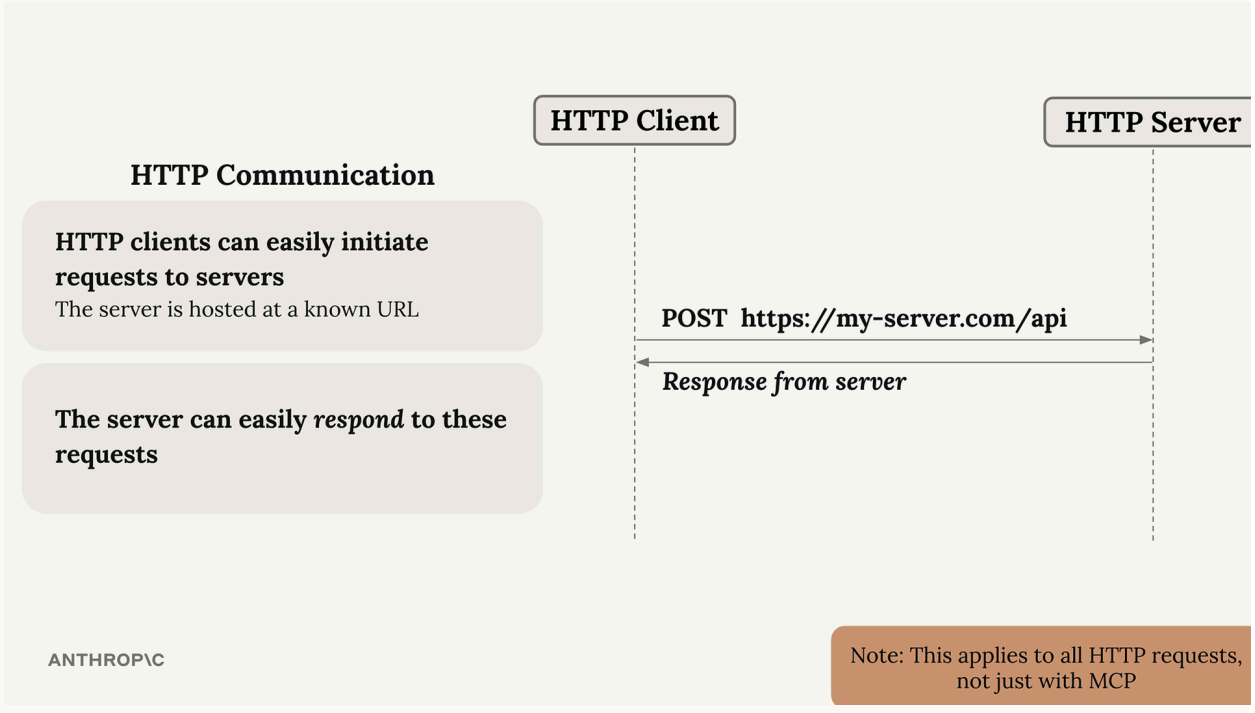
Two key settings control how the streamable HTTP transport behaves:

- stateless_http - Controls connection state management
- json_response - Controls response format handling

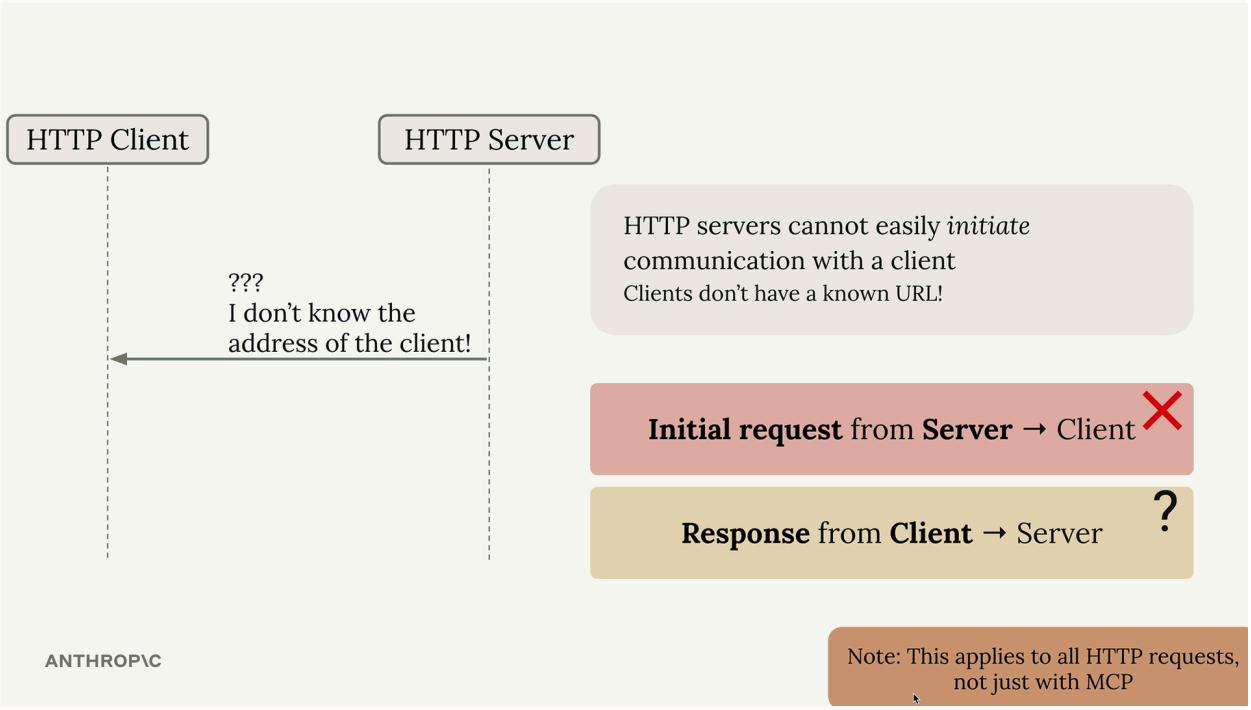
By default, both settings are false, but certain deployment scenarios may force you to set them to true. When enabled, these settings can break core functionality like progress notifications, logging, and server-initiated requests.

The HTTP Communication Challenge

To understand why these limitations exist, we need to review how HTTP communication works. In standard HTTP:



- Clients can easily initiate requests to servers (the server has a known URL)
- Servers can easily respond to these requests
- Servers cannot easily initiate requests to clients (clients don't have known URLs)
- Response patterns from client back to server become problematic



MCP Message Types Affected

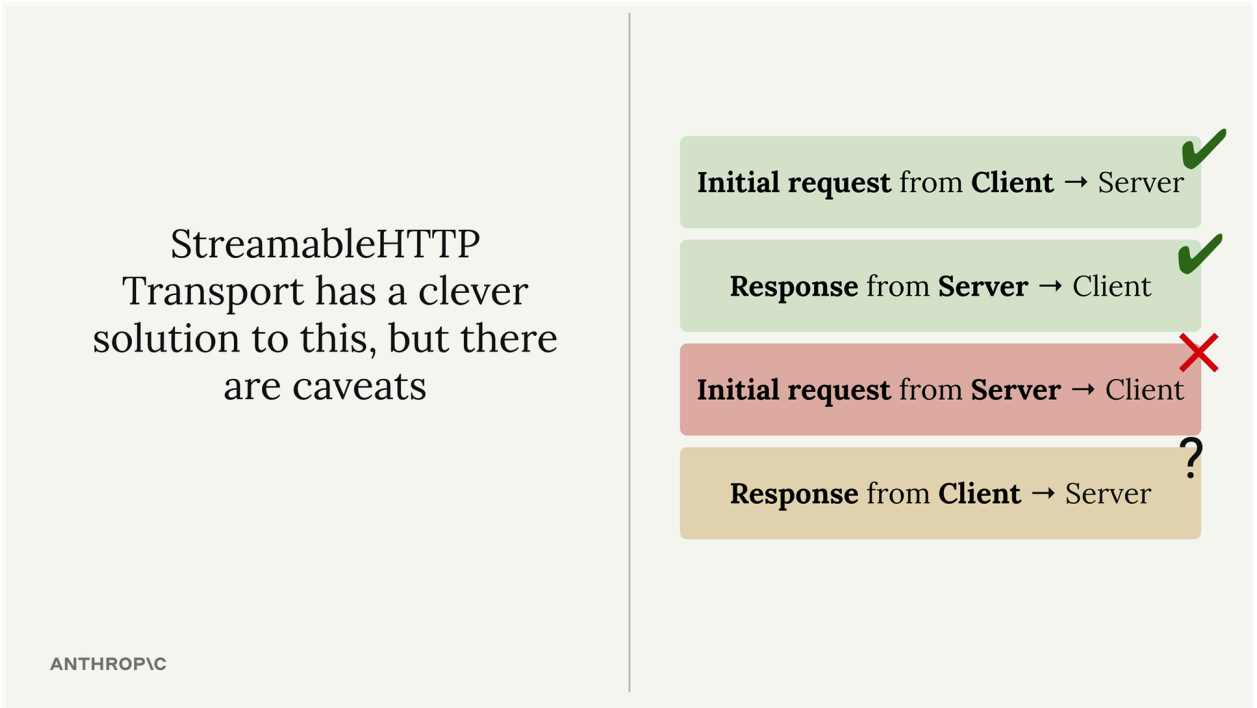
This HTTP limitation impacts specific MCP communication patterns. The following message types become difficult to implement with plain HTTP:

- Server-initiated requests: Create Message requests, List Roots requests
- Notifications: Progress notifications, Logging notifications, Initialized notifications, Cancelled notifications

These are exactly the features that break when you enable the restrictive HTTP settings. Progress bars disappear, logging stops working, and server-initiated sampling requests fail.

The Streamable HTTP Solution

The streamable HTTP transport does provide a clever solution to work around HTTP's limitations, but it comes with trade-offs. When you're forced to use stateless_http=True or json_response=True, you're essentially telling the transport to operate within HTTP's constraints rather than working around them.



Understanding these limitations helps you make informed decisions about:

- Which transport to use for different deployment scenarios
- How to design your MCP server to gracefully handle HTTP constraints
- When to accept reduced functionality for the benefits of remote hosting

The key is knowing that these restrictions exist and planning your MCP server architecture accordingly. If your application heavily relies on server-initiated requests or real-time notifications, you may need to reconsider your transport choice or implement alternative communication patterns.

StreamableHTTP in depth

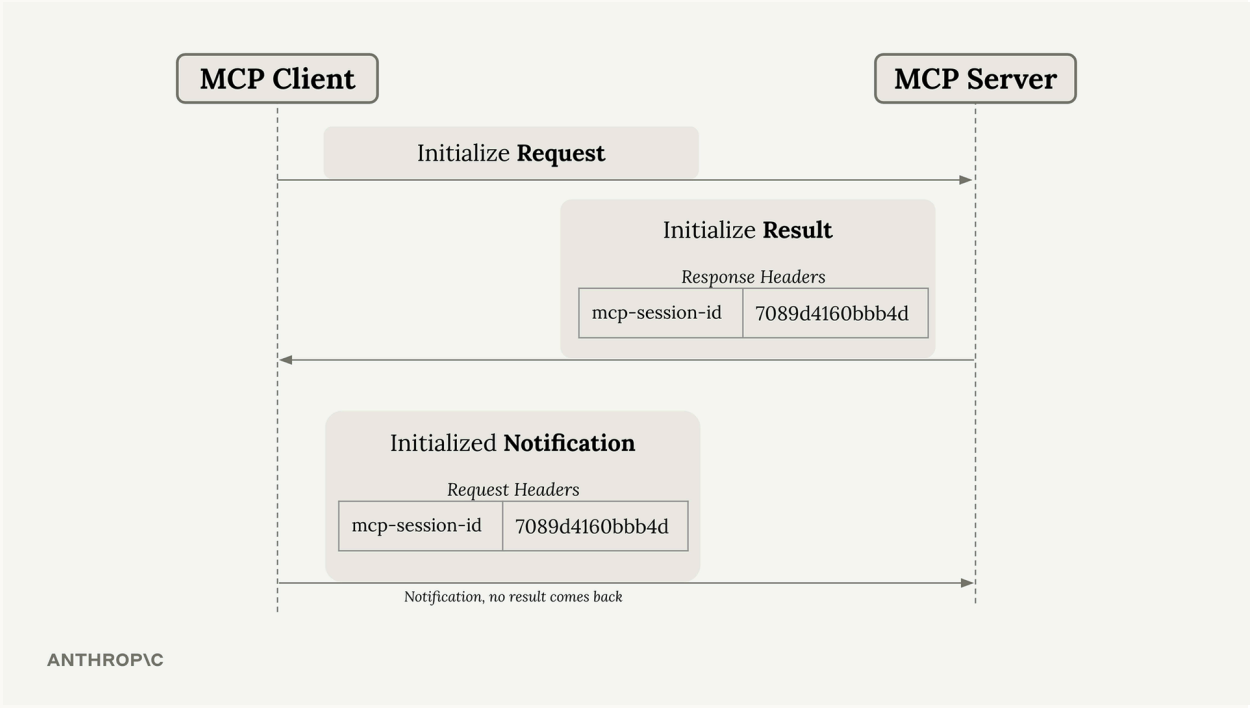
Summary
StreamableHTTP is MCP's solution to a fundamental problem: some MCP functionality requires the server to make requests to the client, but HTTP makes this challenging. Let's explore how StreamableHTTP works around this limitation and when you might need to break that workaround.

The Core Problem

Some MCP features like sampling, notifications, and logging rely on the server initiating requests to the client. However, HTTP is designed for clients to make requests to servers, not the other way around. StreamableHTTP solves this with a clever workaround using Server-Sent Events (SSE).

How StreamableHTTP Works

The magic happens through a multi-step process that establishes persistent connections between client and server.



Initial Connection Setup

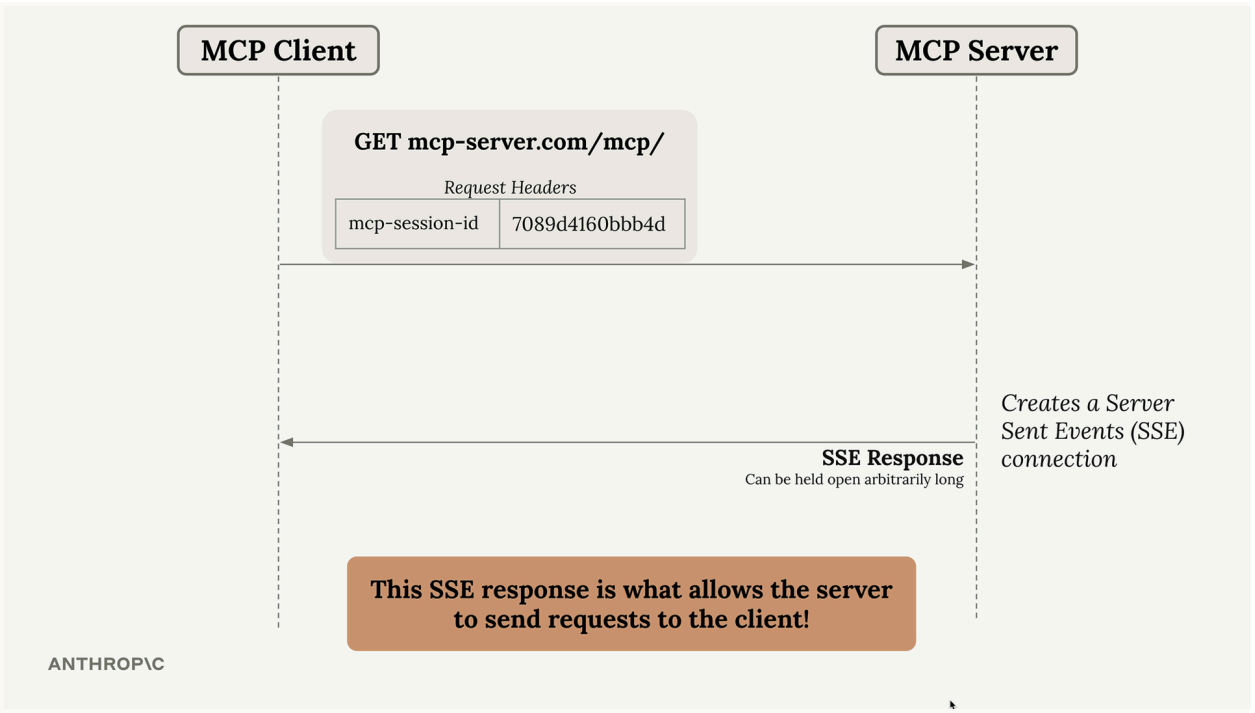
The process starts like any MCP connection:

- Client sends an Initialize Request to the server
- Server responds with an Initialize Result that includes a special mcp-session-id header
- Client sends an Initialized Notification with the session ID

This session ID is crucial - it uniquely identifies the client and must be included in all future requests.

The SSE Workaround

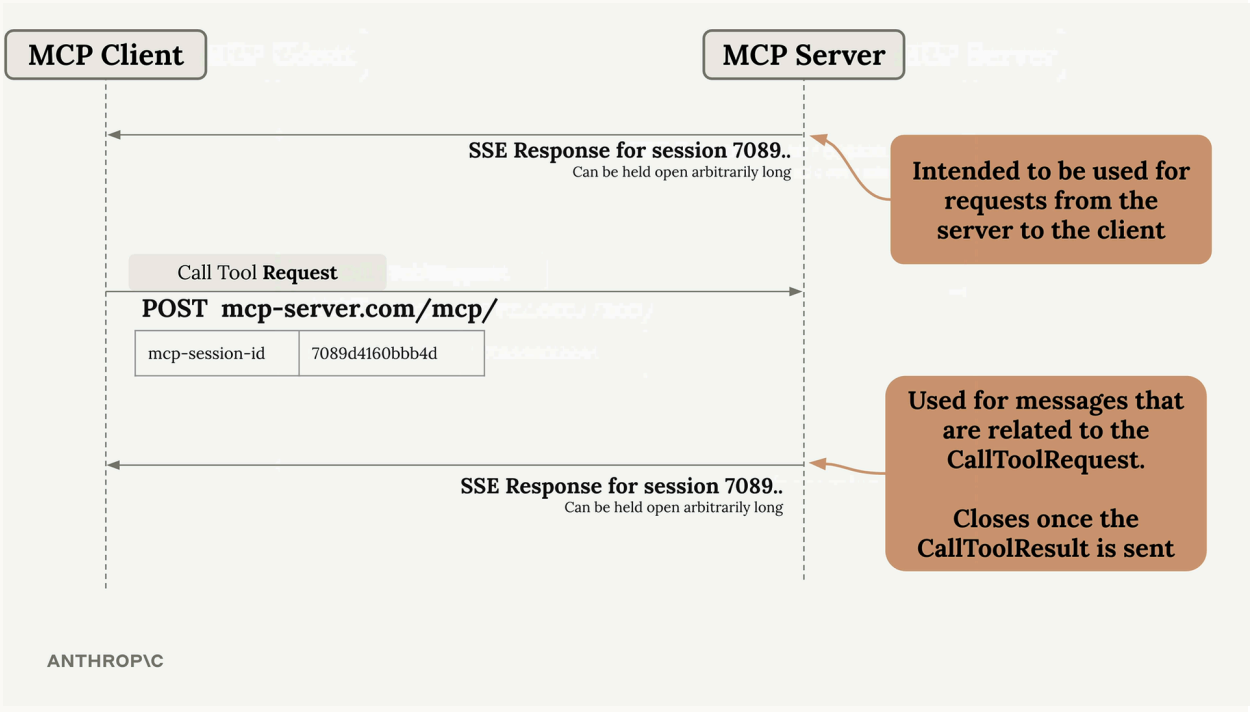
After initialization, the client can make a GET request to establish a Server-Sent Events connection. This creates a long-lived HTTP response that the server can use to stream messages back to the client at any time.



This SSE connection is the key to allowing server-to-client communication. The server can now send requests, notifications, and other messages through this persistent channel.

Tool Calls and Dual SSE Connections

When the client makes a tool call, things get more complex. The system creates two separate SSE connections:

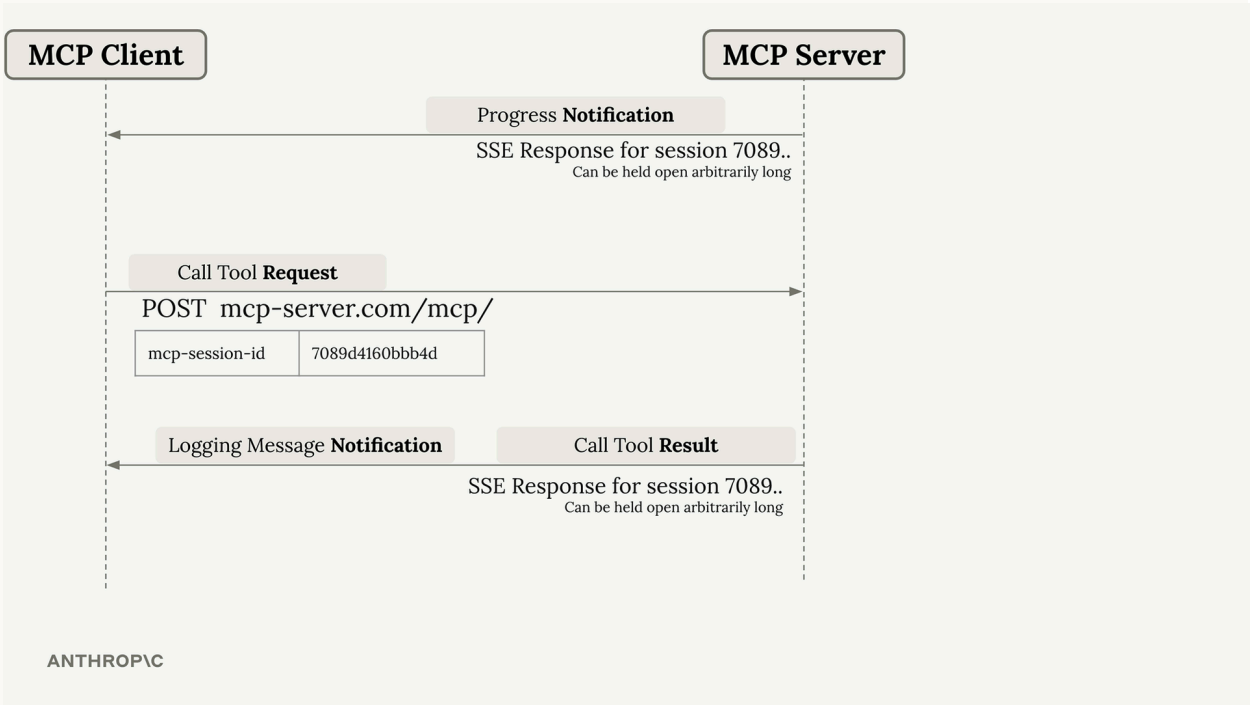


- Primary SSE Connection: Used for server-initiated requests and stays open indefinitely
- Tool-Specific SSE Connection: Created for each tool call and closes automatically when the tool result is sent

Message Routing

Different types of messages get routed through different connections:

- Progress notifications: Sent through the primary SSE connection
- Logging messages and tool results: Sent through the tool-specific SSE connection



Configuration Flags That Break the Workaround

StreamableHTTP includes two important configuration options:

- stateless_http
- json_response

Setting these to True can break the SSE workaround mechanism. You might want to enable these flags in certain scenarios, but doing so limits the full MCP functionality that depends on server-to-client communication.

Key Takeaways

StreamableHTTP is more complex than other MCP transports because it has to work around HTTP's limitations. The SSE-based workaround enables full MCP functionality over HTTP, but understanding the dual-connection model is crucial for debugging and optimization.

When building MCP applications with StreamableHTTP, remember that session IDs are required for all requests after initialization, and the system automatically manages multiple SSE connections to handle different types of server-to-client communication.

State and the StreamableHTTP transport

Downloads

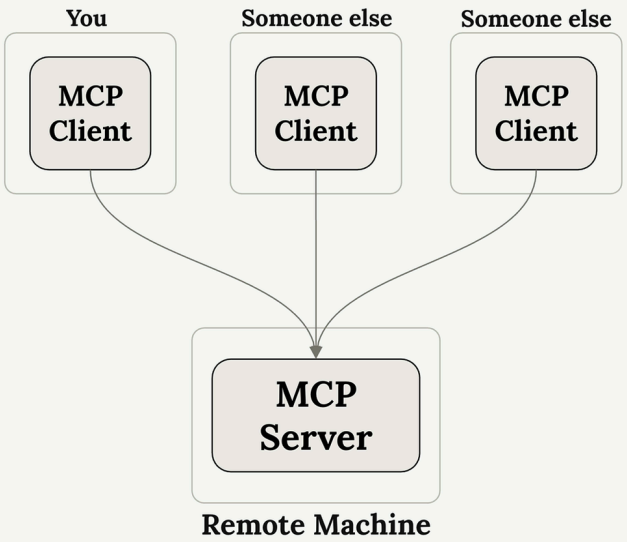
Transport-http.zip

Summary

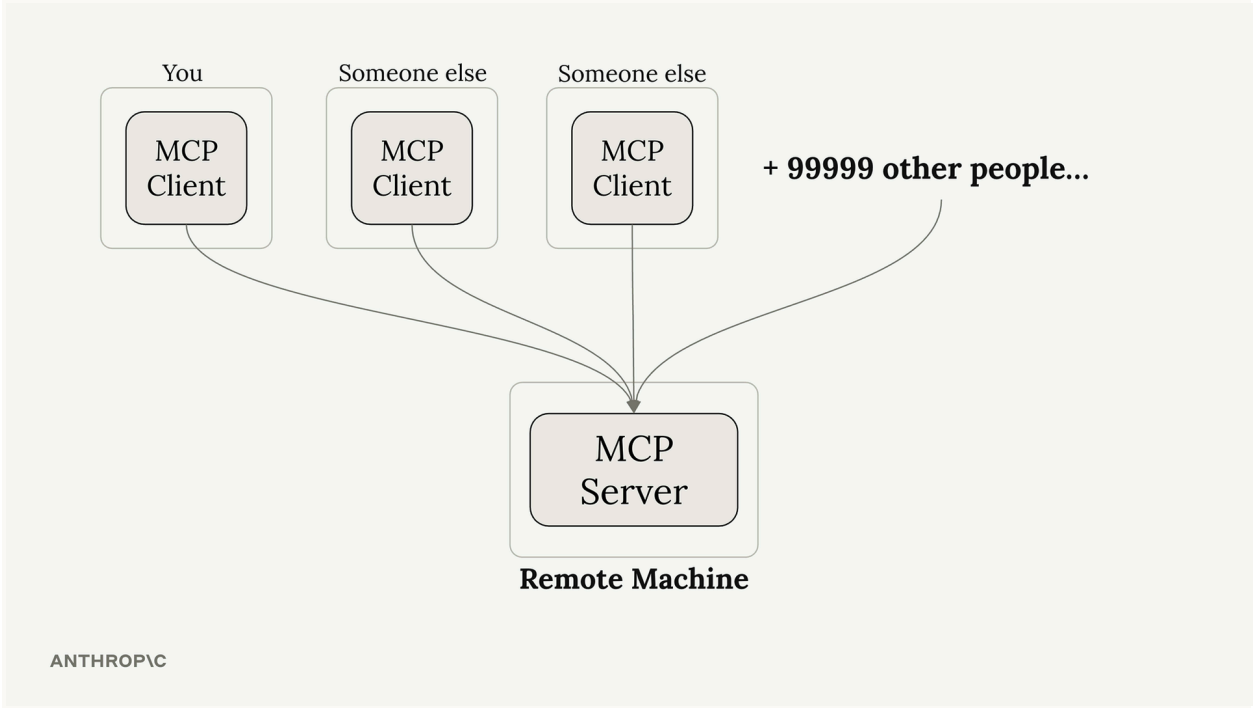
The stateless_http and json_response flags in MCP servers control fundamental aspects of how your server behaves. Understanding when and why to use them is crucial, especially if you're planning to scale your server or deploy it in production.

When You Need Stateless HTTP

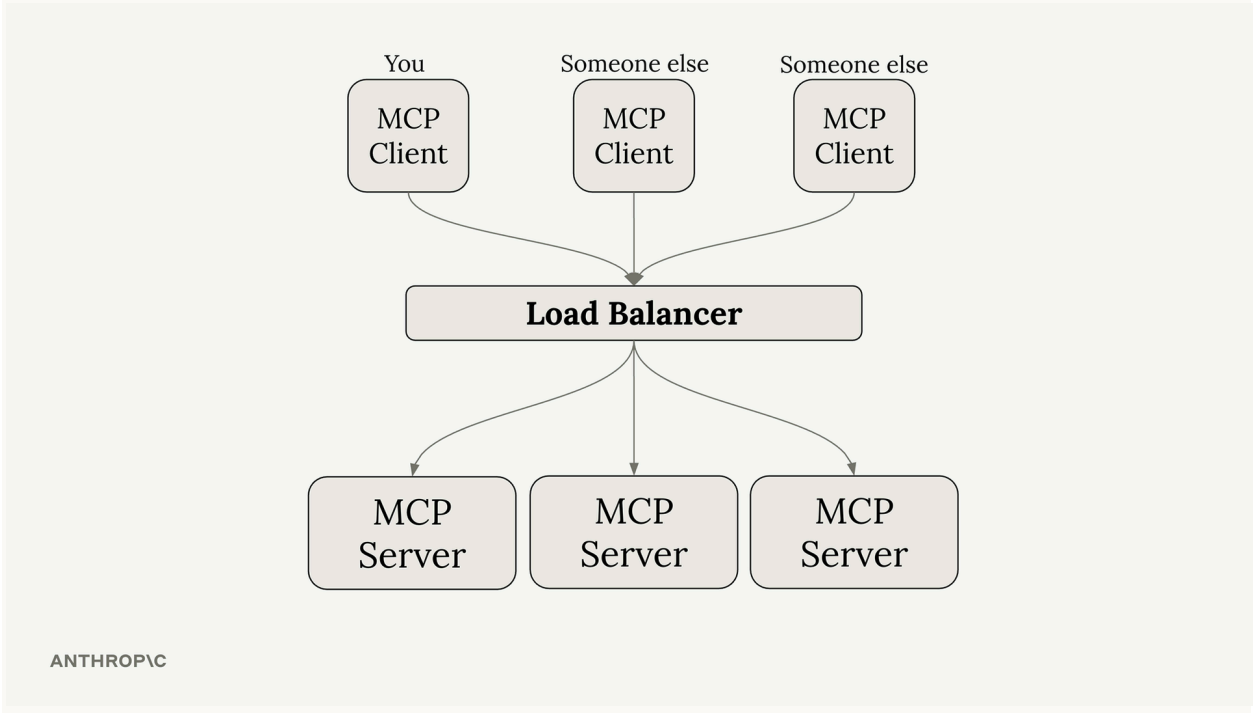
Imagine you build an MCP server that becomes popular. Initially, you might have just a few clients connecting to a single server instance:



As your server grows, you might have thousands of clients trying to connect. Running a single server instance won't scale to handle all that traffic:

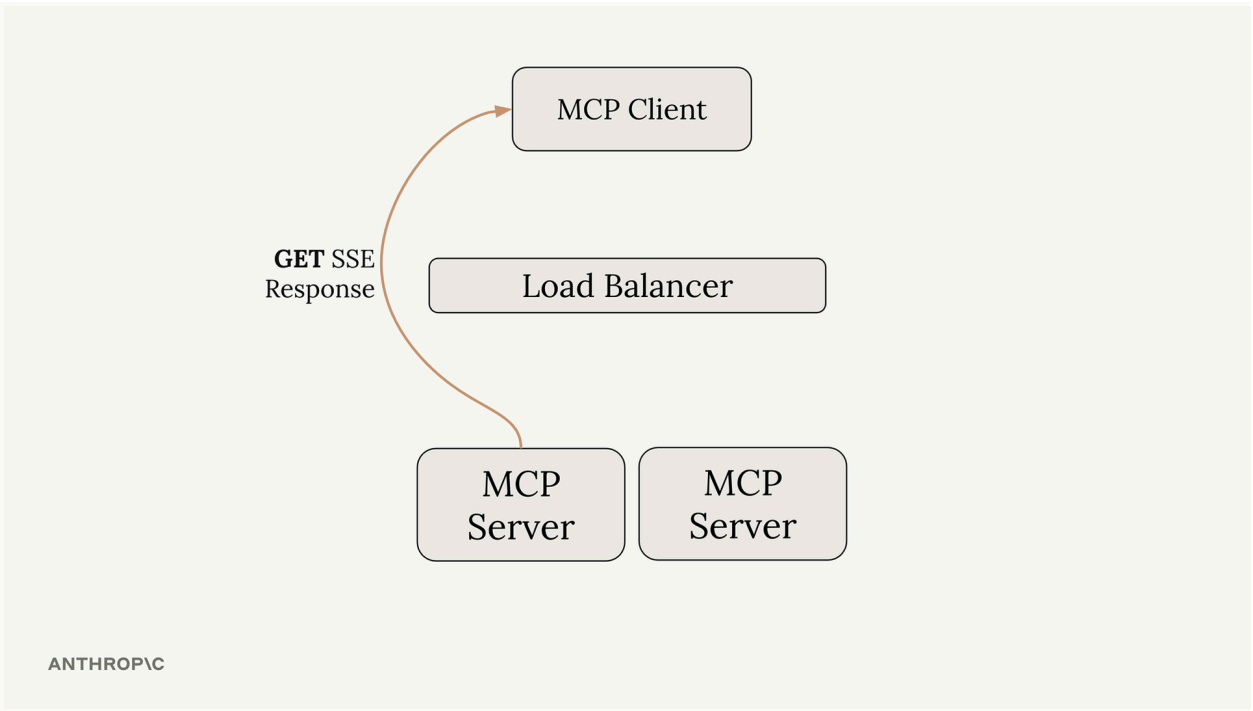


The typical solution is horizontal scaling - running multiple server instances behind a load balancer:

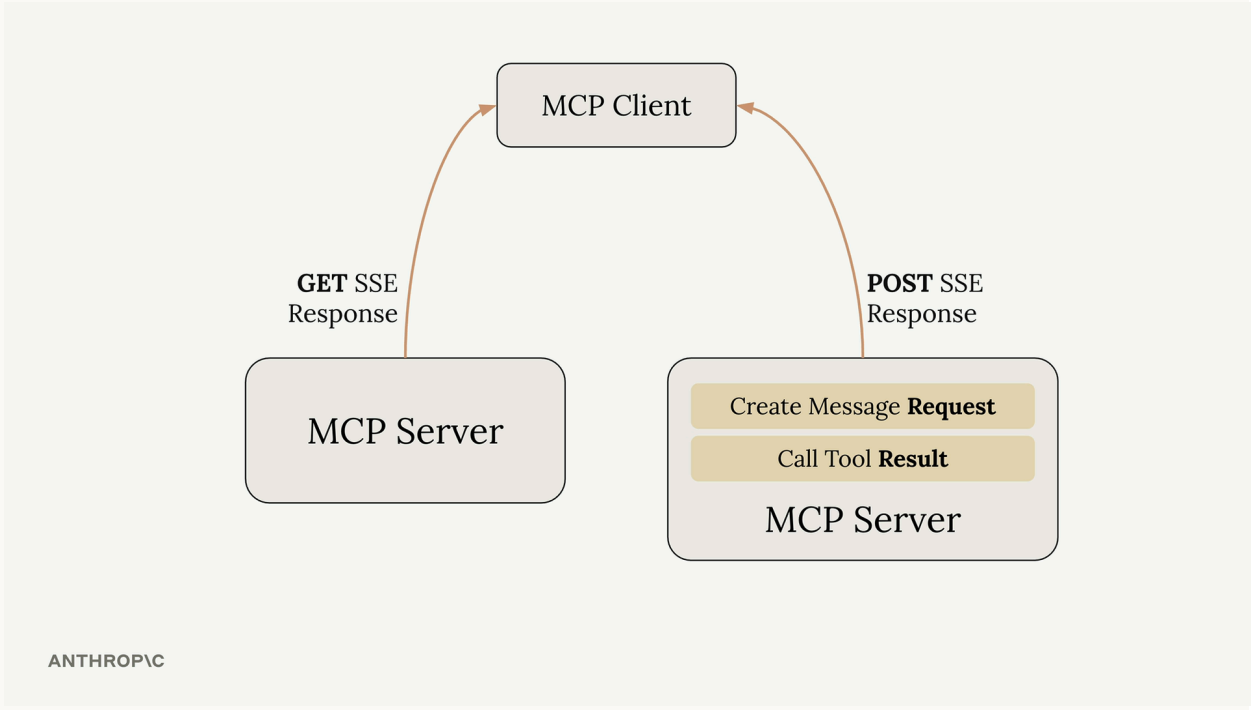


But here's where things get complicated. Remember that MCP clients need two separate connections:

- A GET SSE connection for receiving server-to-client requests
- POST requests for calling tools and receiving responses

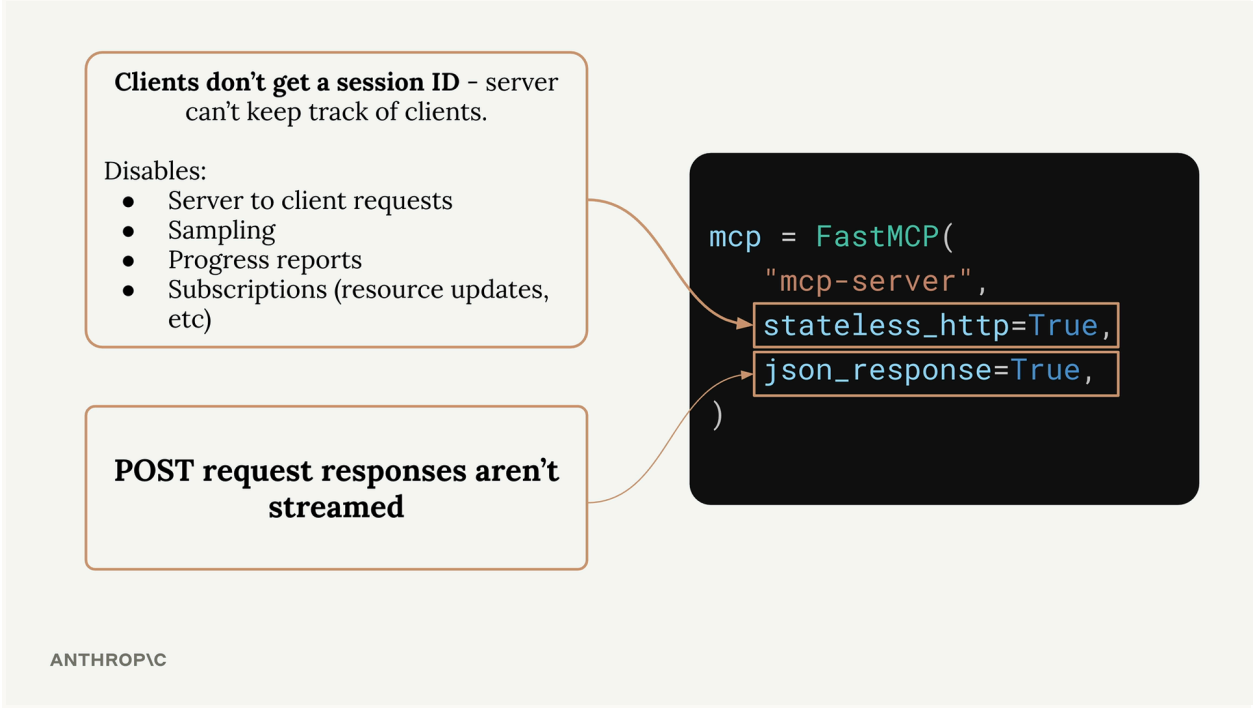


With a load balancer, these requests might get routed to different server instances. If your tool needs to use Claude (through sampling), the server handling the POST request would need to coordinate with the server handling the GET SSE connection. This creates a complex coordination problem between servers.



How Stateless HTTP Solves This

Setting stateless_http=True eliminates this coordination problem, but with significant trade-offs:



When stateless HTTP is enabled:

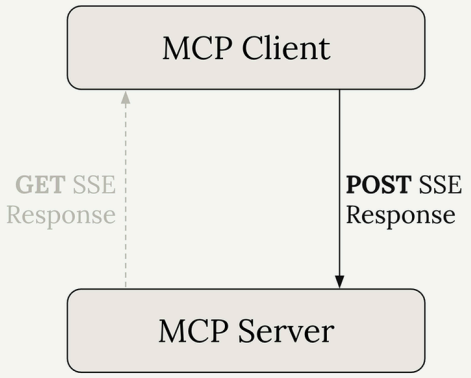
- Clients don't get session IDs - the server can't track individual clients
- No server-to-client requests - the GET SSE pathway becomes unavailable
- No sampling - can't use Claude or other AI models
- No progress reports - can't send progress updates during long operations
- No subscriptions - can't notify clients about resource updates

However, there's one benefit: client initialization is no longer required. Clients can make requests directly without the initial handshake process.

No session id?

- GET SSE response can't be used anymore - the server can't figure out how to pair that response pathway with any incoming request
- Without the GET SSE response, we can't use sampling, progress logging, subscriptions
- In stateless mode, client initialization is no longer required

ANTHROP\C



Understanding JSON Response

The json_response=True flag is simpler - it just disables streaming for POST request responses. Instead of getting multiple SSE messages as a tool executes, you get only the final result as plain JSON.

With streaming disabled:

- No intermediate progress messages
- No log statements during execution
- Just the final tool result

When to Use These Flags

Use stateless HTTP when:

- You need horizontal scaling with load balancers
- You don't need server-to-client communication
- Your tools don't require AI model sampling
- You want to minimize connection overhead

Use JSON response when:

- You don't need streaming responses
- You prefer simpler, non-streaming HTTP responses

- You're integrating with systems that expect plain JSON

Development vs Production

If you're developing locally with standard I/O transport but planning to deploy with HTTP transport, test with the same transport you'll use in production. The behavior differences between stateful and stateless modes can be significant, and it's better to catch any issues during development rather than after deployment.

These flags fundamentally change how your MCP server operates, so choose them based on your specific scaling and functionality requirements.