First cover the Transport and Communication section and then get hands on Core Features.

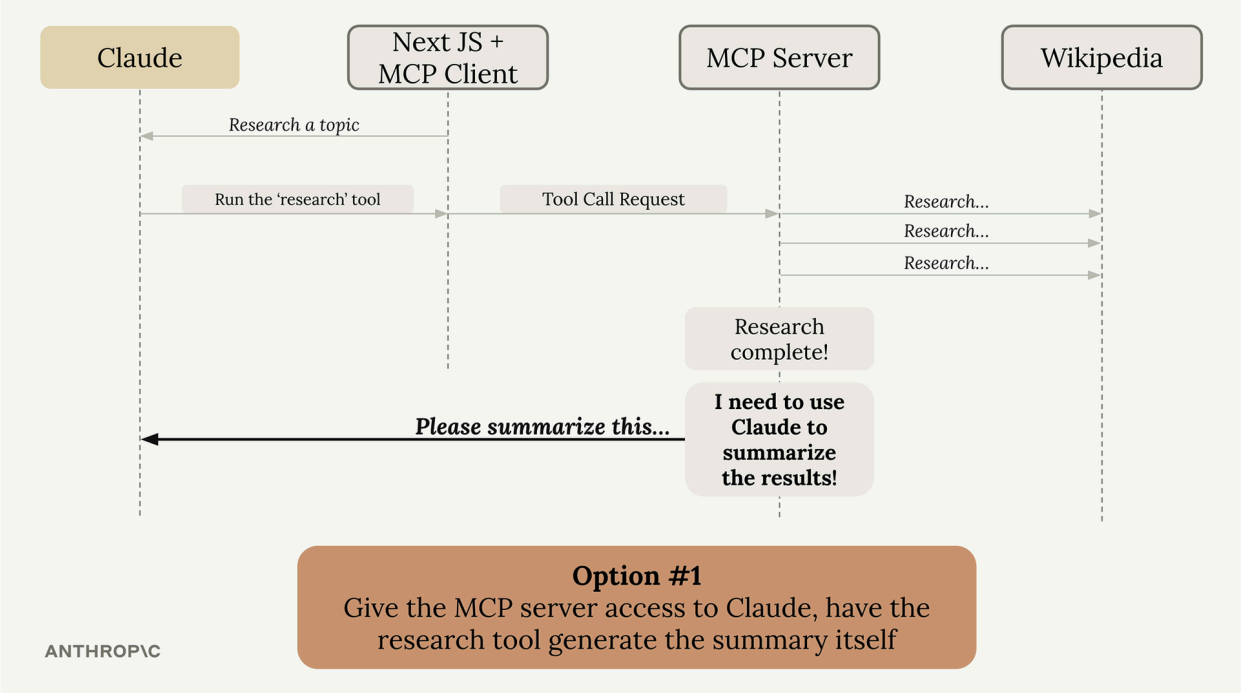
Core MCP features

1. Sampling

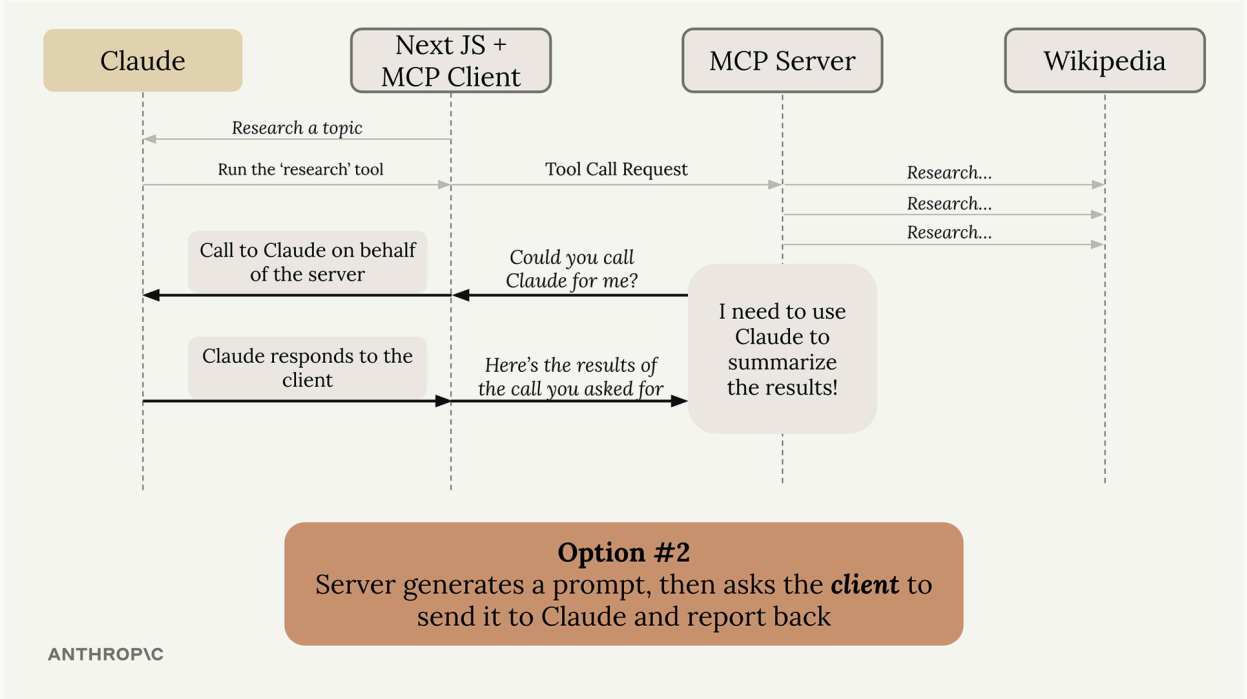
Sampling allows a server to access a language model like Claude through a connected MCP client. Instead of the server directly calling Claude, it asks the client to make the call on its behalf. This shifts the responsibility and cost of text generation from the server to the client.

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

2. Log and progress notifications

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.

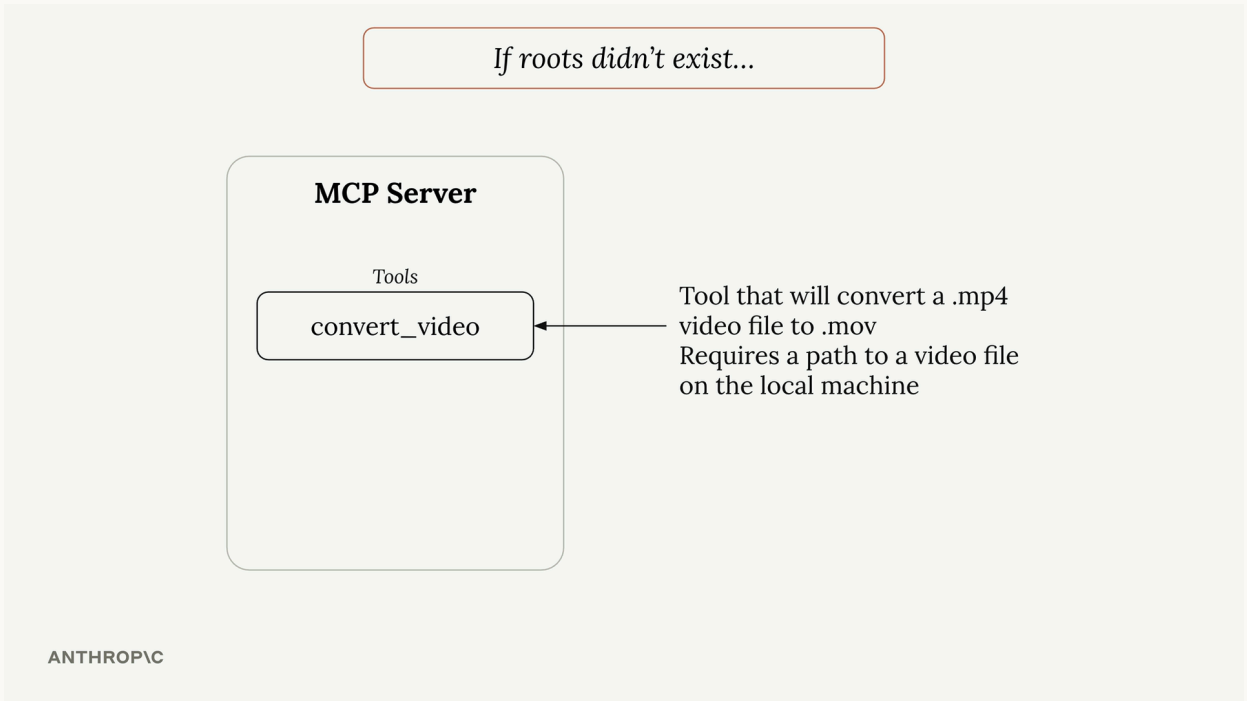
Notifications walkthrough

3. Roots

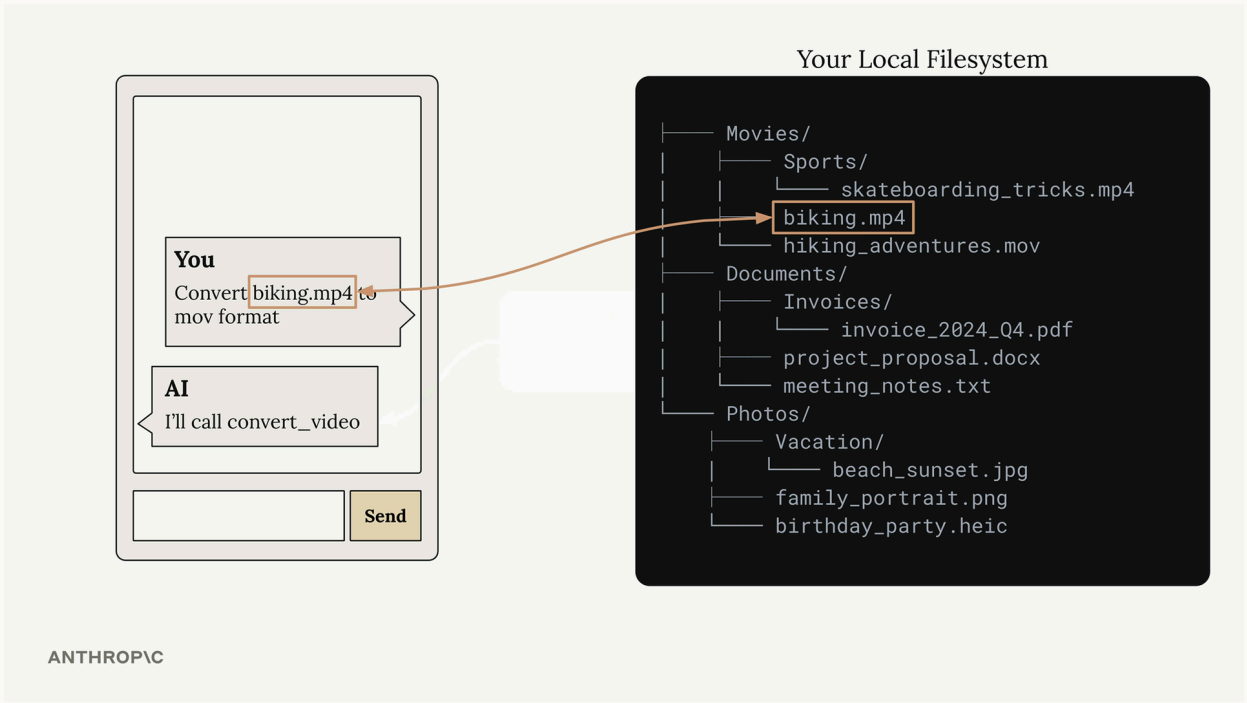
Roots are a way to grant MCP servers access to specific files and folders on your local machine. Think of them as a permission system that says "Hey, MCP server, you can access these files" - but they do much more than just grant permission.

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

Transports and communication

1. **JSON message types**

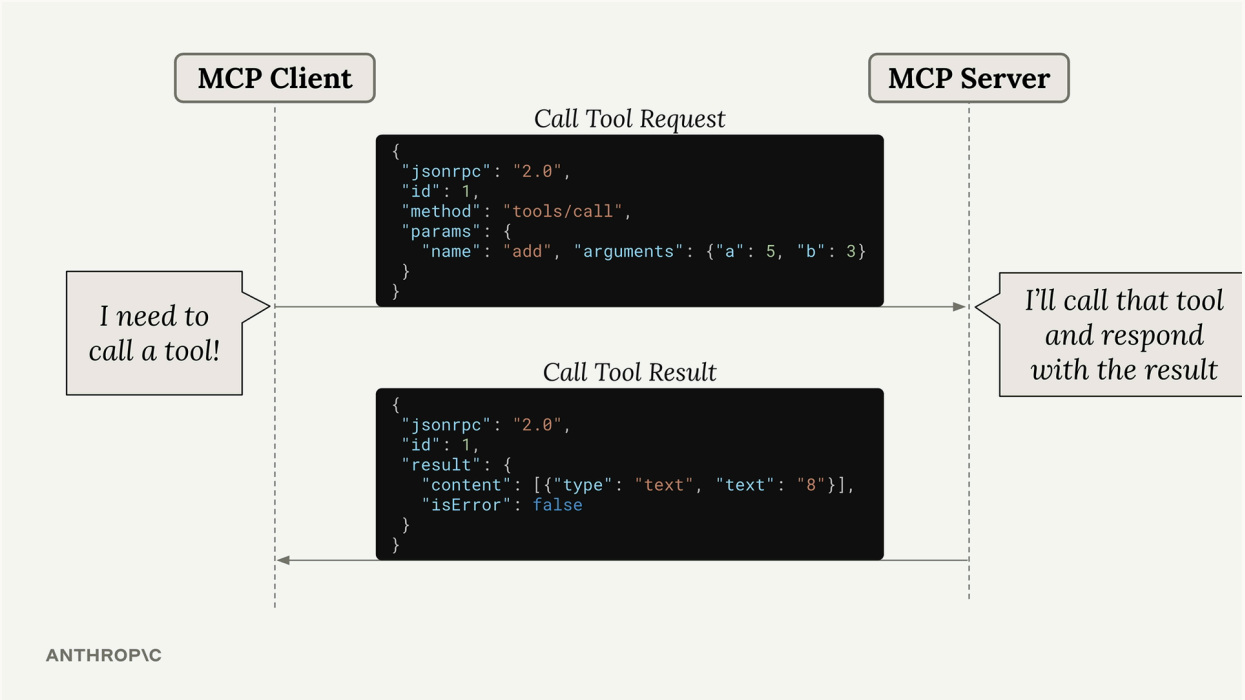
Summary

MCP (Model Context Protocol) uses JSON messages to handle communication between clients and servers. Understanding these message

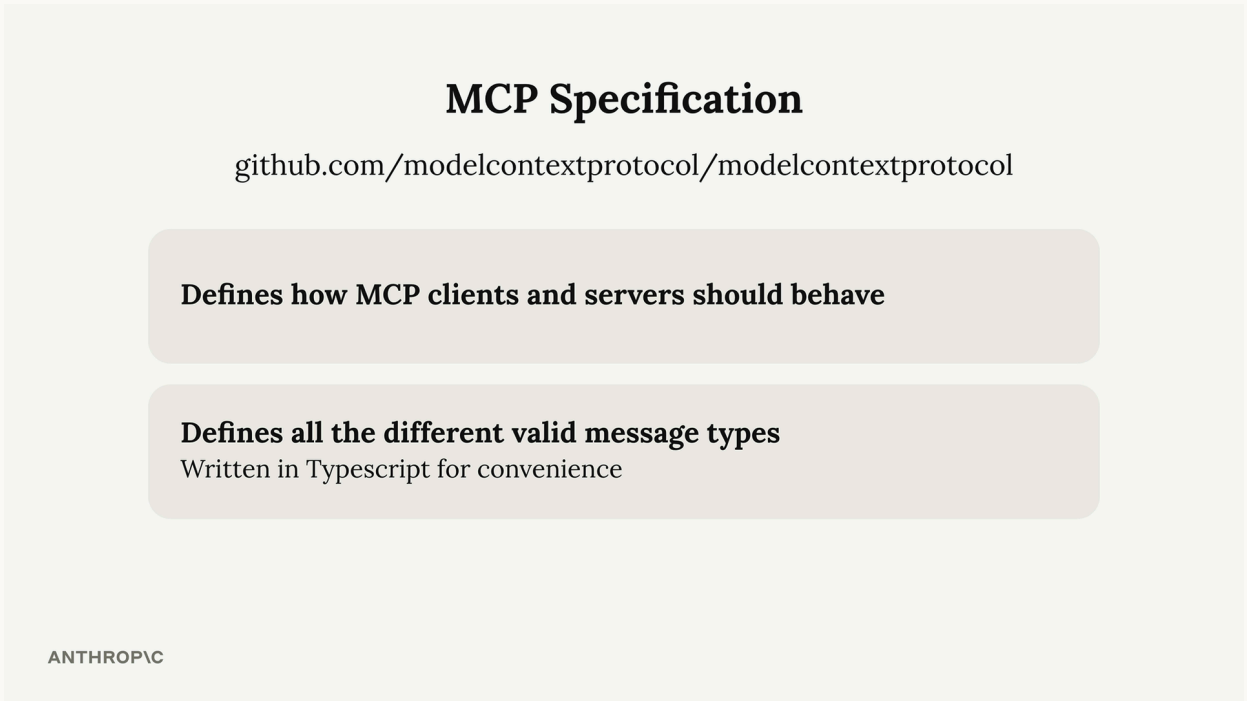
types is crucial for working with MCP, especially when dealing with different transport methods like the streamable HTTP transport.

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.



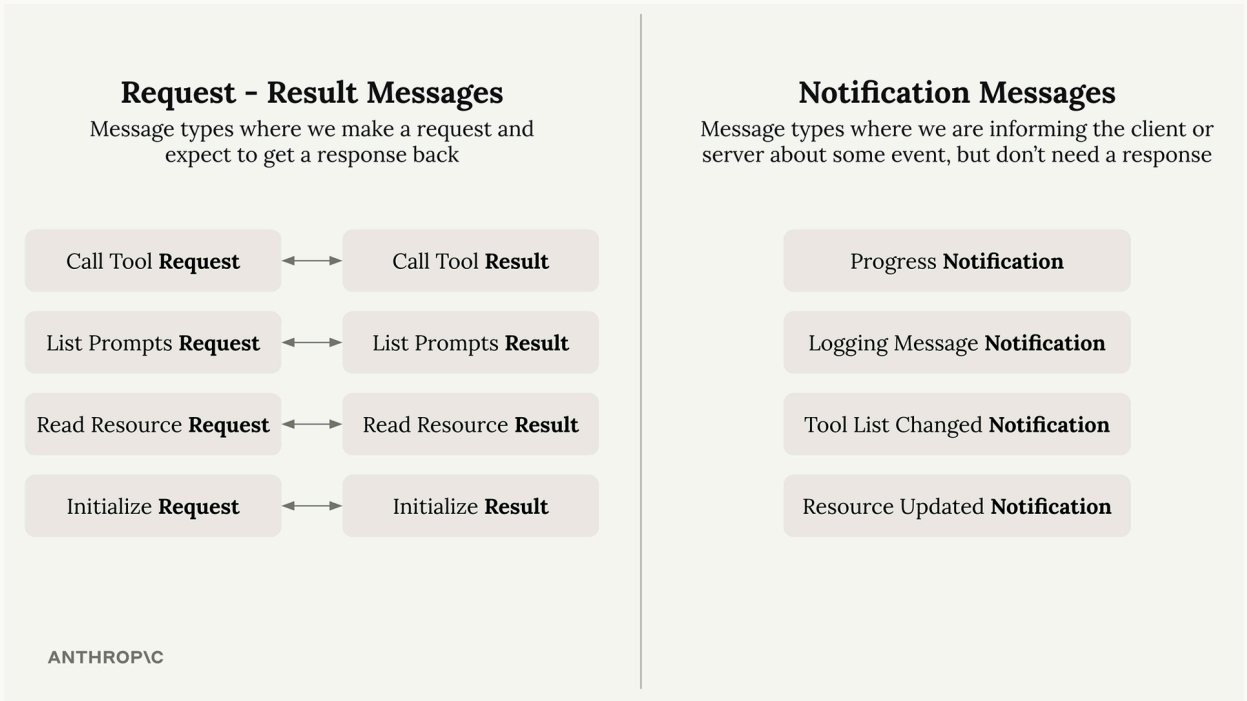
MCP Specification

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.

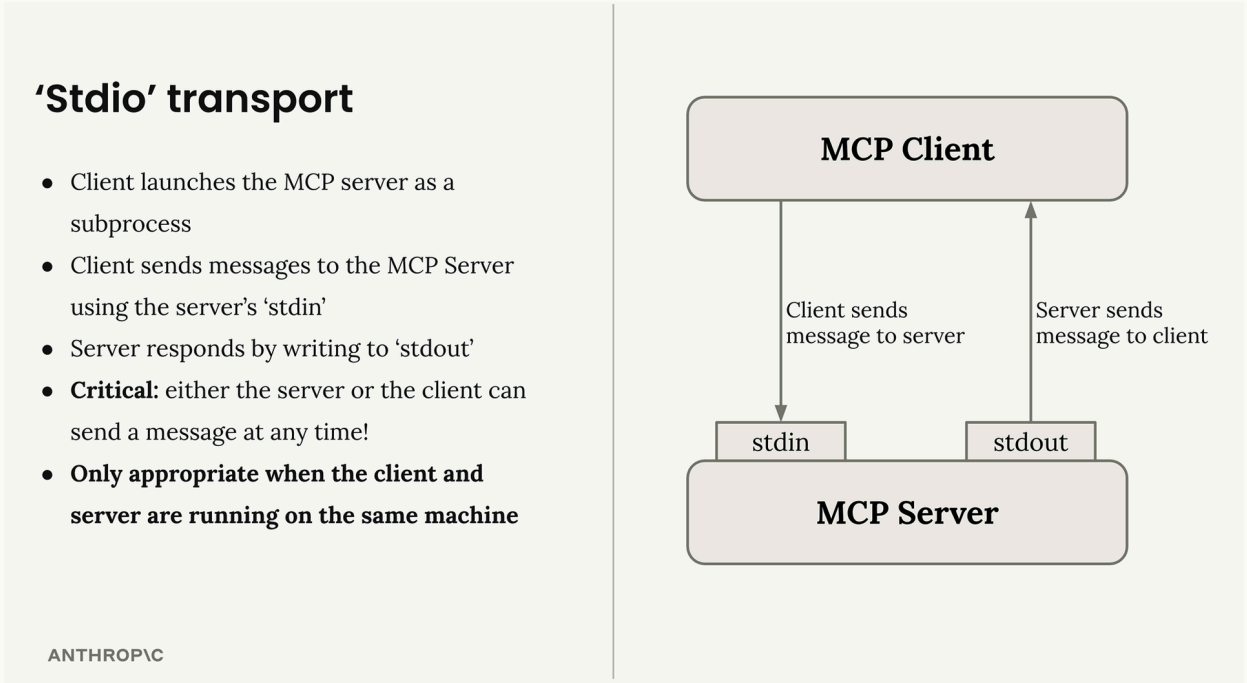
2. The STDIO transport

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.



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

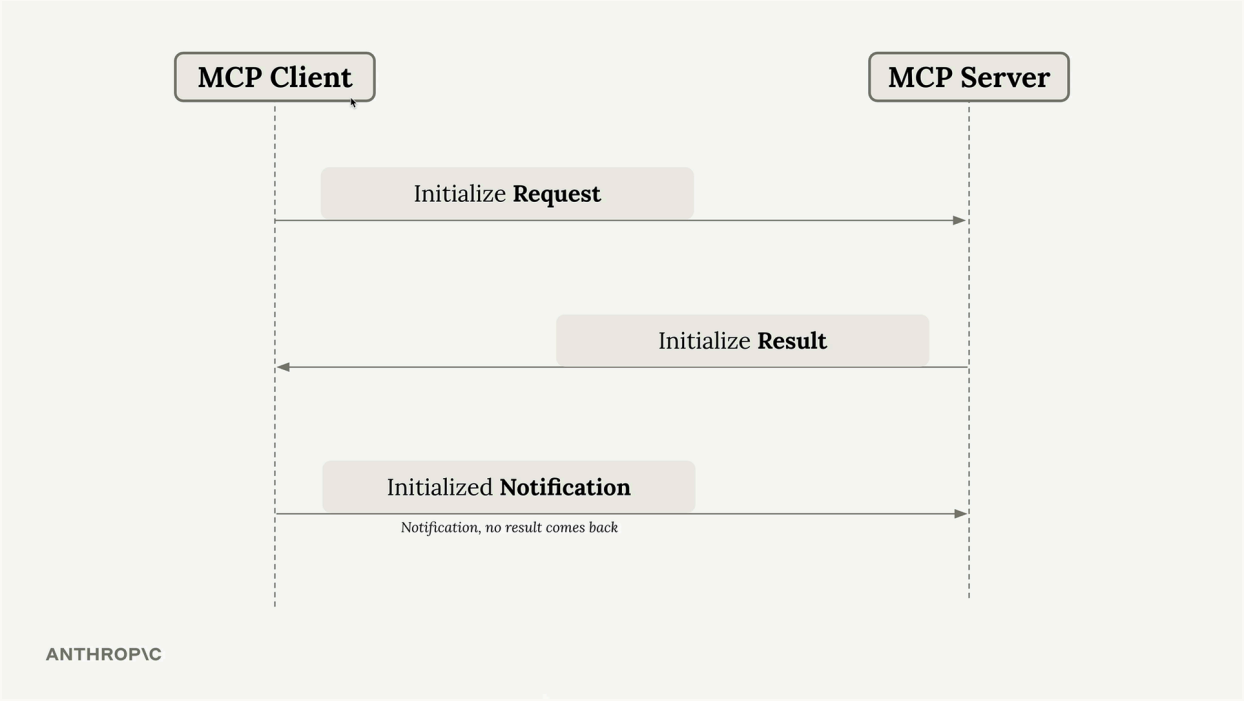
Seeing Stdio in Action

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:

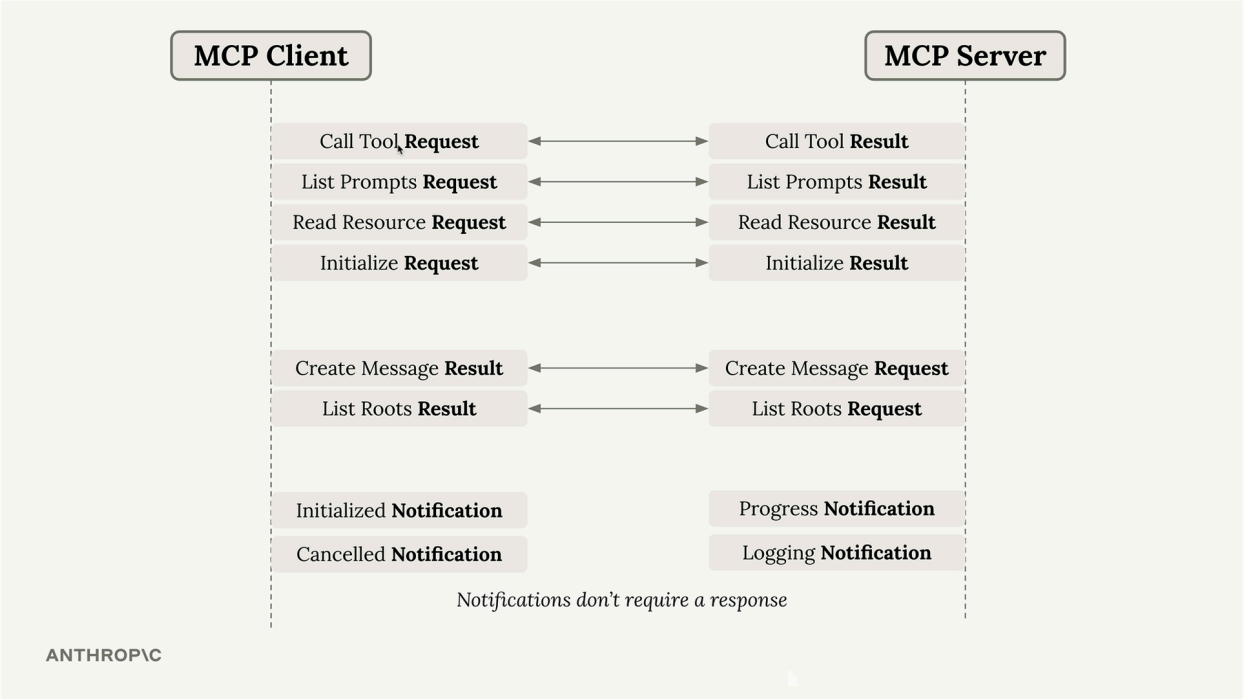


1. Initialize Request - Client sends this first
2. Initialize Result - Server responds with capabilities
3. Initialized Notification - Client confirms (no response expected)

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

Message Types and Flow

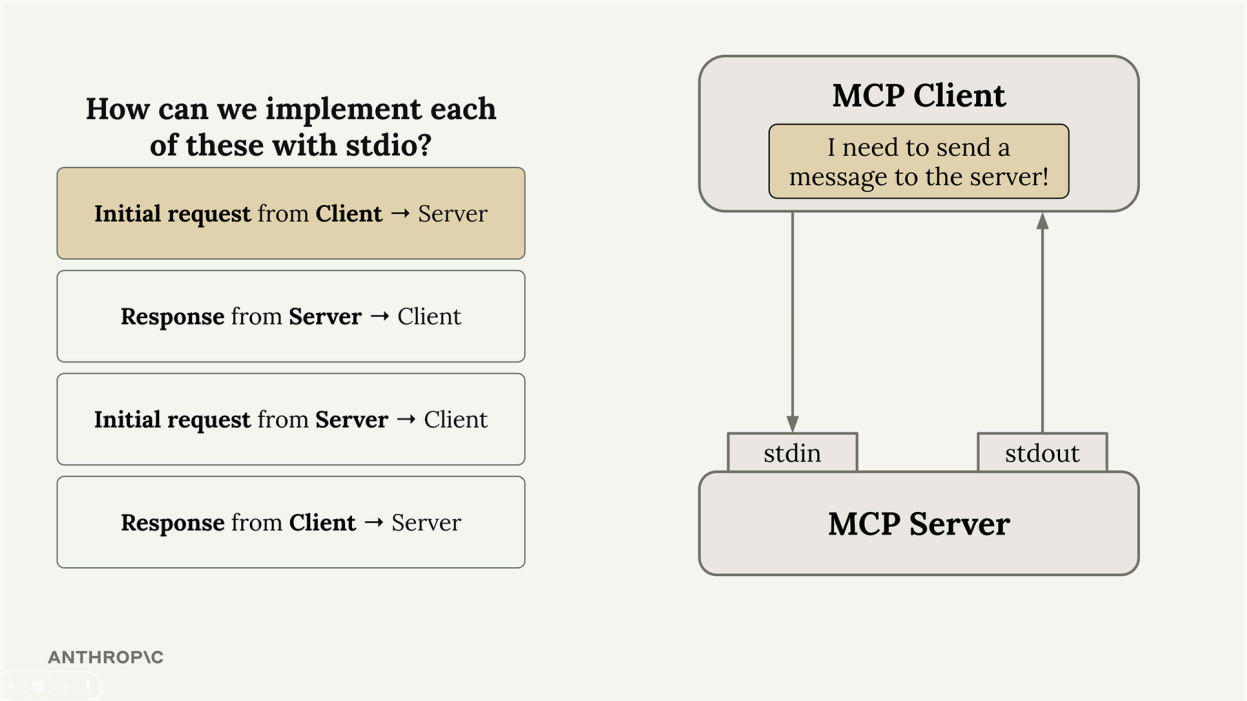
MCP supports various message types that flow in both directions:



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

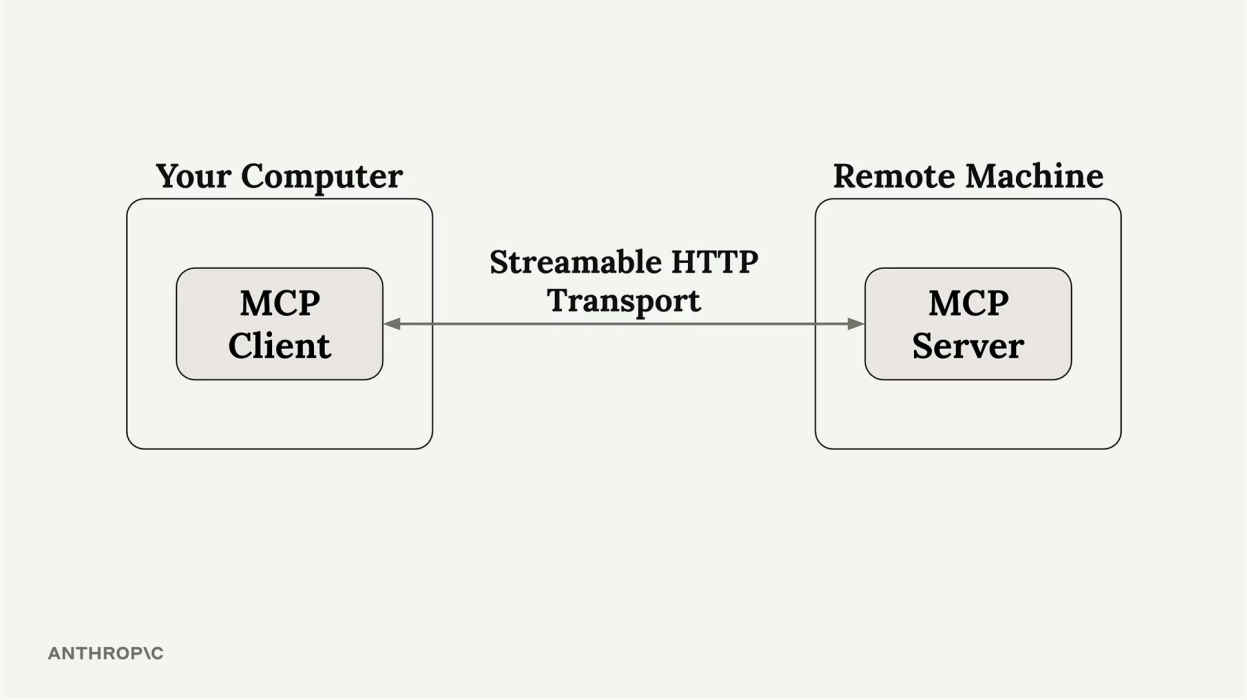
baseline for understanding what full MCP communication looks like before we tackle the constraints of other transport methods.

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.

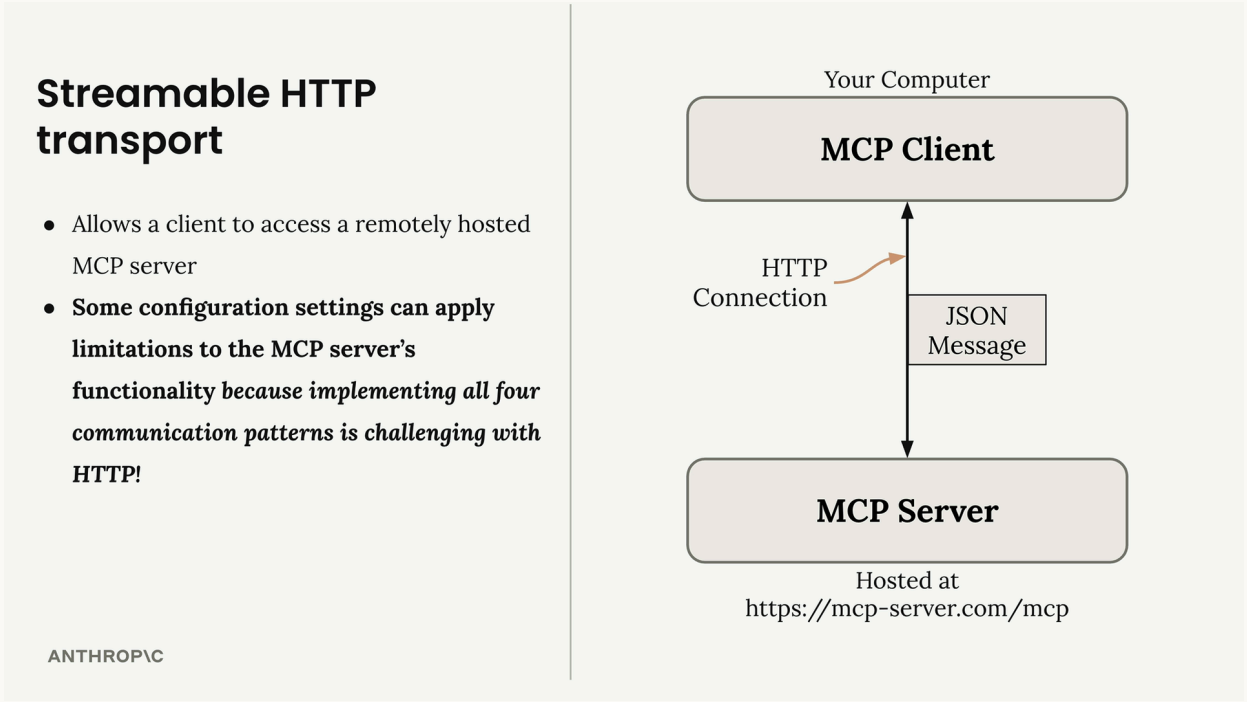
3. The StreamableHTTP transport

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:

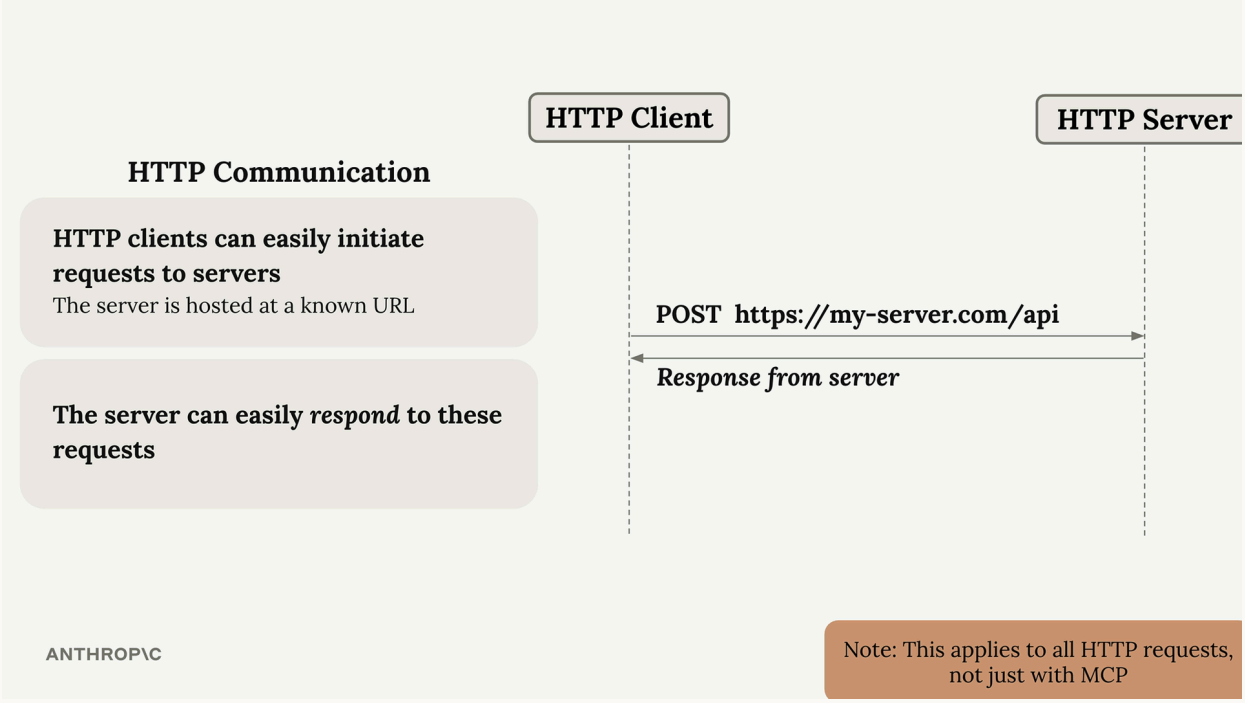
- **stateless_http** - Controls connection state management
- **json_response** - Controls response format handling

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

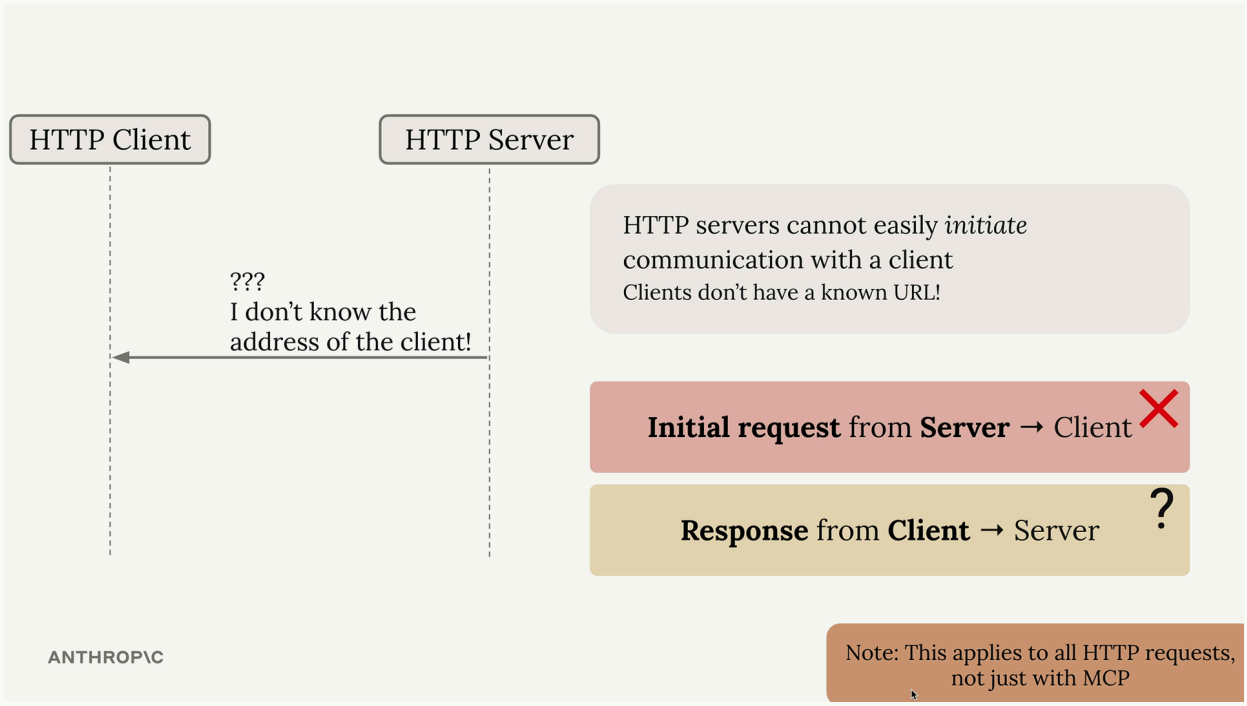
functionality like progress notifications, logging, and server-initiated requests.

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

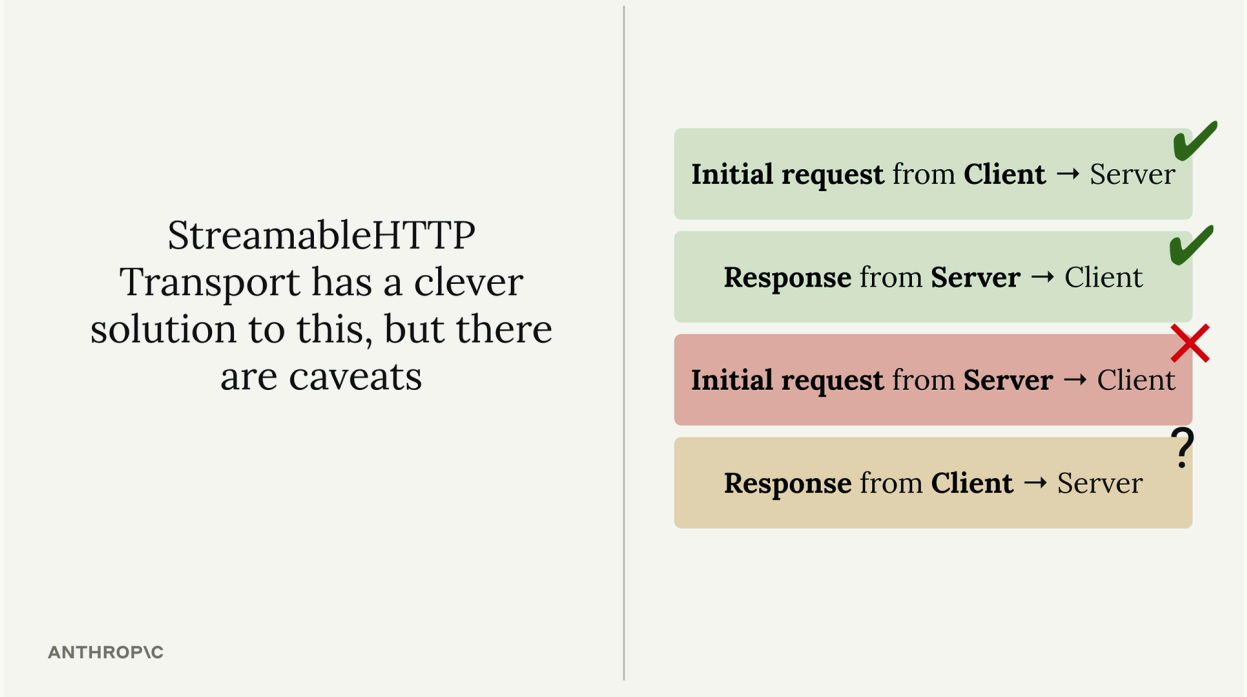
This HTTP limitation impacts specific MCP communication patterns. The following message types become difficult to implement with plain HTTP:

- 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.

4. StreamableHTTP in depth

Summary

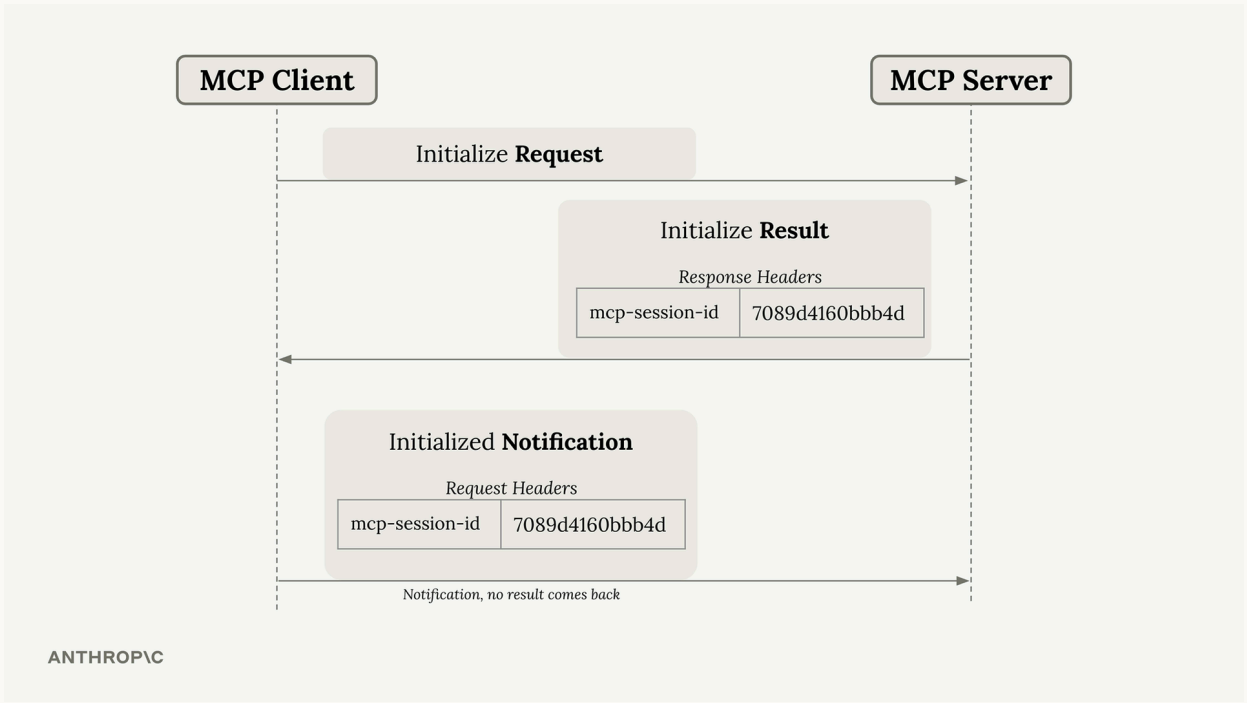
StreamableHTTP is MCP's solution to a fundamental problem: some MCP functionality requires the server to make requests to the client, but HTTP makes this challenging. Let's explore how StreamableHTTP works around this limitation and when you might need to break that workaround.

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

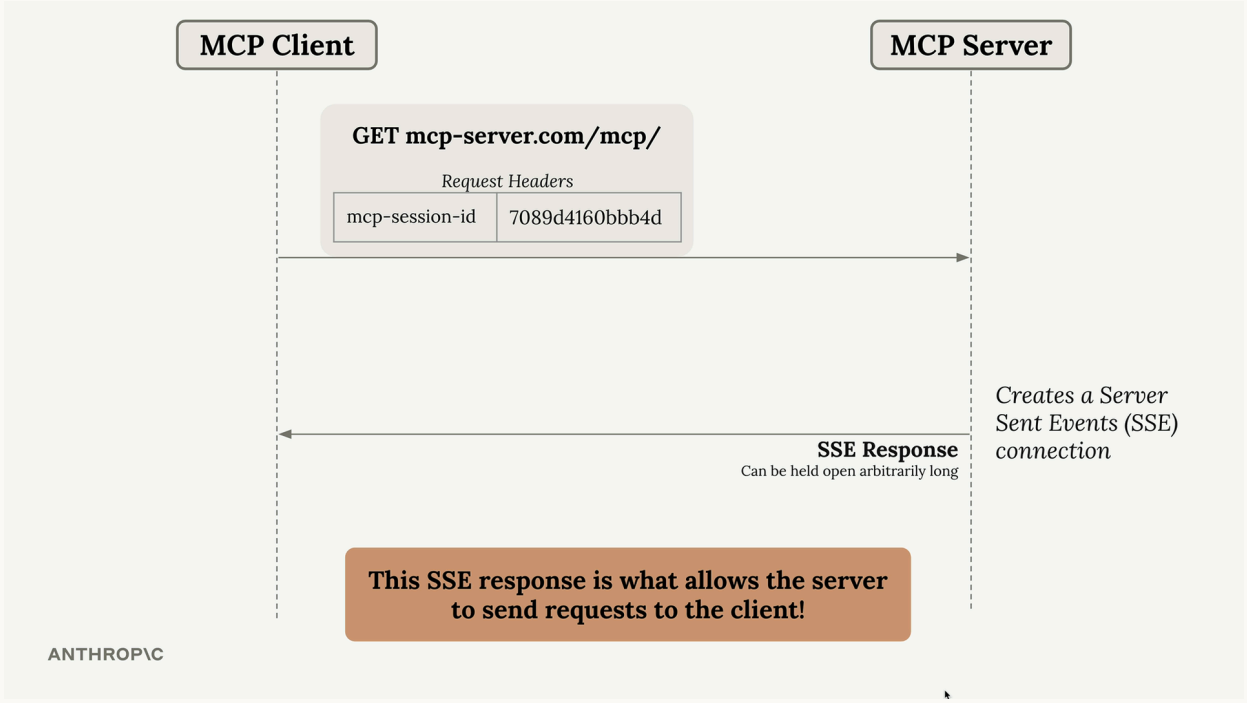
The process starts like any MCP connection:

- 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

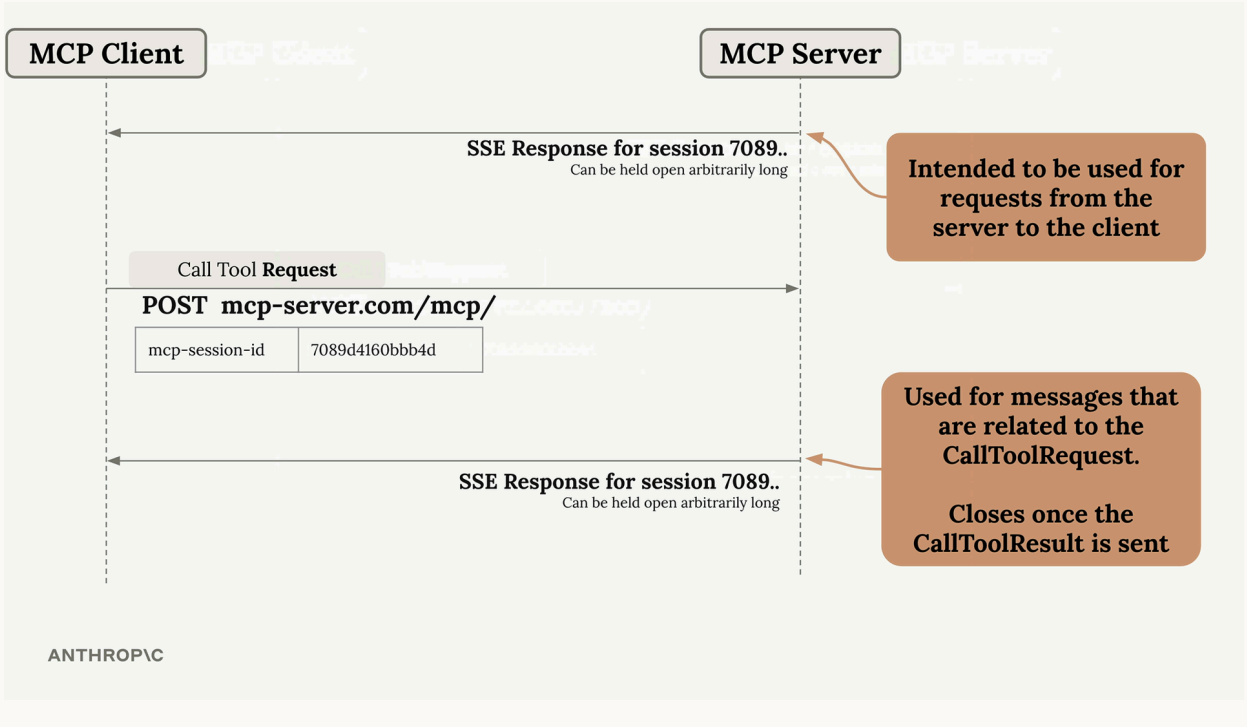
After initialization, the client can make a GET request to establish a Server-Sent Events connection. This creates a long-lived HTTP response that the server can use to stream messages back to the client at any time.



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:

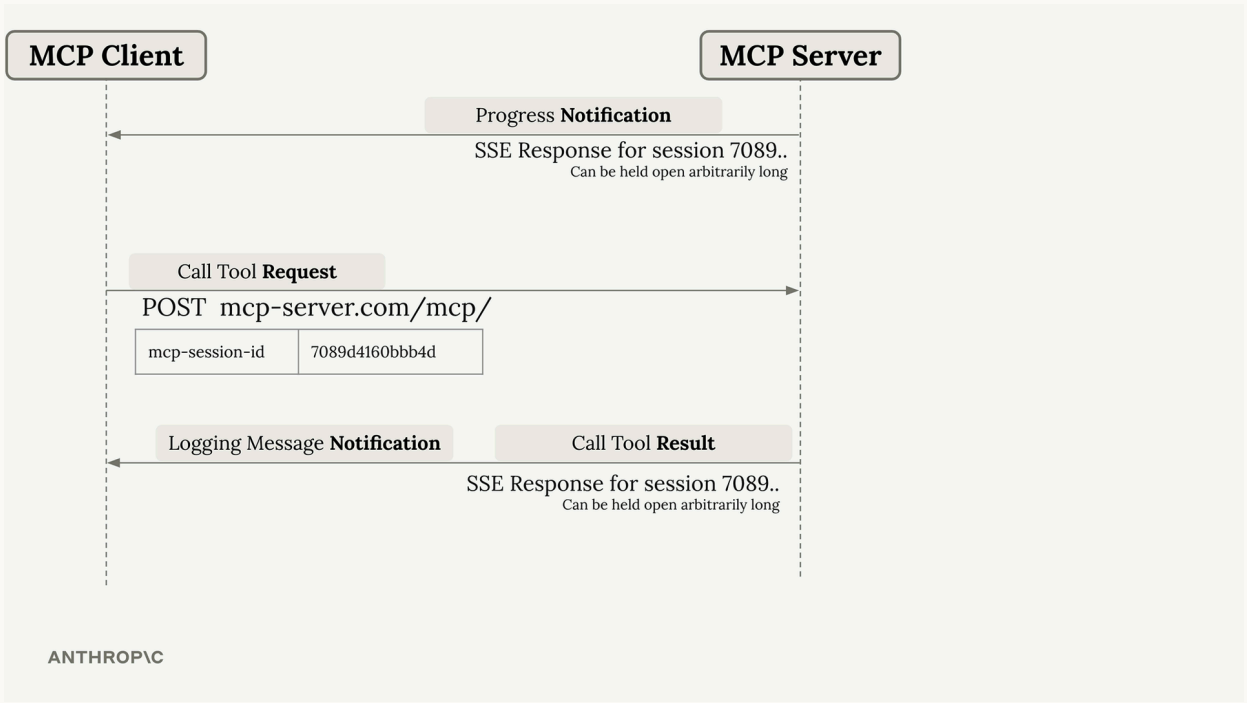


- Primary SSE Connection: Used for server-initiated requests and stays open indefinitely
- Tool-Specific SSE Connection: Created for each tool call and closes automatically when the tool result is sent

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.

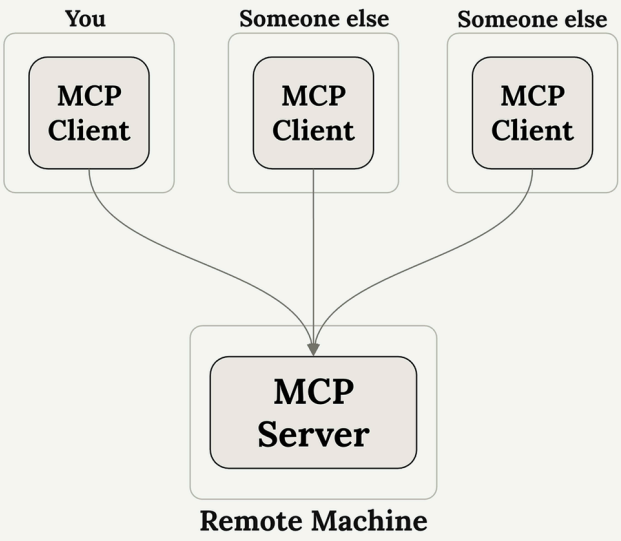
5. State and the StreamableHTTP transport

Summary

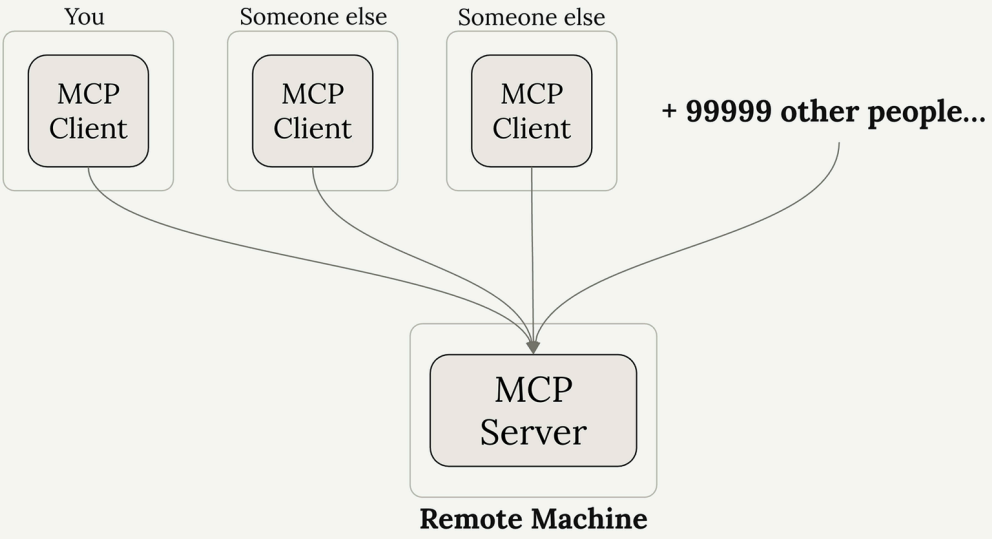
The **stateless_http** and **json_response** flags in MCP servers control fundamental aspects of how your server behaves. Understanding when and why to use them is crucial, especially if you're planning to scale your server or deploy it in production.

When You Need Stateless HTTP

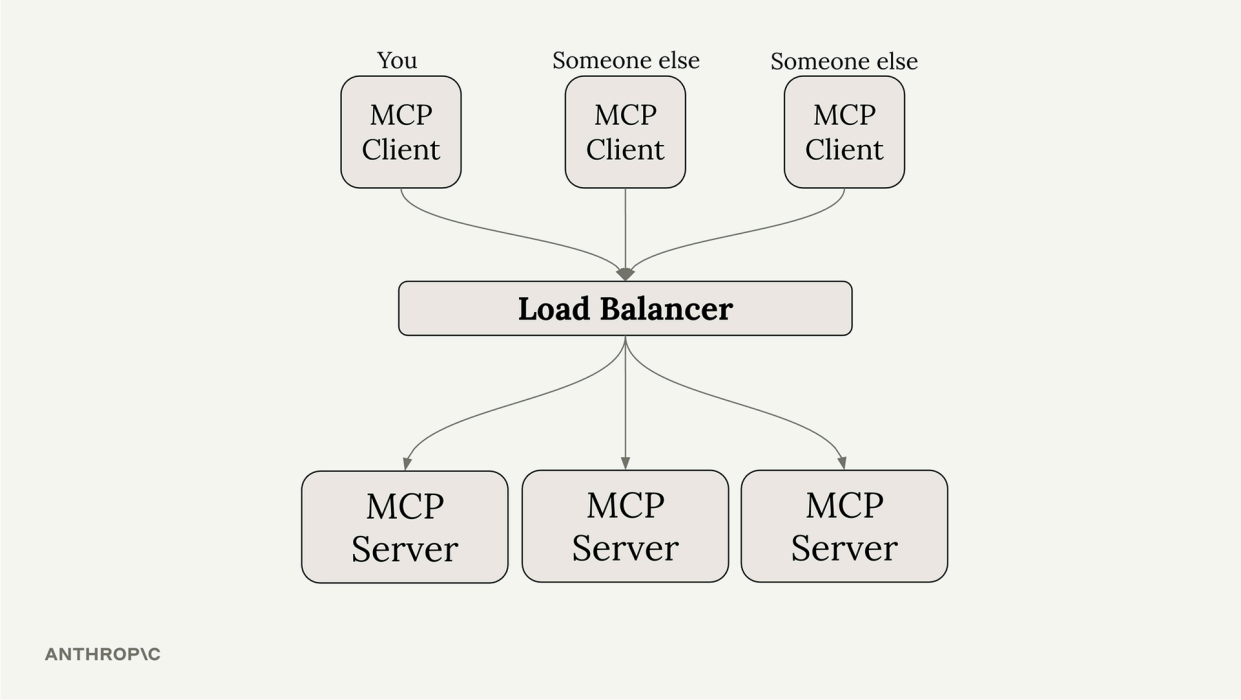
Imagine you build an MCP server that becomes popular. Initially, you might have just a few clients connecting to a single server instance:



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:

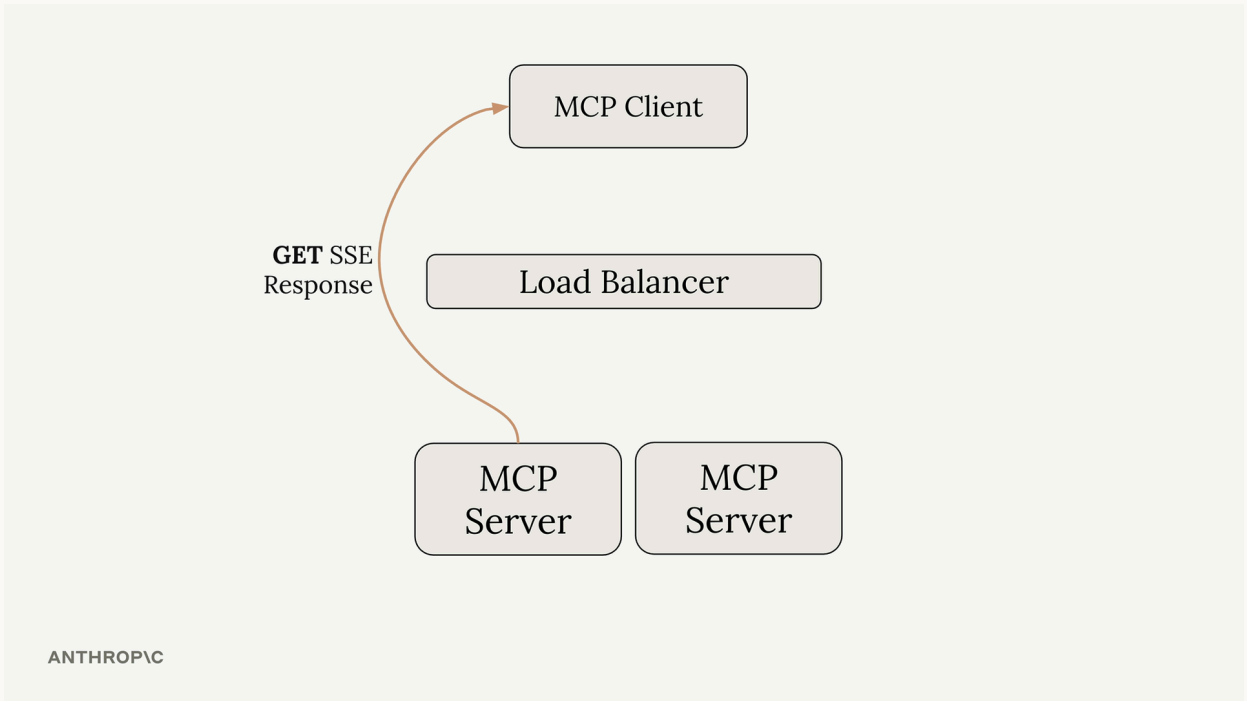


The typical solution is horizontal scaling - running multiple server instances behind a load balancer:

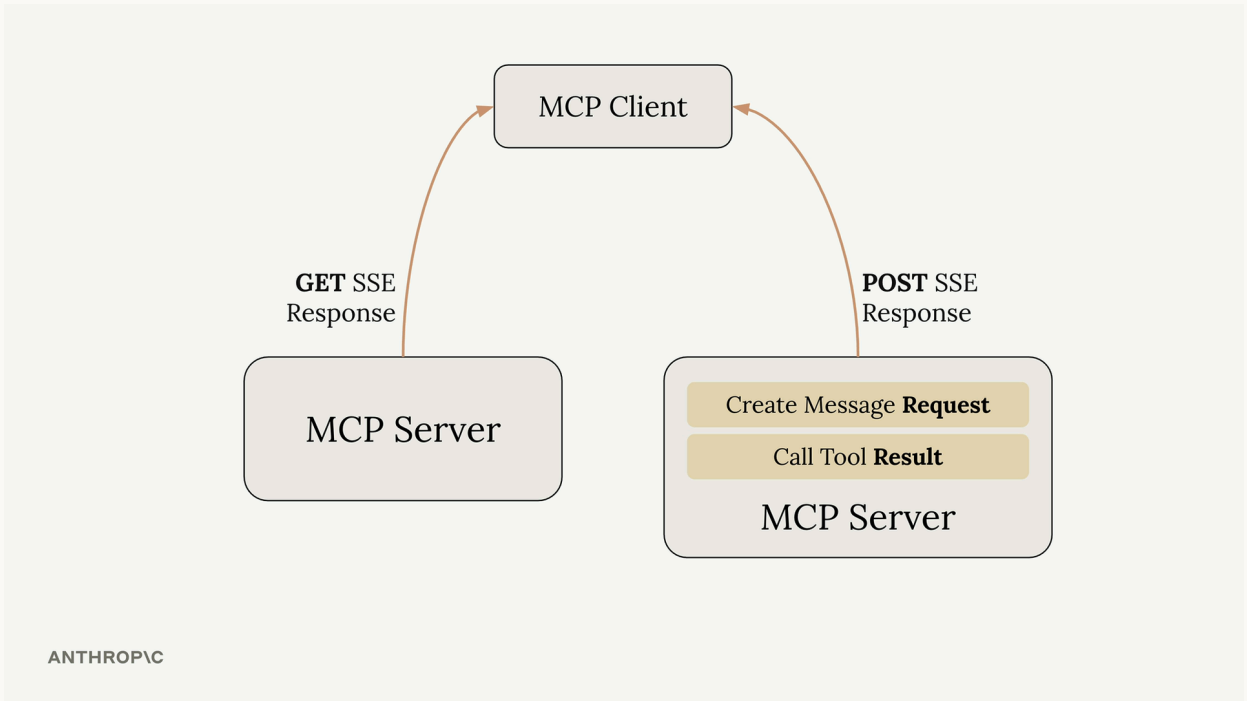


But here's where things get complicated. Remember that MCP clients need two separate connections:

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

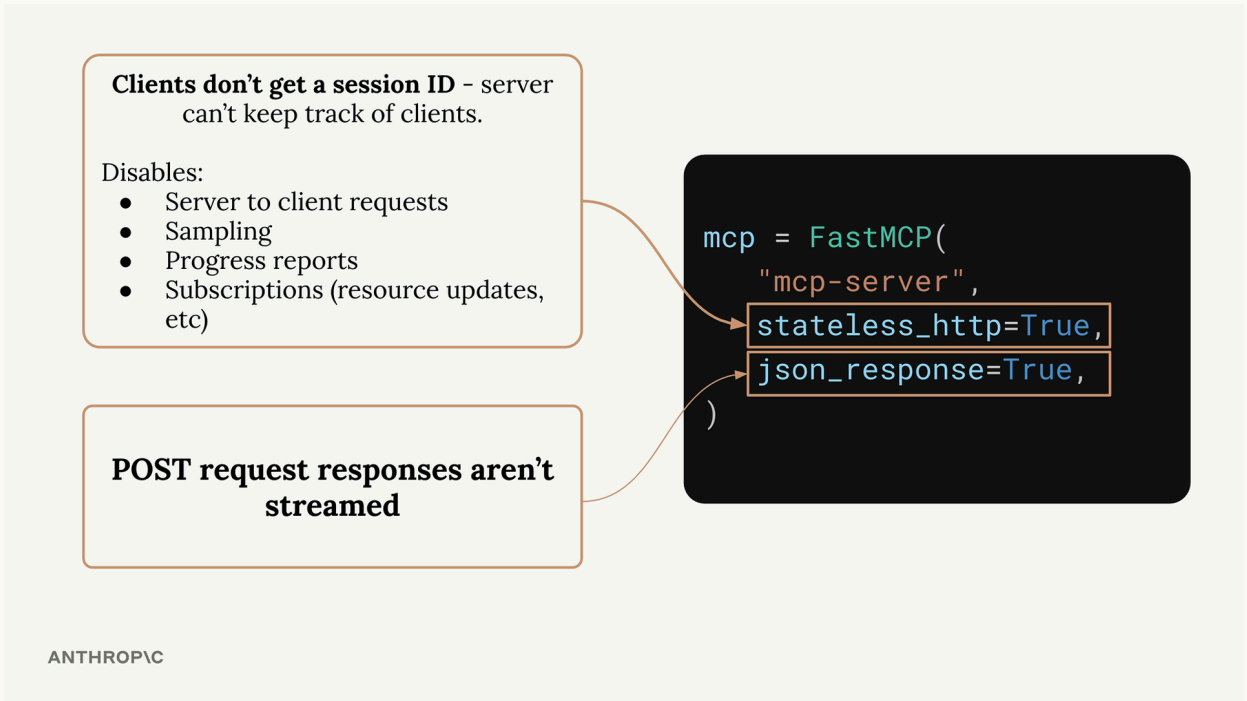


With a load balancer, these requests might get routed to different server instances. If your tool needs to use Claude (through sampling), the server handling the POST request would need to coordinate with the server handling the GET SSE connection. This creates a complex coordination problem between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



How Stateless HTTP Solves This

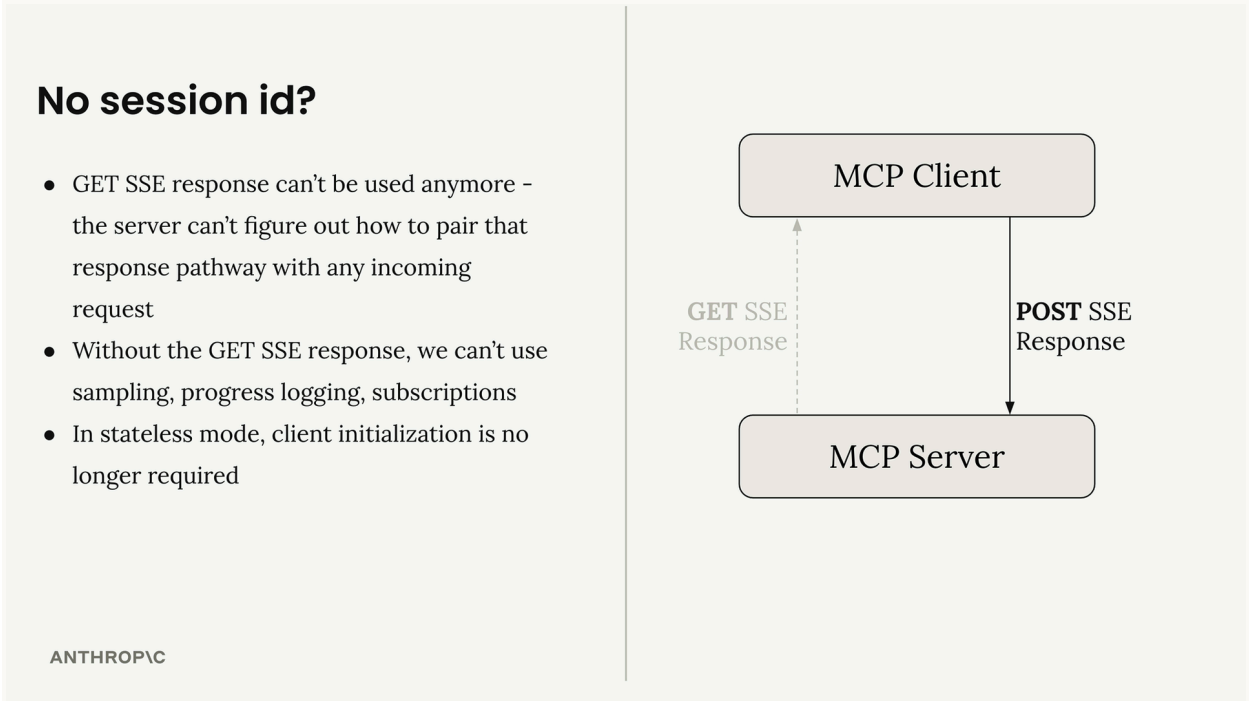
Setting **stateless_http=True** eliminates this coordination problem, but with significant trade-offs:



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.



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.