About this Course

In this course, you'll discover how to build powerful AI applications using the Model Context Protocol, a revolutionary approach that eliminates tedious integration code when connecting Claude to external services and data sources. This course takes you from understanding MCP's core architecture to building fully functional MCP servers and clients that can interact with real-world systems. You'll learn the three essential MCP primitives - tools, resources, and prompts - and understand when to use each one. Through hands-on implementation, you'll create your own MCP server with the Python SDK, defining tools that Claude can call autonomously, resources that provide data to your applications, and prompts that deliver professionally crafted workflows to end users. By building both client and server components, you'll gain complete insight into how MCP enables seamless communication between AI models and external services. You'll work with the MCP Inspector to test your implementations, understand transport protocols, and see exactly how messages flow through the entire system. Whether you're building GitHub integrations, document management systems, or custom AI workflows, this course gives you the practical skills to leverage MCP's standardized approach and dramatically reduce your development burden.

SELF UNDERSTANDING

WHAT IS MCP

MCP usually means **Model Context Protocol** (in the LLM / GenAI ecosystem).
I'll explain it **cleanly, from first principles**, without hype.

What is MCP (Model Context Protocol)?

MCP is a standard way for an AI model to connect to external tools, data, and services
— *without hard-coding integrations inside the model*.

Think of MCP as a **USB-C port for AI models**.

Just like USB-C lets devices talk to keyboards, monitors, storage, etc.,
MCP lets LLMs talk to tools, APIs, databases, files, and services in a uniform way.

What MCP actually does (core idea)

MCP defines **3 things**:

1️⃣ **How tools describe themselves**

```
{
 "name": "run_sql",
 "description": "Executes SQL on Teradata",
 "input_schema": {
  "query": "string"
 }
}
```

2️⃣ **How models discover tools dynamically**

- No prompt hard-coding
- Tools advertise:
    - what they do
    - required inputs
    - outputs

3️⃣ How models call tools safely

- Standard request / response
- Clear separation:
    - **Model reasoning**
    - **Tool execution**

Concrete example (simple)

Question:

“Show internet connects for California stores in Jan 2025”

Flow with MCP:

1. **Model reasons**
    - Needs SQL
    - Needs database
2. **Model discovers tools**
3. Available tools:
4. - run_sql
5. - list_tables
6. - get_schema
7. **Model calls tool**

```
{
  "tool": "run_sql",
  "input": {
    "query": "SELECT SUM(connects) ..."
  }
}
```

4. **Tool executes**
5. **Model explains result**

No fragile prompt logic.
No embedded SQL execution instructions.

---

8. MCP vs LangChain vs Agents (quick clarity)

Thing	**What it is**
MCP	Standard protocol
LangChain	Framework built *on top*
Agents	Behavioral pattern
Tools	Exposed via MCP

👉 MCP is **below** all of them.

When should MCP be used?

Use MCP when:

- Multiple tools
- Multiple teams
- Long-lived systems
- Security & governance matter
- You want **less prompt engineering**

Don’t bother if:

- One-off demo
- Single script
- No tools

One-line summary (remember this)

MCP standardizes how LLMs talk to tools — cleanly, safely, and at scale.

The clean truth (no extras)

MCP is the standard way an AI model communicates with external tools.

That's it.

One small but important refinement

A *slightly more precise* version:

MCP defines how an AI model discovers, calls, and receives results from external tools — safely and consistently.

What MCP handles vs what it doesn't

MCP handles

- Tool discovery (what tools exist)
- Tool calling (how to ask for a tool)
- Tool results (how results come back)
- Message format (no prompt hacks)

MCP does NOT

- Make decisions for the AI
- Execute logic itself
- Replace business logic
- Make AI smarter

One-line memory hook (use this)

AI thinks → MCP talks → tools act

Why this matters (even if it sounds obvious)

Without MCP:

- Every app invents its own way
- Prompts become fragile
- Scaling is painful

With MCP:

- Clean separation
- Predictable behavior
- Enterprise-safe

LET'S TAKE AN EXAMPLE WHERE I ASK A QUESTION, AND LETS UNDERSTAND MCP VIA THAT.

Question you ask

"What is the temperature in New Delhi now?"

WITHOUT MCP (old way)

The AI must already be told:

- Which weather API to use
- How to call it
- What parameters mean city, time, units
- How to parse the response

All of this is **hard-coded in prompts/code**.

If the API changes → things break.

WITH MCP (simple & clean)

Step 1: AI understands your question

AI thinks:

"This is a **weather question**. I need live data."

Step 2: AI looks at available tools

Through MCP, tools say what they can do:

Available tools:

- get_weather

And the tool describes itself:

“I can give current temperature if you give me a city name.”

Step 3: AI calls the tool (via MCP)

The AI doesn’t worry about APIs.

It just sends:

```
{
  "city": "New Delhi",
  "time": "now"
}
```

Step 4: Tool does the real work

The **weather tool**:

- Calls real weather service
- Gets live data
- Returns result

Example response:

```
{
  "temperature": 32,
  "unit": "°C"
}
```

Step 5: AI replies to you

AI converts it into human language:

“The current temperature in New Delhi is 32°C.”

Key idea (VERY IMPORTANT)

AI does thinking

Tool does doing

MCP makes sure they talk **cleanly**.

One-line summary

MCP lets the AI ask tools for real-world data instead of guessing or hard-coding logic.

Why this matters (even in this simple question)

Without MCP:

- AI might **hallucinate** temperature
- Or use outdated data

With MCP:

- AI **must ask the tool**
- Answer is **real and reliable**

HOW DOES IT ACTUALLY WORK

Big picture (one line)

MCP is just a structured conversation between the AI and tools.

Nothing magical.

Let’s walk the exact flow

Example again:

“What is the temperature in New Delhi now?”

STEP 0: Setup (done once by developers)

Some developer has already done this:

- Created a **Weather Tool**
- Exposed it using **MCP**

The tool says:

“I exist, this is what I do, this is how you talk to me”

That’s it.

STEP 1: You ask a question

User → AI

"What is the temperature in New Delhi now?"

STEP 2: AI only THINKS (no data yet)

The AI **does not know the temperature**.

It just reasons:

- This needs **live data**
- I cannot guess
- I need a **weather tool**

Important:
👉 **No API calls yet**

STEP 3: AI checks available tools (via MCP)

MCP gives the AI a list like:

Tools available:

- get_weather(city, time)

- read_file(path)

- run_sql(query)

Each tool includes:

- What it does
- What inputs it needs
- What output it gives

STEP 4: AI decides to use a tool

AI internally decides:

“I should use get_weather”

This decision is **pure reasoning**, not execution.

STEP 5: AI sends a TOOL REQUEST (not an API call)

AI sends a **structured message**:

```
{
  "tool": "get_weather",
  "input": {
    "city": "New Delhi",
    "time": "now"
  }
}
```

Key point:

❗ AI is **not calling the weather API**

❗ AI is just asking MCP to handle it

STEP 6: MCP hands it to the tool

Now MCP:

- Takes the request

- Sends it to the Weather Tool

MCP → Weather Tool

STEP 7: Tool does REAL work

The Weather Tool:

- Calls external weather API
- Fetches live temperature
- Formats output

Example:

{

"temperature": 32,

"unit": "C"

}

AI is **not involved** here.

STEP 8: Tool sends result back (via MCP)

Weather Tool → MCP → AI

Still structured, no guessing.

STEP 9: AI converts result to human language

Now AI gets:

{

"temperature": 32,

"unit": "C"

}

AI replies:

“The current temperature in New Delhi is 32°C.”

What actually runs where (important clarity)

Part	**Who does it**
Understanding question	AI
Deciding tool	AI
Calling API	❌ NOT AI
API execution	Tool
Security / auth	Tool
Formatting answer	AI

Why this design is powerful

Because:

- AI **cannot hallucinate**
- AI **cannot access things directly**
- Tools are:
    - controlled
    - logged
    - secured

VERY IMPORTANT REALIZATION

MCP does **not** make AI smarter
MCP makes AI **safer and reliable**

One sentence you should remember

AI decides *what* to do, MCP decides *how* it happens.

WHY IS IT CALLED SO

Start with the last word: Protocol

A **protocol** means:

- A **set of rules**
- About **how two things talk to each other**

Examples you already know:

- HTTP → how browsers talk to servers
- SQL → how you talk to databases

So **MCP = rules for communication**.

So far so good.

Now the key word: Context

What is "context" for a model?

For an AI model, **context = everything it is allowed to see right now**:

- The user's question
- System instructions
- Chat history
- Available tools
- Tool schemas
- Tool results
- Constraints ("you may / may not do X")

Context is NOT just text
Context includes capabilities

Example:

If the model knows:

"I have a weather tool"

That knowledge itself is **context**.

Now the first word: Model

This simply means:

- The protocol is **for AI models**
- Not for humans
- Not for browsers

Put it together (plain English)

Model Context Protocol = rules for telling an AI model what it can see and what it can use

That's the real meaning.

Why the name actually makes sense (important insight)

Before MCP

Models were given context like this:

"You can call the weather API like this...

Use this JSON format...

Don't forget auth...

Parse response like this..."

Tools were **embedded inside text prompts**
Context was **messy, implicit, fragile**

With MCP

Context is **structured and explicit**:

- Tools are listed formally
- Inputs & outputs are defined
- Results are injected cleanly

So MCP **standardizes how context is constructed**.

Why MCP is NOT called "Tool Protocol"

Because tools are **not the only context**.

MCP also standardizes:

- Tool discovery
- Tool invocation
- Tool results
- Boundaries of model authority

It defines:

"This is what the model is allowed to know and do."

Example (very concrete)

Question:

"What is the temperature in Delhi?"

Context *without* MCP

The model might:

- Guess
- Hallucinate
- Use outdated knowledge

Context *with* MCP

The model is told:

- "You have access to a live weather tool"
- "Here is the result returned by the tool"

That **changes the model's context**, not its intelligence.

Why the name feels confusing

Because:

- “Context” sounds abstract
- People think only of *token window*
- They expect MCP to be about memory

But MCP is about **operational context**, not memory.

The most accurate one-line definition

MCP standardizes how an AI model’s operational context (tools, capabilities, results) is provided and updated.

Even simpler (remember this)

MCP decides what the model is allowed to see and use, and how that information flows.

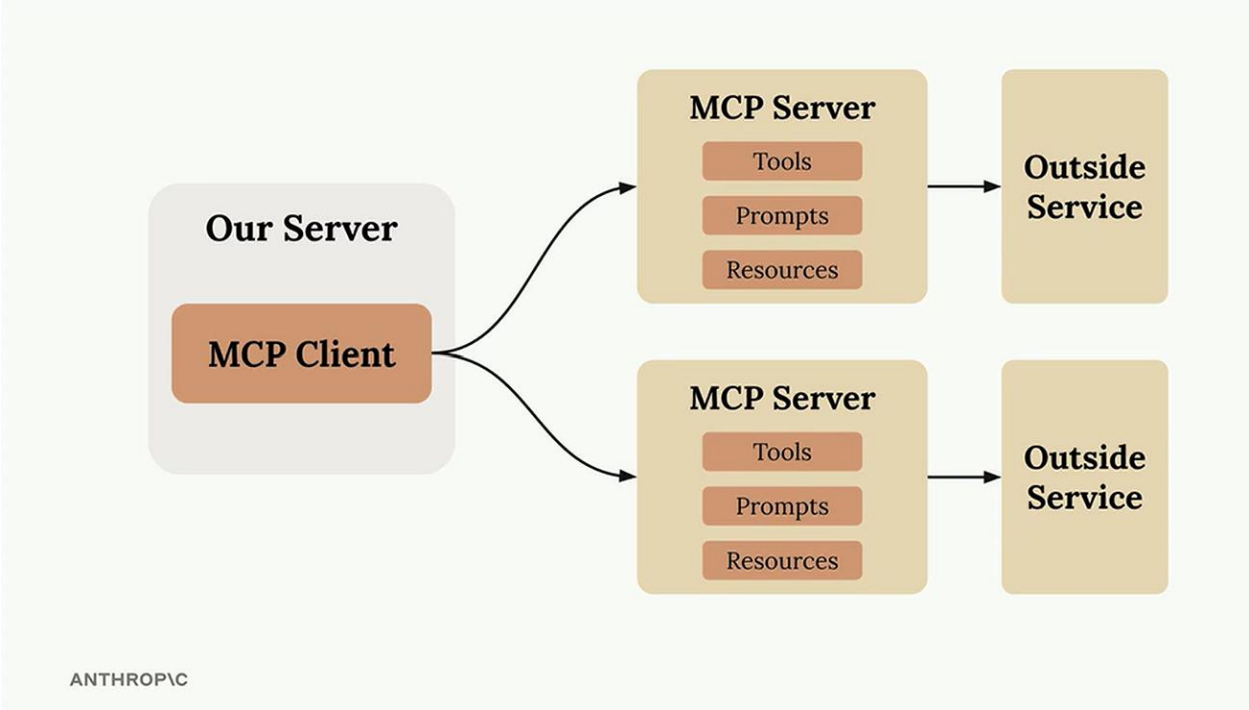
MODULE-1

INTRO TO MCP

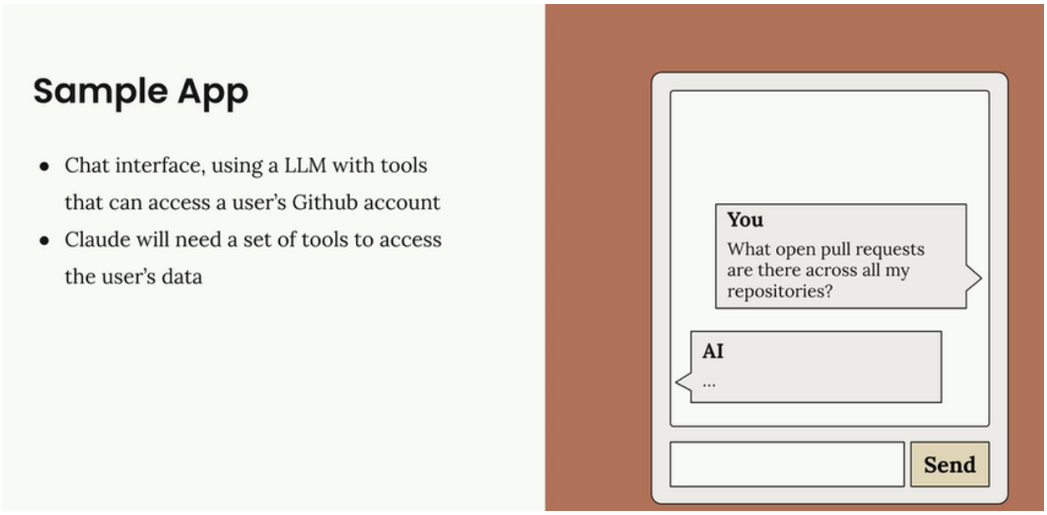
Model Context Protocol (MCP), which simplifies communication between developers and cloud services by reducing the need for extensive coding.

Understanding MCP

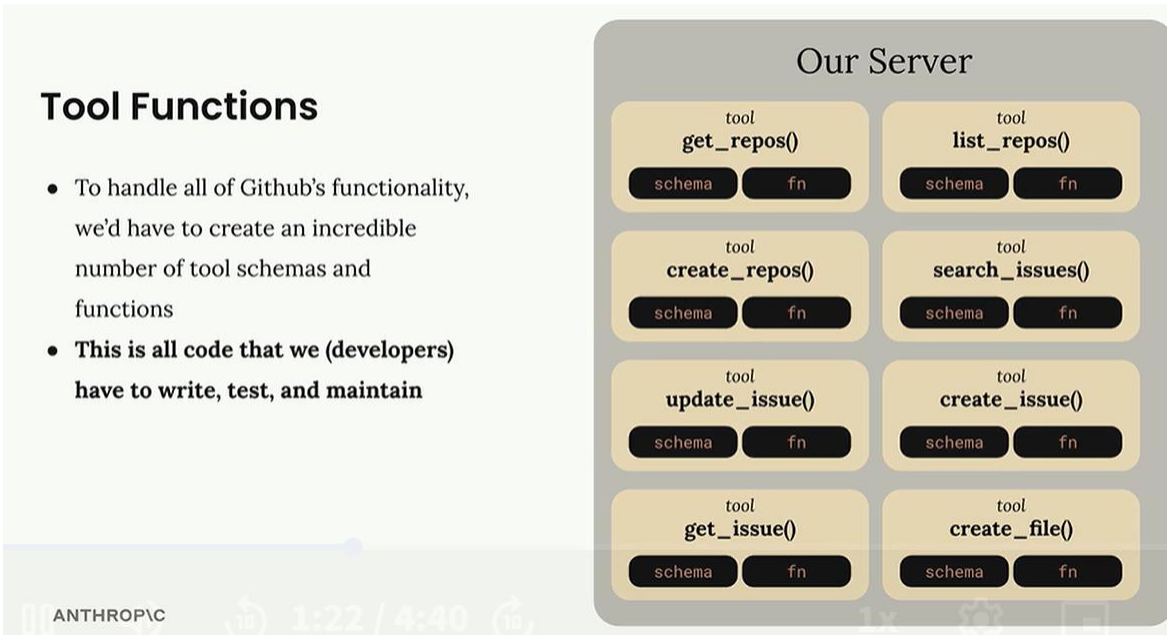
- MCP serves as a communication layer that connects clients and servers, allowing developers to access tools and resources without writing complex code.



- The module uses a chat interface app example to illustrate how MCP can interact with external services like GitHub.
    - Example: User asks about open pull requests of a github repo
    - Claude uses tool to reach github, reach user account and looks for what's required. Underlying this is that Claude will use set of tools

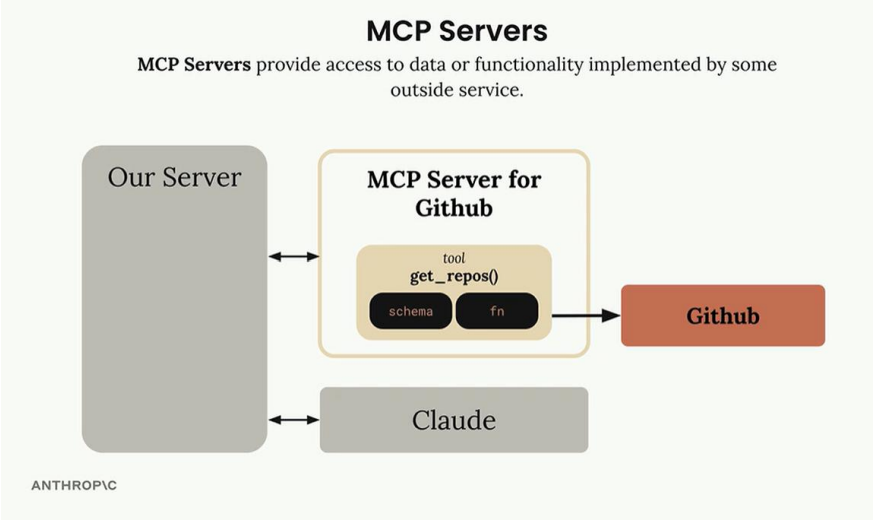


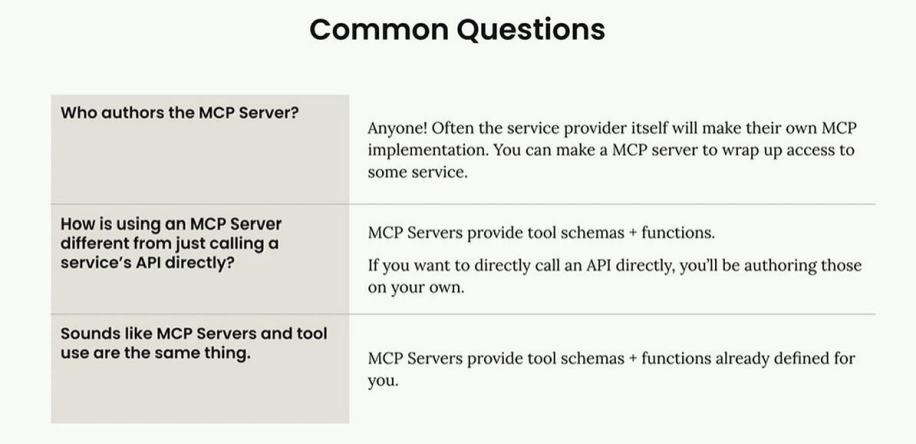
- All these tools will have to be defined by us (and written by us) in order to utilize – challenge !



MCP Server Functionality

- MCP servers handle the definition and execution of tools, relieving developers from the burden of creating and maintaining numerous integrations.
    - So, in above example, I will have github MCP servers (kind of like 3rd party servers) which will provide access to data and services by github





- These servers act as interfaces to external services, providing access to various functionalities without requiring developers to author all tool schemas and functions.

Common Questions about MCP

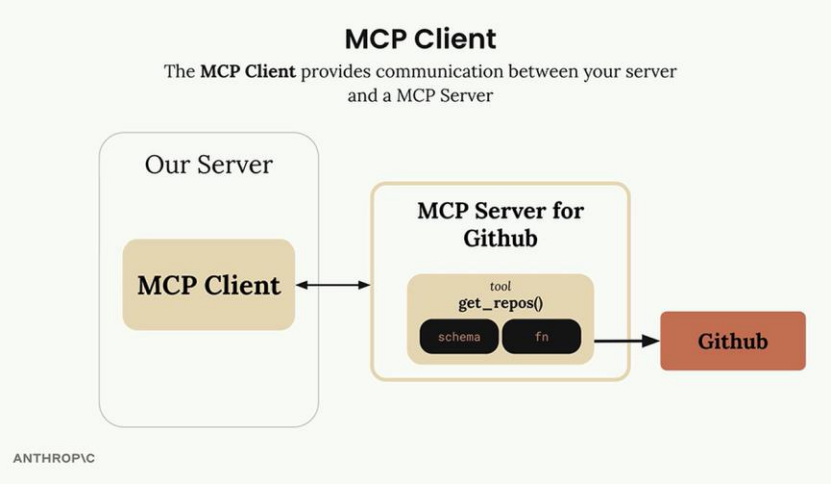
- Anyone can create an MCP server, but service providers often release official implementations with a variety of tools.
- Using an MCP server saves time compared to directly calling an API, as it eliminates the need for developers to write tool functions themselves.
- MCP servers and tool use are complementary; MCP focuses on who manages the tool functions, while tool use refers to the actual application of those tools.

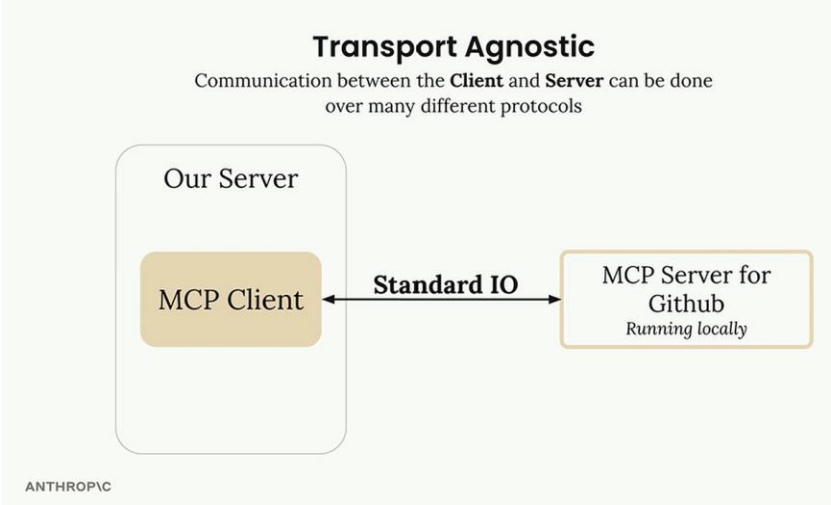
MCP CLIENTS

role of the client in the Model Context Protocol (MCP) and how it facilitates communication between a server and an MCP server.

Client Communication

- The client serves as the access point for tools provided by the MCP server, enabling communication over various protocols like HTTP or WebSockets.

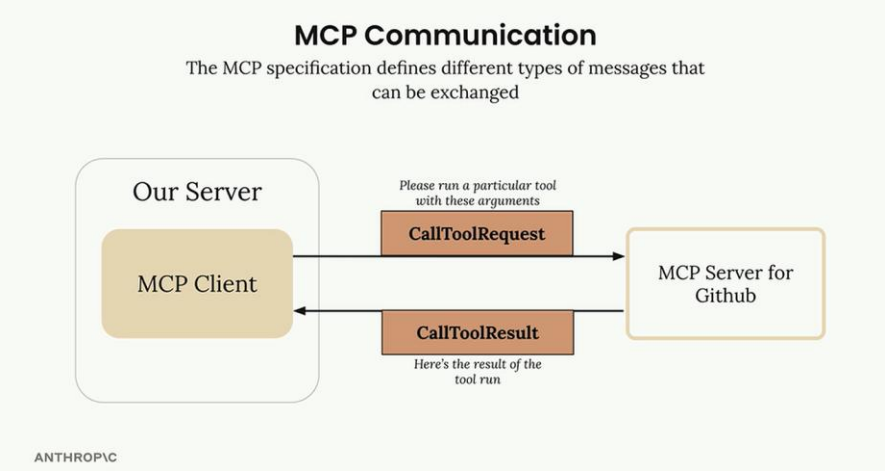




- When the client and server are on the same machine, they can communicate via standard input/output.

Message Exchange

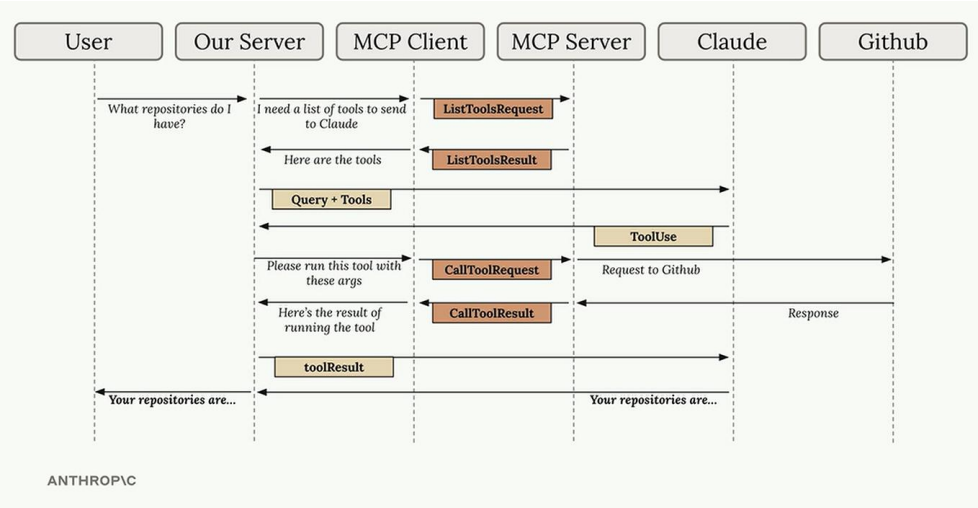
- Communication involves exchanging messages defined in the MCP specification, including ListToolsRequest (client to server) and ListToolsResult (server to client).



- Other important message types include CallToolRequest (to execute a tool) and CallToolResult (result of the tool execution).

Example Workflow

- The process begins with a user query, which the server uses to request a list of tools from the MCP client.
- The MCP client retrieves the tools from the MCP server, allowing the server to make a request to an external service (e.g., Cloud) with the user's query and tool list.



This summary outlines the essential functions of the client in the MCP framework and illustrates the message flow in a practical example.

SELF TO UNDERSTAND

First: ignore all fancy words

Forget:

- transport agnostic
- Cloud
- message types
- ListToolsRequest, CallToolRequest, etc.

Those are **implementation details**, not concepts.

What is REALLY going on (one sentence)

Your app asks an AI a question → AI decides it needs a tool → MCP is the middleman that runs the tool safely and gives the result back.

That's it.

Everything else is ceremony.

Let's rename the characters (this helps A LOT)

The video uses confusing names. Let's simplify:

Video term	**Simple name**
User	You
Server (your server)	**Your App**
Cloud	**AI (LLM)**
MCP Client	**Tool Messenger**
MCP Server	**Tool Runner**
GitHub	**Real external service**

Now it will click.

The REAL flow (simple story)

Step 1: You ask a question

“What repositories do I have on GitHub?”

You ask **Your App**.

Step 2: Your App asks the AI

Your App sends this to the AI:

“User asked this question. Also, here are the tools you can use.”

But wait…

👉 **Your App does NOT know the tools yet**

So first it asks MCP.

Step 3: Your App asks MCP:

“Hey MCP, what tools exist?”

This is the **ListTools** thing.

- MCP Client → MCP Server
- MCP Server replies:
    - “I have a tool called get_github_repos”

Your App now knows:
✅ what tools exist
✅ what they do

Step 4: Your App sends tools + question to AI

Now Your App sends to AI:

“User asked: *what repositories do I have?*
Tools available: get_github_repos”

Step 5: AI decides to use a tool

AI thinks:

“I cannot answer this without GitHub data. I should use the tool.”

So AI replies:

“Please call get_github_repos with these arguments.”

⚠️ Important:

- AI **does NOT run the tool**
- AI just **asks** for it

Step 6: Your App runs the tool (via MCP)

Your App says to MCP:

“Run this tool with these inputs.”

This is **CallToolRequest**.

- MCP Client → MCP Server
- MCP Server actually:
    - calls GitHub API
    - fetches repos

Step 7: MCP sends result back

GitHub → MCP Server → MCP Client → Your App

Now Your App has:

["repo1", "repo2", "repo3"]

Step 8: Your App sends result back to AI

Your App tells AI:

“Here is the result of the tool you asked for.”

Step 9: AI gives final answer

AI replies:

“Your repositories are repo1, repo2, repo3.”

Your App sends this to **you**.

Why so many steps? (this is the KEY insight)

Because **AI is NOT allowed to**:

- call GitHub
- access secrets
- run code
- touch databases

AI only decides WHAT to do
MCP decides HOW it is done

This separation is **intentional and very important**.

What the speaker was REALLY trying to teach

He wanted to show:

1. MCP Client = talks to MCP Server
2. MCP Server = owns tools
3. AI never executes tools directly
4. Everything is message-based so it's:
    - safe
    - auditable
    - replaceable

But he explained it in a **developer-spec way**, not a learning way.

One sentence you should remember (this locks it in)

The MCP client is just a messenger, the MCP server is the worker, and the AI is the brain.

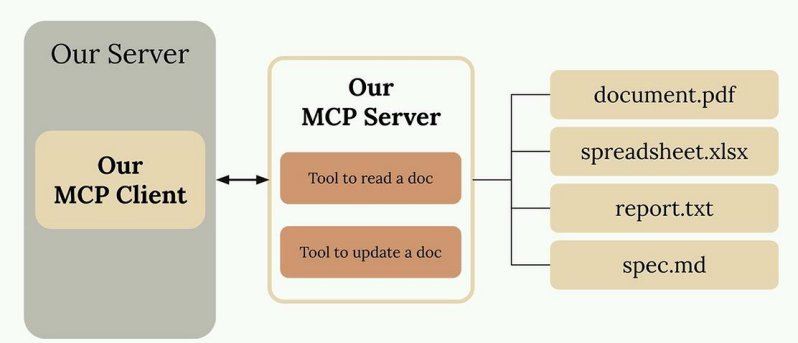
MODULE-2

PROJECT SETUP

Implementing a CLI-based chatbot to understand the interaction between clients and servers in the context of the Model Context Protocol (MCP).

Project Overview

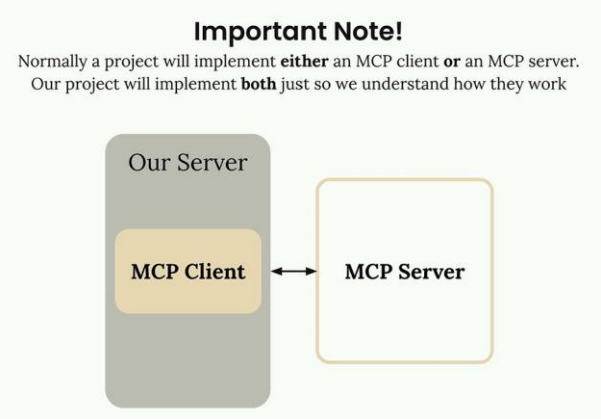
- The project involves creating a CLI-based chatbot that interacts with a collection of fake documents stored in memory.



- A small MCP client will connect to a custom MCP server, which will have tools for reading and updating document contents.

Client and Server Implementation

- Typically, projects involve either creating a client or a server; this project combines both for educational purposes.



- Understanding how clients and servers work together is emphasized through this dual implementation.

Setup Instructions

- Learners are instructed to download a starter code file and follow setup directions in a readme.md file.

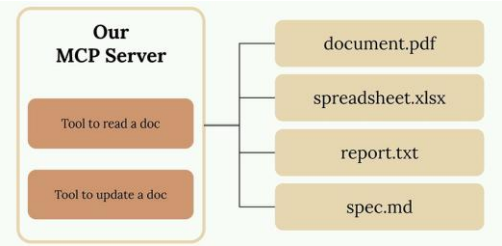
- The project can be run using specific commands in the terminal, depending on whether the user is utilizing UV or not.

DEFINING TOOLS WITH MCP

building an MCP server for a CLI chatbot, specifically implementing two tools for document management.

MCP Server Setup

- The MCP server will include tools to read and update document contents.



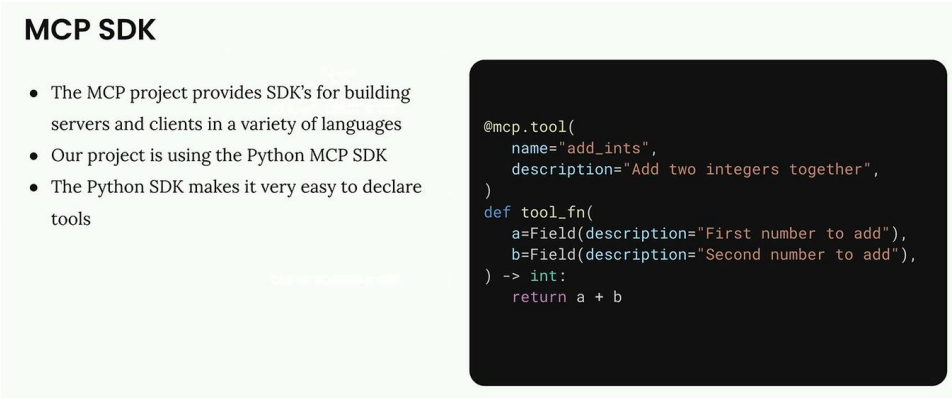
- The implementation is done in the mcpserver.py file, with a collection of documents stored in memory.

Tool Implementation

- The first tool, read doc contents, retrieves the contents of a specified document from a dictionary.
- The second tool, edit document, allows for replacing a string in a document's content with a new string.

Using MCP Python SDK

- The MCP Python SDK simplifies tool definition and server creation with minimal code.



- It automatically generates a JSON schema for the defined tools, streamlining the development process.

SERVER INSPECTOR

testing functionality within an MCP server using a Python SDK and an in-browser debugger.

MCP Server Testing

- The Python SDK provides access to an in-browser debugger for testing the MCP server.
- To start, activate the Python environment and run the command to initiate the server.

Using the MCP Inspector

- The MCP inspector allows for live development and debugging without needing a real application.
- Users can connect to the server and access tools for testing various functionalities.

Testing Tools

- The read doc contents tool retrieves document contents using a document ID.
- The edit document tool allows for editing specific strings in a document, with success messages confirming the actions taken.

SELF TO UNDERSTAND – SUMMARY

MCP servers can be created very easily using the official Python SDK. The SDK abstracts away most of the low-level complexity involved in defining MCP-compliant tools and servers.

To define a tool, we can directly use the @mcp.tool() decorator. One of the biggest advantages of this approach is that we **do not need to manually write or maintain complex JSON schemas**. The decorator, together with Python type hints and metadata, automatically generates the correct tool schema that is sent to the AI model.

Inside the @mcp.tool() decorator, we specify:

- **The tool name** – this is the identifier that the AI model sees and uses when deciding which tool to call.
- **The tool description** – this becomes part of the model's context and helps it understand *when* and *why* the tool should be used.

After defining the decorator, we simply implement the function itself. The function signature plays a critical role in schema generation:

- **Function parameters and their type hints** (e.g., str, int) define the expected input types.
- Using Field(description=...) on parameters adds semantic meaning, which becomes part of the tool's argument schema and helps the model supply correct inputs.
- The function body contains the actual business logic or algorithm that runs when the tool is invoked.

All of this information—the tool name, description, parameter types, and parameter descriptions—is automatically combined by the MCP SDK to produce a valid JSON schema behind the scenes, which is then exposed to the AI model.

Once tools are defined, the SDK also provides a convenient way to test them locally using the **MCP Inspector**. By running the inspector (typically via the mcp dev command), we can start a local MCP server that opens in a browser-based UI. This UI:

- Lists all available tools exposed by the server
- Allows us to select a tool
- Lets us manually enter arguments
- Executes the tool and displays the output or errors

This local inspection workflow makes it very easy to validate that tools are correctly defined, schemas are generated as expected, and the underlying logic works properly—all before integrating the MCP server with a CLI chatbot or an AI model.

Overall, the MCP Python SDK enables rapid, clean, and reliable development of MCP servers by focusing on Python-native constructs instead of verbose schema definitions.

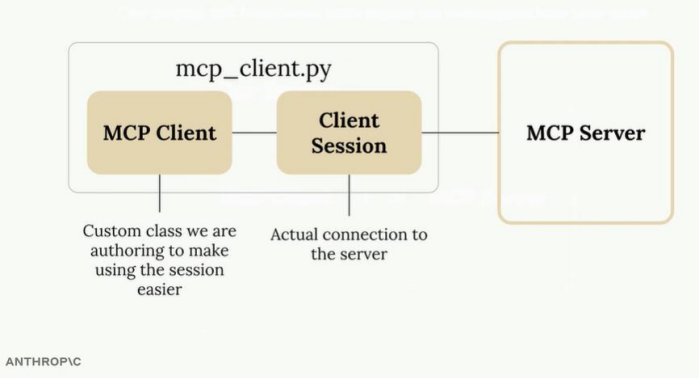
MODULE-3

IMPLEMENTING A CLIENT

implementation of the MCP client, which is essential for connecting to the MCP server.

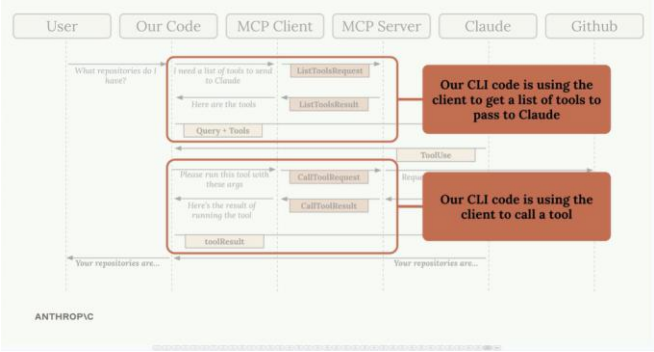
MCP Client Overview

- The MCP client is defined in the MCP client.py file and consists of a single class that manages a client session.
- This client session is part of the MCP Python SDK and facilitates the connection to the MCP server, requiring resource cleanup when closing.



Functionality of the MCP Client

- The client exposes functionalities of the server, allowing the codebase to request tools and execute them.



- Two key functions, list_tools and call_tool, are implemented to retrieve a list of tools and execute a specific tool based on user requests.

Testing the MCP Client

- A testing harness is included to run the MCP client directly, enabling the retrieval of tool definitions from the server.

- The client can also interact with Claude to perform tasks like reading document contents, demonstrating the practical application of the client in real-world scenarios.

SELF TO UNDERSTAND

Here is the **same summary**, now with the **New Delhi weather example fully integrated**, so it's concrete and easy to recall.

What is an MCP Resource?

An **MCP Resource** is a **read-only data endpoint** that the **client (system)** fetches **deterministically** and injects into the model's context.

- Used to **read existing data**
- Triggered by **system rules** (syntax, UI action, configuration)
- The **model never decides** to use a resource

Example (Document)

User asks:

"What is today's weather in New Delhi according to @report.pdf?"

What happens:

1. Client detects @report.pdf
2. Client fetches report.pdf via a **resource**
3. Client injects the document text into the prompt
4. Model reads it and answers

✓ No reasoning about fetching
✓ No tool call needed

Resource = "Always fetch this data when the rule matches."

What is an MCP Tool?

An **MCP Tool** is an **action** the **model may choose to invoke** when it needs information or computation it doesn't already have.

- Used to **do something**
- Triggered by **model reasoning**
- Executed only if the model requests it

Example (Weather API)

User asks:

"What is the weather in New Delhi today?"

What happens:

1. Model sees no weather data in context
2. Model decides it needs live data
3. Model requests the get_weather tool
4. Client runs the weather API
5. Model formats the answer

✓ Model-driven
✓ Not deterministic

Tool = "Model, call this if you need it."

Combined Weather + Document Scenario (important)

User asks:

"What is today's weather in New Delhi? Use @report.pdf if helpful."

Flow:

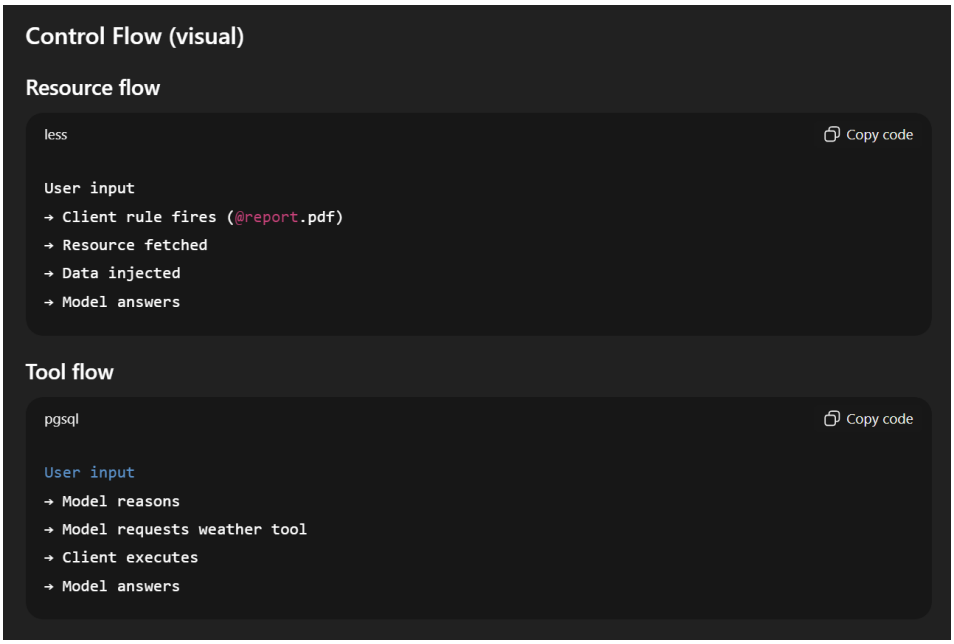
1. Client fetches report.pdf (resource)
2. Model reads document
3. Model checks:
    - If document clearly has today's weather → answer
    - If missing / outdated → call weather tool

Here:

- **Resource happens first (always)**
- **Tool happens only if needed**

The Key Difference (one line)

Resources are system-triggered and deterministic.
Tools are model-triggered and probabilistic.



Final mental hook (lock this in)

- **Resource = read**
- **Tool = act**
- **Client controls resources**
- **Model controls tools**

With this model, the weather + document case becomes obvious every time.

SELF TO UNDERSTAND - BUT WHERE DO WE DEFINE THE TRIGGER RULES?

Where resource *triggers* are defined

Triggers are defined in the CLIENT (your app), not in the MCP server.

MCP Server

- Defines **what data is available**
- Exposes it via **resources** (URIs)

- Has **no idea when or why** they are used

Client (CLI / UI / App)

- Parses user input
- Applies deterministic rules
- Decides **when to fetch which resource**

Why you don't see triggers in server code

Your MCP server code only contains things like:

```python
@mcp.resource("docs://documents")
def list_docs():
    ...
```

This means:

> "If someone asks for `docs://documents`, here's how to respond."

It does **NOT** mean:

> "Fetch this when user types @"

That logic **cannot** live on the server.

Where the `@` rule actually lives (concrete)

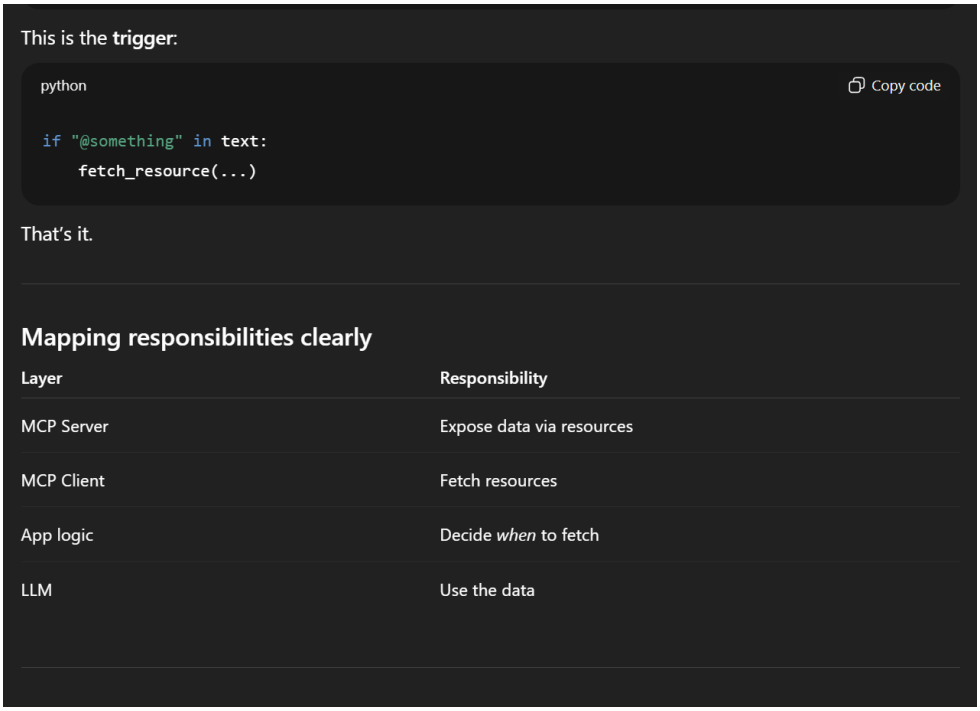
In the **client**, you'd have code like:

```python
def preprocess_user_message(text):
    mentions = extract_mentions(text)  # e.g. @report.pdf

    for doc in mentions:
        content = mcp_client.read_resource(
            f"docs://documents/{doc}"
        )
        text = inject_into_prompt(text, content)

    return text
```

This is the **trigger**:

```python
if "@something" in text:
    fetch_resource(...)
```

That's it.



Analogy (very simple)

- **Server**: "I have files. Ask me by name."
- **Client**: "If user types @file, I'll go get it."
- **Model**: "I just answer using what I'm given."

Final mental model (lock it in)

- **Resources** = *What* data exists (server)
- **Triggers** = *When* to fetch data (client)
- **Tools** = *How* to act (model)

One sentence answer to your question

Trigger rules are not part of MCP — they live entirely in your client application logic.

SELF TO UNDERSTAND - SUMMARY

Summary

The video explains **MCP "resources"** and how they're used to expose data from an MCP server to a client, using a document-mention feature as an example.

Goal of the feature

- Let users reference documents by typing @document_name.
- Show an autocomplete list of documents when @ is typed.
- When a message is submitted with a mention, automatically fetch the document's contents and inject it into the prompt sent to the LLM (instead of relying on the LLM to fetch it via tools).

Key idea: Resources

- **Resources** are read-only data endpoints exposed by an MCP server.
- Each distinct read operation should usually map to one resource.
- In this example, two resources are needed:
    1. A resource to **list document names** (for autocomplete).
    2. A resource to **fetch the contents of a single document**.

How resources work

- The client sends a **read resource request** to the MCP server with a **URI**.
- The server matches the URI, runs the associated function, and returns the result.
- The response includes a text payload and a **MIME type** that tells the client how to interpret it (e.g., JSON vs plain text).

Types of resources

1. **Direct (static) resources**
    - Fixed URI (e.g., docs://documents)
    - Used for things like listing all document IDs.
2. **Templated resources**
    - URI includes parameters (e.g., docs://documents/{doc_id})
    - Parameters are automatically parsed and passed to the function.
    - Used when the request varies (e.g., fetching a specific document).

Implementation highlights

- Use @mcp.resource decorators to define resources.
- Specify:
    - URI
    - MIME type (application/json for lists, text/plain for raw content)

- The MCP SDK automatically serializes return values.
- Validate inputs (e.g., raise an error if a document ID doesn’t exist).

Testing

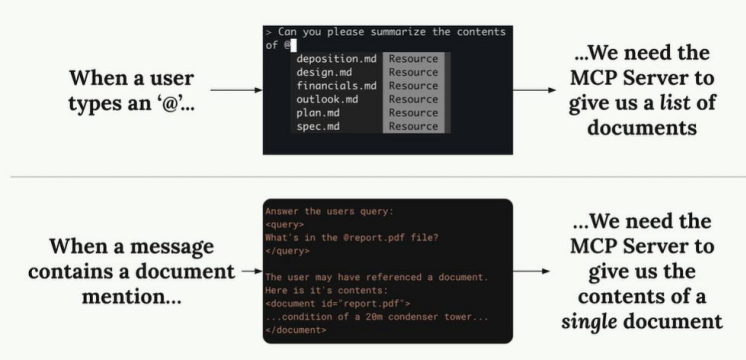
- Use the MCP Inspector to:
    - List available resources.
    - Call static resources (document list).
    - Call templated resources (fetch a document by ID).
- The MIME type guides the client on whether to deserialize JSON or treat the response as plain text.

DEFINING RESOURCES

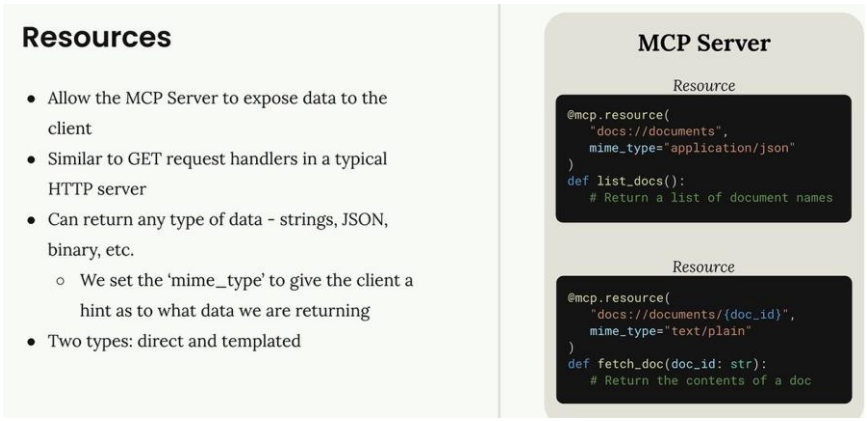
Implementing resources in MCP servers to enhance user interaction with documents.

Resource Management in MCP Servers

- Users can mention documents using an "at" symbol, triggering an autocomplete feature that lists available documents.

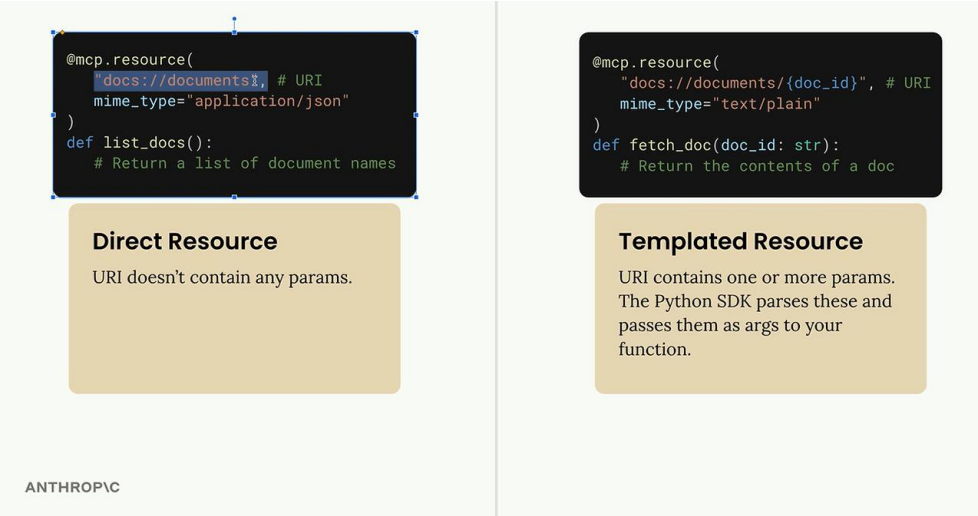


- When a user submits a message with a document mention, the server fetches the document's contents to include in the prompt sent to Claude.



Types of Resources

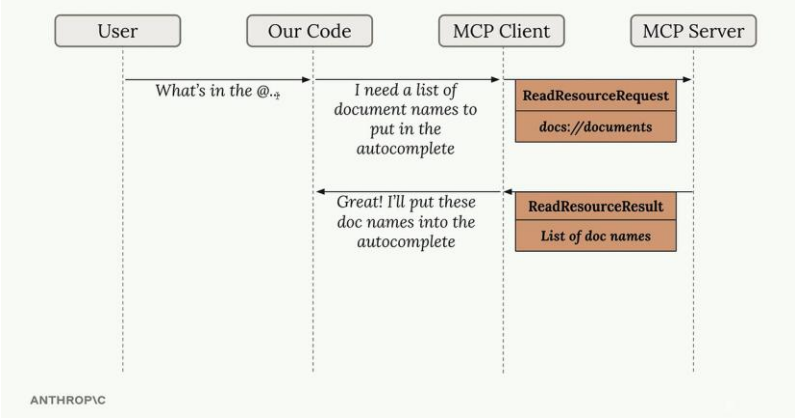
- Two types of resources are defined: one for listing document names (static resource) and another for fetching the contents of a specific document (templated resource).



- Static resources have a fixed URI, while templated resources allow for dynamic parameters, such as document IDs.

Implementation Process

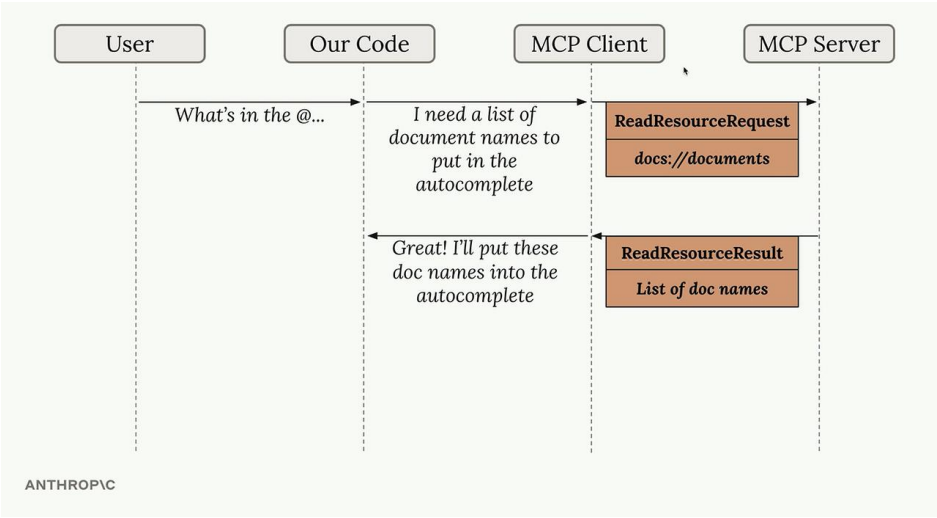
- The MCP client sends a read resource request to the MCP server, including the URI of the desired resource.



- The server processes the request and returns the relevant data, which can then be displayed in the user interface or used as needed.

ACCESSING RESOURCES

implementing a function in the MCP client to request resources from the MCP server.



MCP Client Functionality

- The function is designed to read a specific resource by making a request to the MCP server.
- It requires the URI of the resource to fetch the data.

Handling MIME Types

- The function checks the MIME type of the returned data to determine how to parse it.
- If the MIME type is JSON, the text content is parsed as JSON; otherwise, it is returned as plain text.

Testing the Implementation

- The implementation is tested using a command-line interface (CLI) to ensure that resources can be accessed and displayed correctly.
- Successful execution allows users to view and select resources, confirming that the function works as intended.

DEFINING PROMPTS

implementing prompts within the MCP server to enhance user interaction and document management.

Understanding Prompts

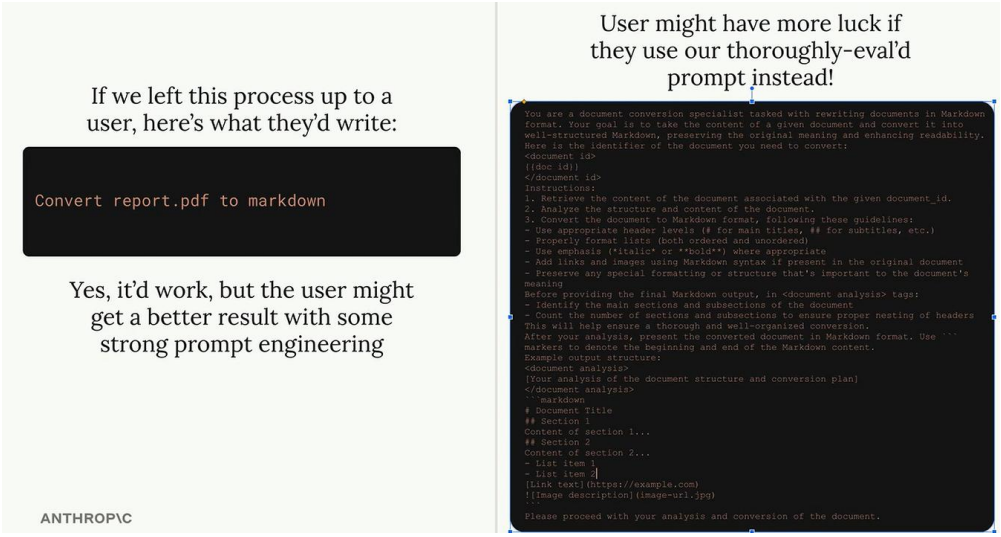
- The main feature being added is support for slash commands, starting with a "format" command.
- Users can type a slash to see available commands, with "format" prompting for a document ID.

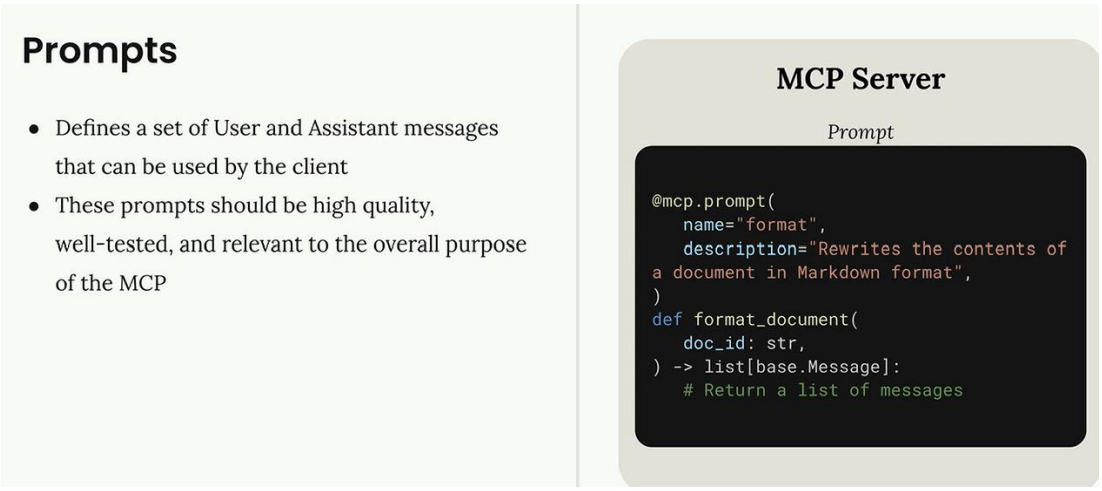
Reformatting Documents

- The goal is to reformat documents into markdown syntax using Claude, improving the output quality through tailored prompts.
- While users can manually execute the reformatting, a well-crafted prompt is expected to yield better results

Defining and Using Prompts

- Prompts are defined in the MCP server using a specific syntax, allowing for high-quality, tested prompts to be exposed for client applications.
- The process involves creating a prompt decorator, specifying a name and description, and returning a list of messages for Claude to process.





SELF TO UNDERSTAND - SUMMARY

An **MCP client** is the part of your application that **actively connects to an MCP server and drives all interaction with it**. The server only *exposes capabilities* (tools and resources); the client is responsible for **discovering, invoking, and orchestrating** those capabilities.

What an MCP client does

At a high level, an MCP client:

1. **Establishes a connection** to an MCP server (commonly over STDIO, HTTP, or WebSockets).
2. **Initializes a session** using the MCP protocol (capability negotiation and handshake).
3. **Discovers server capabilities**, such as:
    - What **tools** the server exposes
    - What **resources** (read-only data) are available
4. **Invokes capabilities** by sending structured protocol requests:
    - Reading a resource
    - Calling a tool with parameters
5. **Processes responses** and integrates them into the application's logic (or into an LLM workflow).

Crucially, the **client decides *when* to do all of this**. The server never runs tools or returns data on its own.

How this helps your application

The MCP client acts as a **bridge between your application logic and external capabilities**:

- It lets your app **use tools without hard-coding integrations**.
- It keeps **control and orchestration** on the application side.
- It enables safe patterns where:
    - **Resources** are fetched deterministically by the client.
    - **Tools** are invoked explicitly (either by client logic or on behalf of an LLM).

This separation makes systems more auditable, predictable, and modular.

Example 1: Discovering available tools

The client can ask the server what tools it exposes:

python Copy code

```python
tools = await session.list_tools()
```

Conceptually, this sends a protocol request like:

> "Server, tell me all the tools you provide."

The server responds with structured metadata, e.g.:

json Copy code

```json
{
  "tools": [
    { "name": "add" },
    { "name": "get_weather" }
  ]
}
```

Your application can then decide which tools are relevant.

Example 2: Calling a tool

If the application (or an LLM acting through the client) wants to perform an action, the client explicitly calls a tool:

```python
result = await session.call_tool(
    "get_weather",
    { "city": "New Delhi" }
)
```

This tells the server:

> "Execute the `get_weather` tool with these parameters."

The server runs the tool and returns the result, which the client then uses in the application.

PROMPTS IN CLIENT

Implementing functionality within an MCP client to manage prompts defined in the MCP server.

Implementing List Prompts

- The client will list all defined prompts by calling self.session.list_prompts() and returning the results.
- This allows users to see available prompts for interaction.

Retrieving Individual Prompts

- When fetching a specific prompt, the client will receive arguments, such as a document ID, which will be used in the prompt function.
- The get_prompt function retrieves the prompt by name and passes the necessary arguments to interpolate values into the prompt.

Testing the Implementation

- The client can be tested in the command line interface (CLI) by invoking the format command and selecting a document.
- The system fetches the document's contents and reformats it into markdown syntax, demonstrating the functionality of the prompts.