Introduction to Agentic AI

-- AI Agent Tool Design

Instructor: Guangjing Wang

guangjingwang@usf.edu

Last Lecture

- LLM Agent: Next Token Generation
- LLM Agent Framework
- Principles of Tool Calling and Code Execution

Why Tool Calling is Important

- LLMs have vast (but static) world knowledge that only updates when we retrain
- To build fully autonomous systems we need robust ways to feed dynamic data in
    - What’s the weather today
    - Who’s president
    - What’s the price of Bitcoin
    - Who’s the narrator in Nike’s latest ad campaign
- RAG and tool-calling are the best answer we have today

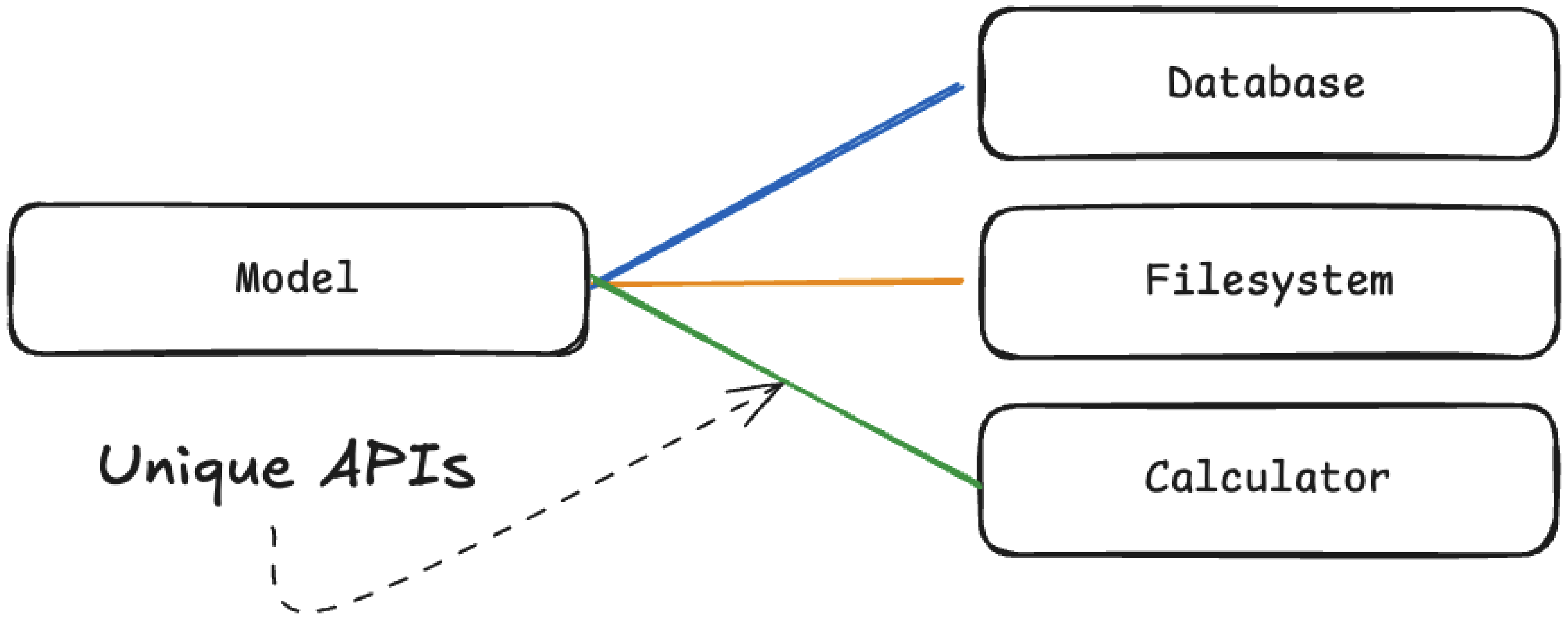
Review: How LLM Tool Calling Works

- Read in terminal and keep appending to conversation
  - Tell LLM what tools are available;
  - LLM asks for tool use at appropriate time
  - You/Framework execute tool offline and return response

https://www.youtube.com/watch?v=QiRdYCNXAxk (10 minutes)

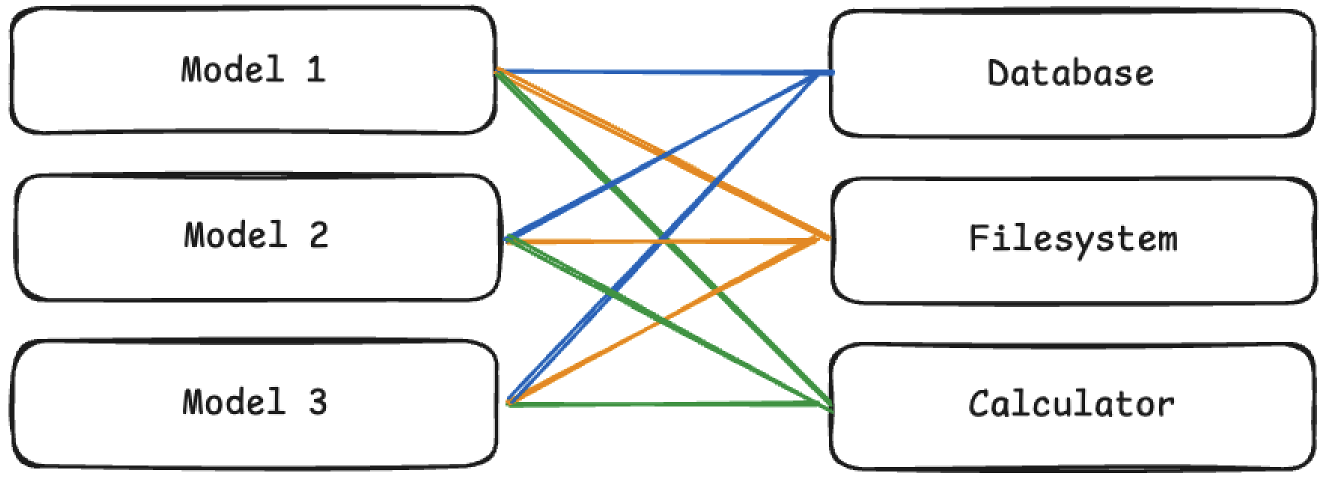
The MxN Integration Problem (1)

Each AI application would need to integrate with each tool/data source individually.



The MxN Integration Problem (2)

The **M×N Integration Problem** refers to the challenge of connecting M different AI applications to N different external tools or data sources without a standardized approach.

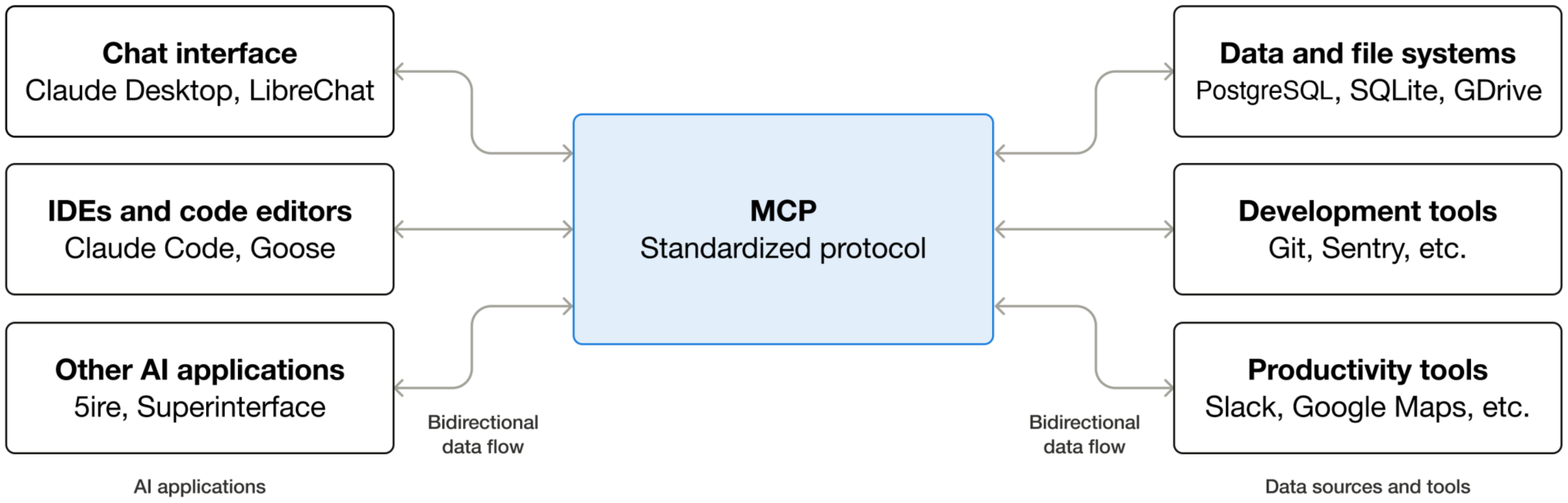


A very complex and expensive process which introduces a lot of friction for developers, and high maintenance costs.

Model Context Protocol (MCP)

- USB-C provides a standardized way to connect electronic devices.
- Like USB-C, MCP provides a standardized way to connect AI applications to external systems.
- MCP offers a consistent **protocol/Standard Format** for linking AI models to external capabilities (e.g., tools).
- MCP transforms MxN into an M+N problem by providing a standard interface.

MCP: USB-C for AI applications



- Each AI application implements the client side of MCP once.
- Each tool/data source implements the server side of MCP once.
- This dramatically reduces integration complexity and maintenance burden.

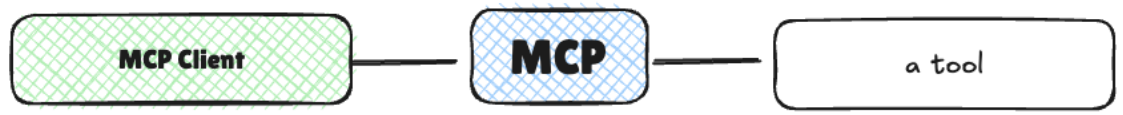
Why Does MCP Matter?

- **Developers**: MCP reduces development time and complexity when building, or integrating with, an AI application or agent.
- **AI applications or agents**: MCP provides access to an ecosystem of data sources, tools and apps which will enhance capabilities and improve the end-user experience.
- **End-users**: MCP results in more capable AI applications or agents which can access your data and take actions on your behalf when necessary.

Model Context Protocol Scope

- **MCP Specification:** A specification of MCP that outlines the implementation requirements for clients and servers.
- **MCP SDKs:** SDKs for different programming languages that implement MCP.
- **MCP Development Tools:** Tools for developing MCP servers and clients, including the MCP Inspector.
- **MCP Server Implementations:** Reference implementations of MCP servers.

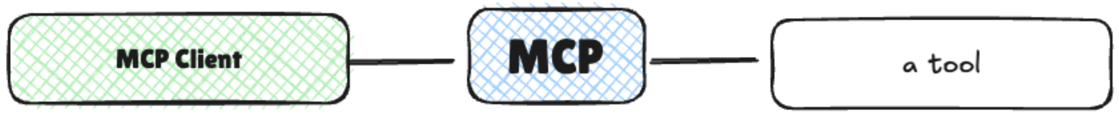
MCP Components (1)



- **Host**: The user-facing AI application that end-users interact with directly. Example: Anthropic's Claude Code, AI-enhanced IDEs like Cursor, inference libraries like HuggingFace Python SDK, or custom applications built in libraries like LangChain or smolagents.

- Hosts initiate connections to MCP Servers and orchestrate the overall flow between user requests, LLM processing, and external tools.

MCP Components (2)



- **Client**: A component within the **host** application that manages communication with a specific MCP Server.
- Each Client maintains a 1:1 connection with a single Server, handling the protocol-level details of MCP communication and acting as an intermediary between the Host's logic and the external Server.

- **Server**: An external program or service that exposes capabilities (Tools, Resources, Prompts) via the MCP protocol. MCP servers can execute locally or remotely.

An Example of MCP Components

- Visual Studio Code acts as an MCP host.
- When Visual Studio Code establishes a connection to an MCP server, such as the HuggingFace MCP server, the Visual Studio Code runtime instantiates an MCP client object that maintains the connection to the HuggingFace MCP server.
- When Visual Studio Code subsequently connects to another MCP server, such as the local filesystem server, the Visual Studio Code runtime instantiates an additional MCP client object to maintain this connection.

Communication Flow (1)

1. **User Interaction**: The user interacts with the **Host** application, expressing an intent or query.
2. **Host Processing**: The **Host** processes the user’s input, potentially using an LLM to understand the request and determine which external capabilities might be needed.
3. **Client Connection**: The **Host** directs its **Client** component to connect to the appropriate Server(s).
4. **Capability Discovery**: The **Client** queries the **Server** to discover what capabilities (Tools, Resources, Prompts) it offers.

Communication Flow (2)

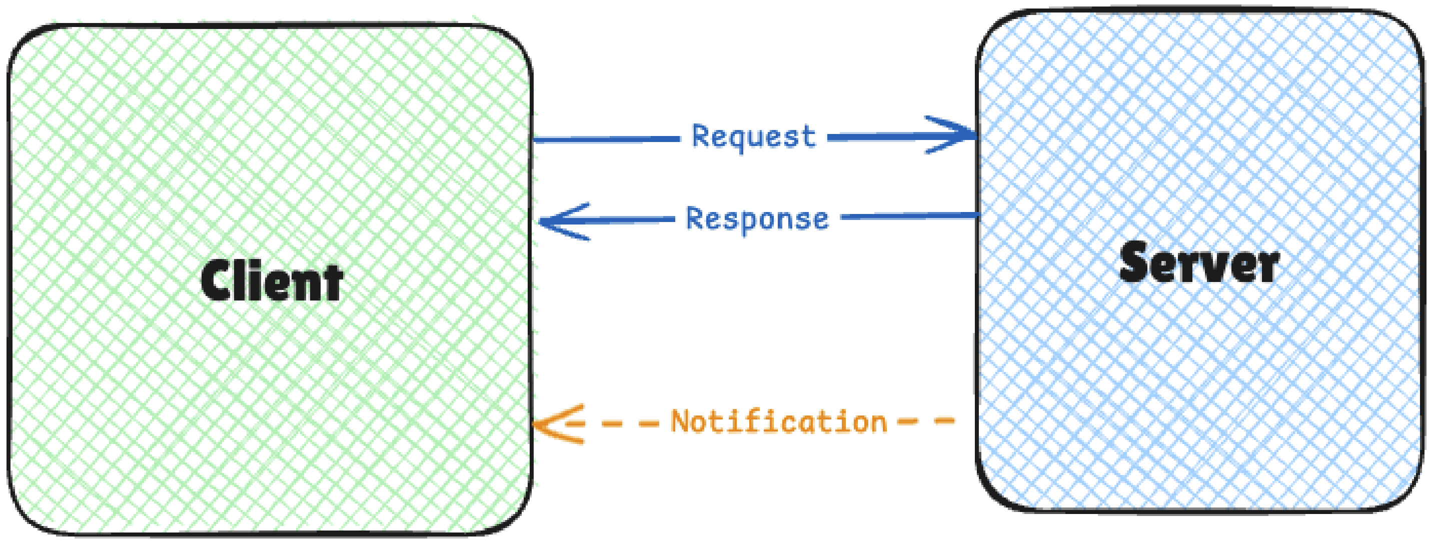
4. Capability Invocation: Based on the user’s needs or the LLM’s determination, the Host instructs the **Client** to invoke specific capabilities from the **Server**.

5. Server Execution: The **Server** executes the requested functionality and returns results to the **Client**.

6. Result Integration: The **Client** relays these results back to the **Host**, which incorporates them into the context for the LLM or presents them directly to the user.

Communication Protocol in MCP

- MCP uses **JSON-RPC 2.0 [1]** as the message format for all communication between Clients and Servers. JSON-RPC is a lightweight remote procedure call protocol encoded in JSON



[1] https://www.jsonrpc.org/specification

Request

- Sent from Client to Server to initiate an operation.
  - A unique identifier (id)
  - The method name to invoke (e.g., tools/call)
  - Parameters for the method (if any)

```json
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "weather",
    "arguments": {
      "location": "San Francisco"
    }
  }
}
```

Response

- Sent from Server to Client in reply to a Request.
  - The same id as the corresponding Request
  - Either a result (for success) or an error (for failure)

Success

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "temperature": 62,
    "conditions": "Partly cloudy"
  }
}
```

Error

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32602,
    "message": "Invalid location parameter"
  }
}
```

Notification

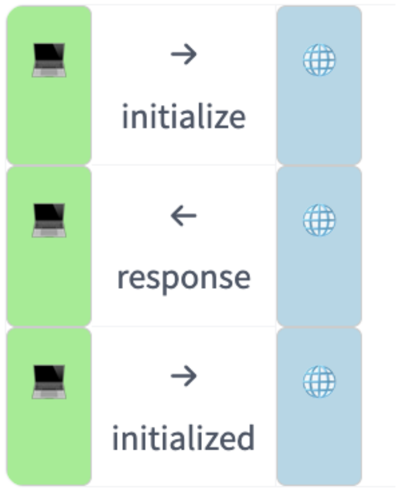
```
{
  "jsonrpc": "2.0",
  "method": "progress",
  "params": {
    "message": "Processing data...",
    "percent": 50
  }
}
```

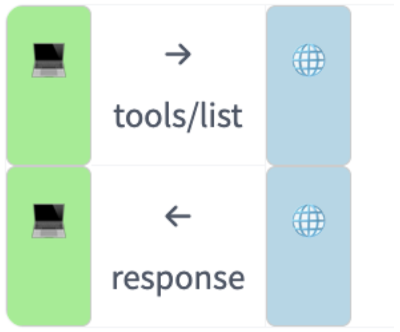
Transport Mechanisms

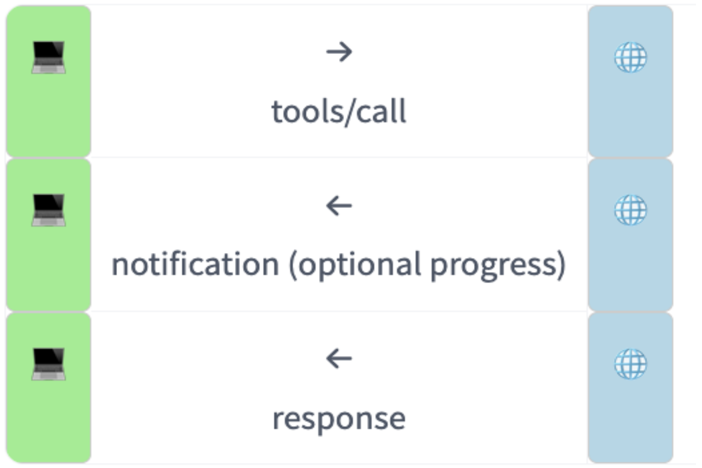
- **stdio (Standard Input/Output)**
  - The stdio transport is used for local communication, where the Client and Server run on the same machine:
  - The Host application launches the Server as a subprocess and communicates with it by writing to its standard input (stdin) and reading from its standard output (stdout).
- **HTTP + SSE (Server-Sent Events) / Streamable HTTP**
  - The HTTP+SSE transport is used for remote communication, where the Client and Server might be on different machines:
  - Communication happens over HTTP, with the Server using Server-Sent Events (SSE) to push updates to the Client over a persistent connection.

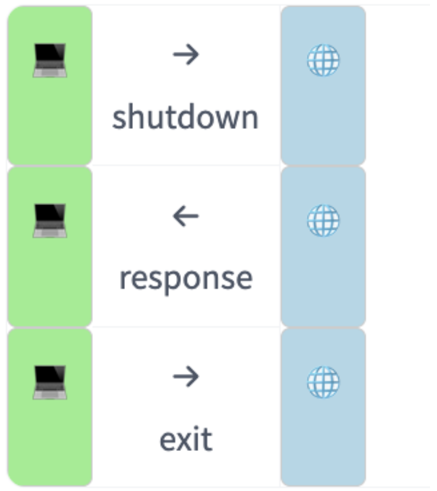
Interaction Lifecycle

Host injects that JSON into model's context









Initialization: exchange protocol versions and capabilities, and the Server responds. (TCP 3-Way Handshake)

Discovery: The Client requests information about available capabilities and the Server responds with JSON describing a list of available tools. (What is the issue here?)

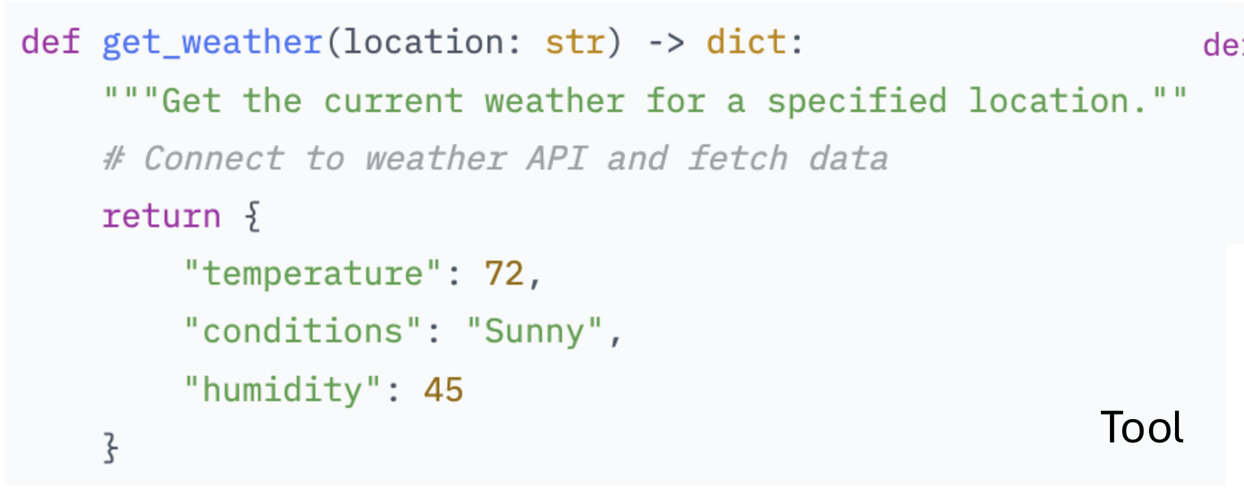
Execution

User prompt triggers model, emitting a structured tool call

Termination

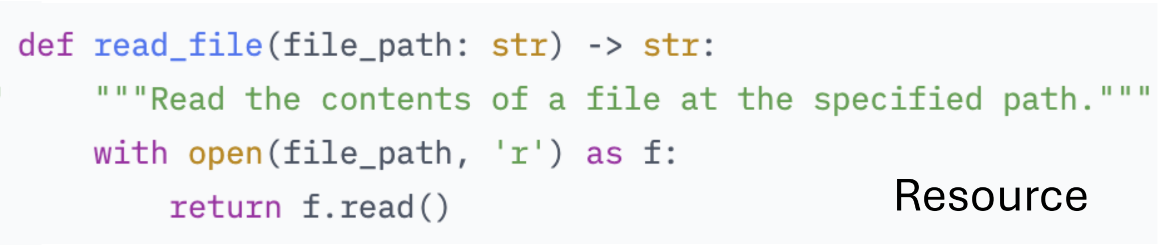
MCP Capabilities (1)

- Tools are executable functions or actions that the AI model can invoke through the MCP protocol.
- Resources provide read-only access to data sources, allowing the AI model to retrieve context without executing complex logic.



```python
def get_weather(location: str) -> dict:
    """Get the current weather for a specified location."""
    # Connect to weather API and fetch data
    return {
        "temperature": 72,
        "conditions": "Sunny",
        "humidity": 45
    }
```

Tool

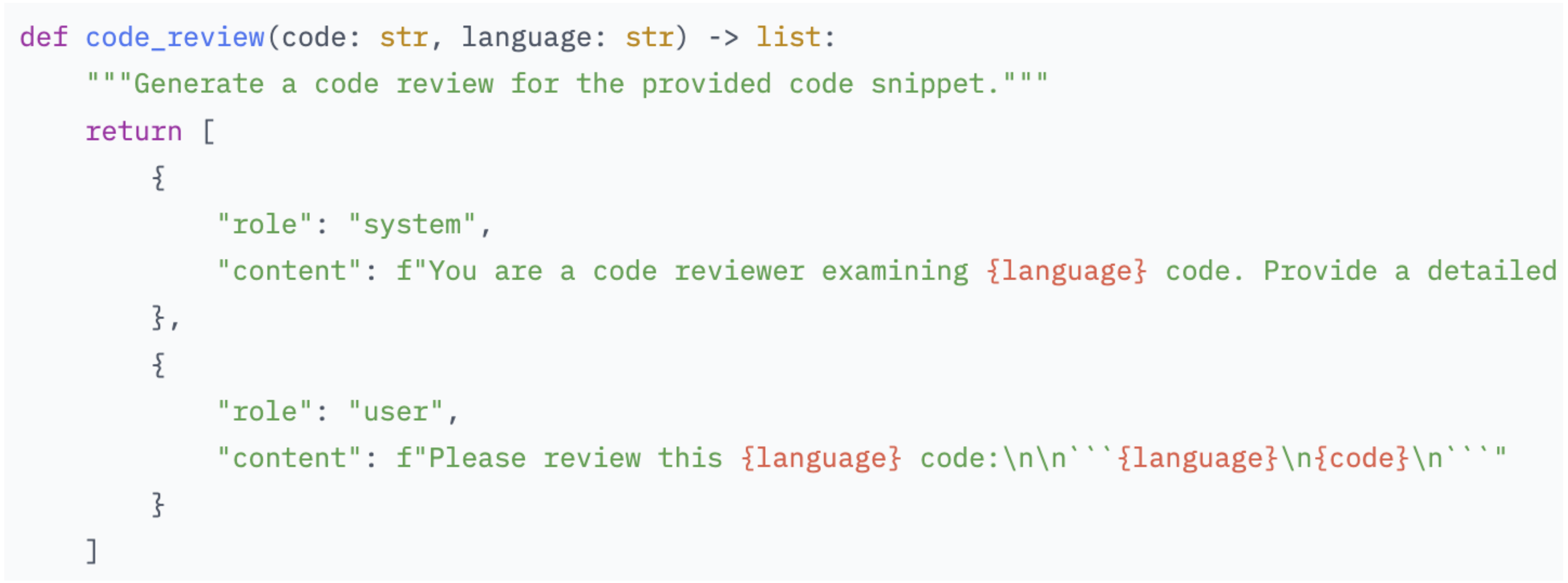


```python
def read_file(file_path: str) -> str:
    """Read the contents of a file at the specified path."""
    with open(file_path, 'r') as f:
        return f.read()
```

Resource

MCP Capabilities (2)

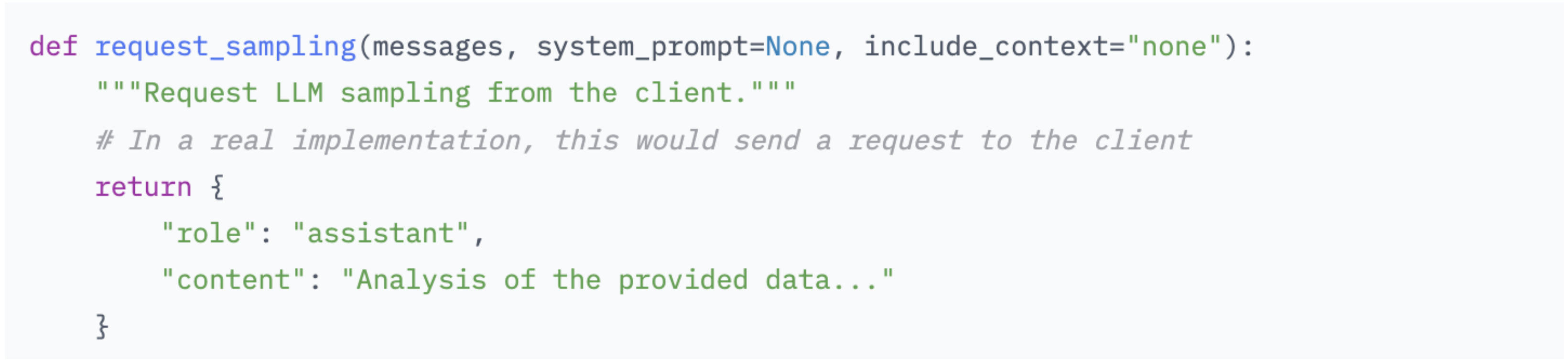
- Prompts are predefined templates or workflows that guide the interaction between the user, the AI model, and the Server's capabilities.



```python
def code_review(code: str, language: str) -> list:
    """Generate a code review for the provided code snippet."""
    return [
        {
            "role": "system",
            "content": f"You are a code reviewer examining {language} code. Provide a detailed
        },
        {
            "role": "user",
            "content": f"Please review this {language} code:\n\n```{language}\n{code}\n```"
        }
    ]
```

MCP Capabilities (3)

- Sampling allows Servers to request the Client (specifically, the Host application) to perform LLM interactions.



```python
def request_sampling(messages, system_prompt=None, include_context="none"):
    """Request LLM sampling from the client."""
    # In a real implementation, this would send a request to the client
    return {
        "role": "assistant",
        "content": "Analysis of the provided data..."
    }
```

1. Server sends a sampling/createMessage request to the client
2. Client reviews the request and can modify it
3. Client samples from an LLM
4. Client reviews the completion
5. Client returns the result to the server

MCP Capabilities Work Together

1. A user might select a **Prompt** to start a specialized workflow
2. The Prompt might include context from **Resources**
3. During processing, the AI model might call **Tools** to perform specific actions
4. For complex operations, the Server might use **Sampling** to request additional LLM processing

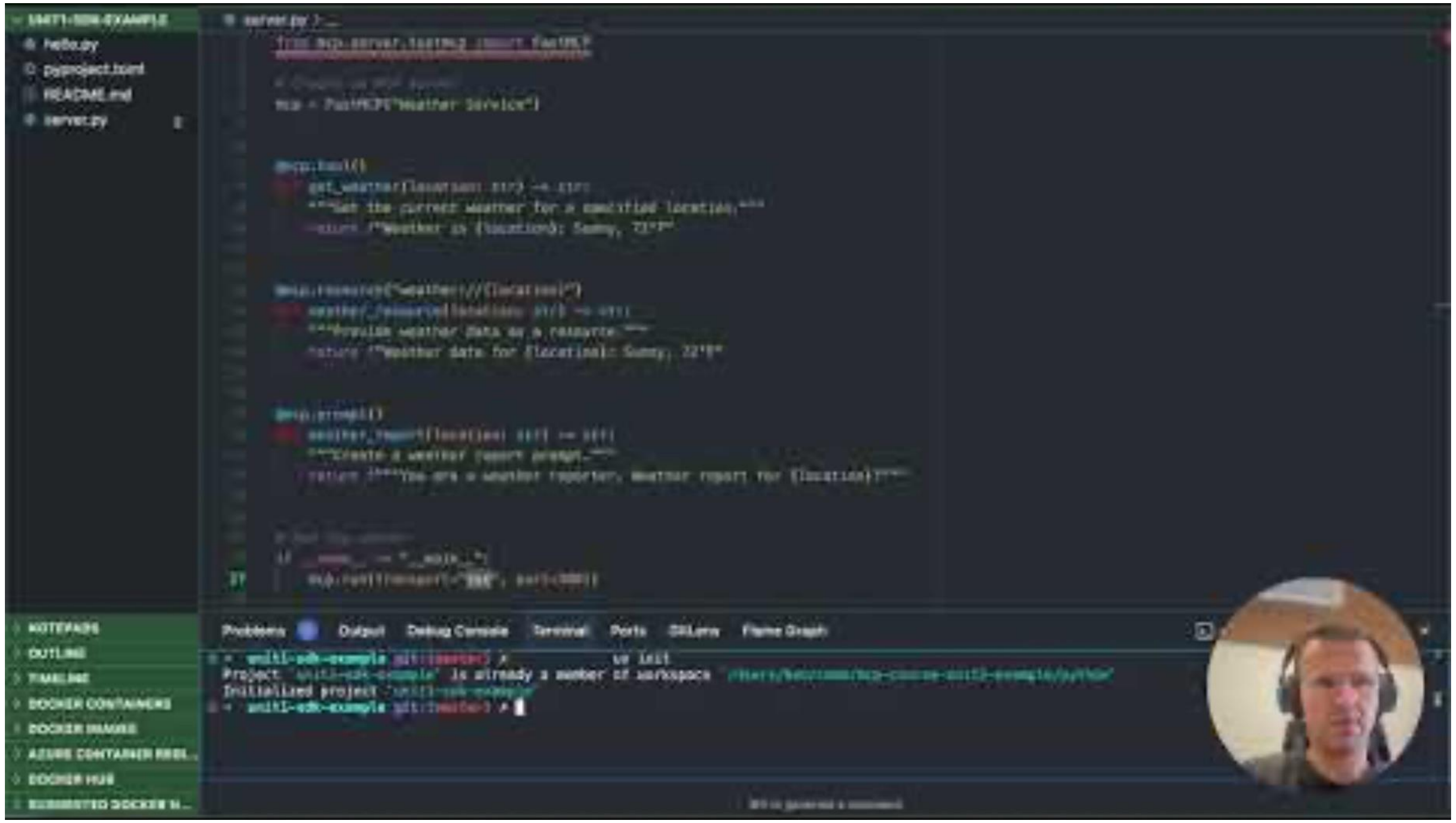
MCP Software Development Kit (SDK)

Language	Repository	Maintainer(s)	Status
TypeScript	github.com/modelcontextprotocol/typescript-sdk	Anthropic	Active
Python	github.com/modelcontextprotocol/python-sdk	Anthropic	Active
Java	github.com/modelcontextprotocol/java-sdk	Spring AI (VMware)	Active
Kotlin	github.com/modelcontextprotocol/kotlin-sdk	JetBrains	Active

SDK makes it easier to implement MCP clients and servers

- Protocol-level communication
- Capability registration and discovery
- Message serialization/deserialization
- Connection management and Error handling

MCP Server Implementation Example

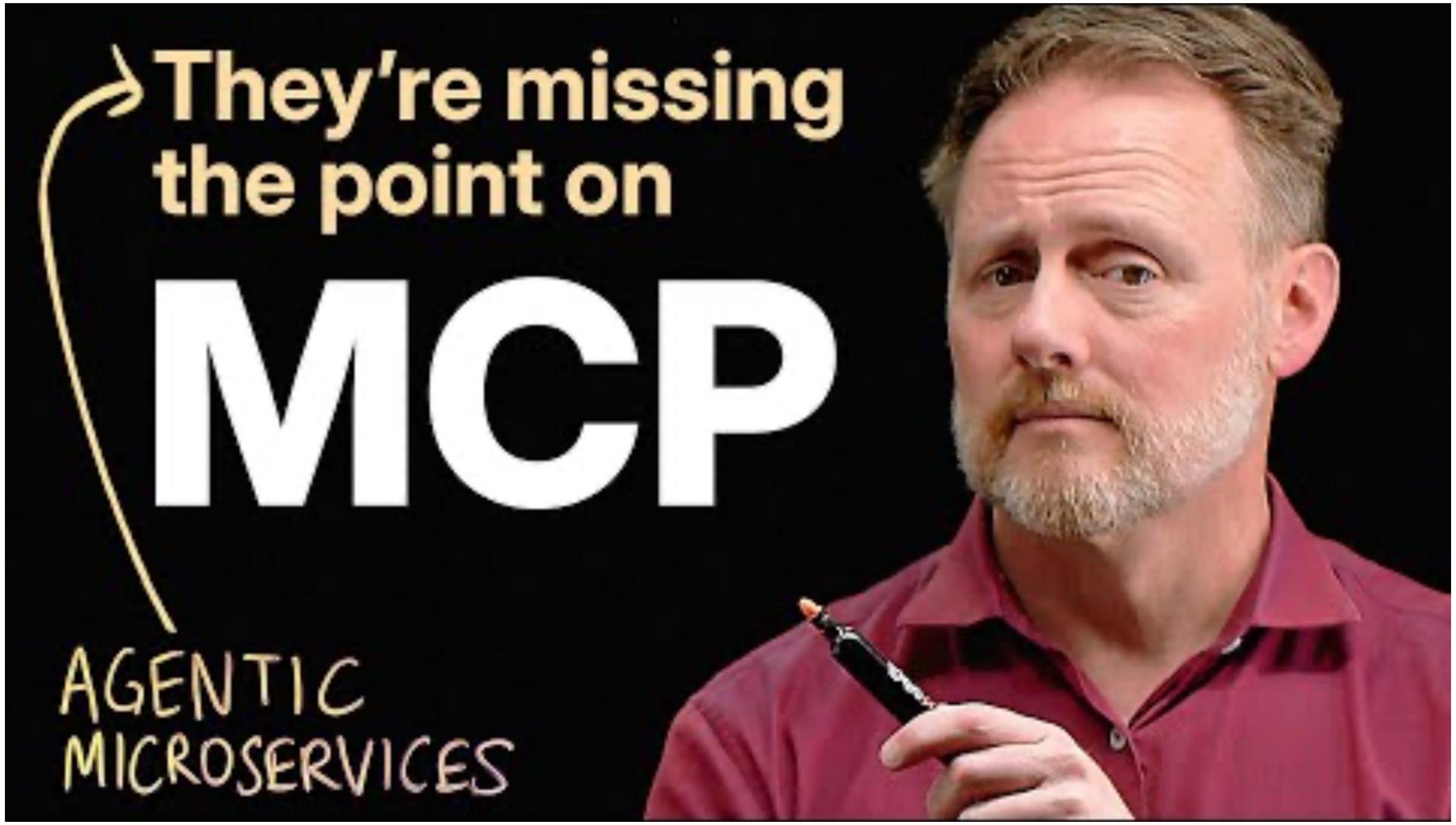


```
from mcp.server.fastmcp import FastMCP

# [illegible]
mcp = FastMCP("Weather Service")


@mcp.tool()
[illegible] get_weather(location: str) -> str:
    """Get the current weather for a specified location."""
    return f"Weather in {location}: Sunny, 72°F"


@mcp.resource("weather://{location}")
[illegible] weather_resource(location: str) -> str:
    """Provide weather data as a resource."""
    return f"Weather data for {location}: Sunny, 72°F"


@mcp.prompt()
[illegible] weather_report(location: str) -> str:
    """Create a weather report prompt."""
    return f"""You are a weather reporter. Weather report for {location}?"""


# [illegible]
if __name__ == "__main__":
    mcp.run(transport="[illegible]", port=[illegible])
```

```
➜  unit1-sdk-example git:(master) ✗          uv init
Project `unit1-sdk-example` is already a member of workspace `[illegible]`
Initialized project `unit1-sdk-example`
➜  unit1-sdk-example git:(master) ✗
```

https://www.youtube.com/watch?v=exzrb5QNUis

A Quick Summary of MCP



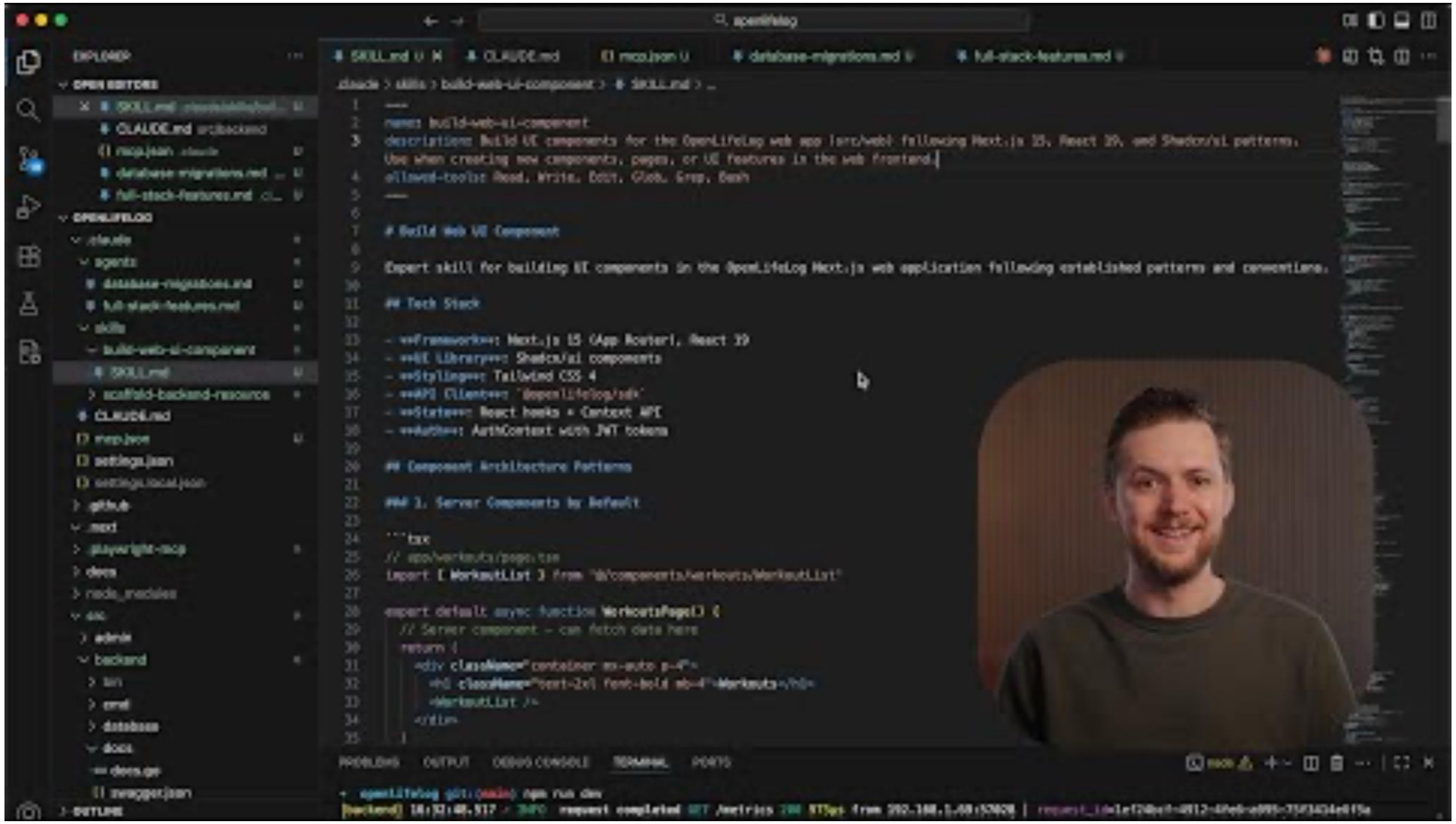
https://www.youtube.com/watch?v=FLpS7OfD5-s

Limitations of MCP

- Each tool and its description takes up huge context window space.
- Agents don’t handle many tools very well today [1].
- Security issues of MCP [2].

[1] Less is More: Optimizing Function Calling for LLM Execution on Edge Devices https://arxiv.org/abs/2411.15399
[2] https://github.com/cosai-oasis/ws4-secure-design-agentic-systems/blob/main/model-context-protocol-security.md

Agent Skills

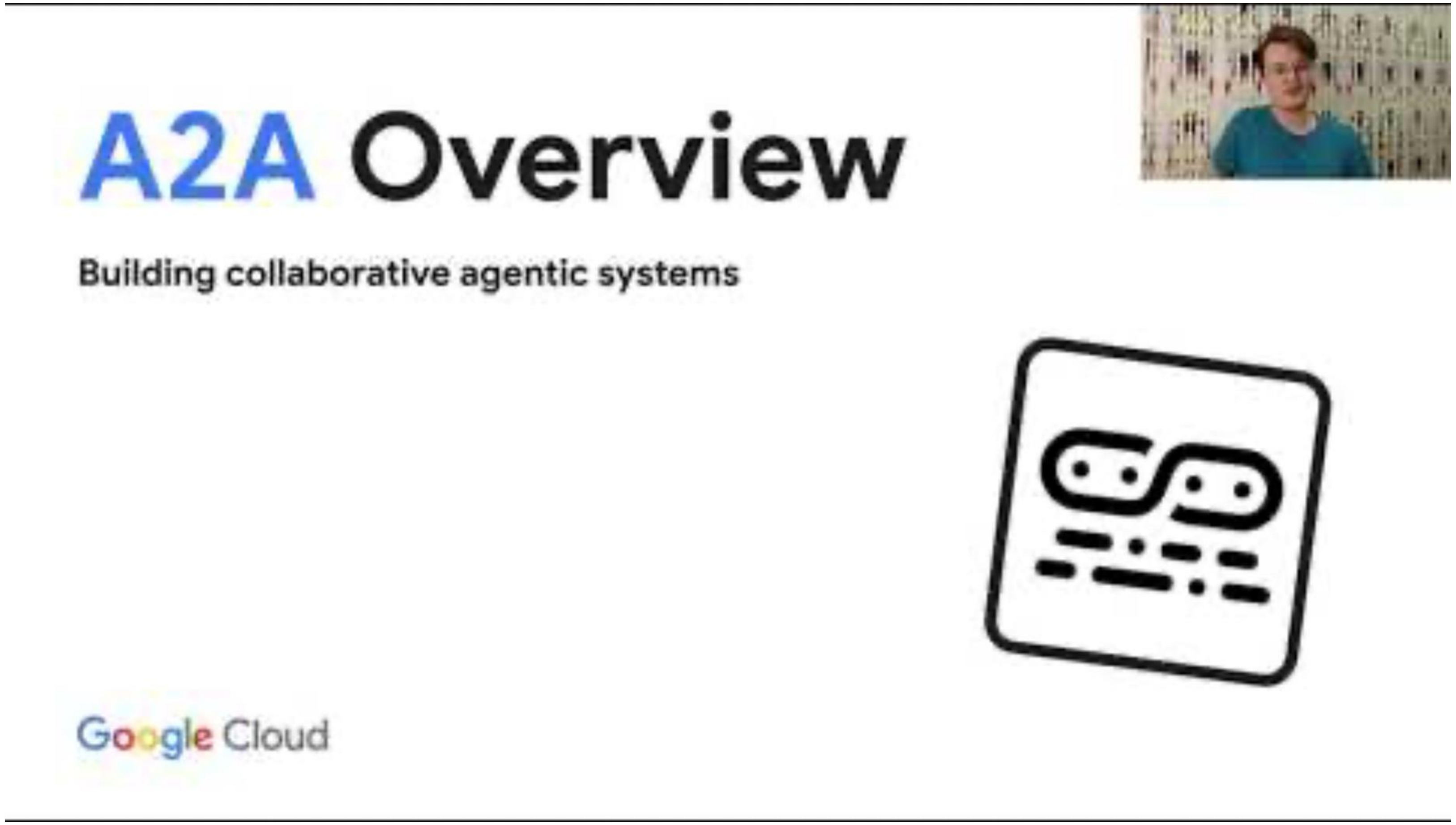


~~~
---
name: build-web-ui-component
description: Build UI components for the OpenLifeLog web app (src/web) following Next.js 15, React 19, and Shadcn/ui patterns.
Use when creating new components, pages, or UI features in the web frontend.
allowed-tools: Read, Write, Edit, Glob, Grep, Bash
---

# Build Web UI Component

Expert skill for building UI components in the OpenLifeLog Next.js web application following established patterns and conventions.

## Tech Stack

- **Framework**: Next.js 15 (App Router), React 19
- **UI Library**: Shadcn/ui components
- **Styling**: Tailwind CSS 4
- **API Client**: `@openlifelog/sdk`
- **State**: React hooks + Context API
- **Auth**: AuthContext with JWT tokens

## Component Architecture Patterns

### 1. Server Components by Default

```tsx
// app/workouts/page.tsx
import { WorkoutList } from "@/components/workouts/WorkoutList"

export default async function WorkoutsPage() {
  // Server component - can fetch data here
  return (
    <div className="container mx-auto p-4">
      <h1 className="text-2xl font-bold mb-4">Workouts</h1>
      <WorkoutList />
    </div>
  )
~~~

```
→ openlifelog git:(main) npm run dev
[backend] 18:32:48.517 › INFO request completed GET /metrics 200 8T5µs from 192.168.1.68:57028 | request_id=[illegible]
```

https://www.youtube.com/watch?v=fOxC44g8vig

Agent2Agent Protocol



A2A Overview

Building collaborative agentic systems

Google Cloud

https://www.youtube.com/watch?v=Fbr_Solax1w

Optional: Try Claude Code

- Vibe Coding Environment
  - **Claude Code: https://code.claude.com/docs/en/desktop**
  - **Cursor**
  - GitHub Copilot
  - Codex
  - …
- Next Generation of Agent?
  - If LLM coding is strong enough, we may only need **Bash+File System**

References

- https://huggingface.co/learn/mcp-course/
- https://modelcontextprotocol.io/docs/getting-started/intro
- https://github.com/modelcontextprotocol/servers
- https://developers.cloudflare.com/agents/guides/remote-mcp-server/
- https://github.com/modelcontextprotocol/