2025 EDITION

FREE

MCP

THE ILLUSTRATED GUIDEBOOK

Daily Dose of Data Science

Avi Chawla & Akshay Pachaar
DailyDoseofDS.com

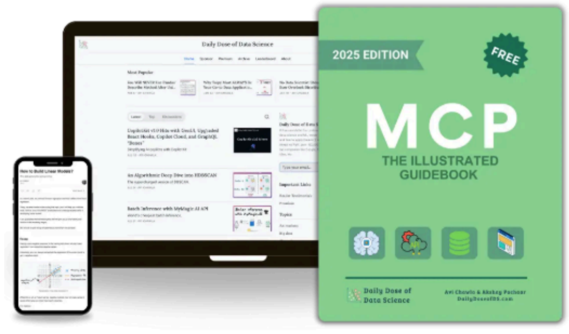
How to make the most out of this book and your time?

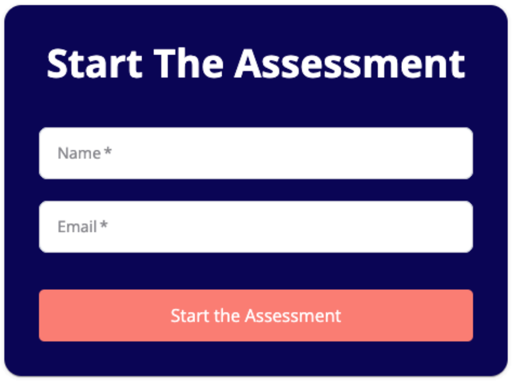
The reading time of this book is about 3 hours. But not all chapters will be of relevance to you. This 2-minute assessment will test your current expertise and recommend chapters that will be most useful to you.

Are you MCP-aware?

Answer 8 yes/no questions and we'll email you the list of chapters that must read to improve your MCP skillset.

Take the Assessment Now!





Scan the QR code below or open this link to start the assessment. It will only take 2 minutes to complete.

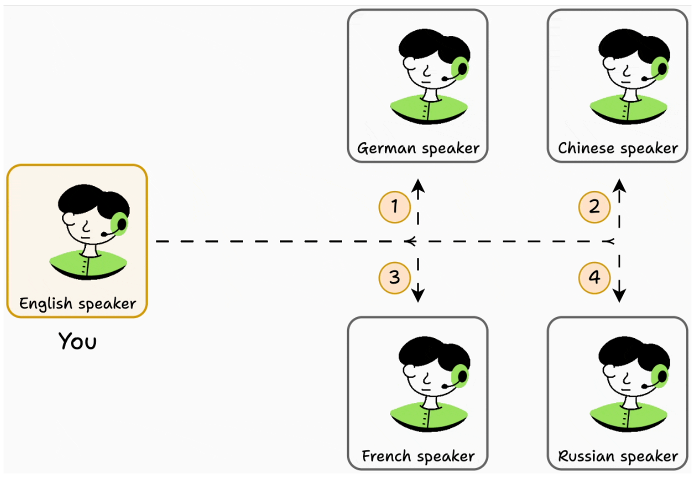
	https://bit.ly/mcp-assessment

Table of contents

Model Context Protocol (MCP)

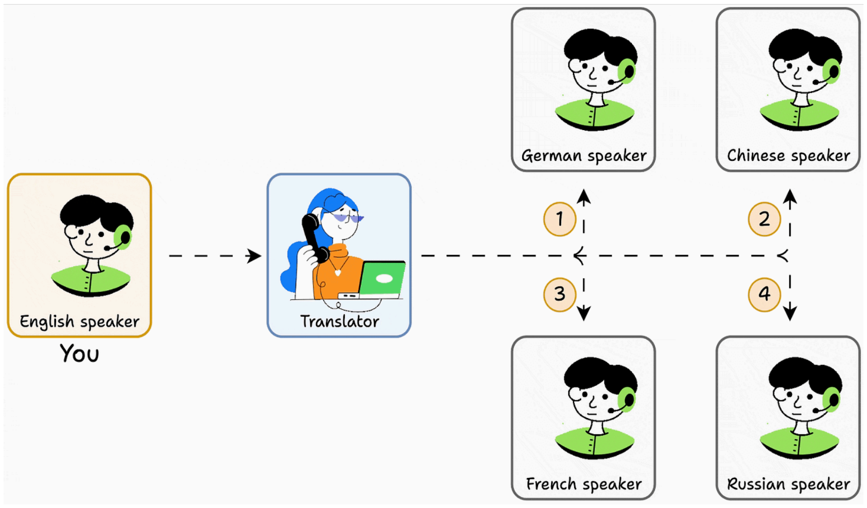
What is MCP?

Imagine you only know English. To get info from a person who only knows:



- French, you must learn French.
- German, you must learn German.
- And so on.

In this setup, learning even 5 languages will be a nightmare for you.
But what if you add a translator that understands all languages?

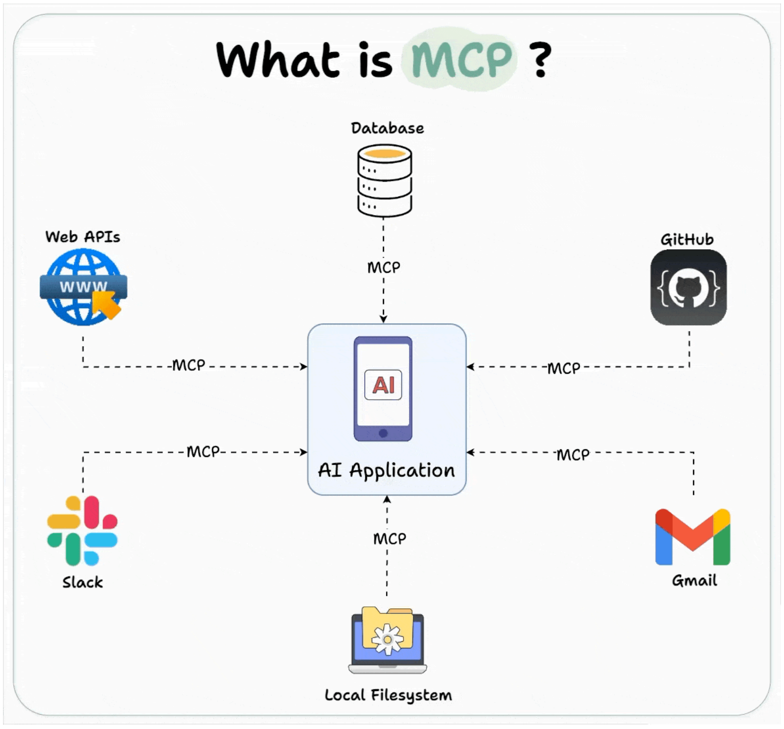


This is simple, isn't it?

The translator is like an MCP!

It lets you (Agents) talk to other people (tools or other capabilities) through a single interface.

To formalize, while LLMs possess impressive knowledge and reasoning skills, which allow them to perform many complex tasks, their knowledge is limited to their initial training data.



If they need to access real-time information, they must use external tools and resources on their own.

Model context protocol (MCP) is a standardized interface and framework that allows AI models to seamlessly interact with external tools, resources, and environments.

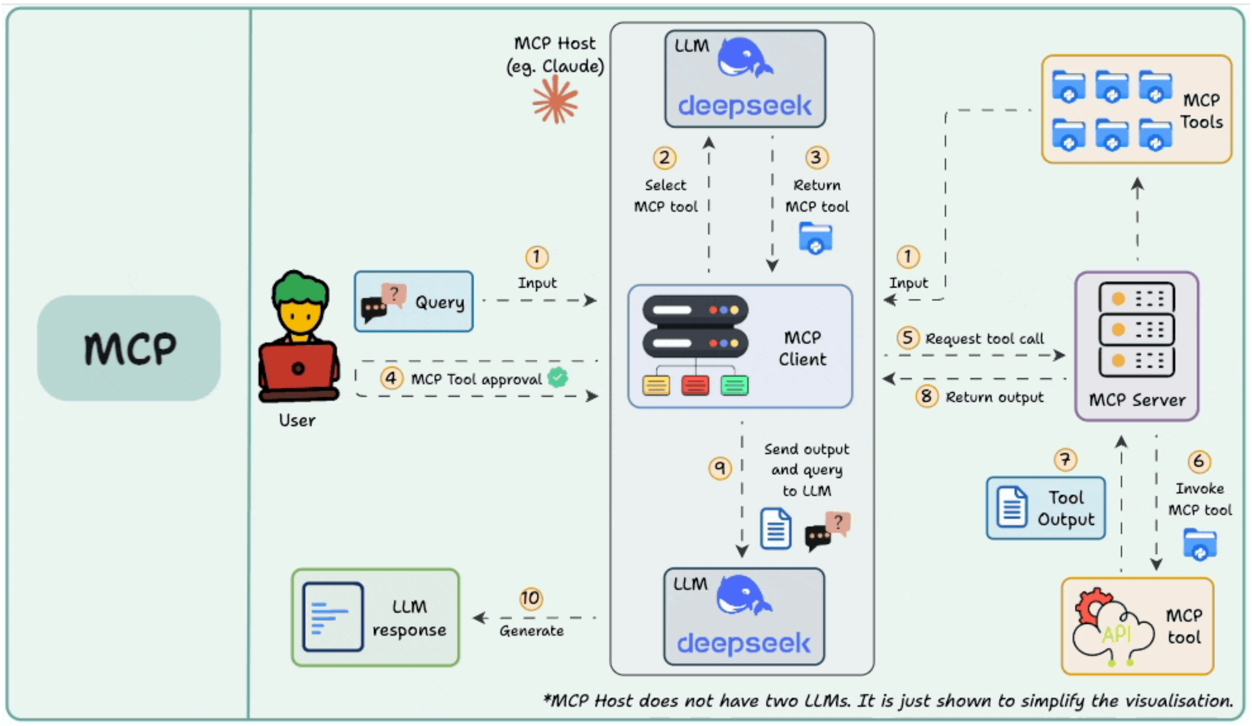
MCP acts as a universal connector for AI systems to capabilities (tools, etc.), similar to how USB-C standardizes connections between electronic devices.

Why was MCP created?

Without MCP, adding a new tool or integrating a new model was a headache.

If you had three AI applications and three external tools, you might end up writing nine different integration modules (each AI x each tool) because there was no common standard. This doesn't scale.

Developers of AI apps were essentially reinventing the wheel each time, and tool providers had to support multiple incompatible APIs to reach different AI platforms.



Let's understand this in detail.

The problem

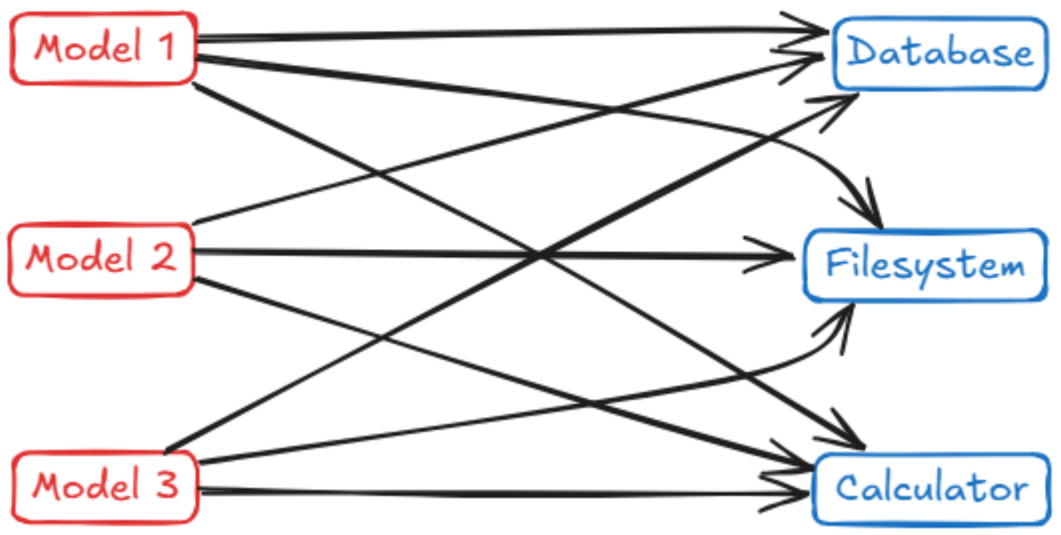
Before MCP, the landscape of connecting AI to external data and actions looked like a patchwork of one-off solutions.

Either you hard-coded logic for each tool, managed prompt chains that were not robust, or you used vendor-specific plugin frameworks.

This led to the infamous M×N integration problem.

Essentially, if you have M different AI applications and N different tools/data sources, you could end up needing M × N custom integrations.

The diagram below illustrates this complexity: each AI (each "Model") might require unique code to connect to each external service (database, filesystem, calculator, etc.), leading to spaghetti-like interconnections.

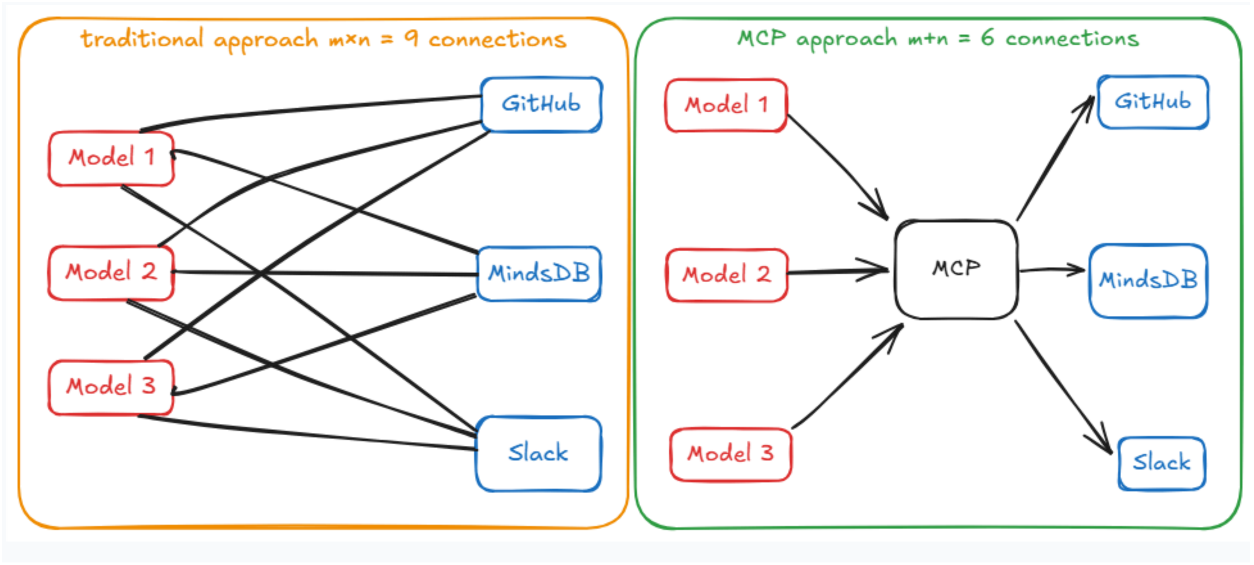


The solution

MCP tackles this by introducing a standard interface in the middle. Instead of M × N direct integrations, we get M + N implementations: each of the M AI applications implements the MCP client side once, and each of the N tools implements an MCP server once.

Now everyone speaks the same "language", so to speak, and a new pairing doesn't require custom code since they already understand each other via MCP.

The following diagram illustrates this shift.

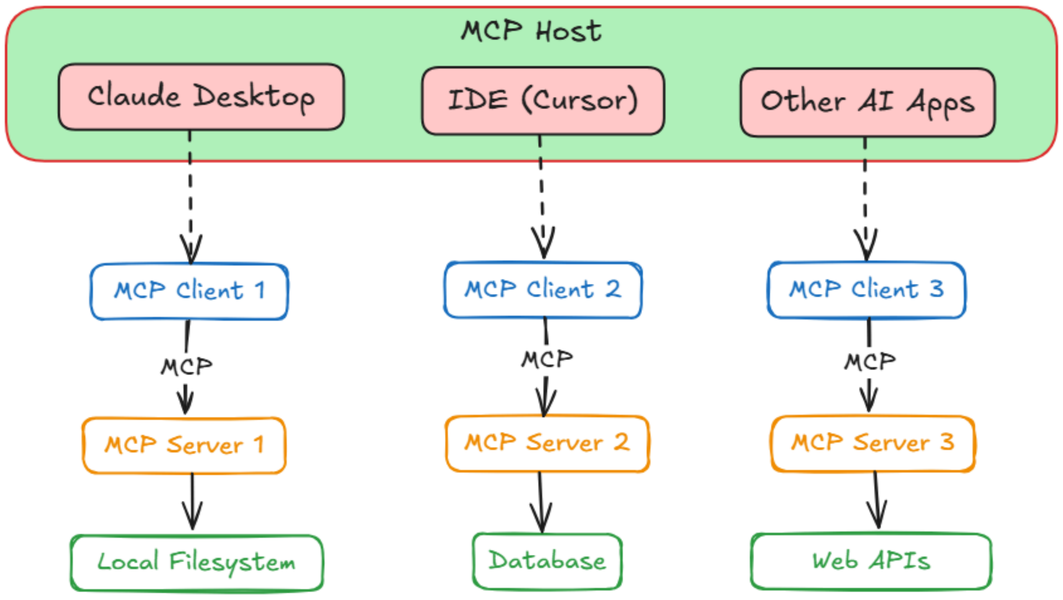


- On the left (pre-MCP), every model had to wire into every tool.
- On the right (with MCP), each model and tool connects to the MCP layer, drastically simplifying connections. You can also relate this to the translator example we discussed earlier.

MCP Architecture Overview

At its heart, MCP follows a client-server architecture (much like the web or other network protocols).

However, the terminology is tailored to the AI context. There are three main roles to understand: the Host, the Client, and the Server.

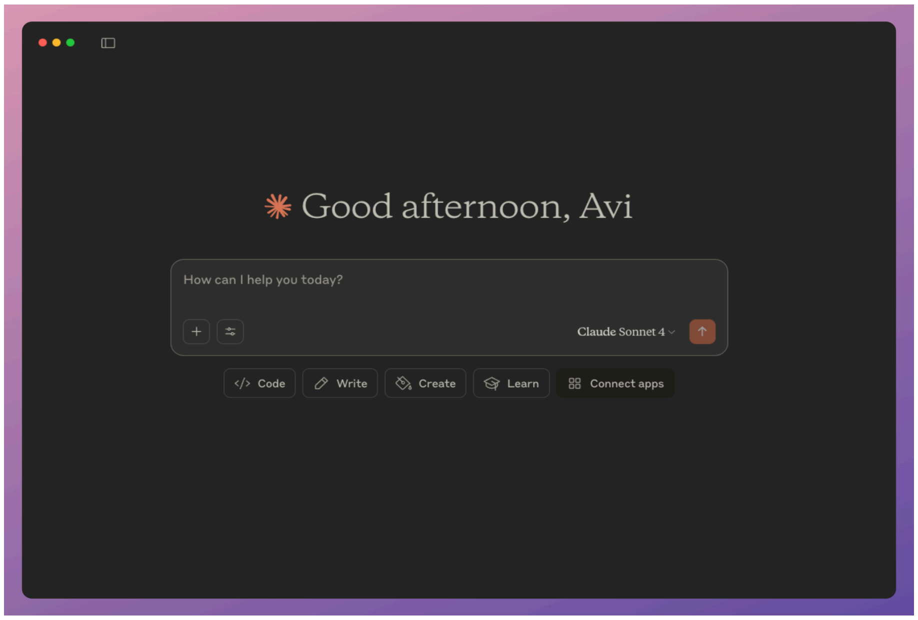


Host

The Host is the user-facing AI application, the environment where the AI model lives and interacts with the user.

This could be a chat application (like OpenAI's ChatGPT interface or Anthropic's Claude desktop app), an AI-enhanced IDE (like Cursor), or any custom app that embeds an AI assistant like Chainlit.

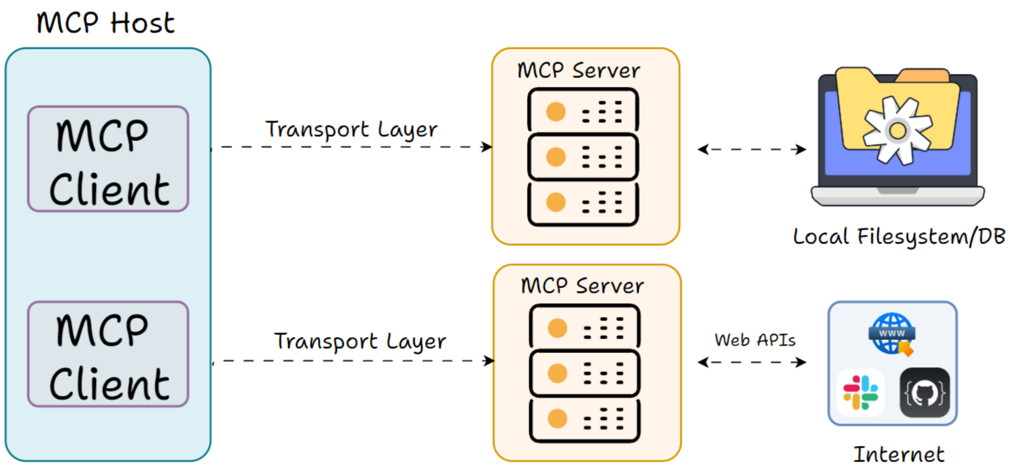
Host is the one that initiates connections to the available MCP servers when the system needs them. It captures the user's input, keeps the conversation history, and displays the model's replies.



Client

The MCP Client is a component within the Host that handles the low-level communication with an MCP Server.

Think of the Client as the adapter or messenger. While the Host decides what to do, the Client knows how to speak MCP to actually carry out those instructions with the server.

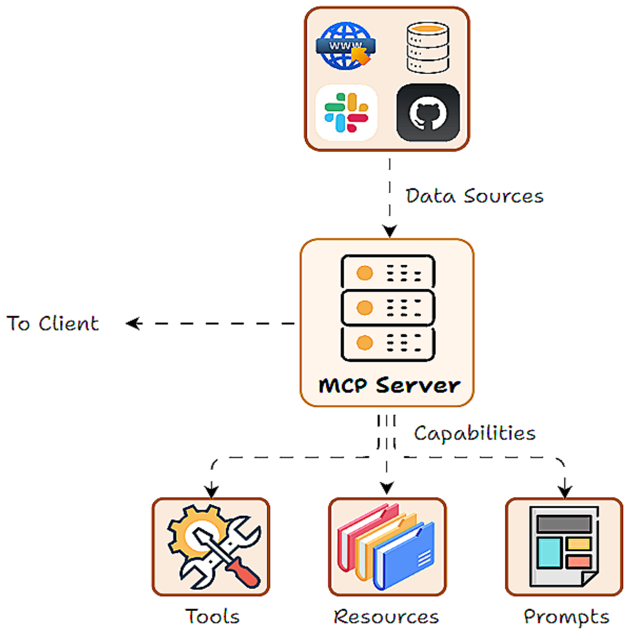


Server

The MCP Server is the external program or service that actually provides the capabilities (tools, data, etc.) to the application.

An MCP Server can be thought of as a wrapper around some functionality, which exposes a set of actions or resources in a standardized way so that any MCP Client can invoke them.

Servers can run locally on the same machine as the Host or remotely on some cloud service since MCP is designed to support both scenarios seamlessly. The key is that the Server advertises what it can do in a standard format (so the client can query and understand available tools) and will execute requests coming from the client, then return results.



Tools, Resources and Prompts

Tools, prompts and resources form the three core capabilities of the MCP framework. Capabilities are essentially the features or functions that the server makes available.

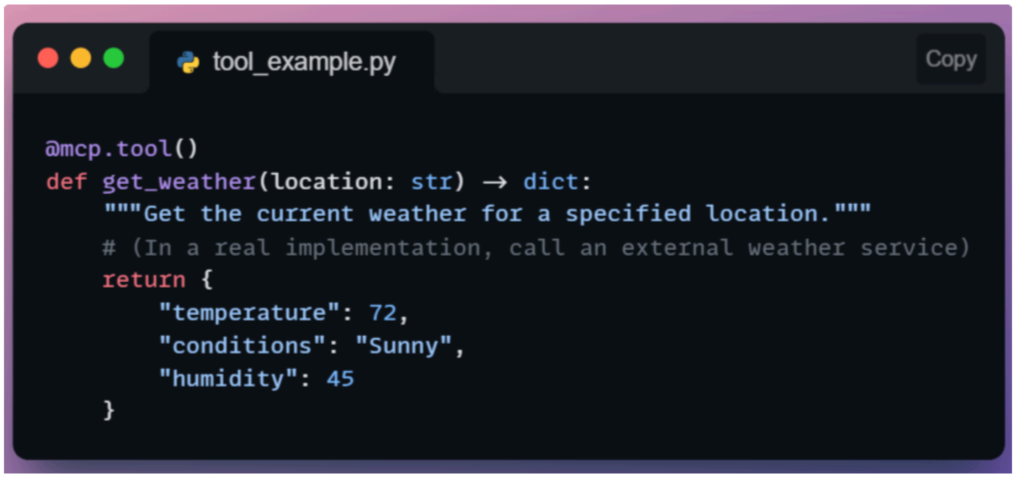
- Tools: Executable actions or functions that the AI (host/client) can invoke (often with side effects or external API calls).
- Resources: Read-only data sources that the AI (host/client) can query for information (no side effects, just retrieval).
- Prompts: Predefined prompt templates or workflows that the server can supply.

Tools

Tools are what they sound like: functions that do something on behalf of the AI model. These are typically operations that can have effects or require computation beyond the AI's own capabilities.

Importantly, Tools are usually triggered by the AI model's choice, which means the LLM (via the host) decides to call a tool when it determines it needs that functionality.

Suppose we have a simple tool for weather. In an MCP server's code, it might look like:



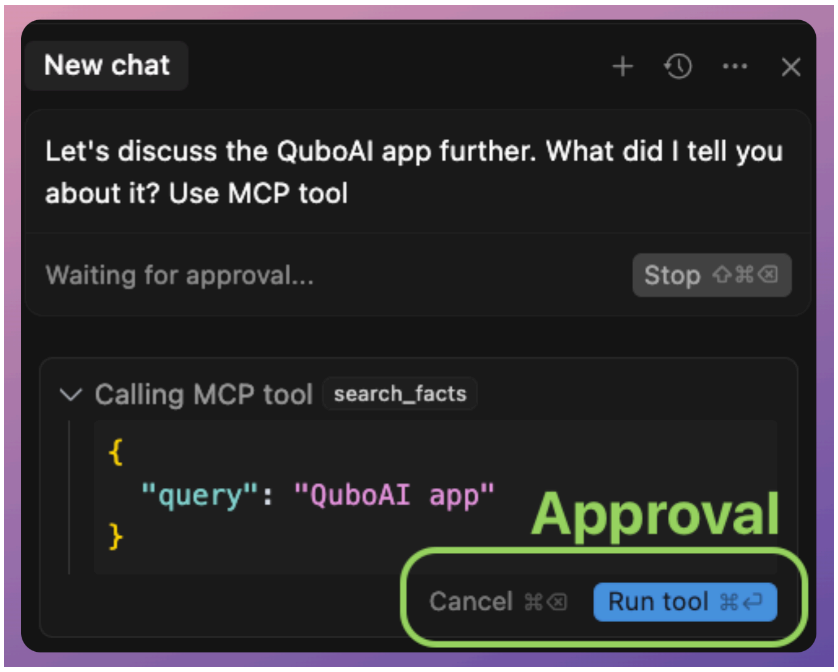
```python
@mcp.tool()
def get_weather(location: str) → dict:
    """Get the current weather for a specified location."""
    # (In a real implementation, call an external weather service)
    return {
        "temperature": 72,
        "conditions": "Sunny",
        "humidity": 45
    }
```

This Python function, registered with @mcp.tool(), can be invoked by the AI via MCP.

When the AI calls tools/call with name "get_weather" and {"location": "San Francisco"} as arguments, the server will execute get_weather("San Francisco") and return the dictionary result.

The client will get that JSON result and make it available to the AI. Notice the tool returns structured data (temperature, conditions), and the AI can then use or verbalize (generate a response) that info.

Since tools can do things like file I/O or network calls, an MCP implementation often requires that the user permit a tool call.



Let's discuss the QuboAI app further. What did I tell you about it? Use MCP tool

Waiting for approval...

Calling MCP tool search_facts

```
{
  "query": "QuboAI app"
}
```

For example, Claude's client might pop up "The AI wants to use the 'get_weather' tool, allow yes/no?" the first time, to avoid abuse. This ensures the human stays in control of powerful actions.

Tools are analogous to "functions" in classic function calling, but under MCP, they are used in a more flexible, dynamic context. They are model-controlled but developer/governance-approved in execution.

Resources

Resources provide read-only data to the AI model.

These are like databases or knowledge bases that the AI can query to get information, but not modify.

Unlike tools, resources typically do not involve heavy computation or side effects, since they are often just information lookup.

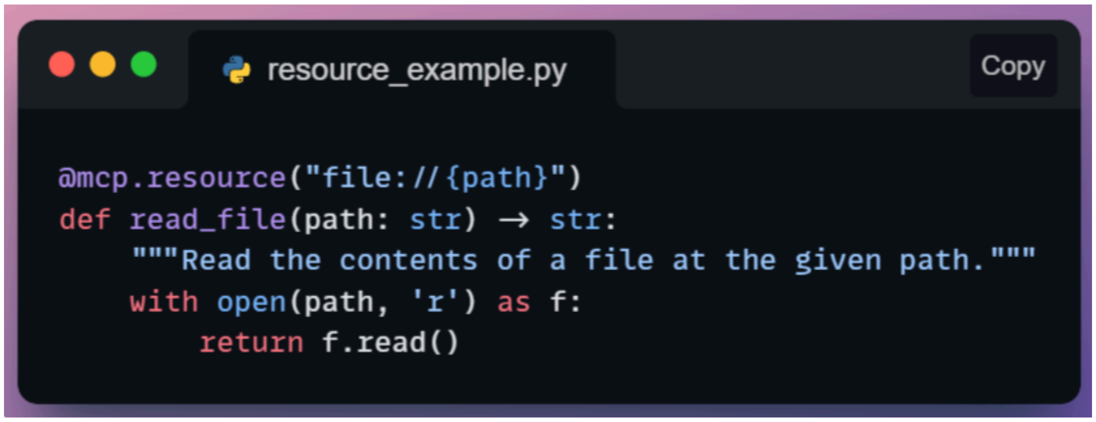
Another key difference is that resources are usually accessed under the host application's control (not spontaneously by the model). In practice, this might mean the Host knows when to fetch a certain context for the model.

For instance, if a user says, “Use the company handbook to answer my question,” the Host might call a resource that retrieves relevant handbook sections and feeds them to the model.

Resources could include a local file’s contents, a snippet from a knowledge base or documentation, a database query result (read-only), or any static data like configuration info.

Essentially anything the AI might need to know as context. An AI research assistant could have resources like “ArXiv papers database,” where it can retrieve an abstract or reference when asked.

A simple resource could be a function to read a file:



```python
@mcp.resource("file://{path}")
def read_file(path: str) -> str:
    """Read the contents of a file at the given path."""
    with open(path, 'r') as f:
        return f.read()
```

Here we use a decorator @mcp.resource("file://{path}") which might indicate a template for resource URIs.

The AI (or Host) could ask the server for resources.get with a URI like file://home/user/notes.txt, and the server would callread_file("/home/user/notes.txt") and return the text.

Notice that resources are usually identified by some identifier (like a URI or name) rather than being free-form functions.

They are also often application-controlled, meaning the app decides when to retrieve them (to avoid the model just reading everything arbitrarily).

From a safety standpoint, since resources are read-only, they are less dangerous, but still, one must consider privacy and permissions (the AI shouldn't read files it's not supposed to).

The Host can regulate which resource URIs it allows the AI to access, or the server might restrict access to certain data.

In summary, Resources give the AI knowledge without handing over the keys to change anything.

They're the MCP equivalent of giving the model reference material when needed, which acts like a smarter, on-demand retrieval system integrated through the protocol.

Prompts

Prompts in the MCP context are a special concept: they are predefined prompt templates or conversation flows that can be injected to guide the AI's behavior.

Essentially, a Prompt capability provides a canned set of instructions or an example dialogue that can help steer the model for certain tasks.

But why have prompts as a capability?

Think of recurring patterns: e.g., a prompt that sets up the system role as "You are a code reviewer," and the user's code is inserted for analysis.

Rather than hardcoding that in the host application, the MCP server can supply it.

Prompts can also represent multi-turn workflows.

For instance, a prompt might define how to conduct a step-by-step diagnostic interview with a user. By exposing this via MCP, any client can retrieve and use these sophisticated prompts on demand.

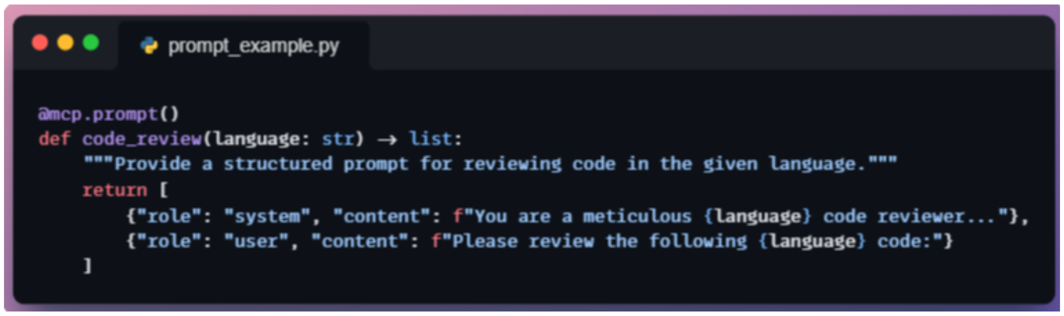
As far as control is concerned, Prompts are usually user-controlled or developer-controlled.

The user might pick a prompt/template from a UI (e.g., “Summarize this document” template), which the host then fetches from the server.

The model doesn’t spontaneously decide to use prompts the way it does tools.

Rather, the prompt sets the stage before the model starts generating. In that sense, prompts are often fetched at the beginning of an interaction or when the user chooses a specific “mode”.

Suppose we have a prompt template for code review. The MCP server might have:



```python
@mcp.prompt()
def code_review(language: str) → list:
    """Provide a structured prompt for reviewing code in the given language."""
    return [
        {"role": "system", "content": f"You are a meticulous {language} code reviewer..."},
        {"role": "user", "content": f"Please review the following {language} code:"}
    ]
```

This prompt function returns a list of message objects (in OpenAI format) that set up a code review scenario.

When the host invokes this prompt, it gets those messages and can insert the actual code to be reviewed into the user content.

Then it provides these messages to the model before the model’s own answer. Essentially, the server is helping to structure the conversation.

While we have personally not seen much applicability of this yet, common use cases for prompt capabilities include things like “brainstorming guide,” “step-by-step problem solver template,” or domain-specific system roles.

By having them on the server, they can be updated or improved without changing the client app, and different servers can offer different specialized prompts.

An important point to note here is that prompts, as a capability, blur the line between data and instructions.

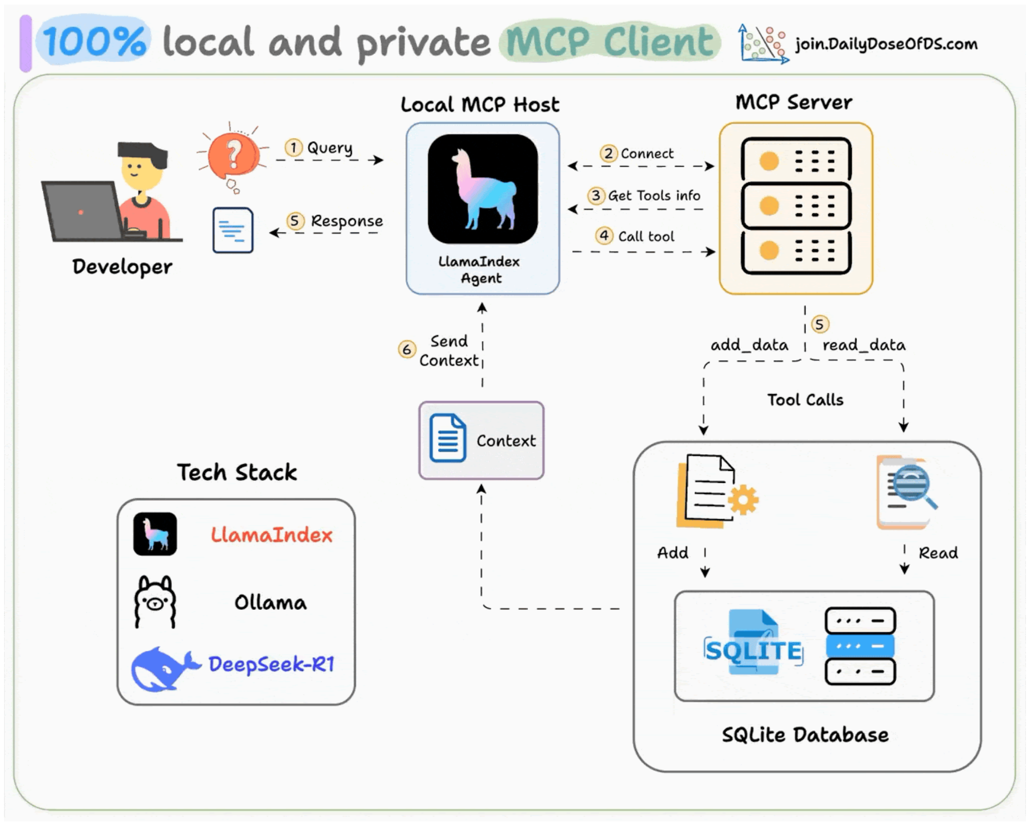
They represent best practices or predefined strategies for the AI to use.

In a way, MCP prompts are similar to how ChatGPT plugins can suggest how to format a query, but here it′s standardized and discoverable via the protocol.

MCP Projects

#1) 100% local MCP Client

An MCP client is a component in an AI app (like Cursor) that establishes connections to external tools. Learn how to build it 100% locally.



Tech stack:

- Llamaindex to build the MCP-powered Agent
- Ollama to locally serve Deepseek-R1.
- LightningAI for development and hosting

Workflow:

- User submits a query.
- Agent connects to the MCP server to discover tools.
- Based on the query, agent invokes the right tool and get context
- Agent returns a context-aware response.

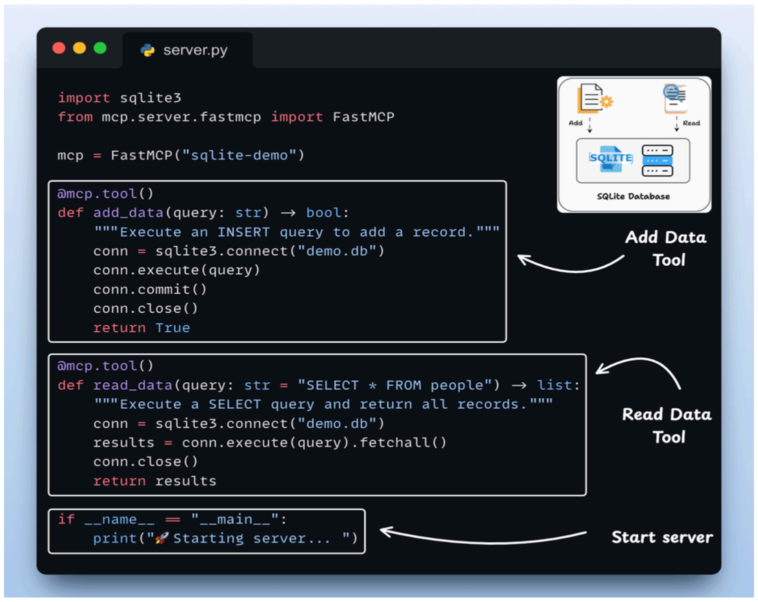
Let's implement this!

#1) Build an SQLite MCP Server

For this demo, we've built a simple SQLite server with two tools:

- add data
- fetch data

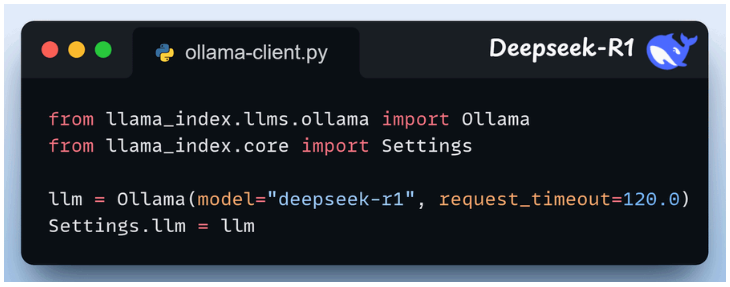
This is done to keep things simple, but the client we're building can connect to any MCP server out there.



```python
import sqlite3
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("sqlite-demo")

@mcp.tool()
def add_data(query: str) -> bool:
    """Execute an INSERT query to add a record."""
    conn = sqlite3.connect("demo.db")
    conn.execute(query)
    conn.commit()
    conn.close()
    return True

@mcp.tool()
def read_data(query: str = "SELECT * FROM people") -> list:
    """Execute a SELECT query and return all records."""
    conn = sqlite3.connect("demo.db")
    results = conn.execute(query).fetchall()
    conn.close()
    return results

if __name__ == "__main__":
    print("🚀 Starting server... ")
```

#2) Set Up LLM

We'll use a locally served Deepseek-R1 via Ollama as the LLM for our MCP-powered agent.

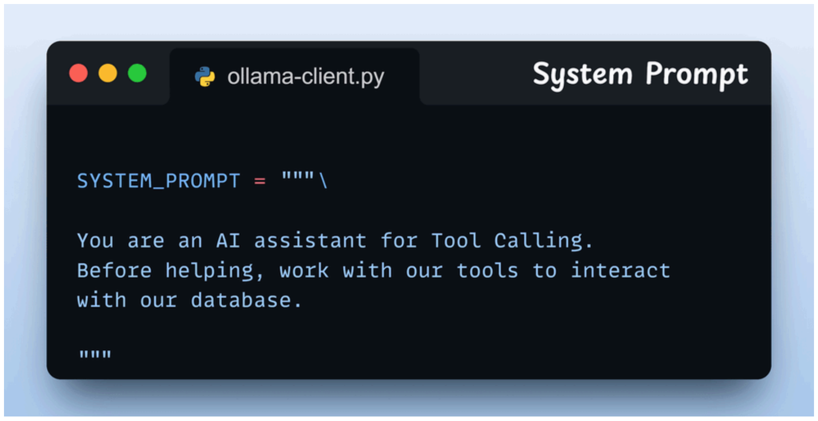


```python
from llama_index.llms.ollama import Ollama
from llama_index.core import Settings

llm = Ollama(model="deepseek-r1", request_timeout=120.0)
Settings.llm = llm
```

#3) Define system prompt

We define our agent's guiding instructions to use tools before answering user queries.

Feel free to tweak this on a need basis.

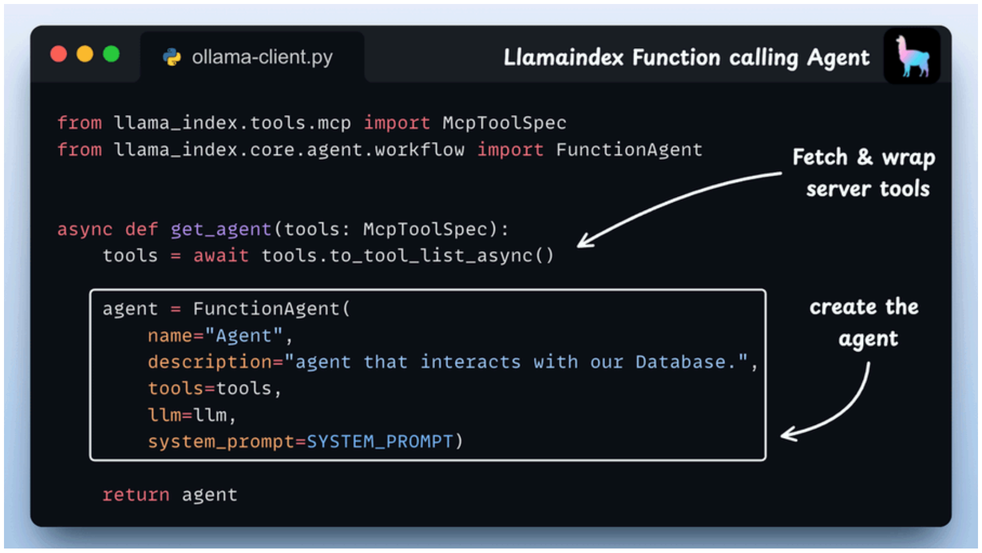


```python
SYSTEM_PROMPT = """\

You are an AI assistant for Tool Calling.
Before helping, work with our tools to interact
with our database.

"""
```

#4) Define the Agent

We define a function that builds a typical LlamaIndex agent with its appropriate arguments.

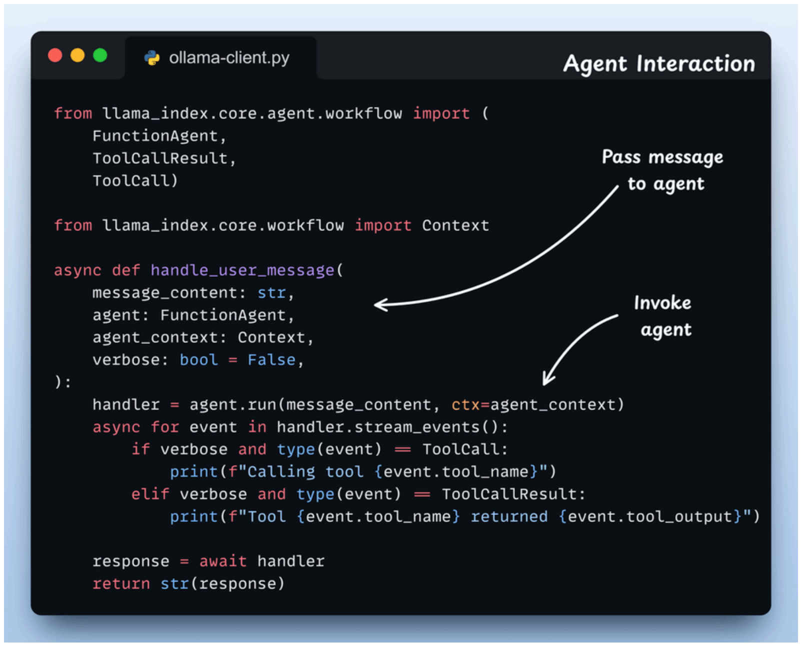
The tools passed to the agent are MCP tools, which llama_index wraps as native tools that can be easily used by our FunctionAgent.



```python
from llama_index.tools.mcp import McpToolSpec
from llama_index.core.agent.workflow import FunctionAgent


async def get_agent(tools: McpToolSpec):
    tools = await tools.to_tool_list_async()

    agent = FunctionAgent(
        name="Agent",
        description="agent that interacts with our Database.",
        tools=tools,
        llm=llm,
        system_prompt=SYSTEM_PROMPT)

    return agent
```

#5) Define Agent Interaction

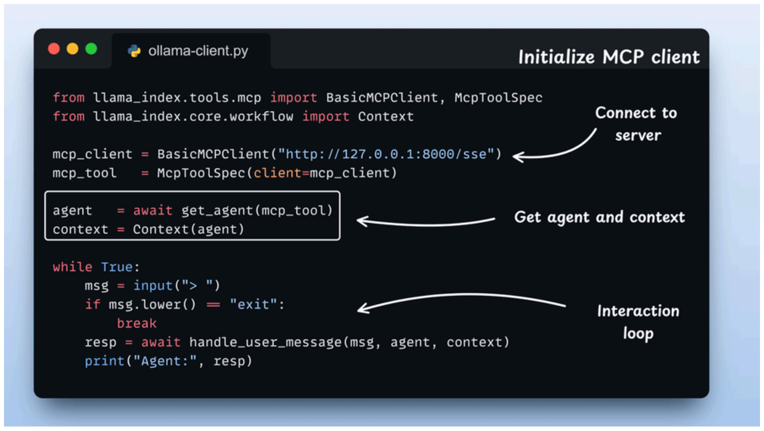
We pass user messages to our FunctionAgent with a shared Context for memory, stream tool calls and return its reply. We manage all the chat history and tool calls here.



```python
from llama_index.core.agent.workflow import (
    FunctionAgent,
    ToolCallResult,
    ToolCall)

from llama_index.core.workflow import Context

async def handle_user_message(
    message_content: str,
    agent: FunctionAgent,
    agent_context: Context,
    verbose: bool = False,
):
    handler = agent.run(message_content, ctx=agent_context)
    async for event in handler.stream_events():
        if verbose and type(event) == ToolCall:
            print(f"Calling tool {event.tool_name}")
        elif verbose and type(event) == ToolCallResult:
            print(f"Tool {event.tool_name} returned {event.tool_output}")

    response = await handler
    return str(response)
```

#6) Initialize MCP Client and the Agent

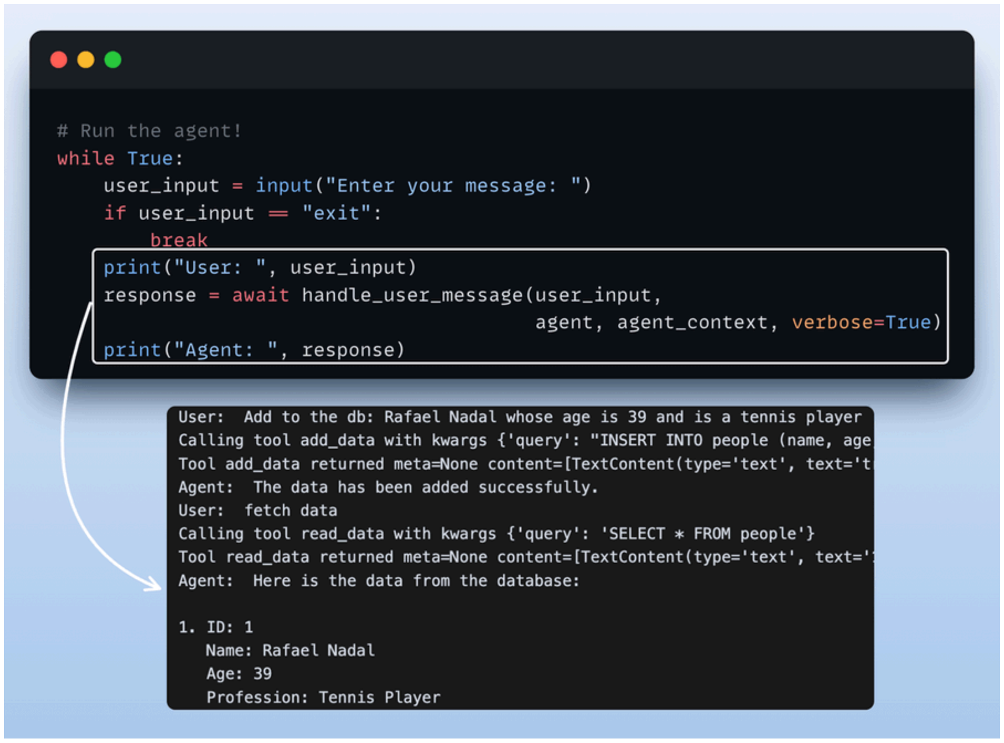
Launch the MCP client, load its tools, and wrap them as native tools for function-calling agents in LlamaIndex. Then, pass these tools to the agents and add the context manager.



```python
from llama_index.tools.mcp import BasicMCPClient, McpToolSpec
from llama_index.core.workflow import Context

mcp_client = BasicMCPClient("http://127.0.0.1:8000/sse")
mcp_tool   = McpToolSpec(client=mcp_client)

agent   = await get_agent(mcp_tool)
context = Context(agent)

while True:
    msg = input("> ")
    if msg.lower() == "exit":
        break
    resp = await handle_user_message(msg, agent, context)
    print("Agent:", resp)
```

#7) Run the Agent:

Finally, we start interacting with our agent and get access to the tools from our SQLite MCP server.



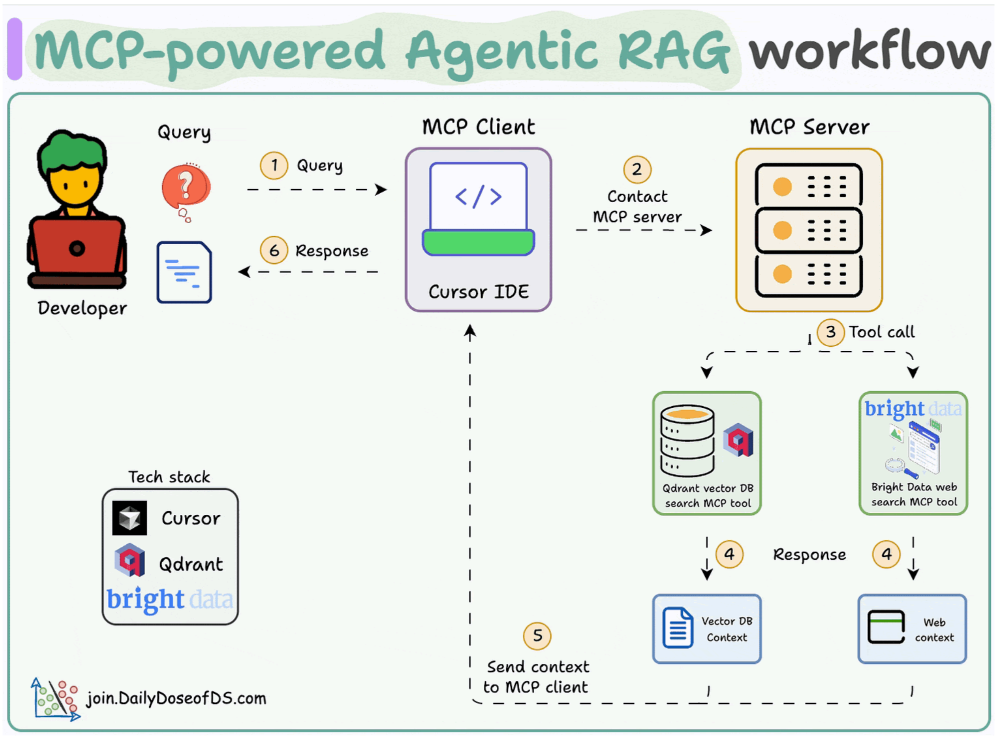
```python
# Run the agent!
while True:
    user_input = input("Enter your message: ")
    if user_input == "exit":
        break
    print("User: ", user_input)
    response = await handle_user_message(user_input,
                                         agent, agent_context, verbose=True)
    print("Agent: ", response)
```

```
User:  Add to the db: Rafael Nadal whose age is 39 and is a tennis player
Calling tool add_data with kwargs {'query': "INSERT INTO people (name, age
Tool add_data returned meta=None content=[TextContent(type='text', text='t
Agent:  The data has been added successfully.
User:  fetch data
Calling tool read_data with kwargs {'query': 'SELECT * FROM people'}
Tool read_data returned meta=None content=[TextContent(type='text', text='
Agent:  Here is the data from the database:

1. ID: 1
   Name: Rafael Nadal
   Age: 39
   Profession: Tennis Player
```

The code is available here: https://www.dailydoseofds.com/p/building-a-100-local-mcp-client/

#2) MCP-powered Agentic RAG

Learn how to create an MCP-powered Agentic RAG that searches a vector database and falls back to web search if needed.



Tech stack:

- Bright Data to scrape the web at scale.
- Qdrant as the vector DB.
- Cursor as the MCP client.

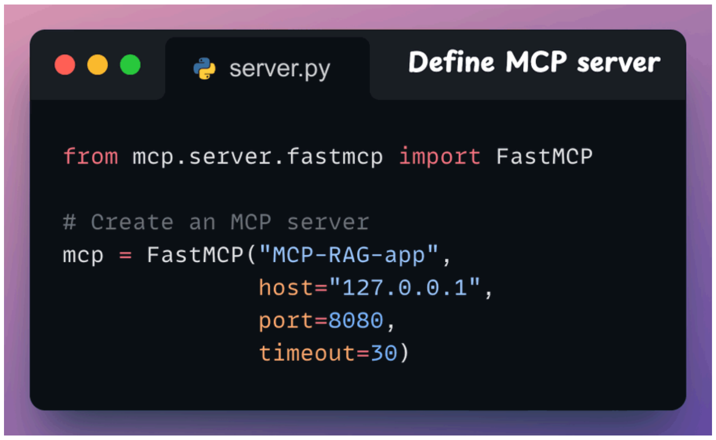
Workflow:

- The user inputs a query through the MCP client (Cursor).
- The client contacts the MCP server to select a relevant tool.
- The tool output is returned to the client to generate a response.

Let's implement this!

#1) Launch an MCP server

First, we define an MCP server with the host URL and port.



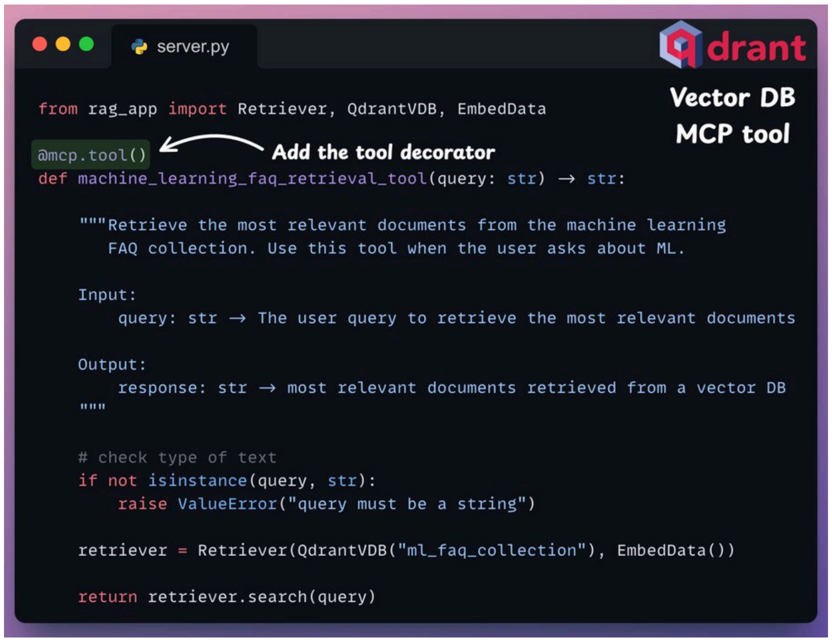
```python
from mcp.server.fastmcp import FastMCP

# Create an MCP server
mcp = FastMCP("MCP-RAG-app",
              host="127.0.0.1",
              port=8080,
              timeout=30)
```

#2) Vector DB MCP tool

A tool exposed through an MCP server has two requirements:

- It must be decorated with the "tool" decorator.
- It must have a clear docstring.

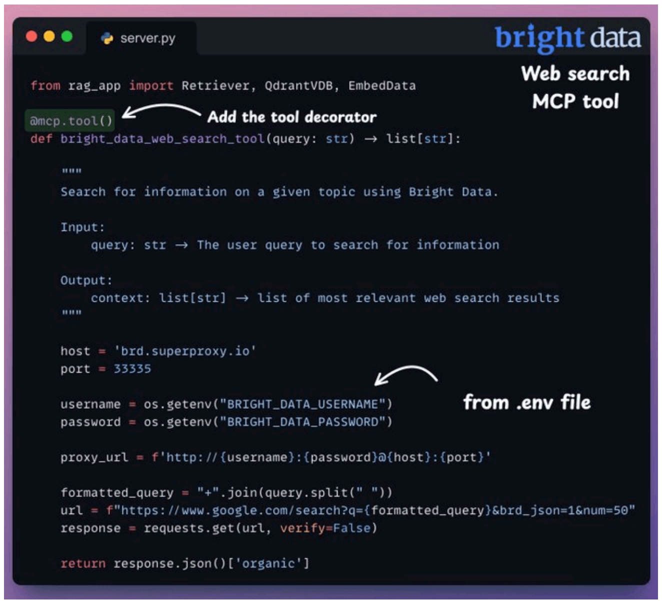
Below, we have an MCP tool to query a vector DB. It stores ML-related FAQs.



```python
from rag_app import Retriever, QdrantVDB, EmbedData

@mcp.tool()
def machine_learning_faq_retrieval_tool(query: str) → str:

    """Retrieve the most relevant documents from the machine learning
       FAQ collection. Use this tool when the user asks about ML.

    Input:
        query: str → The user query to retrieve the most relevant documents

    Output:
        response: str → most relevant documents retrieved from a vector DB
    """

    # check type of text
    if not isinstance(query, str):
        raise ValueError("query must be a string")

    retriever = Retriever(QdrantVDB("ml_faq_collection"), EmbedData())

    return retriever.search(query)
```

#3) Web search MCP tool

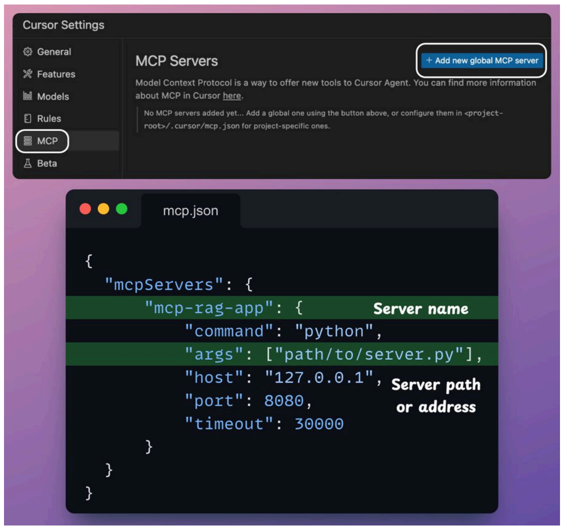
If query is unrelated to ML, we resort to web search using Bright Data's SERP API to scrape data at scale across several sources to get relevant context.



```python
from rag_app import Retriever, QdrantVDB, EmbedData

@mcp.tool()
def bright_data_web_search_tool(query: str) → list[str]:

    """
    Search for information on a given topic using Bright Data.

    Input:
        query: str → The user query to search for information

    Output:
        context: list[str] → list of most relevant web search results
    """

    host = 'brd.superproxy.io'
    port = 33335

    username = os.getenv("BRIGHT_DATA_USERNAME")
    password = os.getenv("BRIGHT_DATA_PASSWORD")

    proxy_url = f'http://{username}:{password}@{host}:{port}'

    formatted_query = "+".join(query.split(" "))
    url = f"https://www.google.com/search?q={formatted_query}&brd_json=1&num=50"
    response = requests.get(url, verify=False)

    return response.json()['organic']
```

#4) Integrate MCP server with Cursor

Go to Settings → MCP → Add new global MCP server. In the JSON file, add what's shown below



Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

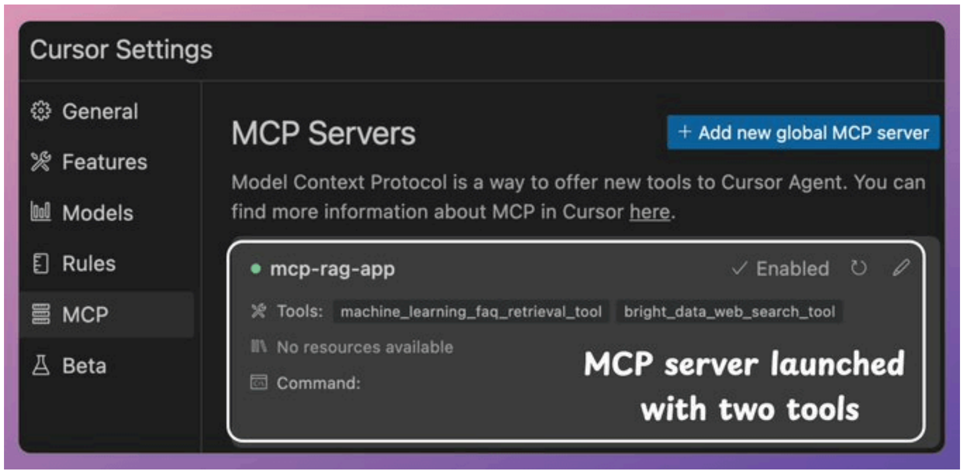
No MCP servers added yet... Add a global one using the button above, or configure them in <project-root>/.cursor/mcp.json for project-specific ones.

```json
{
  "mcpServers": {
      "mcp-rag-app": {
          "command": "python",
          "args": ["path/to/server.py"],
          "host": "127.0.0.1",
          "port": 8080,
          "timeout": 30000
      }
  }
}
```

Done!

Your local MCP server is live and connected to Cursor. It has two MCP tools:

- Bright Data web search tool to scrape data at scale.
- Vector DB search tool to query the relevant documents.


Cursor Settings
General
Features
Models
Rules
MCP
Beta
+ Add new global MCP server
MCP server launched with two tools

MCP Servers

Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

mcp-rag-app ✓ Enabled

Tools: machine_learning_faq_retrieval_tool bright_data_web_search_tool

No resources available

Command:

Next, we interact with the MCP server.

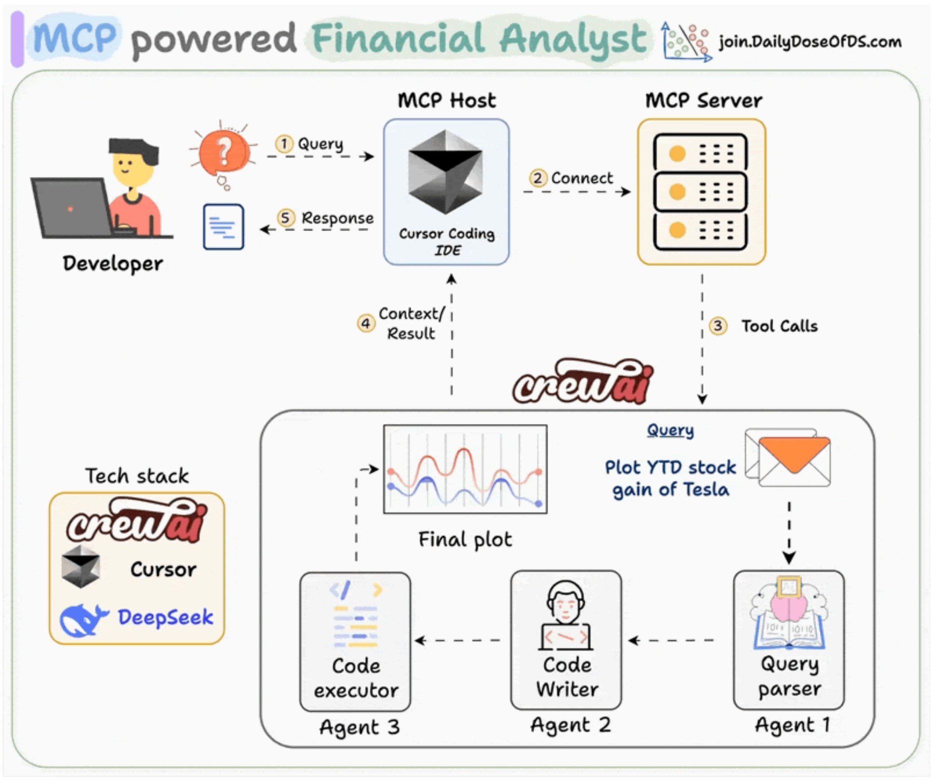
- When we ask an ML-related query, it invokes the vector DB tool.
- But when we ask a general query, it invokes the Bright Data web search tool to gather web data at scale from various sources.

That's Agentic behavior!

The code is available here: https://www.dailydoseofds.com/p/mcp-powered-agentic-rag/

#3) MCP-powered Financial Analyst

Build an MCP-powered AI agent that fetches, analyzes & generates insights on stock market trends, right from Cursor or Claude Desktop.



Tech stack:

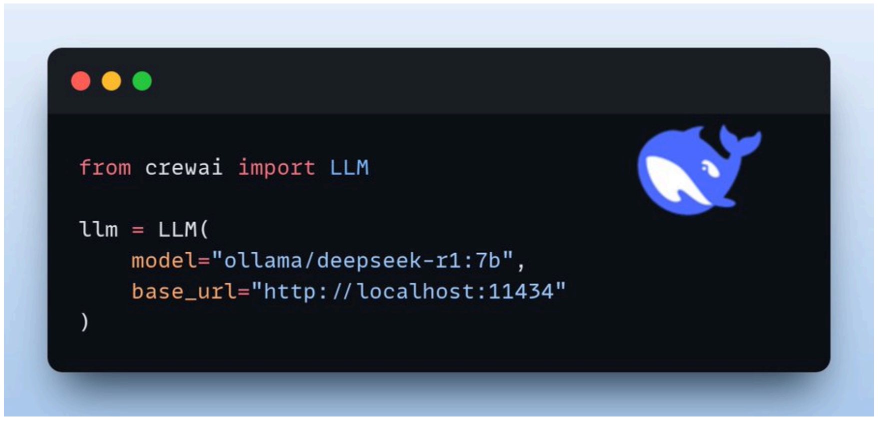
- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek-R1 LLM
- Cursor as the MCP host

Workflow:

- User submits a query.
- The MCP agent kicks off the financial analyst crew.
- The crew conducts research and creates an executable script.
- The agent runs the script to generate an analysis plot.

#1) Setup LLM

We will use Deepseek-R1 as the LLM, served locally using Ollama.

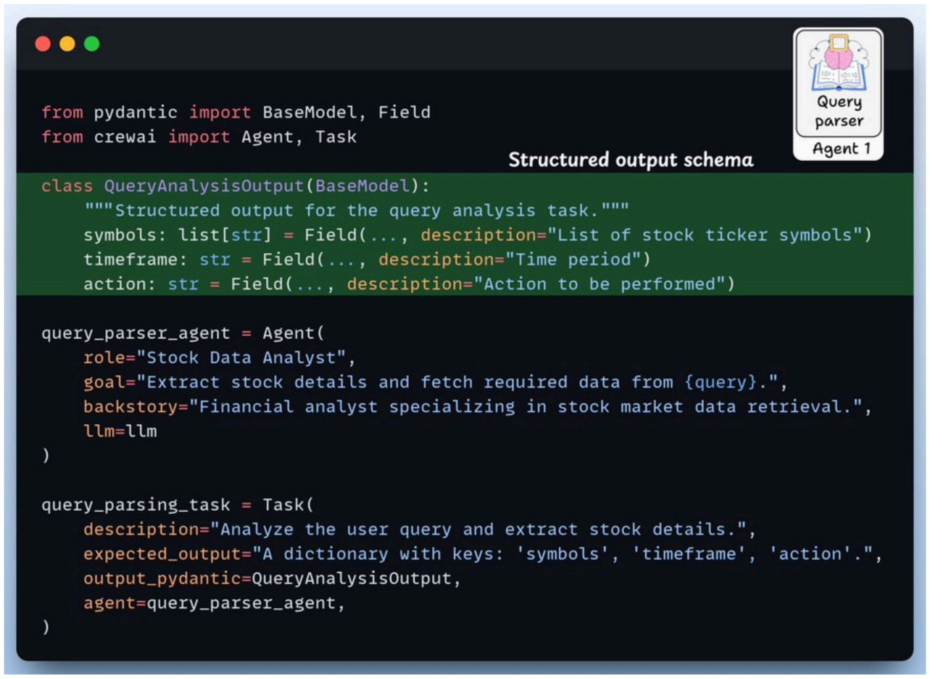


```python
from crewai import LLM

llm = LLM(
    model="ollama/deepseek-r1:7b",
    base_url="http://localhost:11434"
)
```

Let's setup the Crew now

#2) Query Parser Agent

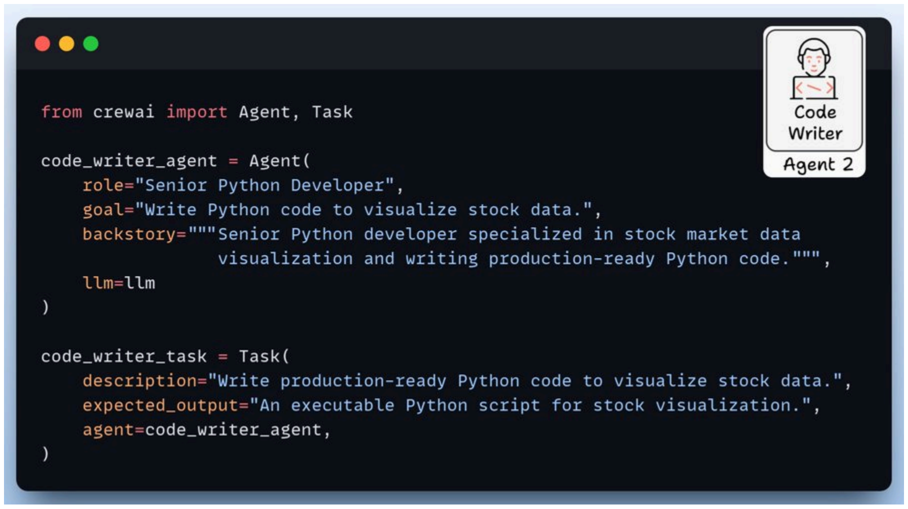
This agent accepts a natural language query and extracts structured output using Pydantic. This guarantees clean and structured inputs for further processing!



```python
from pydantic import BaseModel, Field
from crewai import Agent, Task

class QueryAnalysisOutput(BaseModel):
    """Structured output for the query analysis task."""
    symbols: list[str] = Field(..., description="List of stock ticker symbols")
    timeframe: str = Field(..., description="Time period")
    action: str = Field(..., description="Action to be performed")

query_parser_agent = Agent(
    role="Stock Data Analyst",
    goal="Extract stock details and fetch required data from {query}.",
    backstory="Financial analyst specializing in stock market data retrieval.",
    llm=llm
)

query_parsing_task = Task(
    description="Analyze the user query and extract stock details.",
    expected_output="A dictionary with keys: 'symbols', 'timeframe', 'action'.",
    output_pydantic=QueryAnalysisOutput,
    agent=query_parser_agent,
)
```

#3) Code Writer Agent

This agent writes Python code to visualize stock data using Pandas, Matplotlib, and Yahoo Finance libraries.

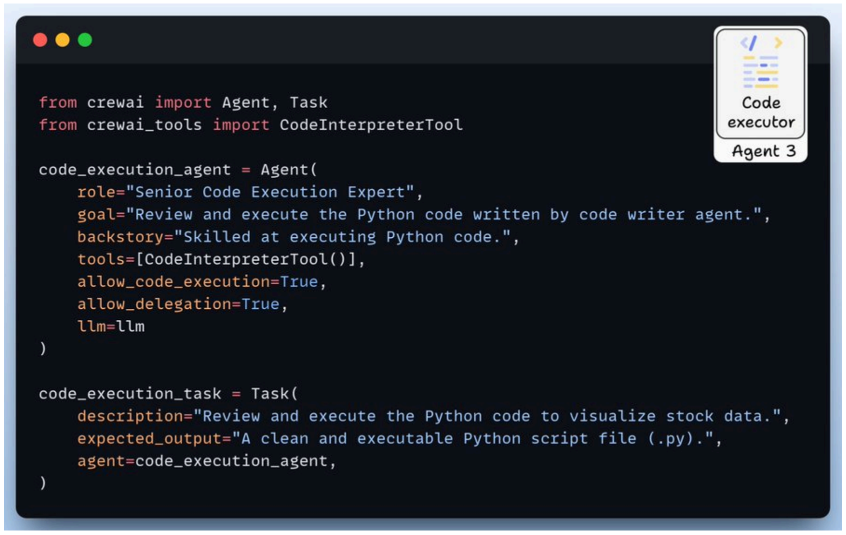


```python
from crewai import Agent, Task

code_writer_agent = Agent(
    role="Senior Python Developer",
    goal="Write Python code to visualize stock data.",
    backstory="""Senior Python developer specialized in stock market data
                 visualization and writing production-ready Python code.""",
    llm=llm
)

code_writer_task = Task(
    description="Write production-ready Python code to visualize stock data.",
    expected_output="An executable Python script for stock visualization.",
    agent=code_writer_agent,
)
```

#4) Code Executor Agent

This agent reviews and executes the generated Python code for stock data visualization.

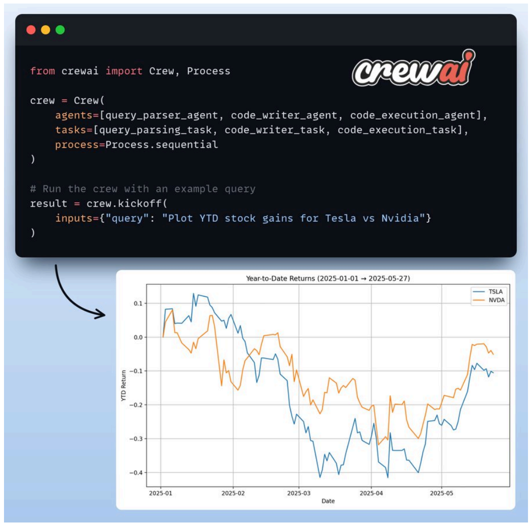
It uses the code interpreter tool by CrewAI to execute the code in a secure sandbox environment.



```python
from crewai import Agent, Task
from crewai_tools import CodeInterpreterTool

code_execution_agent = Agent(
    role="Senior Code Execution Expert",
    goal="Review and execute the Python code written by code writer agent.",
    backstory="Skilled at executing Python code.",
    tools=[CodeInterpreterTool()],
    allow_code_execution=True,
    allow_delegation=True,
    llm=llm
)

code_execution_task = Task(
    description="Review and execute the Python code to visualize stock data.",
    expected_output="A clean and executable Python script file (.py).",
    agent=code_execution_agent,
)
```

#5) Setup Crew and Kickoff

We set up and kick off our financial analysis crew to get the result shown below!

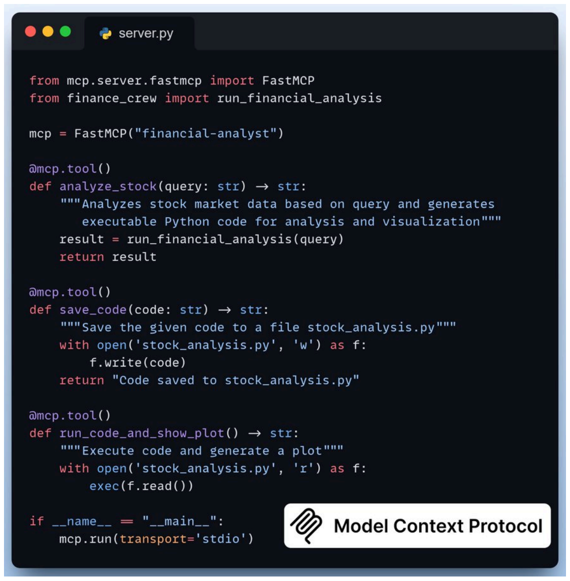


```python
from crewai import Crew, Process

crew = Crew(
    agents=[query_parser_agent, code_writer_agent, code_execution_agent],
    tasks=[query_parsing_task, code_writer_task, code_execution_task],
    process=Process.sequential
)

# Run the crew with an example query
result = crew.kickoff(
    inputs={"query": "Plot YTD stock gains for Tesla vs Nvidia"}
)
```

#6) Create MCP Server

Now, we encapsulate our financial analyst within an MCP tool and add two more tools to enhance the user experience.

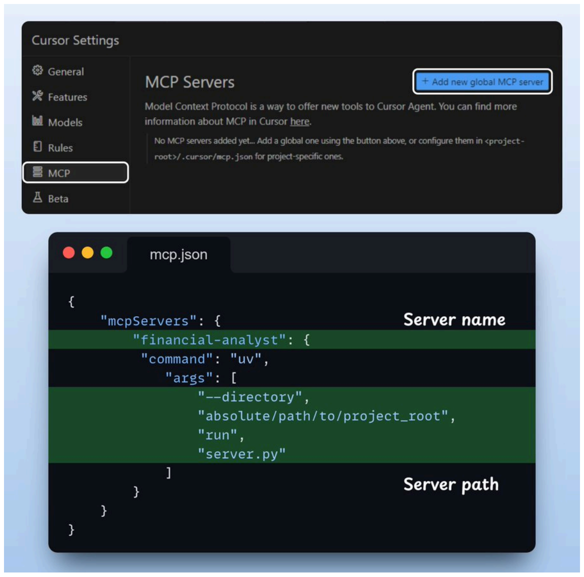
- save_code -> Saves generated code to local directory
- run_code_and_show_plot -> Executes the code and generates a plot



```python
from mcp.server.fastmcp import FastMCP
from finance_crew import run_financial_analysis

mcp = FastMCP("financial-analyst")

@mcp.tool()
def analyze_stock(query: str) → str:
    """Analyzes stock market data based on query and generates
       executable Python code for analysis and visualization"""
    result = run_financial_analysis(query)
    return result

@mcp.tool()
def save_code(code: str) → str:
    """Save the given code to a file stock_analysis.py"""
    with open('stock_analysis.py', 'w') as f:
        f.write(code)
    return "Code saved to stock_analysis.py"

@mcp.tool()
def run_code_and_show_plot() → str:
    """Execute code and generate a plot"""
    with open('stock_analysis.py', 'r') as f:
        exec(f.read())

if __name__ == "__main__":
    mcp.run(transport='stdio')
```

#7) Integrate MCP server with Cursor

Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server. In the JSON file, add what's shown below

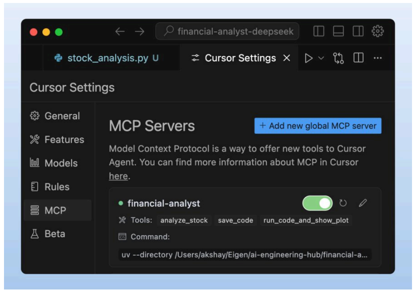


Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

No MCP servers added yet... Add a global one using the button above, or configure them in `<project-root>/.cursor/mcp.json` for project-specific ones.

```json
{
    "mcpServers": {
        "financial-analyst": {
          "command": "uv",
            "args": [
                "--directory",
                "absolute/path/to/project_root",
                "run",
                "server.py"
            ]
        }
    }
}
```

Done! Our financial analyst MCP server is live and connected to Cursor.



Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

● financial-analyst

Tools: analyze_stock save_code run_code_and_show_plot

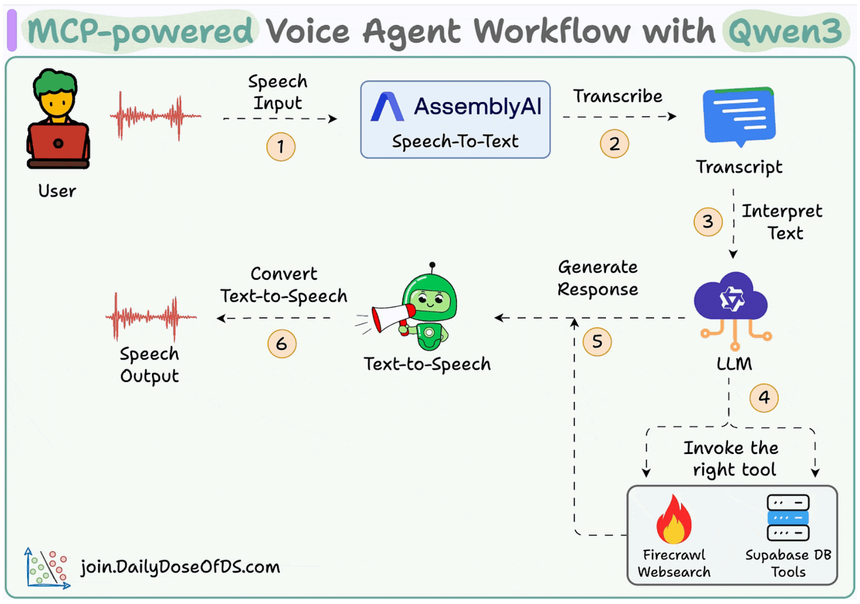
Command:

```
uv --directory /Users/akshay/Eigen/ai-engineering-hub/financial-a...
```

The code is available here: https://www.dailydoseofds.com/p/hands-on-building-an-mcp-powered-financial-analyst/

#4) MCP-powered Voice Agent

This project teaches you how to build an MCP-driven voice Agent that queries a database and falls back to web search if needed.



Tech Stack

- AssemblyAI for Speech-to-Text.
- Firecrawl for web search.
- Supabase for a database.
- Livekit for orchestration.
- Qwen3 as the LLM.

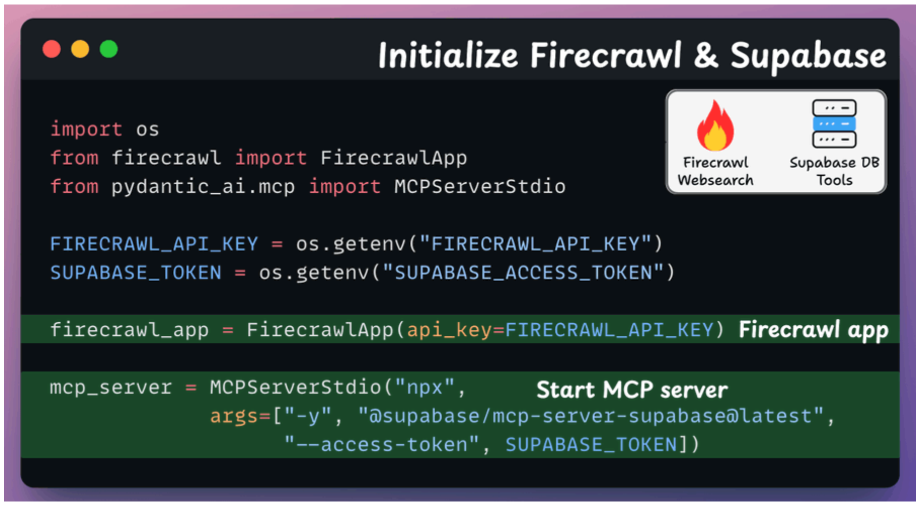
Workflow:

- User's speech query is transcribed to text with AssemblyAI.
- Agent discovers DB & web tools.
- LLM invokes the right tool, fetches data & generates a response.
- The app delivers the response via text-to-speech.

Let's implement this!

#1) Initialize Firecrawl & Supabase

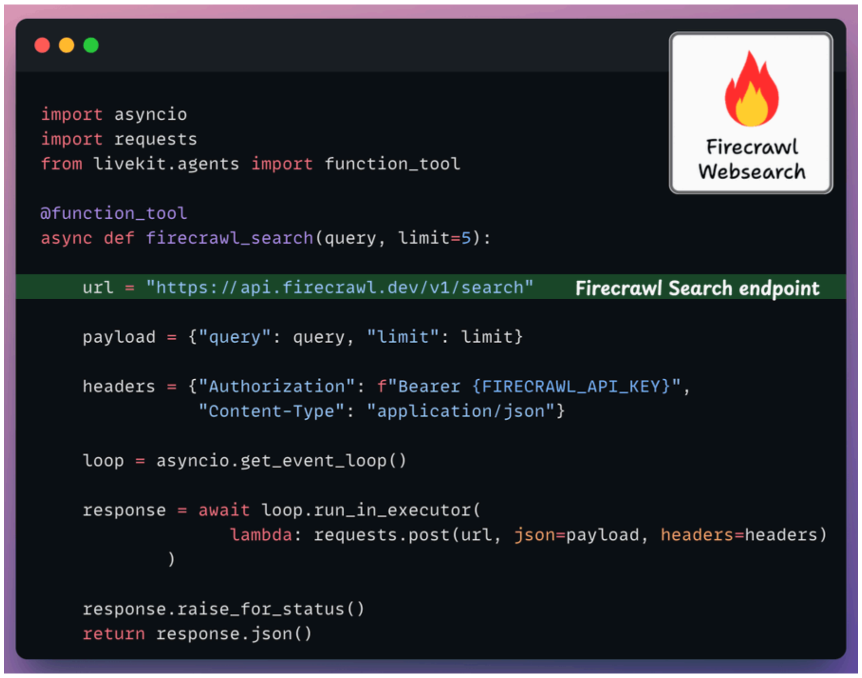
We instantiate Firecrawl to enable web searches and start our MCP server to expose Supabase tools to our Agent.



```python
import os
from firecrawl import FirecrawlApp
from pydantic_ai.mcp import MCPServerStdio

FIRECRAWL_API_KEY = os.getenv("FIRECRAWL_API_KEY")
SUPABASE_TOKEN = os.getenv("SUPABASE_ACCESS_TOKEN")

firecrawl_app = FirecrawlApp(api_key=FIRECRAWL_API_KEY)

mcp_server = MCPServerStdio("npx",
                  args=["-y", "@supabase/mcp-server-supabase@latest",
                        "--access-token", SUPABASE_TOKEN])
```

#2) Define web search tool

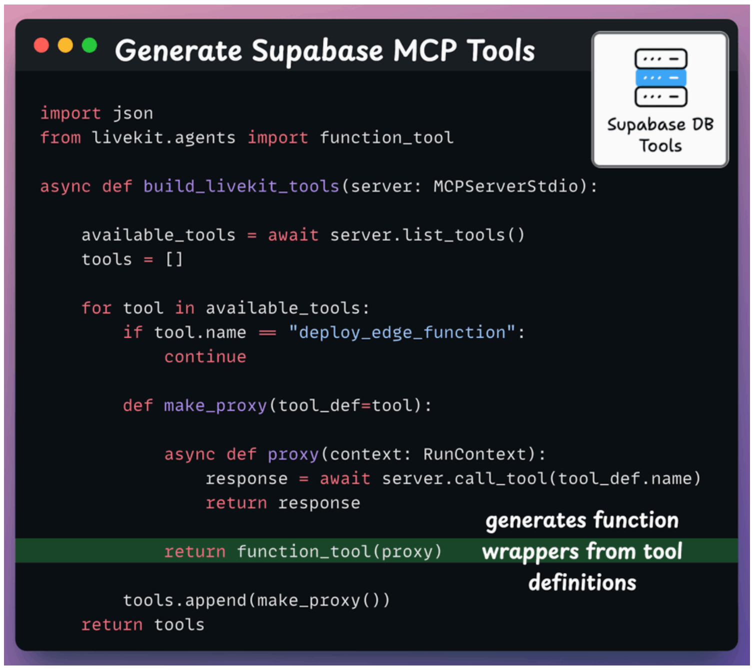
We fetch live web search results using Firecrawl search endpoint. This gives our agent up-to-date online information.



```python
import asyncio
import requests
from livekit.agents import function_tool

@function_tool
async def firecrawl_search(query, limit=5):

    url = "https://api.firecrawl.dev/v1/search"

    payload = {"query": query, "limit": limit}

    headers = {"Authorization": f"Bearer {FIRECRAWL_API_KEY}",
               "Content-Type": "application/json"}

    loop = asyncio.get_event_loop()

    response = await loop.run_in_executor(
                    lambda: requests.post(url, json=payload, headers=headers)
            )

    response.raise_for_status()
    return response.json()
```

#3) Get Supabase MCP Tools

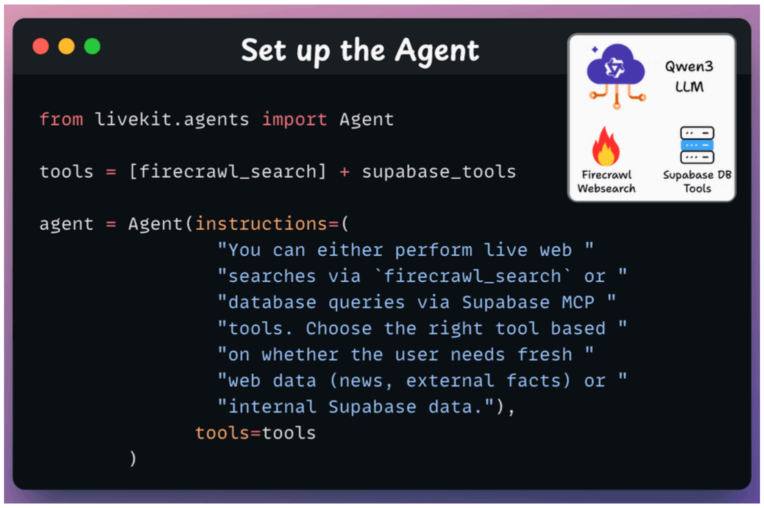
We list our Supabase tools via the MCP server and wrap each of them as LiveKit tools for our Agent.


Generate Supabase MCP Tools
Supabase DB Tools
generates function wrappers from tool definitions

```python
import json
from livekit.agents import function_tool

async def build_livekit_tools(server: MCPServerStdio):

    available_tools = await server.list_tools()
    tools = []

    for tool in available_tools:
        if tool.name == "deploy_edge_function":
            continue

        def make_proxy(tool_def=tool):

            async def proxy(context: RunContext):
                response = await server.call_tool(tool_def.name)
                return response

            return function_tool(proxy)

        tools.append(make_proxy())
    return tools
```

#4) Build the Agent

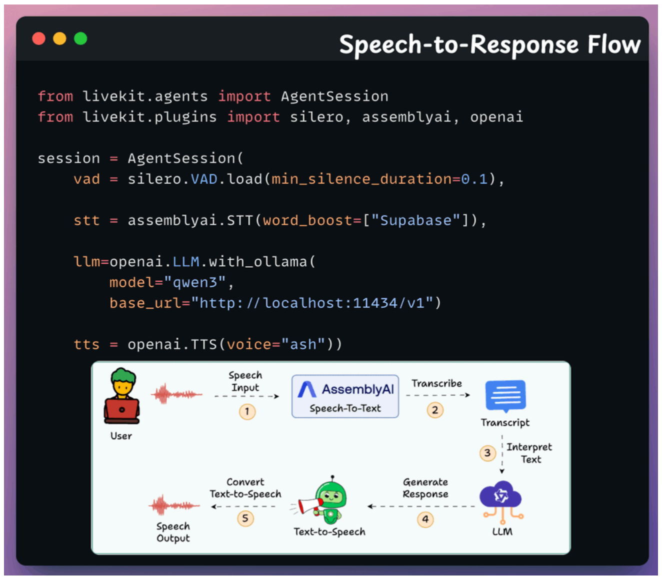
We set up our Agent with instructions on how to handle user queries. We also give it access to the Firecrawl web search and Supabase tools defined earlier.


Set up the Agent

```python
from livekit.agents import Agent

tools = [firecrawl_search] + supabase_tools

agent = Agent(instructions=(
                "You can either perform live web "
                "searches via `firecrawl_search` or "
                "database queries via Supabase MCP "
                "tools. Choose the right tool based "
                "on whether the user needs fresh "
                "web data (news, external facts) or "
                "internal Supabase data."),
            tools=tools
        )
```

#5) Configure Speech-to-Response flow

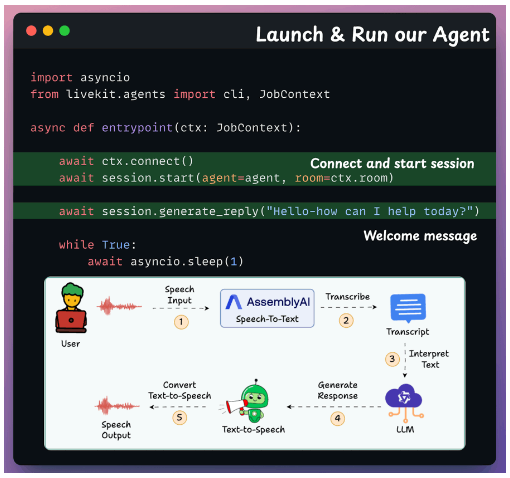
- We transcribe user speech with AssemblyAI Speech-to-Text.
- Qwen 3 LLM, served locally with Ollama, invokes the right tool.
- A voice output is generated via TTS.


Speech-to-Response Flow

```python
from livekit.agents import AgentSession
from livekit.plugins import silero, assemblyai, openai

session = AgentSession(
    vad = silero.VAD.load(min_silence_duration=0.1),

    stt = assemblyai.STT(word_boost=["Supabase"]),

    llm=openai.LLM.with_ollama(
        model="qwen3",
        base_url="http://localhost:11434/v1")

    tts = openai.TTS(voice="ash"))
```

#6) Launch the Agent

We connect to LiveKit and start our session with a greeting. Then continuously listen and respond until the user stops.



```python
import asyncio
from livekit.agents import cli, JobContext

async def entrypoint(ctx: JobContext):

    await ctx.connect()
    await session.start(agent=agent, room=ctx.room)

    await session.generate_reply("Hello-how can I help today?")

    while True:
        await asyncio.sleep(1)
```

Done!

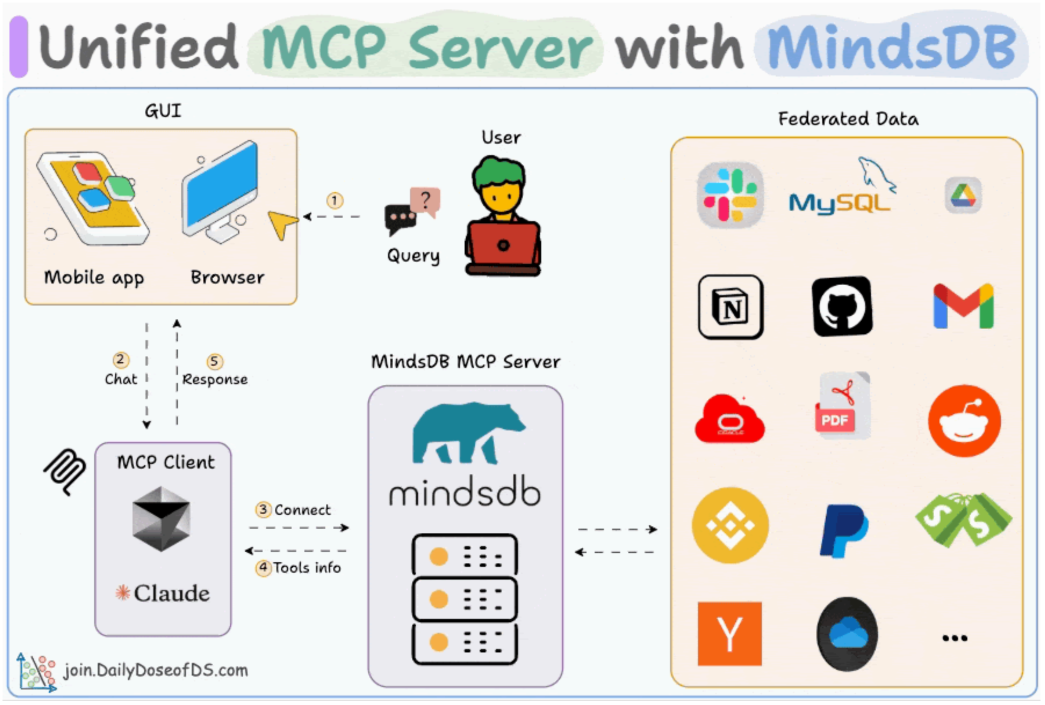
Our MCP-powered Voice Agent is ready.

- If the query is related to a database, it queries Supabase via MCP tools.
- Otherwise, it performs a web search via Firecrawl.

The code is available here:
https://www.dailydoseofds.com/p/an-mcp-powered-voice-agent/

#5) A Unified MCP server

This project builds an MCP server to query and chat with over 200+ data sources using natural language through a unified interface powered by MindsDB and Cursor IDE.



Tech stack

- MindsDB to power our unified MCP server
- Cursor as the MCP host
- Docker to self-host the server

Workflow

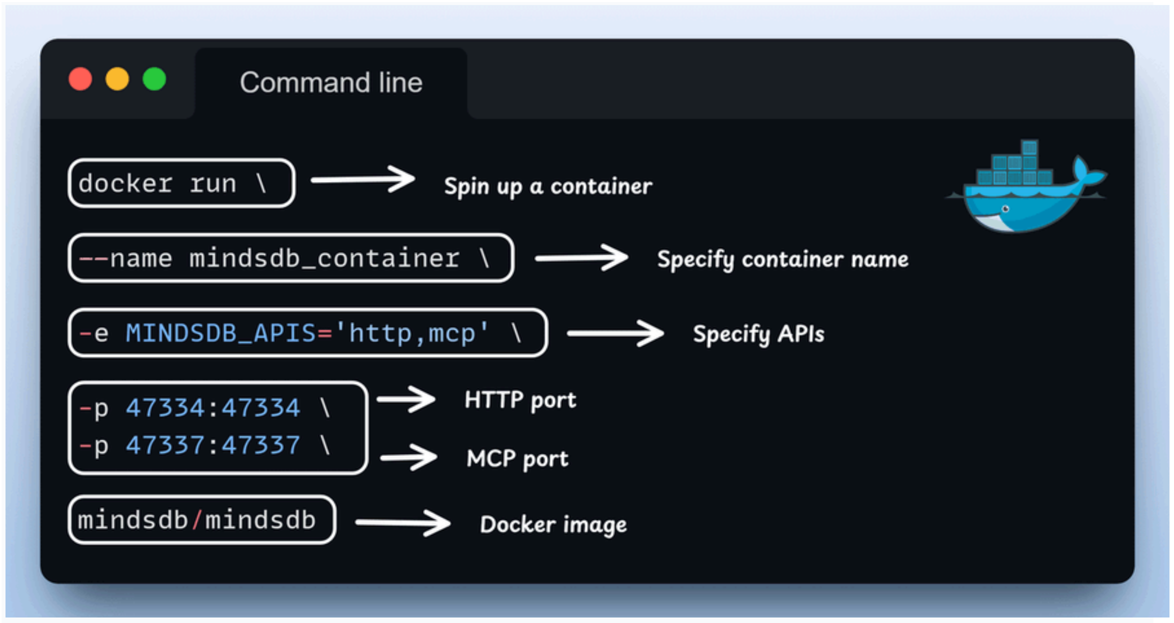
- User submits a query
- Agent connects to the MindsDB MCP server to find tools
- Selects the appropriate tool based on the user query and calls it
- Finally, returns a contextually relevant response

Let's implement this!

#1) Docker Setup

MindsDB provides Docker images that can be run in Docker containers.

Install MindsDB locally using the Docker image by running the command in your terminal.



```
docker run \
--name mindsdb_container \
-e MINDSDB_APIS='http,mcp' \
-p 47334:47334 \
-p 47337:47337 \
mindsdb/mindsdb
```

#2) Start MindsDB GUI

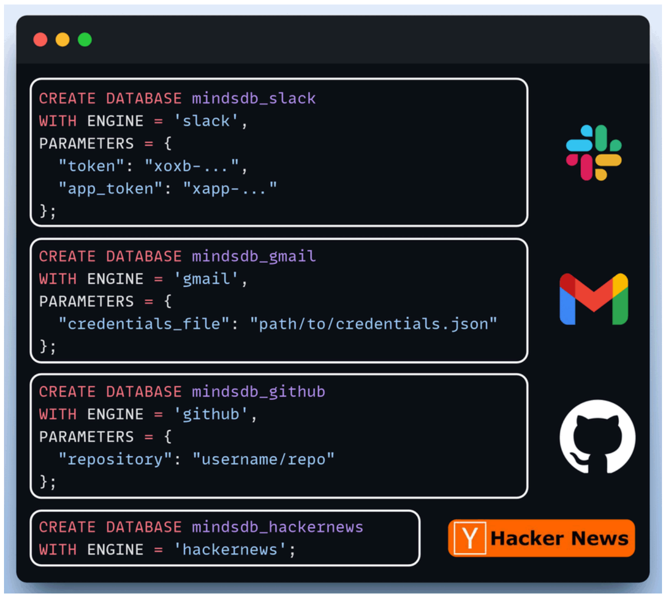
After installing the Docker image, go to 127.0.0.1:47334 in your browser to access the MindsDB editor.

Through this interface, you can connect to over 200 data sources and run SQL queries against them.

#3) Integrate Data Sources

Let's start building our federated query engine by connecting our data sources to MindsDB.

We use Slack, Gmail, GitHub and Hacker News as our federated data sources.



```
CREATE DATABASE mindsdb_slack
WITH ENGINE = 'slack',
PARAMETERS = {
  "token": "xoxb-...",
  "app_token": "xapp-..."
};
```

```
CREATE DATABASE mindsdb_gmail
WITH ENGINE = 'gmail',
PARAMETERS = {
  "credentials_file": "path/to/credentials.json"
};
```

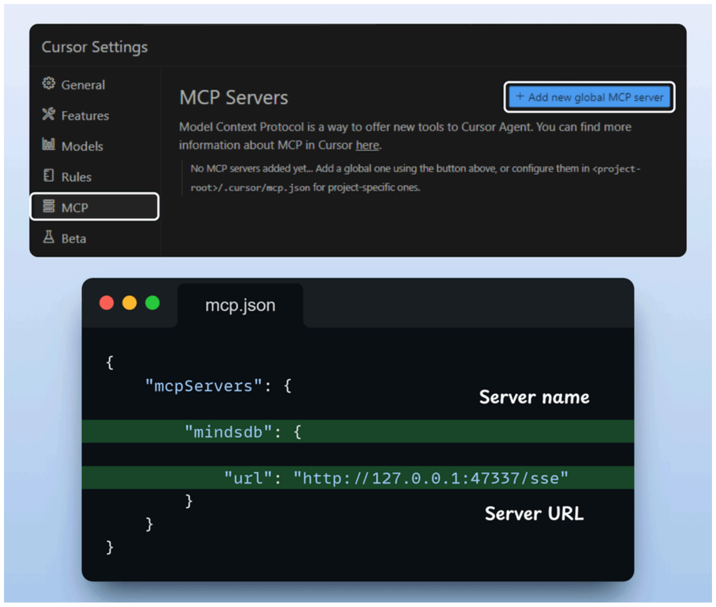
```
CREATE DATABASE mindsdb_github
WITH ENGINE = 'github',
PARAMETERS = {
  "repository": "username/repo"
};
```

```
CREATE DATABASE mindsdb_hackernews
WITH ENGINE = 'hackernews';
```

#4) Integrate MCP Server with Cursor

After building the federated query engine, let's unify our data sources by connecting them to MindsDB's MCP server.

Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server. In the JSON file, add the following



Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

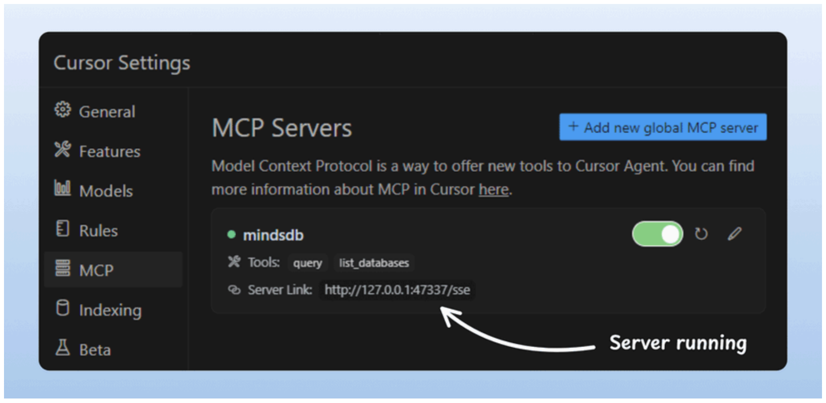
No MCP servers added yet... Add a global one using the button above, or configure them in `<project-root>/.cursor/mcp.json` for project-specific ones.

```
{
    "mcpServers": {
        "mindsdb": {
            "url": "http://127.0.0.1:47337/sse"
        }
    }
}
```

Done! Our MindsDB MCP server is live and connected to Cursor!

The MCP server offers two tools:

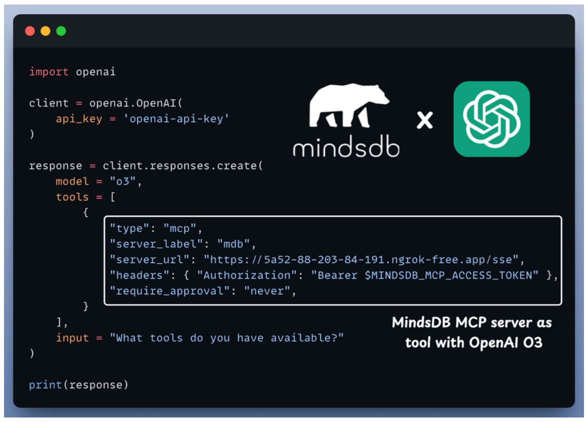
- list_databases: Lists all data sources connected to MindsDB.
- query: Answers user queries on the federated data.



Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

mindsdb
Tools: query list_databases
Server Link: http://127.0.0.1:47337/sse

Apart from Claude and Cursor, MindsDB MCP server also works with the new OpenAI MCP integration.

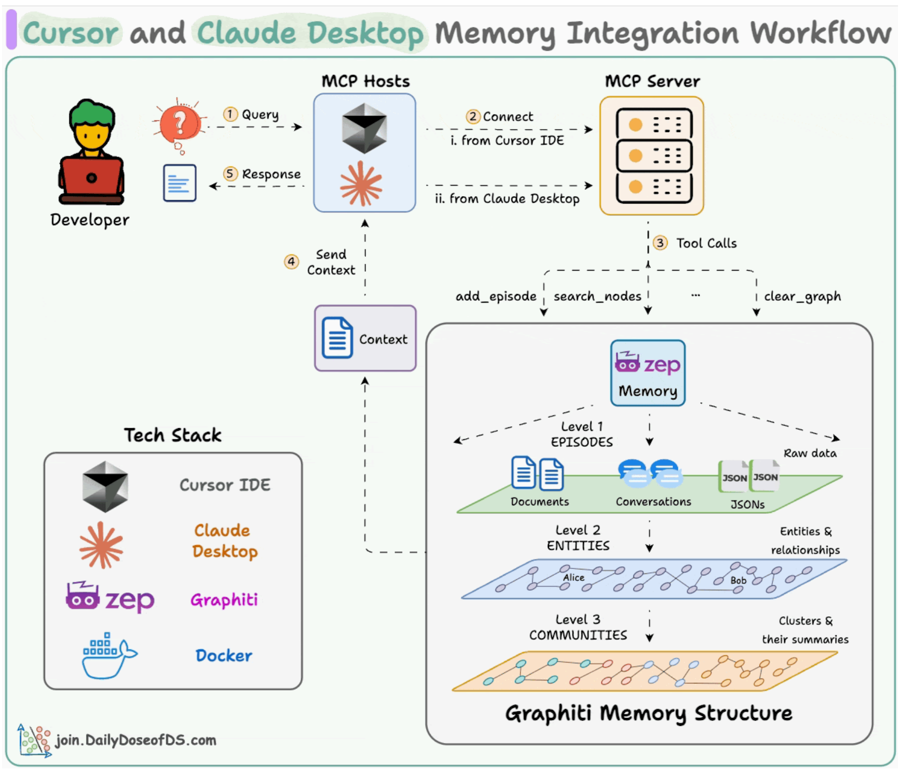


```python
import openai

client = openai.OpenAI(
    api_key = 'openai-api-key'
)

response = client.responses.create(
    model = "o3",
    tools = [
        {
            "type": "mcp",
            "server_label": "mdb",
            "server_url": "https://5a52-88-203-84-191.ngrok-free.app/sse",
            "headers": { "Authorization": "Bearer $MINDSDB_MCP_ACCESS_TOKEN" },
            "require_approval": "never",
        }
    ],
    input = "What tools do you have available?"
)

print(response)
```

The code is available here: https://www.dailydoseofds.com/p/build-an-mcp-server-to-connect-to-200-data-sources/

#6) MCP-powered shared memory for Claude Desktop and Cursor

Devs use Claude Desktop and Cursor independently with no context sharing. Learn how to add a common memory layer to cross-operate without losing context.



Tech Stack

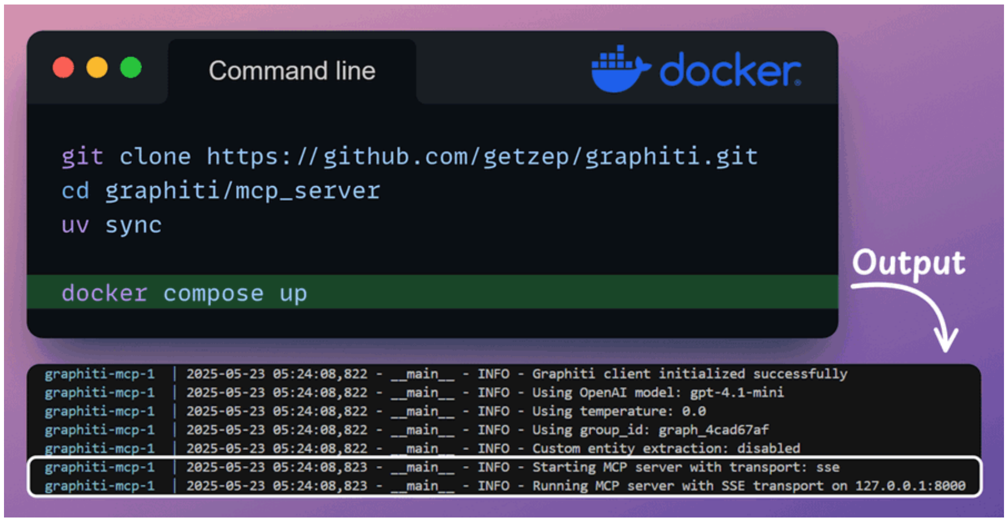
- Zep's Graphiti MCP as a memory layer for AI Agents.
- Cursor and Claude as the MCP hosts.

Workflow

- User submits a query to Cursor & Claude.
- Facts/Info are stored in a common memory layer using Graphiti MCP.
- Memory is queried if context is required in any interaction.
- Graphiti shares memory across multiple hosts.

#1) Docker Setup

Deploy the Graphiti MCP server using Docker Compose. This setup starts the MCP server with Server-Sent Events (SSE) transport.

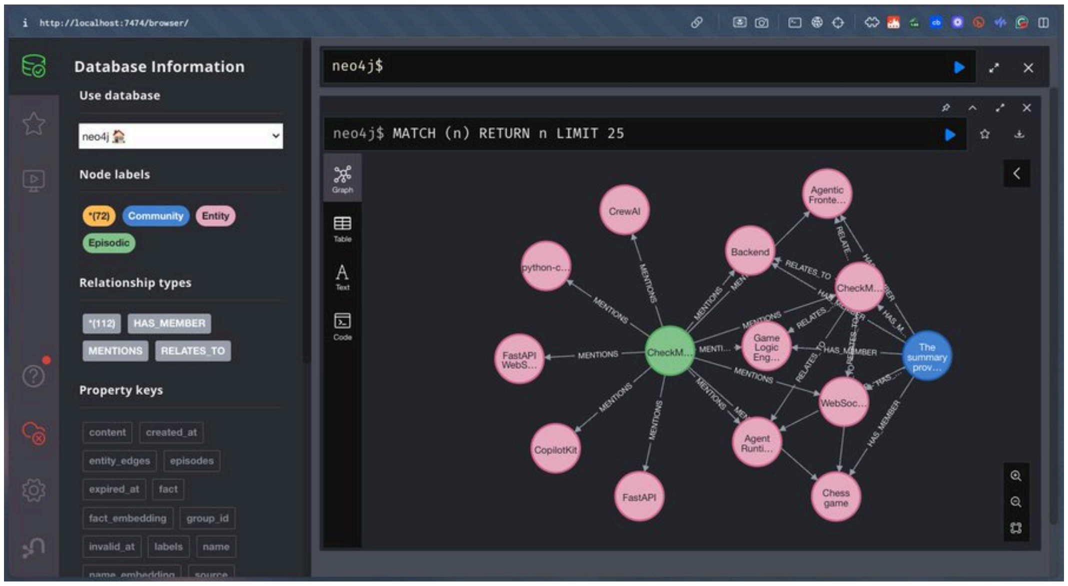


```
git clone https://github.com/getzep/graphiti.git
cd graphiti/mcp_server
uv sync

docker compose up
```

```
graphiti-mcp-1  | 2025-05-23 05:24:08,822 - __main__ - INFO - Graphiti client initialized successfully
graphiti-mcp-1  | 2025-05-23 05:24:08,822 - __main__ - INFO - Using OpenAI model: gpt-4.1-mini
graphiti-mcp-1  | 2025-05-23 05:24:08,822 - __main__ - INFO - Using temperature: 0.0
graphiti-mcp-1  | 2025-05-23 05:24:08,822 - __main__ - INFO - Using group_id: graph_4cad67af
graphiti-mcp-1  | 2025-05-23 05:24:08,822 - __main__ - INFO - Custom entity extraction: disabled
graphiti-mcp-1  | 2025-05-23 05:24:08,823 - __main__ - INFO - Starting MCP server with transport: sse
graphiti-mcp-1  | 2025-05-23 05:24:08,823 - __main__ - INFO - Running MCP server with SSE transport on 127.0.0.1:8000
```

The Docker setup above includes a Neo4j container, which launches the database as a local instance.

This configuration lets you query and visualize the knowledge graph using the Neo4j browser preview.

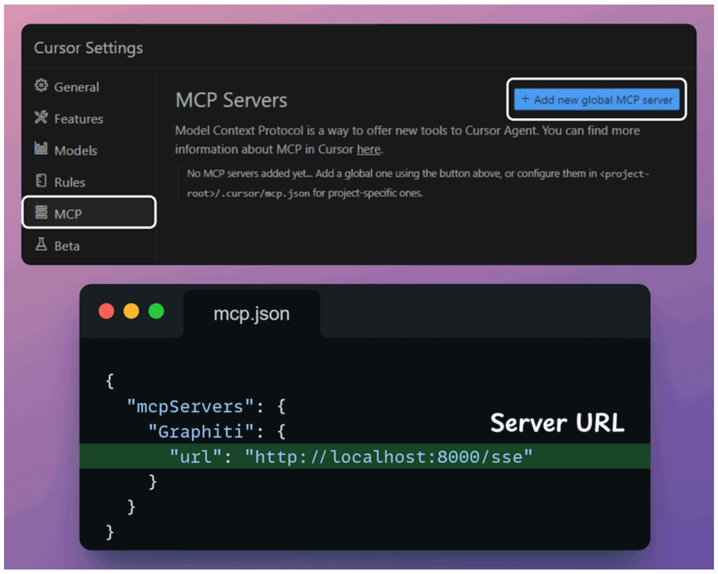


```
neo4j$
neo4j$ MATCH (n) RETURN n LIMIT 25
```

#2) Connect MCP server to Cursor

With tools and our server ready, let's integrate it with our Cursor IDE!

Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server. In the JSON file, add what's shown below



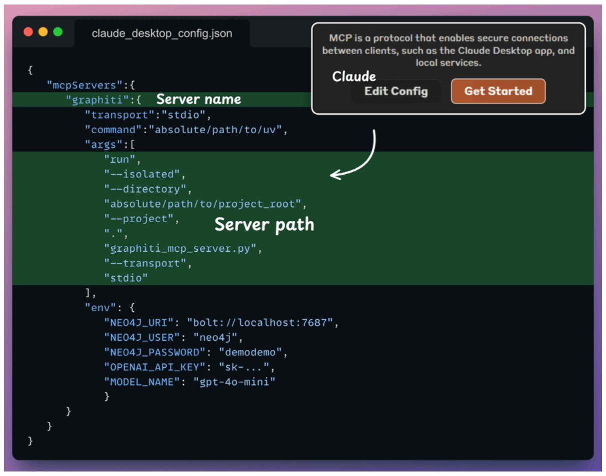
Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

No MCP servers added yet... Add a global one using the button above, or configure them in `<project-root>/.cursor/mcp.json` for project-specific ones.

```
{
  "mcpServers": {
    "Graphiti": {
      "url": "http://localhost:8000/sse"
    }
  }
}
```

#3) Connect MCP server with Claude

Go to File → Settings → Developer → Edit Config, add what's shown below

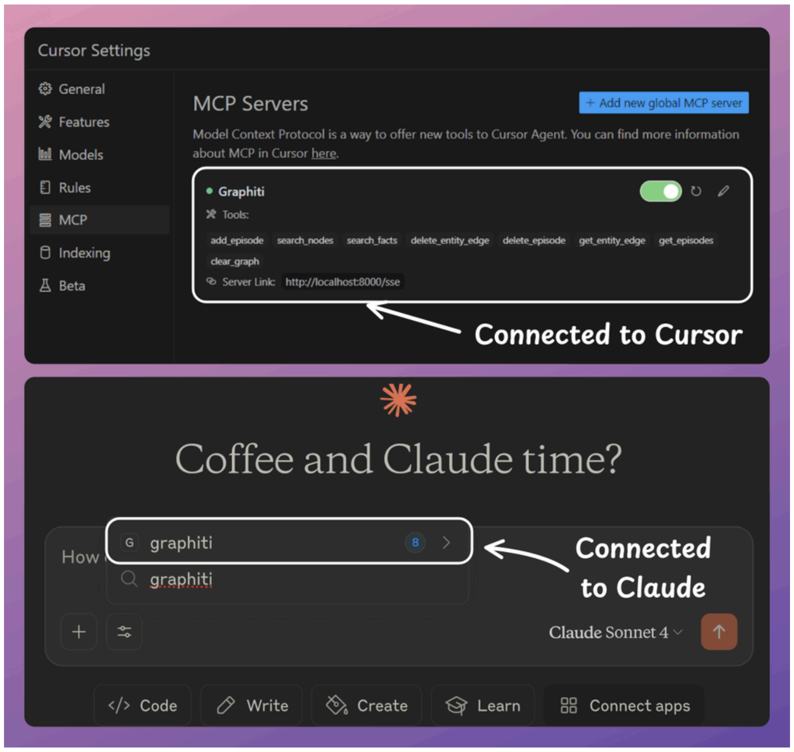


MCP is a protocol that enables secure connections between clients, such as the Claude Desktop app, and local services.

```
{
   "mcpServers":{
      "graphiti":{
         "transport":"stdio",
         "command":"absolute/path/to/uv",
         "args":[
            "run",
            "--isolated",
            "--directory",
            "absolute/path/to/project_root",
            "--project",
            ".",
            "graphiti_mcp_server.py",
            "--transport",
            "stdio"
         ],
         "env": {
            "NEO4J_URI": "bolt://localhost:7687",
            "NEO4J_USER": "neo4j",
            "NEO4J_PASSWORD": "demodemo",
            "OPENAI_API_KEY": "sk-...",
            "MODEL_NAME": "gpt-4o-mini"
            }
      }
   }
}
```

Done!

Our Graphiti MCP server is live and connected to Cursor & Claude!


Cursor Settings
General
Features
Models
Rules
MCP
Indexing
Beta
MCP Servers
+ Add new global MCP server
Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.
Graphiti
Tools:
add_episode search_nodes search_facts delete_entity_edge delete_episode get_entity_edge get_episodes clear_graph
Server Link: http://localhost:8000/sse
Connected to Cursor
Coffee and Claude time?
How
graphiti
graphiti
Connected to Claude
Claude Sonnet 4
Code
Write
Create
Learn
Connect apps

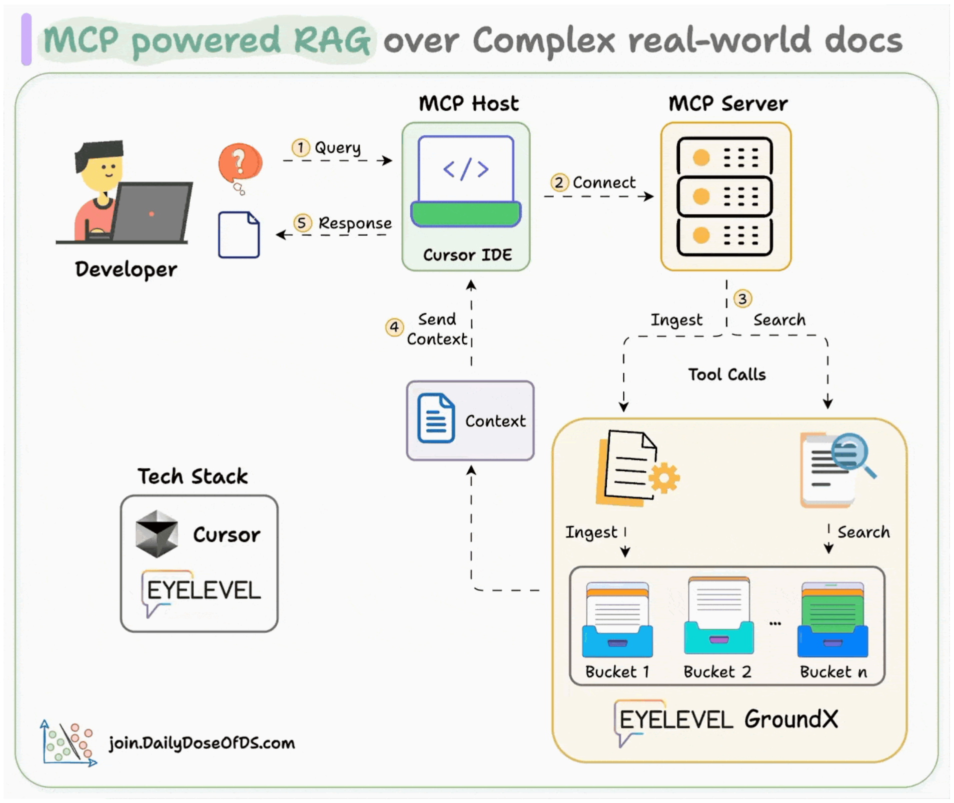
Now you can chat with Claude Desktop, share facts/info, store the response in memory, and retrieve them from Cursor, and vice versa.

This way, you can pipe Claude's insights straight into Cursor, all via a single MCP.

The code is available here: https://www.dailydoseofds.com/p/build-a-shared-memory-for-claude-desktop-and-cursor/

#7) MCP-powered RAG over complex docs

Learn how to use MCP to power an RAG app over complex documents with tables, charts, images, complex layouts, and whatnot.



Tech Stack

- Cursor as the MCP client
- EyelevelAI's GroundX to build an MCP server that can process complex docs

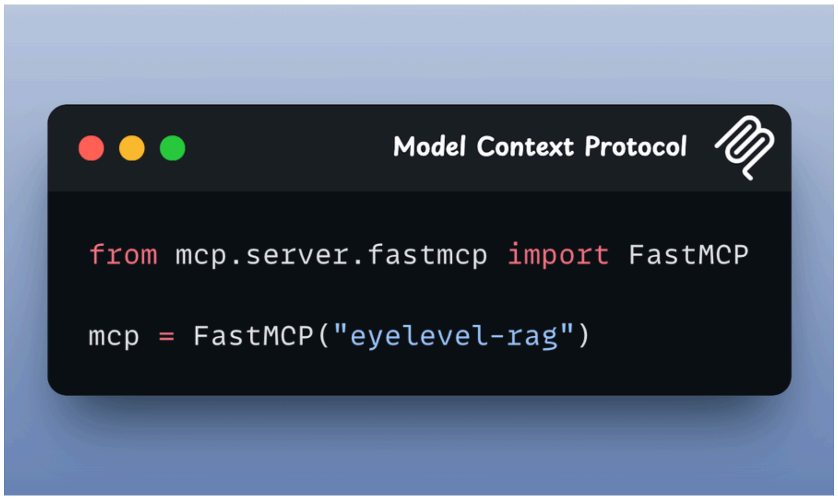
Workflow

- User interacts with the MCP client (Cursor IDE)
- Client connects to the MCP server and selects a tool.
- Tools leverage GroundX to do an advanced search over docs
- Search results are used by Client to generate response

Let's implement this!

#1) Setup server

First we setup a local MCP server, using FastMCP and provide it a name

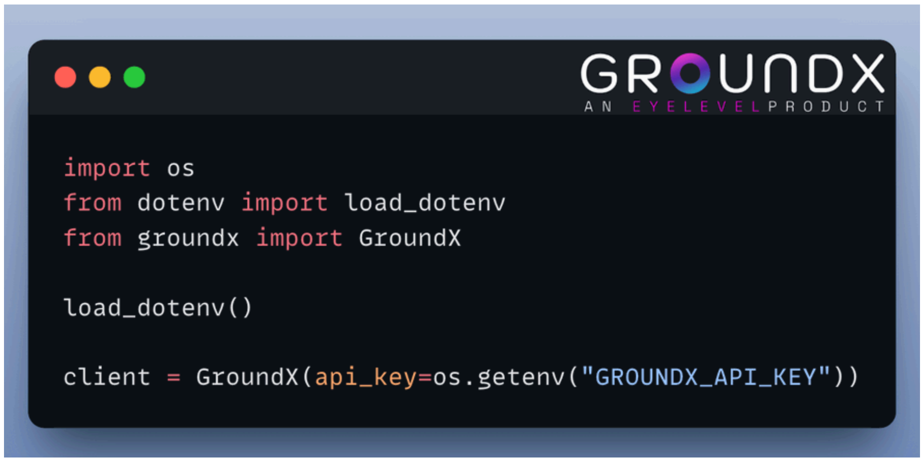


```python
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("eyelevel-rag")
```

#2) Create GroundX Client

GroundX offers capabilities document search and retrieval capabilities for complex real-world documents.

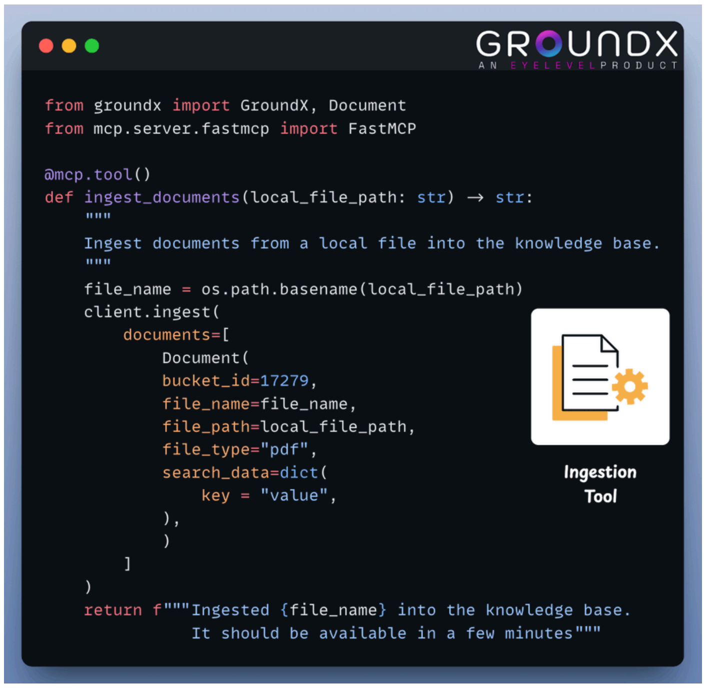
Here's how to set up a client:



```python
import os
from dotenv import load_dotenv
from groundx import GroundX

load_dotenv()

client = GroundX(api_key=os.getenv("GROUNDX_API_KEY"))
```

#3) Create Ingestion tool

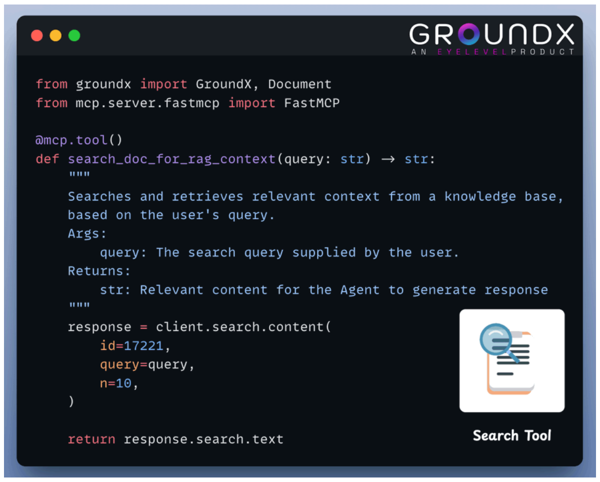
This tool is used to ingest new documents into the knowledge base. User just needs to provide a path to the document to be ingested:



```python
from groundx import GroundX, Document
from mcp.server.fastmcp import FastMCP

@mcp.tool()
def ingest_documents(local_file_path: str) → str:
    """
    Ingest documents from a local file into the knowledge base.
    """
    file_name = os.path.basename(local_file_path)
    client.ingest(
        documents=[
            Document(
            bucket_id=17279,
            file_name=file_name,
            file_path=local_file_path,
            file_type="pdf",
            search_data=dict(
                key = "value",
            ),
            )
        ]
    )
    return f"""Ingested {file_name} into the knowledge base.
               It should be available in a few minutes"""
```

#4) Create Search tool

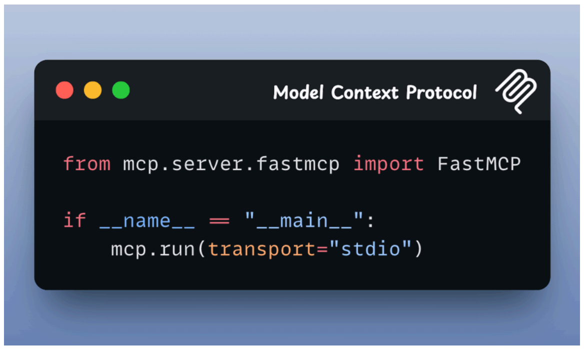
This tool leverages GroundX's advanced capabilities to do search and retrieval from complex real world documents. Here's how to implement it:



```python
from groundx import GroundX, Document
from mcp.server.fastmcp import FastMCP

@mcp.tool()
def search_doc_for_rag_context(query: str) → str:
    """
    Searches and retrieves relevant context from a knowledge base,
    based on the user's query.
    Args:
        query: The search query supplied by the user.
    Returns:
        str: Relevant content for the Agent to generate response
    """
    response = client.search.content(
        id=17221,
        query=query,
        n=10,
    )

    return response.search.text
```

#5) Start the server

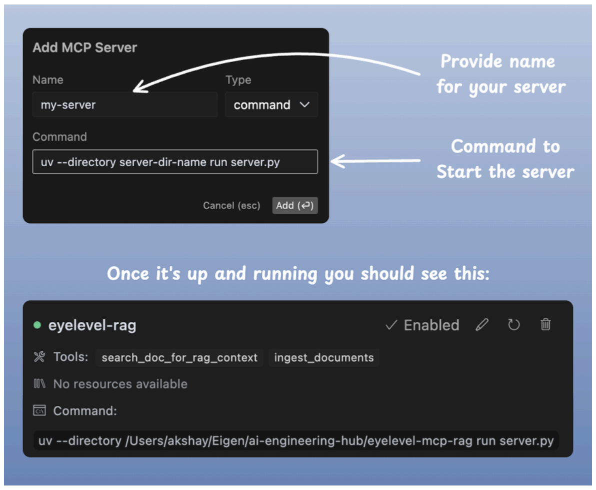
Starts an MCP server using stdio as the transport mechanism:



```python
from mcp.server.fastmcp import FastMCP

if __name__ == "__main__":
    mcp.run(transport="stdio")
```

#6) Connect to Cursor

Inside you Cursor IDE follow this: Cursor → Settings → Cursor Settings → MCP Then add and start your server like this:


Add MCP Server
Name
Type
Command
Cancel (esc)
Add (↩)
Provide name for your server
Command to Start the server
Once it's up and running you should see this:
Enabled

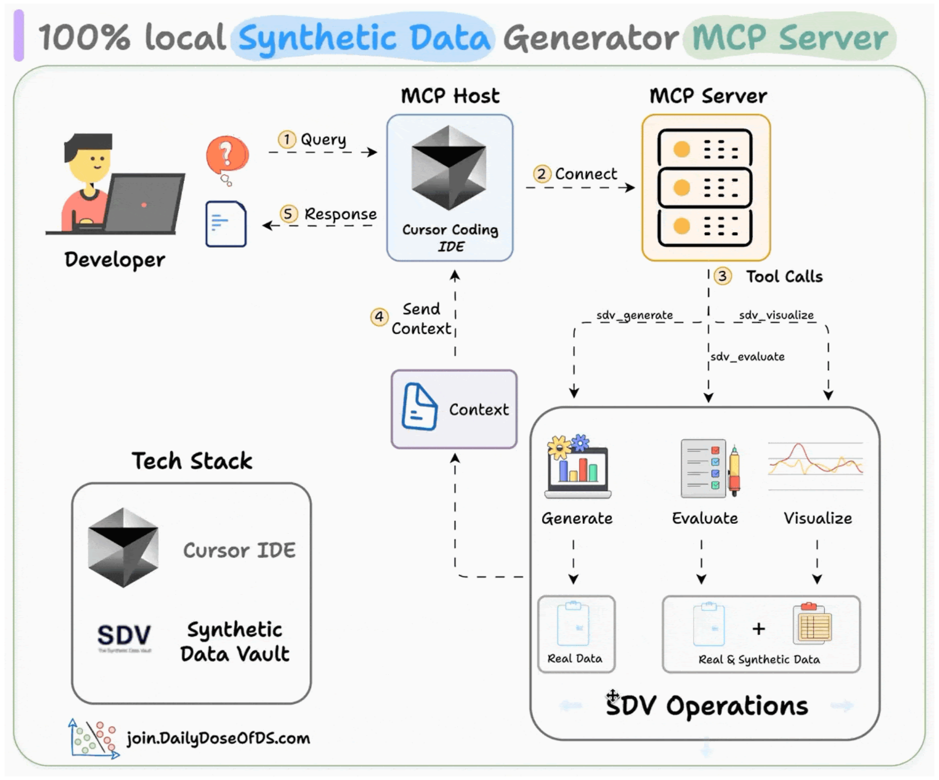
Name: my-server
Type: command
Command: uv --directory server-dir-name run server.py

● eyelevel-rag
Tools: search_doc_for_rag_context ingest_documents
No resources available
Command:
uv --directory /Users/akshay/Eigen/ai-engineering-hub/eyelevel-mcp-rag run server.py

The code is available here:
https://www.dailydoseofds.com/p/mcp-powered-rag-over-complex-docs/

#8) MCP-powered synthetic data generator

Learn how to build an MCP server that can generate any type of synthetic dataset. It uses Cursor as the MCP host and SDV to generate realistic tabular synthetic data.



Tech Stack

- Cursor as the MCP host
- Datacebo's SDV to generate realistic tabular synthetic data

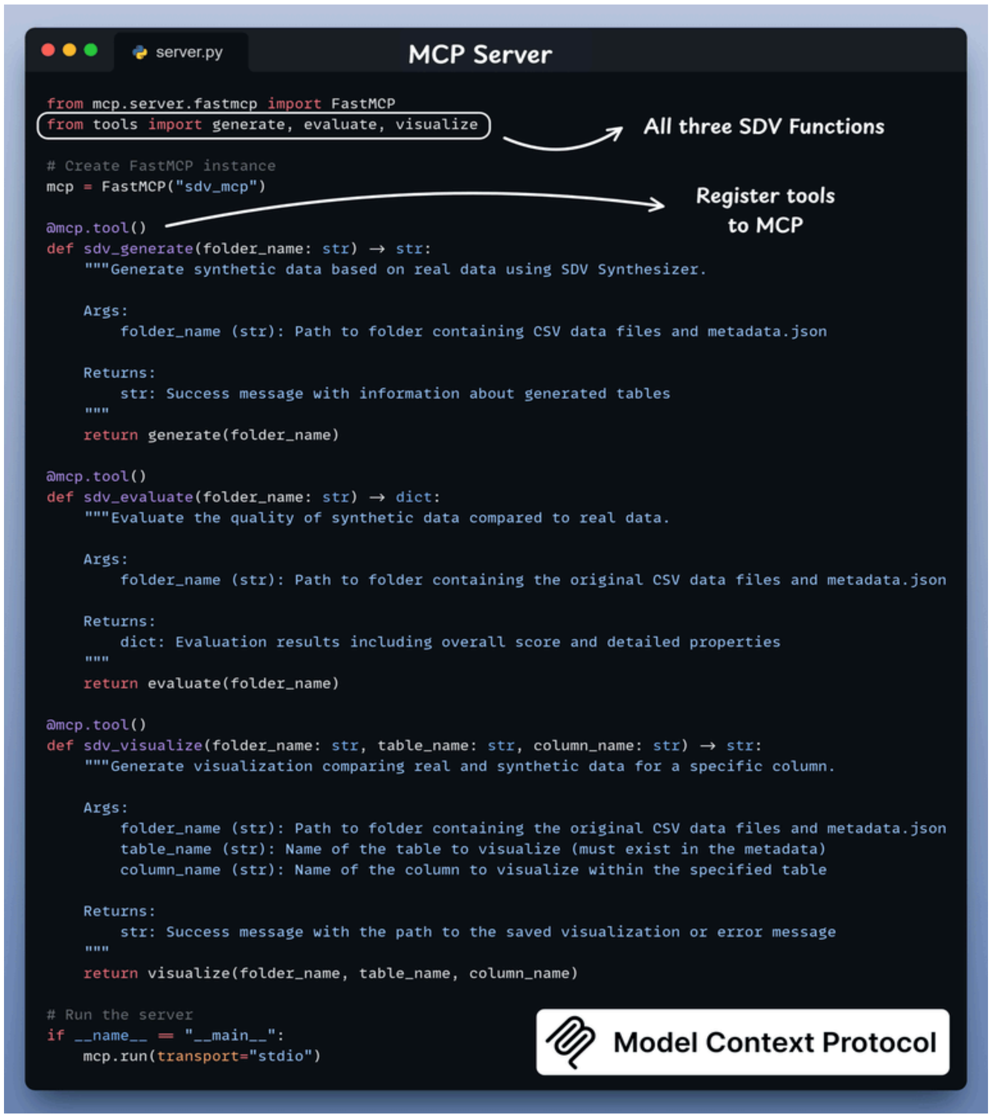
Workflow

- User submits a query
- Agent connects to MCP server to find tools
- Agent uses appropriate tool based on query
- Returns response on synthetic data creation, eval, or visualization

Here's an overview of our MCP server, which includes three tools:

- SDV Generate
- SDV Evaluate
- SDV Visualise

We have kept the actual implementation of these tools using the SDV SDK in a separate file, tools[.]py, that is imported here.



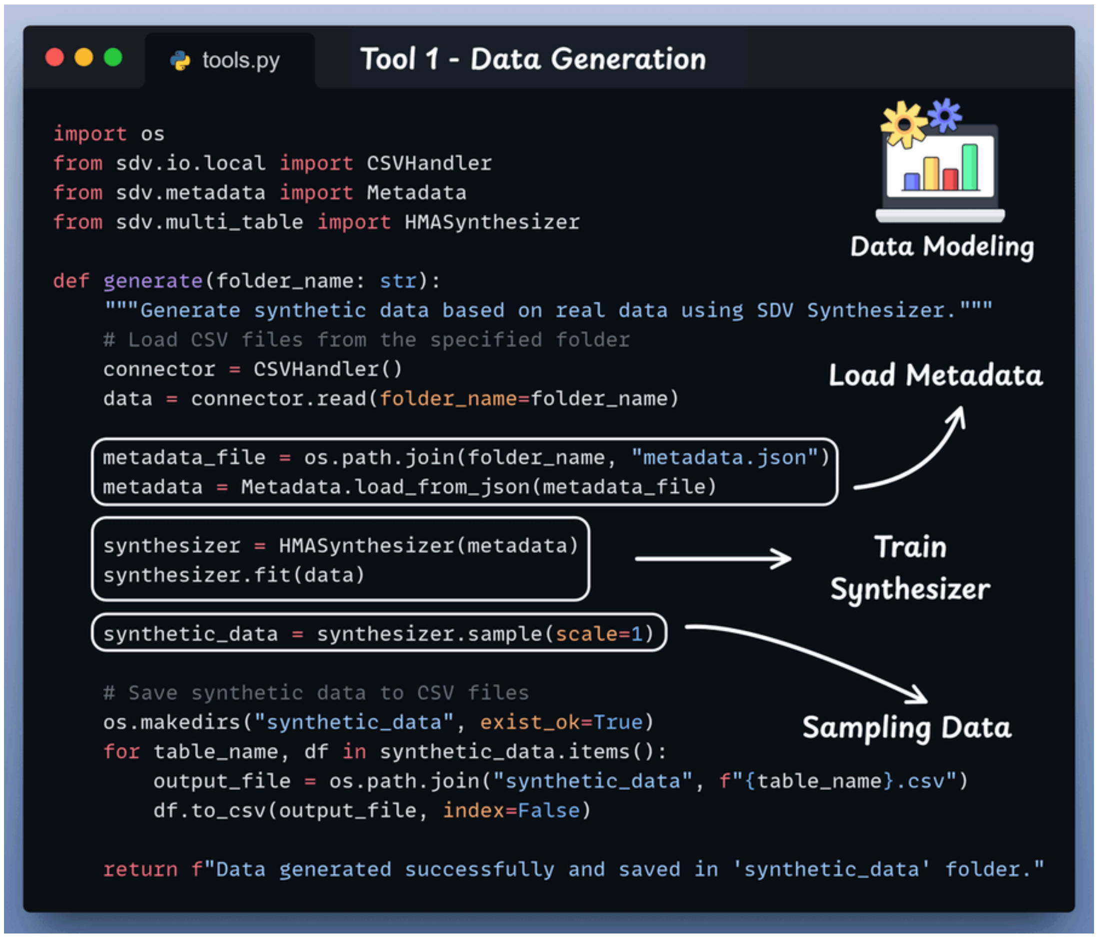
```python
from mcp.server.fastmcp import FastMCP
from tools import generate, evaluate, visualize

# Create FastMCP instance
mcp = FastMCP("sdv_mcp")

@mcp.tool()
def sdv_generate(folder_name: str) -> str:
    """Generate synthetic data based on real data using SDV Synthesizer.

    Args:
        folder_name (str): Path to folder containing CSV data files and metadata.json

    Returns:
        str: Success message with information about generated tables
    """
    return generate(folder_name)

@mcp.tool()
def sdv_evaluate(folder_name: str) -> dict:
    """Evaluate the quality of synthetic data compared to real data.

    Args:
        folder_name (str): Path to folder containing the original CSV data files and metadata.json

    Returns:
        dict: Evaluation results including overall score and detailed properties
    """
    return evaluate(folder_name)

@mcp.tool()
def sdv_visualize(folder_name: str, table_name: str, column_name: str) -> str:
    """Generate visualization comparing real and synthetic data for a specific column.

    Args:
        folder_name (str): Path to folder containing the original CSV data files and metadata.json
        table_name (str): Name of the table to visualize (must exist in the metadata)
        column_name (str): Name of the column to visualize within the specified table

    Returns:
        str: Success message with the path to the saved visualization or error message
    """
    return visualize(folder_name, table_name, column_name)

# Run the server
if __name__ == "__main__":
    mcp.run(transport="stdio")
```

Now let's look at each tool in more details.

#1) SDV Generate Tool

This tool creates synthetic data from real data using the SDV Synthesizer.

SDV offers a variety of synthesizers, each utilizing different algorithms to produce synthetic data.

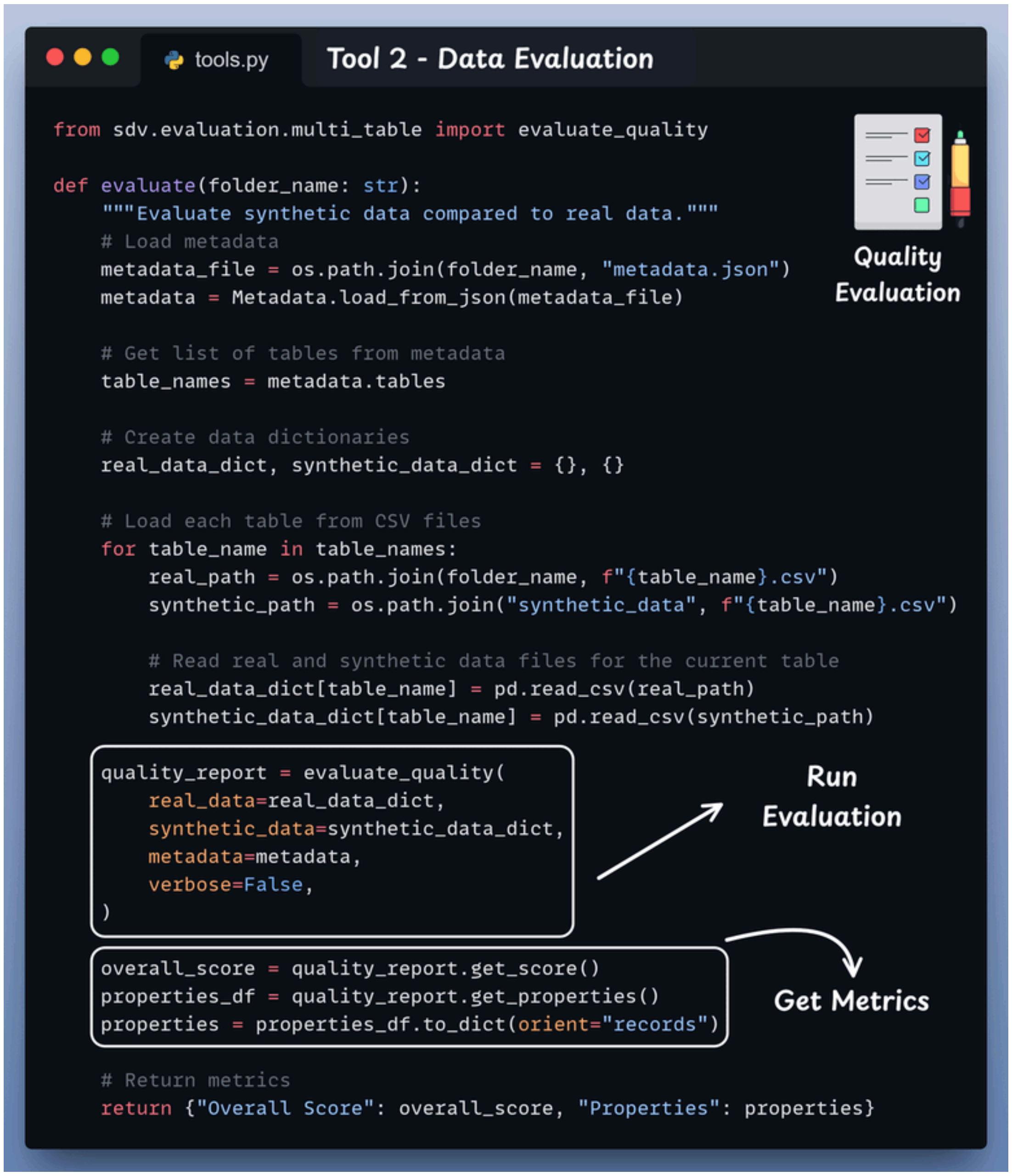


```python
import os
from sdv.io.local import CSVHandler
from sdv.metadata import Metadata
from sdv.multi_table import HMASynthesizer

def generate(folder_name: str):
    """Generate synthetic data based on real data using SDV Synthesizer."""
    # Load CSV files from the specified folder
    connector = CSVHandler()
    data = connector.read(folder_name=folder_name)

    metadata_file = os.path.join(folder_name, "metadata.json")
    metadata = Metadata.load_from_json(metadata_file)

    synthesizer = HMASynthesizer(metadata)
    synthesizer.fit(data)

    synthetic_data = synthesizer.sample(scale=1)

    # Save synthetic data to CSV files
    os.makedirs("synthetic_data", exist_ok=True)
    for table_name, df in synthetic_data.items():
        output_file = os.path.join("synthetic_data", f"{table_name}.csv")
        df.to_csv(output_file, index=False)

    return f"Data generated successfully and saved in 'synthetic_data' folder."
```

#2) SDV Evaluate Tool

This tool evaluates the quality of synthetic data in comparison to real data.

We will assess statistical similarity to determine which real data patterns are captured by the synthetic data.

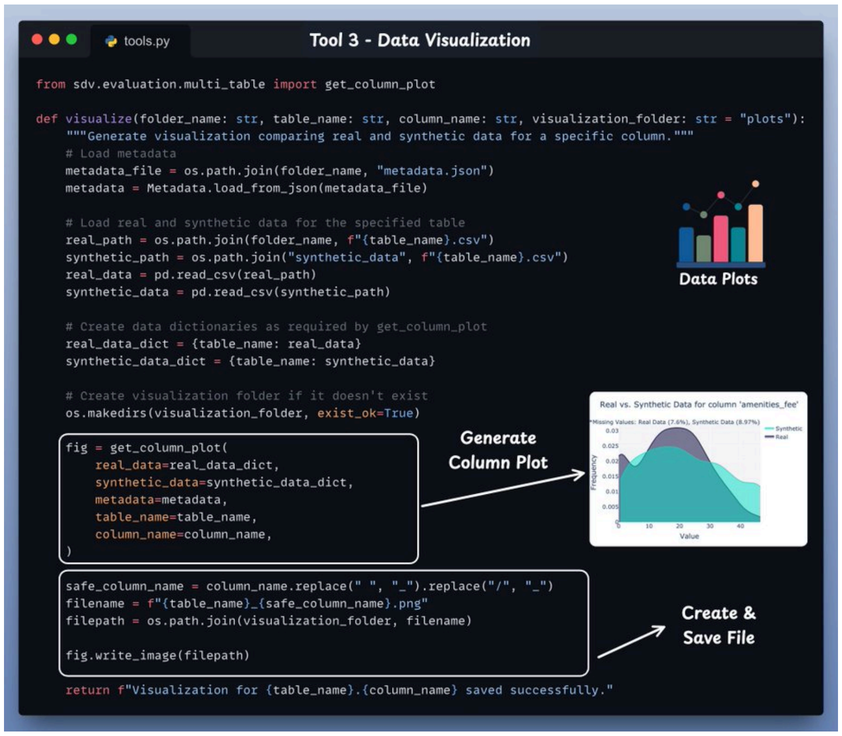


```python
from sdv.evaluation.multi_table import evaluate_quality

def evaluate(folder_name: str):
    """Evaluate synthetic data compared to real data."""
    # Load metadata
    metadata_file = os.path.join(folder_name, "metadata.json")
    metadata = Metadata.load_from_json(metadata_file)

    # Get list of tables from metadata
    table_names = metadata.tables

    # Create data dictionaries
    real_data_dict, synthetic_data_dict = {}, {}

    # Load each table from CSV files
    for table_name in table_names:
        real_path = os.path.join(folder_name, f"{table_name}.csv")
        synthetic_path = os.path.join("synthetic_data", f"{table_name}.csv")

        # Read real and synthetic data files for the current table
        real_data_dict[table_name] = pd.read_csv(real_path)
        synthetic_data_dict[table_name] = pd.read_csv(synthetic_path)

    quality_report = evaluate_quality(
        real_data=real_data_dict,
        synthetic_data=synthetic_data_dict,
        metadata=metadata,
        verbose=False,
    )

    overall_score = quality_report.get_score()
    properties_df = quality_report.get_properties()
    properties = properties_df.to_dict(orient="records")

    # Return metrics
    return {"Overall Score": overall_score, "Properties": properties}
```

#3) SDV Visualize Tool

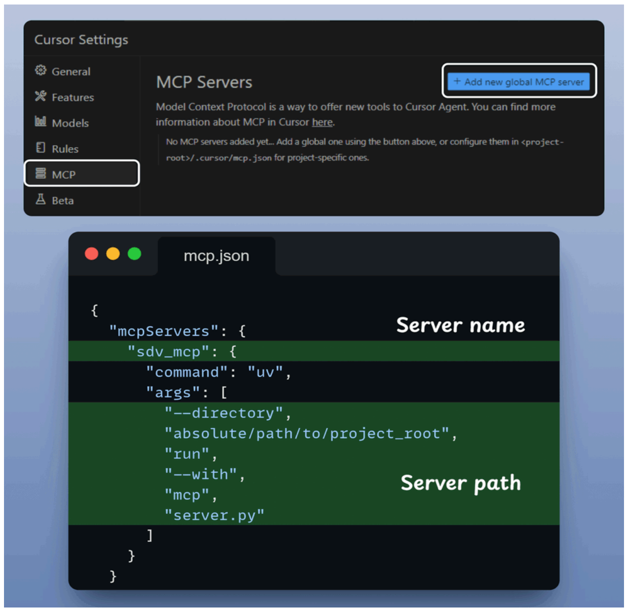
This tool generates a visualization to compare real and synthetic data for a specific column.

Use this function to visualize a real column alongside its corresponding synthetic column.



```python
from sdv.evaluation.multi_table import get_column_plot

def visualize(folder_name: str, table_name: str, column_name: str, visualization_folder: str = "plots"):
    """Generate visualization comparing real and synthetic data for a specific column."""
    # Load metadata
    metadata_file = os.path.join(folder_name, "metadata.json")
    metadata = Metadata.load_from_json(metadata_file)

    # Load real and synthetic data for the specified table
    real_path = os.path.join(folder_name, f"{table_name}.csv")
    synthetic_path = os.path.join("synthetic_data", f"{table_name}.csv")
    real_data = pd.read_csv(real_path)
    synthetic_data = pd.read_csv(synthetic_path)

    # Create data dictionaries as required by get_column_plot
    real_data_dict = {table_name: real_data}
    synthetic_data_dict = {table_name: synthetic_data}

    # Create visualization folder if it doesn't exist
    os.makedirs(visualization_folder, exist_ok=True)

    fig = get_column_plot(
        real_data=real_data_dict,
        synthetic_data=synthetic_data_dict,
        metadata=metadata,
        table_name=table_name,
        column_name=column_name,
    )

    safe_column_name = column_name.replace(" ", "_").replace("/", "_")
    filename = f"{table_name}_{safe_column_name}.png"
    filepath = os.path.join(visualization_folder, filename)

    fig.write_image(filepath)

    return f"Visualization for {table_name}.{column_name} saved successfully."
```

With tools and server ready, lets integrate it with our Cursor IDE! Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server. In the JSON file, add what's shown below



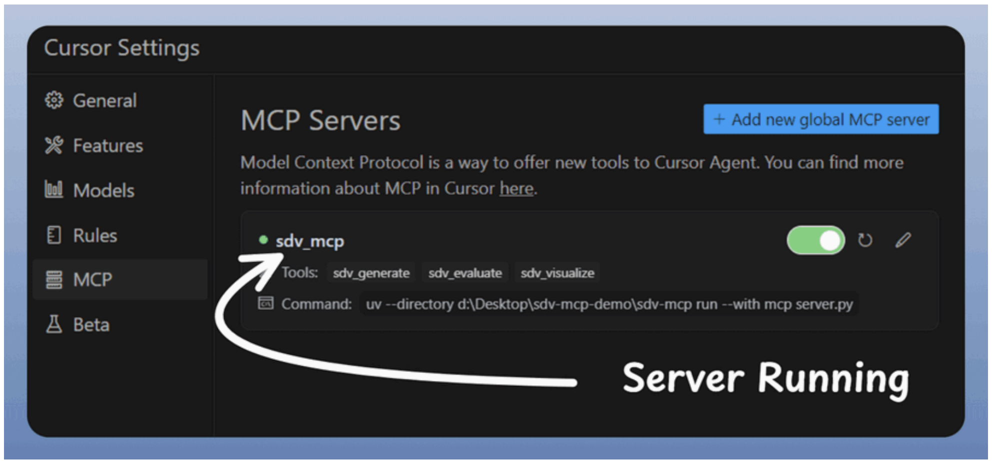
MCP Servers

Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

No MCP servers added yet... Add a global one using the button above, or configure them in `<project-root>/.cursor/mcp.json` for project-specific ones.

```json
{
  "mcpServers": {
    "sdv_mcp": {
      "command": "uv",
      "args": [
        "--directory",
        "absolute/path/to/project_root",
        "run",
        "--with",
        "mcp",
        "server.py"
      ]
    }
  }
```

Done! Your synthetic data generator MCP server is live and connected to Cursor.



Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

sdv_mcp

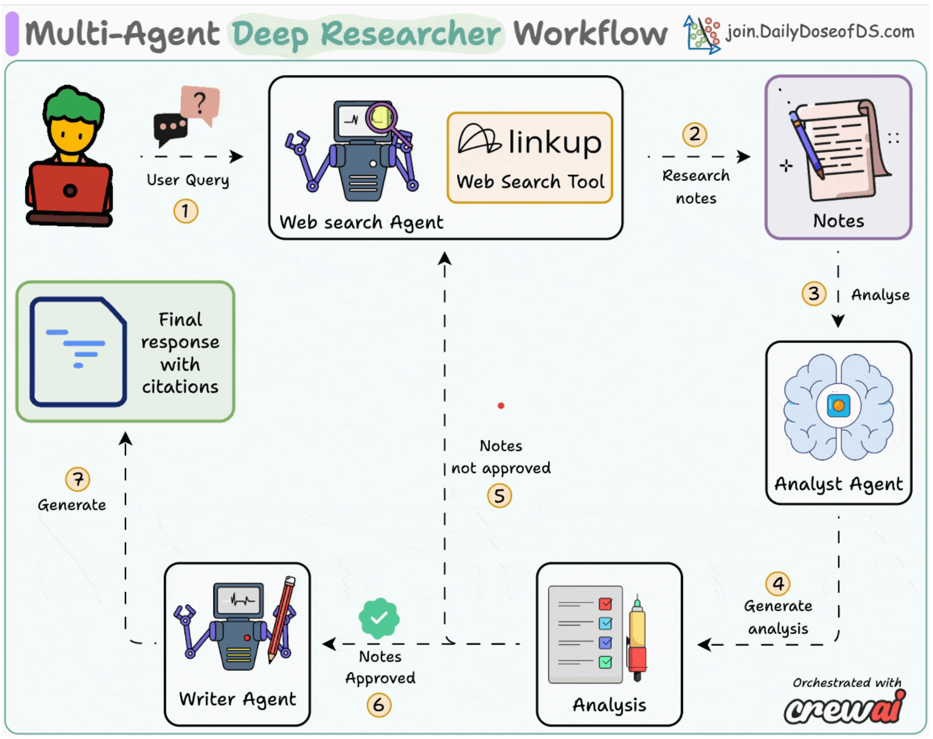
Tools: sdv_generate sdv_evaluate sdv_visualize

Command: uv --directory d:\Desktop\sdv-mcp-demo\sdv-mcp run --with mcp server.py

The code is available here:
https://www.dailydoseofds.com/p/hands-on-mcp-powered-synthetic-data-generator/

#9) MCP-powered deep researcher

ChatGPT has a deep research feature. It helps you get detailed insights on any topic. Learn how you can build a 100% local alternative to it.



Tech Stack

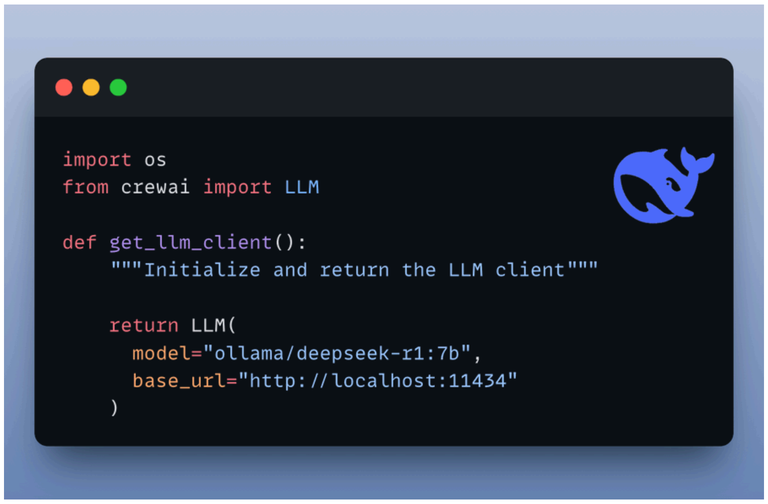
- Linkup platform for deep web research
- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek
- Cursor as MCP host

Workflow

- User submits a query
- Web search agent runs deep web search via Linkup
- Research analyst verifies and deduplicates results
- Technical writer crafts a coherent response with citations

#1) Setup LLM

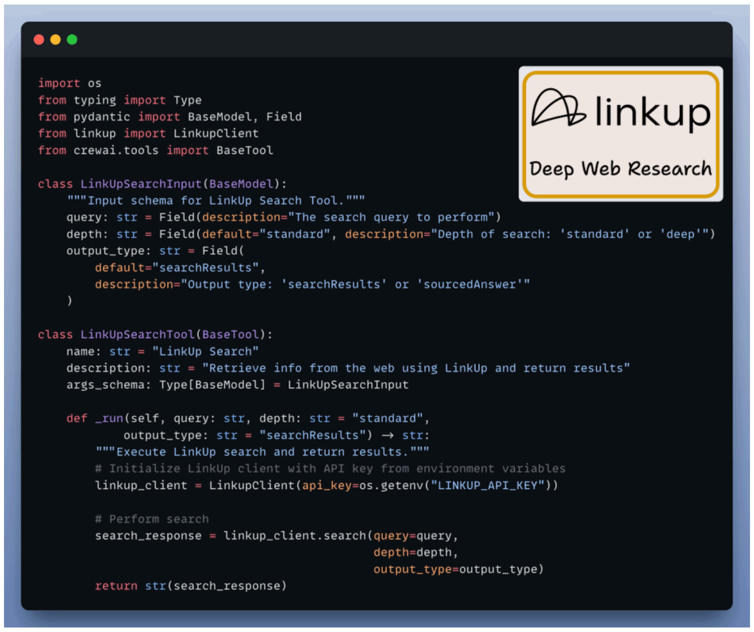
We'll use a locally served DeepSeek-R1 using Ollama.



```python
import os
from crewai import LLM

def get_llm_client():
    """Initialize and return the LLM client"""

    return LLM(
      model="ollama/deepseek-r1:7b",
      base_url="http://localhost:11434"
    )
```

#2) Define Web Search Tool

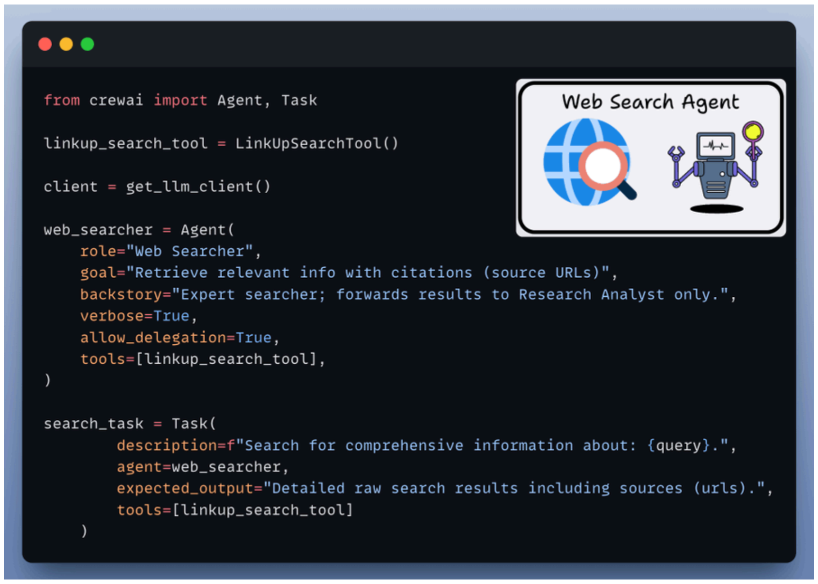
We'll use Linkup platform's powerful search capabilities, which rival Perplexity and OpenAI, to power our web search agent. This is done by defining a custom tool that our agent can use.



```python
import os
from typing import Type
from pydantic import BaseModel, Field
from linkup import LinkupClient
from crewai.tools import BaseTool

class LinkUpSearchInput(BaseModel):
    """Input schema for LinkUp Search Tool."""
    query: str = Field(description="The search query to perform")
    depth: str = Field(default="standard", description="Depth of search: 'standard' or 'deep'")
    output_type: str = Field(
        default="searchResults",
        description="Output type: 'searchResults' or 'sourcedAnswer'"
    )

class LinkUpSearchTool(BaseTool):
    name: str = "LinkUp Search"
    description: str = "Retrieve info from the web using LinkUp and return results"
    args_schema: Type[BaseModel] = LinkUpSearchInput

    def _run(self, query: str, depth: str = "standard",
            output_type: str = "searchResults") → str:
        """Execute LinkUp search and return results."""
        # Initialize LinkUp client with API key from environment variables
        linkup_client = LinkupClient(api_key=os.getenv("LINKUP_API_KEY"))

        # Perform search
        search_response = linkup_client.search(query=query,
                                               depth=depth,
                                               output_type=output_type)
        return str(search_response)
```

#3) Define Web Search Agent

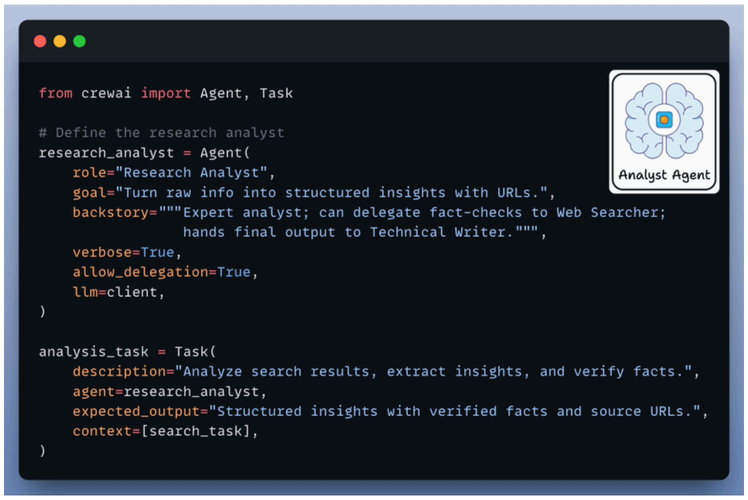
The web search agent gathers up-to-date information from the internet based on user query. The linkup tool we defined earlier is used by this agent.



```python
from crewai import Agent, Task

linkup_search_tool = LinkUpSearchTool()

client = get_llm_client()

web_searcher = Agent(
    role="Web Searcher",
    goal="Retrieve relevant info with citations (source URLs)",
    backstory="Expert searcher; forwards results to Research Analyst only.",
    verbose=True,
    allow_delegation=True,
    tools=[linkup_search_tool],
)

search_task = Task(
        description=f"Search for comprehensive information about: {query}.",
        agent=web_searcher,
        expected_output="Detailed raw search results including sources (urls).",
        tools=[linkup_search_tool]
    )
```

#4) Define Research Analyst Agent

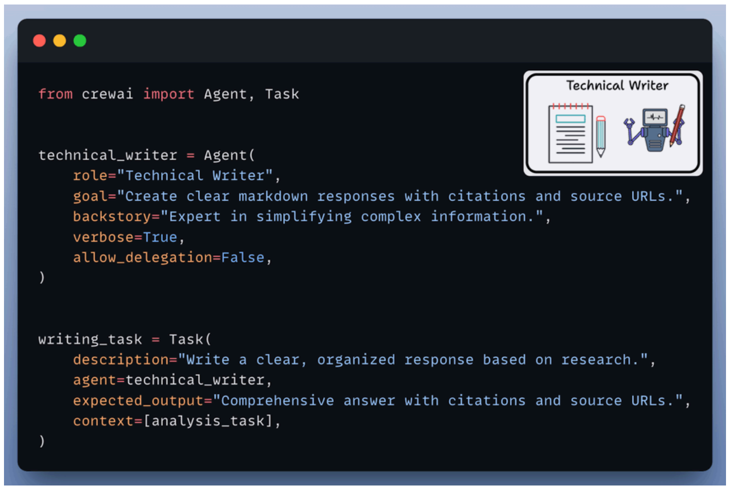
This agent transforms raw web search results into structured insights, with source URLs. It can also delegate tasks back to the web search agent for verification and fact-checking.



```python
from crewai import Agent, Task

# Define the research analyst
research_analyst = Agent(
    role="Research Analyst",
    goal="Turn raw info into structured insights with URLs.",
    backstory="""Expert analyst; can delegate fact-checks to Web Searcher;
                 hands final output to Technical Writer.""",
    verbose=True,
    allow_delegation=True,
    llm=client,
)

analysis_task = Task(
    description="Analyze search results, extract insights, and verify facts.",
    agent=research_analyst,
    expected_output="Structured insights with verified facts and source URLs.",
    context=[search_task],
)
```

#5) Define Technical Writer Agent

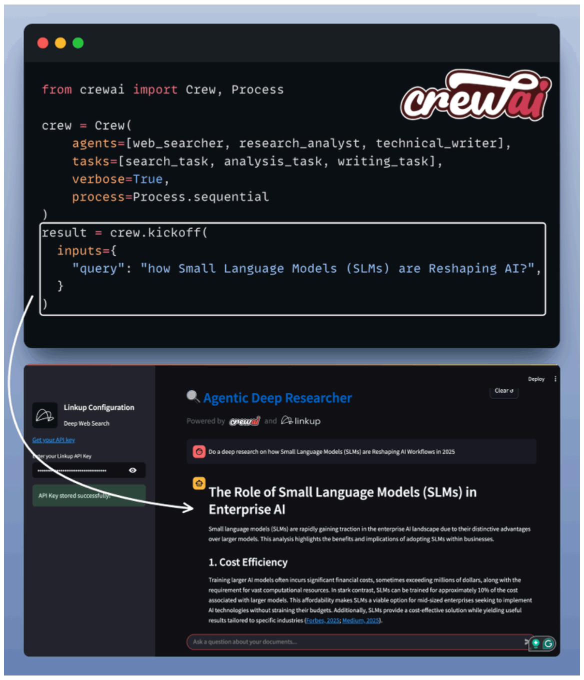
It takes the analyzed and verified results from the analyst agent and drafts a coherent response with citations for the end user.



```python
from crewai import Agent, Task


technical_writer = Agent(
    role="Technical Writer",
    goal="Create clear markdown responses with citations and source URLs.",
    backstory="Expert in simplifying complex information.",
    verbose=True,
    allow_delegation=False,
)


writing_task = Task(
    description="Write a clear, organized response based on research.",
    agent=technical_writer,
    expected_output="Comprehensive answer with citations and source URLs.",
    context=[analysis_task],
)
```

#6) Setup Crew

Finally, once we have all the agents and tools defined we set up and kickoff our deep researcher crew.



```python
from crewai import Crew, Process

crew = Crew(
    agents=[web_searcher, research_analyst, technical_writer],
    tasks=[search_task, analysis_task, writing_task],
    verbose=True,
    process=Process.sequential
)
result = crew.kickoff(
  inputs={
    "query": "how Small Language Models (SLMs) are Reshaping AI?",
  }
)
```

Deploy
Clear
Linkup Configuration
Deep Web Search
Get your API key
Enter your Linkup API Key
API Key stored successfully!
Powered by crewai and linkup
Ask a question about your documents...

Agentic Deep Researcher

Do a deep research on how Small Language Models (SLMs) are Reshaping AI Workflows in 2025

The Role of Small Language Models (SLMs) in Enterprise AI

Small language models (SLMs) are rapidly gaining traction in the enterprise AI landscape due to their distinctive advantages over larger models. This analysis highlights the benefits and implications of adopting SLMs within businesses.

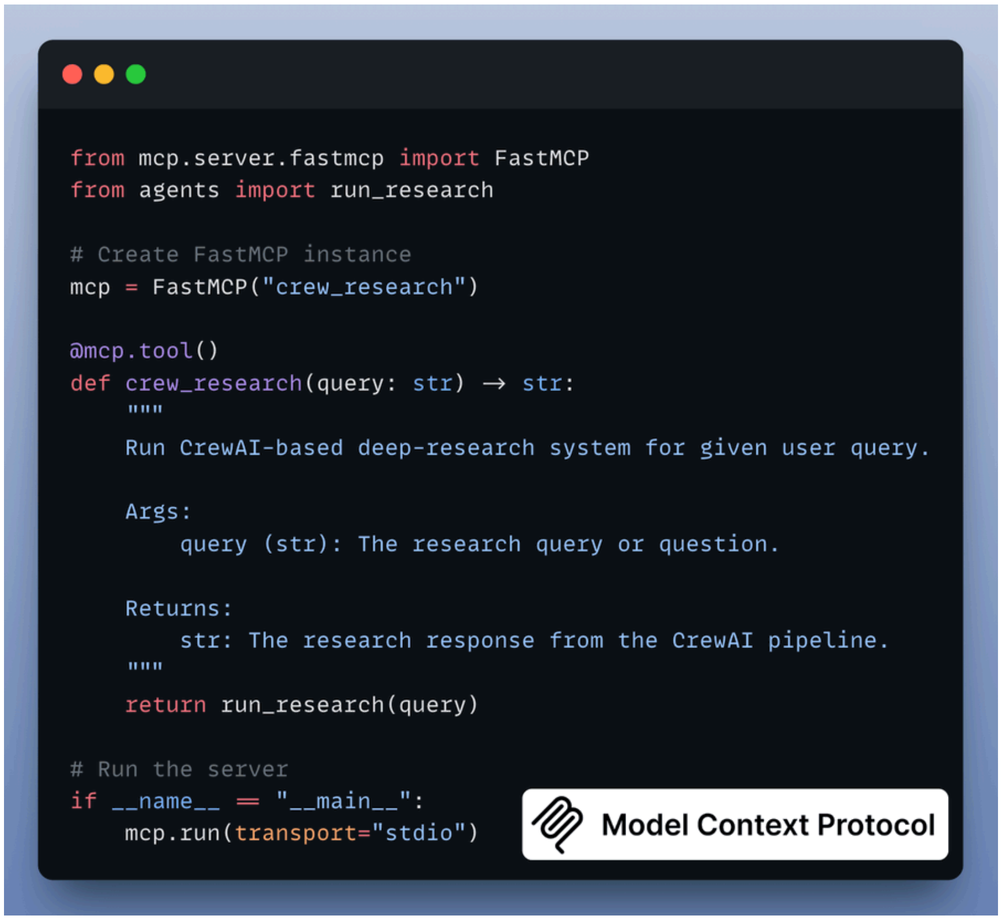
1. Cost Efficiency

Training larger AI models often incurs significant financial costs, sometimes exceeding millions of dollars, along with the requirement for vast computational resources. In stark contrast, SLMs can be trained for approximately 10% of the cost associated with larger models. This affordability makes SLMs a viable option for mid-sized enterprises seeking to implement AI technologies without straining their budgets. Additionally, SLMs provide a cost-effective solution while yielding useful results tailored to specific industries (Forbes, 2025; Medium, 2025).

#7) Create MCP Server

Now, we'll encapsulate our deep research team within an MCP tool. With just a few lines of code, our MCP server will be ready.

Let's see how to connect it with Cursor.

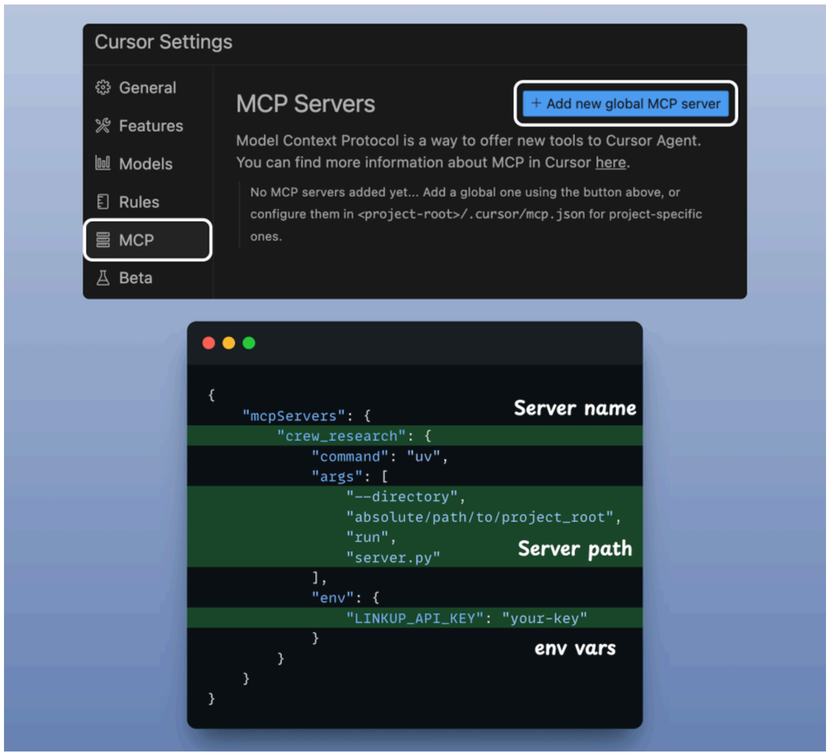


```python
from mcp.server.fastmcp import FastMCP
from agents import run_research

# Create FastMCP instance
mcp = FastMCP("crew_research")

@mcp.tool()
def crew_research(query: str) → str:
    """
    Run CrewAI-based deep-research system for given user query.

    Args:
        query (str): The research query or question.

    Returns:
        str: The research response from the CrewAI pipeline.
    """
    return run_research(query)

# Run the server
if __name__ == "__main__":
    mcp.run(transport="stdio")
```

#8) Integrate MCP server with Cursor

Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server

In the JSON file, add what's shown below

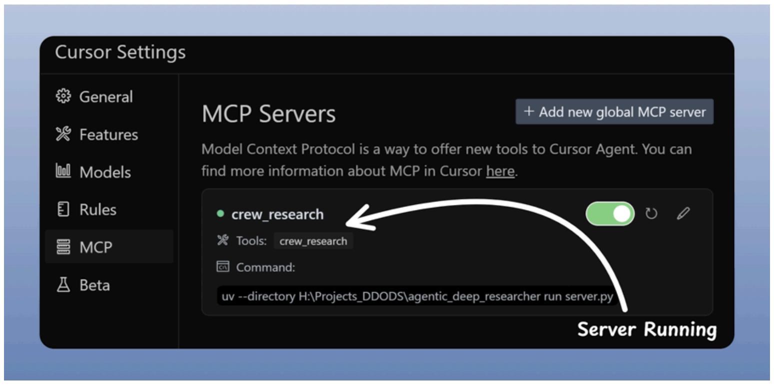


Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

No MCP servers added yet... Add a global one using the button above, or configure them in <project-root>/.cursor/mcp.json for project-specific ones.

```
{
    "mcpServers": {
        "crew_research": {
            "command": "uv",
            "args": [
                "--directory",
                "absolute/path/to/project_root",
                "run",
                "server.py"
            ],
            "env": {
                "LINKUP_API_KEY": "your-key"
            }
        }
    }
}
```

Done! Your deep research MCP server is live and connected to Cursor.



Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

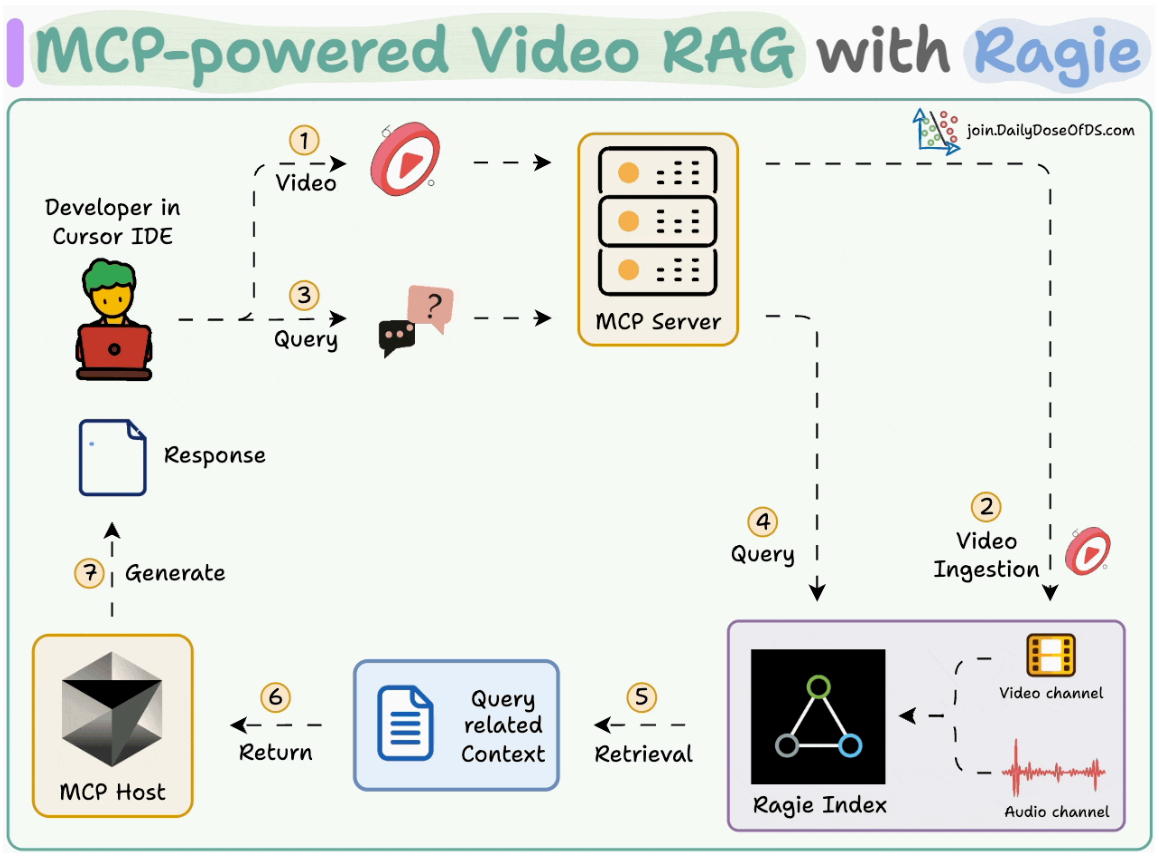
crew_research
Tools: crew_research
Command:

```
uv --directory H:\Projects_DDODS\agentic_deep_researcher run server.py
```

The code is available here:
https://www.dailydoseofds.com/p/hands-on-mcp-powered-deep-researcher/

#10) MCP-powered RAG over videos

We have an MCP-driven video RAG that ingests a video and lets you chat with it. It also fetches the exact video chunk where an event occurred.



Tech Stack

- RagieAI for video ingestion and retrieval.
- Cursor as the MCP host.

Workflow

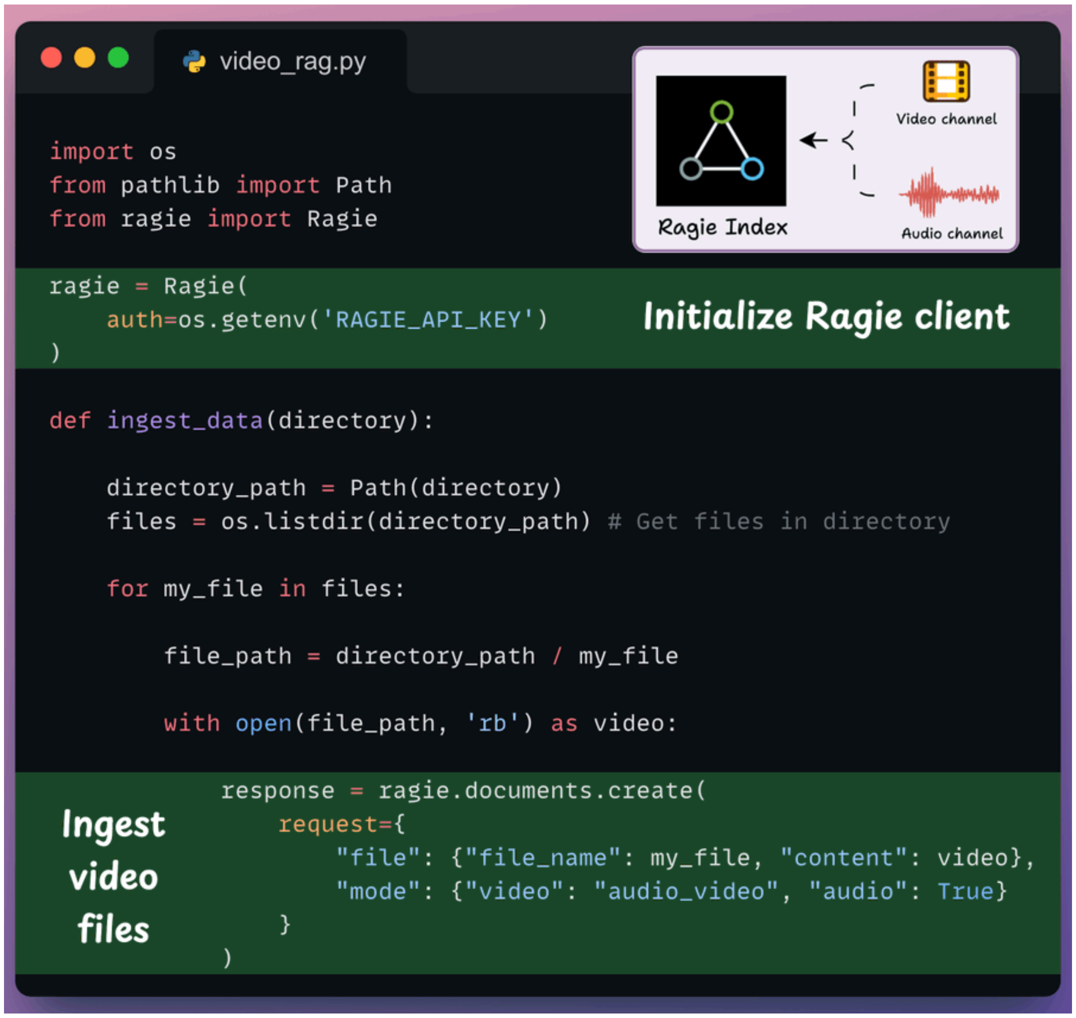
- User specifies video files and a query.
- An Ingestion tool indexes the videos in Ragie.
- A Query tool retrieves info from Ragie Index with citations.
- Show-video tool returns the video chunk that answers the query

Let's implement this!

#1) Ingest data

We implement a method to ingest video files into the Ragie index.

We also specify the audio-video mode to load both audio and video channels during ingestion.

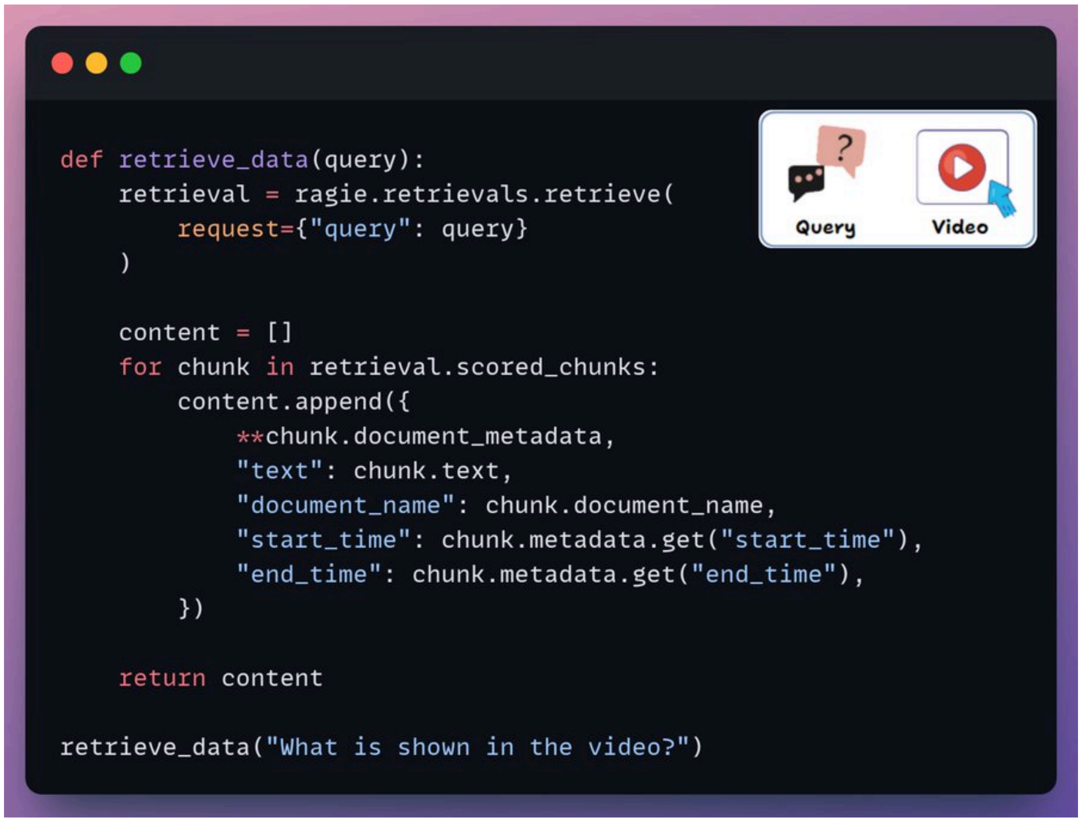


```python
import os
from pathlib import Path
from ragie import Ragie

ragie = Ragie(
    auth=os.getenv('RAGIE_API_KEY')
)

def ingest_data(directory):

    directory_path = Path(directory)
    files = os.listdir(directory_path) # Get files in directory

    for my_file in files:

        file_path = directory_path / my_file

        with open(file_path, 'rb') as video:

            response = ragie.documents.create(
                request={
                    "file": {"file_name": my_file, "content": video},
                    "mode": {"video": "audio_video", "audio": True}
                }
            )
```

#2) Retrieve data

We retrieve the relevant chunks from the video based on the user query.

Each chunk has a start time, an end time, and a few more details that correspond to the video segment.

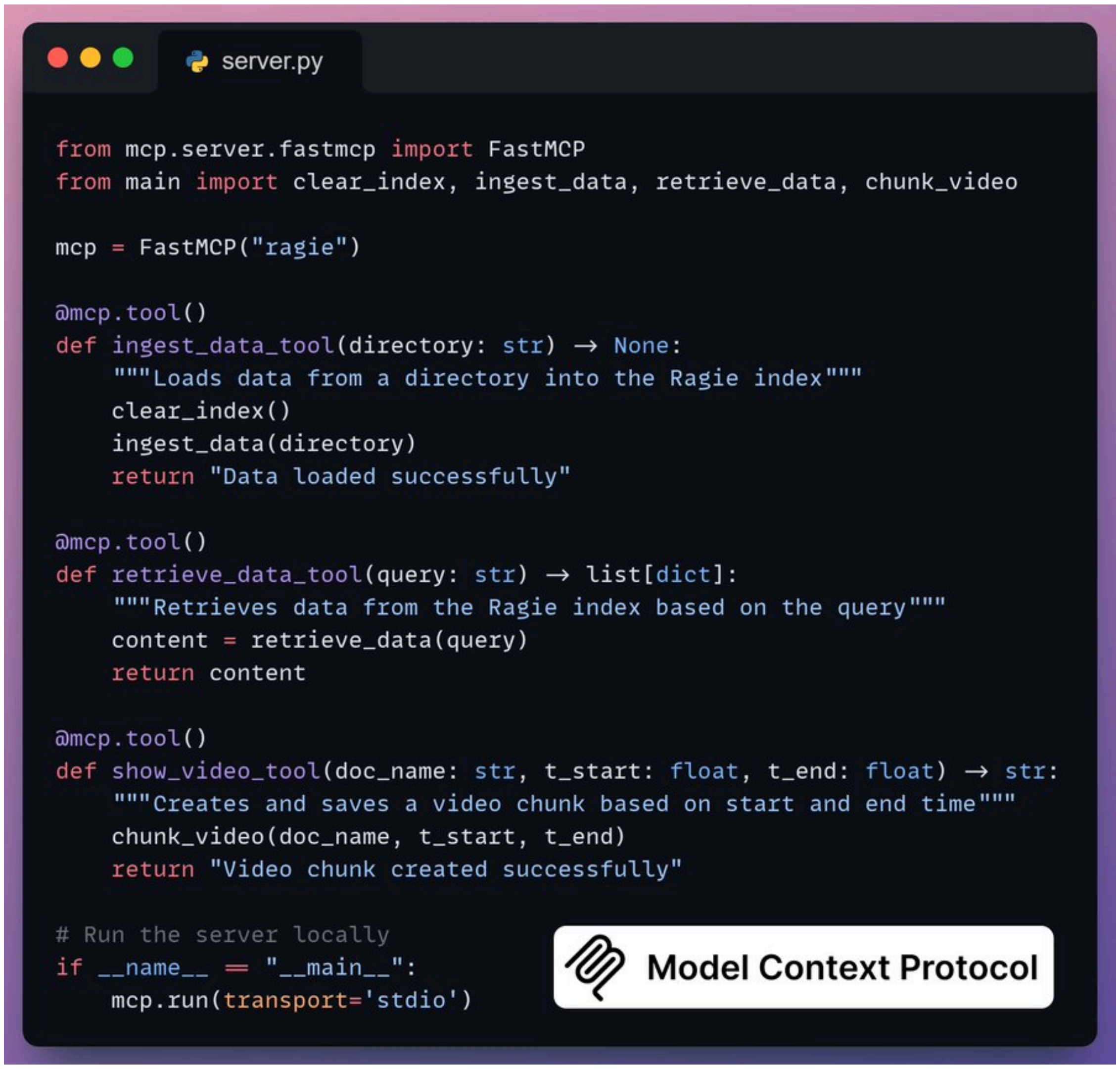


```python
def retrieve_data(query):
    retrieval = ragie.retrievals.retrieve(
        request={"query": query}
    )

    content = []
    for chunk in retrieval.scored_chunks:
        content.append({
            **chunk.document_metadata,
            "text": chunk.text,
            "document_name": chunk.document_name,
            "start_time": chunk.metadata.get("start_time"),
            "end_time": chunk.metadata.get("end_time"),
        })

    return content

retrieve_data("What is shown in the video?")
```

#3) Create MCP Server

We integrate our RAG pipeline into an MCP server with 3 tools:

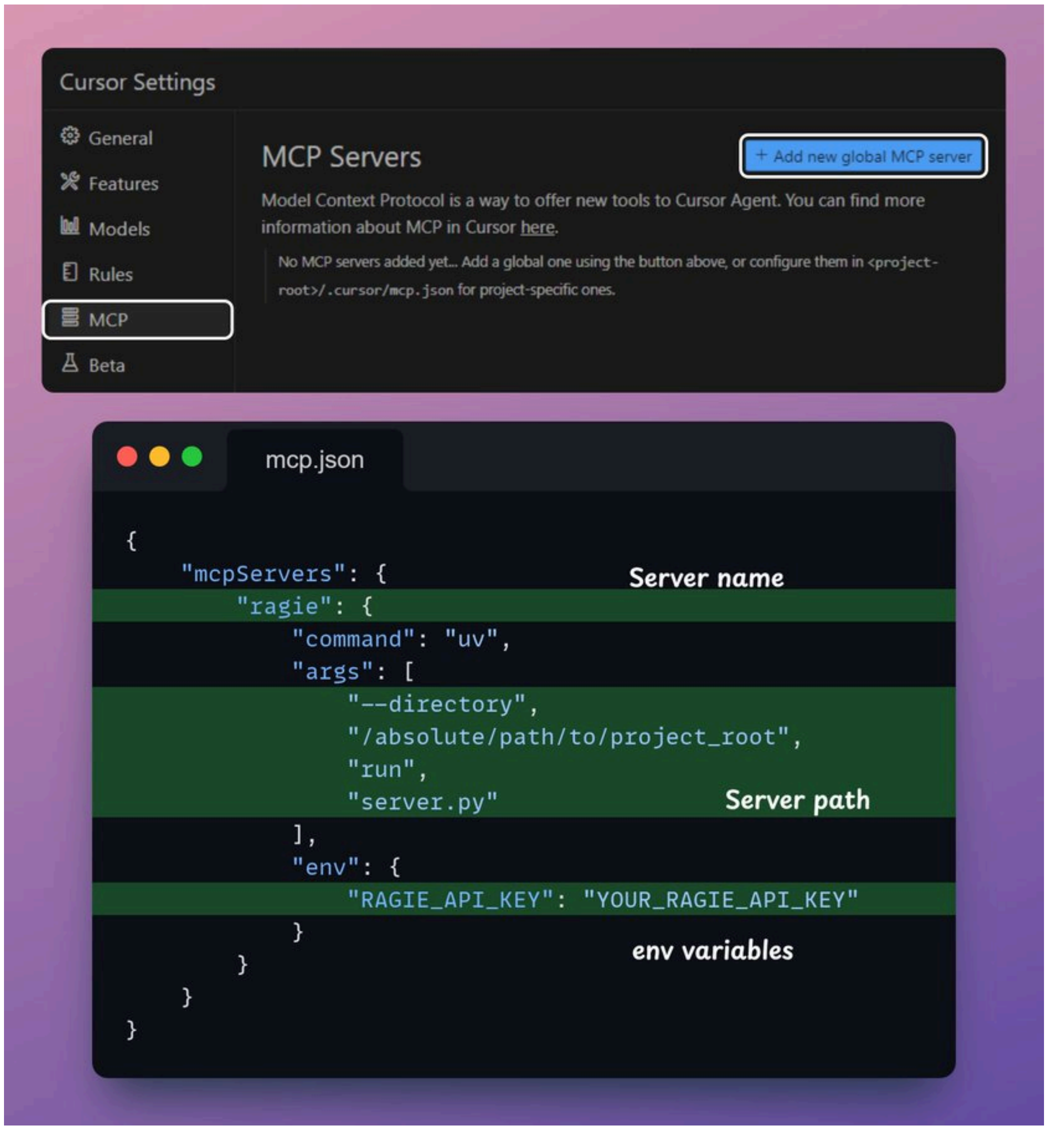
- ingest_data_tool: Ingests data into Ragie index
- retrieve_data_tool: Retrieves data based on the user query
- show_video_tool: Creates video chunks from the original video



```python
from mcp.server.fastmcp import FastMCP
from main import clear_index, ingest_data, retrieve_data, chunk_video

mcp = FastMCP("ragie")

@mcp.tool()
def ingest_data_tool(directory: str) -> None:
    """Loads data from a directory into the Ragie index"""
    clear_index()
    ingest_data(directory)
    return "Data loaded successfully"

@mcp.tool()
def retrieve_data_tool(query: str) -> list[dict]:
    """Retrieves data from the Ragie index based on the query"""
    content = retrieve_data(query)
    return content

@mcp.tool()
def show_video_tool(doc_name: str, t_start: float, t_end: float) -> str:
    """Creates and saves a video chunk based on start and end time"""
    chunk_video(doc_name, t_start, t_end)
    return "Video chunk created successfully"

# Run the server locally
if __name__ == "__main__":
    mcp.run(transport='stdio')
```

#4) Integrate MCP server with Cursor

To integrate the MCP server with Cursor, go to Settings → MCP → Add new global MCP server.



MCP Servers

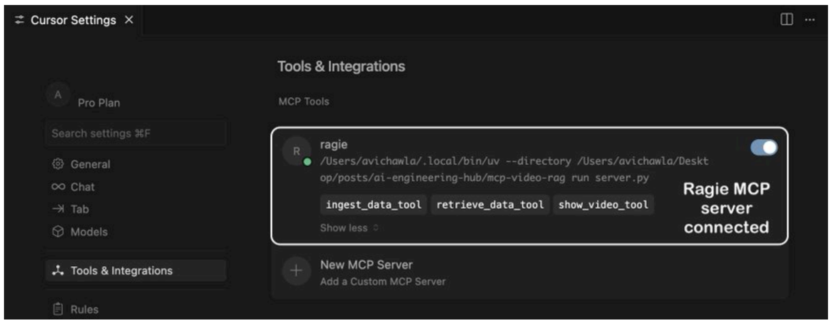
Model Context Protocol is a way to offer new tools to Cursor Agent. You can find more information about MCP in Cursor here.

No MCP servers added yet... Add a global one using the button above, or configure them in `<project-root>/.cursor/mcp.json` for project-specific ones.

```json
{
    "mcpServers": {
        "ragie": {
            "command": "uv",
            "args": [
                "--directory",
                "/absolute/path/to/project_root",
                "run",
                "server.py"
            ],
            "env": {
                "RAGIE_API_KEY": "YOUR_RAGIE_API_KEY"
            }
        }
    }
}
```

Done!

Your local Ragie MCP server is live and connected to Cursor!



ragie

```
/Users/avichawla/.local/bin/uv --directory /Users/avichawla/Deskt
op/posts/ai-engineering-hub/mcp-video-rag run server.py
```

`ingest_data_tool` `retrieve_data_tool` `show_video_tool`

Next, we interact with the MCP server through Cursor.

Based on the query, it can:

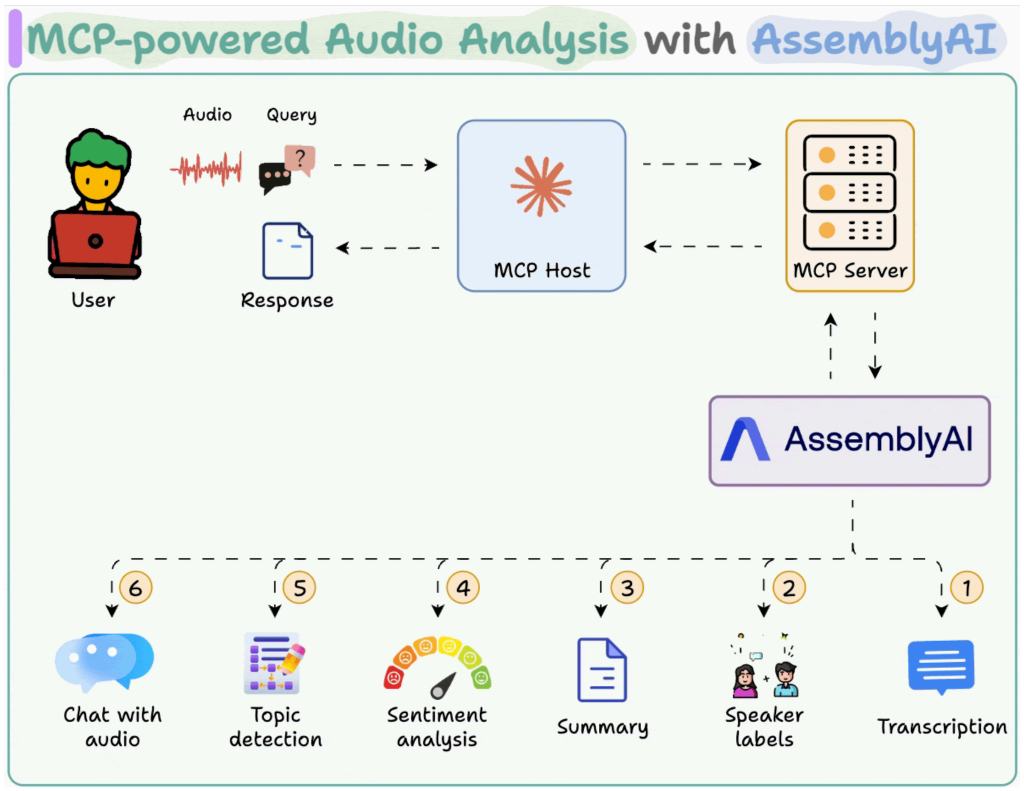
- Ingest a new video into the Ragie Index.
- Fetch detailed information about an existing video.
- Retrieve the video segment where a specific event occurred.

And that was your MCP-powered video RAG.

The code is available here: https://www.dailydoseofds.com/p/build-an-mcp-powered-rag-over-videos/

#11) MCP-powered Audio Analysis Toolkit

We have an MCP-driven audio analysis toolkit that accepts an audio file and lets you transcribe it and extract insights such as sentiment analysis, speaker labels, summary and topic detection. It also lets you chat with audio.



Tech stack

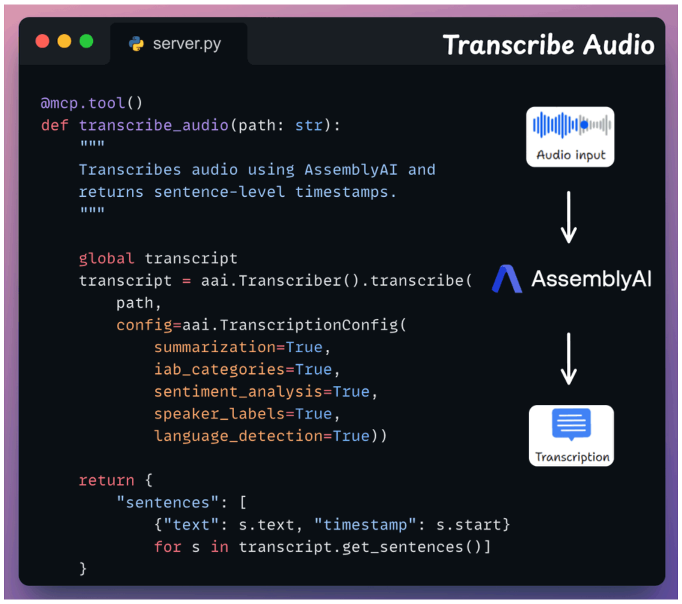
- AssemblyAI for transcription and audio analysis.
- Claude Desktop as the MCP host.
- Streamlit for the UI

Workflow

- User's audio input is sent to AssemblyAI via a local MCP server.
- AssemblyAI transcribes it while providing the summary, speaker labels, sentiment, and topics.
- Post-transcription, the user can also chat with audio.

#1) Transcription MCP tool

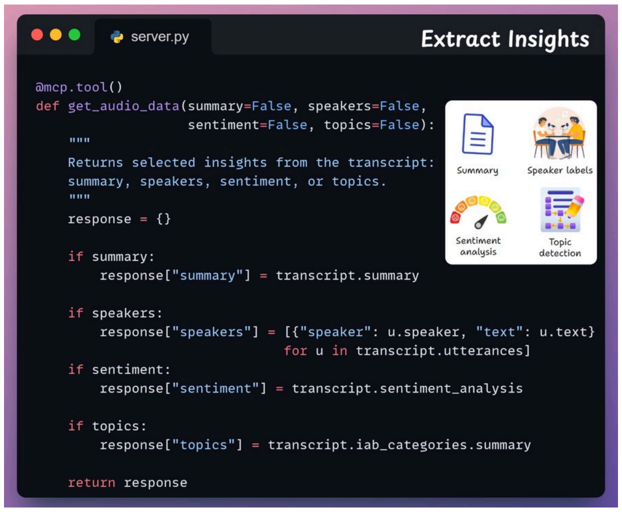
This tool accepts an audio input from the user and transcribes it using AssemblyAI. We also store the full transcript to use in the next tool.



```python
@mcp.tool()
def transcribe_audio(path: str):
    """
    Transcribes audio using AssemblyAI and
    returns sentence-level timestamps.
    """

    global transcript
    transcript = aai.Transcriber().transcribe(
        path,
        config=aai.TranscriptionConfig(
            summarization=True,
            iab_categories=True,
            sentiment_analysis=True,
            speaker_labels=True,
            language_detection=True))

    return {
        "sentences": [
            {"text": s.text, "timestamp": s.start}
            for s in transcript.get_sentences()]
    }
```

#2) Audio analysis tool

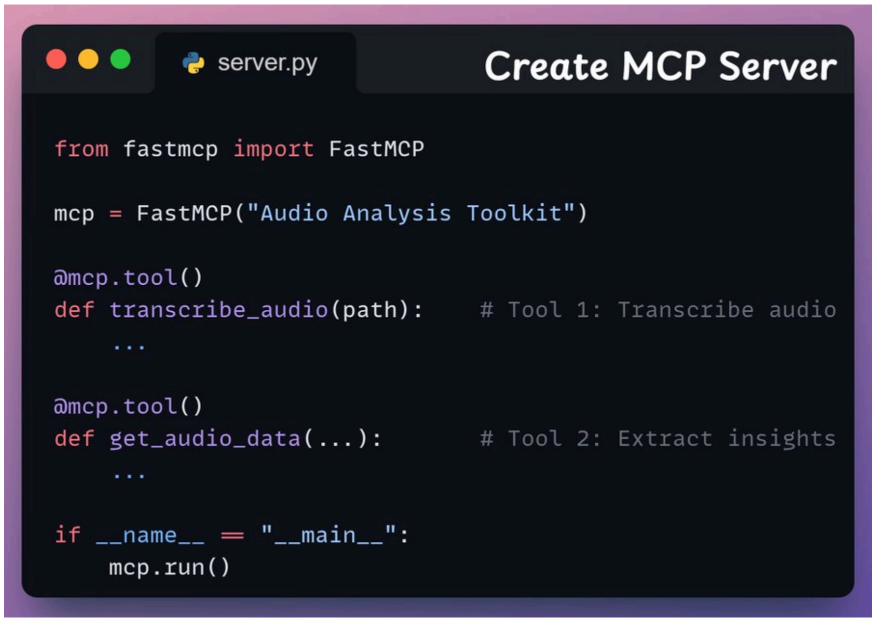
Next, we have a tool that returns specific insights from the transcript, like speaker labels, sentiment, topics, and summary.



```python
@mcp.tool()
def get_audio_data(summary=False, speakers=False,
                   sentiment=False, topics=False):
    """
    Returns selected insights from the transcript:
    summary, speakers, sentiment, or topics.
    """
    response = {}

    if summary:
        response["summary"] = transcript.summary

    if speakers:
        response["speakers"] = [{"speaker": u.speaker, "text": u.text}
                                for u in transcript.utterances]
    if sentiment:
        response["sentiment"] = transcript.sentiment_analysis

    if topics:
        response["topics"] = transcript.iab_categories.summary

    return response
```

#3) Create MCP Server

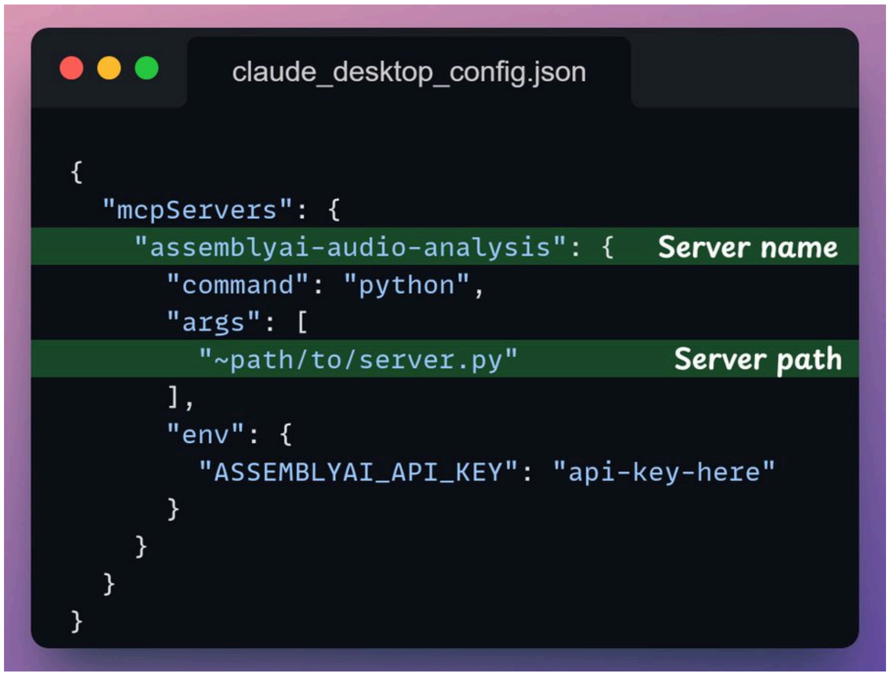
Now, we'll set up an MCP server to use the tools we created above.



```python
from fastmcp import FastMCP

mcp = FastMCP("Audio Analysis Toolkit")

@mcp.tool()
def transcribe_audio(path):     # Tool 1: Transcribe audio
    ...

@mcp.tool()
def get_audio_data(...):        # Tool 2: Extract insights
    ...

if __name__ == "__main__":
    mcp.run()
```

#4) Integrate MCP server with Claude Desktop

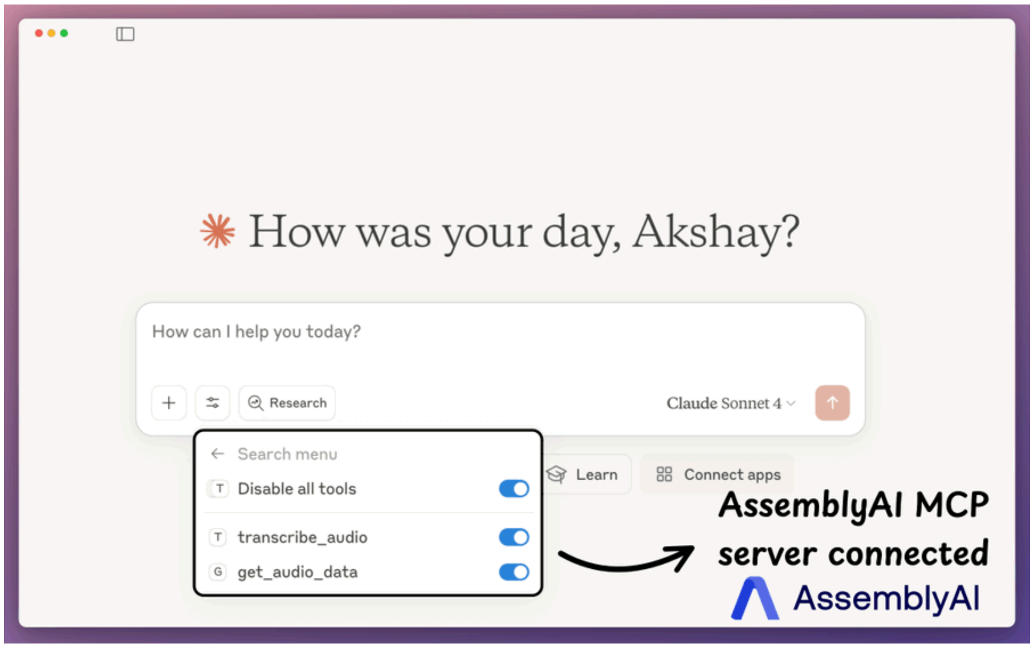
Go to File → Settings → Developer → Edit Config and add the following code.



```json
{
  "mcpServers": {
    "assemblyai-audio-analysis": {
      "command": "python",
      "args": [
        "~path/to/server.py"
      ],
      "env": {
        "ASSEMBLYAI_API_KEY": "api-key-here"
      }
    }
  }
}
```

Once the server is configured, Claude Desktop will show the two tools we built above in the tools menu:

- transcribe_audio
- get_audio_data



And that was our MCP-powered audio analysis toolkit!

For accessibility, we have created a Streamlit UI for the audio analysis app.

You can upload the audio, extract insights, and chat with it using AssemblyAI's LeMUR. Find the code below.

The code is available here: https://www.dailydoseofds.com/p/hands-on-build-an-mcp-powered-audio-analysis-toolkit/