2025 EDITION

FREE

MCP

THE ILLUSTRATED GUIDEBOOK

Daily Dose of Data Science

Avi Chawla & Akshay Pachaar
DailyDoseofDS.com

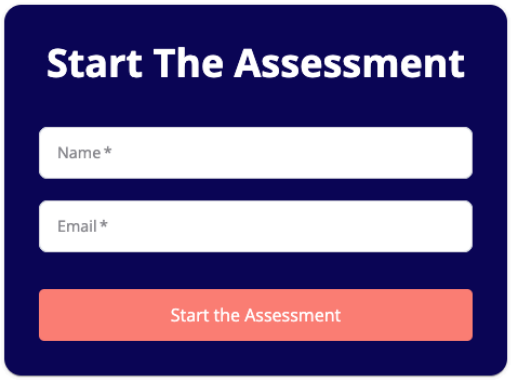
How to make the most out of this book and your time?

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.

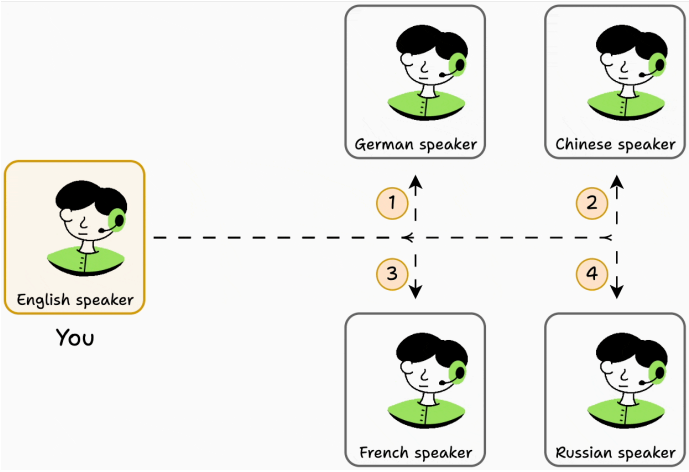
https://bit.ly/mcp-assessment

Table of contents

Model Context Protocol (MCP)

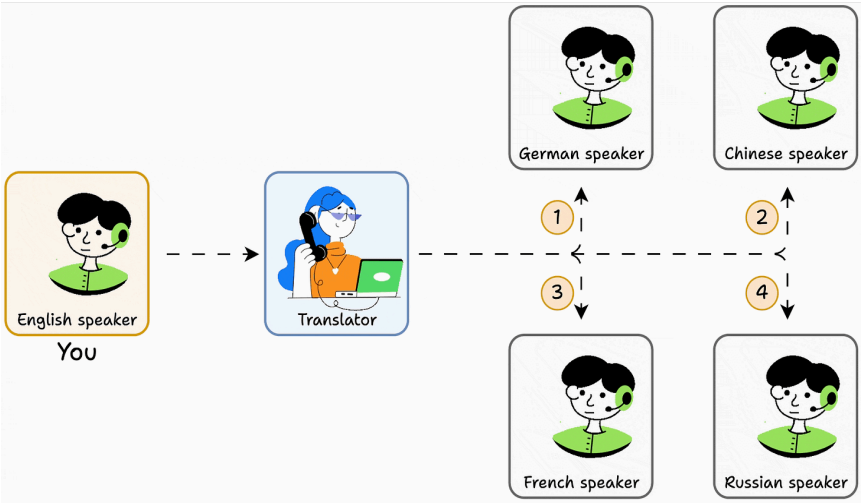
What is MCP?

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?

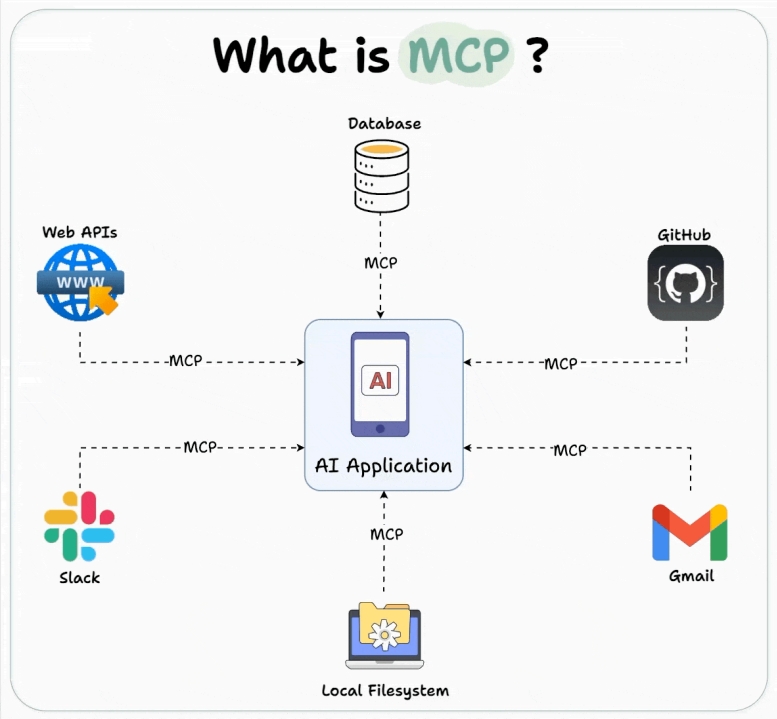


This is simple, isn't it?

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.

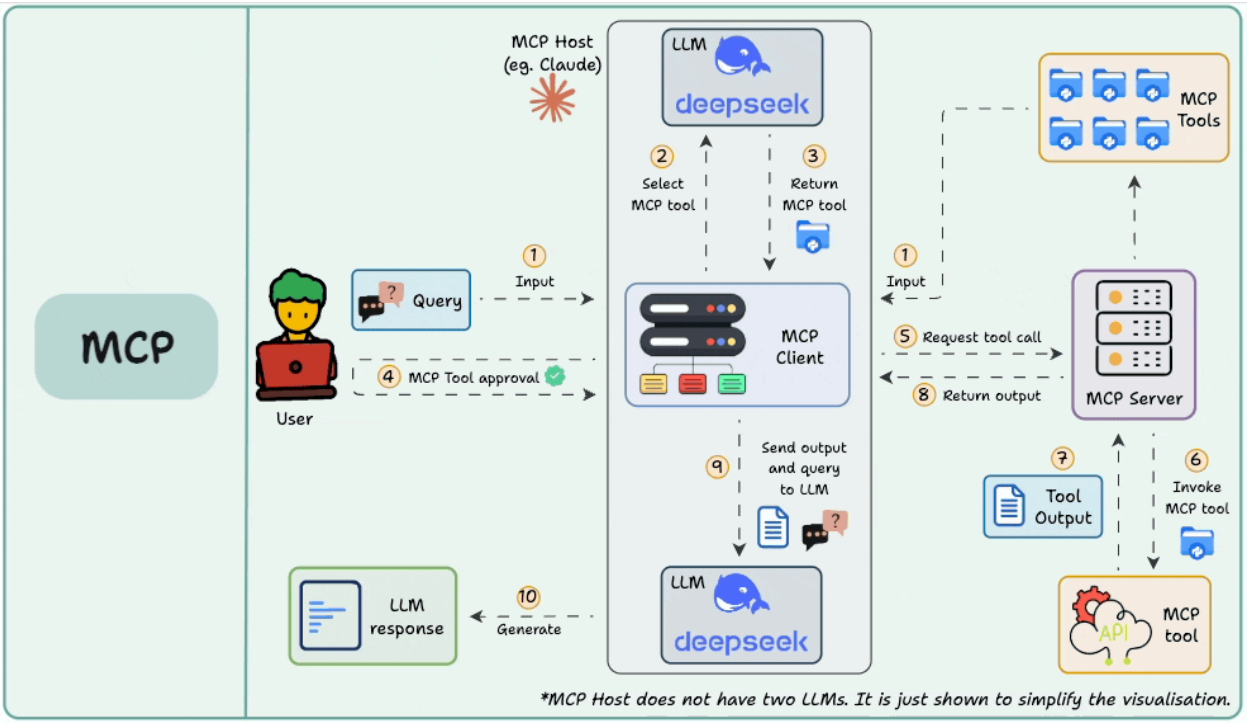
MCP acts as a universal connector for AI systems to capabilities (tools, etc.), similar to how USB-C standardizes connections between electronic devices.

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

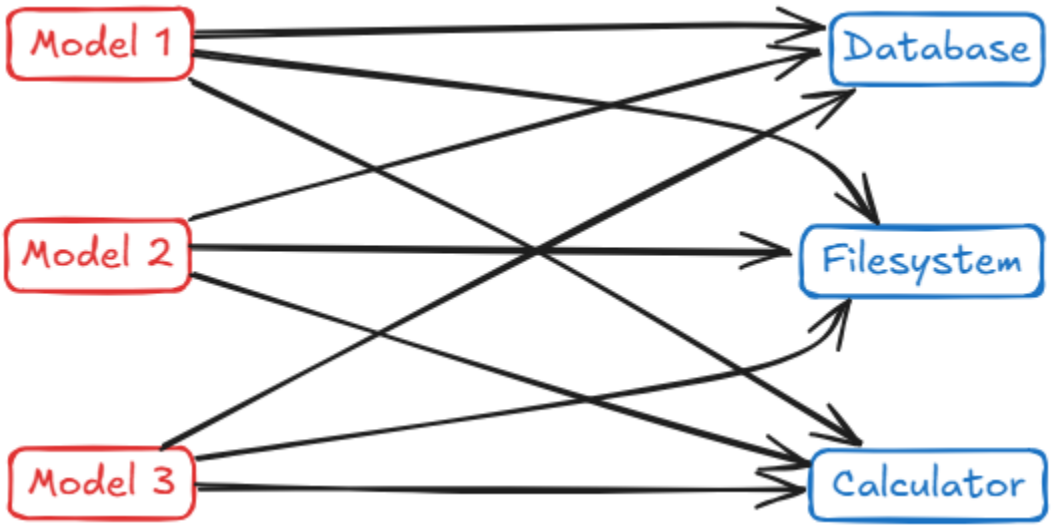
Before MCP, the landscape of connecting AI to external data and actions looked like a patchwork of one-off solutions.

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.



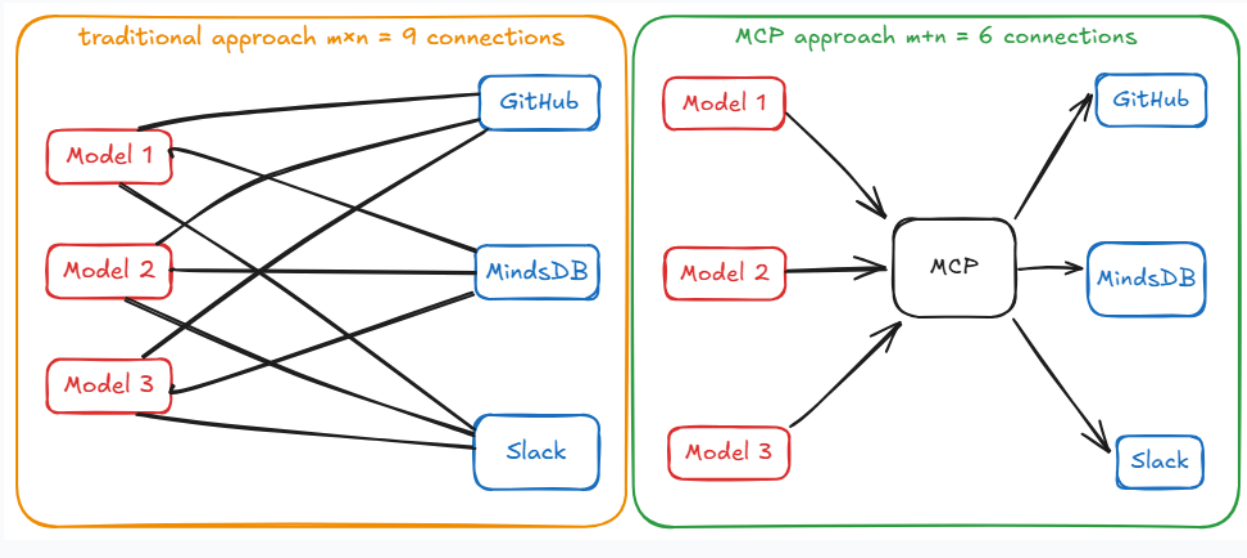
The solution

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.

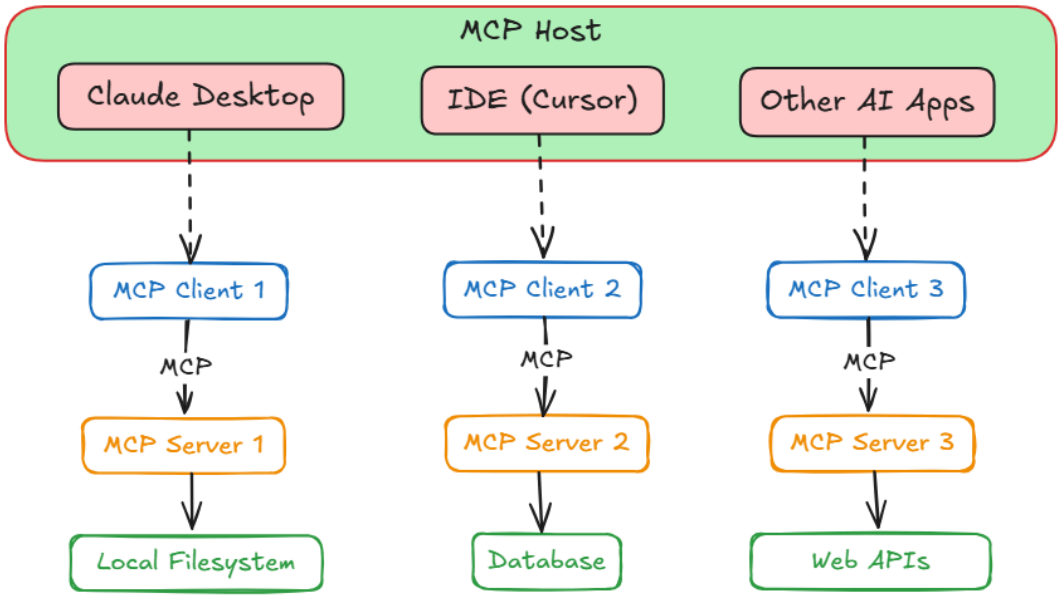


- On the left (pre-MCP), every model had to wire into every tool.
- On the right (with MCP), each model and tool connects to the MCP layer, drastically simplifying connections. You can also relate this to the translator example we discussed earlier.

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.

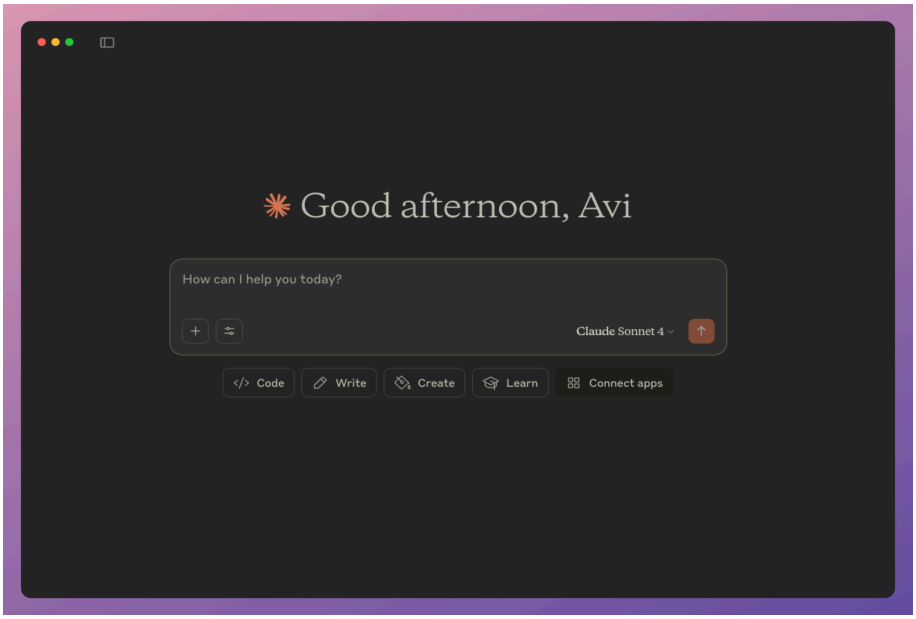


Host

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.

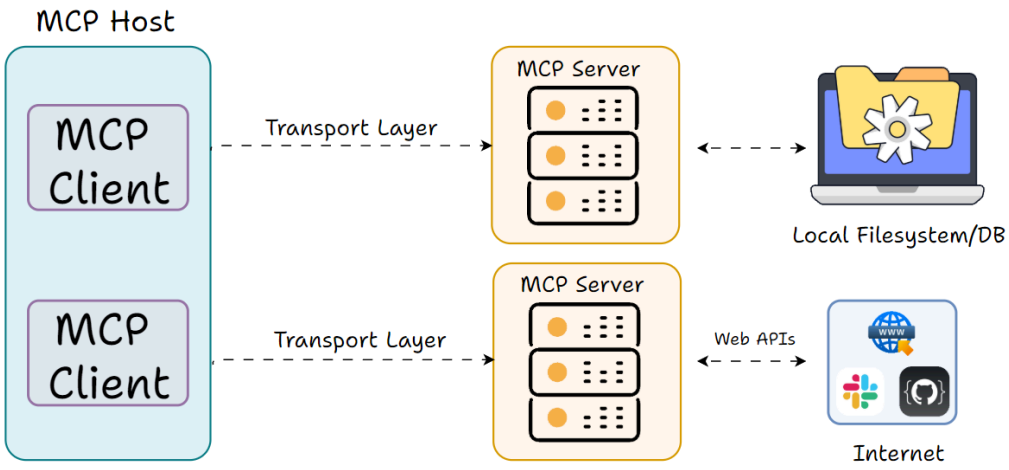
Host is the one that initiates connections to the available MCP servers when the system needs them. It captures the user's input, keeps the conversation history, and displays the model's replies.



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.

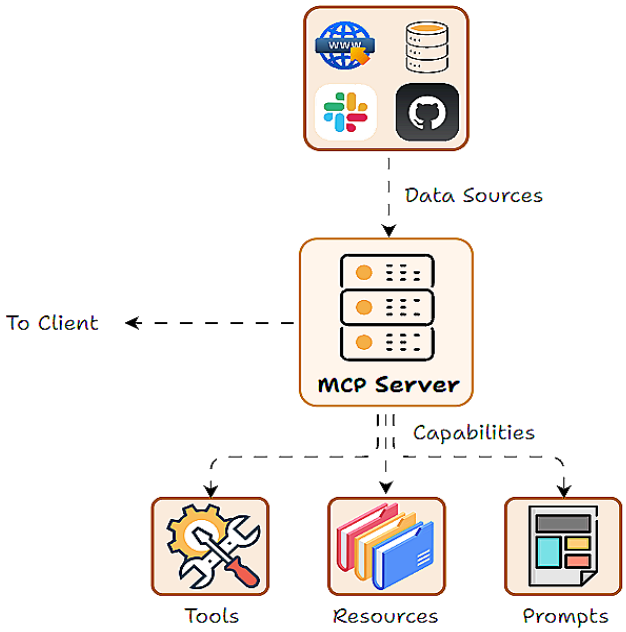


Server

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.

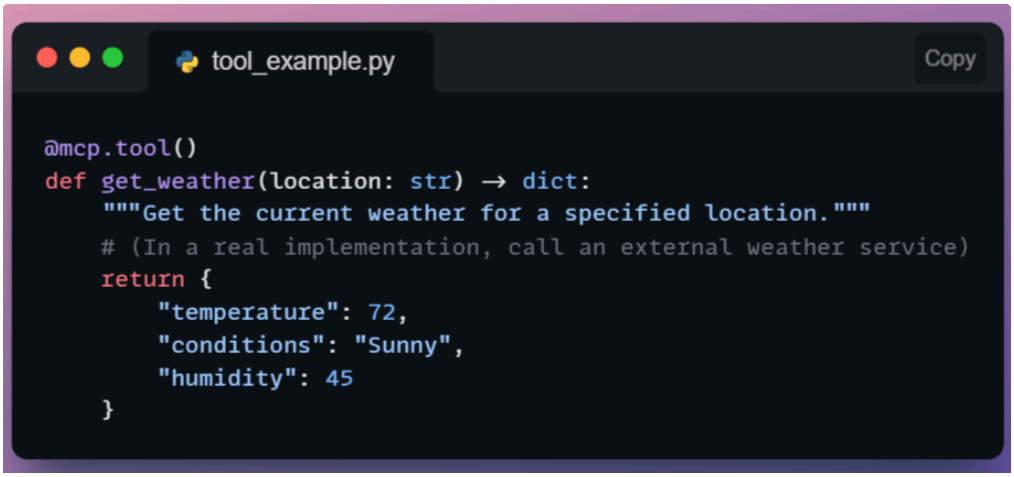
- Tools: Executable actions or functions that the AI (host/client) can invoke (often with side effects or external API calls).
- Resources: Read-only data sources that the AI (host/client) can query for information (no side effects, just retrieval).
- Prompts: Predefined prompt templates or workflows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

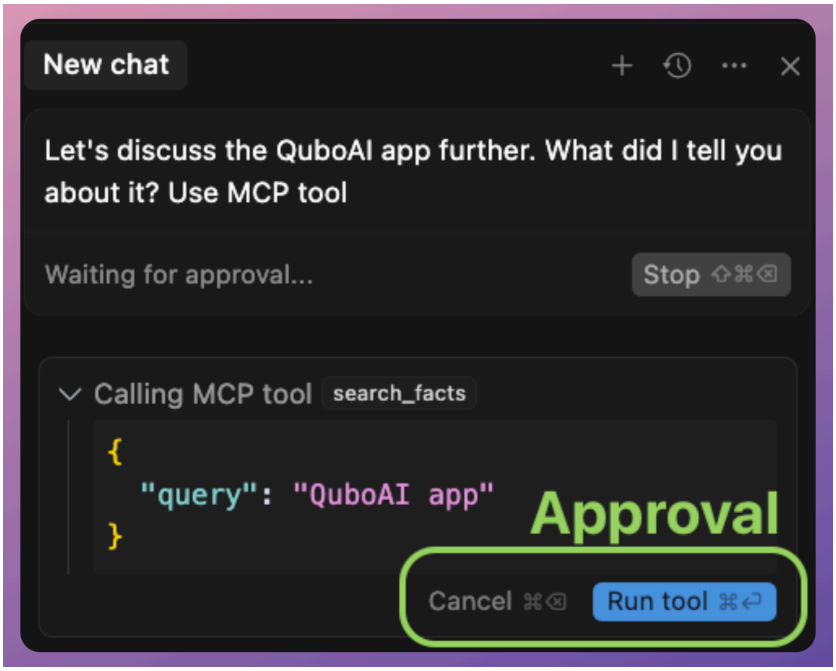


This Python function, registered with @mcp.tool(), can be invoked by the AI via MCP.

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.



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.

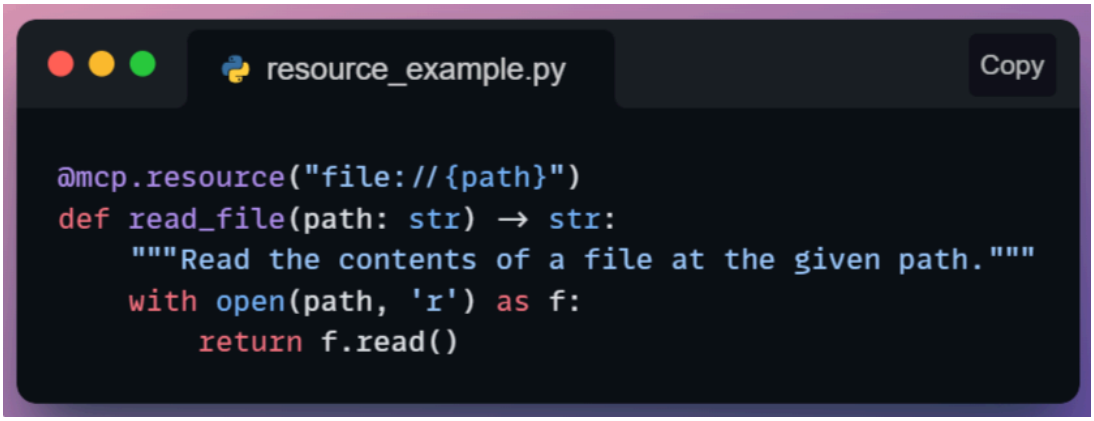
Another key difference is that resources are usually accessed under the host application's control (not spontaneously by the model). In practice, this might mean the Host knows when to fetch a certain context for the model.

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:



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:

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.

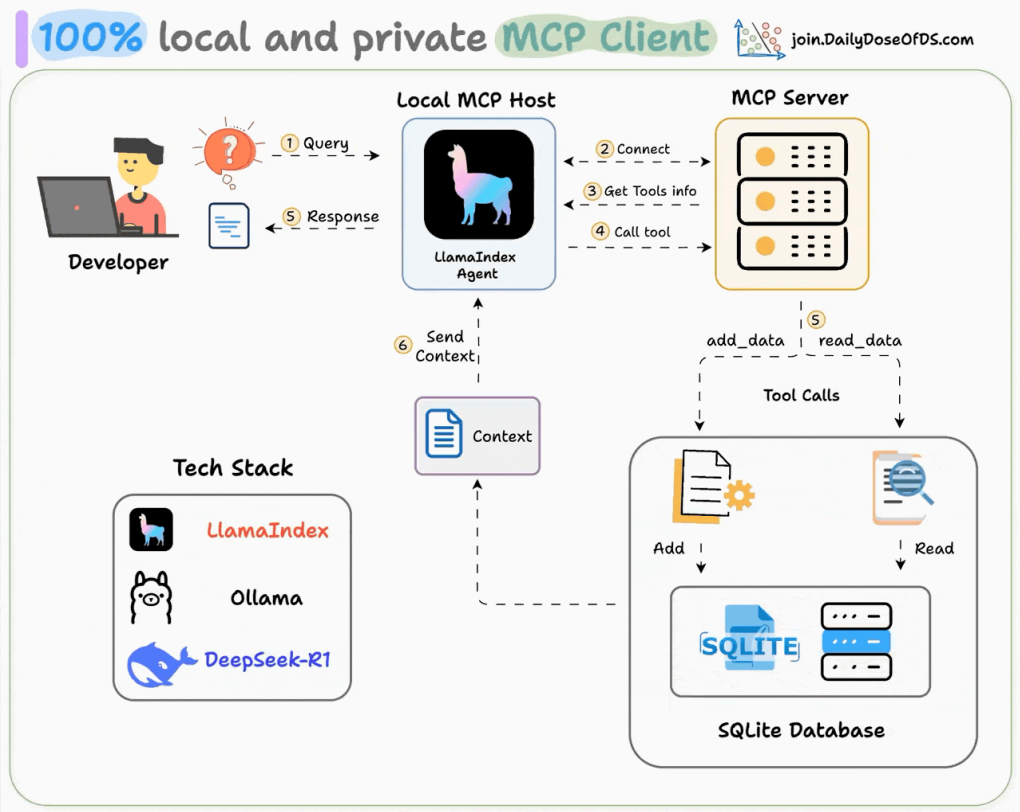
They represent best practices or predefined strategies for the AI to use.

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.

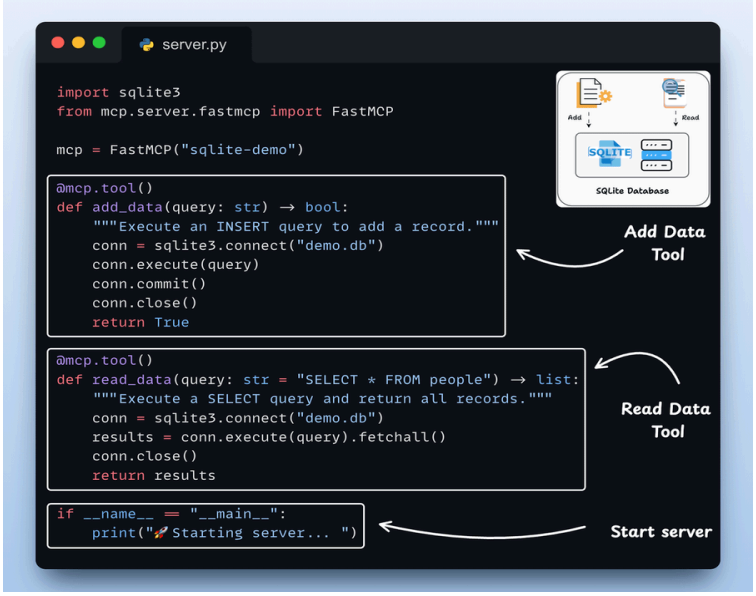
Let's implement this!

#1) Build an SQLite MCP Server

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

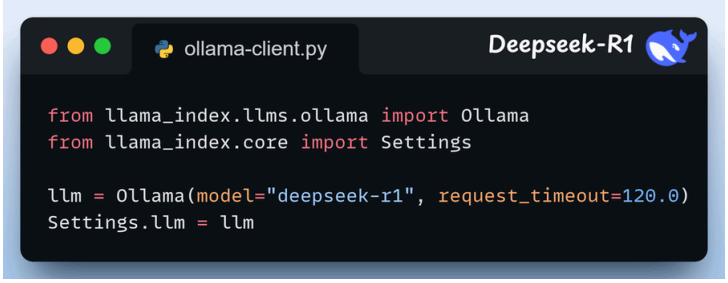
- add data
- fetch data

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



#2) Set Up LLM

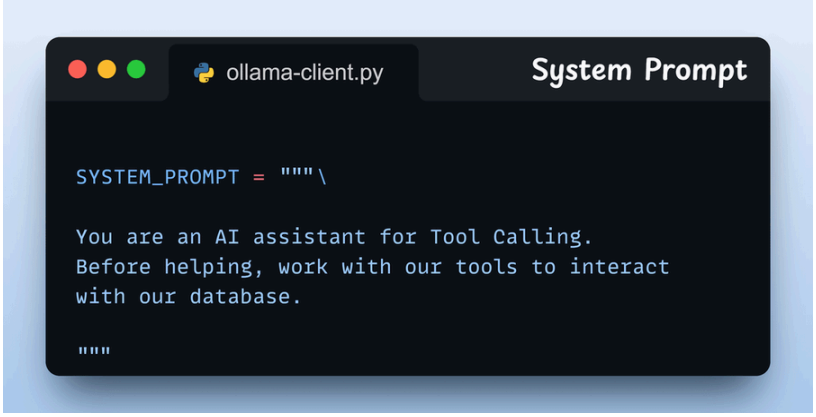
We'll use a locally served Deepseek-R1 via Ollama as the LLM for our MCP-powered agent.



#3) Define system prompt

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

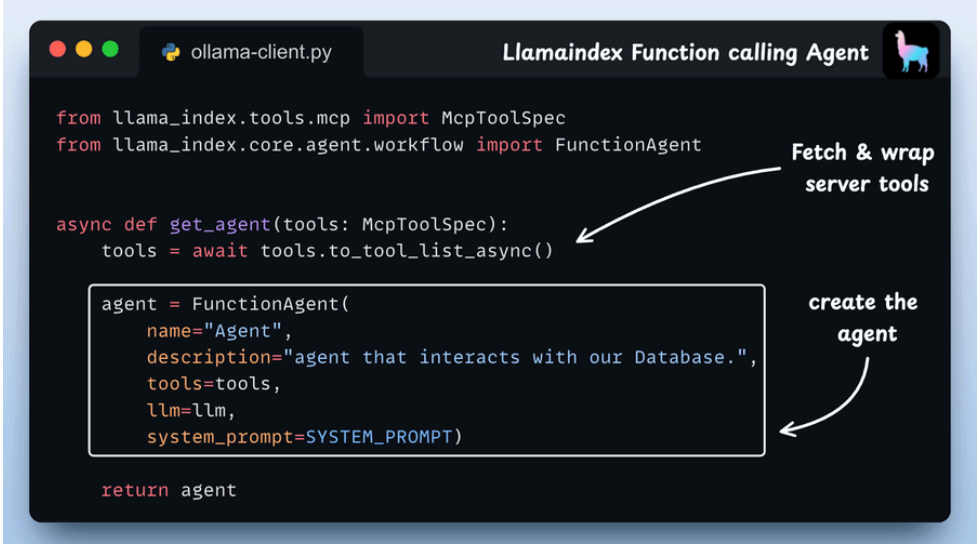
Feel free to tweak this on a need basis.



#4) Define the Agent

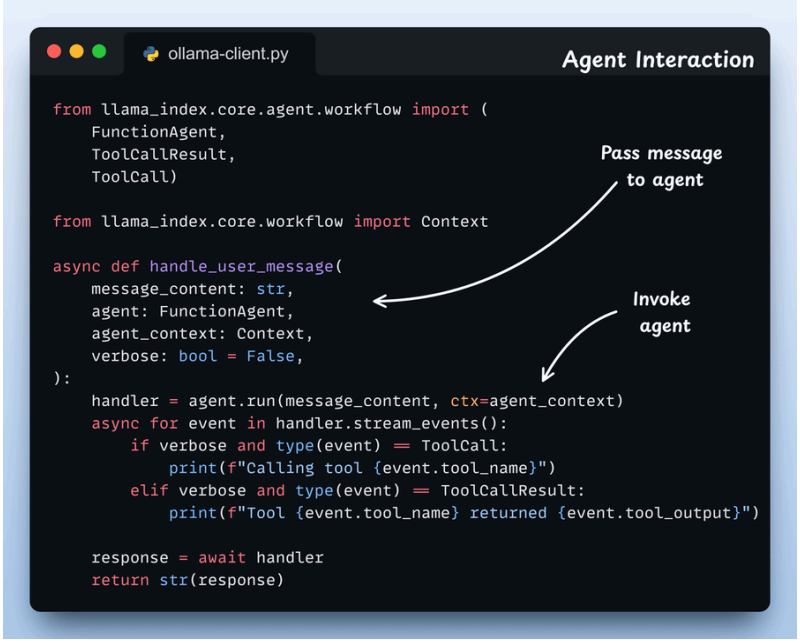
We define a function that builds a typical LlamaIndex agent with its appropriate arguments.

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



#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.

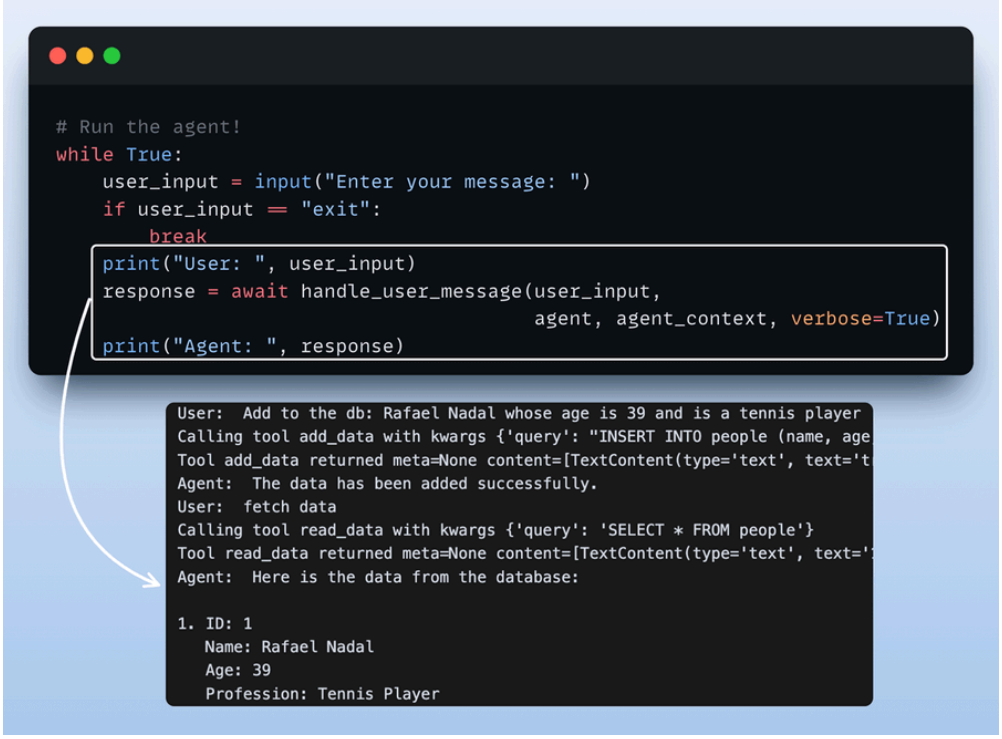


#6) Initialize MCP Client and the Agent

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.

#7) Run the Agent:

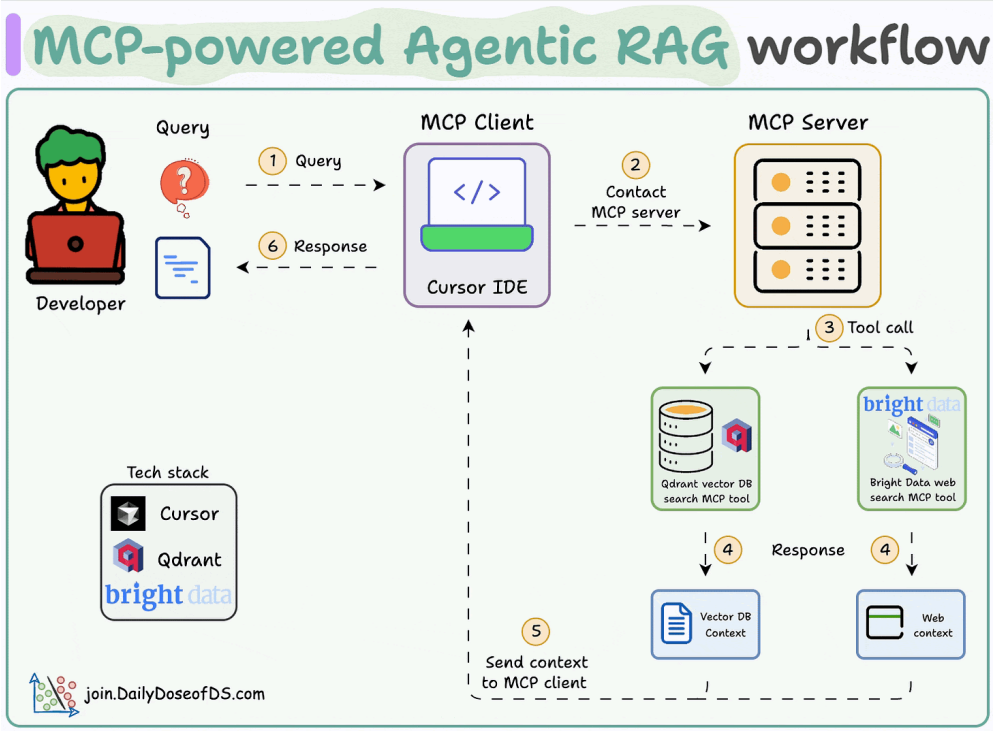
Finally, we start interacting with our agent and get access to the tools from our SQLite MCP server.



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.

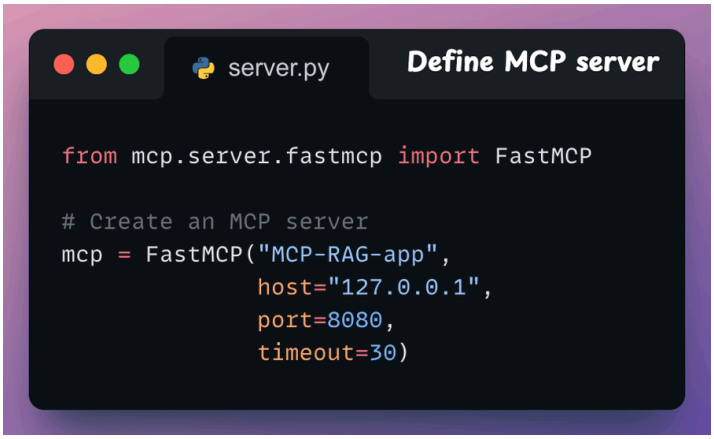
Workflow:

- 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.



#2) Vector DB MCP tool

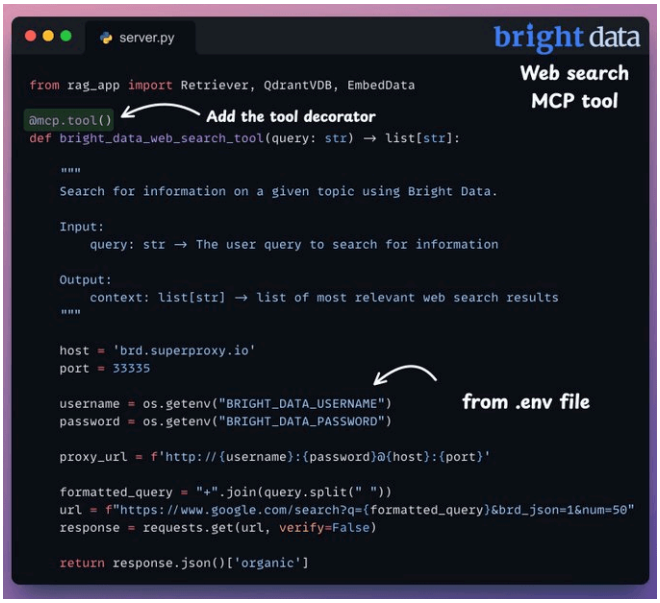
A tool exposed through an MCP server has two requirements:

- 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.

#3) Web search MCP tool

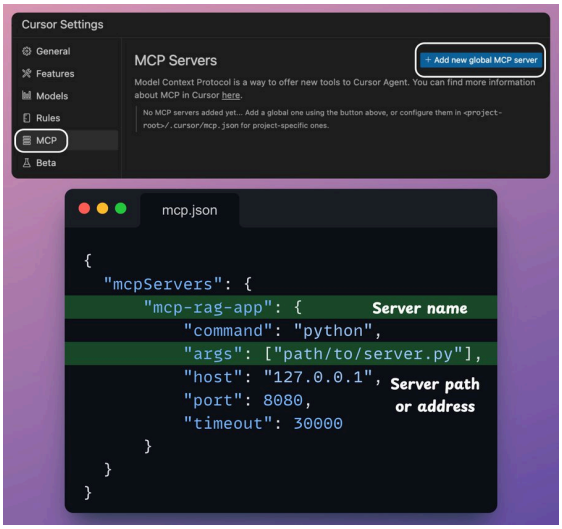
If query is unrelated to ML, we resort to web search using Bright Data's SERP API to scrape data at scale across several sources to get relevant context.




```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

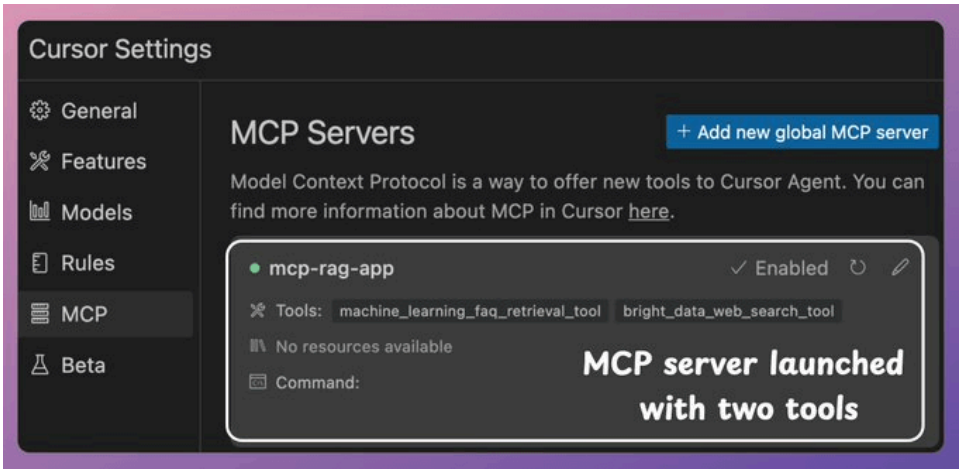



```json
{
  "mcpServers": {
      "mcp-rag-app": {
          "command": "python",
          "args": ["path/to/server.py"],
          "host": "127.0.0.1",
          "port": 8080,
          "timeout": 30000
      }
  }
}
```

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.



Next, we interact with the MCP server.

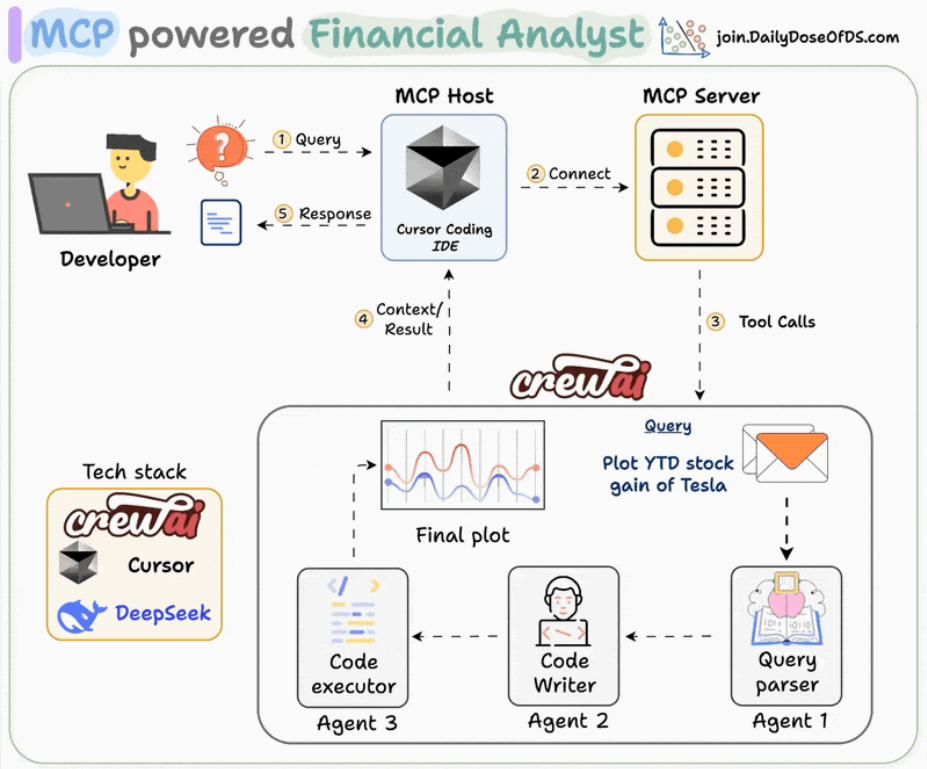
- When we ask an ML-related query, it invokes the vector DB tool.
- But when we ask a general query, it invokes the Bright Data web search tool to gather web data at scale from various sources.

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:

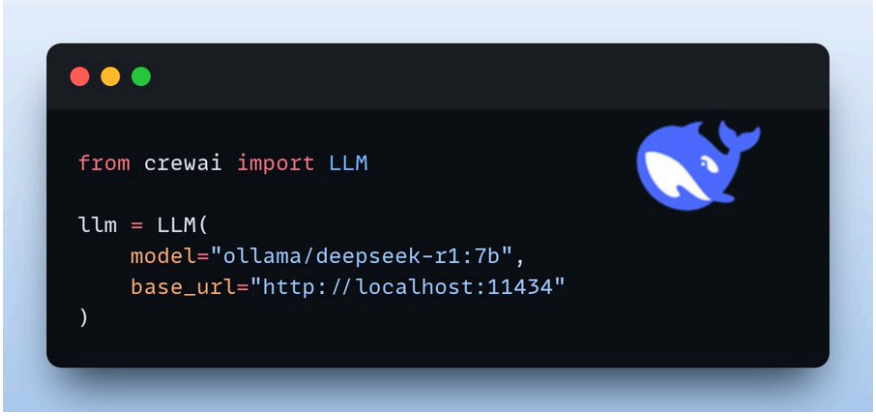
- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek-R1 LLM
- Cursor as the MCP host

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

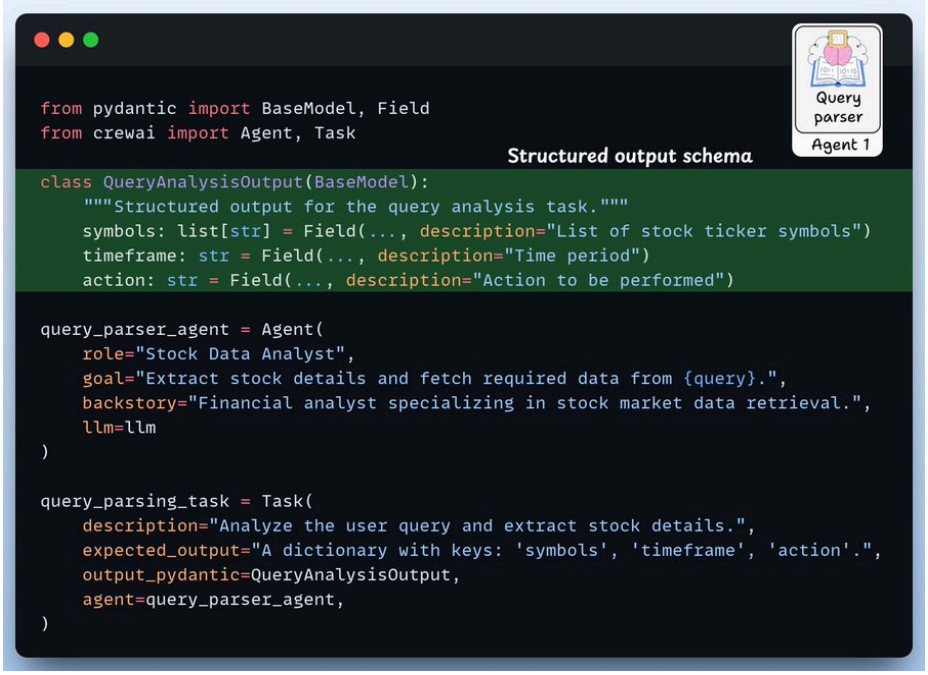
We will use Deepseek-R1 as the LLM, served locally using Ollama.



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!



#3) Code Writer Agent

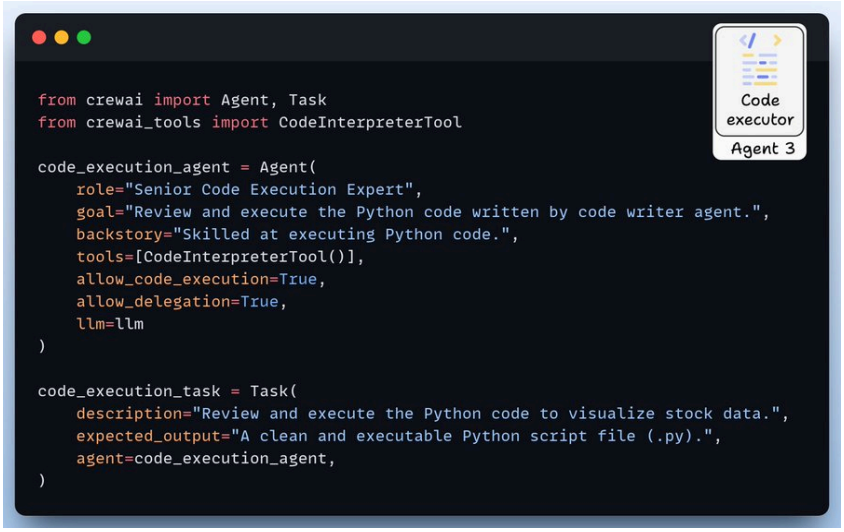
This agent writes Python code to visualize stock data using Pandas, Matplotlib, and Yahoo Finance libraries.


```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.

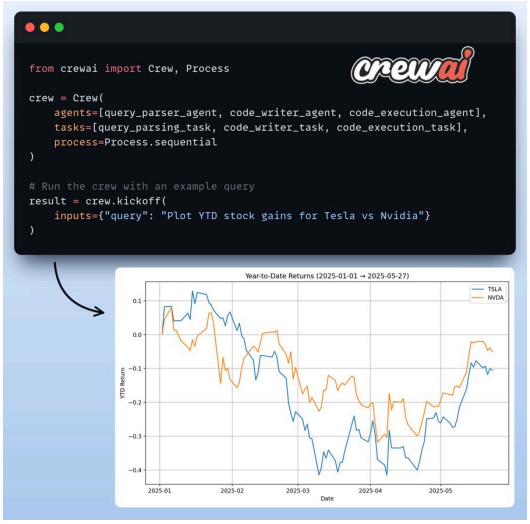
It uses the code interpreter tool by CrewAI to execute the code in a secure sandbox environment.




```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!



#6) Create MCP Server

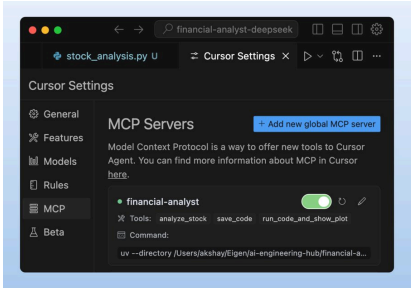
Now, we encapsulate our financial analyst within an MCP tool and add two more tools to enhance the user experience.

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

#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

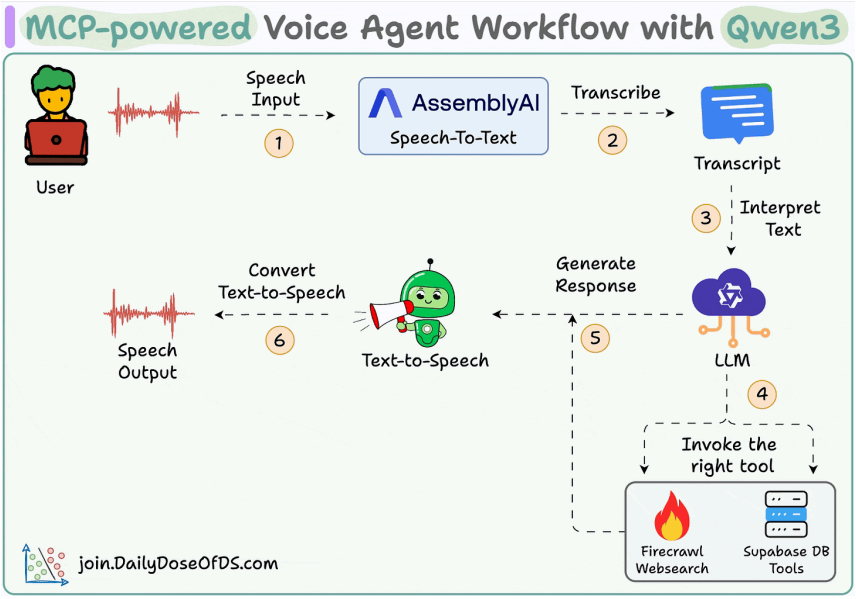
Done! Our financial analyst MCP server is live and connected to Cursor.



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.

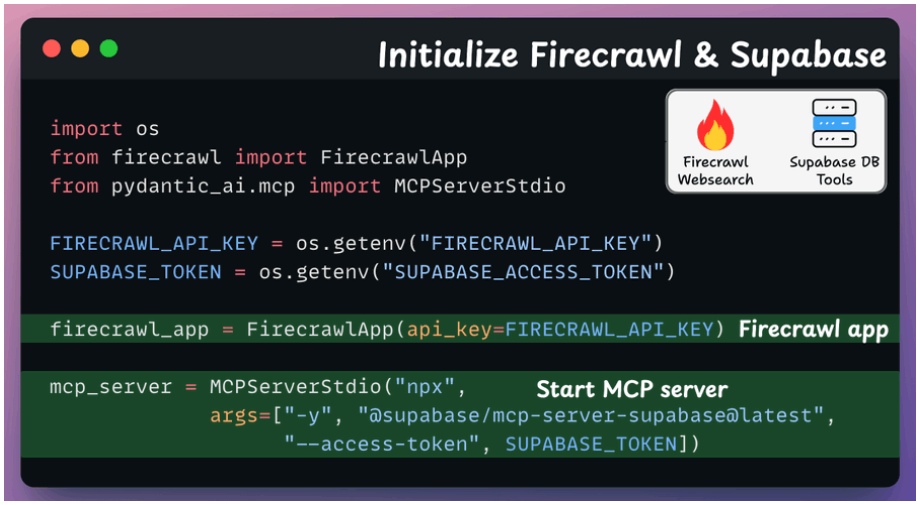
Workflow:

- 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!

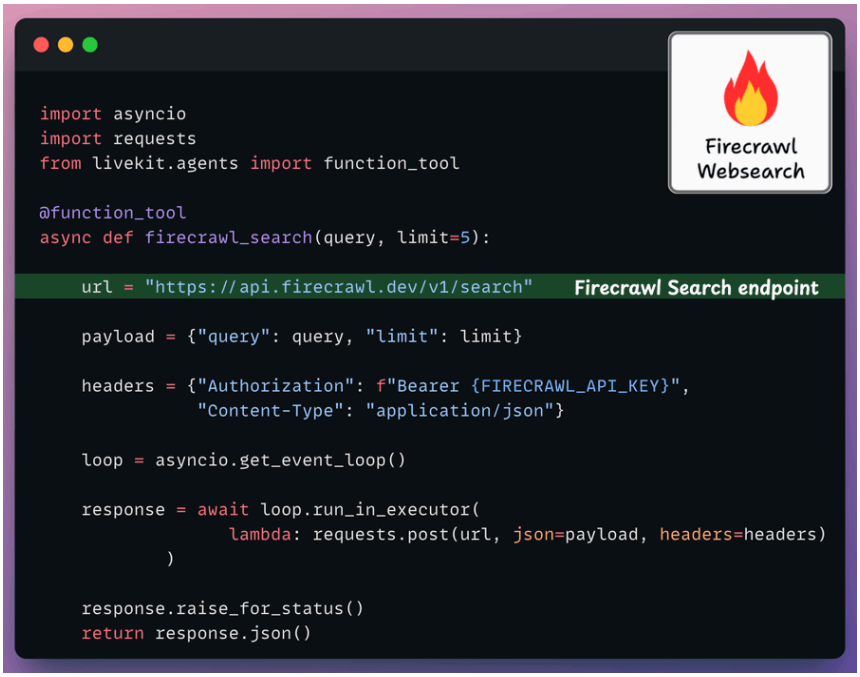
#1) Initialize Firecrawl & Supabase

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



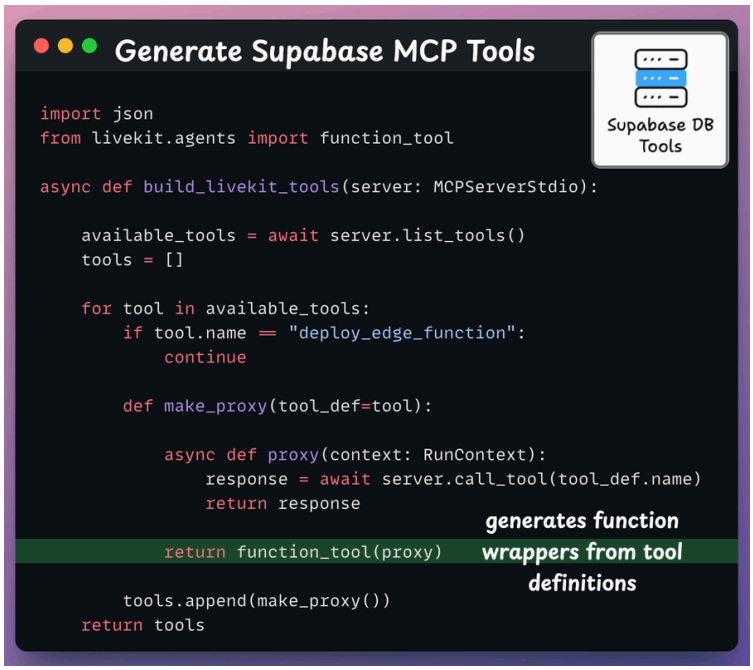
#2) Define web search tool

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



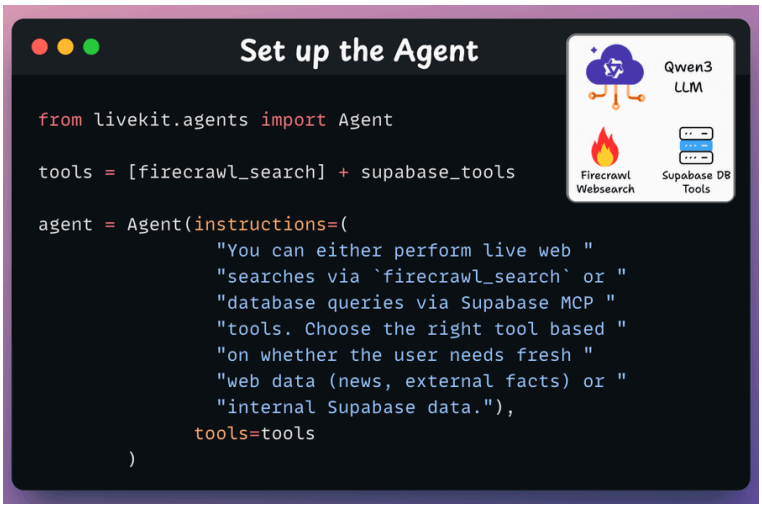
#3) Get Supabase MCP Tools

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



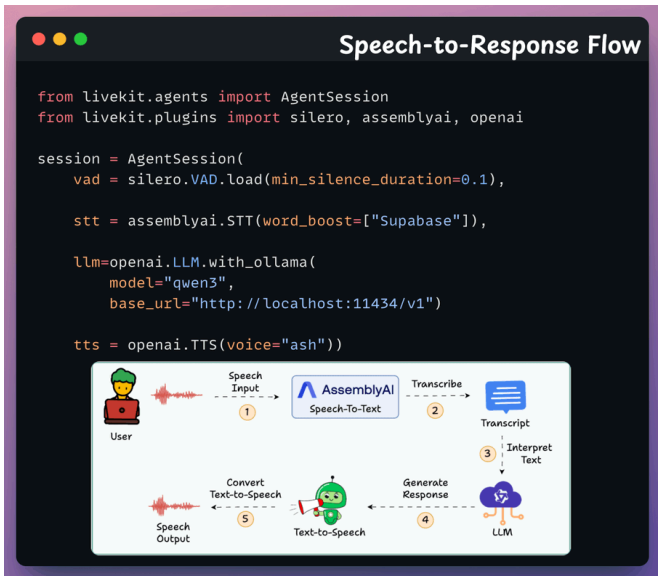
#4) Build the Agent

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.



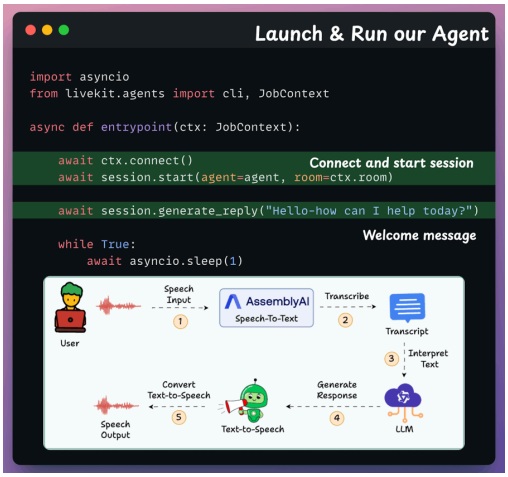
#5) Configure Speech-to-Response flow

- 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.



#6) Launch the Agent

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



Done!

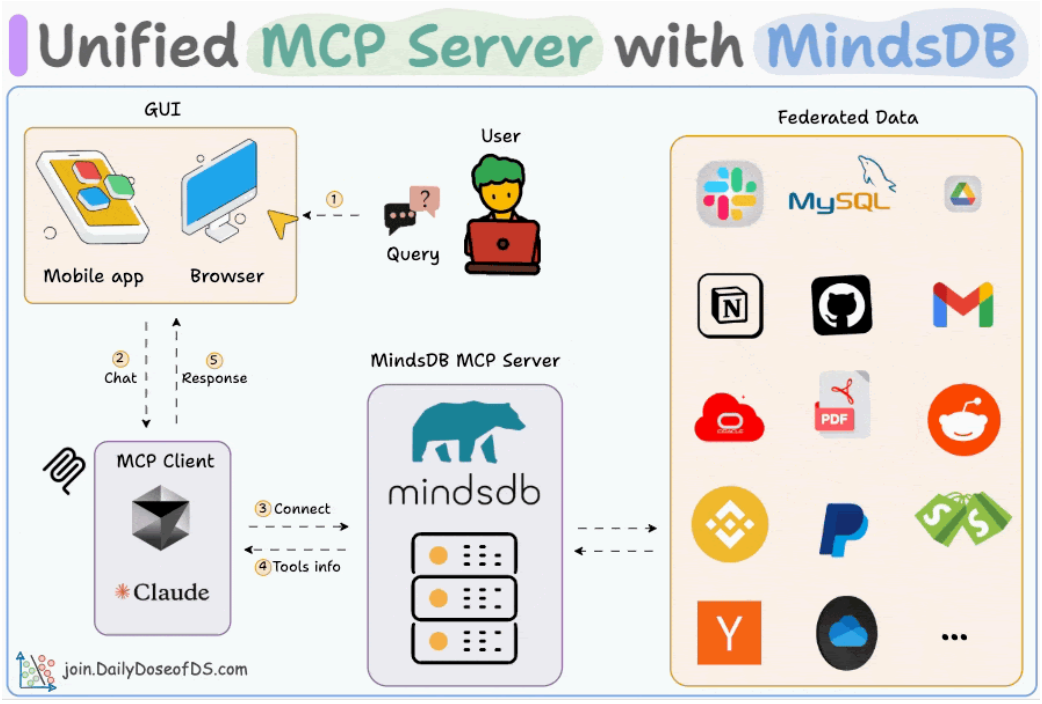
Our MCP-powered Voice Agent is ready.

- 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

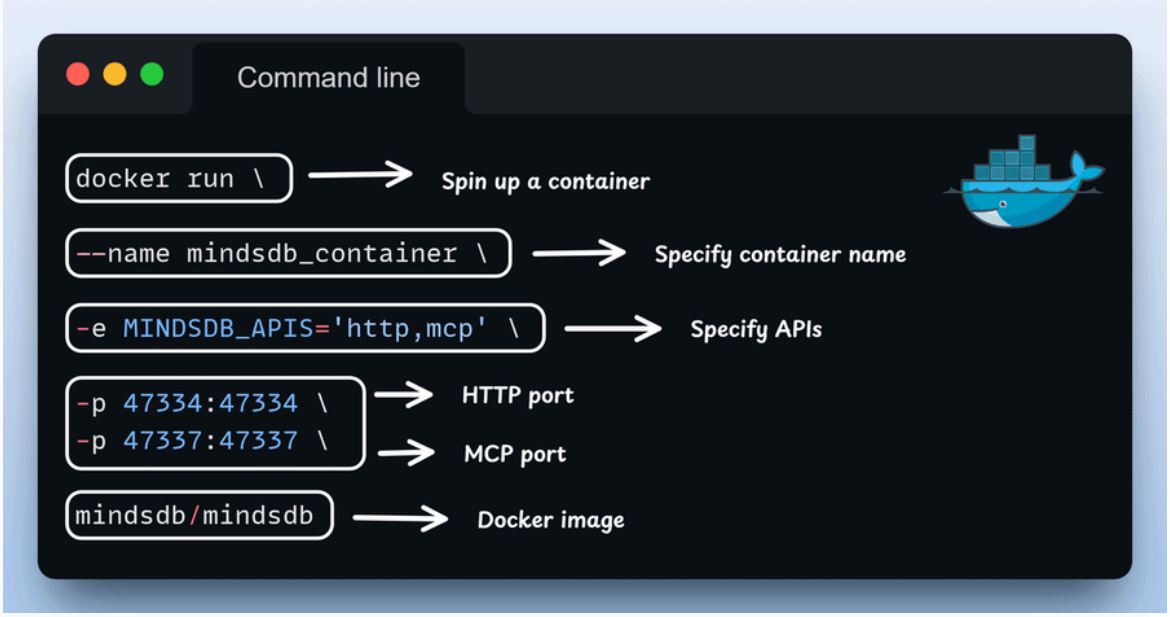
- User submits a query
- Agent connects to the MindsDB MCP server to find tools
- Selects the appropriate tool based on the user query and calls it
- Finally, returns a contextually relevant response

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.



#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

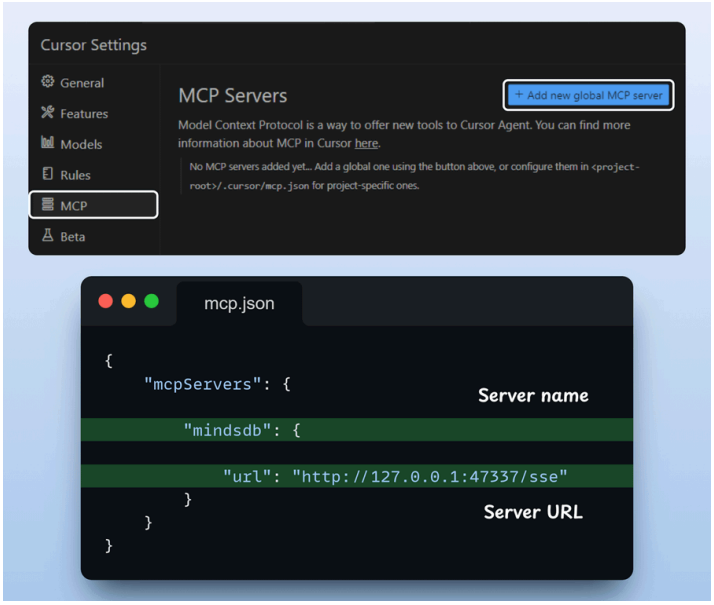
Let's start building our federated query engine by connecting our data sources to MindsDB.

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

#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.

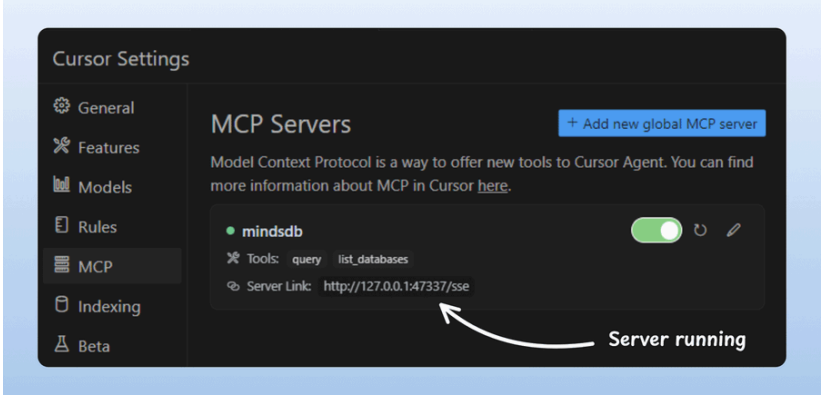
Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server. In the JSON file, add the following



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.

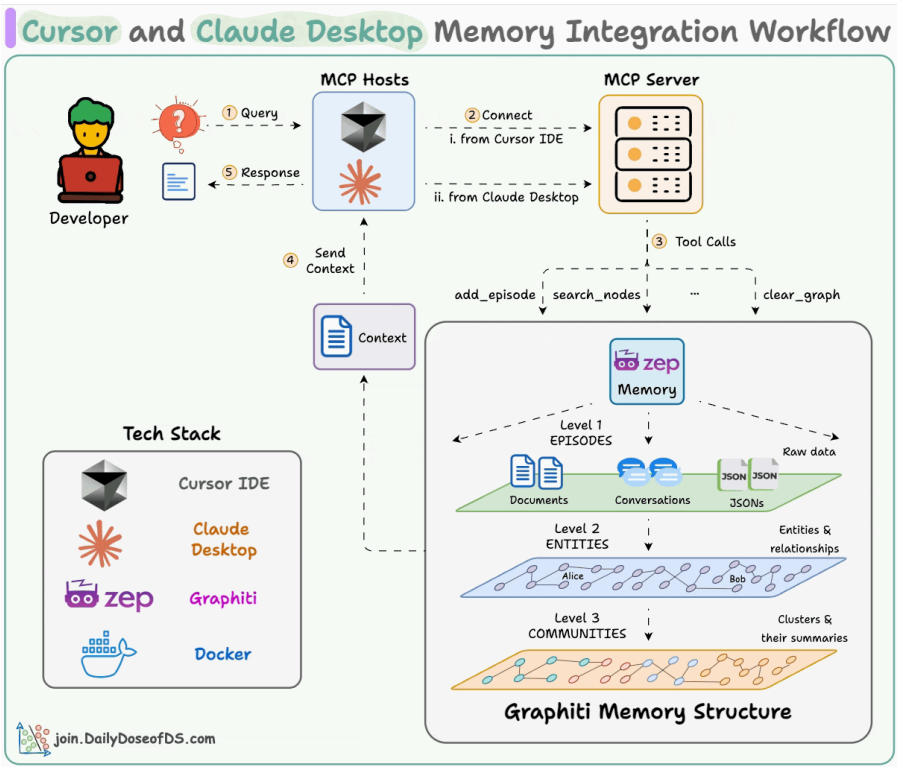


Apart from Claude and Cursor, MindsDB MCP server also works with the new OpenAI MCP integration.

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

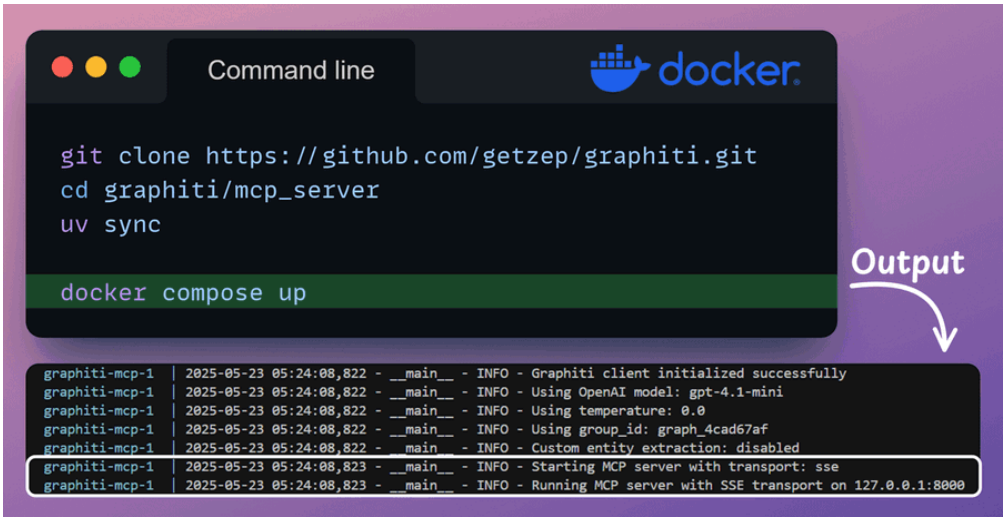
- Zep's Graphiti MCP as a memory layer for AI Agents.
- Cursor and Claude as the MCP hosts.

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.

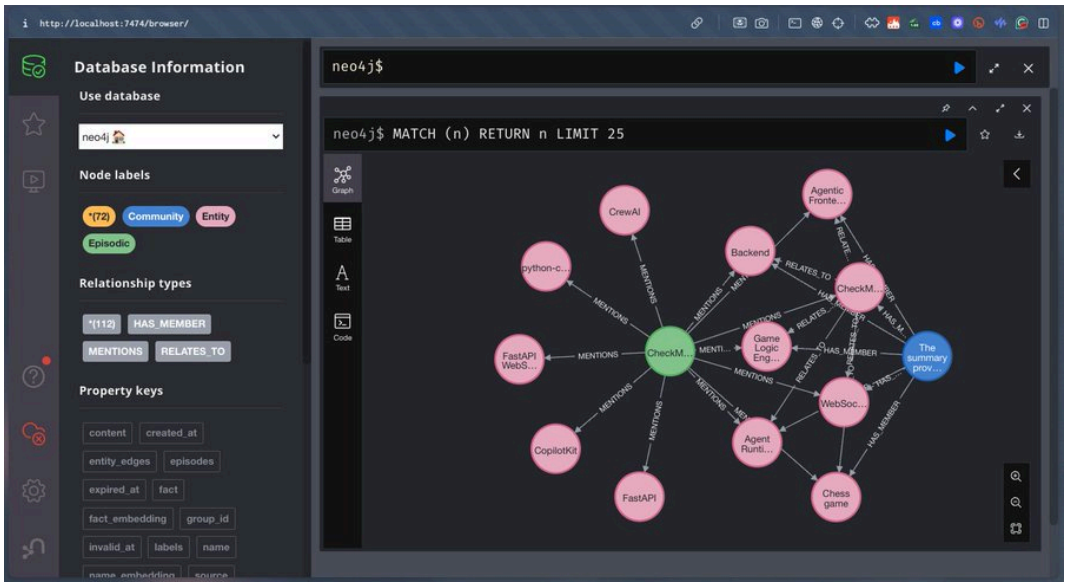
#1) Docker Setup

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



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

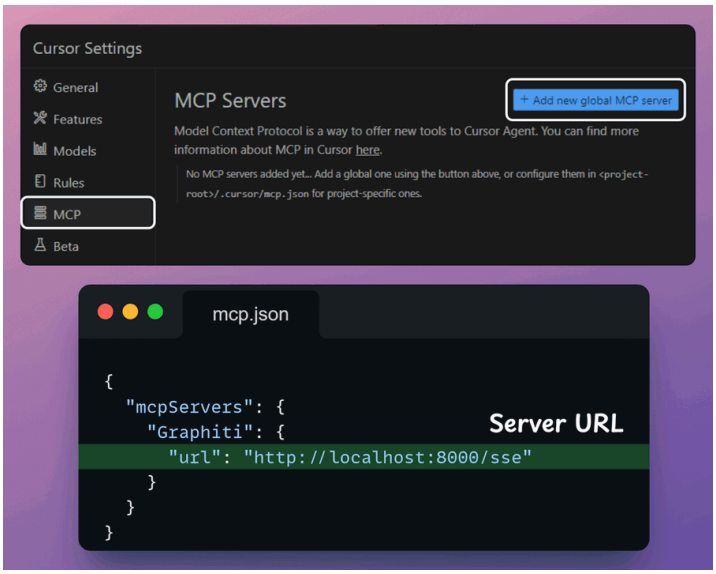
This configuration lets you query and visualize the knowledge graph using the Neo4j browser preview.



#2) Connect MCP server to Cursor

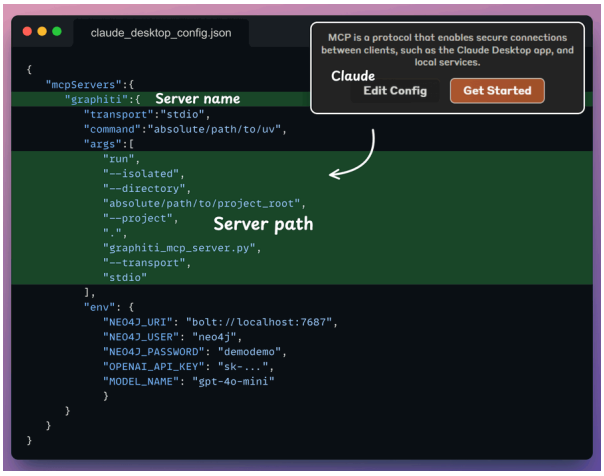
With tools and our server ready, let's integrate it with our Cursor IDE!

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



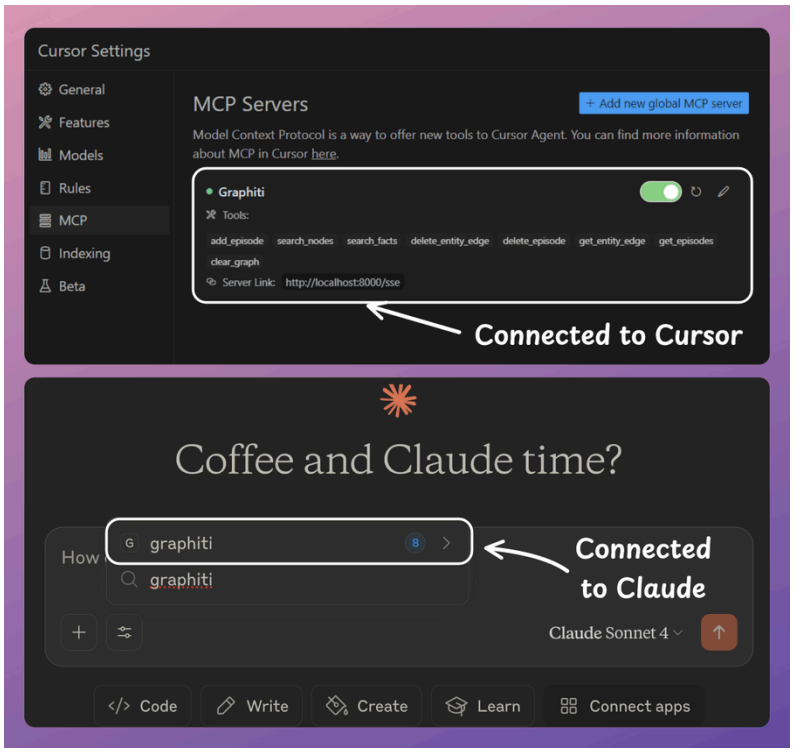
#3) Connect MCP server with Claude

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



Done!

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



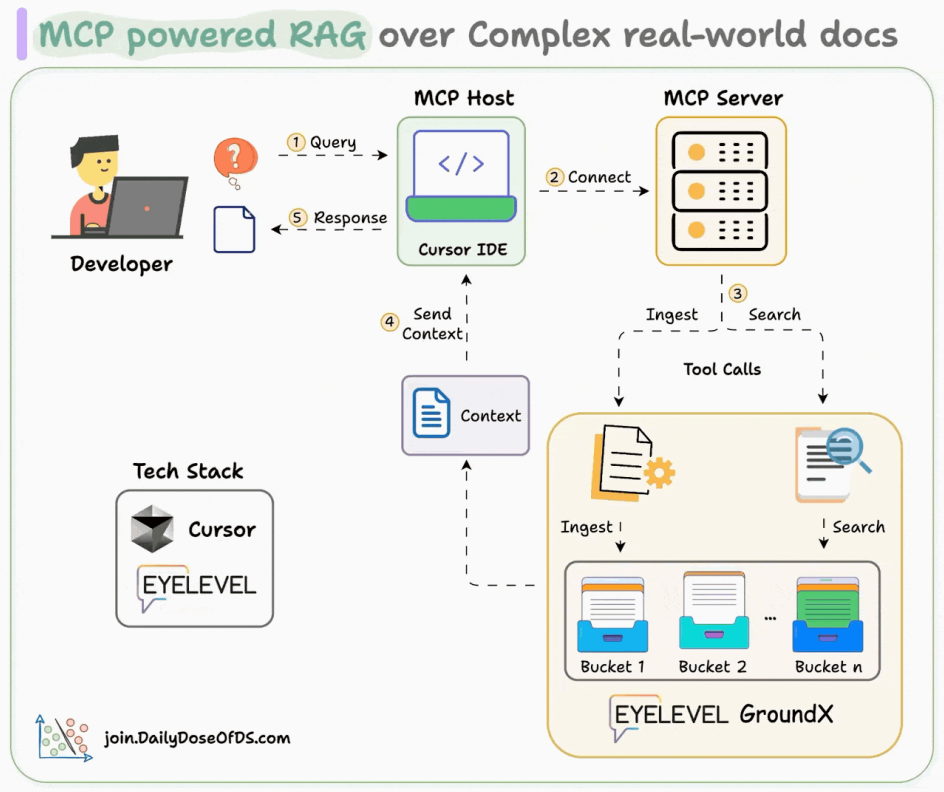
Now you can chat with Claude Desktop, share facts/info, store the response in memory, and retrieve them from Cursor, and vice versa.

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

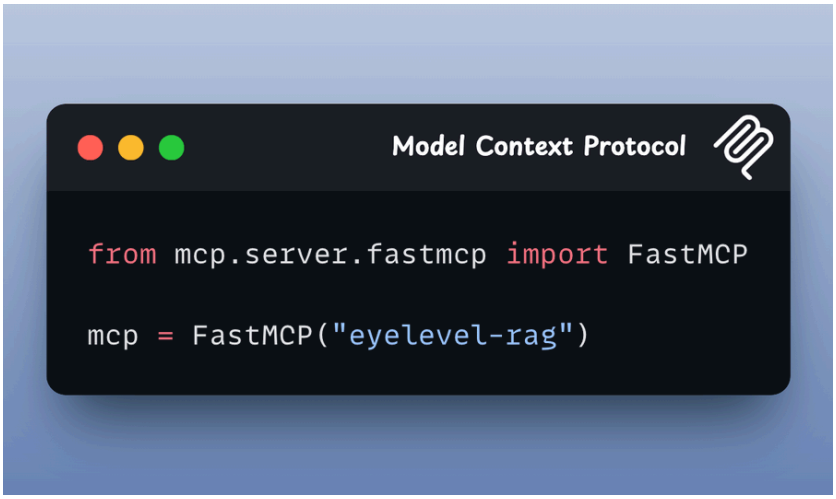
Workflow

- 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



#2) Create GroundX Client

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

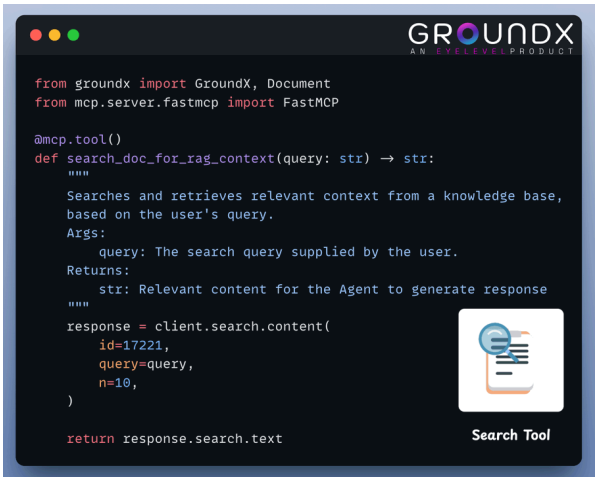
Here's how to set up a client:

#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:

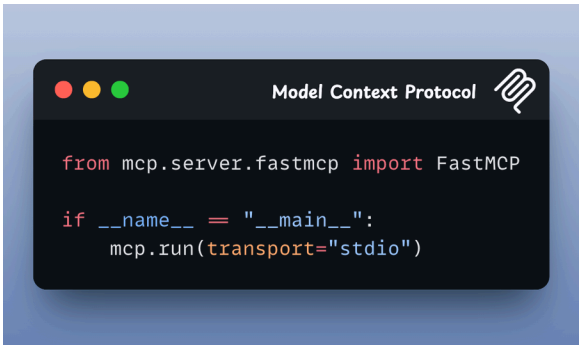
#4) Create Search tool

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



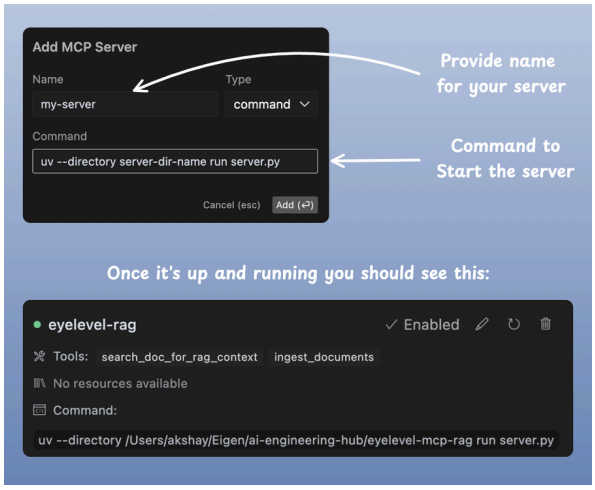
#5) Start the server

Starts an MCP server using stdio as the transport mechanism:



#6) Connect to Cursor

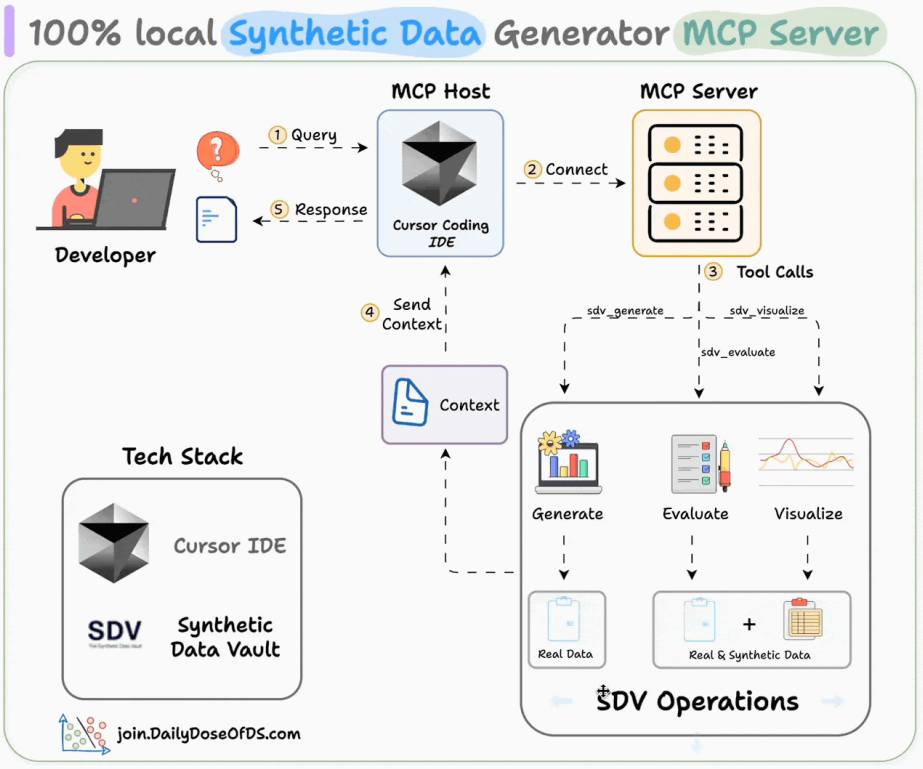
Inside you Cursor IDE follow this: Cursor → Settings → Cursor Settings → MCP
Then add and start your server like this:



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

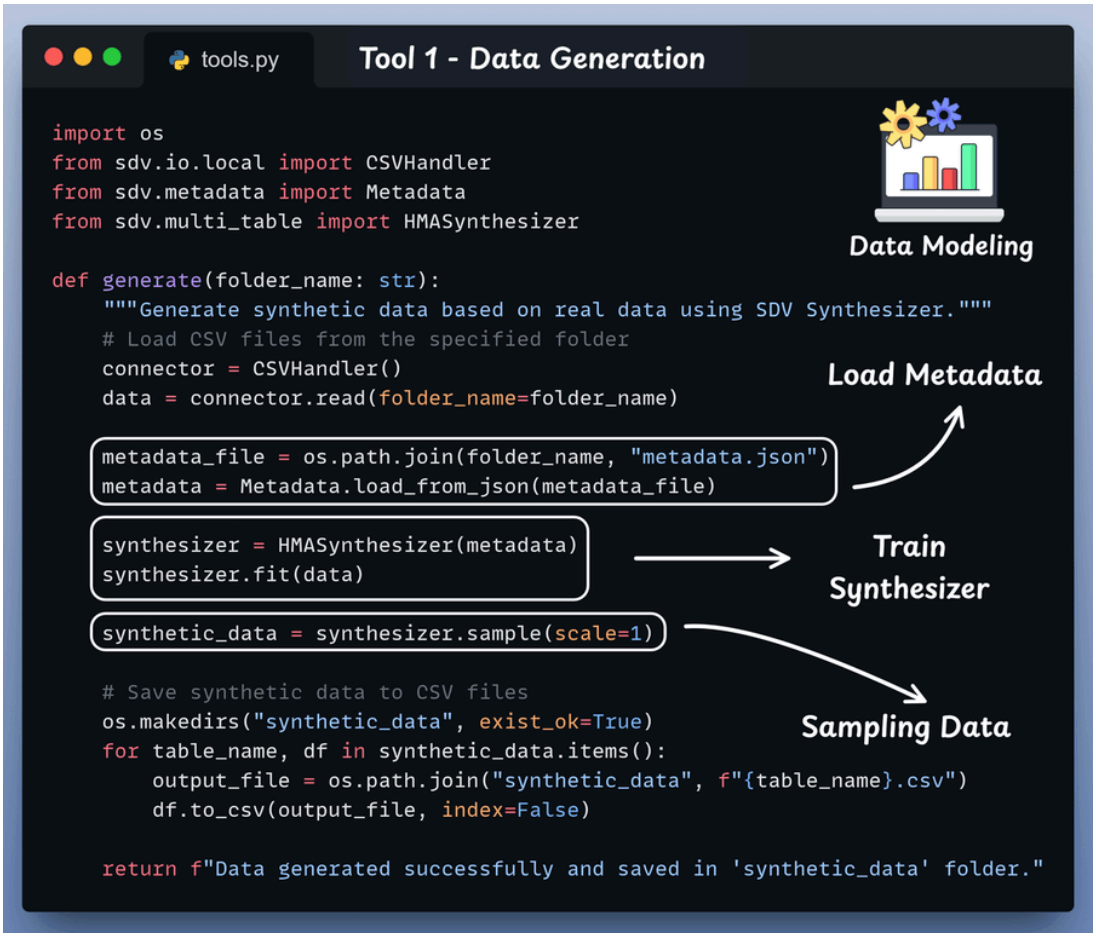
We have kept the actual implementation of these tools using the SDV SDK in a separate file, tools[.]py, that is imported here.

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.



#2) SDV Evaluate Tool

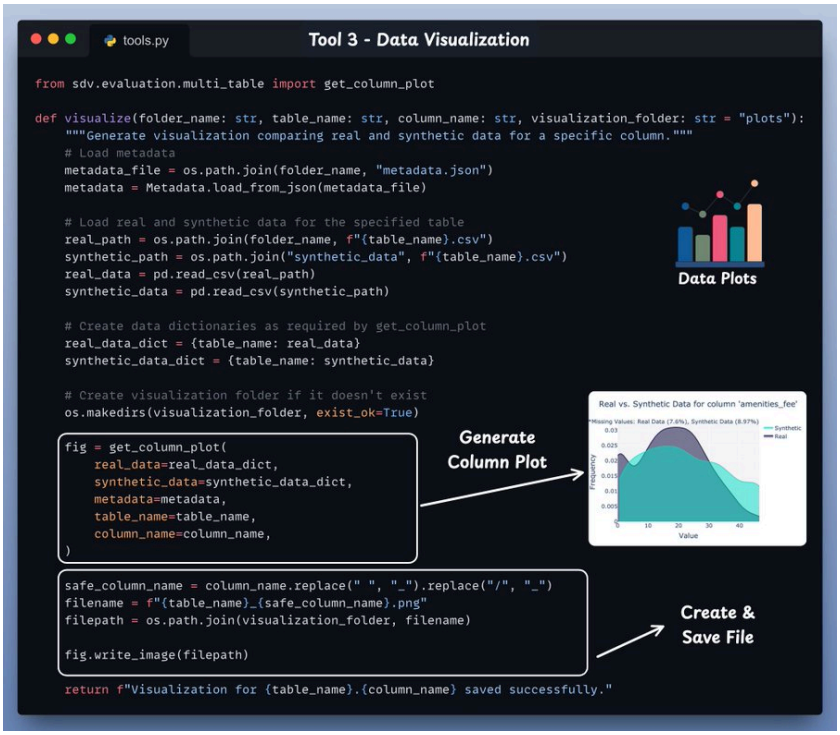
This tool evaluates the quality of synthetic data in comparison to real data.

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

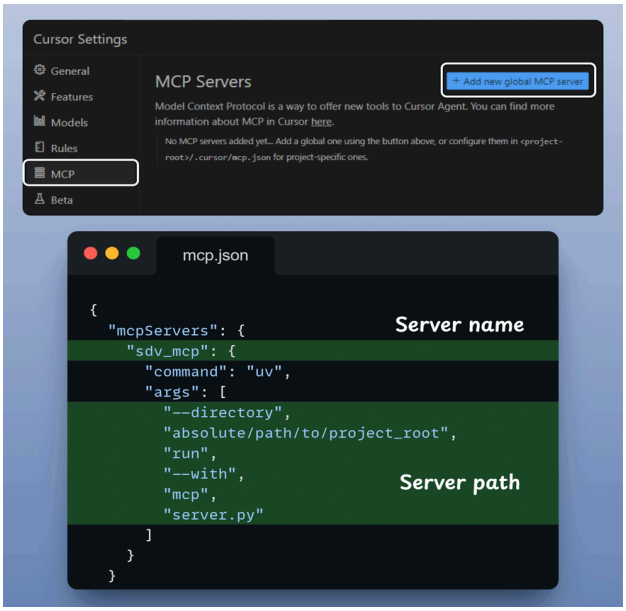
#3) SDV Visualize Tool

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

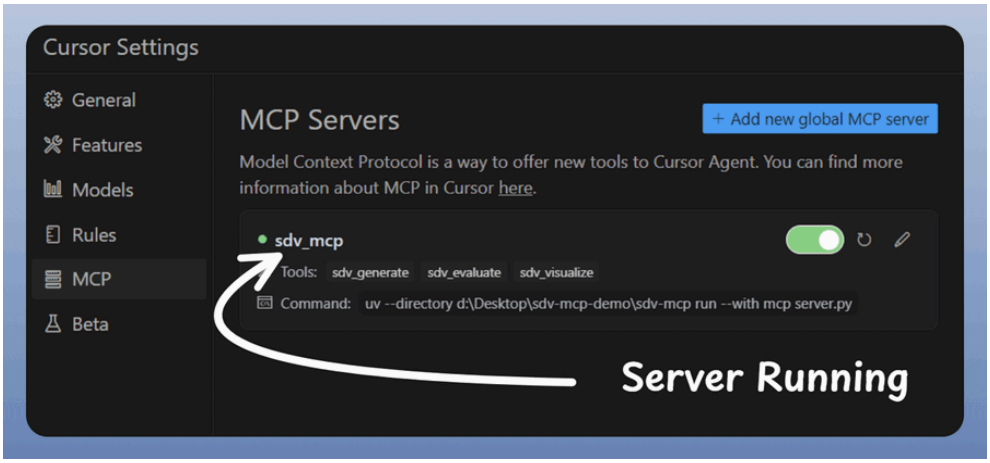
Use this function to visualize a real column alongside its corresponding synthetic column.



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



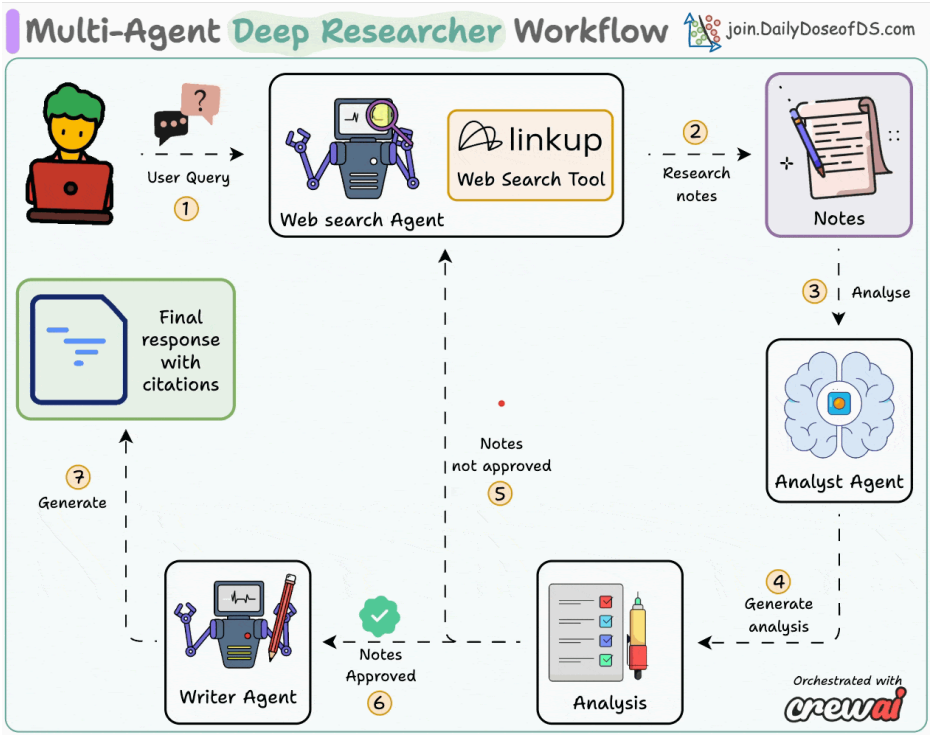
Done! Your synthetic data generator MCP server is live and connected to Cursor.



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

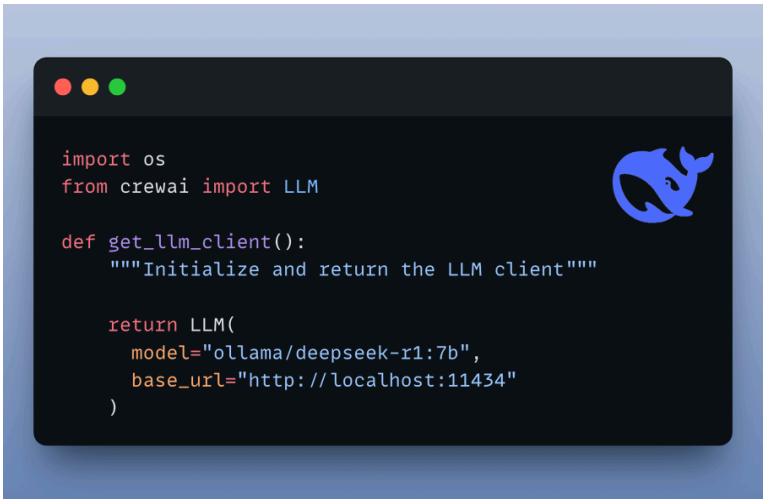
- Linkup platform for deep web research
- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek
- Cursor as MCP host

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.

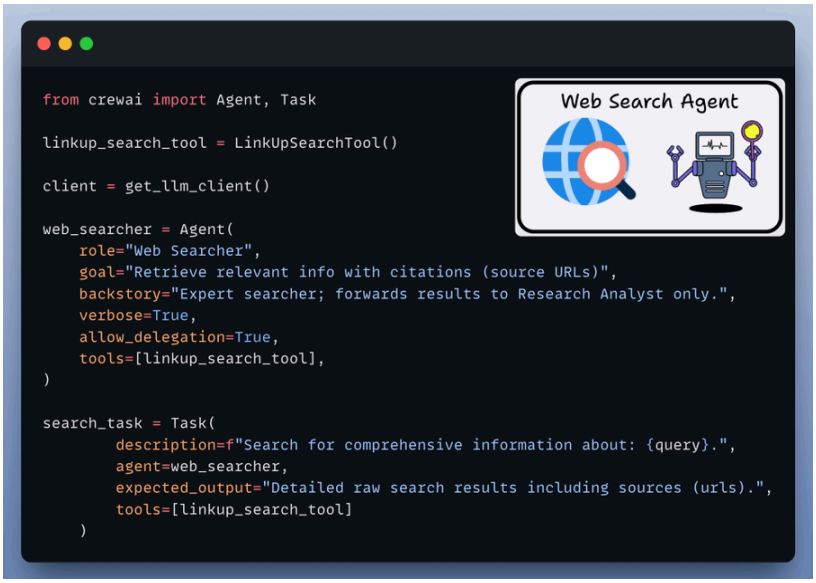


#2) Define Web Search Tool

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.

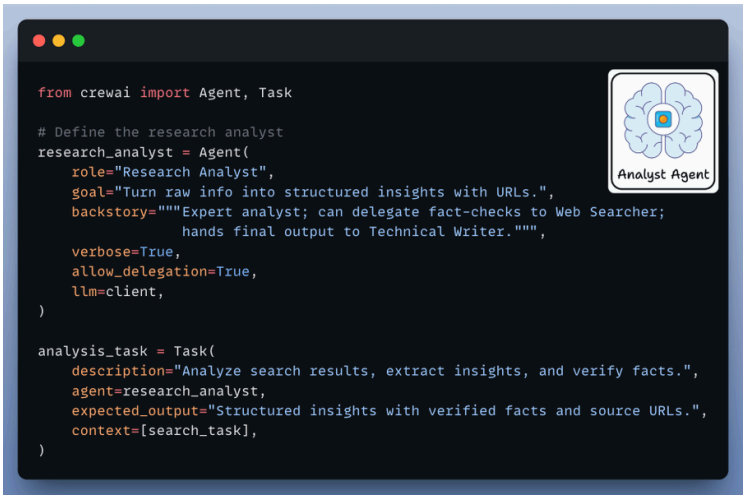
#3) Define Web Search Agent

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.



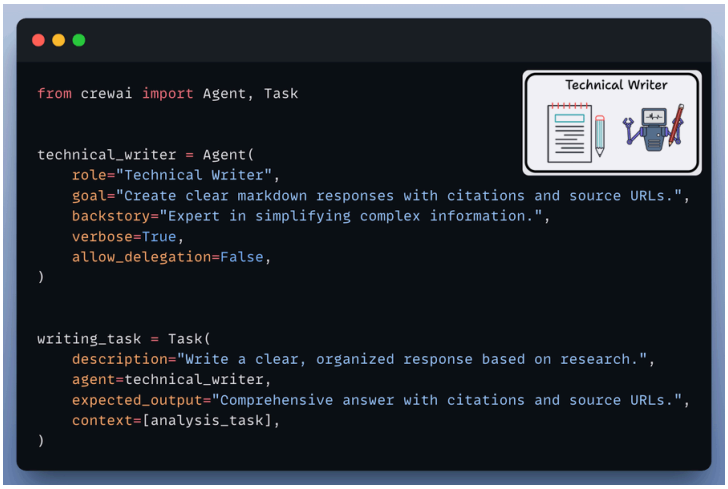
#4) Define Research Analyst Agent

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.



#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.



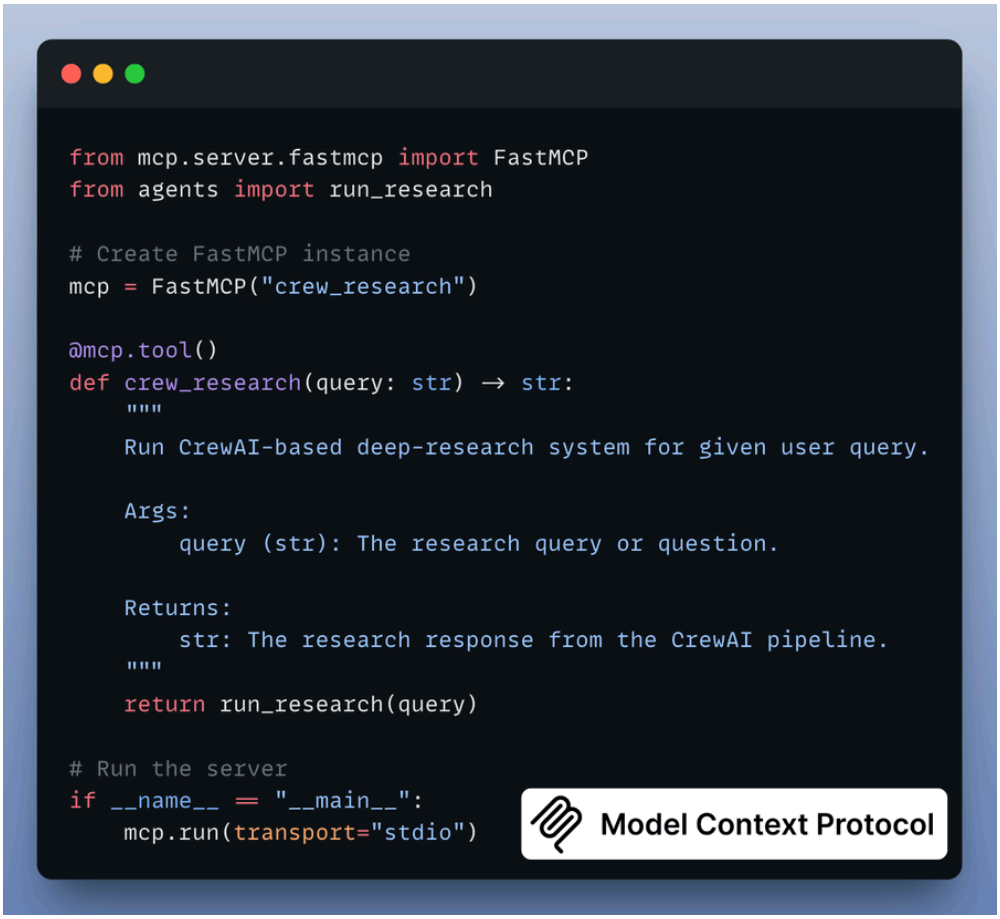
#6) Setup Crew

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

#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.

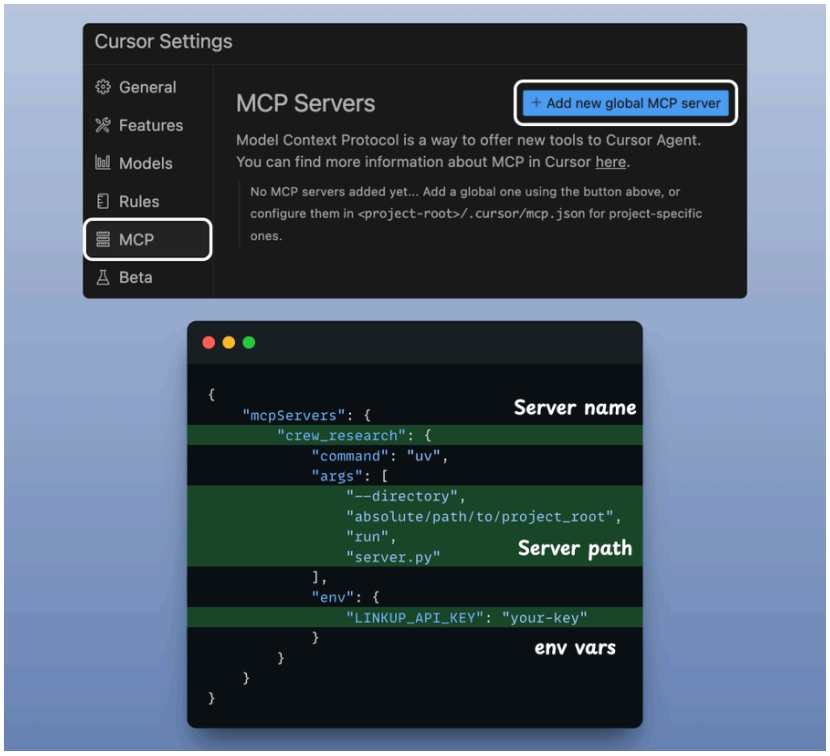
Let's see how to connect it with Cursor.



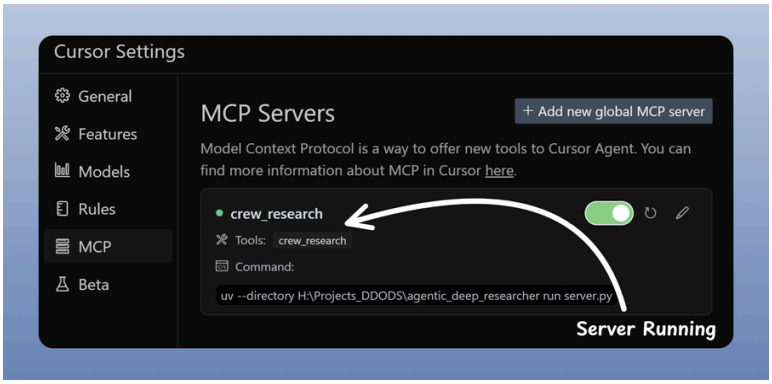
#8) Integrate MCP server with Cursor

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

In the JSON file, add what's shown below



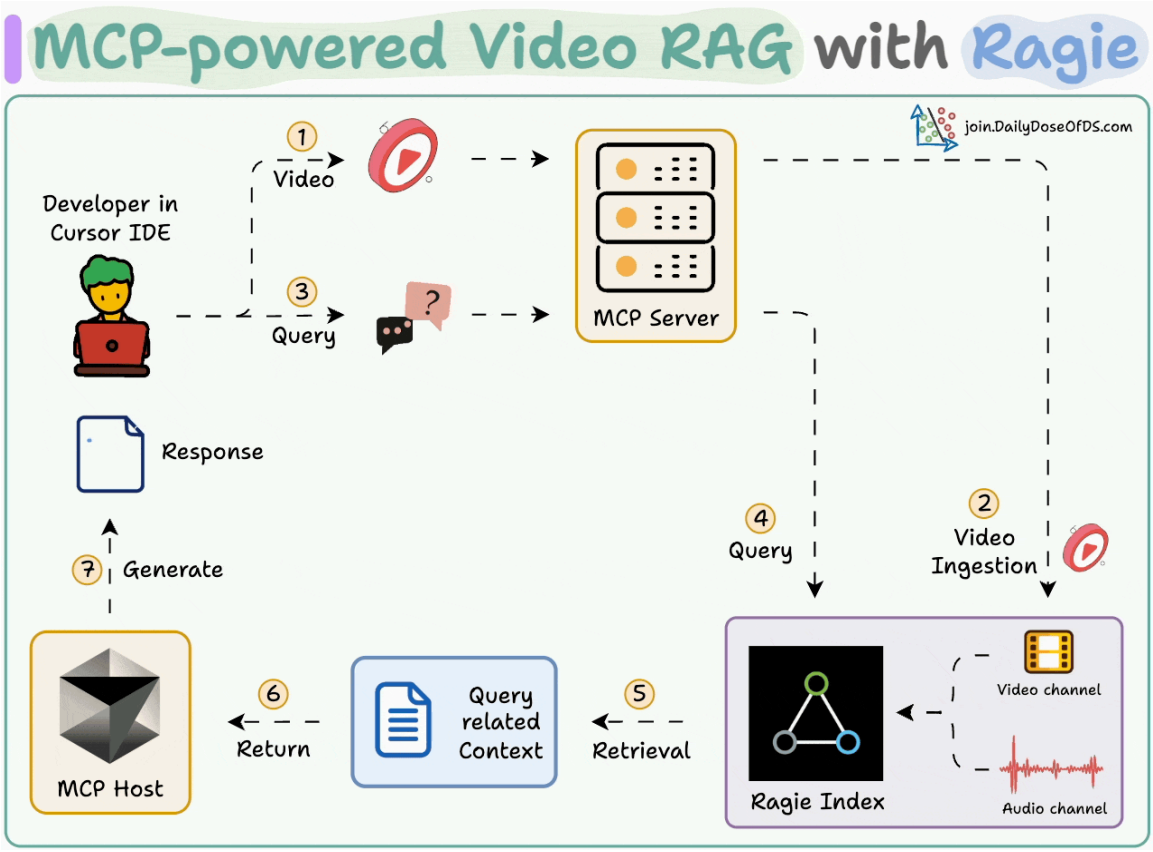
Done! Your deep research MCP server is live and connected to Cursor.



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

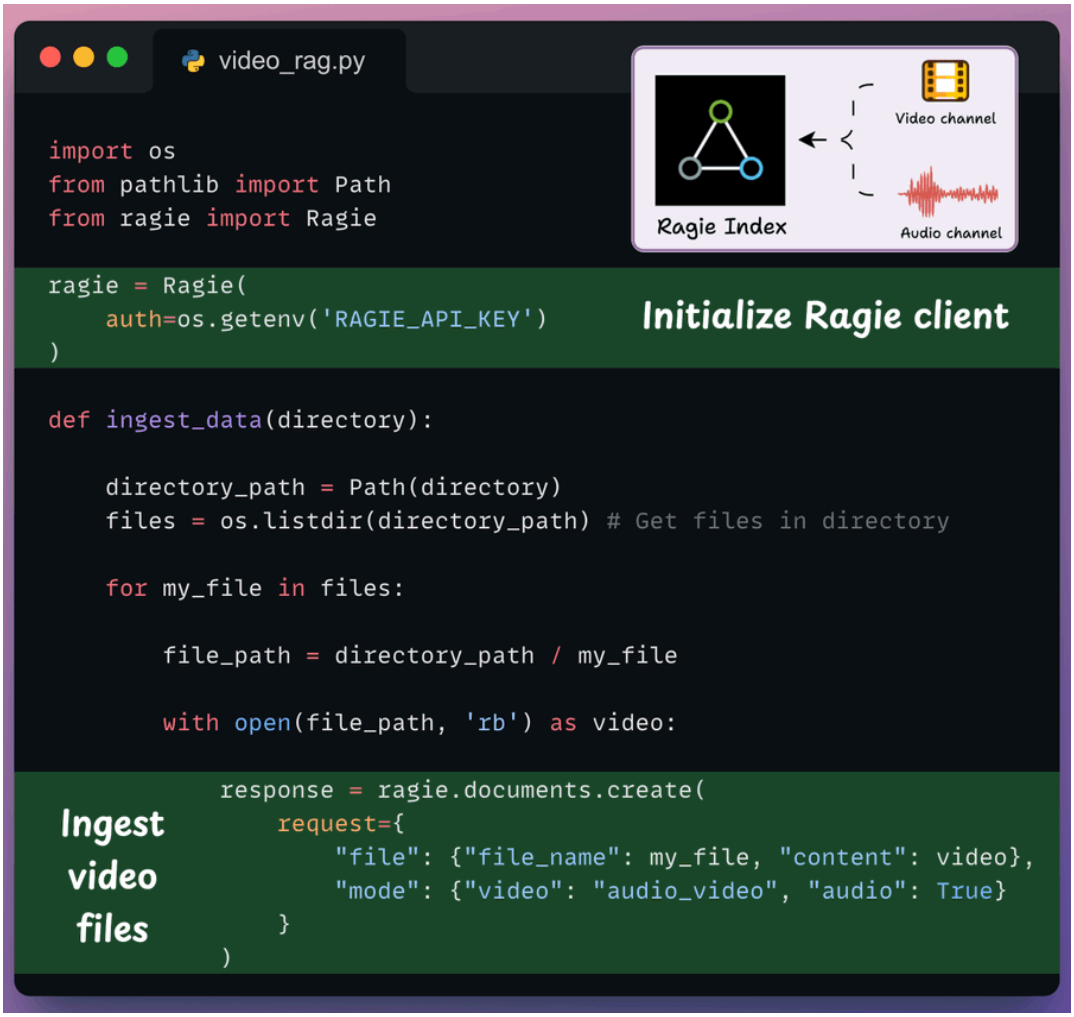
- User specifies video files and a query.
- An Ingestion tool indexes the videos in Ragie.
- A Query tool retrieves info from Ragie Index with citations.
- Show-video tool returns the video chunk that answers the query

Let's implement this!

#1) Ingest data

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

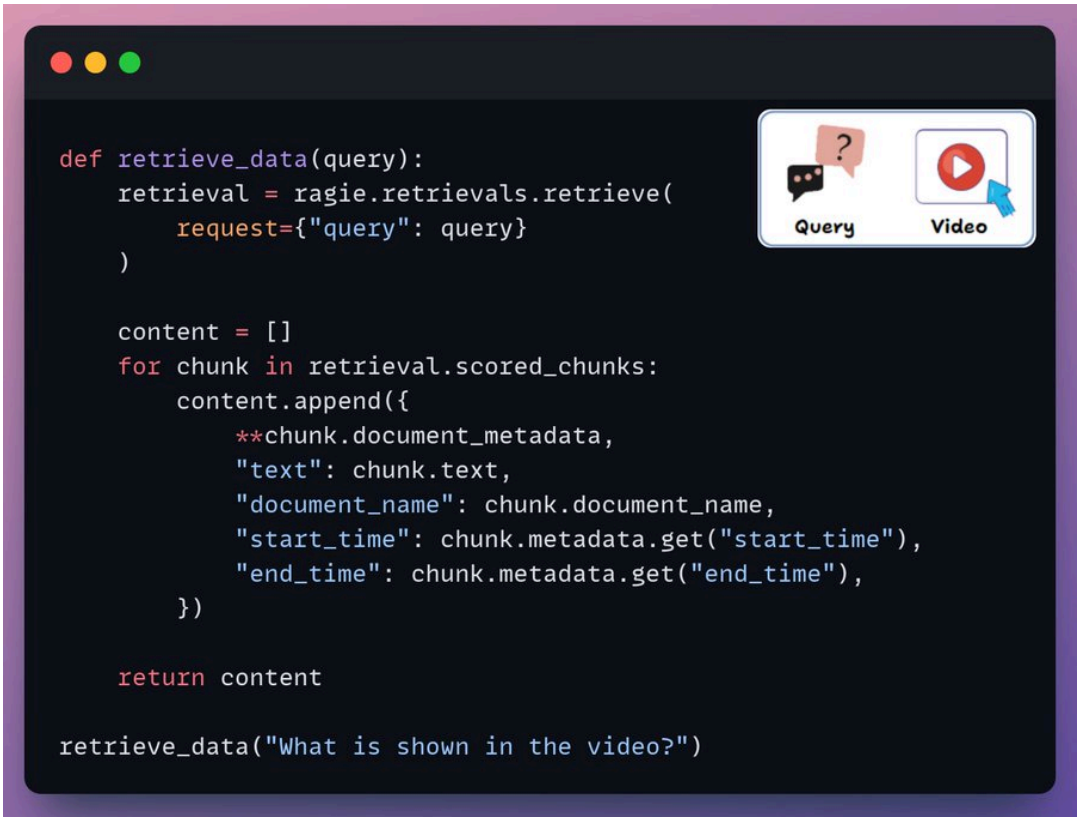
We also specify the audio-video mode to load both audio and video channels during ingestion.



#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.

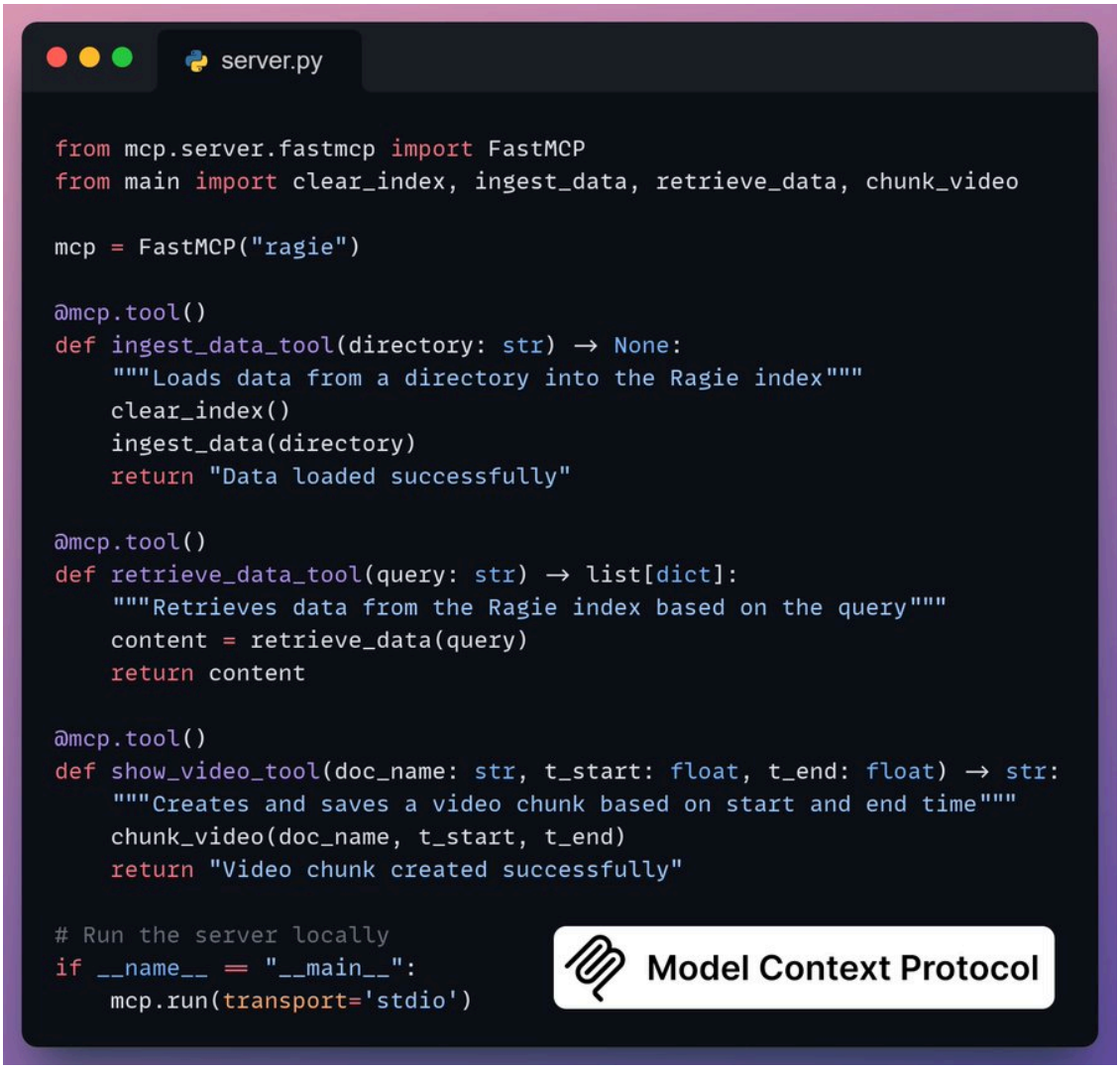



```python
def retrieve_data(query):
    retrieval = ragie.retrievals.retrieve(
        request={"query": query}
    )

    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

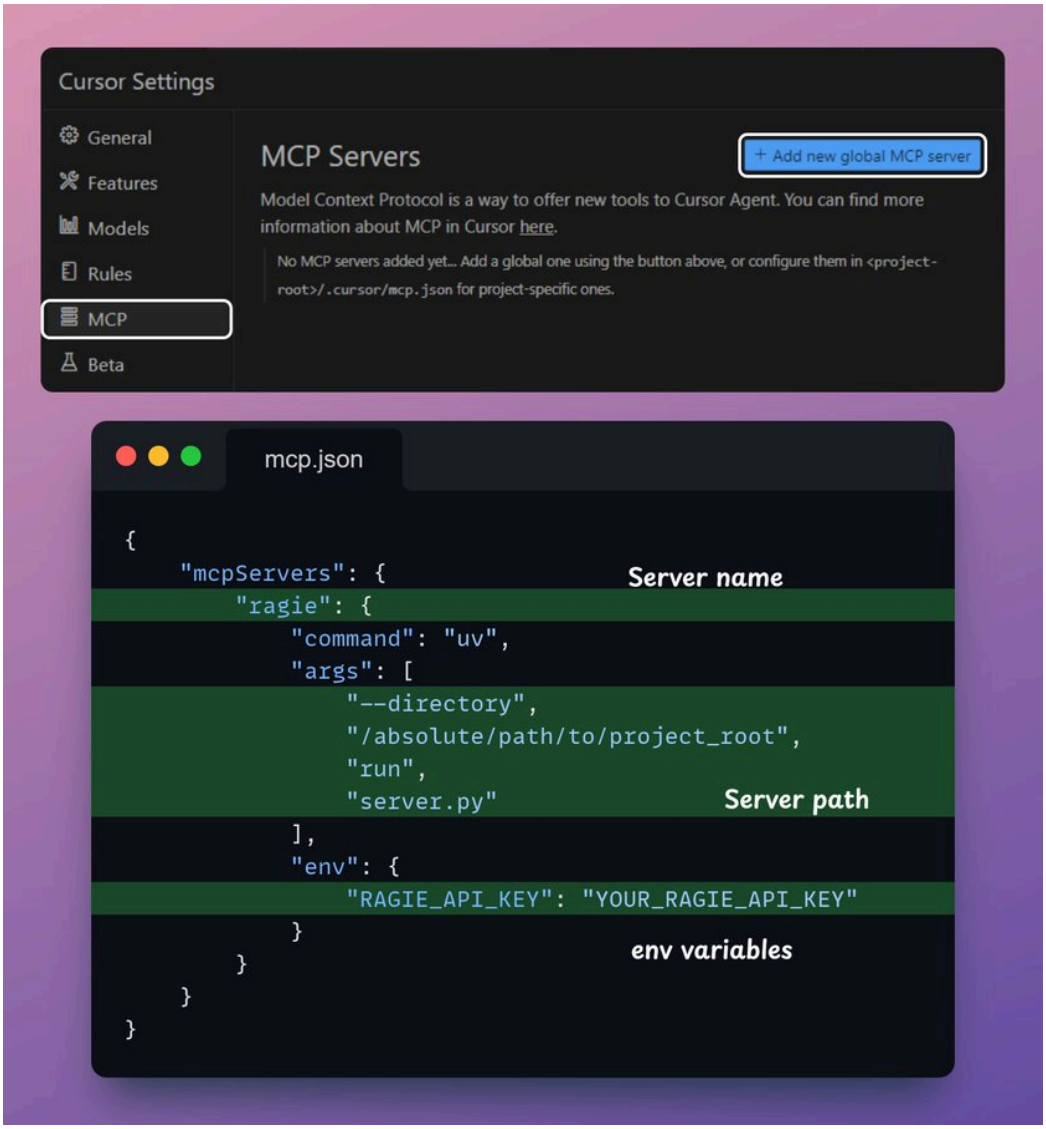
We integrate our RAG pipeline into an MCP server with 3 tools:

- 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



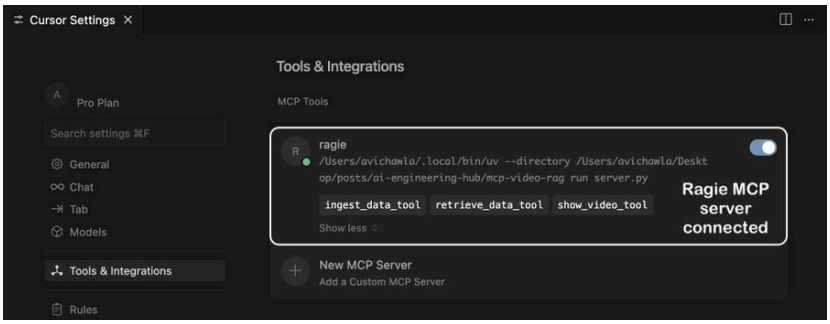
#4) Integrate MCP server with Cursor

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



Done!

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



Next, we interact with the MCP server through Cursor.

Based on the query, it can:

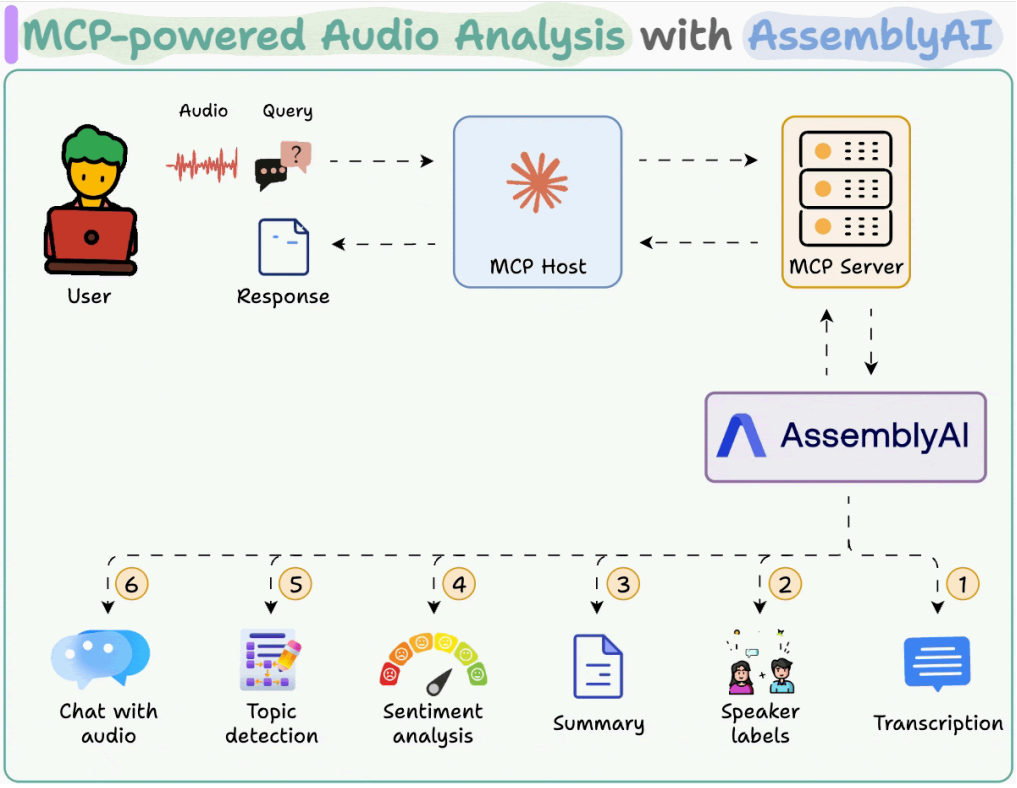
- Ingest a new video into the Ragie Index.
- Fetch detailed information about an existing video.
- Retrieve the video segment where a specific event occurred.

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

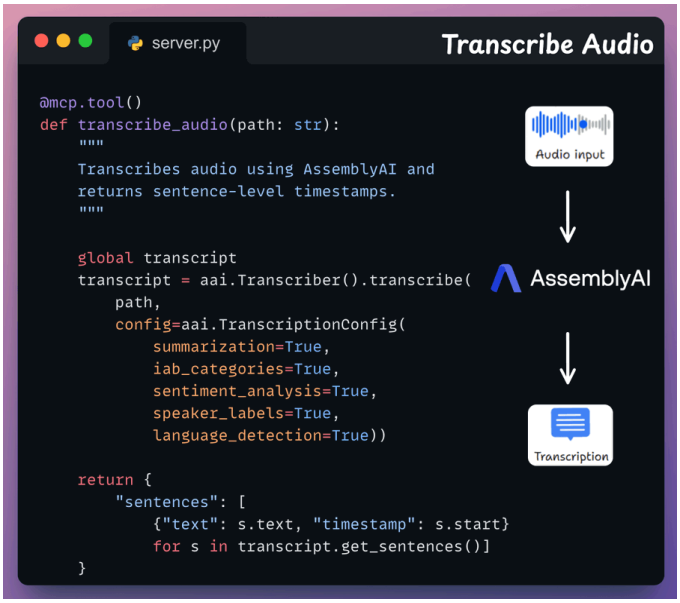
- AssemblyAI for transcription and audio analysis.
- Claude Desktop as the MCP host.
- Streamlit for the UI

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.

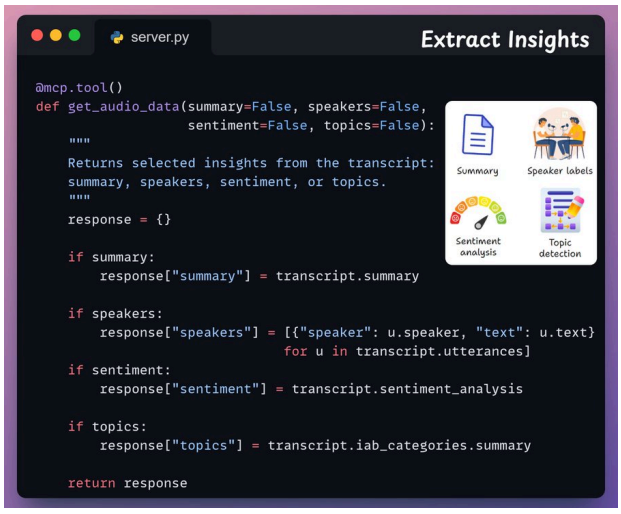
#1) Transcription MCP tool

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.



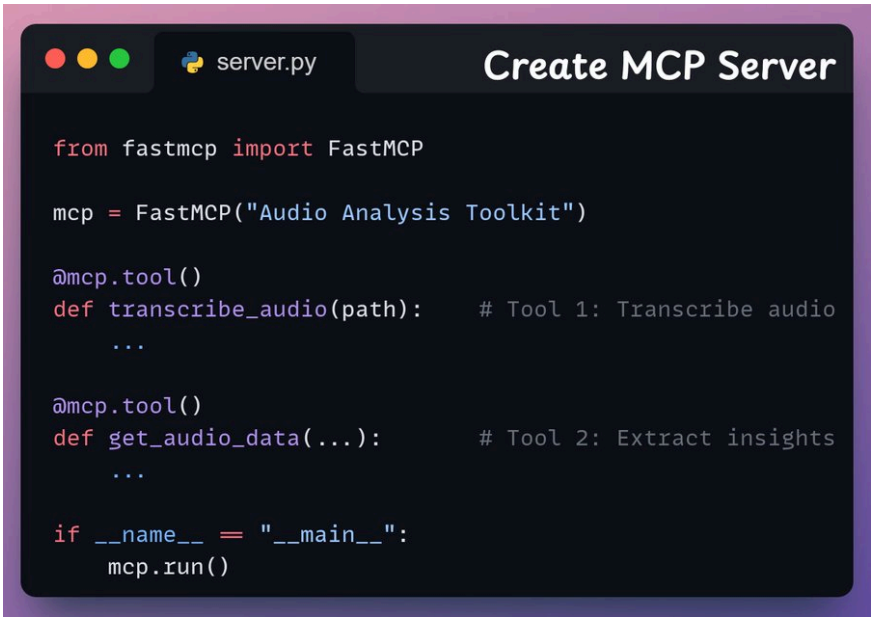
#2) Audio analysis tool

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



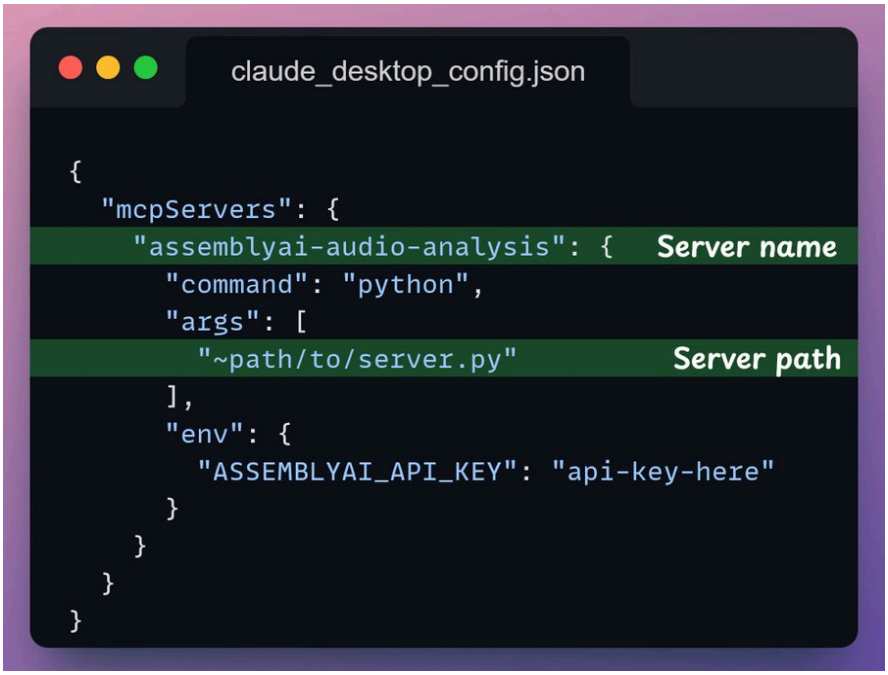
#3) Create MCP Server

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



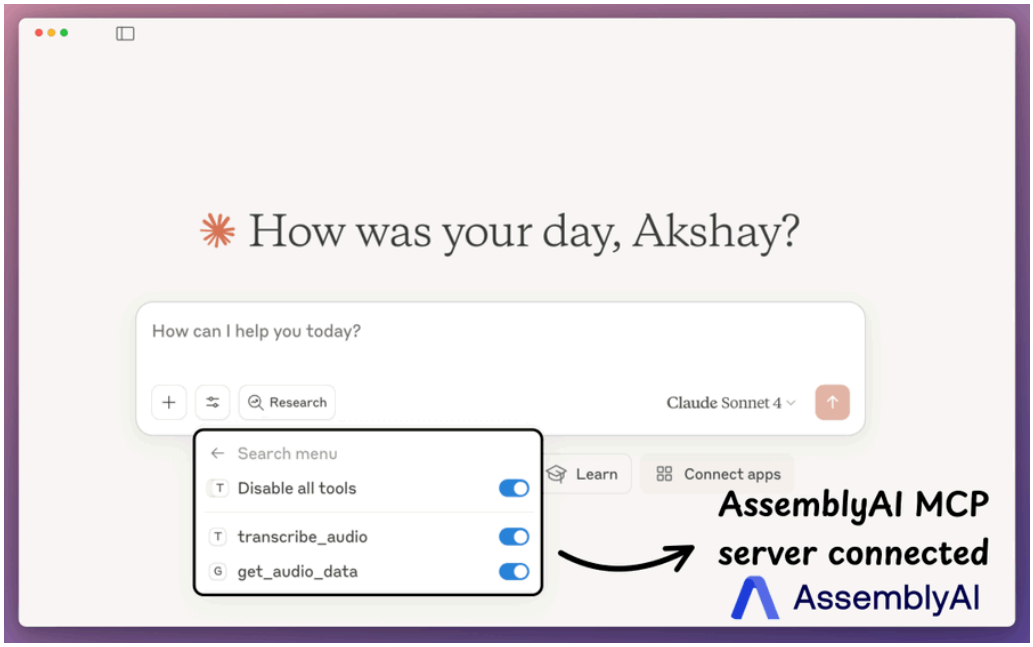
#4) Integrate MCP server with Claude Desktop

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



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/