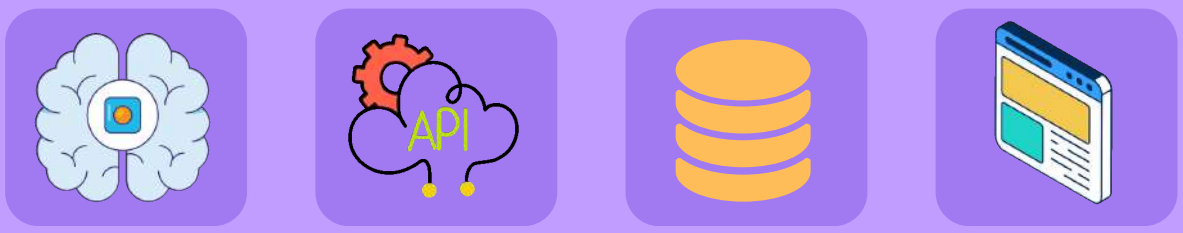
2025 EDITION

FREE

AI AGENTS

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 8 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.

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

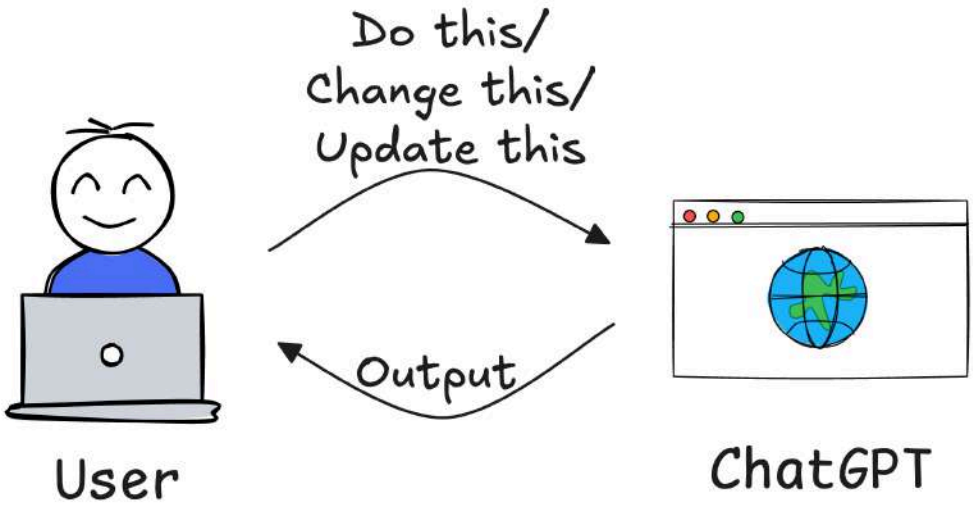
	https://bit.ly/agents-assessment

Table of contents

AI Agents

What is an AI Agent?

Imagine you want to generate a report on the latest trends in AI research. If you use a standard LLM, you might:

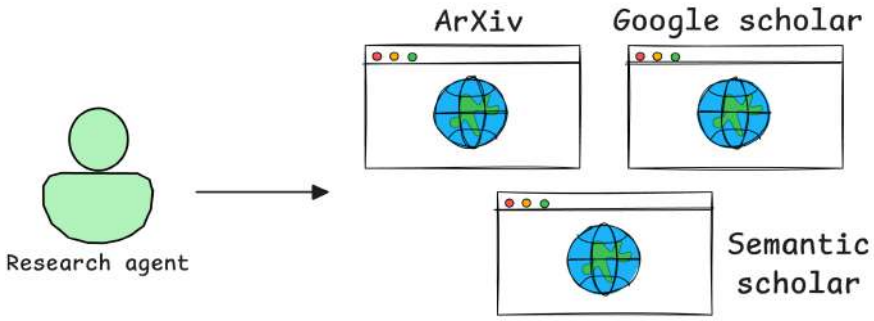


1. Ask for a summary of recent AI research papers.
2. Review the response and realize you need sources.
3. Obtain a list of papers along with citations.
4. Find that some sources are outdated, so you refine your query.
5. Finally, after multiple iterations, you get a useful output.

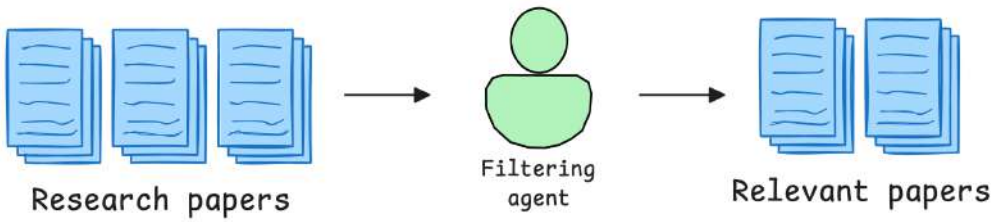
This iterative process takes time and effort, requiring you to act as the decision-maker at every step.

Now, let's see how AI agents handle this differently:

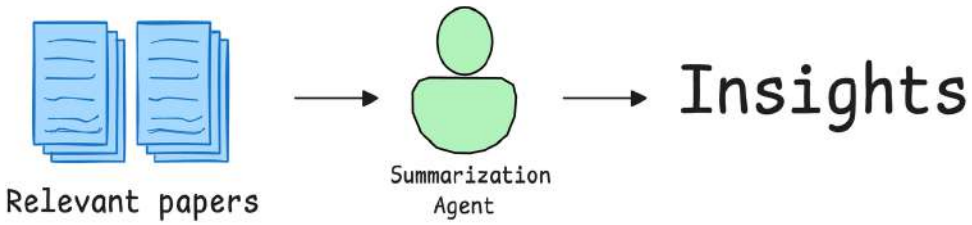
A Research Agent autonomously searches and retrieves relevant AI research papers from arXiv, Semantic Scholar, or Google Scholar.



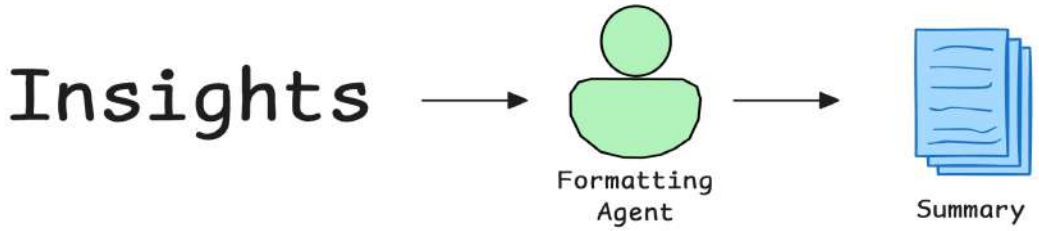
- A Filtering Agent scans the retrieved papers, identifying the most relevant ones based on citation count, publication date, and keywords.



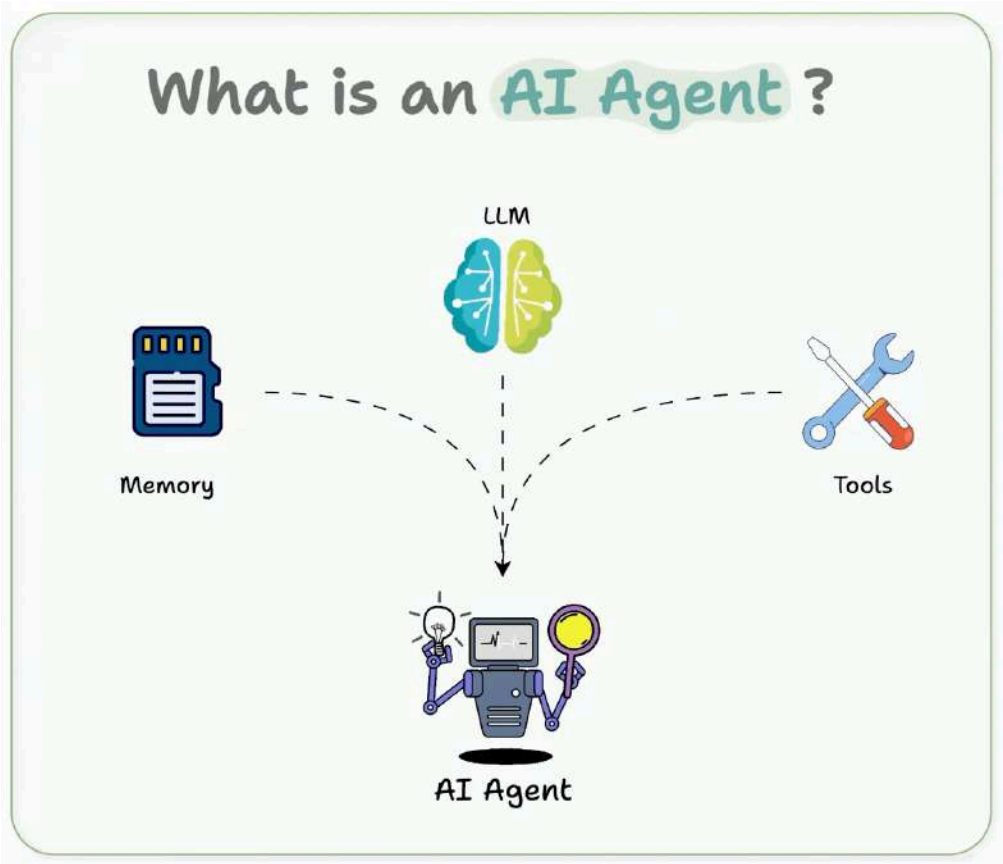
- A Summarization Agent extracts key insights and condenses them into an easy-to-read report.



- A Formatting Agent structures the final report, ensuring it follows a clear, professional layout.



Here, the AI agents not only execute the research process end-to-end but also self-refine their outputs, ensuring the final report is comprehensive, up-to-date, and well-structured - all without requiring human intervention at every step.



To formalize AI Agents are autonomous systems that can reason, think, plan, figure out the relevant sources and extract information from them when needed, take actions, and even correct themselves if something goes wrong.

Agent vs LLM vs RAG

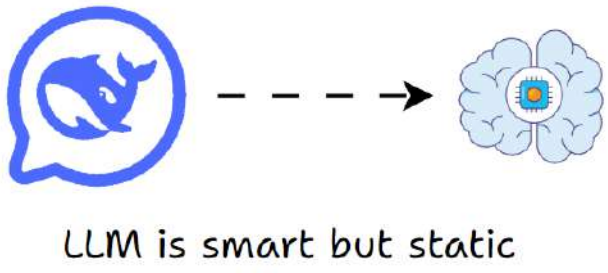
Let's break it down with a simple analogy:

- LLM is the brain.
- RAG is feeding that brain with fresh information.
- An agent is the decision-maker that plans and acts using the brain and the tools.

LLM (Large Language Model)

An LLM like GPT-4 is trained on massive text data.

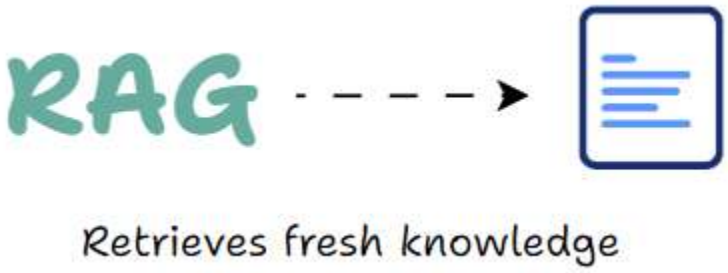
It can reason, generate, summarize but only using what it already knows (i.e., its training data).



It's smart, but static. It can't access the web, call APIs, or fetch new facts on its own.

RAG (Retrieval-Augmented Generation)

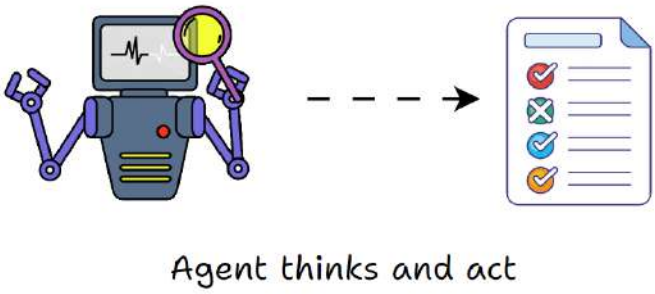
RAG enhances an LLM by retrieving external documents (from a vector DB, search engine, etc.) and feeding them into the LLM as context before generating a response.



RAG makes the LLM aware of updated, relevant info without retraining.

Agent

An Agent adds autonomy to the mix.



It doesn't just answer a question—it decides what steps to take:

Should it call a tool? Search the web? Summarize? Store info?

An Agent uses an LLM, calls tools, makes decisions, and orchestrates workflows just like a real assistant.

Building blocks of AI Agents

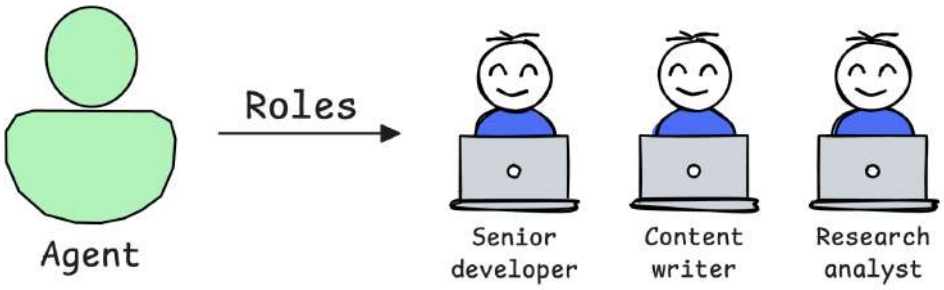
AI agents are designed to reason, plan, and take action autonomously. However, to be effective, they must be built with certain key principles in mind. There are six essential building blocks that make AI agents more reliable, intelligent, and useful in real-world applications:

1. Role-playing
2. Focus
3. Tools
4. Cooperation
5. Guardrails
6. Memory

Let's explore each of these concepts and understand why they are fundamental to building great AI agents.

1) Role-playing

One of the simplest ways to boost an agent's performance is by giving it a clear, specific role.



A generic AI assistant may give vague answers. But define it as a "Senior contract lawyer," and it responds with legal precision and context.

Why?

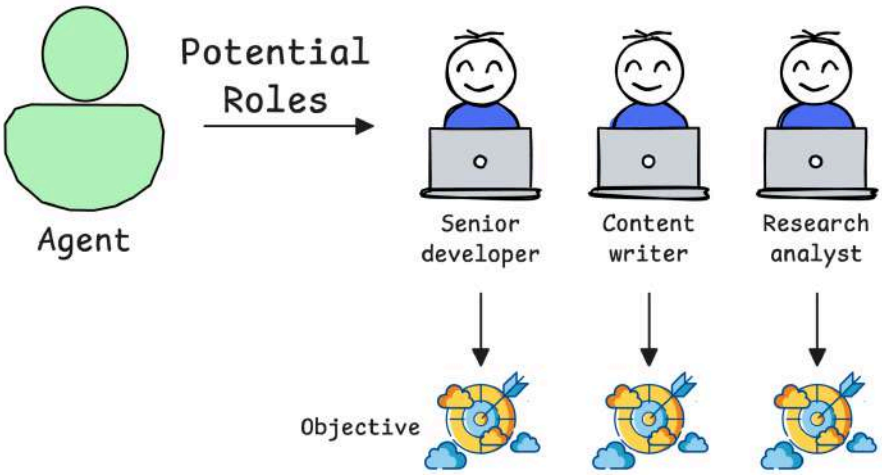
Because role assignment shapes the agent's reasoning and retrieval process. The

more specific the role, the sharper and more relevant the output.

2) Focus/Tasks

Focus is key to reducing hallucinations and improving accuracy.

Giving an agent too many tasks or too much data doesn't help - it hurts.



Overloading leads to confusion, inconsistency, and poor results.

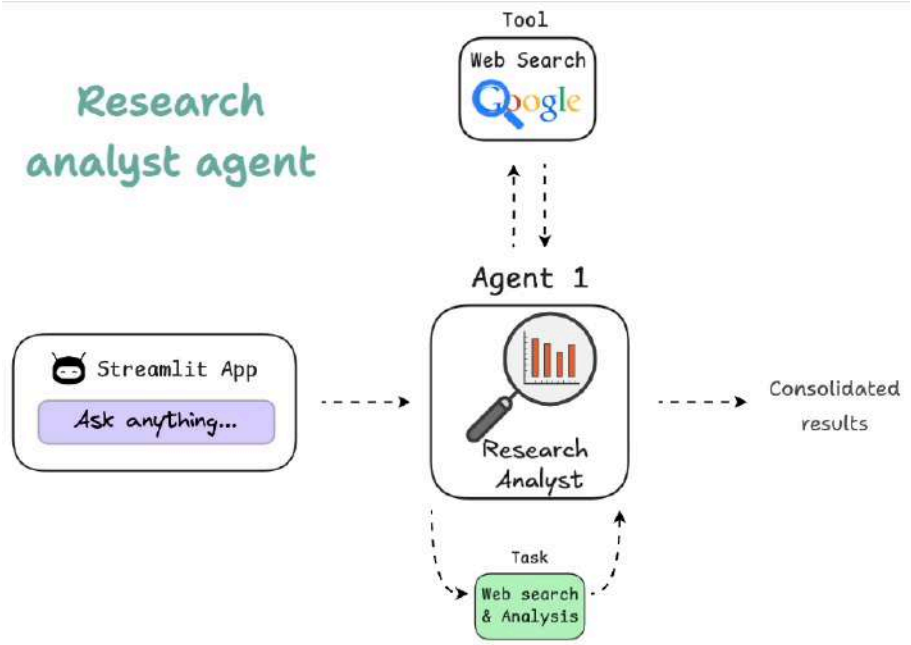
For example, a marketing agent should stick to messaging, tone, and audience not pricing or market analysis.

Instead of trying to make one agent do everything, a better approach is to use multiple agents, each with a specific and narrow focus.
Specialized agents perform better - every time.

3) Tools

Agents get smarter when they can use the right tools.

But more tools ≠ better results.



For example, an AI research agent could benefit from:

- A web search tool for retrieving recent publications.
- A summarization model for condensing long research papers.
- A citation manager to properly format references.

But if you add unnecessary tools—like a speech-to-text module or a code execution environment—it could confuse the agent and reduce efficiency.

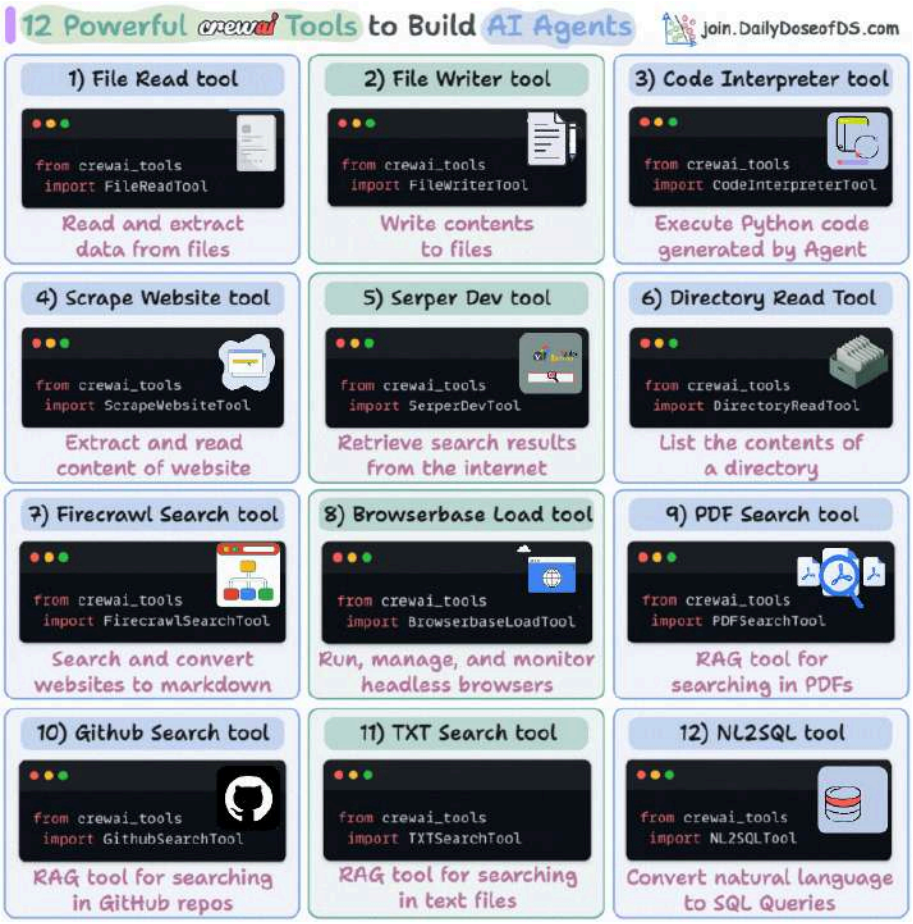
#3.1) Custom tools

While LLM-powered agents are great at reasoning and generating responses, they lack direct access to real-time information, external systems, and specialized computations.

Tools allow the Agent to:

- Search the web for real-time data.
- Retrieve structured information from APIs and databases.
- Execute code to perform calculations or data transformations.
- Analyze images, PDFs, and documents beyond just text inputs.

CrewAI supports several tools that you can integrate with Agents, as depicted below:

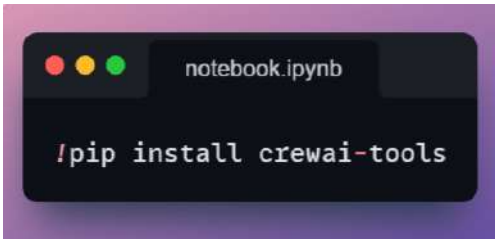


However, you may need to build custom tools at times.

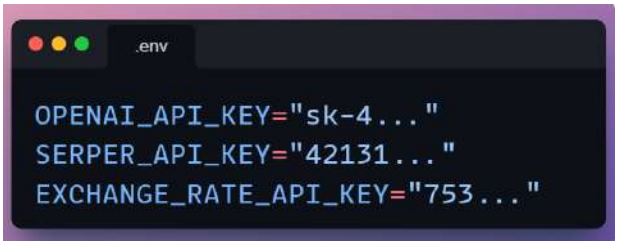
In this example, we're building a real-time currency conversion tool inside CrewAI. Instead of making an LLM guess exchange rates, we integrate a custom tool that fetches live exchange rates from an external API and provides some insights.

Below, let's look at how you can build one for your custom needs in the CrewAI framework.

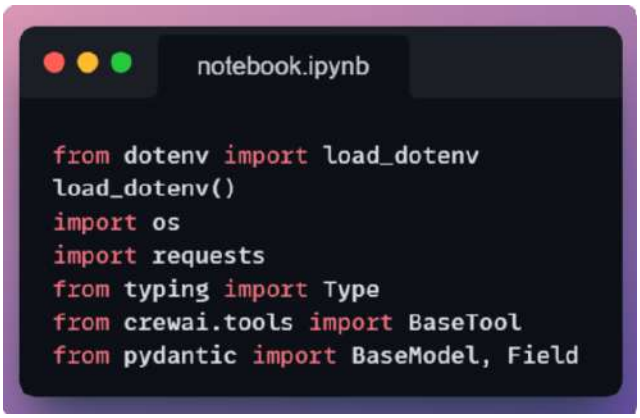
Firstly, make sure the tools package is installed:



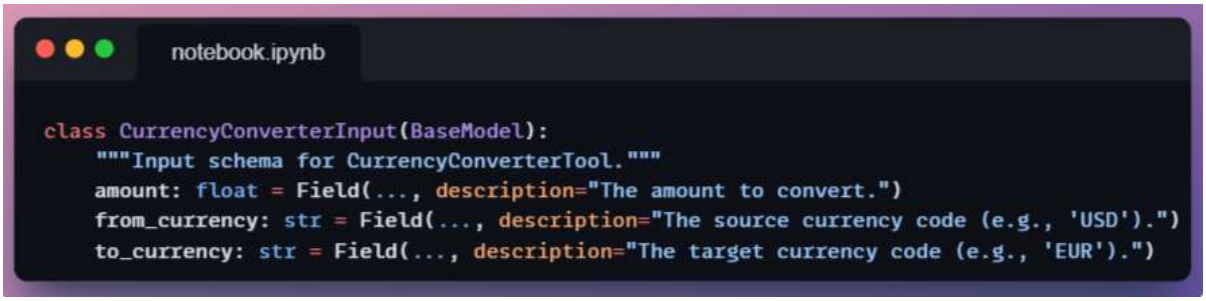
You would also need an API key from here: https://www.exchangerate-api.com/ (it's free). Specify it in the .env file as shown below:



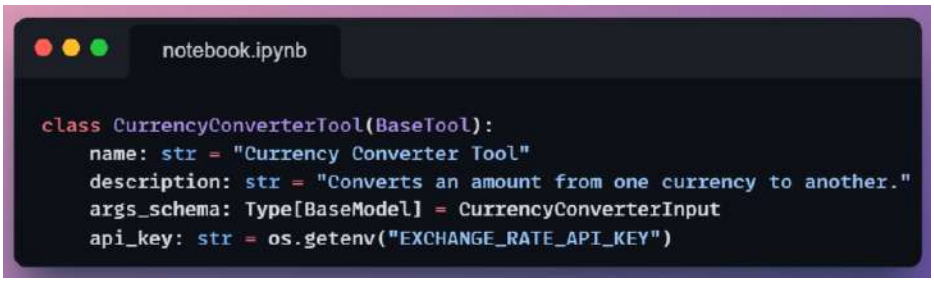
Once that's done, we start with some standard import statements:



Next, we define the input fields the tool expects using Pydantic.

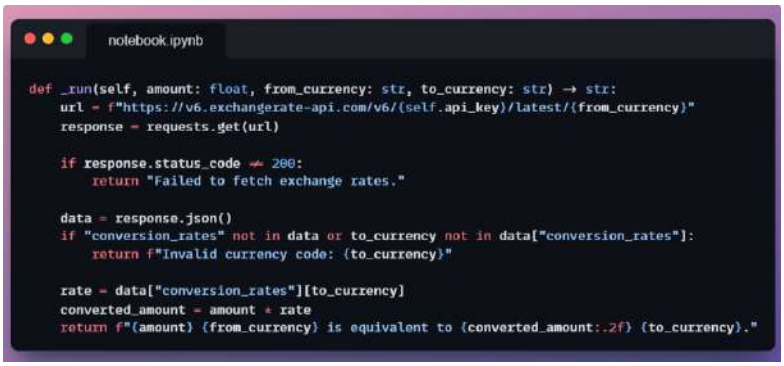


Now, we define the CurrencyConverterTool by inheriting from *BaseTool:*



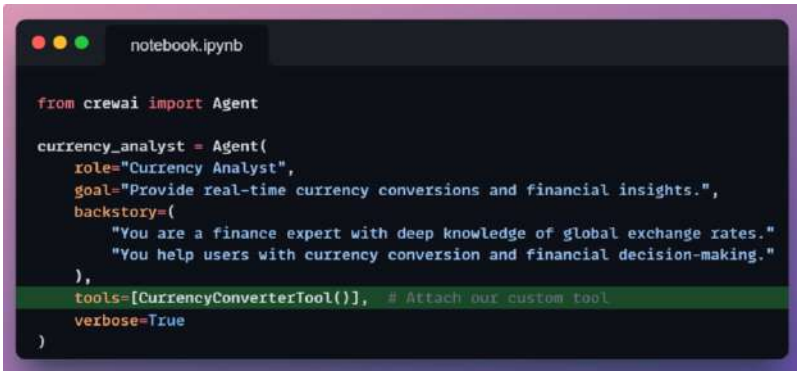
Every tool class should have the _run method which we will execute whenever the Agents wants to make use of it.

For our use case, we implement it as follows:

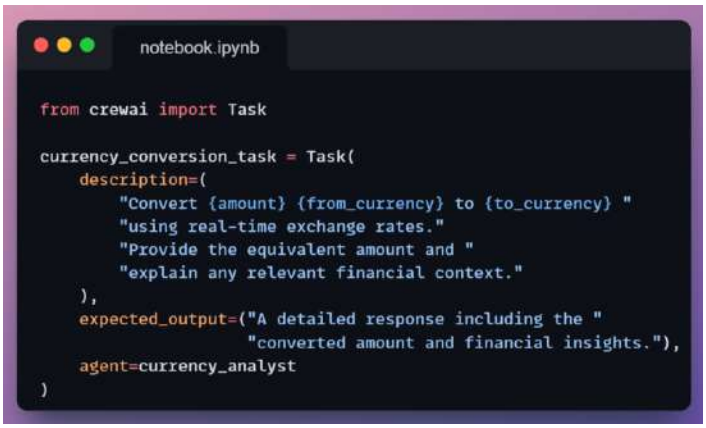


In the above code, we fetch live exchange rates using an API request. We also handle errors if the request fails or the currency code is invalid.

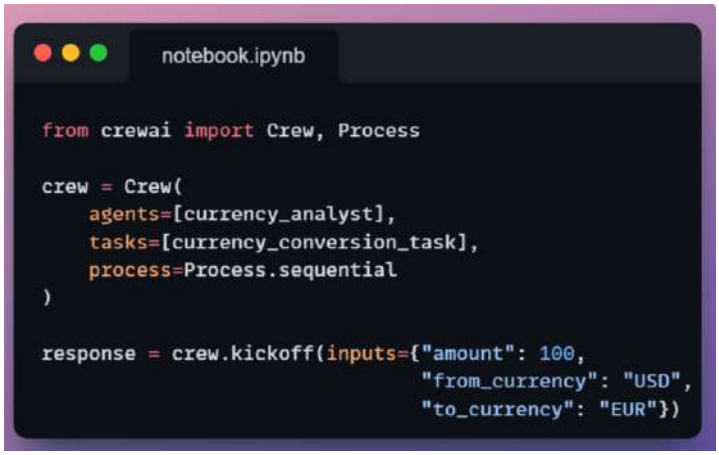
Now, we define an agent that uses the tool for real-time currency analysis and attach our CurrencyConverterTool, allowing the agent to call it directly if needed:



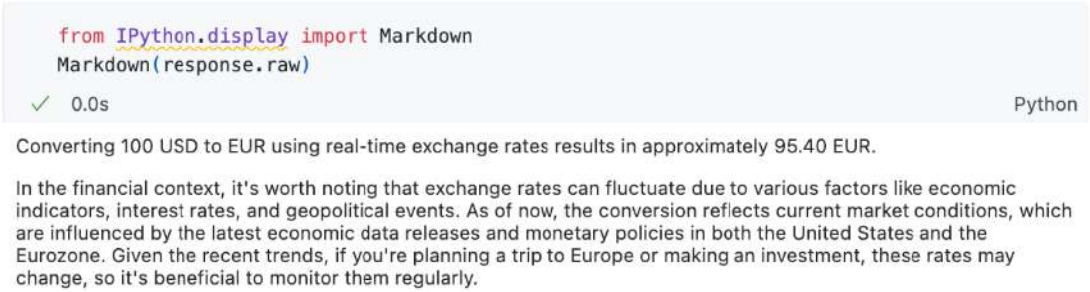
We assign a task to the currency_analyst agent.



Finally, we create a Crew, assign the agent to the task, and execute it.



Printing the response, we get the following output:



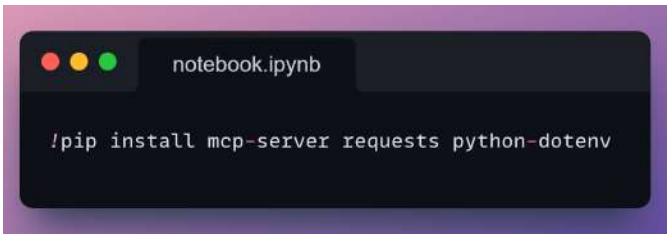
Works as expected!

#3.2) Custom tools via MCP

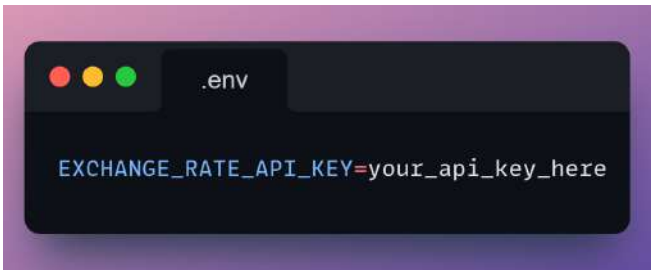
Now, let's take it a step further.

Instead of embedding the tool directly in every Crew, we'll expose it as a reusable MCP tool—making it accessible across multiple agents and flows via a simple server.

First, install the required packages:



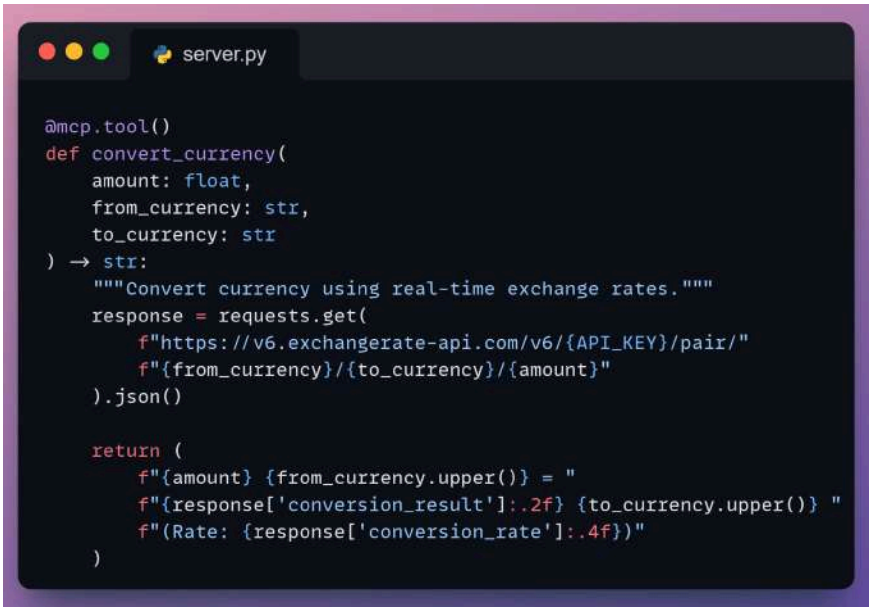
We'll continue using ExchangeRate-API in our .env file:



We'll now write a lightweight server.py script that exposes the currency converter tool. We start with the standard imports:

Now, we load environment variables and initialize the server:

Next, we define the tool logic with @mcp.tool():



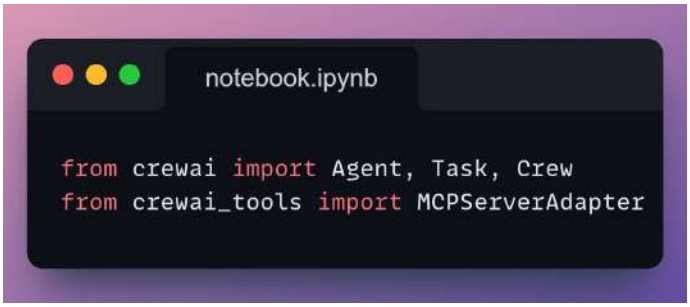
This function takes three inputs—amount, source currency, and target currency—and returns the converted result using the real-time exchange rate API.

To make the tool accessible, we need to run the MCP server. Add this at the end of your script:

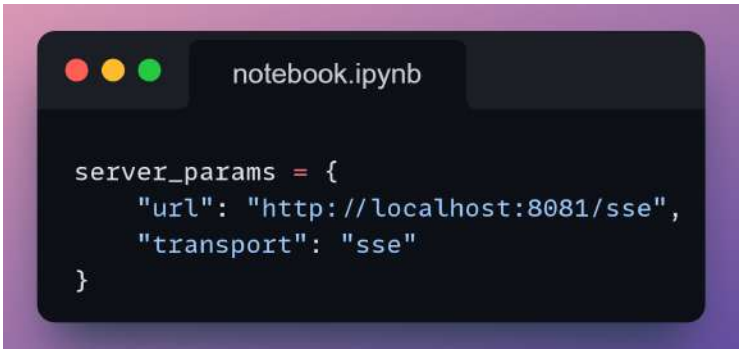
This starts the server and exposes your convert_currency tool at: http://localhost:8081/sse.

Now any CrewAI agent can connect to it using MCPServerAdapter. Let's now consume this tool from within a CrewAI agent.

First, we import the required CrewAI classes. We'll use Agent, Task, and Crew from CrewAI, and MCPServerAdapter to connect to our tool server.

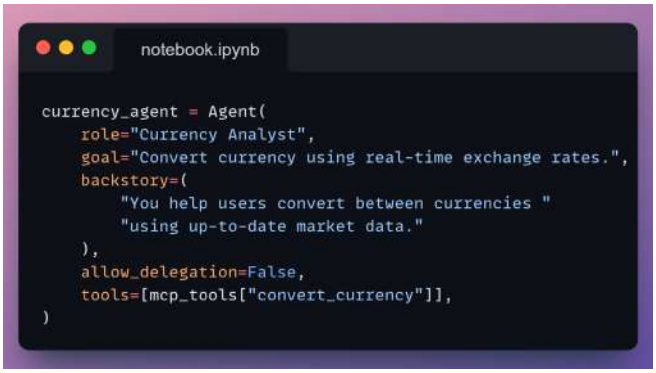


Next, we connect to the MCP tool server. Define the server parameters to connect to your running tool (from server.py).



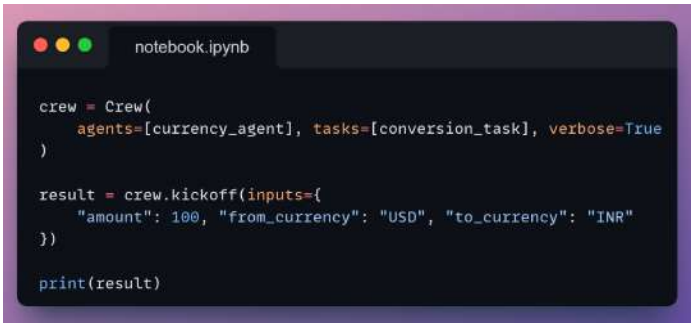
Now, we use the discovered MCP tool in an agent:

This agent is assigned the convert_currency tool from the remote server. It can now call the tool just like a locally defined one.

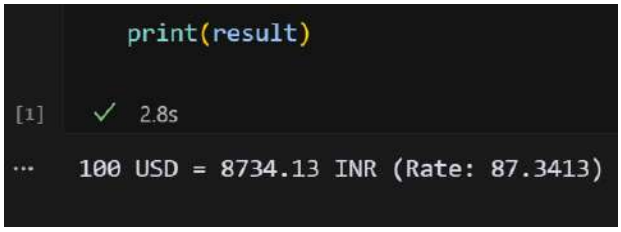


We give the agent a task description:

Finally, we create the Crew, pass in the inputs and run it:



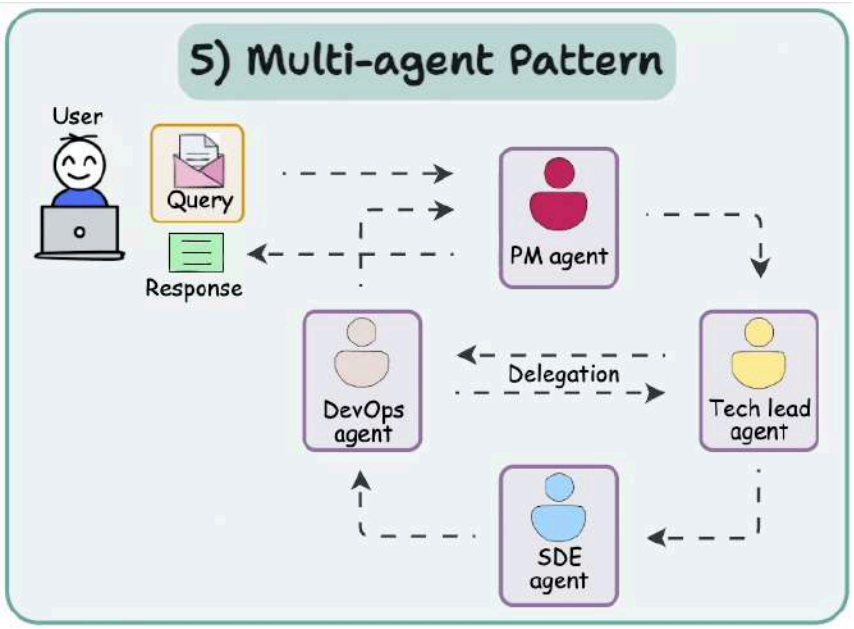
Printing the result, we get the following output:



4) Cooperation

Multi-agent systems work best when agents collaborate and exchange feedback.

Instead of one agent doing everything, a team of specialized agents can split tasks and improve each other's outputs.



Consider an AI-powered financial analysis system:

- One agent gathers data
- another assesses risk,
- a third builds strategy,
- and a fourth writes the report

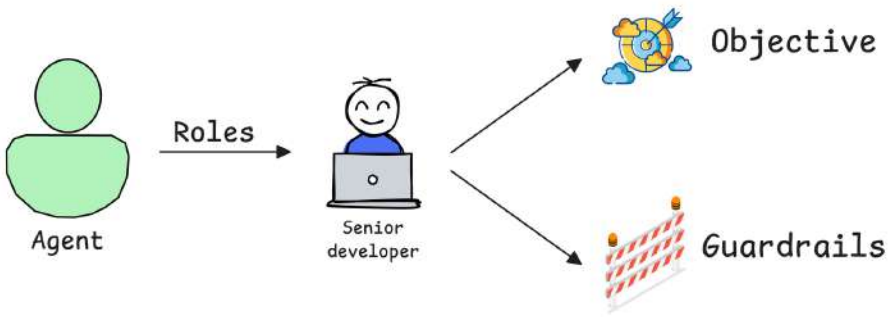
Collaboration leads to smarter, more accurate results.

The best practice is to enable agent collaboration by designing workflows where agents can exchange insights and refine their responses together.

5) Guardrails

Agents are powerful but without constraints, they can go off track. They might hallucinate, loop endlessly, or make bad calls.

Guardrails ensure that agents stay on track and maintain quality standards.



Examples of useful guardrails include:

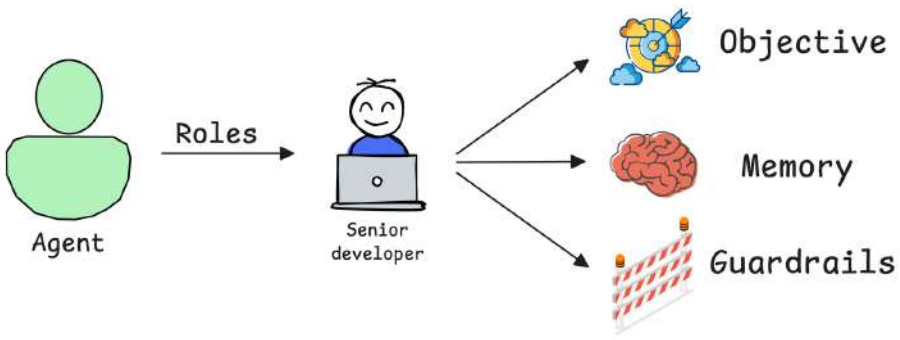
- Limiting tool usage: Prevent an agent from overusing APIs or generating irrelevant queries.
- Setting validation checkpoints: Ensure outputs meet predefined criteria before moving to the next step.
- Establishing fallback mechanisms: If an agent fails to complete a task, another agent or human reviewer can intervene.

For example, an AI-powered legal assistant should avoid outdated laws or false claims - guardrails ensure that.

6) Memory

Finally, we have memory, which is one of the most critical components of AI agents.

Without memory, an agent would start fresh every time, losing all context from previous interactions. With memory, agents can improve over time, remember past actions, and create more cohesive responses.



Different types of memory in AI agents include:

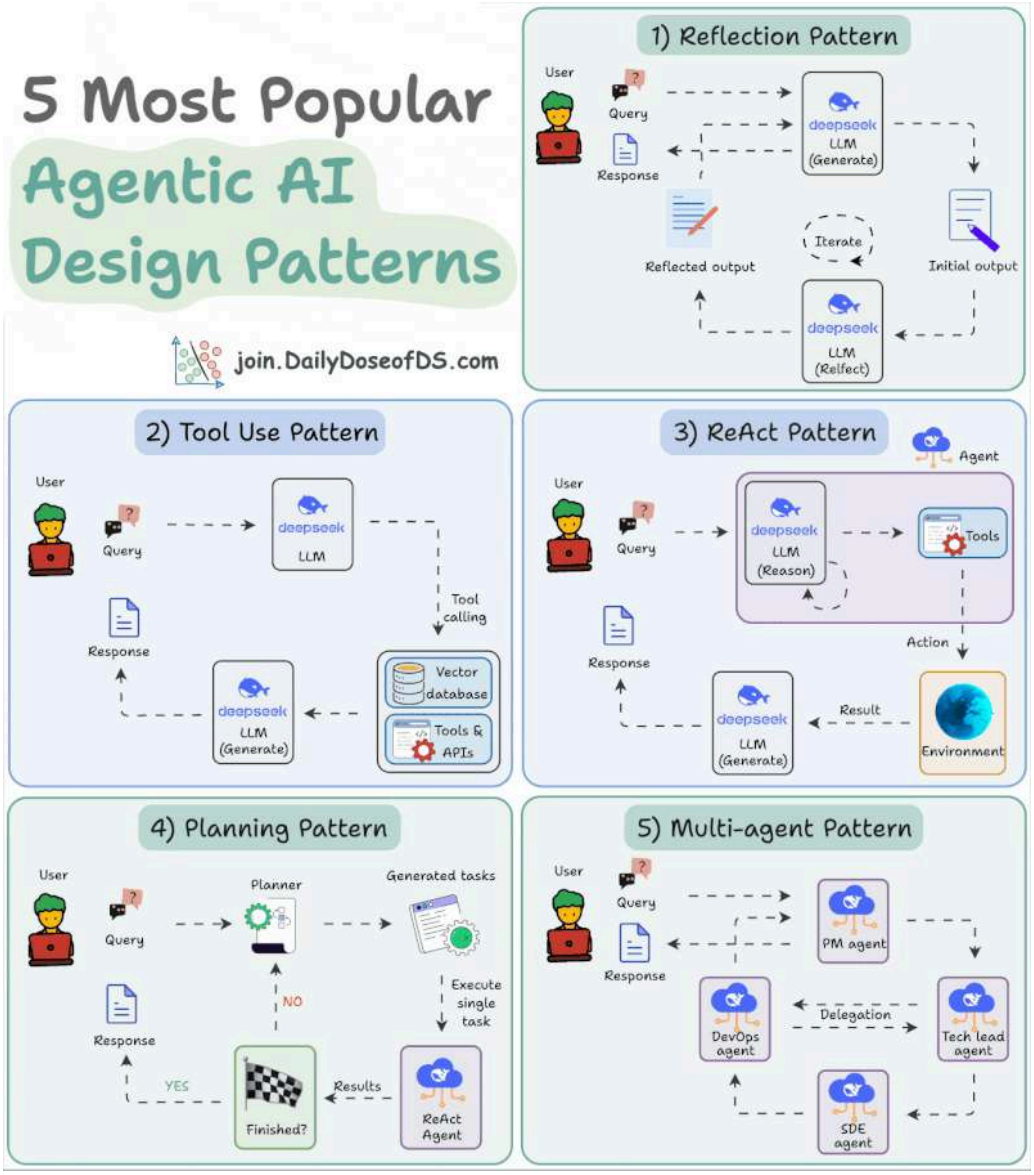
- Short-term memory – Exists only during execution (e.g., recalling recent conversation history).
- Long-term memory – Persists after execution (e.g., remembering user preferences over multiple interactions).
- Entity memory – Stores information about key subjects discussed (e.g., tracking customer details in a CRM agent).

For example, in an AI-powered tutoring system, memory allows the agent to recall past lessons, tailor feedback, and avoid repetition.

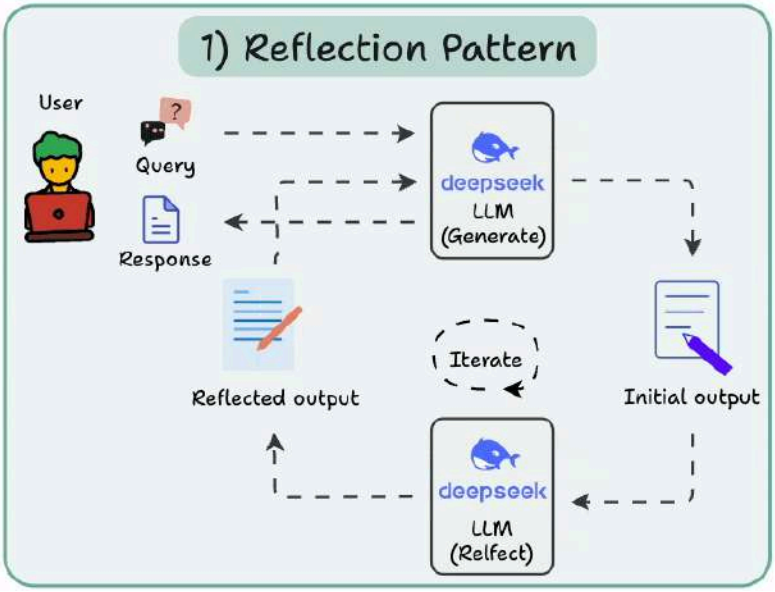
5 Agentic AI Design Patterns

Agentic behaviors allow LLMs to refine their output by incorporating self-evaluation, planning, and collaboration!

The following visual depicts the 5 most popular design patterns employed in building AI agents.

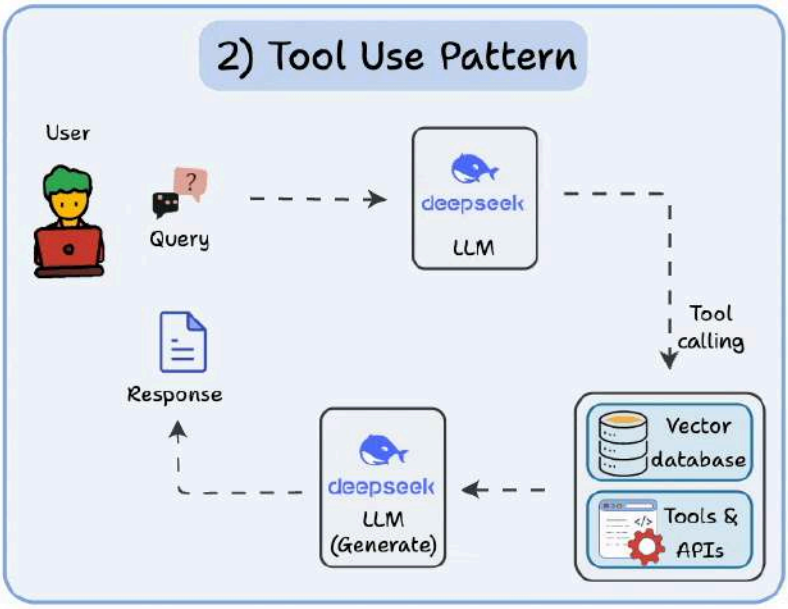


#1) Reflection pattern



The AI reviews its own work to spot mistakes and iterate until it produces the final response.

#2) Tool use pattern

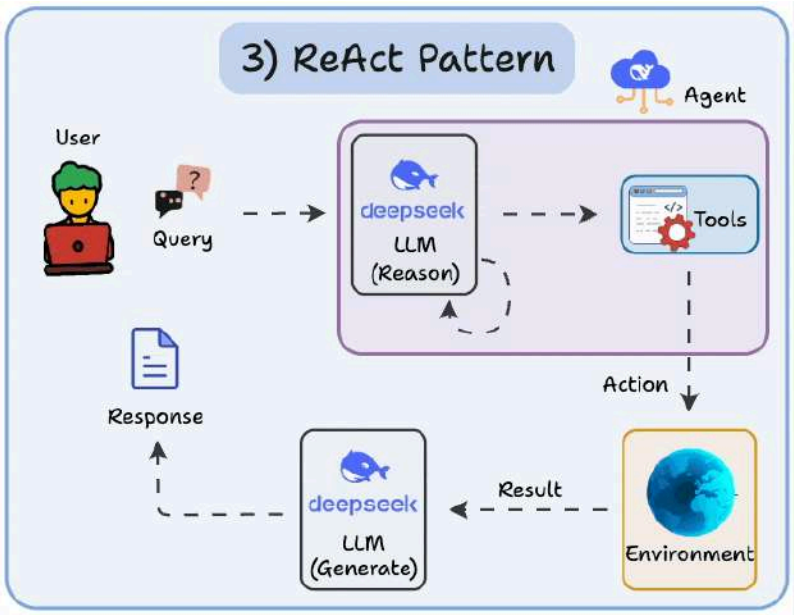


Tools allow LLMs to gather more information by:

- Querying a vector database
- Executing Python scripts
- Invoking APIs, etc.

This is helpful since the LLM is not solely reliant on its internal knowledge.

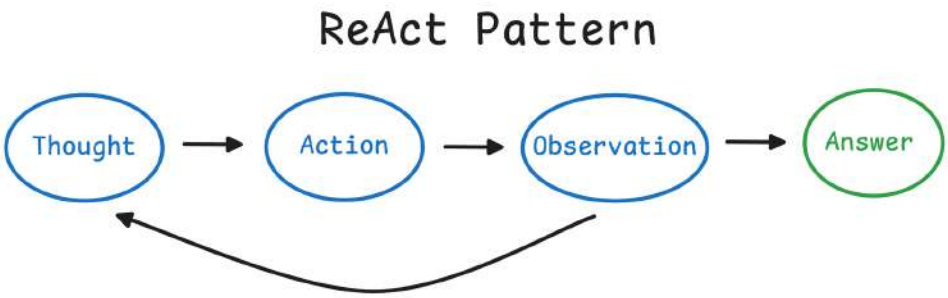
#3) ReAct (Reason and Act) pattern



ReAct combines the above two patterns:

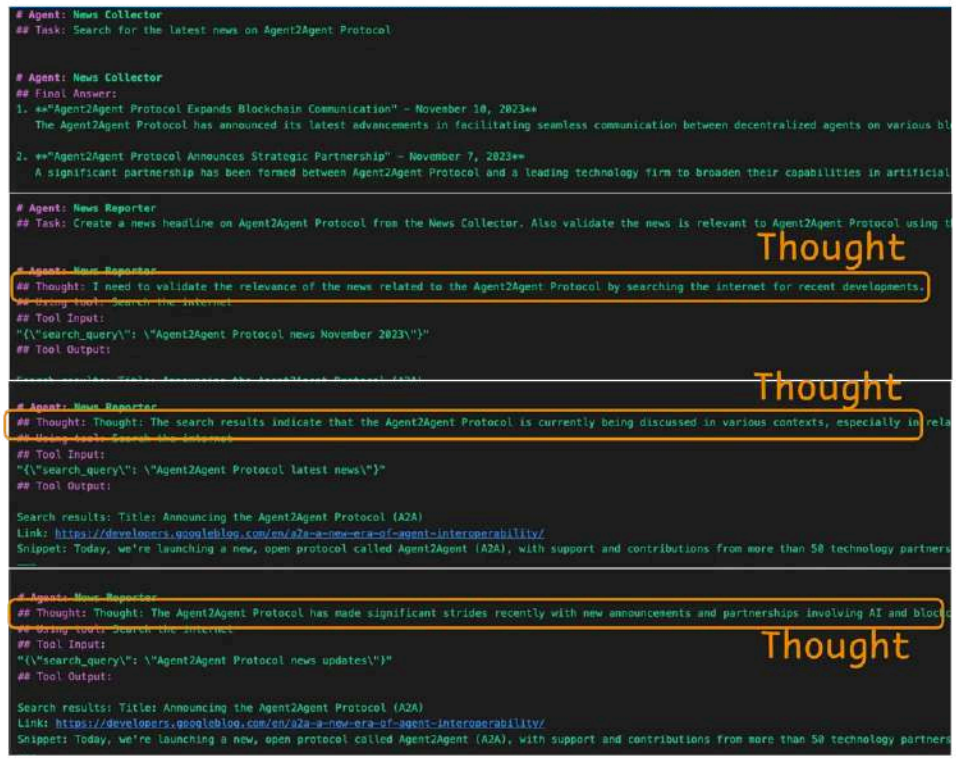
- The Agent reflects on the generated outputs.
- It interacts with the world using tools.

A ReAct agent operates in a loop of Thought → Action → Observation, repeating until it reaches a solution or a final answer. This is analogous to how humans solve problems:



Note: Frameworks like CrewAI primarily use this by default.

To understand this, consider the output of a multi-agent system below:



As shown above, the Agent is going through a series of thought activities before producing a response.

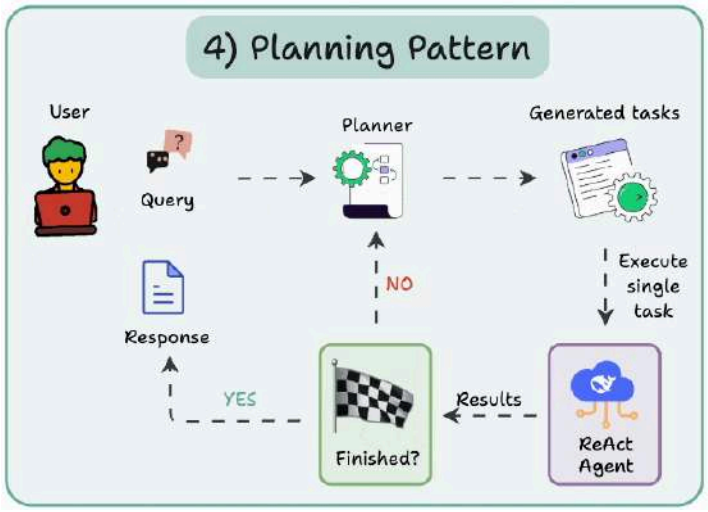
This is the ReAct pattern in action!

More specifically, under the hood, many such frameworks use the ReAct (Reasoning and Acting) pattern to let LLM think through problems and use tools to act on the world.

For example, an agent in CrewAI typically alternates between reasoning about a task and acting (using a tool) to gather information or execute steps, following the ReAct paradigm.

This enhances an LLM agent's ability to handle complex tasks and decisions by combining chain-of-thought reasoning with external tool use like in this ReAct implementation from scratch.

#4) Planning pattern



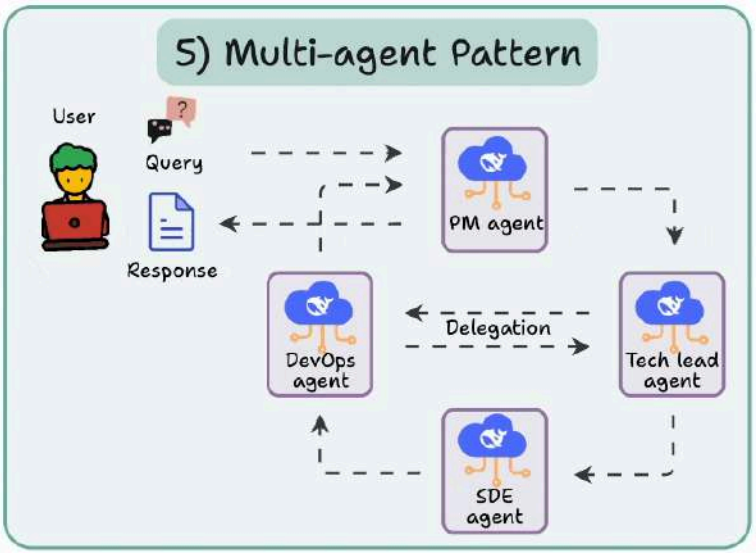
Instead of solving a task in one go, the AI creates a roadmap by:

- Subdividing tasks
- Outlining objectives

This strategic thinking solves tasks more effectively.

Note: In CrewAI, specify `planning=True` to use Planning.

#5) Multi-Agent pattern



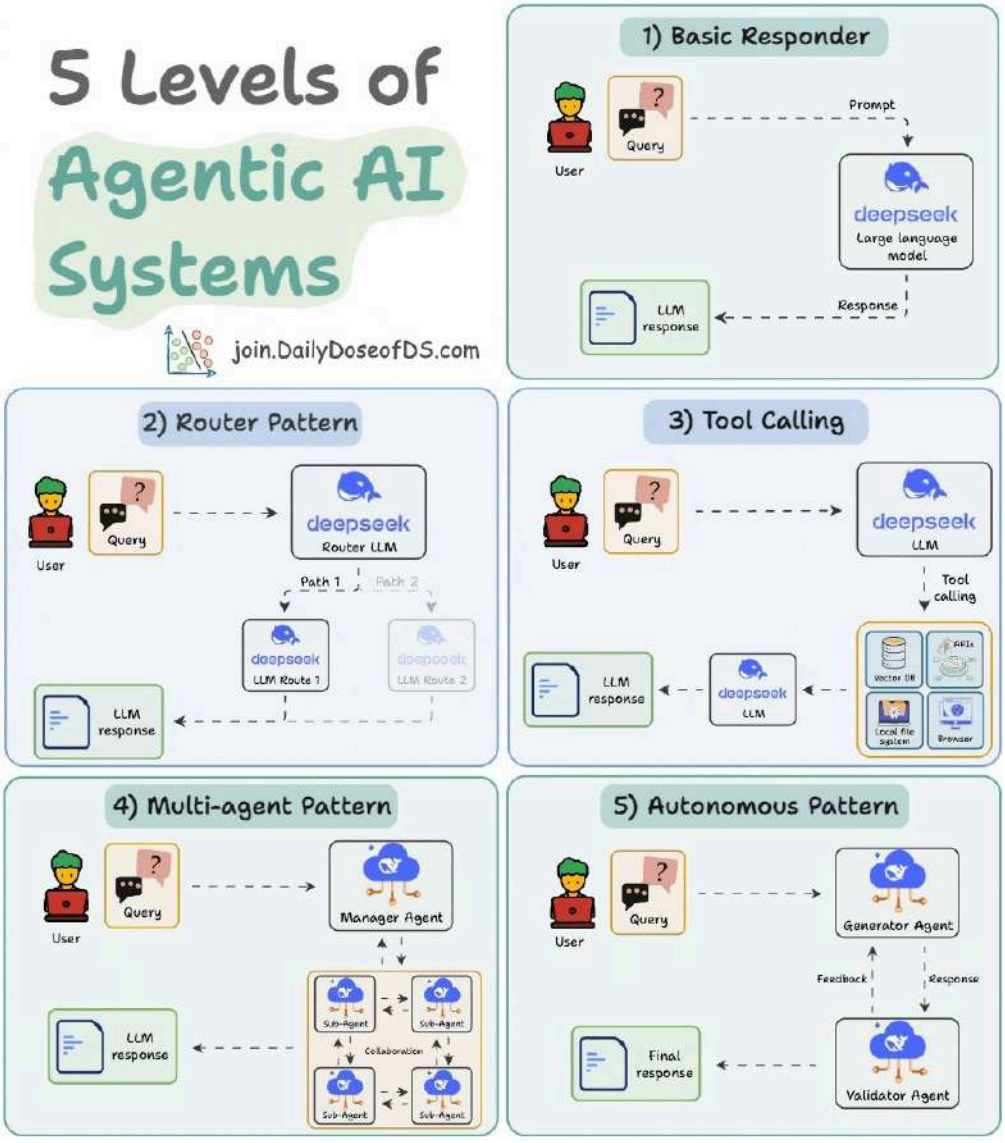
- There are several agents, each with a specific role and task.
- Each agent can also access tools.

All agents work together to deliver the final outcome, while delegating tasks to other agents if needed.

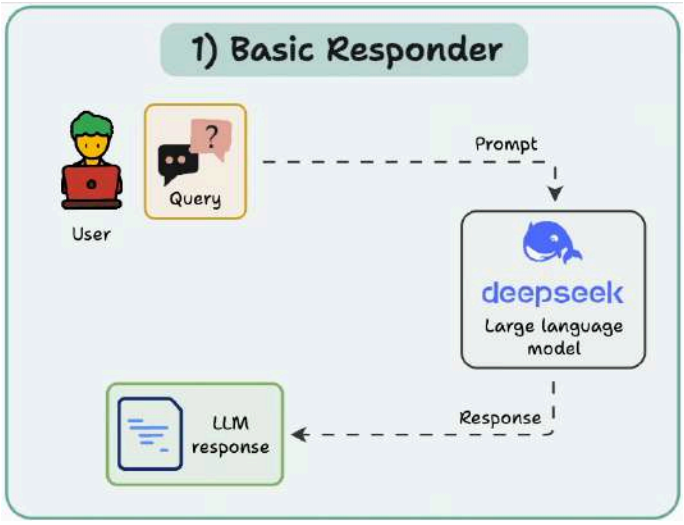
5 Levels of Agentic AI Systems

Agentic AI systems don't just generate text; they can make decisions, call functions, and even run autonomous workflows.

The visual explains 5 levels of AI agency—from simple responders to fully autonomous agents.



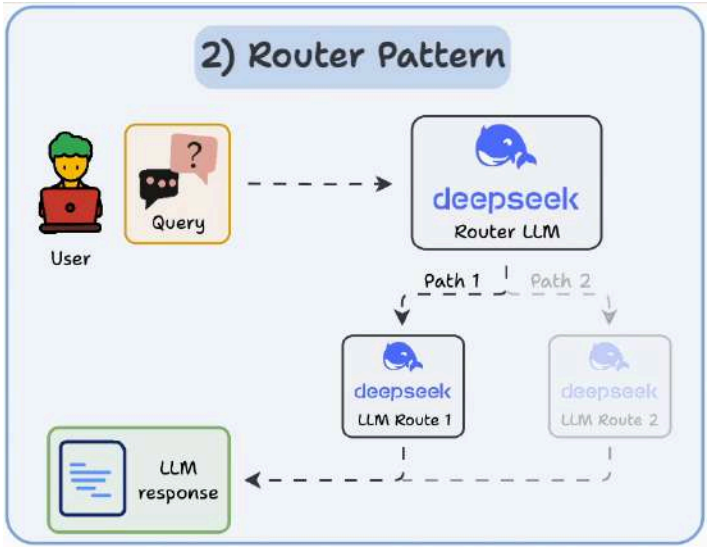
#1) Basic responder



A human guides the entire flow.

The LLM is just a generic responder that receives an input and produces an output. It has little control over the program flow.

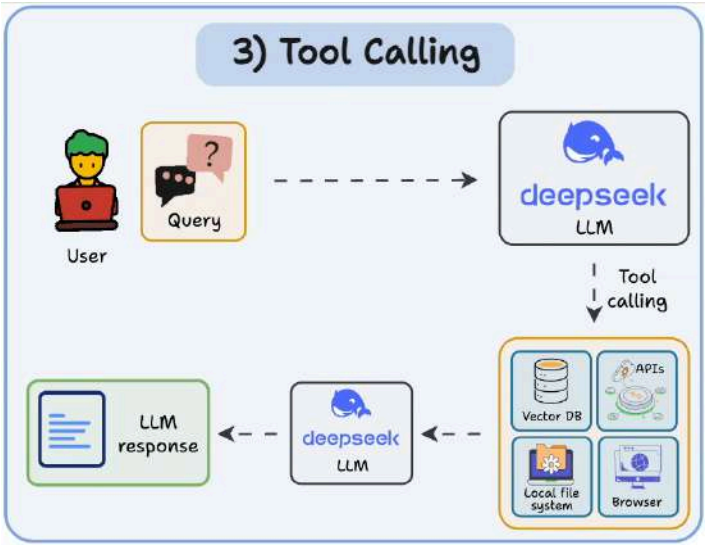
#2) Router pattern



A human defines the paths/functions that exist in the flow.

The LLM makes basic decisions on which function or path it can take.

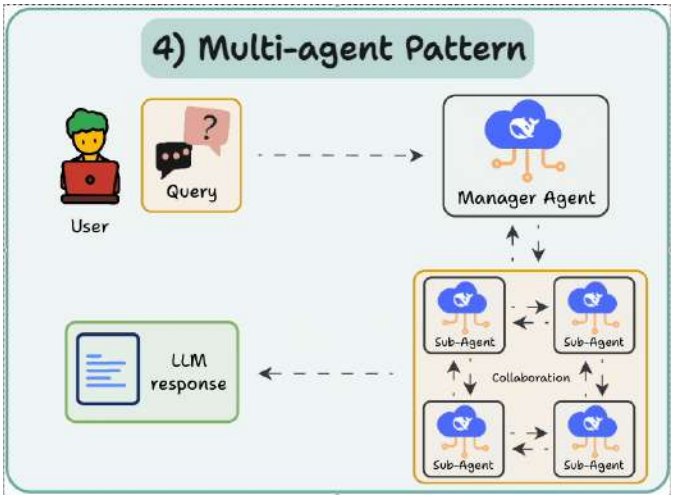
#3) Tool calling



A human defines a set of tools the LLM can access to complete a task.

LLM decides when to use them and also the arguments for execution.

#4) Multi-agent pattern

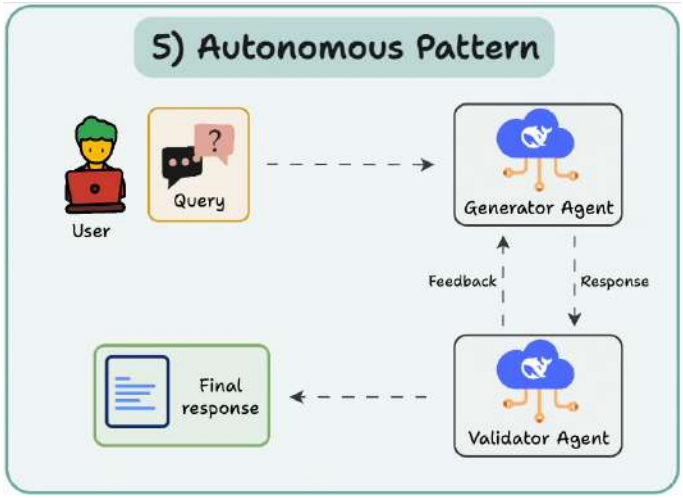


A manager agent coordinates multiple sub-agents and decides the next steps iteratively.

A human lays out the hierarchy between agents, their roles, tools, etc.

The LLM controls execution flow, deciding what to do next.

#5) Autonomous pattern

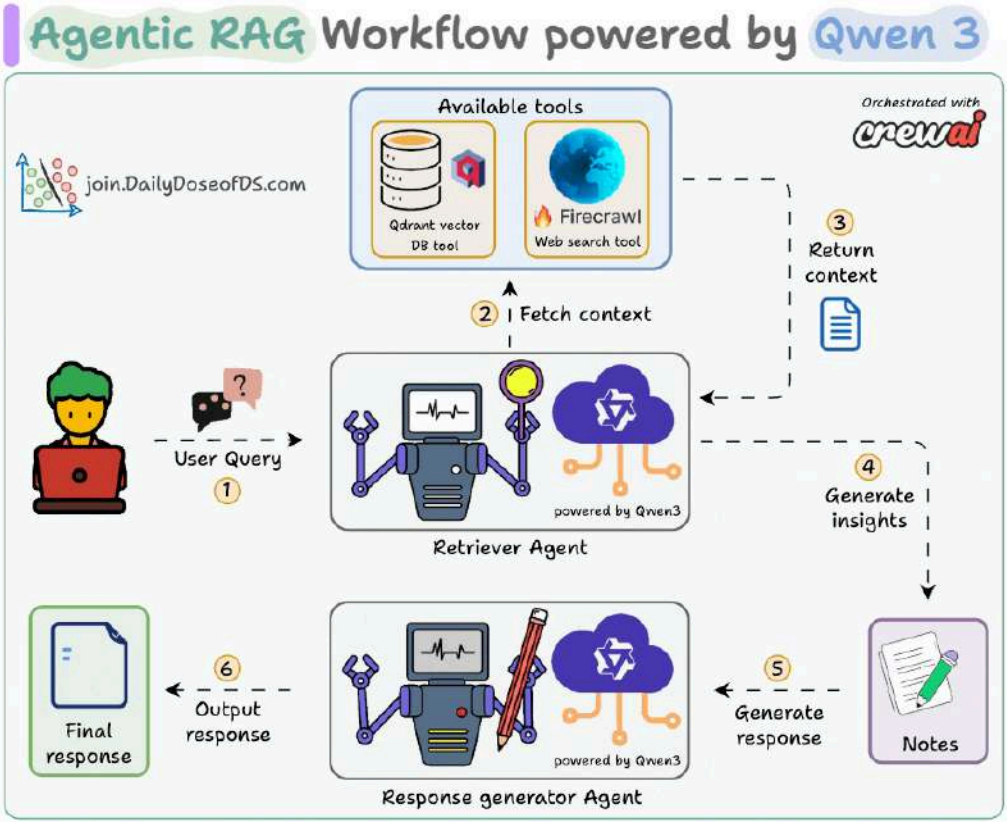


The most advanced pattern, wherein, the LLM generates and executes new code independently, effectively acting as an independent AI developer.

AI Agents Projects

#1) Agentic RAG

Build a RAG pipeline with agentic capabilities that can dynamically fetch context from different sources, like a vector DB and the internet.



Tech stack:

- CrewAI for Agent orchestration.
- Firecrawl for web search.
- LightningAI's LitServe for deployment.

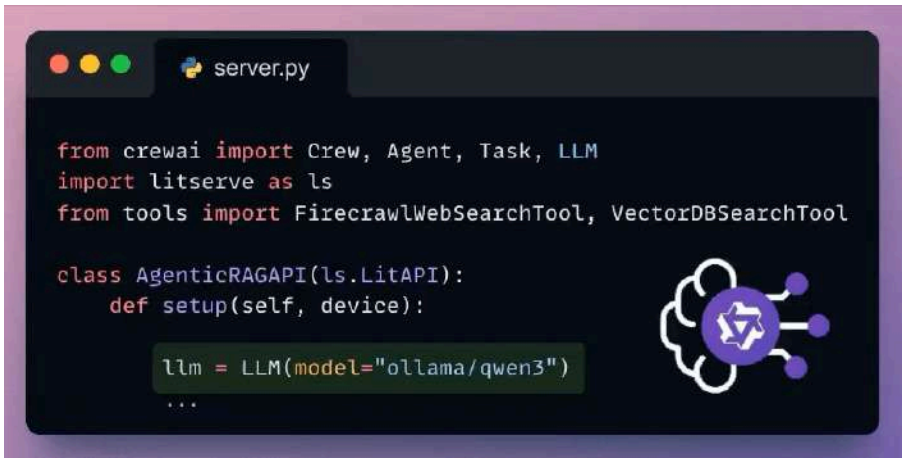
Workflow:

- The Retriever Agent accepts the user query.
- It invokes a relevant tool (Firecrawl web search or vector DB tool) to get context and generate insights.
- The Writer Agent generates a response.

Let's implement it!

#1) Set up LLM

CrewAI seamlessly integrates with all popular LLMs and providers. Here's how we set up a local Qwen 3 via Ollama:



#2) Define Research Agent and Task

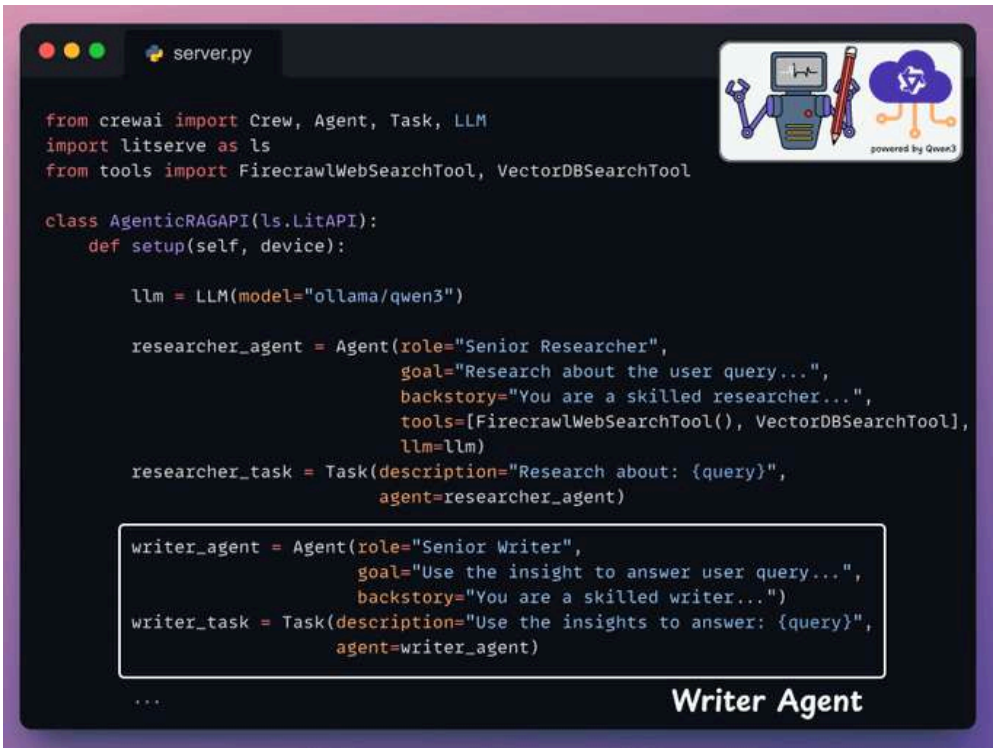
This Agent accepts the user query and retrieves the relevant context using a vectorDB tool and a web search tool powered by Firecrawl.

Again, put this in the LitServe setup() method:

#3) Define Writer Agent and Task

Next, the Writer Agent accepts the insights from the Researcher Agent to generate a response.

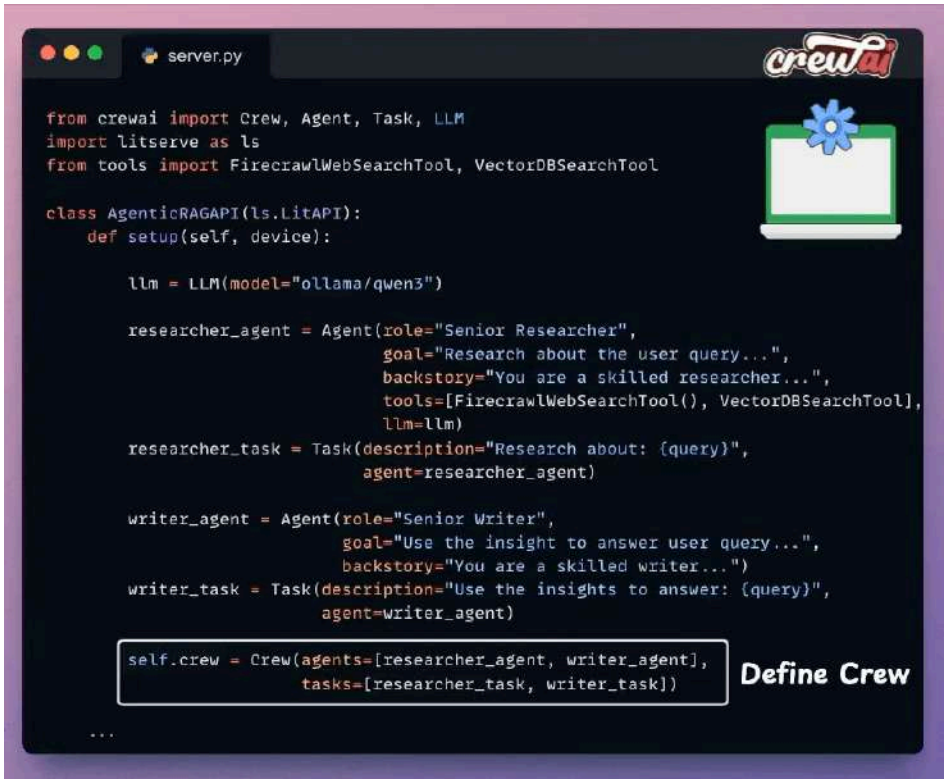
Yet again, we add this in the LitServe setup method:



#4) Set up the Crew

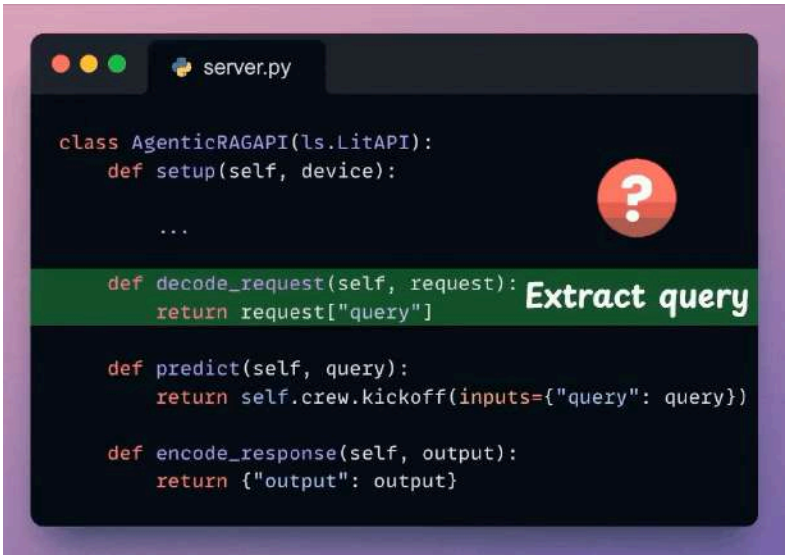
Once we have defined the Agents and their tasks, we orchestrate them into a crew using CrewAI and put that into a setup method.

Check this code:



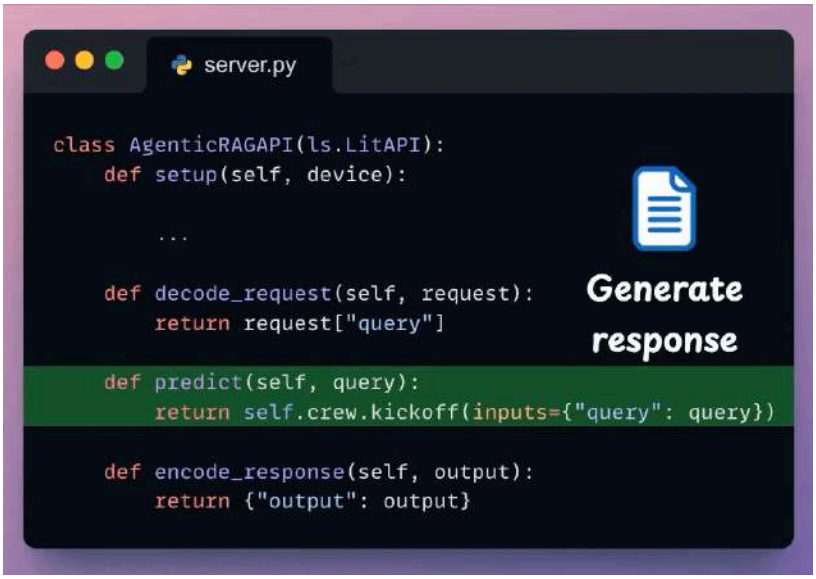
#5) Decode request

With that, we have orchestrated the Agentic RAG workflow, which will be executed upon an incoming request. Next, from the incoming request body, we extract the user query. Check the highlighted code below:



#6) Predict

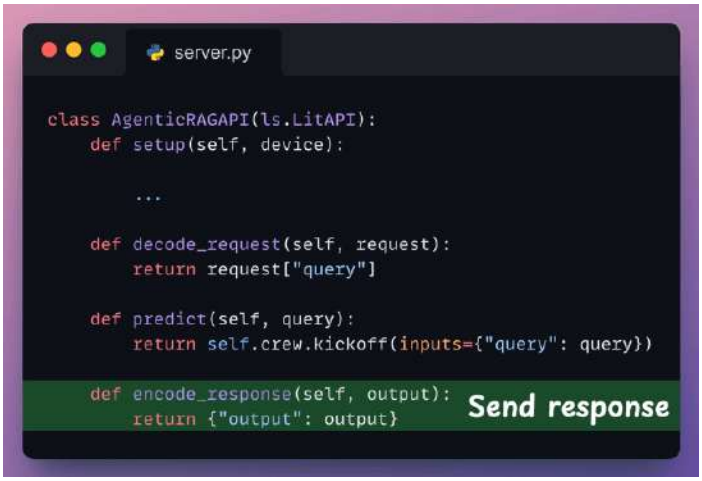
We use the decoded user query and pass it to the Crew defined earlier to generate a response from the model. Check the highlighted code below:



#7) Encode response

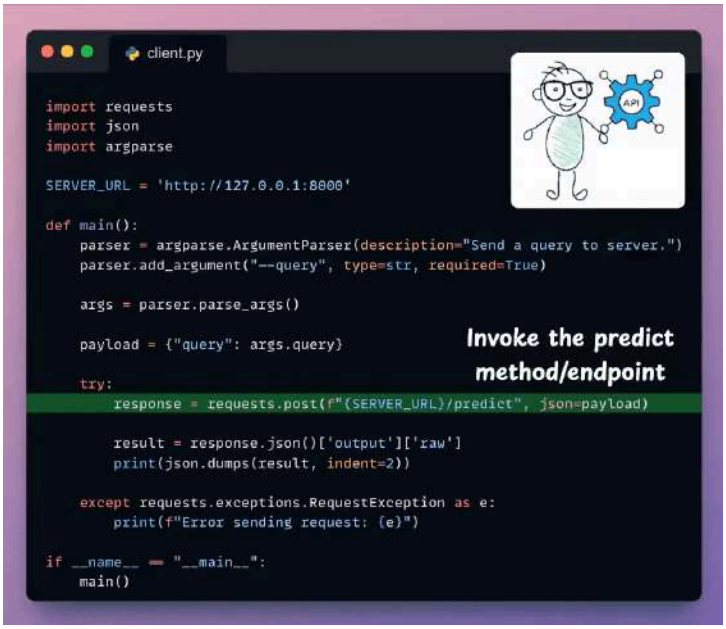
Here, we can post-process the response & send it back to the client.

Note: LitServe internally invokes these methods in order: decode_request → predict → encode_request. Check the highlighted code below:



#8) With that, we are done with the server code.

Next, we have the basic client code to invoke the API we created using the requests Python library. Check this:

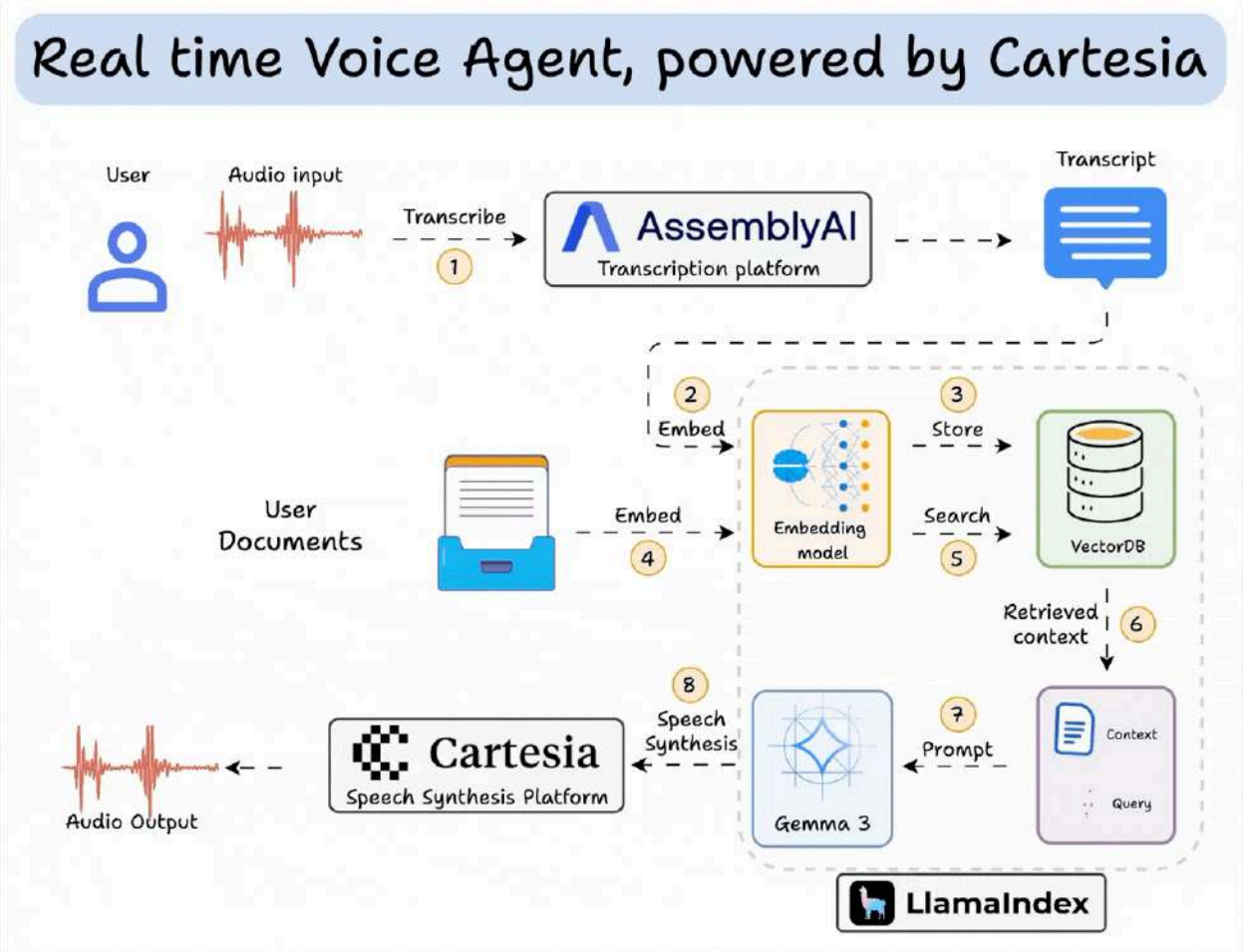


Done! We have deployed our fully private Qwen 3 Agentic RAG using LitServe.

The code is available here: https://www.dailydoseofds.com/p/deploy-a-qwen-3-agentic-rag/

#2) Voice RAG Agent

Real-time voice interactions are becoming more and more popular in AI apps. Learn how to build a real-time Voice RAG Agent, step-by-step.



Tech stack:

- CartesiaAI for SOTA text-to-speech
- AssemblyAI for speech-to-text
- LlamaIndex to power RAG
- Livekit for orchestration

Workflow:

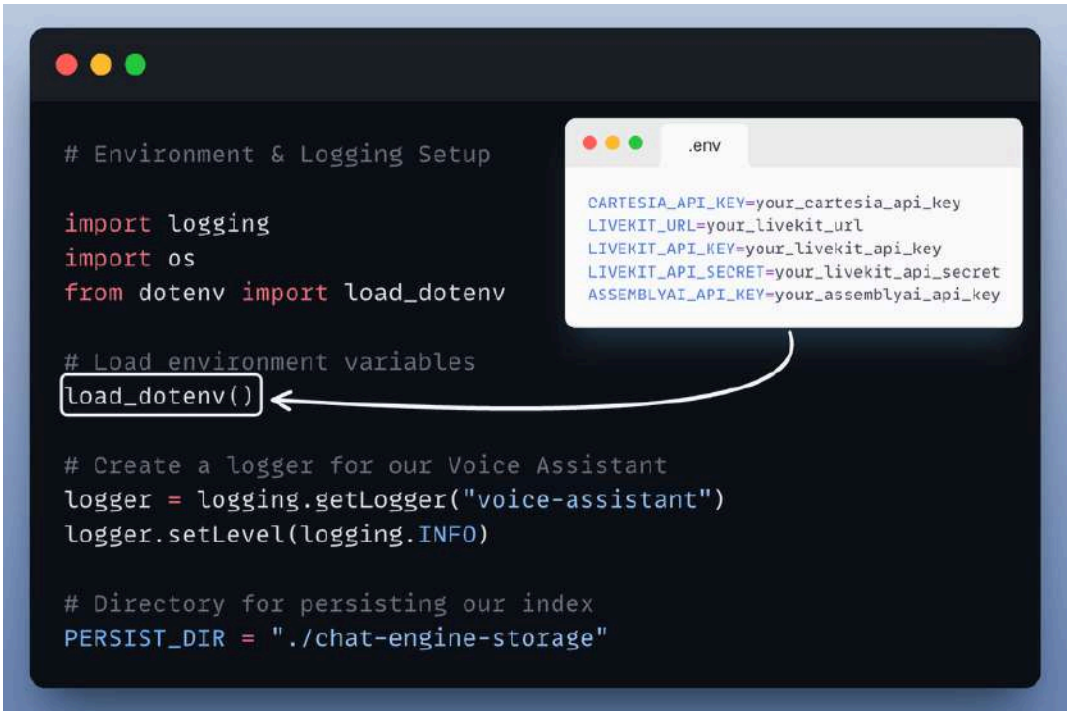
- Listens to real-time audio
- Transcribes it via AssemblyAI
- Uses your docs (via LlamaIndex) to craft an answer
- Speaks that answer back with Cartesia

Let's implement this!

#1) Set up environment and logging

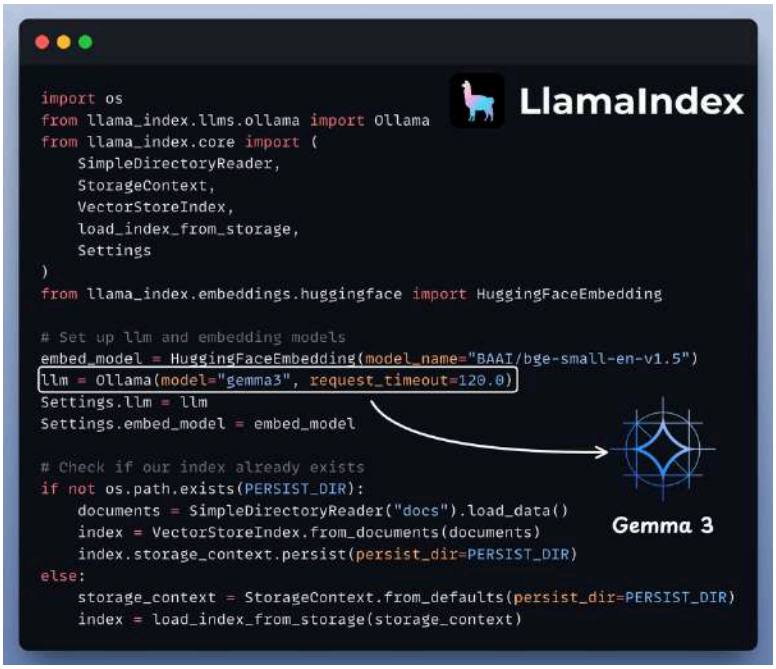
This ensures we can load configurations from .env and keep track of everything in real time.

Check this out:



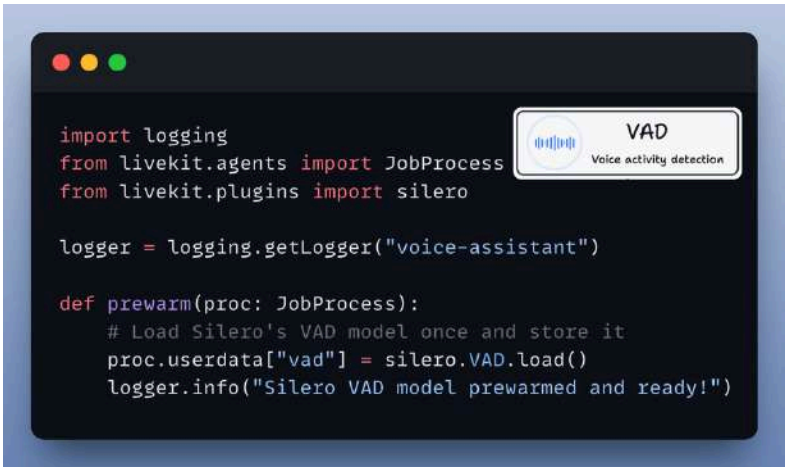
#2) Setup RAG

This is where your documents get indexed for search and retrieval, powered by LlamaIndex. The agent's answers would be grounded to this knowledge base.



#3) Setup Voice Activity Detection

We also want Voice Activity Detection (VAD) for smooth real-time experience—so we'll "prewarm" the Silero VAD model. This helps us detect when someone is actually speaking. Check this out:

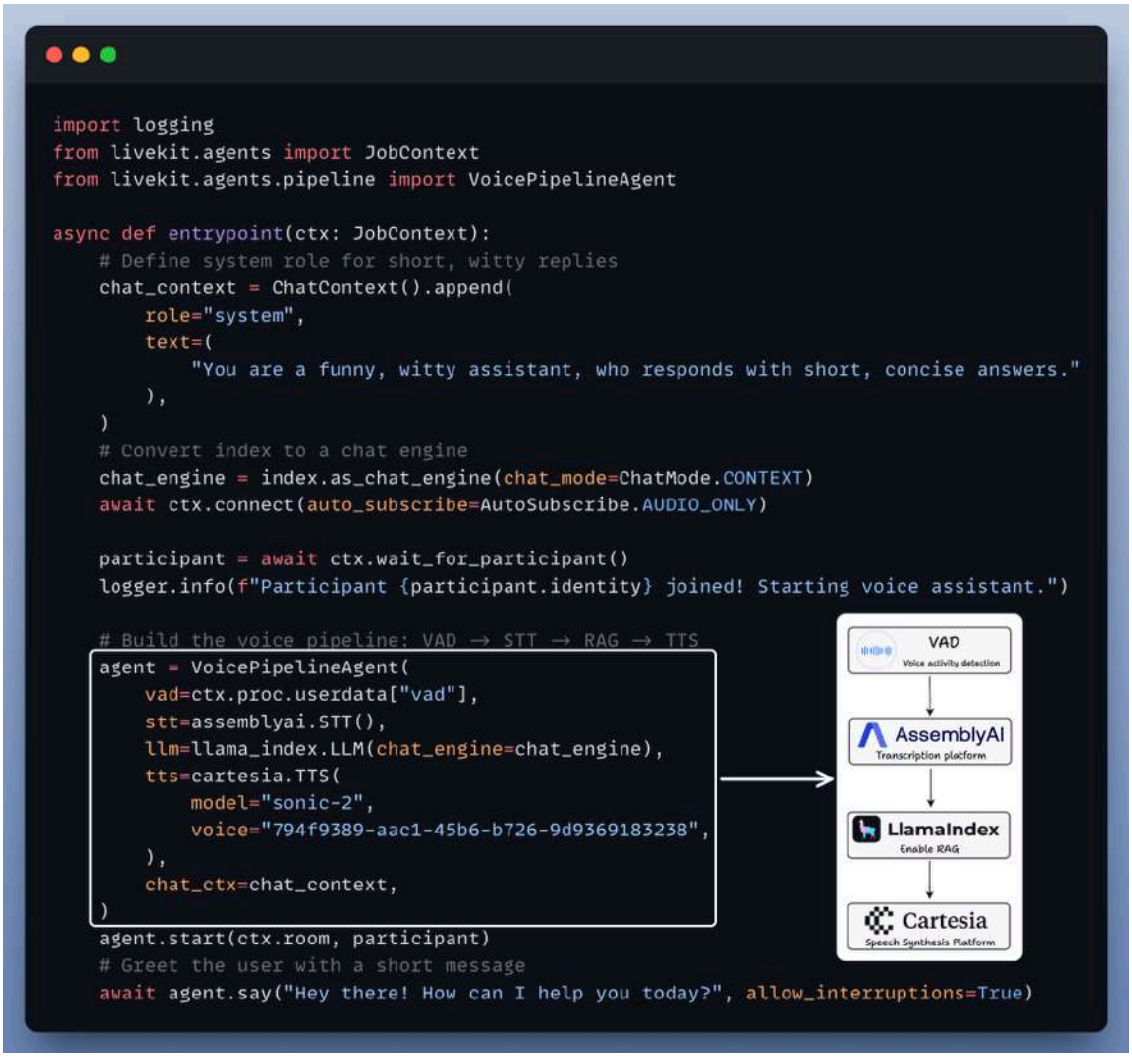


#4) The VoicePipelineAgent and Entry Point

This is where we bring it all together. The agent:

1. Listens to real-time audio.
2. Transcribes it using AssemblyAI.
3. Craft an answer with your documents via LlamaIndex.
4. Speaks that answer back using Cartesia.

Check this out:



#5) Run the app

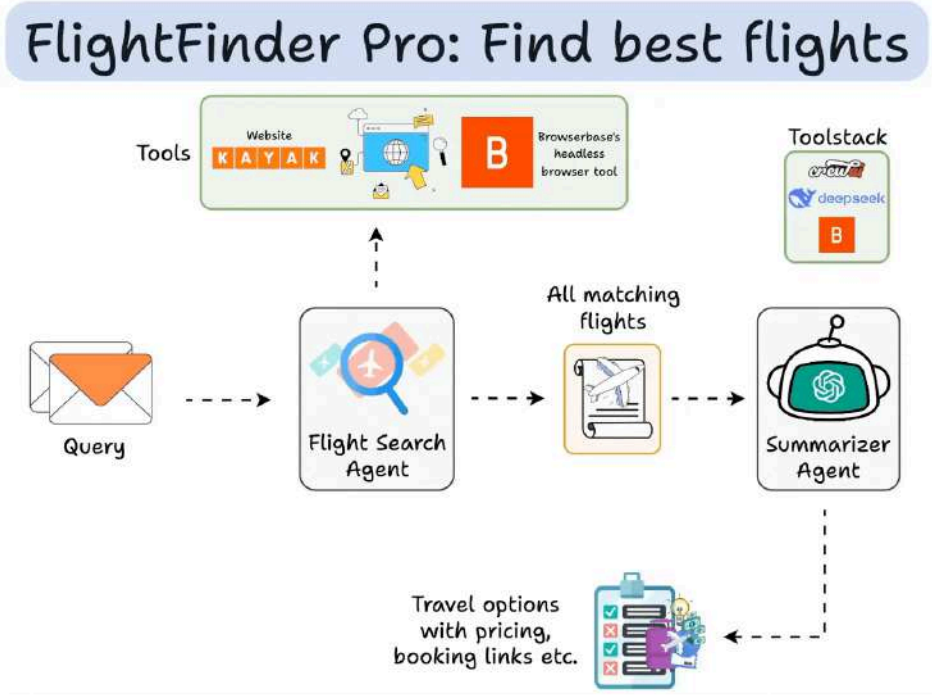
Finally, we tie it all together. We run our agent with, specifying the prewarm function and main entrypoint.

That's it—your Real-Time Voice RAG Agent is ready to roll!

The code is available here: https://www.dailydoseofds.com/p/building-a-real-time-voice-rag-agent/

#3) Multi-agent Flight finder

Build a flight search pipeline with agentic capabilities that can parse natural language queries and fetch live results from Kayak.



Tech stack:

- CrewAIInc for multi-agent orchestration
- BrowserbaseHQ's headless browser tool
- Ollama to locally serve DeepSeek-R1

Workflow:

- Parse the query (SF to New York on 21st September) to create a Kayak search URL
- Visit the URL and extract top 5 flights
- For each flight, go to the details to find available airlines
- Summarize flight info

Let's implement this!

#1) Define LLM

CrewAI nicely integrates with all the popular LLMs and providers out there. Here's how you set up a local DeepSeek using Ollama:

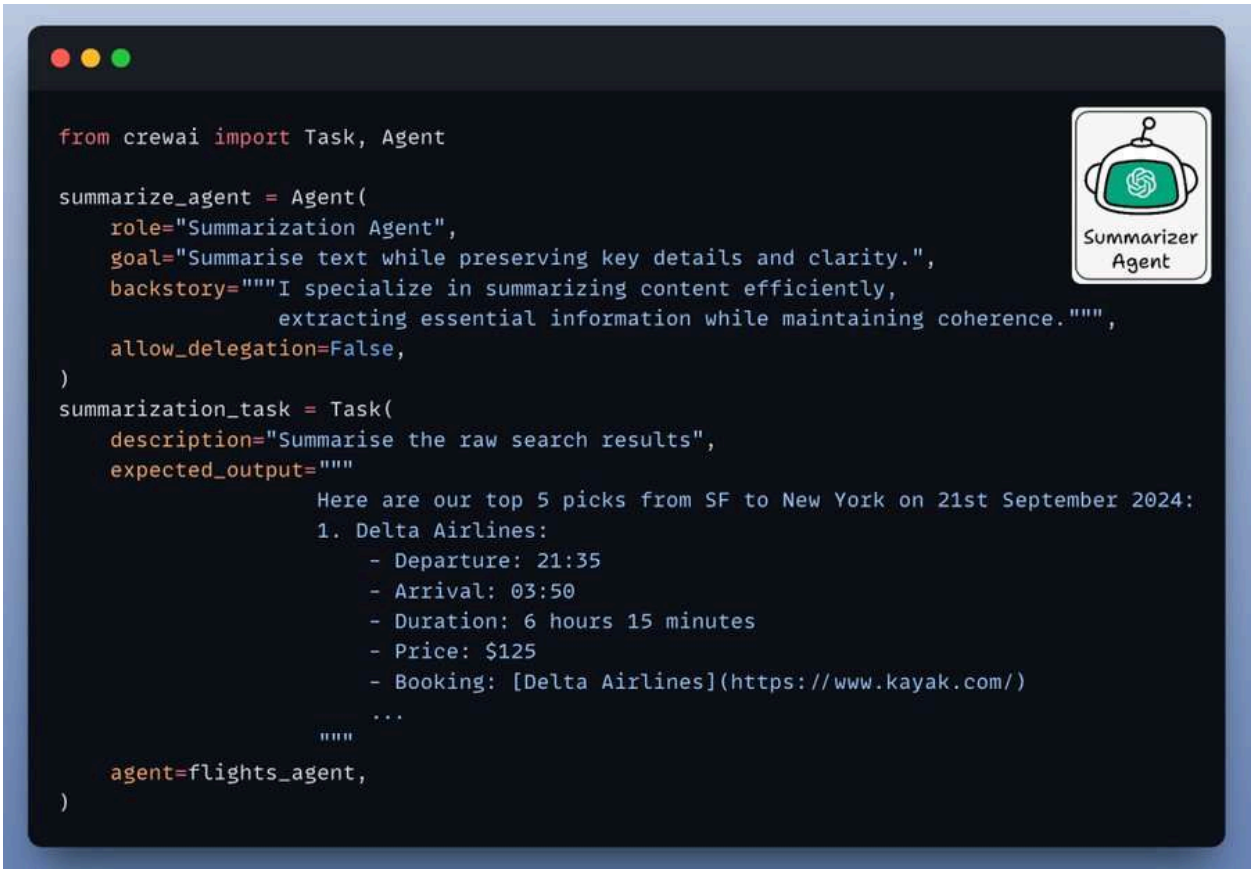
#2) Flight Search Agent

This agent mimics a real human searching for flights by browsing the web, powered by Browserbase's headless-browser tool and can look up flights on sites like Kayak.

#3) Summarisation Agent

After retrieving the flight details, we need a concise summary of all available options.

This is where our Summarization Agent steps in to make sense of the results for easy reading.

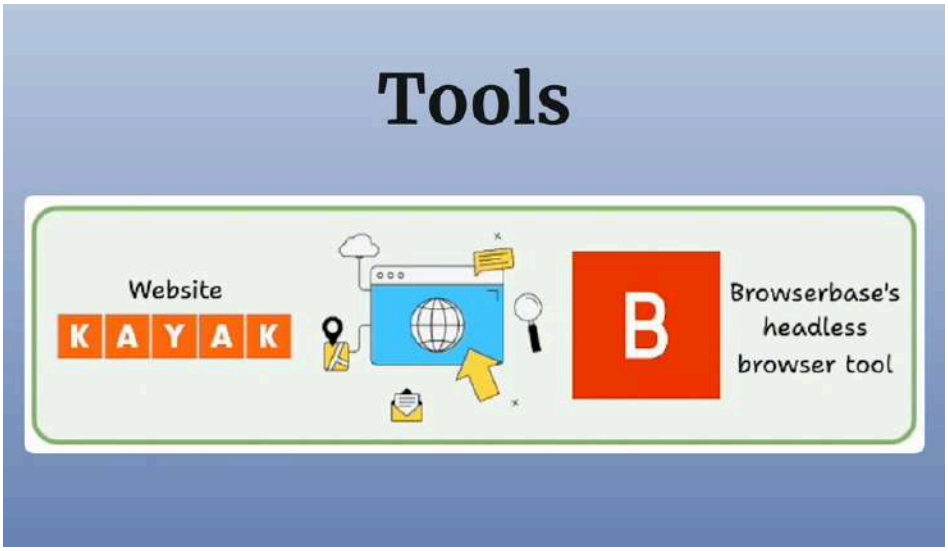



```
from crewai import Task, Agent

summarize_agent = Agent(
    role="Summarization Agent",
    goal="Summarise text while preserving key details and clarity.",
    backstory="""I specialize in summarizing content efficiently,
                 extracting essential information while maintaining coherence.""",
    allow_delegation=False,
)
summarization_task = Task(
    description="Summarise the raw search results",
    expected_output="""
                    Here are our top 5 picks from SF to New York on 21st September 2024:
                    1. Delta Airlines:
                        - Departure: 21:35
                        - Arrival: 03:50
                        - Duration: 6 hours 15 minutes
                        - Price: $125
                        - Booking: [Delta Airlines](https://www.kayak.com/)
                        ...
                    """
    agent=flights_agent,
)
```

Now that we have both our agents ready, it's time to understand the tools powering them.

1. Kayak tool
2. Browserbase tool

Let's write their code one-by-one.



#4) Kayak tool

A custom Kayak tool to translate the user input into a valid Kayak search URL.

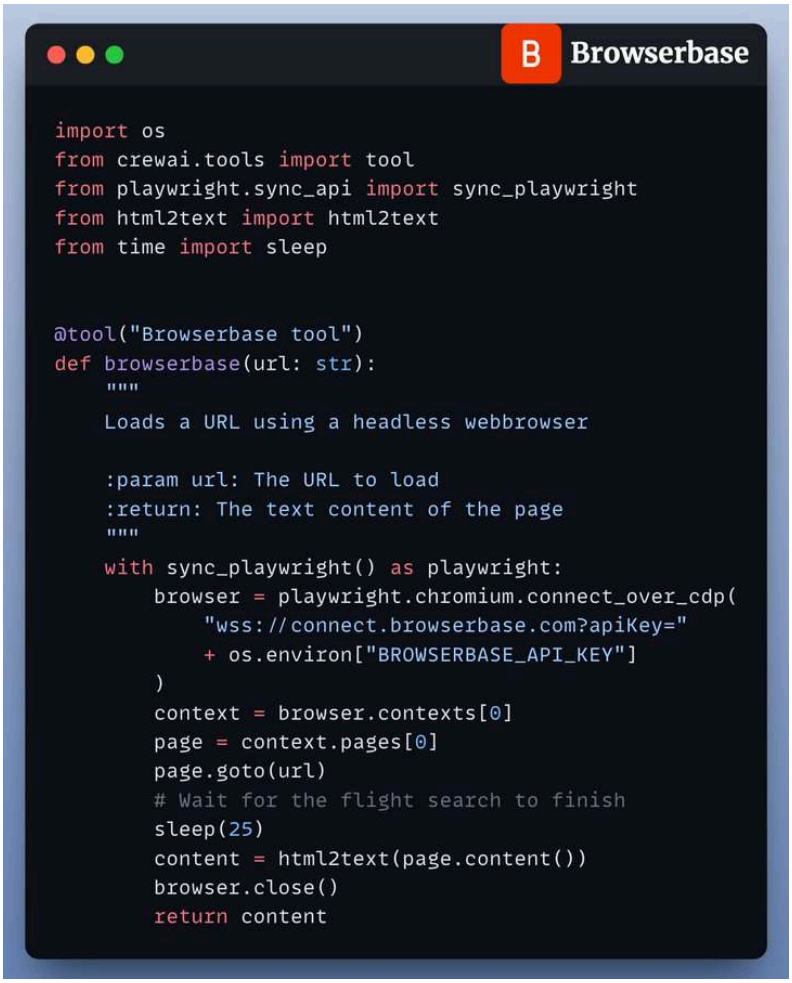
(FYI Kayak is a popular fight and hotel booking)

```python
from crewai.tools import tool
from typing import Optional

@tool("Kayak tool")
def kayak_search(
    departure: str, destination: str, date: str, return_date: Optional[str] = None
) -> str:
    """
    Generates a Kayak URL for flights between departure and destination on the specified date.

    :param departure: The IATA code for the departure airport (e.g., 'SOF' for Sofia)
    :param destination: The IATA code for the destination airport (e.g., 'BER' for Berlin)
    :param date: The date of the flight in the format 'YYYY-MM-DD'
    :return_date: Only for two-way tickets. The date of return flight in the format 'YYYY-MM-DD'
    :return: The Kayak URL for the flight search
    """
    print(f"Generating Kayak URL for {departure} to {destination} on {date}")
    URL = f"https://www.kayak.com/flights/{departure}-{destination}/{date}"
    if return_date:
        URL += f"/{return_date}"
    URL += "?currency=USD"
    return URL

# Export the decorated function
kayak = kayak_search
```

#5) Browserbase Tool

The flight search agent uses the Browserbase tool to simulate human browsing and gather flight data.

To be precise it automatically navigates the Kayak website and interacts with the web page. Check this out:



#6) Setup Crew

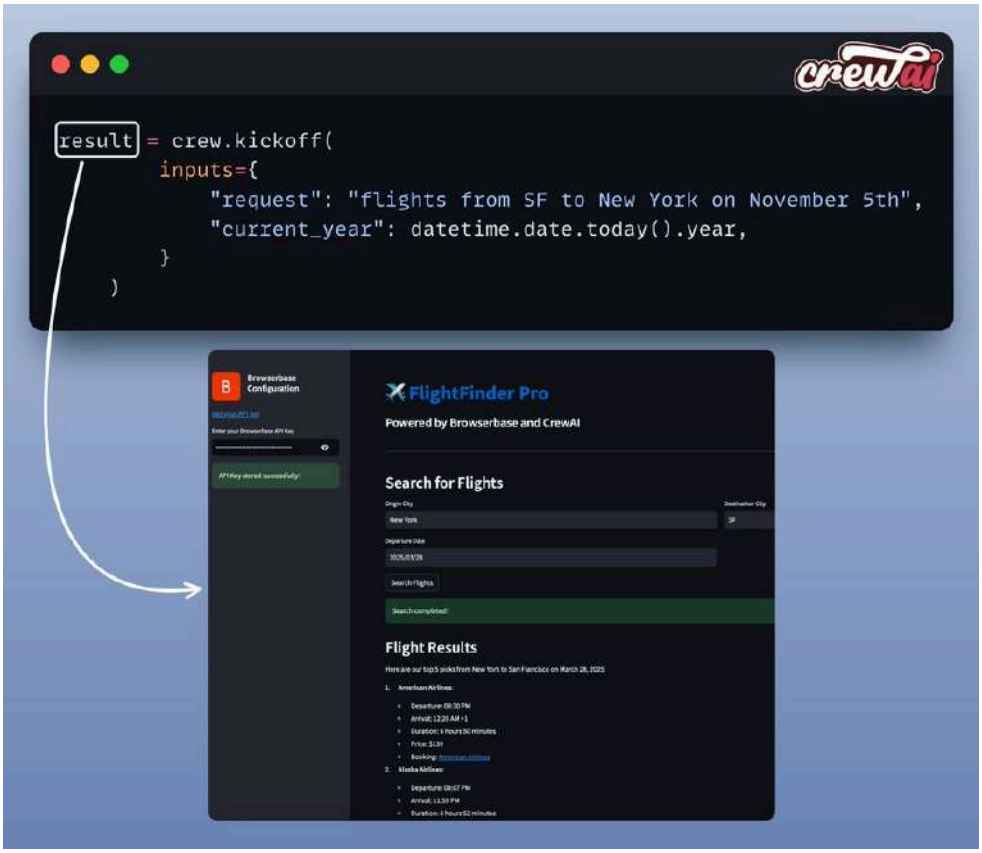
Once the agents and tools are defined, we orchestrate them using CrewAI.

Define their tasks, sequence their actions, and watch them collaborate in real time. Check this out:

Note: here the 'planning = True' is using the planning pattern we discussed in the 5 Agentic AI Design Patterns above.

#7) Kickoff and results

Finally, we feed the user's request (departure city, arrival city, travel dates) into the Crew and let it run:

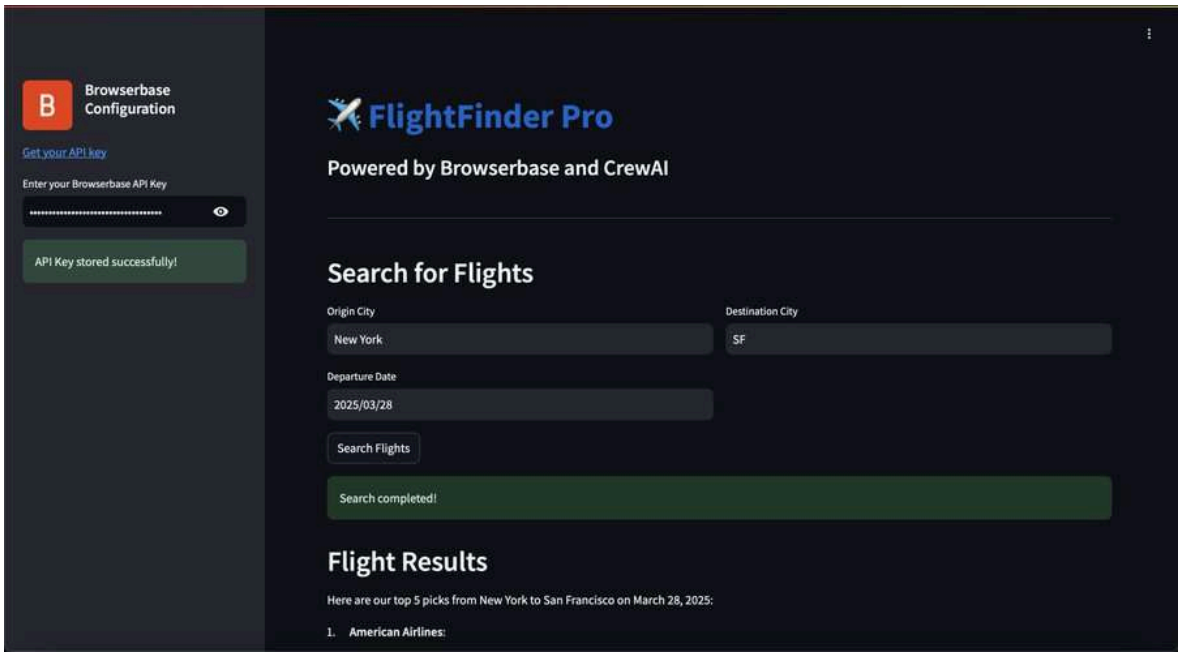


Streamlit UI

To make this accessible, we wrapped the entire system in a Streamlit interface.

It's a simple chat-like UI where you enter your flight details and see the results in real time.

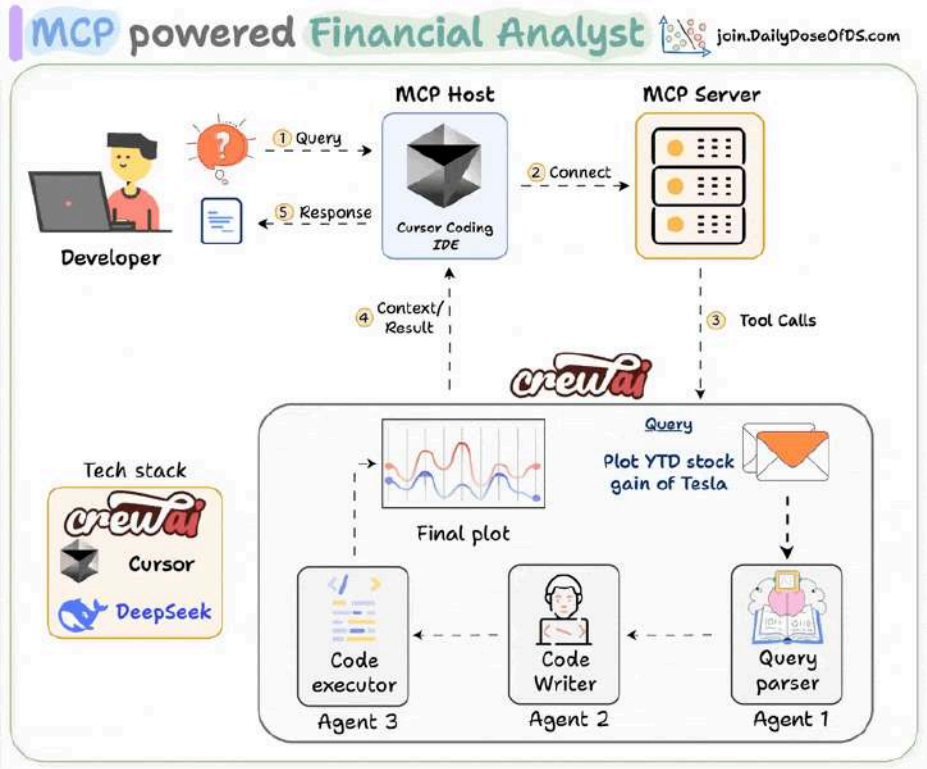
Check this out:



The code is available here:
https://blog.dailydoseofds.com/p/hands-on-a-multi-agent-flight-finder/

#4) Financial Analyst

Build an 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:

- The 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.

Let's implement this!

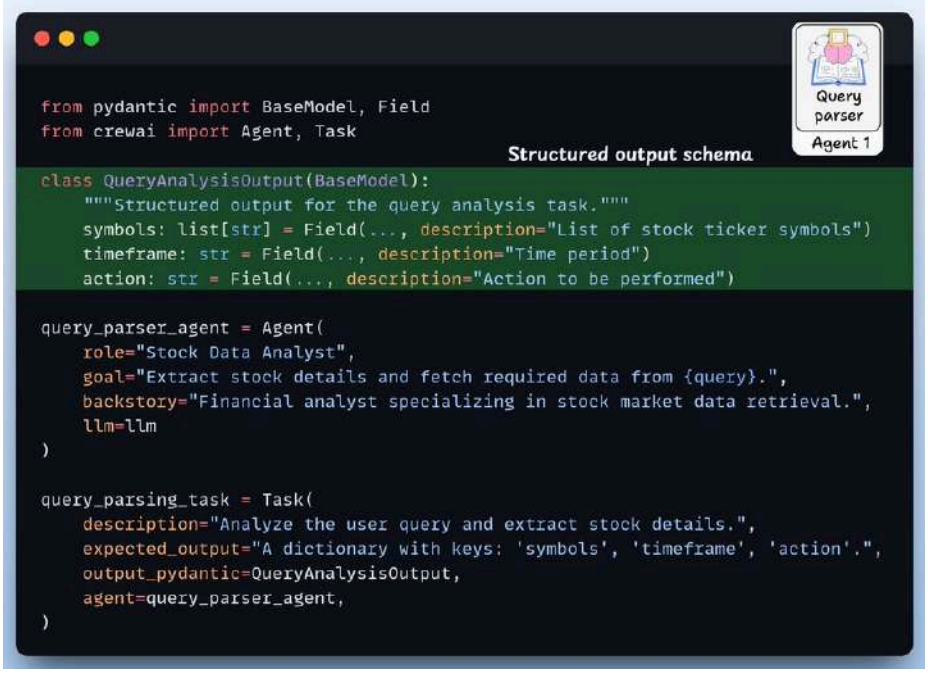
#1) Setup LLM

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!



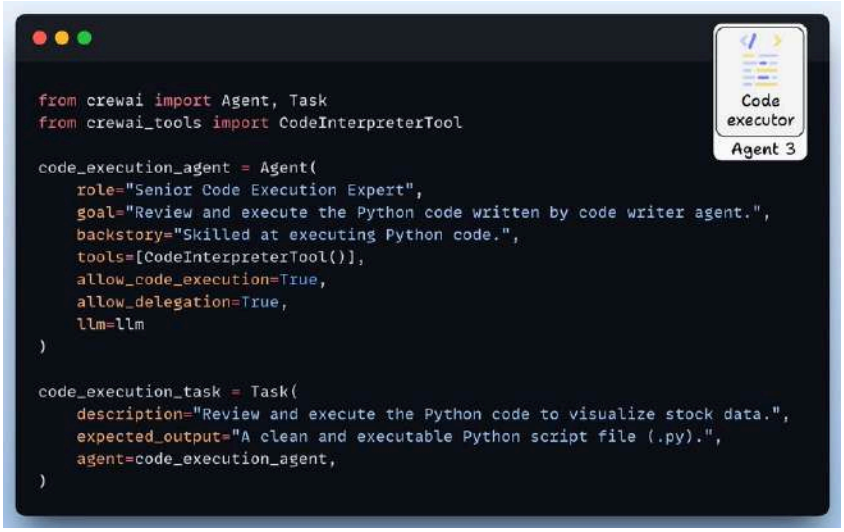
#3) Code Writer Agent

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

#4) Code Executor Agent

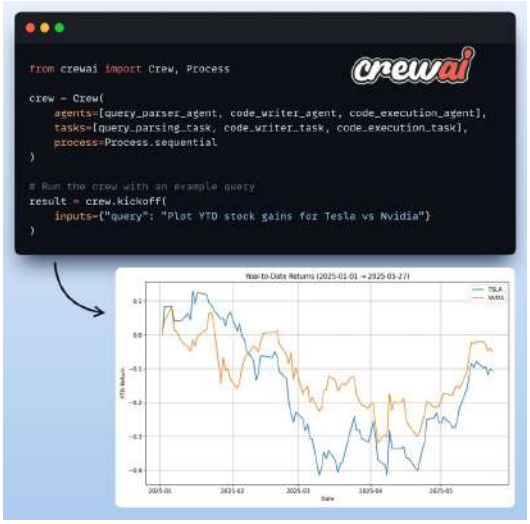
This agent reviews and executes the generated Python code for stock data visualization.

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



#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

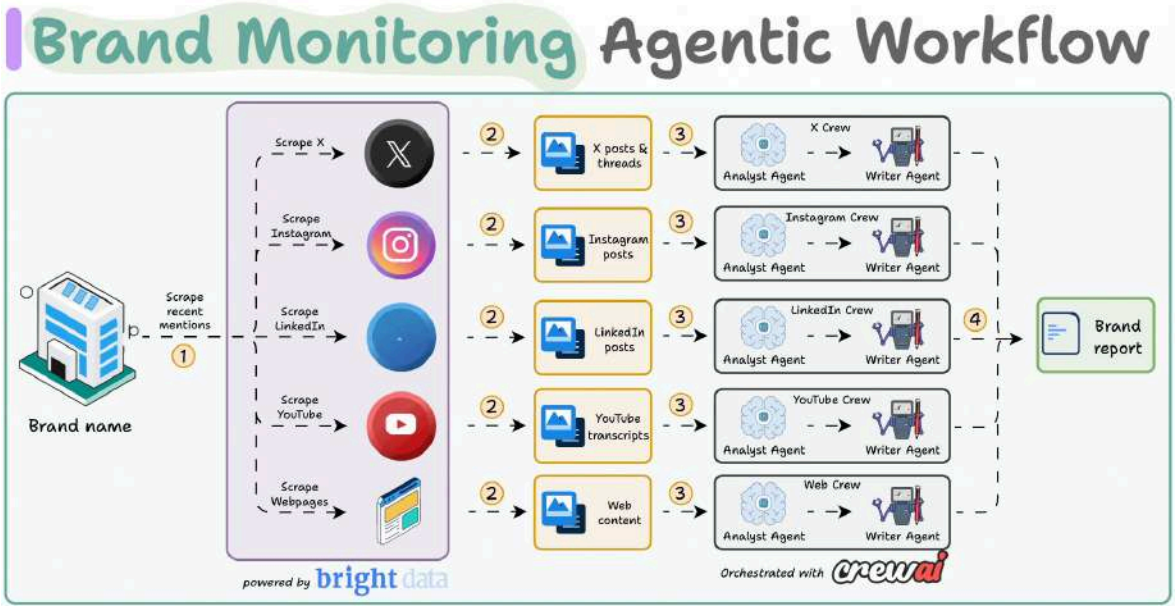
Go to: File → Preferences → Cursor Settings → MCP → Add new global MCP server. In the JSON file, add what's shown below

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/

#5) Brand Monitoring System

Build a multi agent brand monitoring app that scraps web mentions and produces insights about a company.



Tech stack:

- Bright Data to scrape data at scale
- CrewAI for orchestration
- Ollama to serve DeepSeek locally

Workflow:

- Use Bright Data to scrape brand mentions across X, Instagram, YouTube, websites, etc.
- Invoke platform-specific Crews to analyze the data and generate insights.
- Merge all insights to get the final report.

Let's implement this!

#1) Scraping tool

To monitor a brand, we must scrape data across various sources—X, YouTube, Instagram, websites, etc.

Thus, we'll first gather recent search results from Bright Data's SERP API.

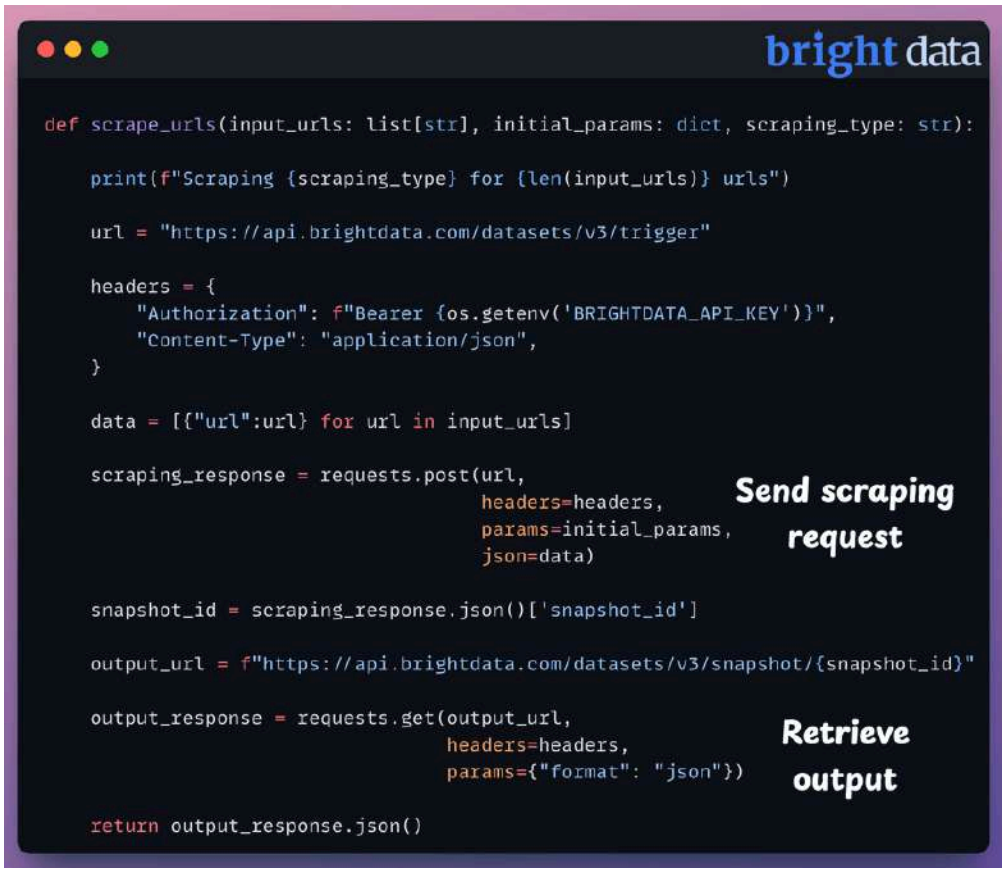
See this code:

#2) Platform-specific scraping function

The above output will contain links to web pages, X posts, YouTube videos, Instagram posts, etc.

To scrape those sources, we use Bright Data's platform-specific scrapers.

Check this code:



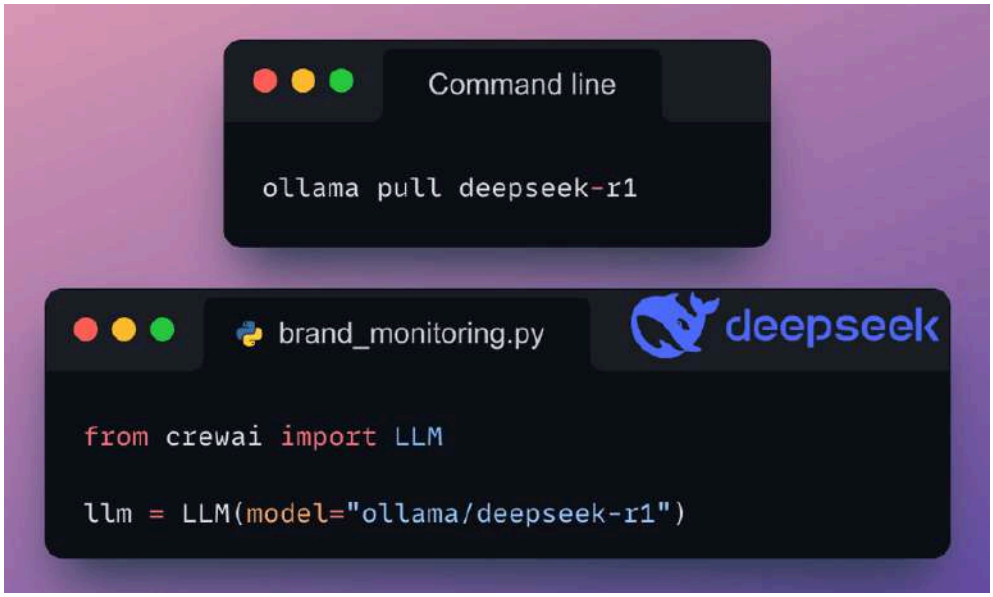
#3) Set up DeepSeek R1 locally

We'll serve R1 locally through Ollama.

To do this:

- First, we download it locally.
- Next, we define it with the CrewAI's LLM class.

Here's the code:



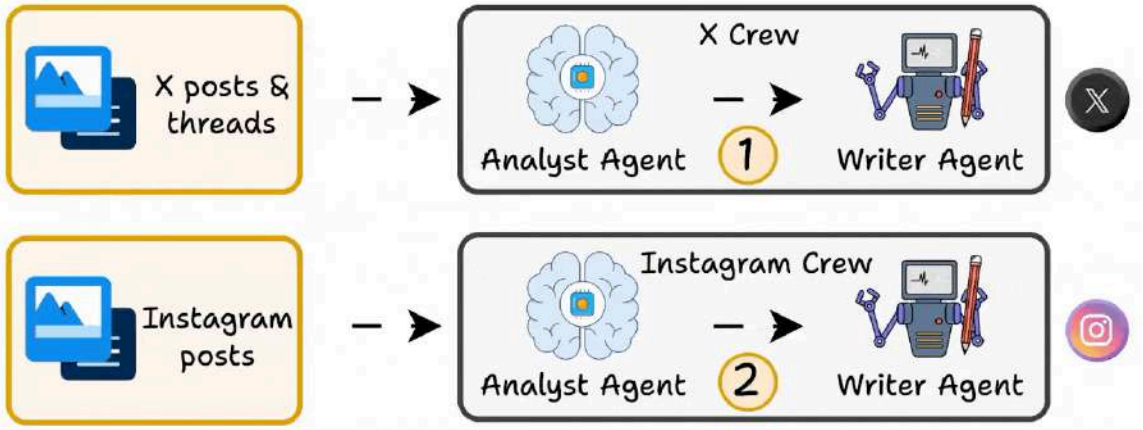
#4) Crew Setup

We will have multiple Crews, one for each platform (X, Instagram, YouTube, etc.)

Each Crew will have two Agents:

- Analysis Agent → It analyses the scraped content.
- Writer Agent → It produces insights from the analysis.

Below, let's implement the X Crew!



#5) X Analyst Agent

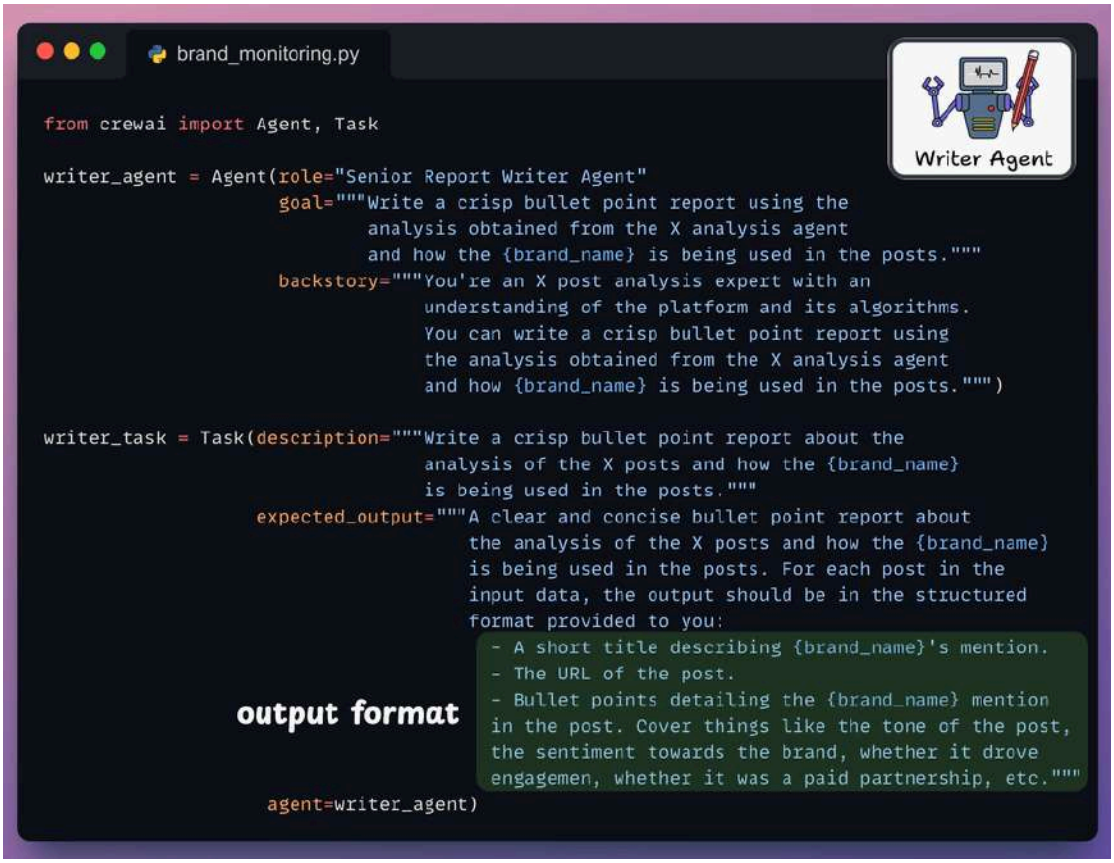
This Agent analyzes the posts scraped by Bright Data and extracts key insights. It is also assigned a task to do so.

Here's how it's done:

#6) X Writer Agent

The Agent takes the output of the X analyst agent and generates insights.

Here's the code:

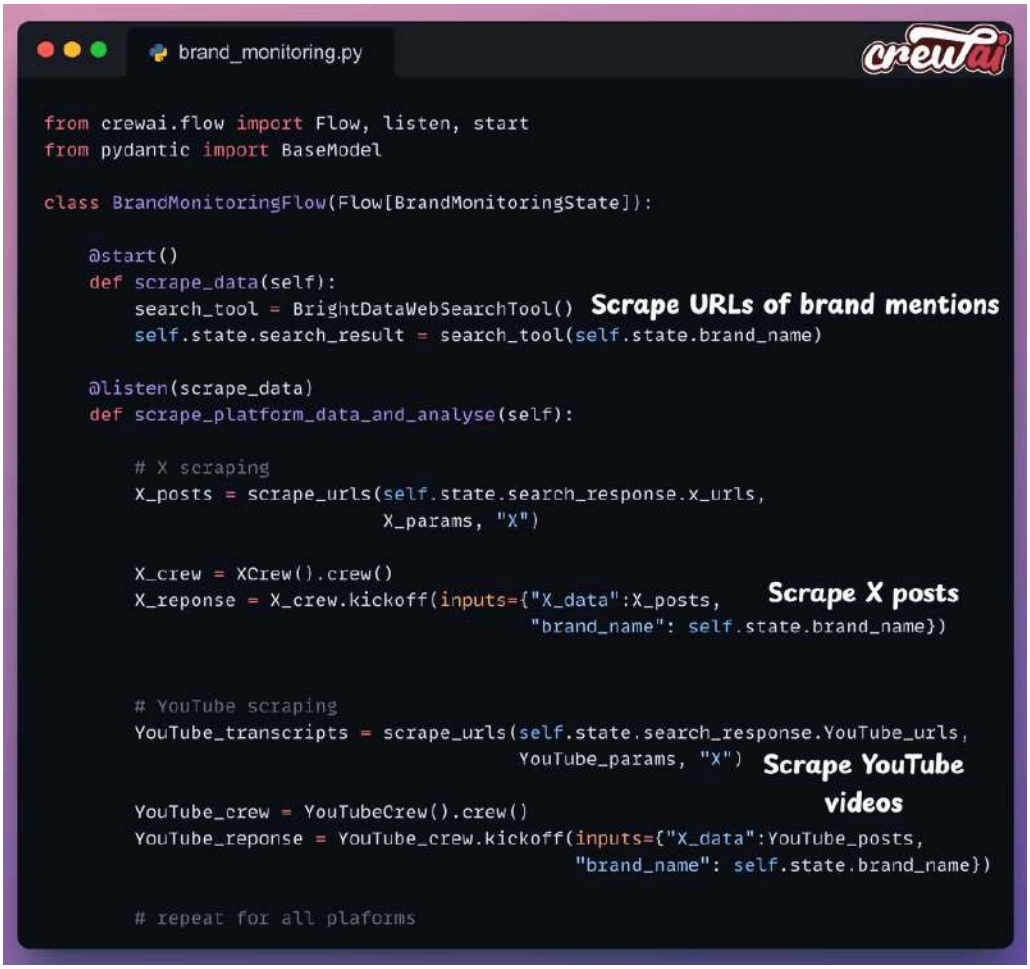


#7) Create a Flow

Finally, we use CrewAI Flows to orchestrate the workflow:

- We start the Flow by using the Scraping tool.
- Next, we invoke platform-specific scrapers.
- Finally, we invoke platform-specific Crews.

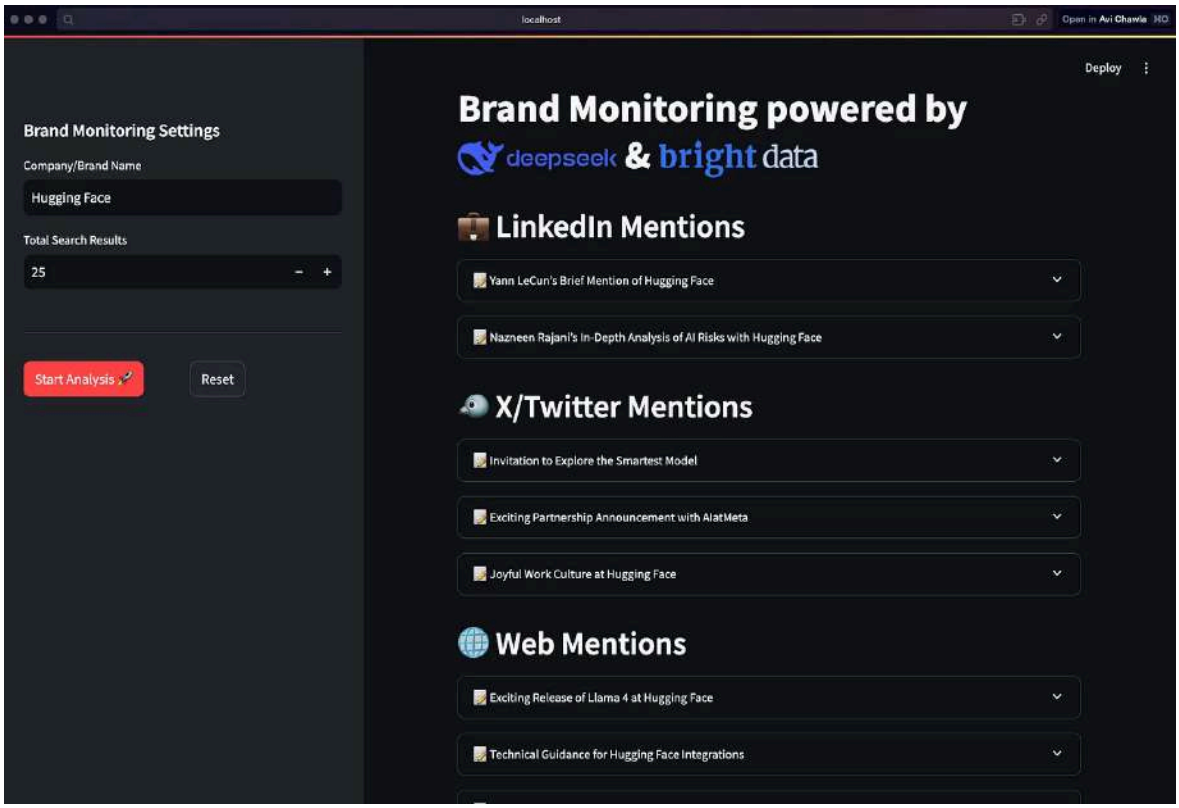
Check this out:



#8) Streamlit UI and Kick off the Flow

Finally, we wrap the app in a clear Streamlit interface for interactivity and run the Flow.

Check the final outcome:



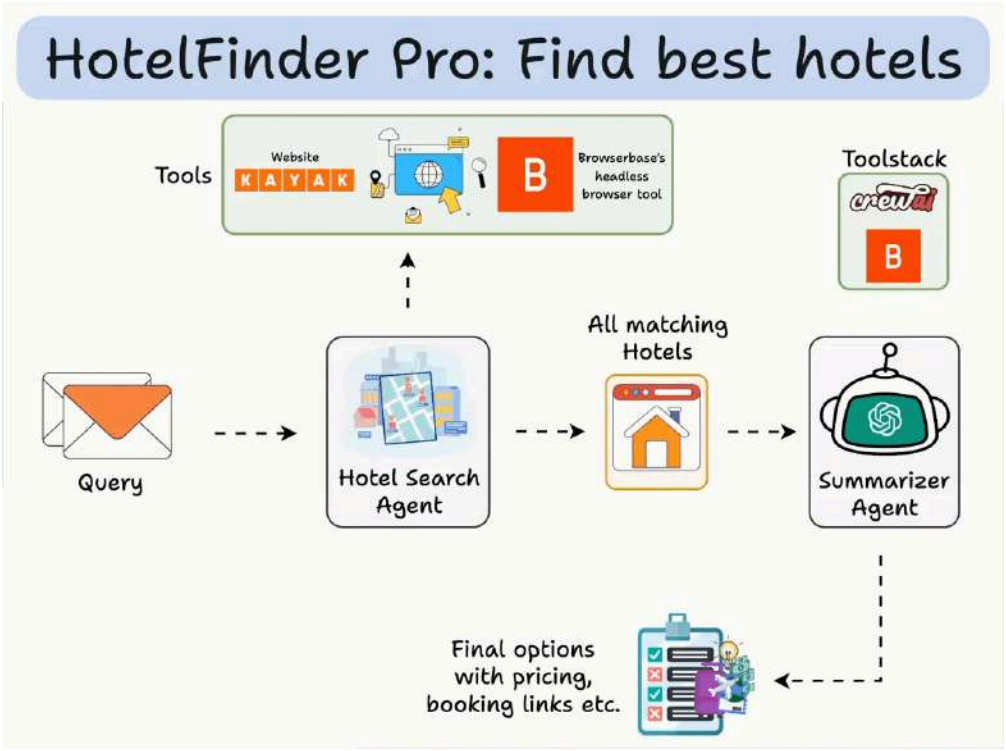
Done!

You're now ready to monitor any brand, track all mentions, and generate insights about a company.

The code is available here: https://www.dailydoseofds.com/p/hands-on-build-a-multi-agent-brand-monitoring-system/

#6) Multi-agent Hotel Finder

Build an Agentic workflow that parses a travel query, fetches live flights and hotel data from Kayak, and summarizes the best options.



Tech stack:

- CrewAI for multi-agent orchestration
- Browserbase's headless browser tool
- Ollama to locally serve DeepSeek-R1

Workflow:

- Parse the query (location, dates etc.) to create a Kayak search URL
- Visit the URL and extract top 5 flights
- For each hotel, find pricing and more info
- Summarize hotel info

Let's implement this!

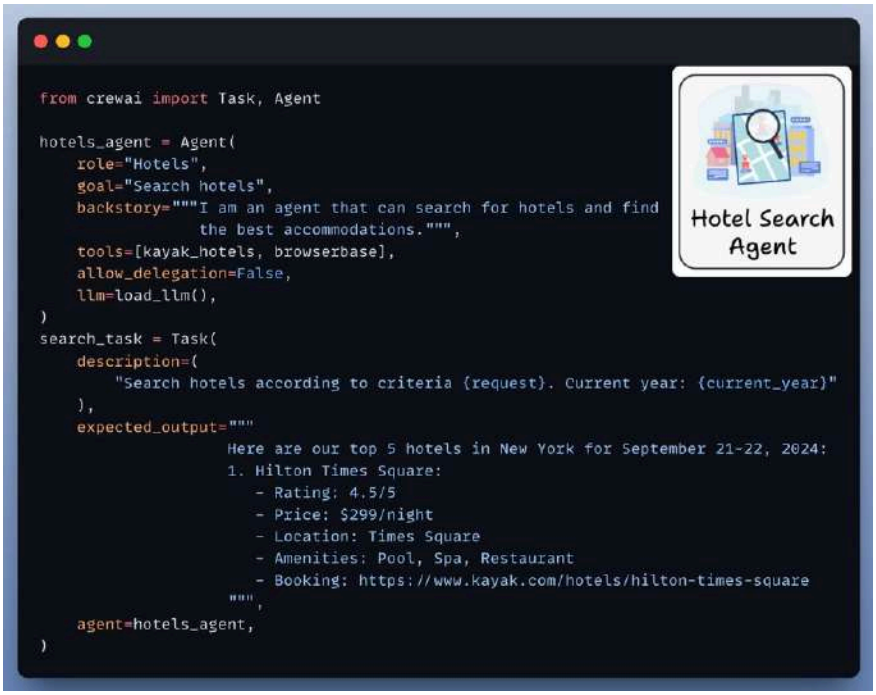
#1) Define LLM

CrewAI nicely integrates with all the popular LLMs and providers out there!

Here's how you set up a local DeepSeek using Ollama:

#2) Hotel Search Agent

This agent mimics a real human searching for hotels by browsing the web. It's powered by browserbase's headless-browser tool and can look up hotels on sites like Kayak.

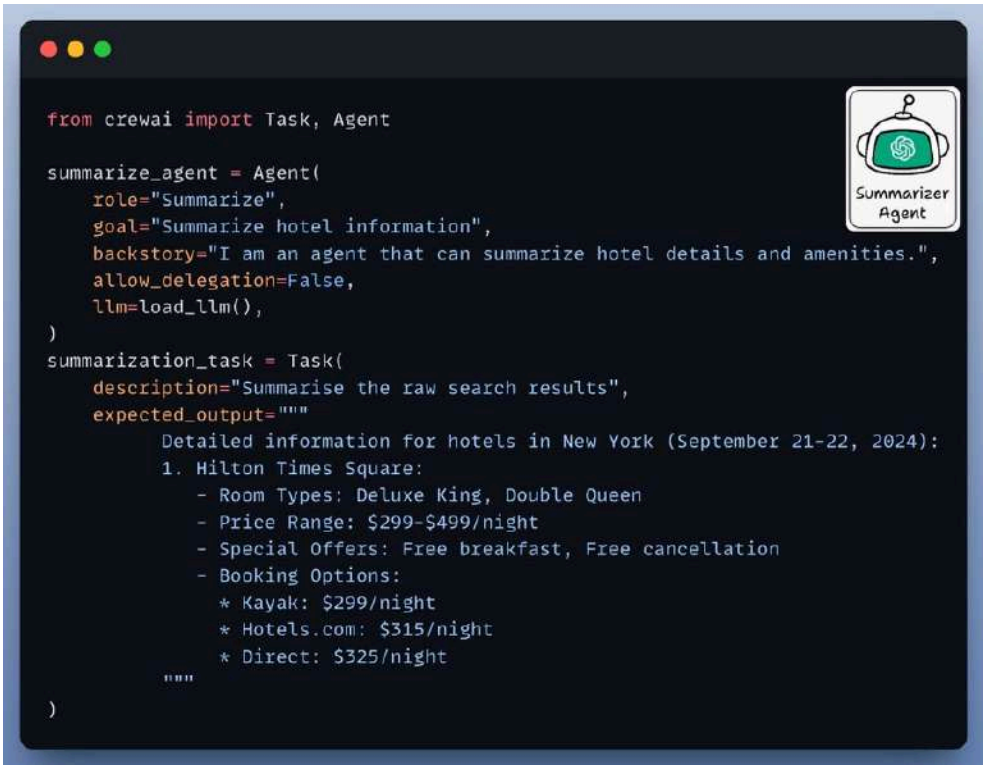


#3) Summarisation Agent

After retrieving the hotel details, we need a concise summary of all available options.

This is where our Summarization Agent steps in to make sense of the results for easy reading.

Check this out:



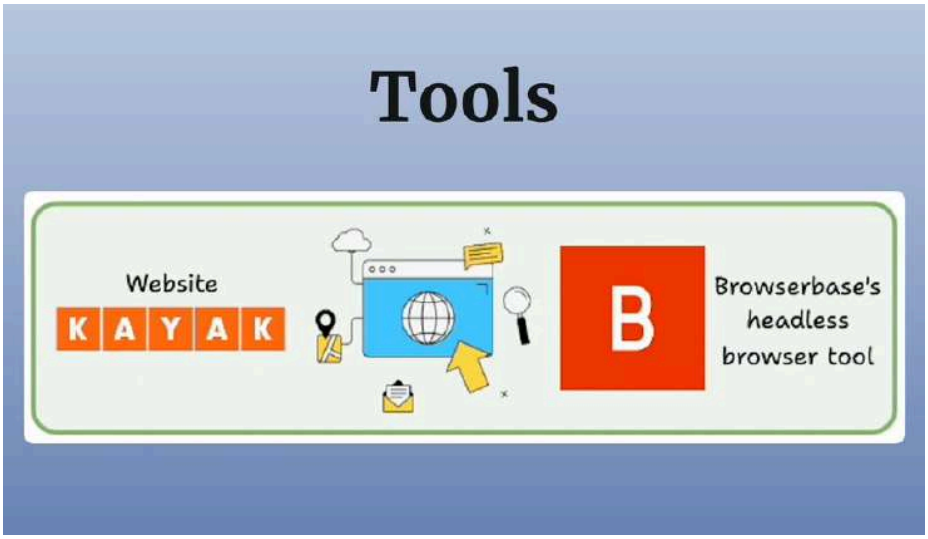

```python
from crewai import Task, Agent

summarize_agent = Agent(
    role="Summarize",
    goal="Summarize hotel information",
    backstory="I am an agent that can summarize hotel details and amenities.",
    allow_delegation=False,
    llm=load_llm(),
)
summarization_task = Task(
    description="Summarise the raw search results",
    expected_output="""
          Detailed information for hotels in New York (September 21-22, 2024):
          1. Hilton Times Square:
             - Room Types: Deluxe King, Double Queen
             - Price Range: $299-$499/night
             - Special Offers: Free breakfast, Free cancellation
             - Booking Options:
               * Kayak: $299/night
               * Hotels.com: $315/night
               * Direct: $325/night
          """
)
```

Now that we have both our agents ready, it's time to understand the tools powering them.

1. Kayak tool

2. Browserbase tool

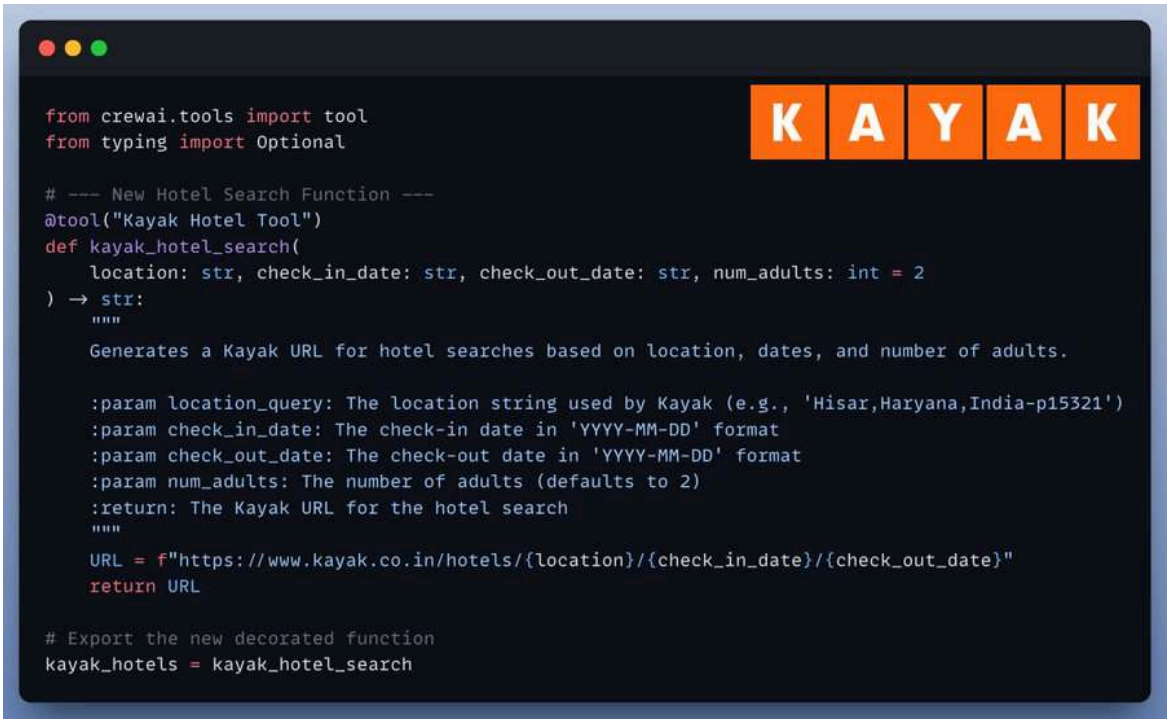
Let's write their code one-by-one.



#4) Kayak tool

A custom Kayak tool to translate the user input into a valid Kayak search URL.

(FYI Kayak is a popular for hotel and fight booking)



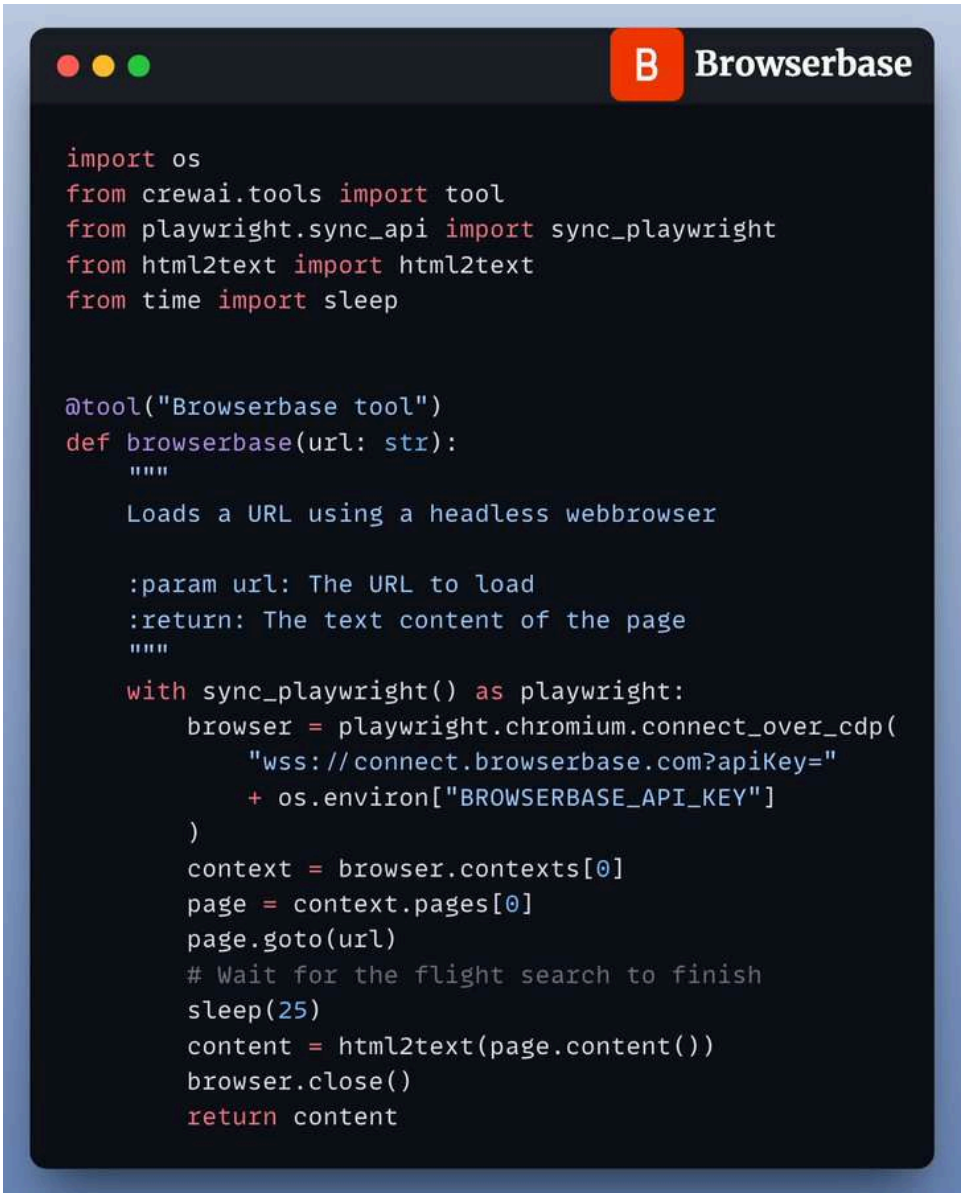
```python
from crewai.tools import tool
from typing import Optional

# --- New Hotel Search Function ---
@tool("Kayak Hotel Tool")
def kayak_hotel_search(
    location: str, check_in_date: str, check_out_date: str, num_adults: int = 2
) -> str:
    """
    Generates a Kayak URL for hotel searches based on location, dates, and number of adults.

    :param location_query: The location string used by Kayak (e.g., 'Hisar,Haryana,India-p15321')
    :param check_in_date: The check-in date in 'YYYY-MM-DD' format
    :param check_out_date: The check-out date in 'YYYY-MM-DD' format
    :param num_adults: The number of adults (defaults to 2)
    :return: The Kayak URL for the hotel search
    """
    URL = f"https://www.kayak.co.in/hotels/{location}/{check_in_date}/{check_out_date}"
    return URL

# Export the new decorated function
kayak_hotels = kayak_hotel_search
```

#5) Browserbase Tool

The hotel search agent uses the Browserbase tool to simulate human browsing and gather hotel data.

To be precise it automatically navigates the Kayak website and interacts with the web page.

Check this out:



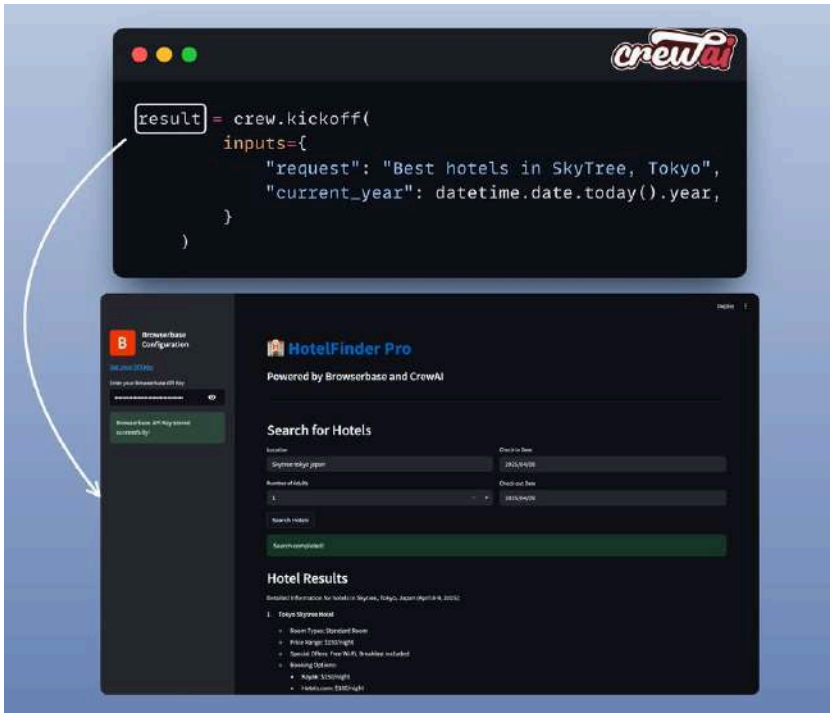
#6) Setup Crew

Once the agents and tools are defined, we orchestrate them using CrewAI.

Define their tasks, sequence their actions, and watch them collaborate in real time! Check this out:

#7) Kickoff and results

Finally, we feed the user's request (location, dates etc.) into the Crew and let it run! Check this out:

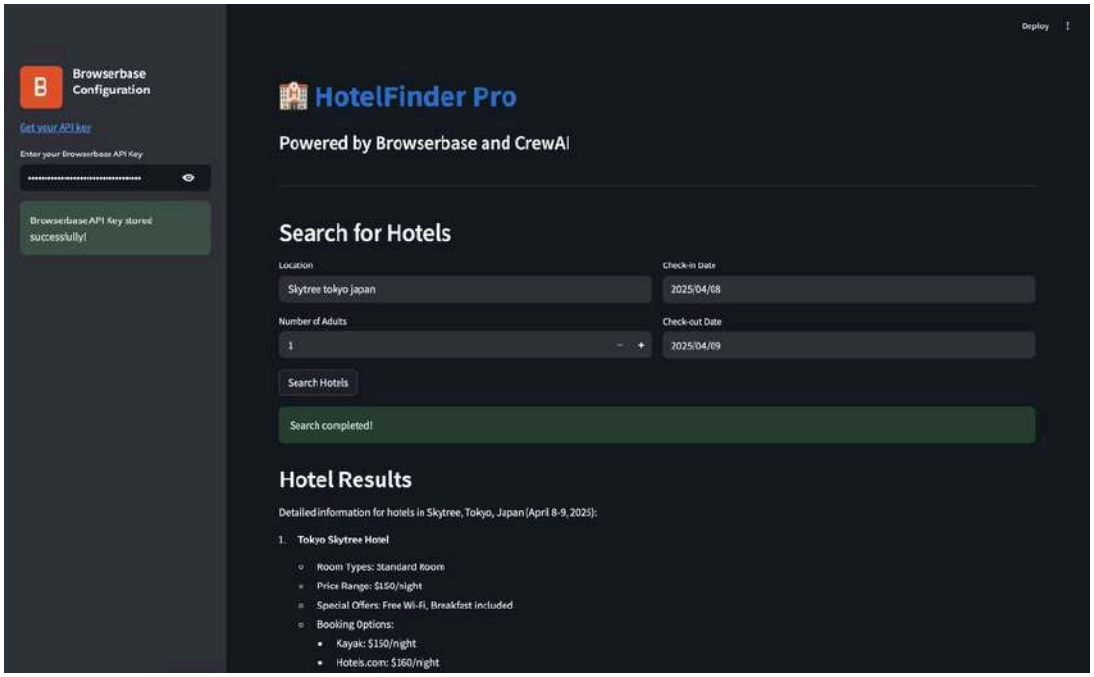


Streamlit UI

To make this accessible, we wrapped the entire system in a @Streamlit interface.

It's a simple chat-like UI where you enter your location and other details and see the results in real time!

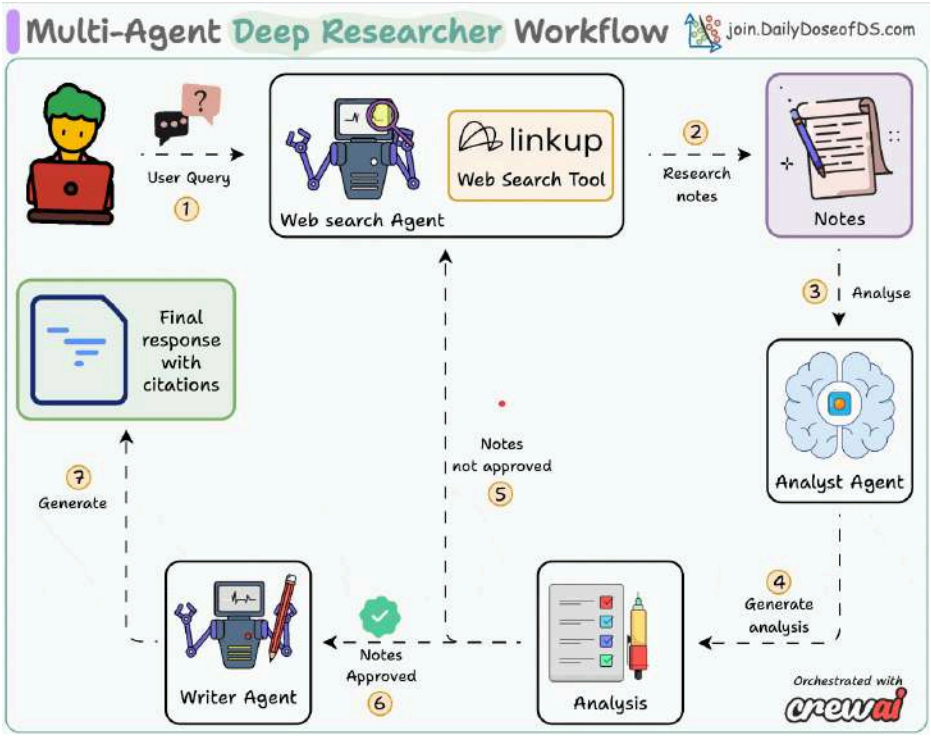
Check this out:



The code is available here: https://github.com/patchy631/ai-engineering-hub/tree/main/hotel-booking-crew

#7) Multi-agent 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

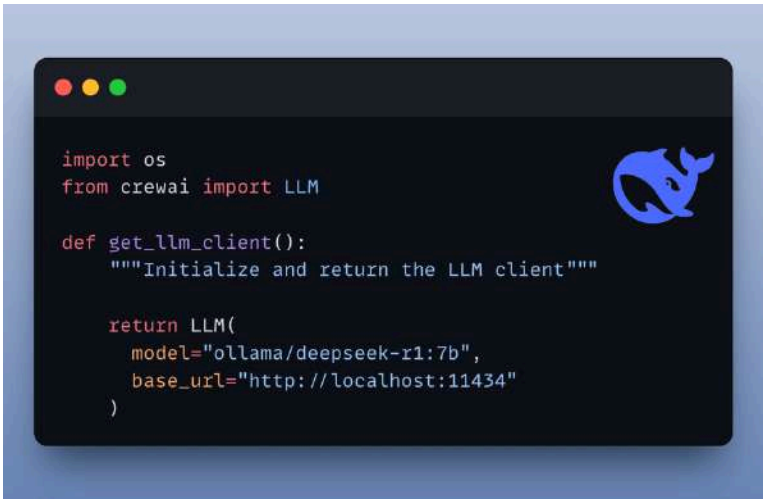
Workflow:

- 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

Let's implement this!

#1) Setup LLM

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



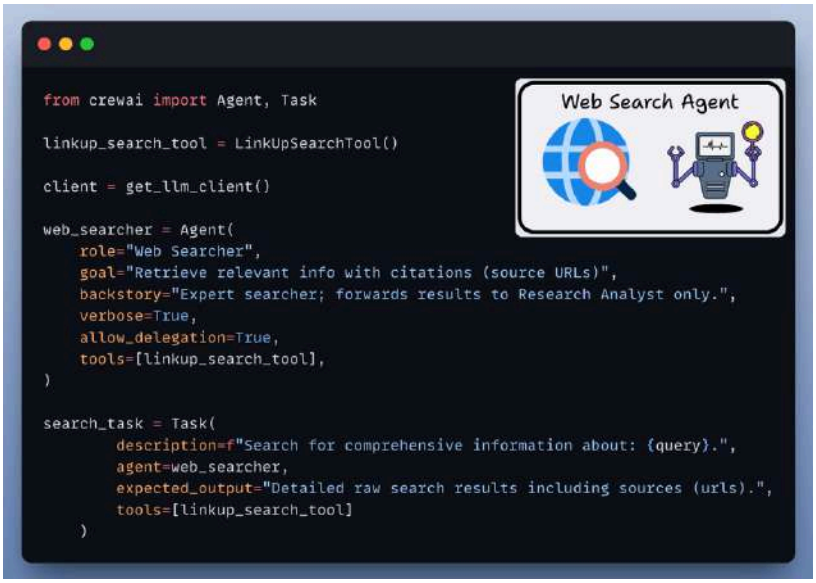
#2) Define Web Search Tool

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.


```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]
    )
```

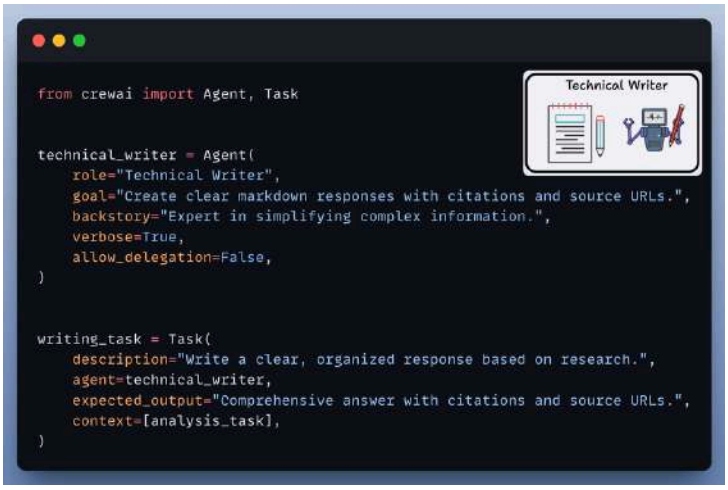
#4) Define Research Analyst Agent

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],
)
```

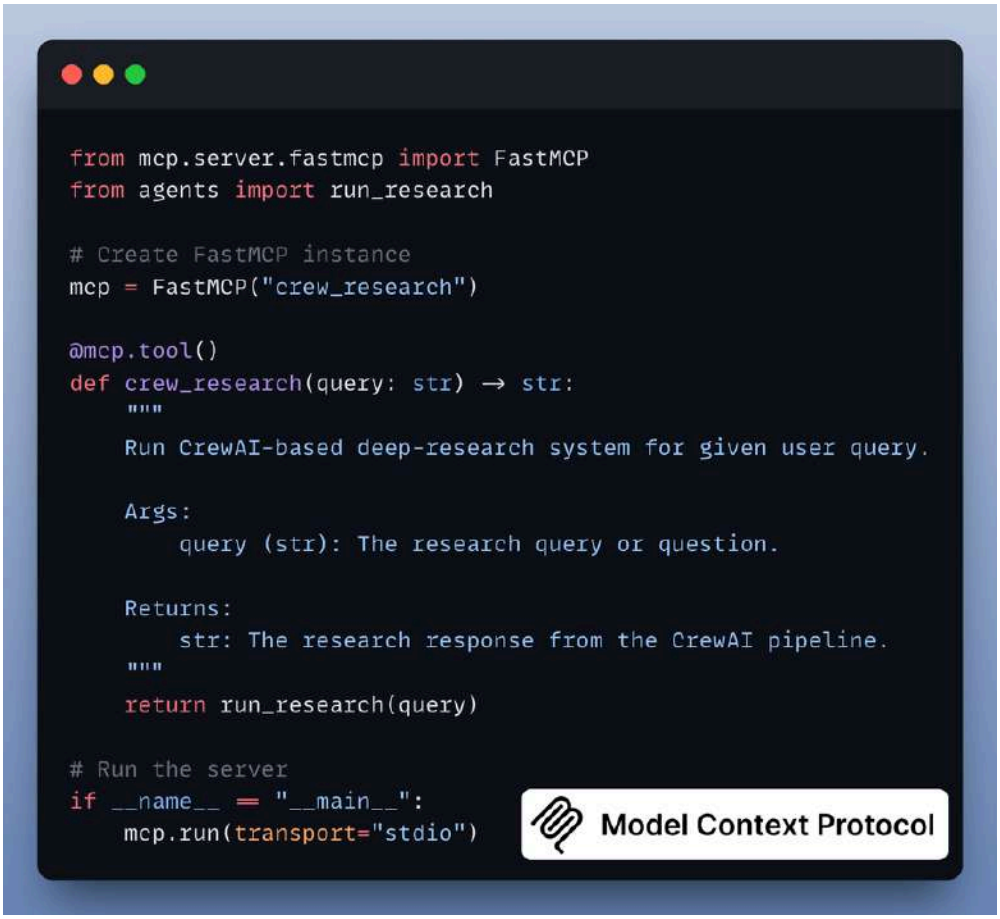
#6) Setup Crew

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

#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.

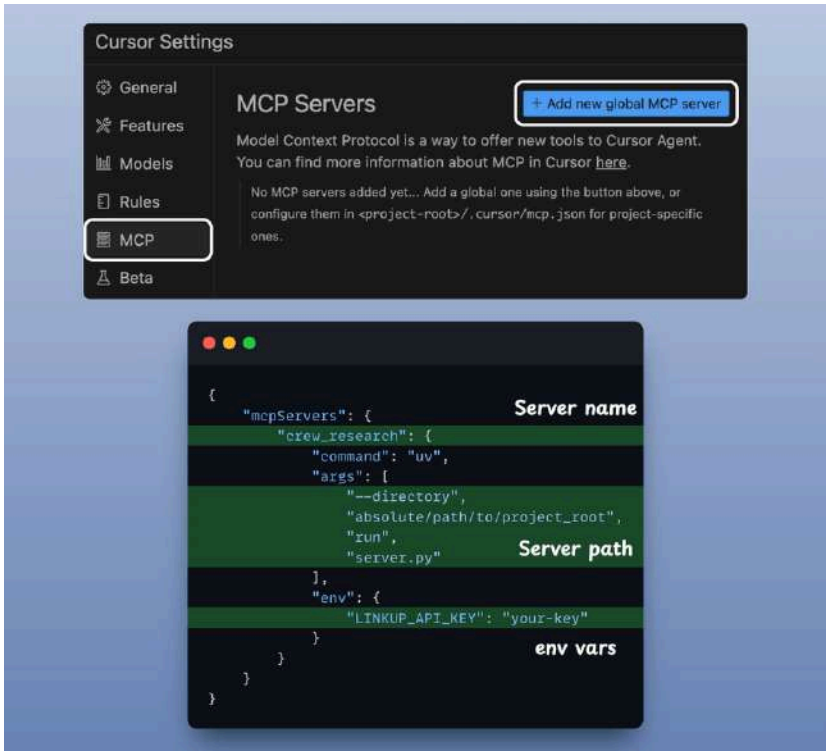
Let's see how to connect it with Cursor.



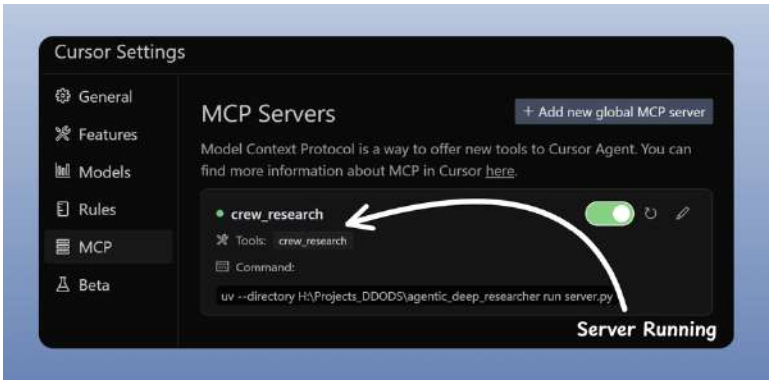
#8) Integrate MCP server with Cursor

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

In the JSON file, add what's shown below



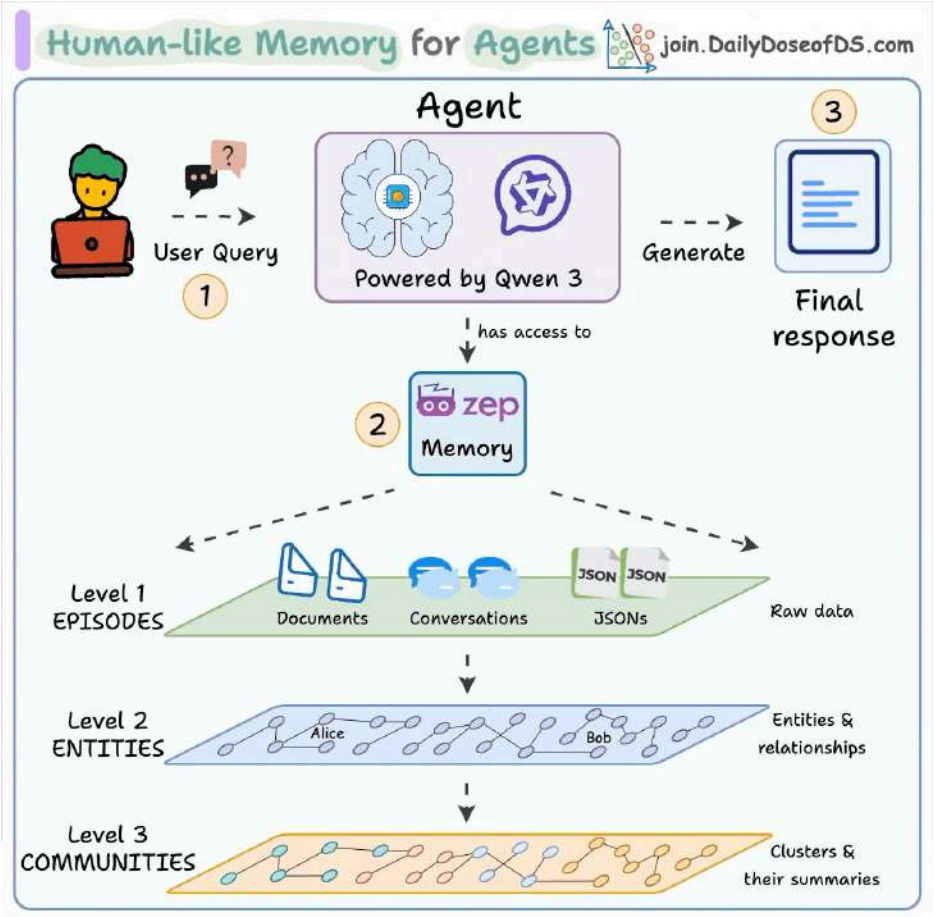
Done! Your deep research MCP server is live and connected to Cursor.



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

#8) Human-like Memory for Agents

If a memory-less AI Agent is deployed in production, every interaction with the Agent will be a blank slate. Learn how to build an AI Agent with human-like memory to solve this.



Tech stack:

- Zep AI for the memory layer to AI agent
- Microsoft AutoGen for agent orchestration
- Ollama to locally serve Qwen3

Workflow:

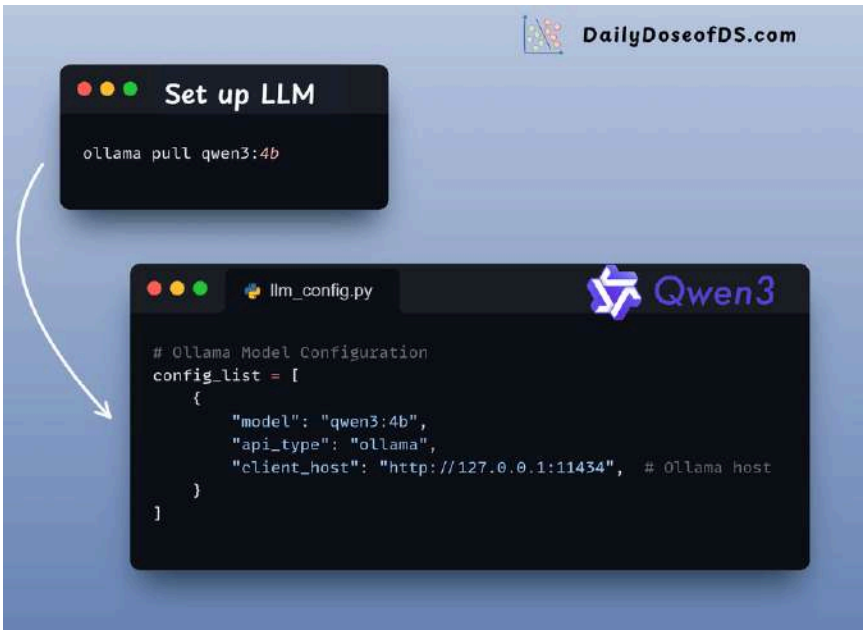
- User submits a query

- Agent saves the conversation and extracts facts into memory
- Agent retrieves facts and summarizes
- Uses facts and history for informed responses

Let's implement this!

#1) Setup LLM

We'll use a locally served Qwen 3 via Ollama. Check this out:

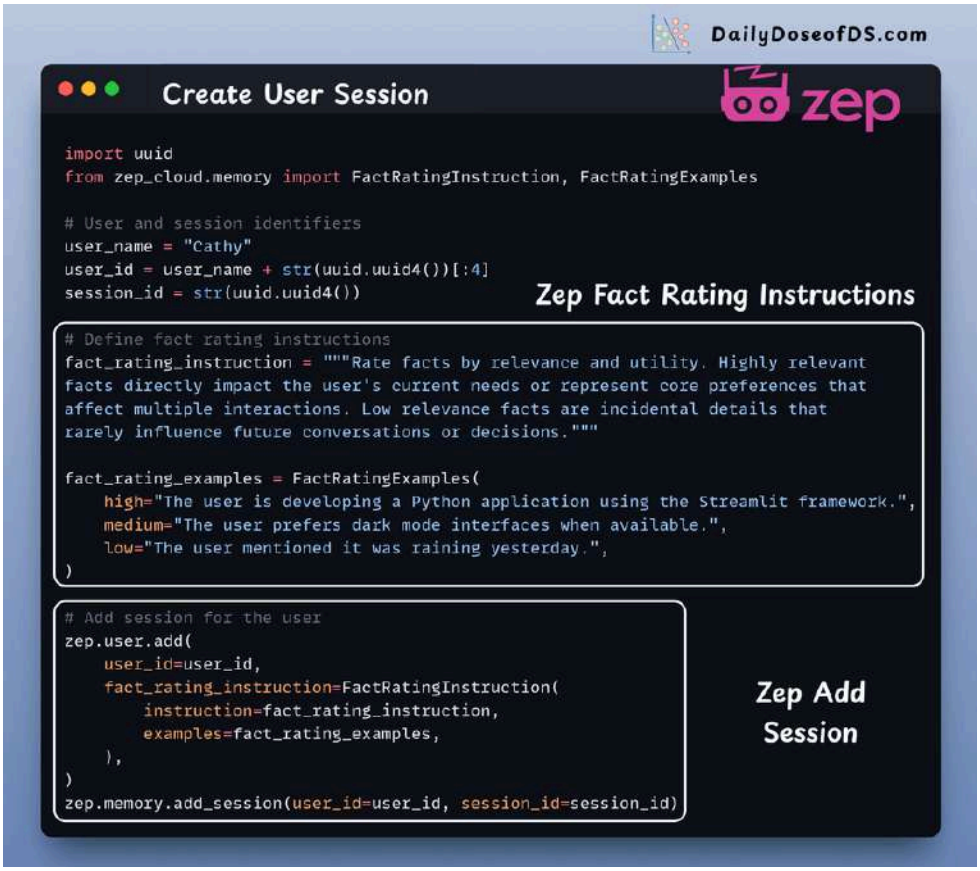


#2) Initialise Zep Client

We're leveraging zep_ai's Foundational Memory Layer to equip our Autogen agent with genuine task-completion capabilities.

#3) Create User Session

Create a Zep client session for the user, which the agent will use to manage memory. A user can have multiple sessions. Here's how it looks:



#4) Define Zep Conversable Agent

Our Zep Memory Agent builds on Autogen's Conversable Agent, drawing live memory context from Zep Cloud with each user query.

It remains efficient by utilizing the session we just established.

Here's how it comes together:

#5) Setting up Agents

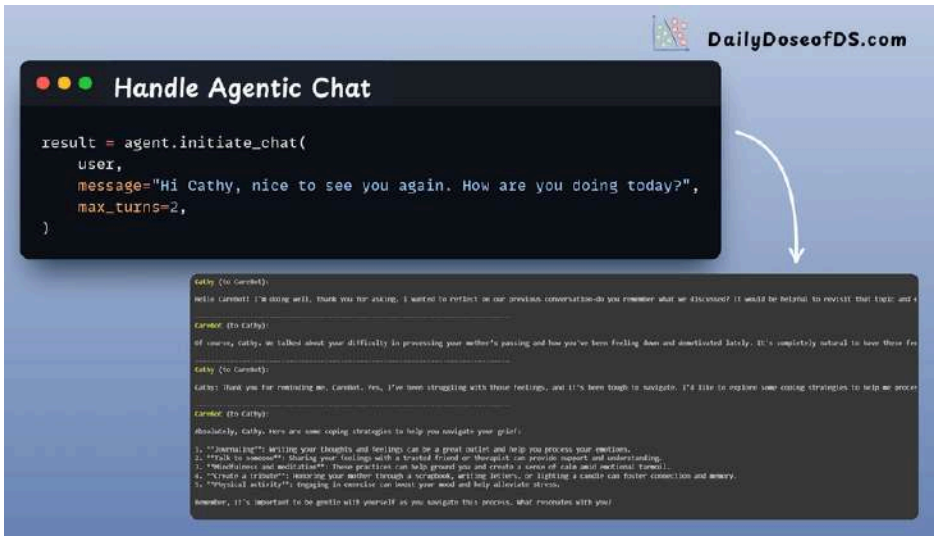
We initialize the Conversable Agent and a Stand-in Human Agent to manage chat interactions.

Here's the setup:

#6) Handle Agentic Chat

The Zep Conversable Agent steps in to create a coherent, personalized response.

It seamlessly integrates memory and conversation. Here's how it works:



#7) Streamlit UI

We created a streamlined Streamlit UI to ensure smooth and simple interactions with the Agent.

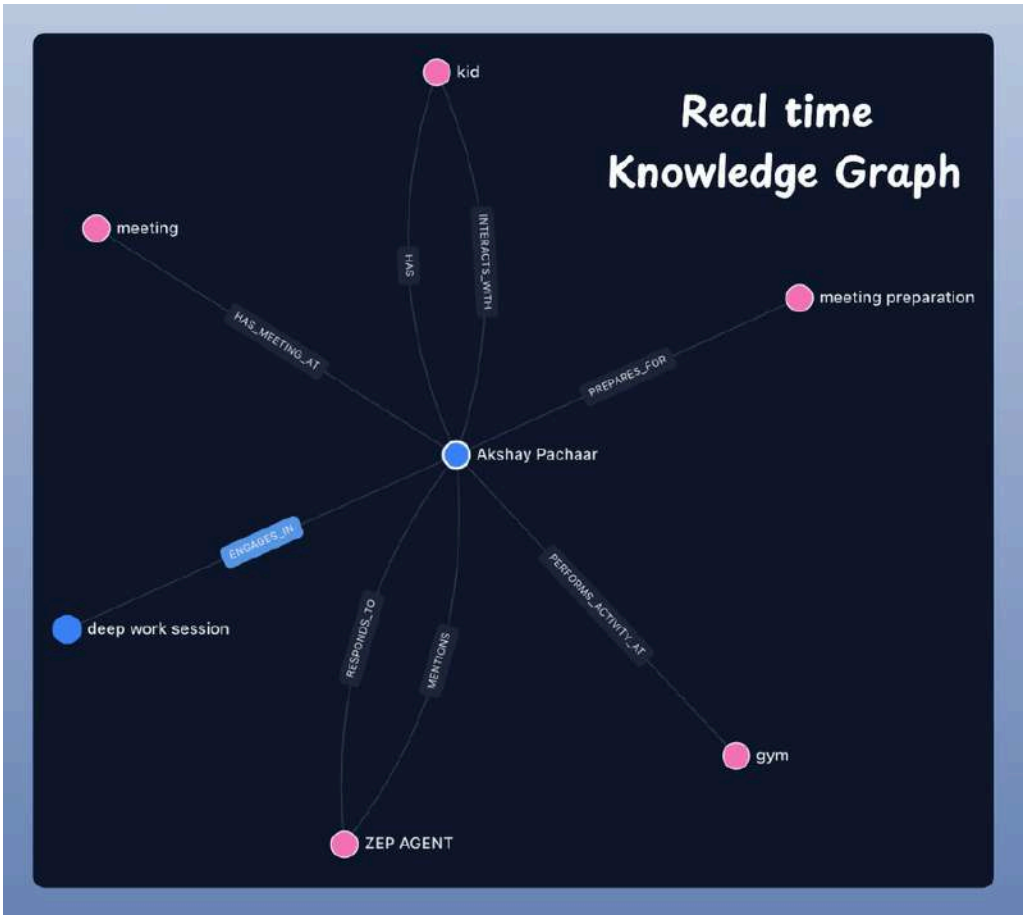
Here's what it looks like:

#8) Visualize Knowledge Graph

We can interactively map users' conversations across multiple sessions with Zep Cloud's UI.

This powerful tool allows us to visualize how knowledge evolves through a graph.

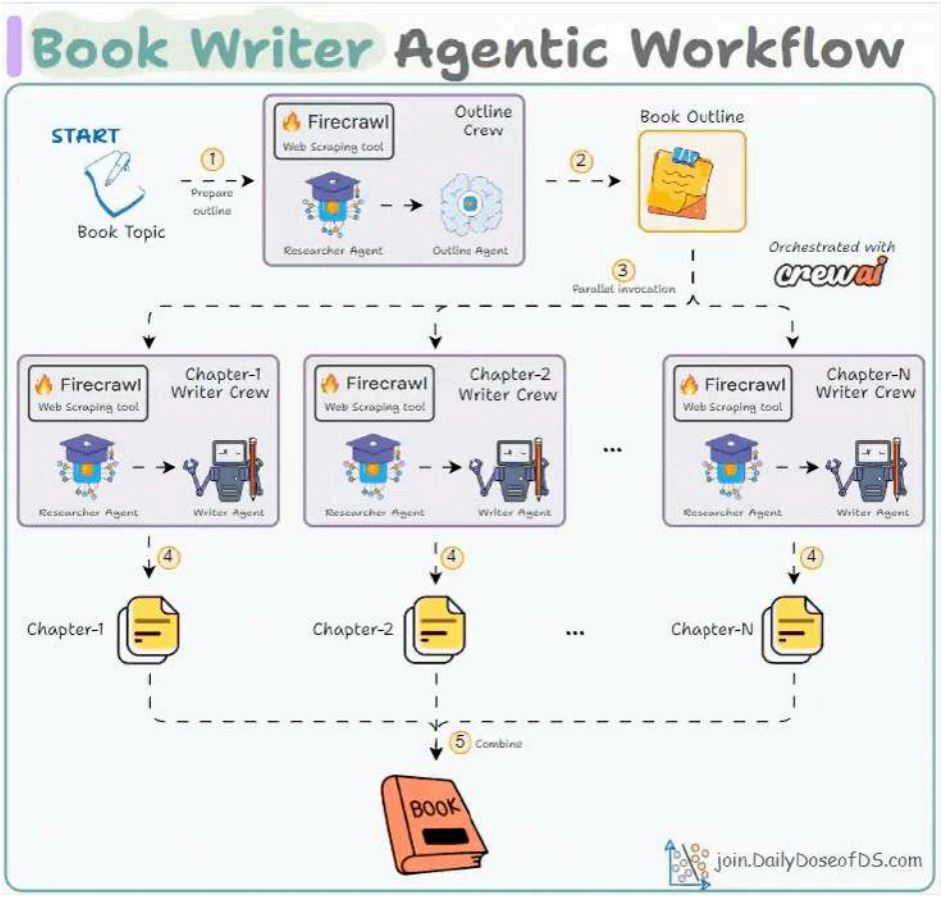
Take a look:



The code is available here:
https://www.dailydoseofds.com/p/hands-on-build-an-ai-agent-with-human-like-memory/

#9) Multi-agent Book Writer

Build an Agentic workflow that writes a 20k-word book from a 3-5 word book title.



Tech stack:

- Firecrawl for web scraping.
- CrewAI for orchestration.
- Ollama to serve Qwen 3 locally.
- LightningAI for development and hosting

Workflow:

- Using Firecrawl, Outline Crew scrapes data related to the book title and decides the chapter count and titles.
- Multiple writer Crews work in parallel to write one chapter each.
- Combine all chapters to get the book.

Let's implement this!

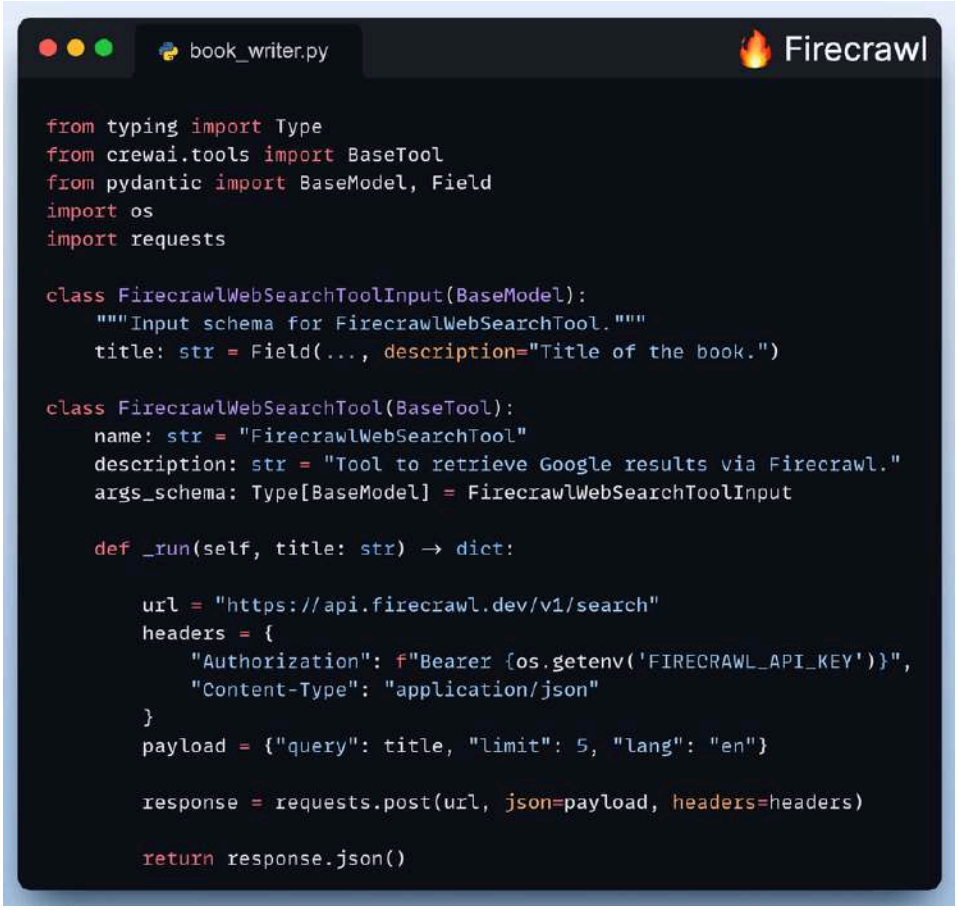
#1) Scraping tool - SERP API

Books demand research. Thus, we'll use Firecrawl's SERP API to scrape data.

Tool usage:

- Outline Crew → to research the book title and prepare an outline.
- Writer Crew → to research the chapter title and write it.

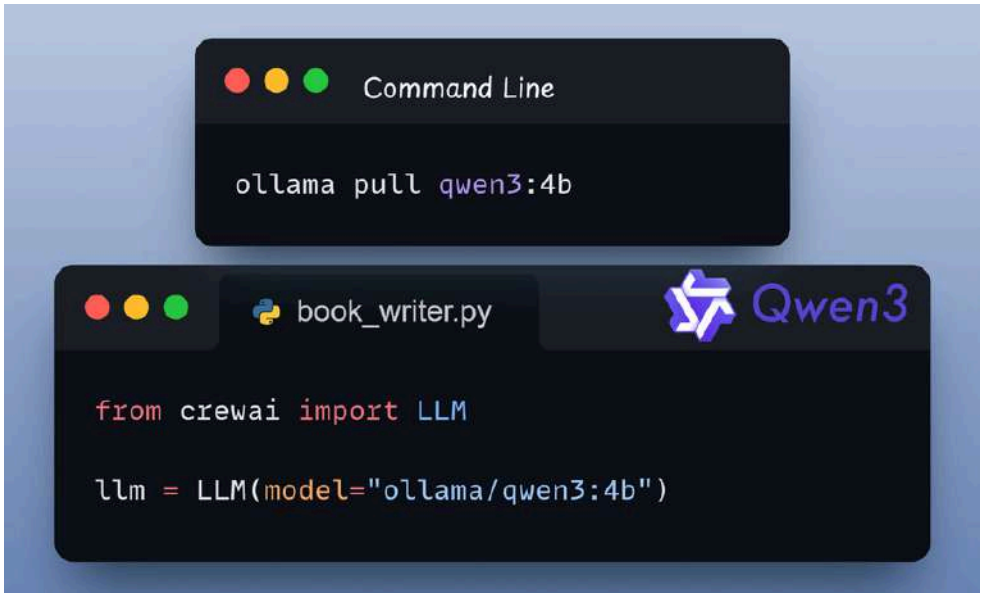
See this code:



#2) Setup Qwen 3 locally

We'll serve Qwen 3 locally through Ollama. To do this:

- First, we download it locally.
- Next, we define it with the CrewAI's LLM class.



#3) Outline Crew

This Crew has two Agents:

- Research Agent → Uses the Firecrawl scraping tool to scrape data related to the book's title and prepare insights.
- Outline Agent → Uses the insights to output total chapters and titles as Pydantic output.

#4) Writer Crew

This Crew has two Agents:

- Research Agent → Uses the Firecrawl Scraping tool to scrape data related to a chapter's title and prepare insights.
- Write Agent → Uses the insights to write a chapter.

Check this code:

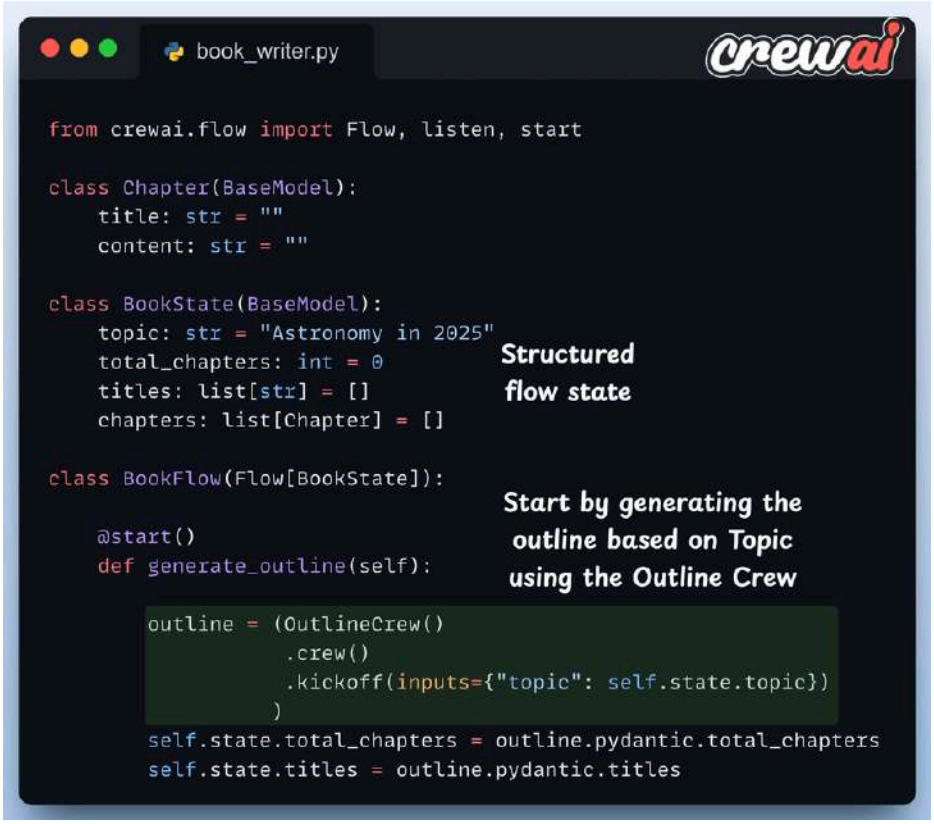
#5) Create a Flow

We use CrewAI Flows to orchestrate the workflow.

First, the outline method invokes the Outline Crew, which:

- research the topic using the scraping tool.
- returns the total number of chapters and the corresponding titles.

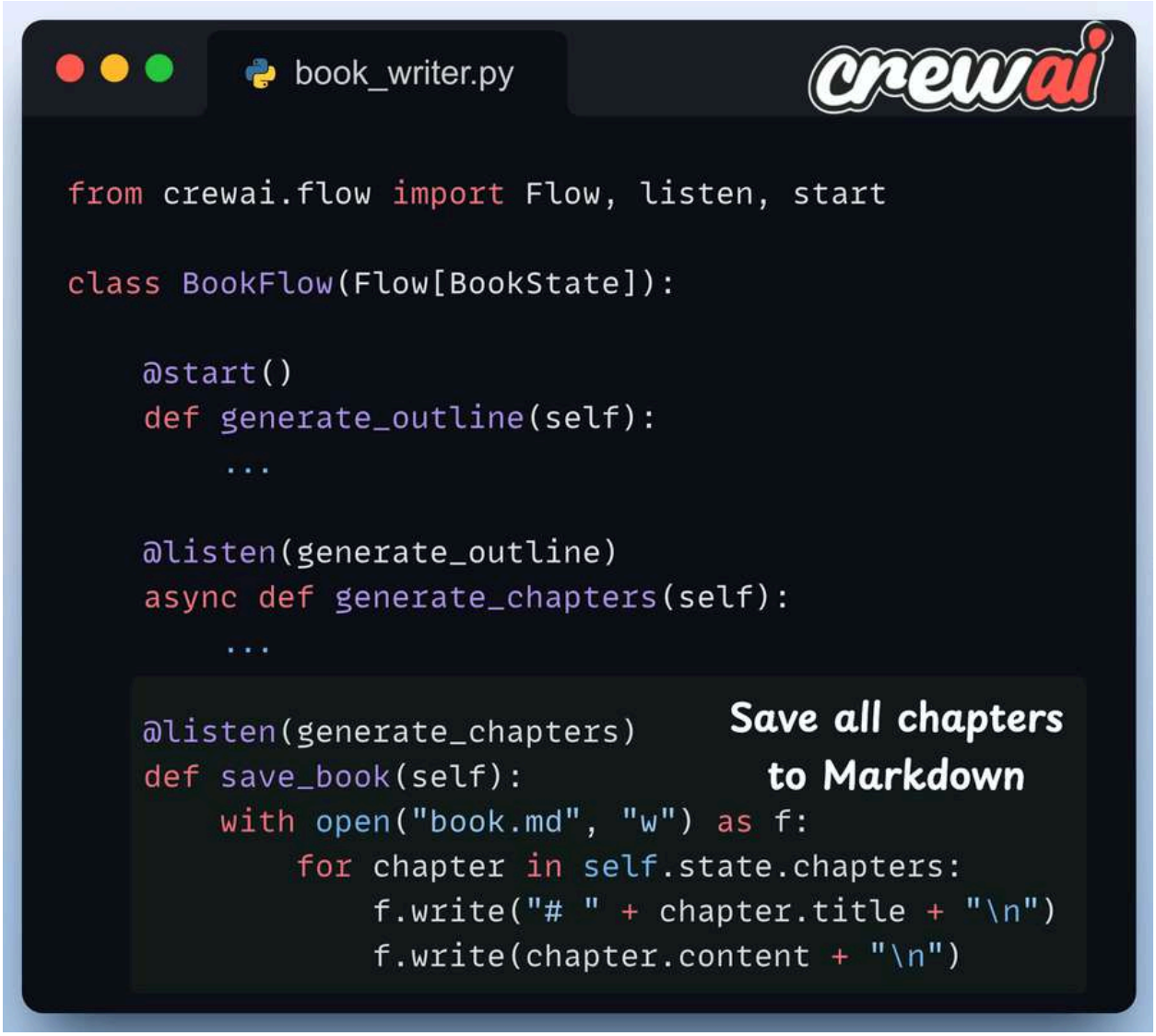
This is implemented below:



#6) Save the book

Once all Writer Crews have finished execution, we save the book as a Markdown file.

Check this code:

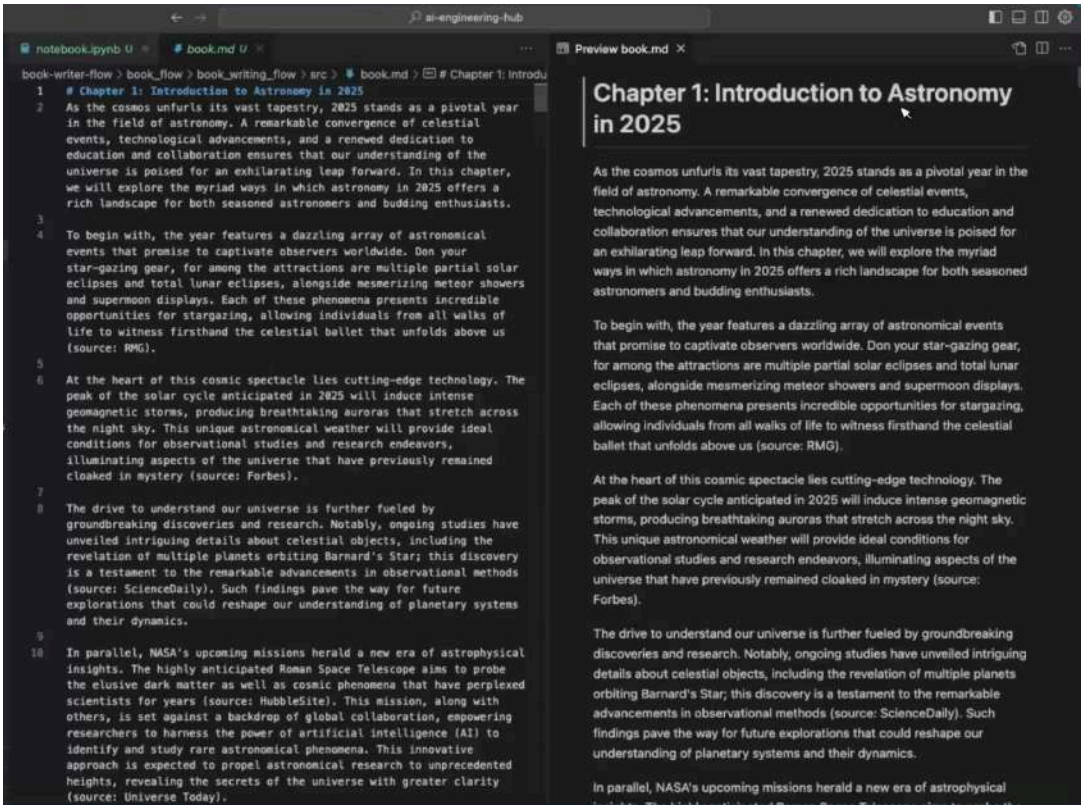


#7) Kickoff the Flow

Finally, we run the Flow.

- First, the Outline Crew is invoked. It utilizes the Firecrawl scraping tool to prepare the outline.
- Next, many Writer Crews are invoked in parallel to write one chapter each.

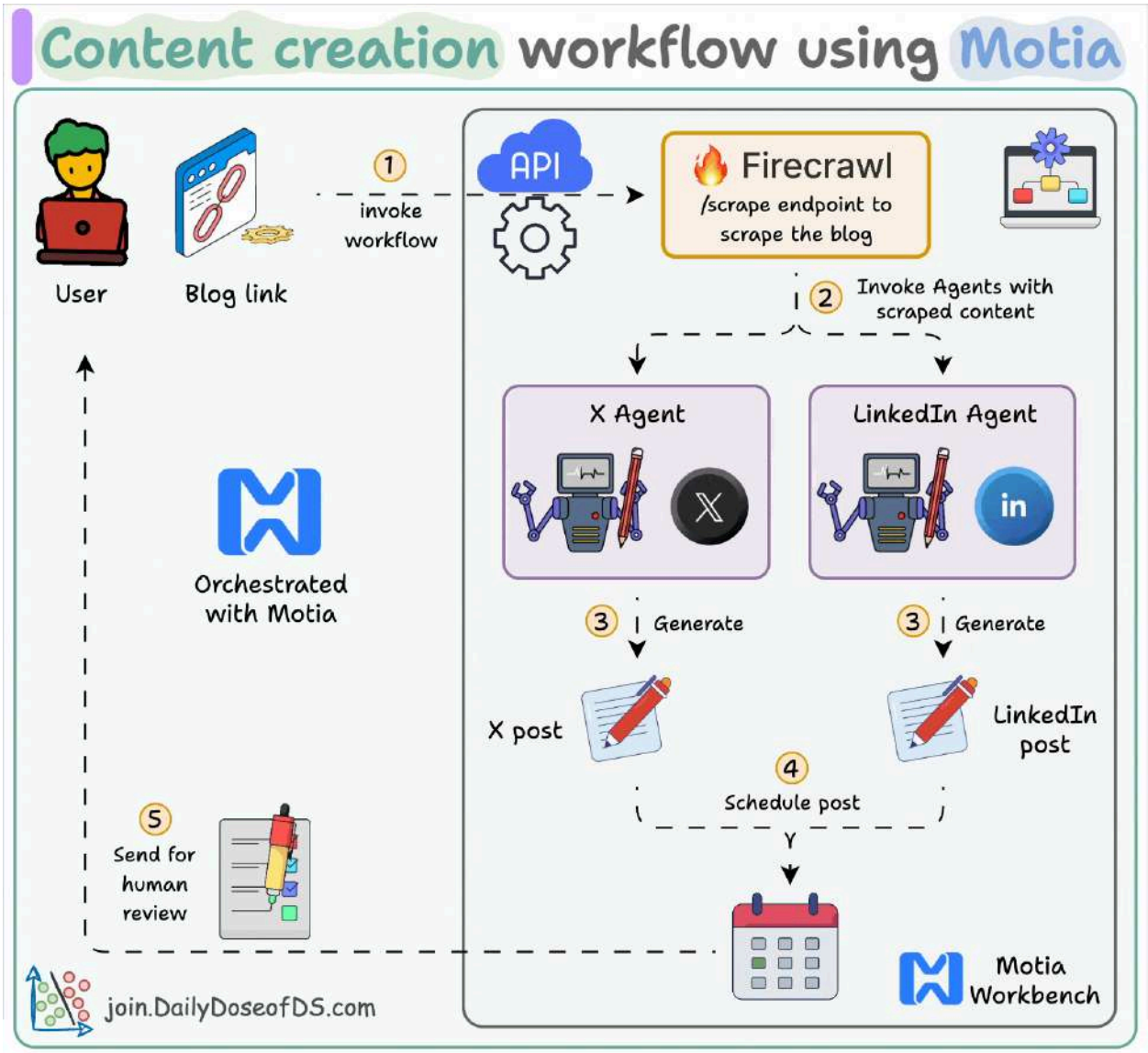
This workflow runs for ~2 minutes, and we get a neatly written book about the specified topic—"Astronomy in 2025." Here's the book our Agentic workflow wrote:



The code is available here: https://blog.dailydoseofds.com/p/building-a-multi-agent-book-writer/

#10) Multi-agent Content Creation System

Build an Agentic workflow that turns any URL into social media posts and auto-schedules them via Typefully.



Tech stack:

- Motia as the unified backend framework
- Firecrawl to scrape web content
- Ollama to locally serve Deepseek-R1 LLM

Workflow:

- User submits URL to scrape
- Firecrawl scrapes content and converts it to markdown
- Twitter and LinkedIn agents run in parallel to generate content
- Generated content gets scheduled via Typefully

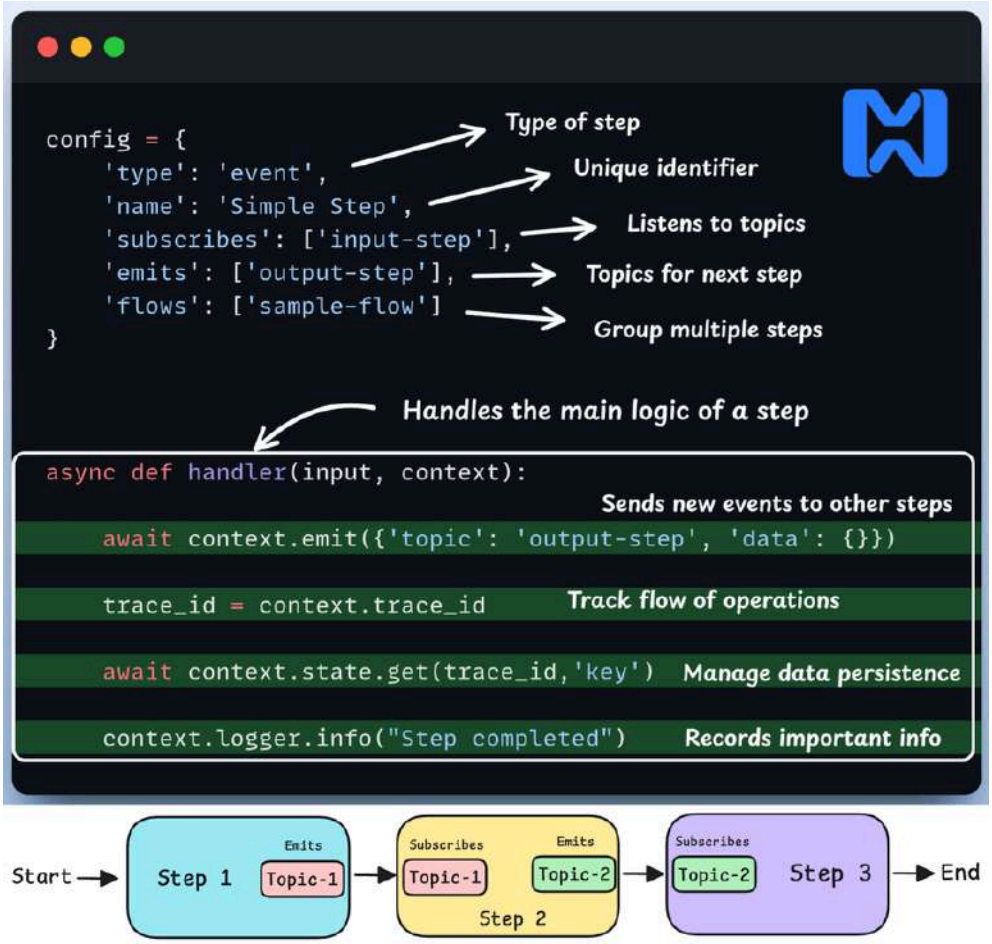
Let's implement this:

Steps are the fundamental building blocks of Motia.

They consist of two main components:

- The Config object: It instructs Motia on how to interact with a step.
- The handler function: It defines the main logic of a step.

Check this out:



With that understanding in mind, let's start building our content creation workflow.

#1) Entry point (API)

We start our content generation workflow by defining an API step that takes in a URL from the user via a POST request.

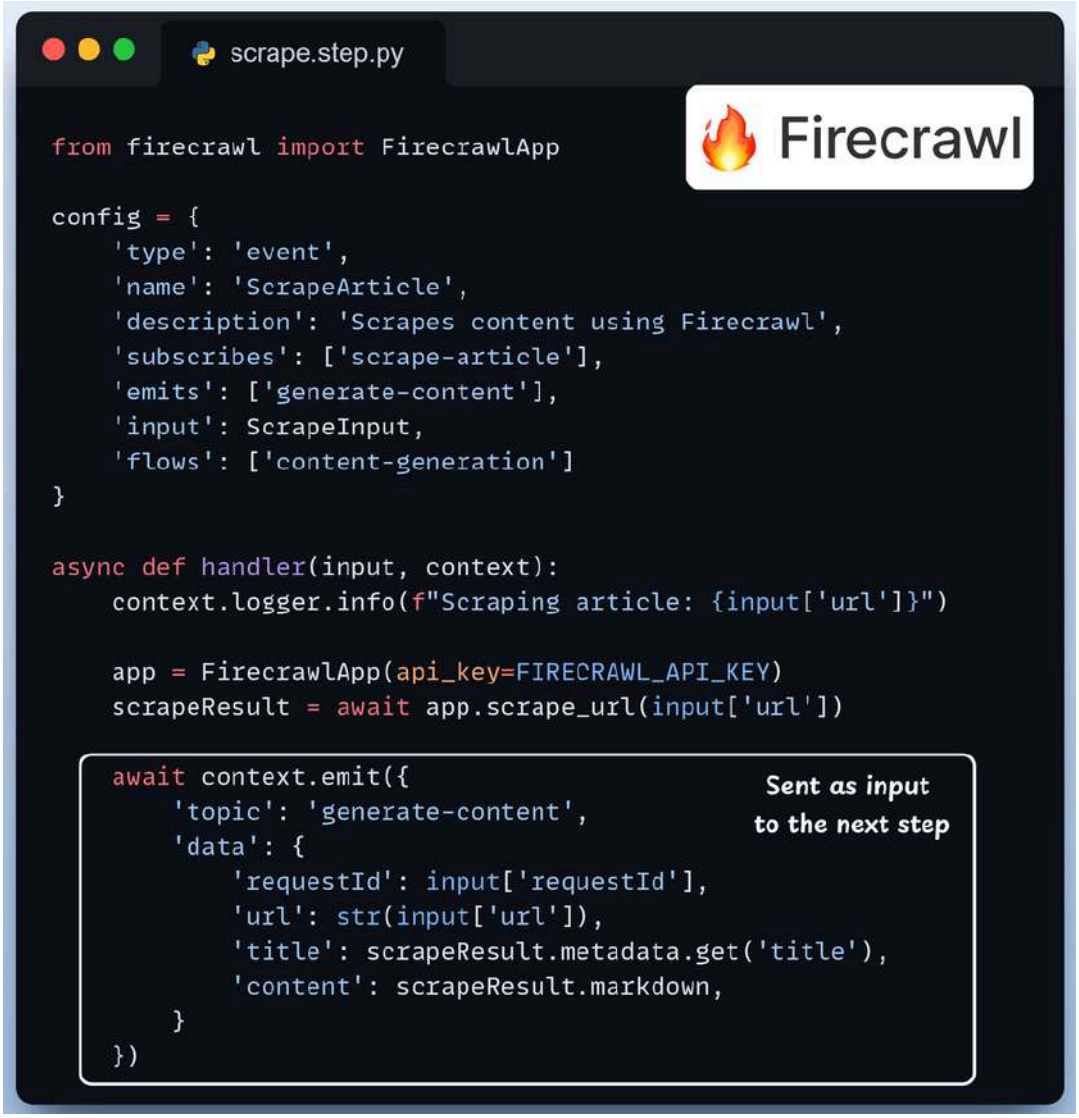
Check this out:

#2) Web scraping

This step scrapes the article content using Firecrawl and emits the next step in the workflow.

Steps can be connected together in a sequence, where the output of one step becomes the input for another.

Check this out:

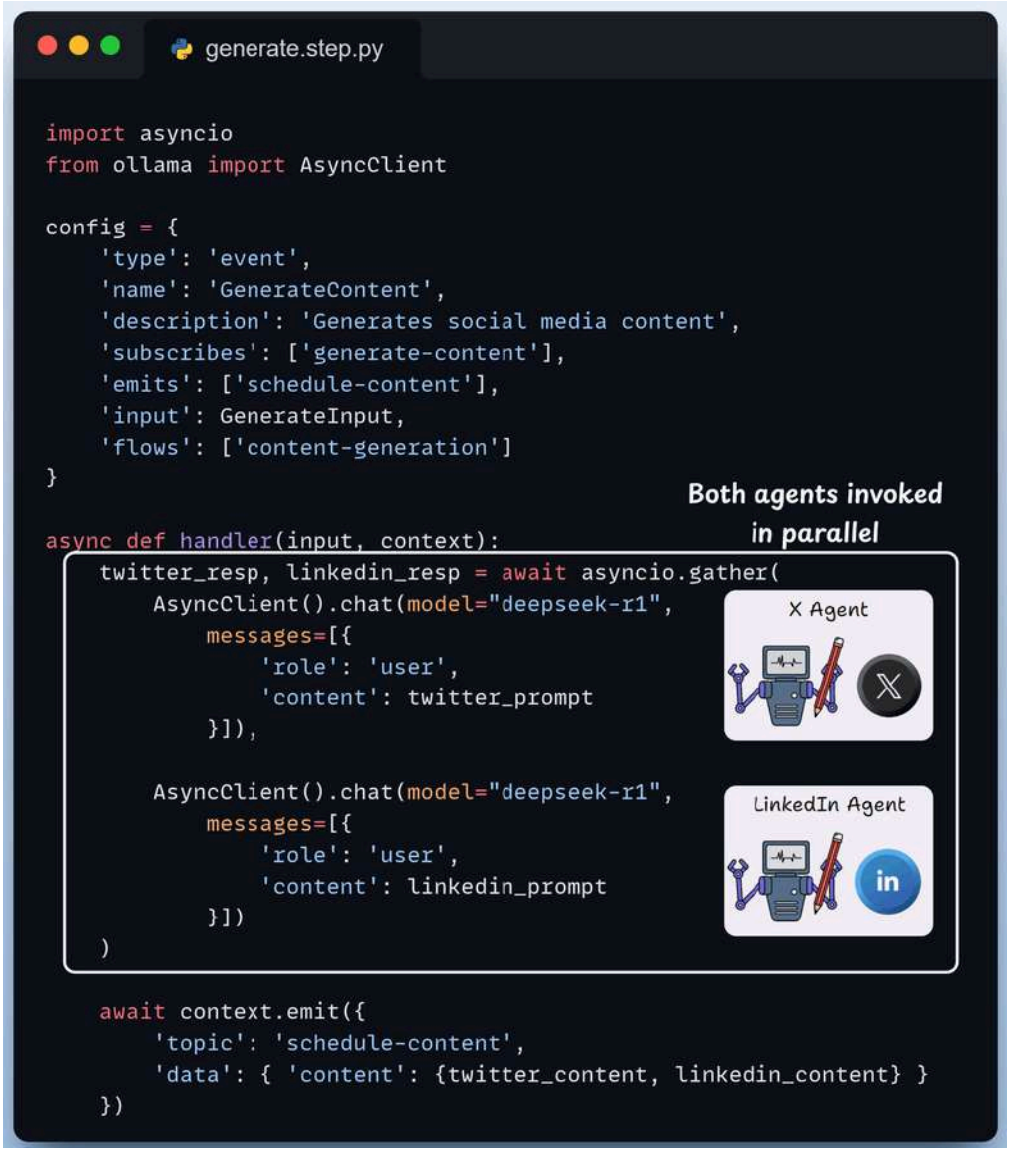


#3) Content generation

The scraped content gets fed to the X and LinkedIn agents that run in parallel and generate curated posts.

We define all our prompting and AI logic in the handler that runs automatically when a step is triggered.

Check this out:



#4) Scheduling

After the content is generated we draft it in Typefully where we can easily review our social media posts.

Motia also allows us to mix and match different languages within the same workflow providing great flexibility.

Check this typescript code:

After defining our steps, we install required dependencies using `npm install` and run the Motia workbench using `npm run dev` commands.

Check this out:

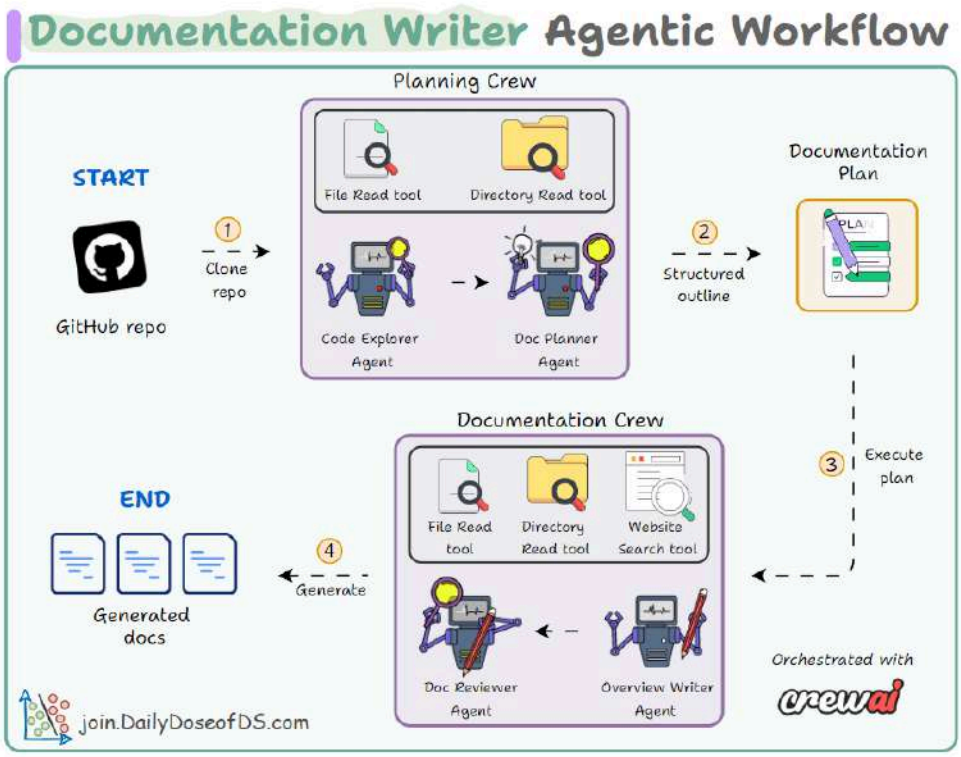
Motia workbench provides an interactive UI to help build, monitor and debug our flows.

With one-click you can also deploy it to the cloud!

The code is available here: https://blog.dailydoseofds.com/p/build-a-multi-agent-content-creation/

#11) Documentation Writer Flow

Build an Agentic workflow that generates full project documentation from just a GitHub repo URL.



Tech stack:

- CrewAI for multi-agent orchestration
- Ollama to locally serve DeepSeek-R1 LLM

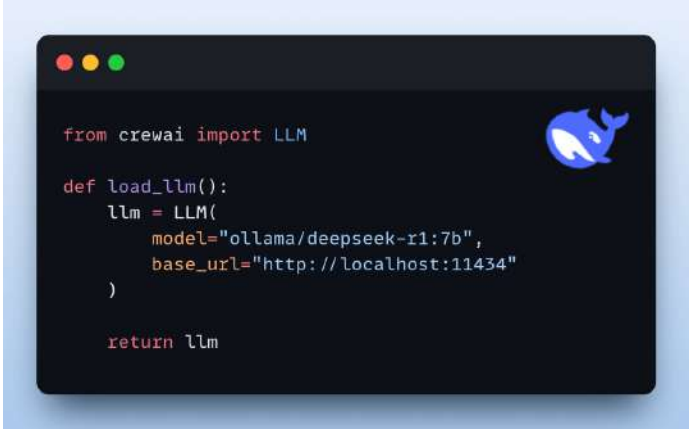
Workflow:

- User specifies GitHub repo
- Planning crew creates documentation plan
- Documentation crew writes documentation according to plan
- Generated docs get saved to local directory

Let's implement this!

#1) Setup LLM

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



#2) Define Pydantic schema

We define the following pydantic schemas for robust structured outputs.

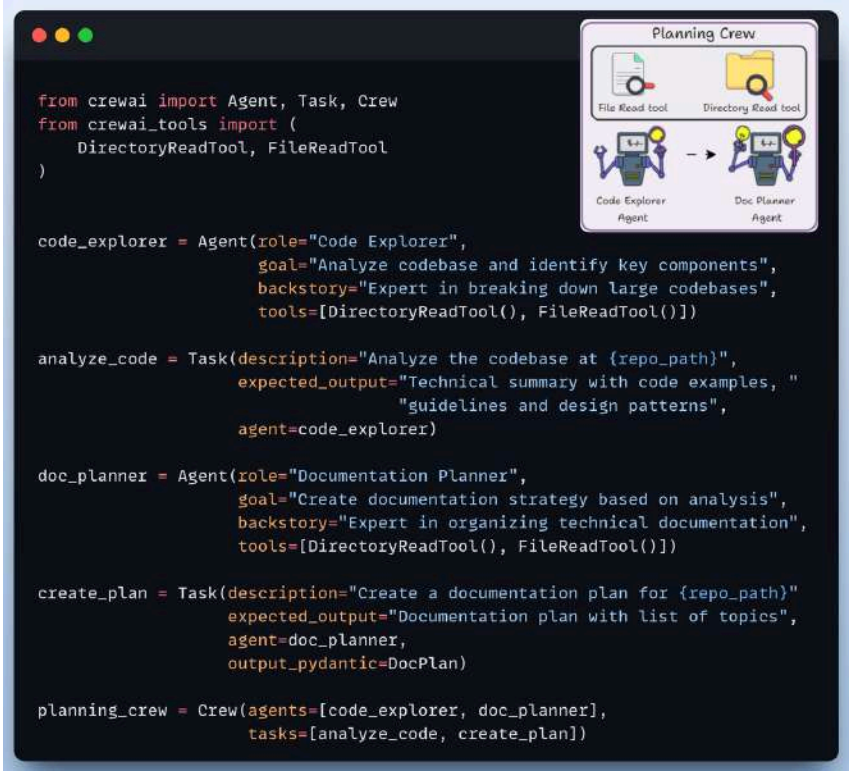
This ensures data validation and integrity before generating the documentation files. Check this code:

#3) Planning Crew

This crew oversees strategy for the outline via:

- Code explorer agent -> Analyzes codebase for key components, patterns and relationships.
- Doc planner agent -> Creates outline based on codebase analysis as pydantic output.

Check this out:



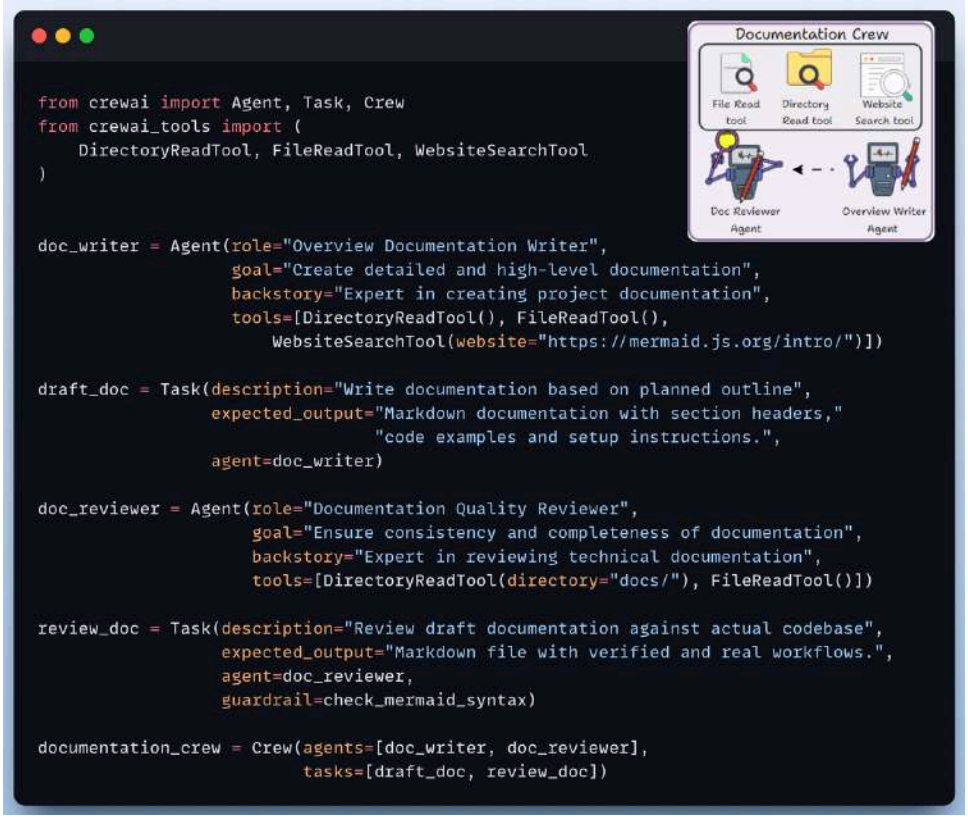
#4) Documentation Crew

This crew writes and reviews the documentation via:

- Doc writer agent -> Generates a high-level draft based on the planned outline.

- Doc reviewer agent -> Reviews the draft for consistency, accuracy and completeness.

Check this out:




```
from crewai import Agent, Task, Crew
from crewai_tools import (
    DirectoryReadTool, FileReadTool, WebsiteSearchTool
)


doc_writer = Agent(role="Overview Documentation Writer",
                   goal="Create detailed and high-level documentation",
                   backstory="Expert in creating project documentation",
                   tools=[DirectoryReadTool(), FileReadTool(),
                       WebsiteSearchTool(website="https://mermaid.js.org/intro/")])

draft_doc = Task(description="Write documentation based on planned outline",
                 expected_output="Markdown documentation with section headers,"
                                 "code examples and setup instructions.",
                 agent=doc_writer)

doc_reviewer = Agent(role="Documentation Quality Reviewer",
                     goal="Ensure consistency and completeness of documentation",
                     backstory="Expert in reviewing technical documentation",
                     tools=[DirectoryReadTool(directory="docs/"), FileReadTool()])

review_doc = Task(description="Review draft documentation against actual codebase",
                  expected_output="Markdown file with verified and real workflows.",
                  agent=doc_reviewer,
                  guardrail=check_mermaid_syntax)

documentation_crew = Crew(agents=[doc_writer, doc_reviewer],
                          tasks=[draft_doc, review_doc])
```

#5) Create Documentation Flow

After setting up our crews, we create the main workflow that:

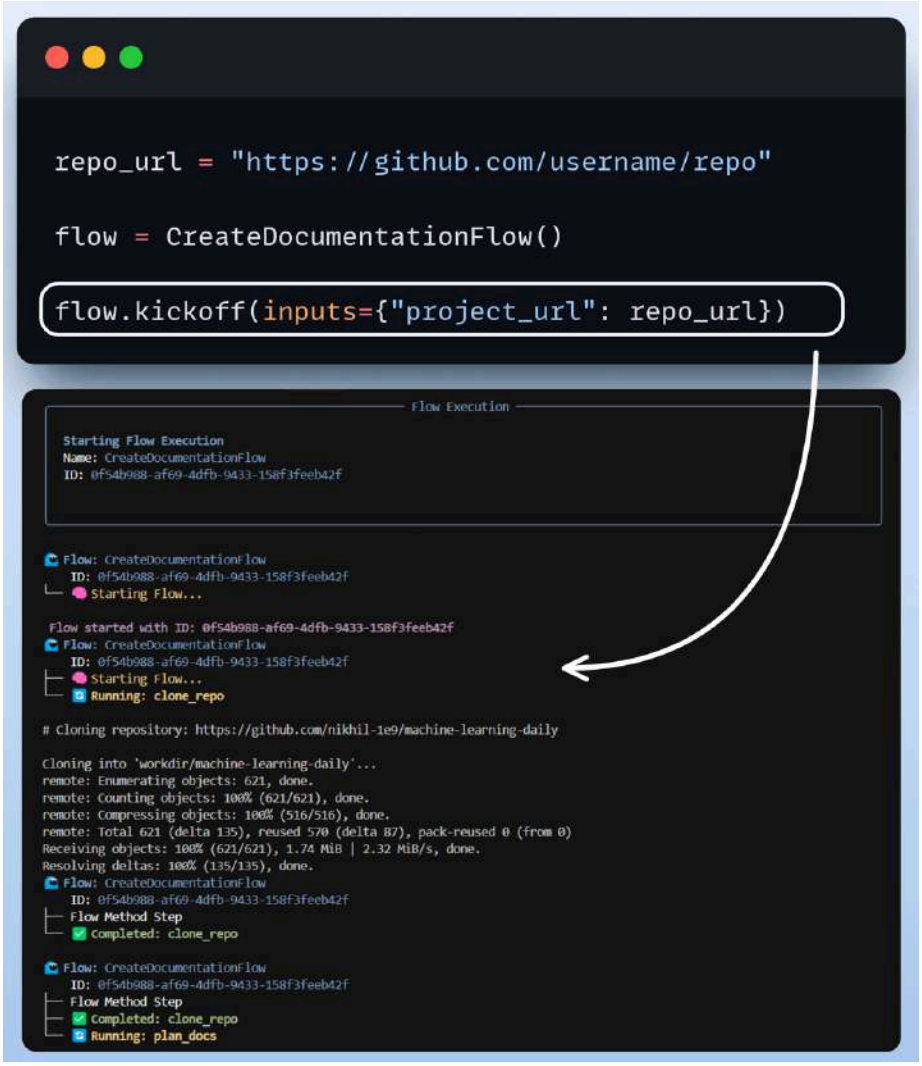
- Clones the GitHub repo
- Plans and saves the outline
- Generates documentation based on the outline
- Saves final docs to local directory

Check this code:

#6) Kickoff the flow

Finally, when we have everything ready, we kick off our documentation flow with the GitHub repo URL.

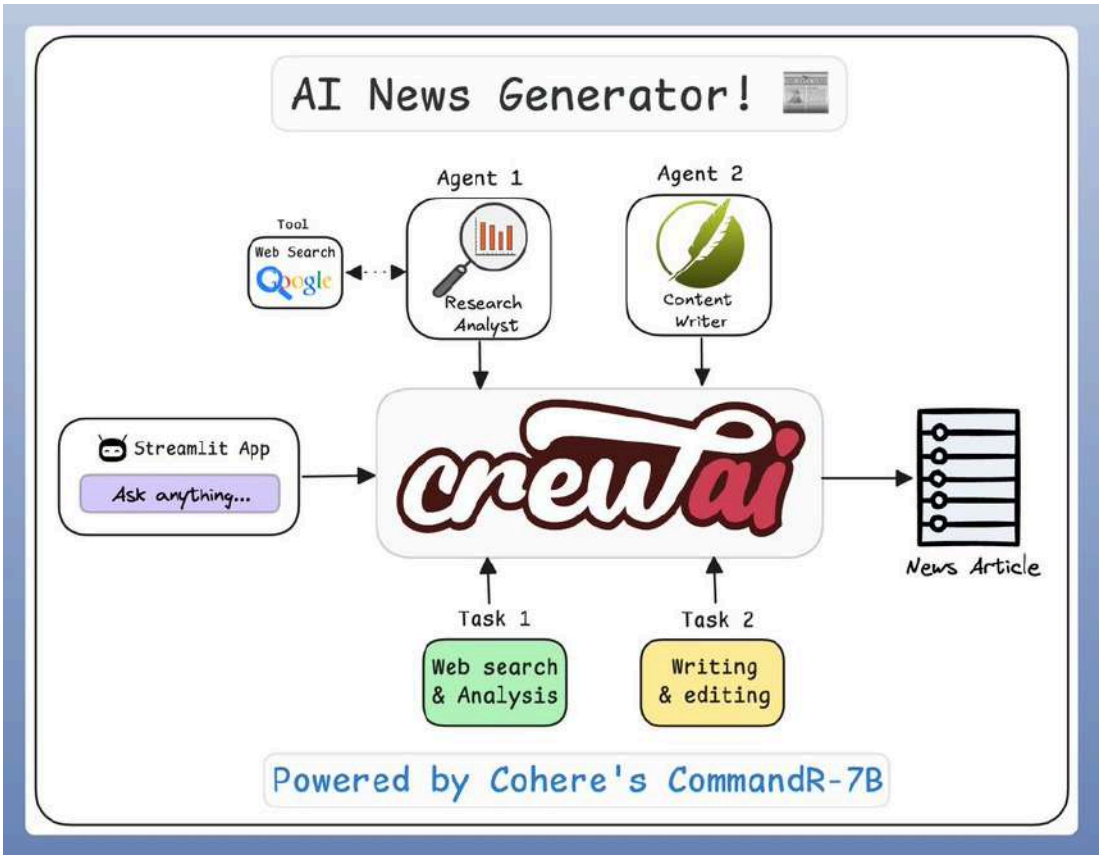
Check this out:



The code is available here:
https://github.com/patchy631/ai-engineering-hub/tree/main/documentation-writer-flow

#12) News Generator

The app takes a user query, searches the web for it, and turns it into a well-crafted news article, with citations.



Tech stack:

- Cohere ultra-fast Command R 7B as LLM
- CrewAI for multi-agent orchestration

Workflow:

We'll have two agents in this multi-agent app:

Research analyst agent:

- Accepts a user query.
- Uses the Serper web search tool to fetch results from the internet.
- Consolidates the results.

Content writer agent:

- Uses the curated results to prepare a polished, publication-ready article.

Let's implement this:

#1) Setup

Create a .env file for their corresponding API keys:

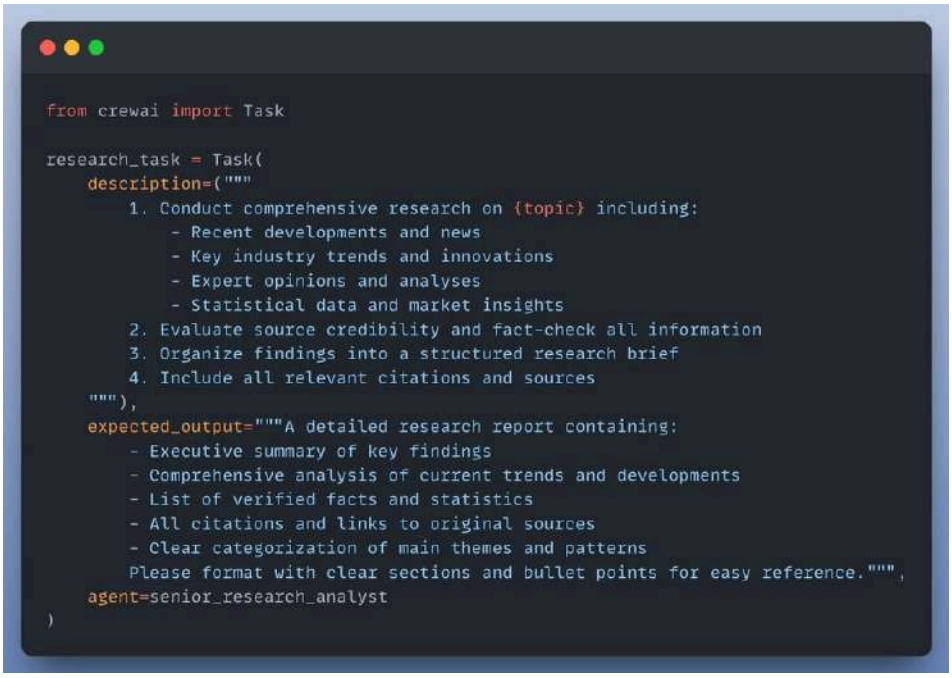
- Cohere API key
- Serper API key

Next, setup the LLM and web search tool as follows:

#2) Senior Research Analyst Agent

The web-search agent takes a user query and then uses the Serper web search tool to fetch results from the internet and consolidate them. Check this out:

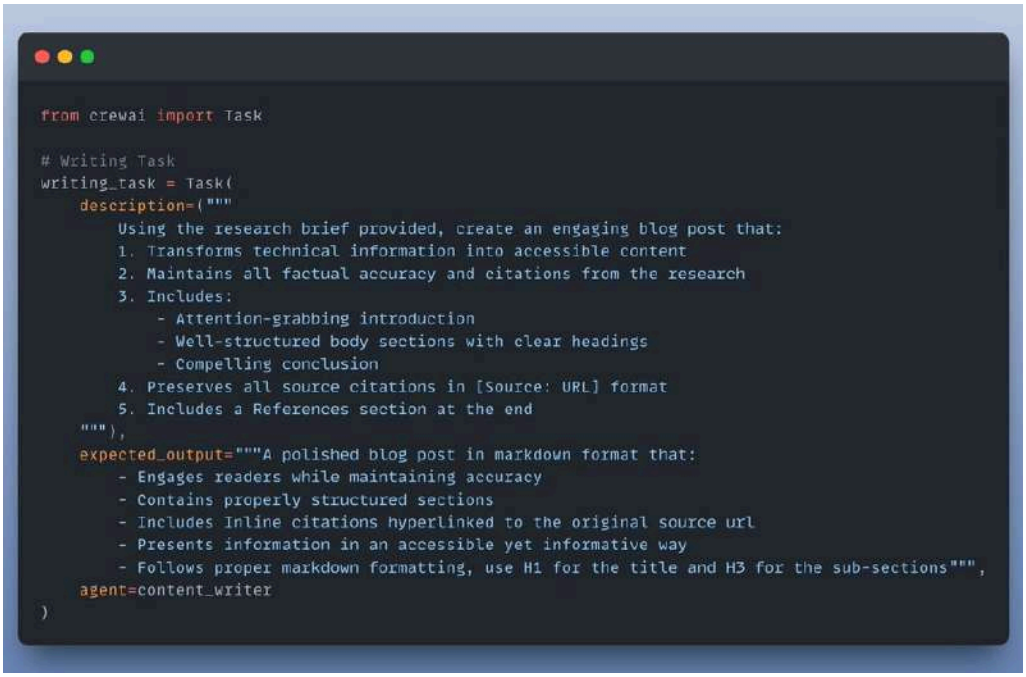
This is the research task that we assign to our senior research analyst agent, with description and expected output.



#3) Content writer agent

The role of content writer is to use the curated results and turn them into a polished, publication-ready news article .

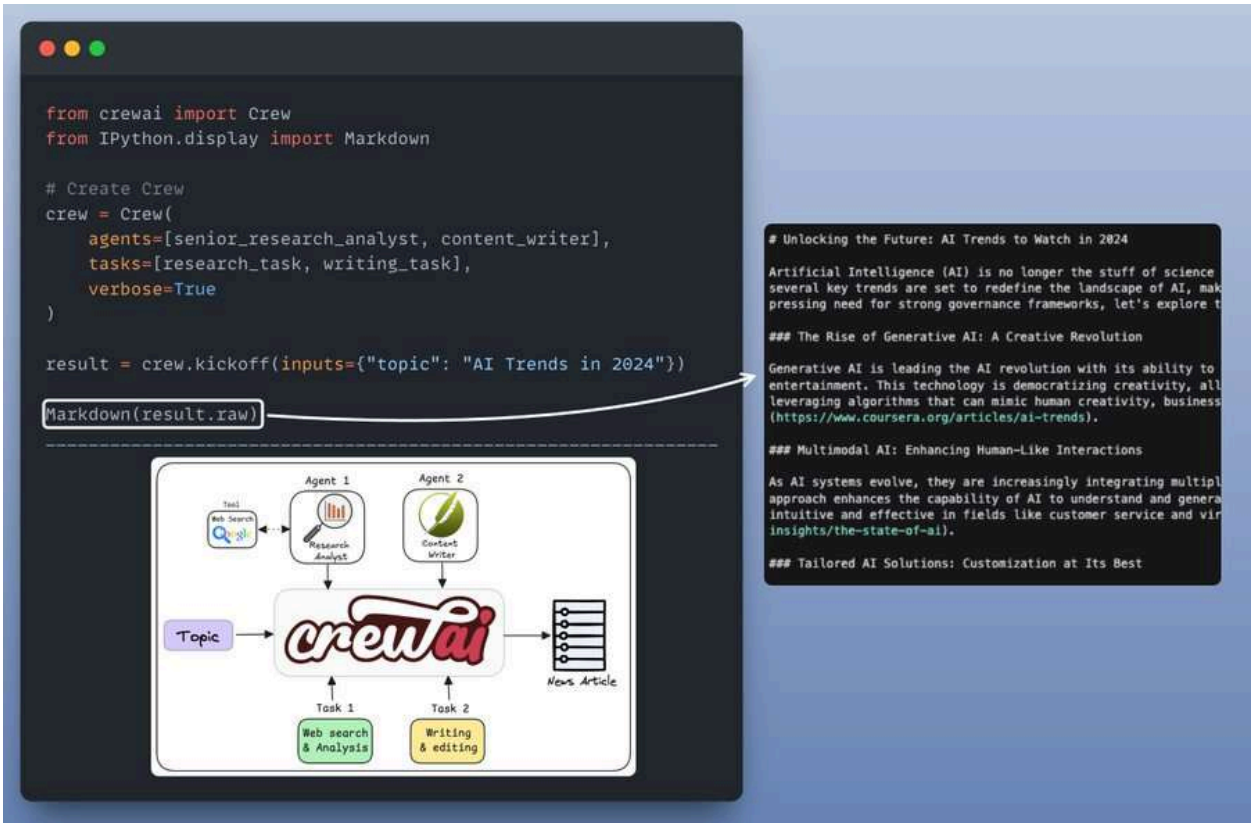
This is how we describe the writing task with all the details and expected output:



#4) Setup Crew

And we're done!

Just build a crew and kick it off!



The code is available here:
https://www.dailydoseofds.com/p/hands-on-building-a-multi-agent-news-generator/