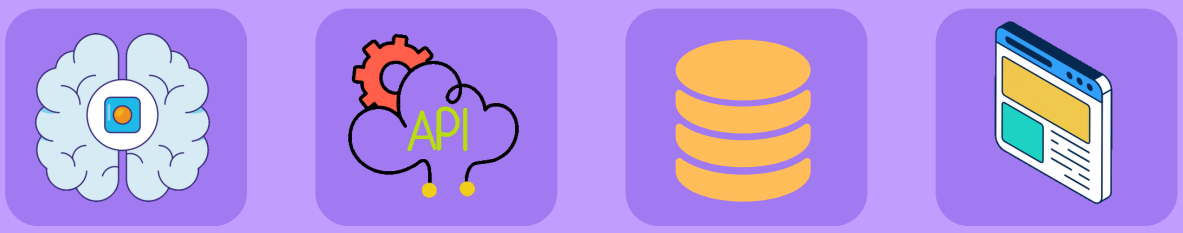
2025 EDITION

FREE

AI AGENTS

THE ILLUSTRATED GUIDEBOOK



Daily Dose of Data Science

Avi Chawla & Akshay Pachaar
DailyDoseofDS.com

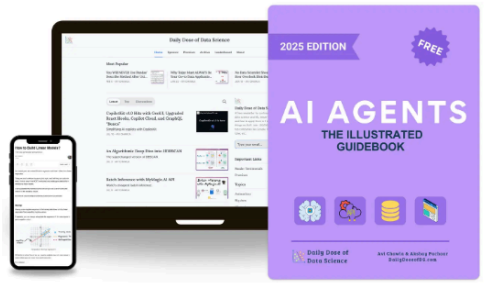
How to make the most out of this book and your time?

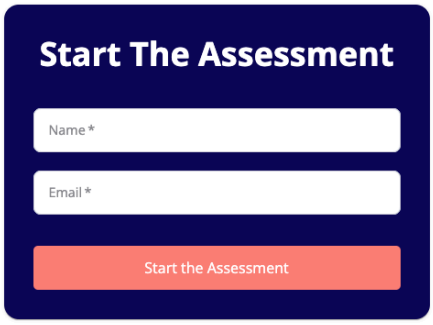
The reading time of this book is about 8 hours. But not all chapters will be of relevance to you. This 2-minute assessment will test your current expertise and recommend chapters that will be most useful to you.

Do you have the skills to build production-ready Agents?

Answer 10 yes/no questions and we'll email you the list of chapters you must read to level up your AI Agent skills

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.

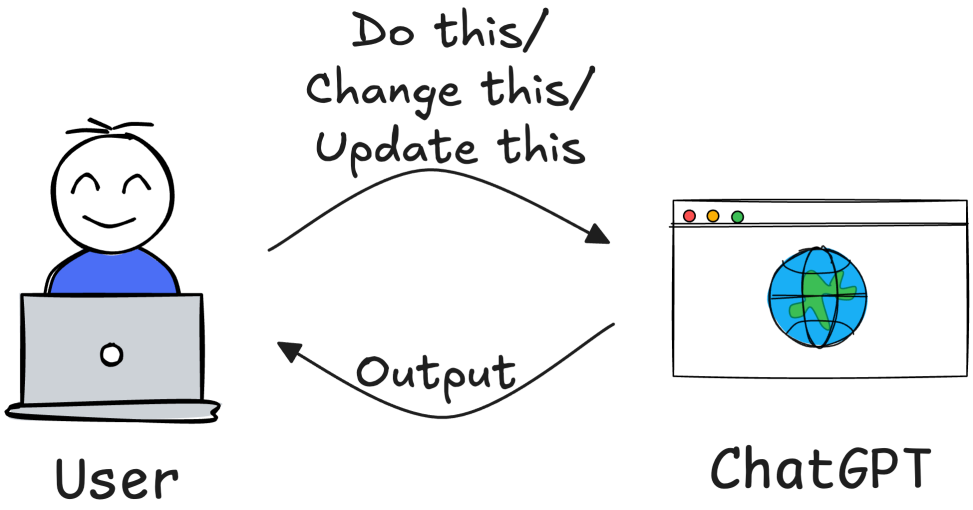
https://bit.ly/agents-assessment

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:

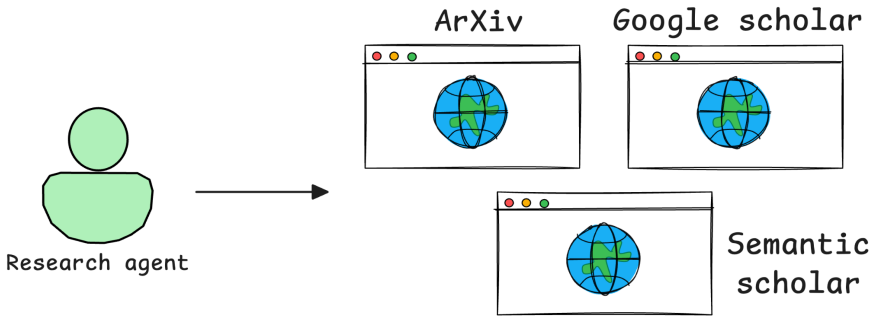


1. Ask for a summary of recent AI research papers.
2. Review the response and realize you need sources.
3. Obtain a list of papers along with citations.
4. Find that some sources are outdated, so you refine your query.
5. Finally, after multiple iterations, you get a useful output.

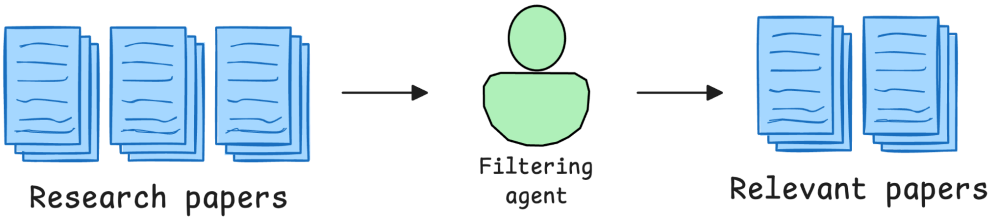
This iterative process takes time and effort, requiring you to act as the decision-maker at every step.

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

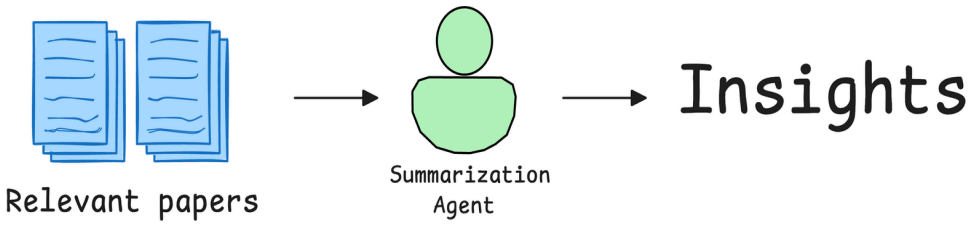
A Research Agent autonomously searches and retrieves relevant AI research papers from arXiv, Semantic Scholar, or Google Scholar.



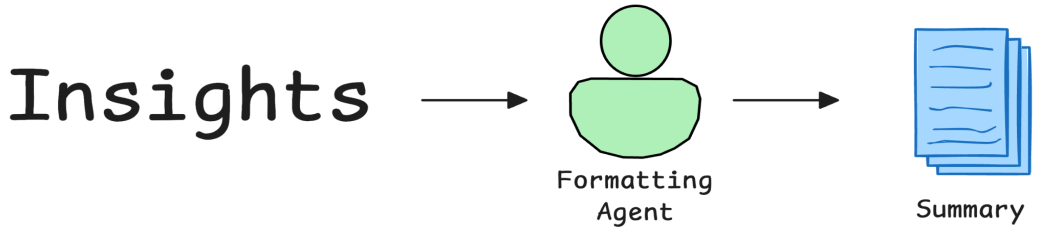
- A Filtering Agent scans the retrieved papers, identifying the most relevant ones based on citation count, publication date, and keywords.



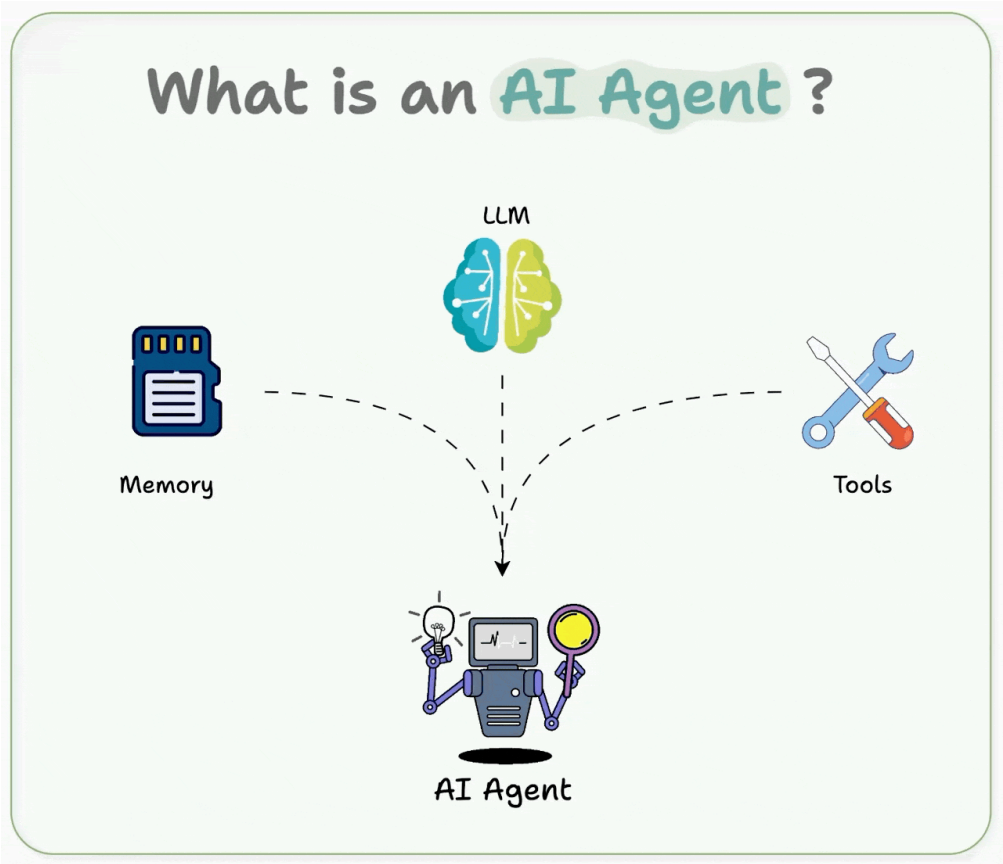
- A Summarization Agent extracts key insights and condenses them into an easy-to-read report.



- 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

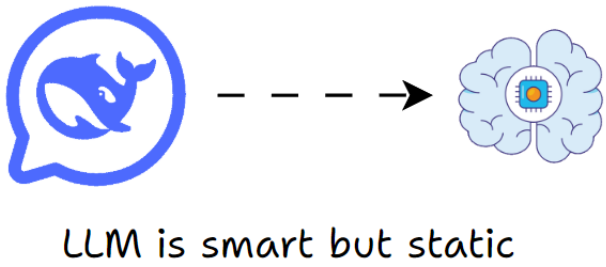
Let's break it down with a simple analogy:

- 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.

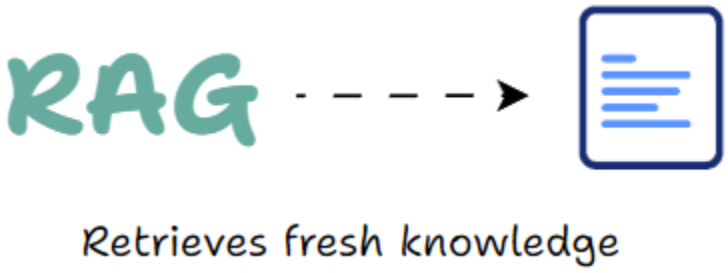
It can reason, generate, summarize but only using what it already knows (i.e., its training data).



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

RAG (Retrieval-Augmented Generation)

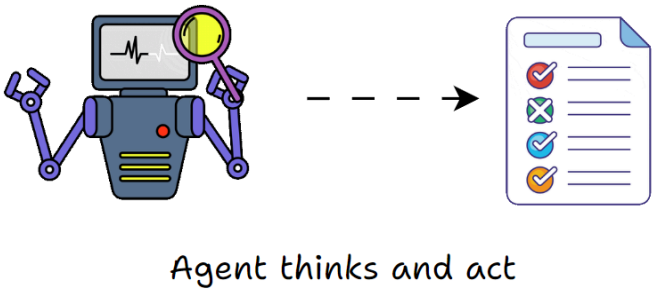
RAG enhances an LLM by retrieving external documents (from a vector DB, search engine, etc.) and feeding them into the LLM as context before generating a response.



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

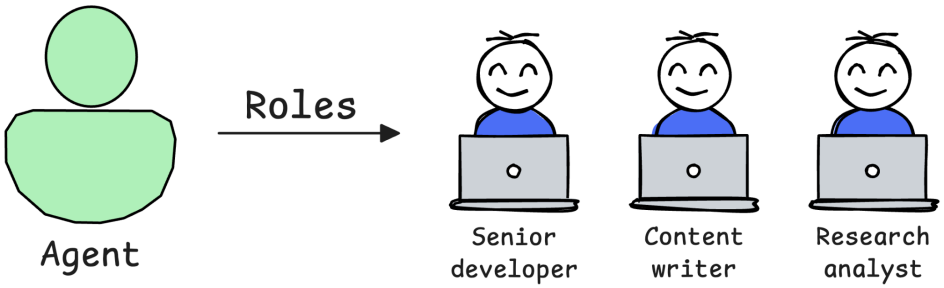
AI agents are designed to reason, plan, and take action autonomously. However, to be effective, they must be built with certain key principles in mind. There are six essential building blocks that make AI agents more reliable, intelligent, and useful in real-world applications:

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?

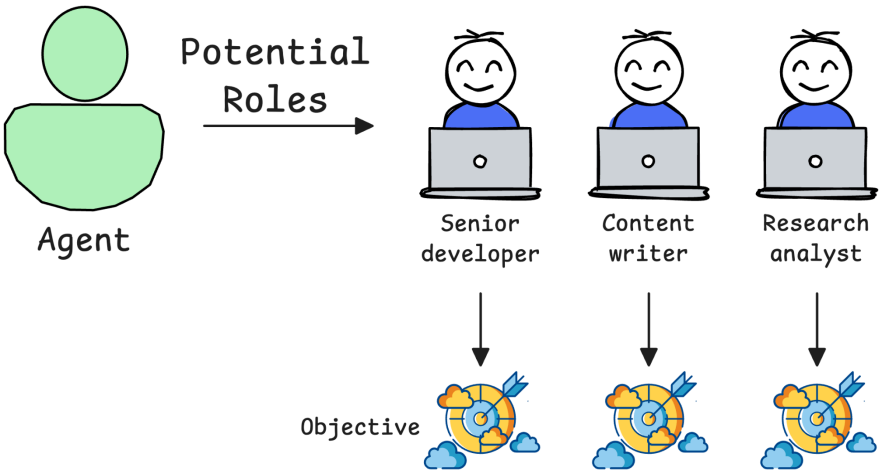
Because role assignment shapes the agent's reasoning and retrieval process. The

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.

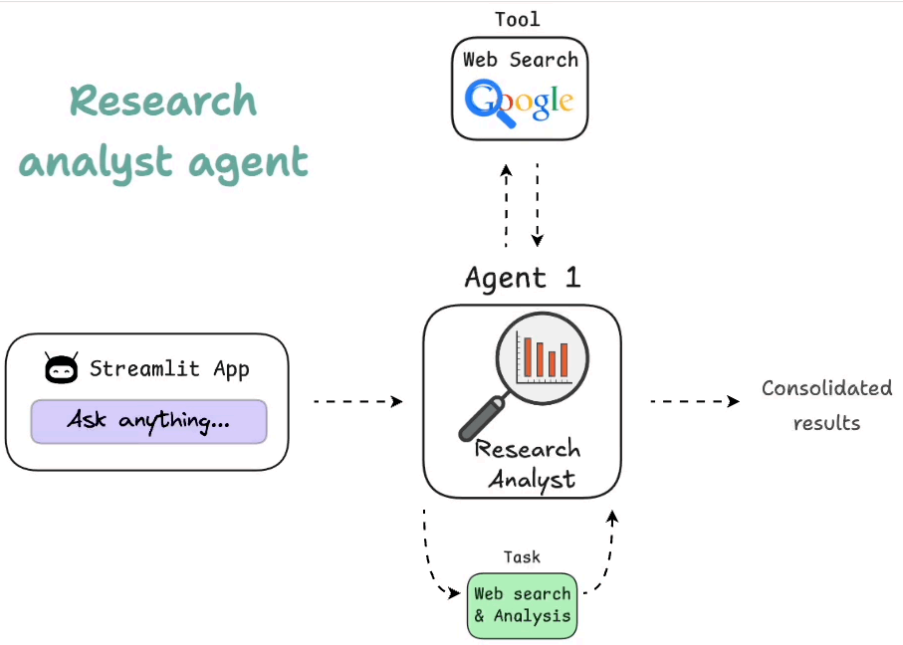
For example, a marketing agent should stick to messaging, tone, and audience not pricing or market analysis.

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.

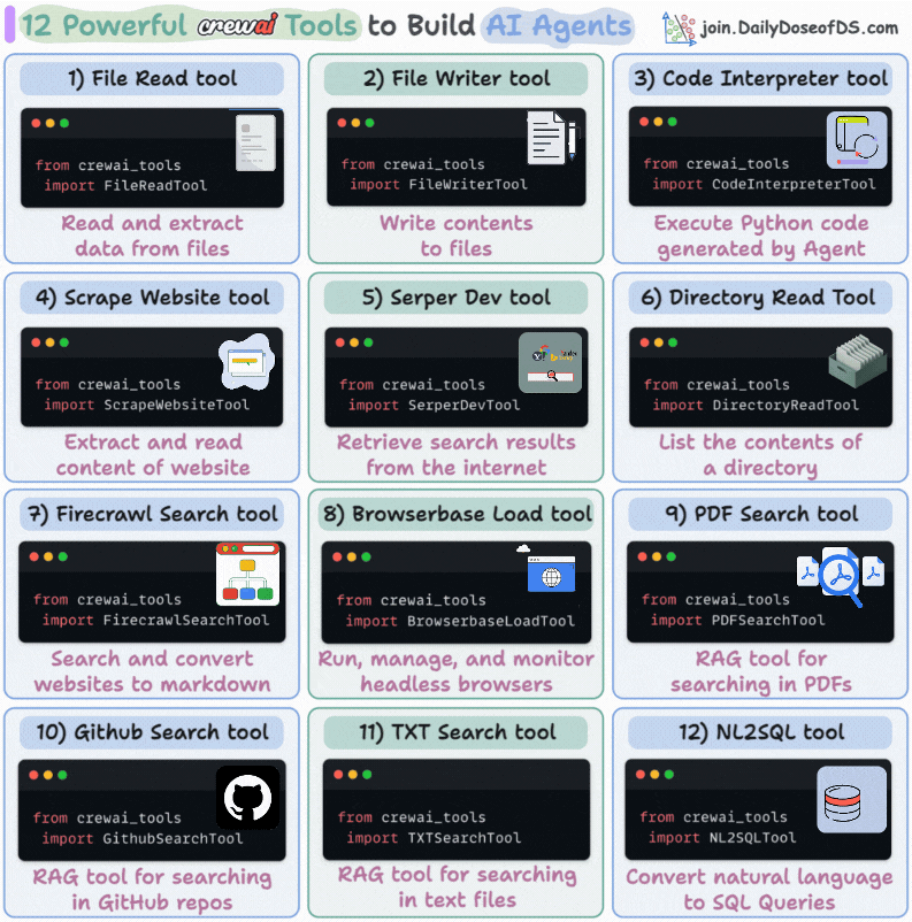
#3.1) Custom tools

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:

```
notebook.ipynb

!pip install crewai-tools
```

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:

```
.env

OPENAI_API_KEY="sk-4..."
SERPER_API_KEY="42131..."
EXCHANGE_RATE_API_KEY="753..."
```

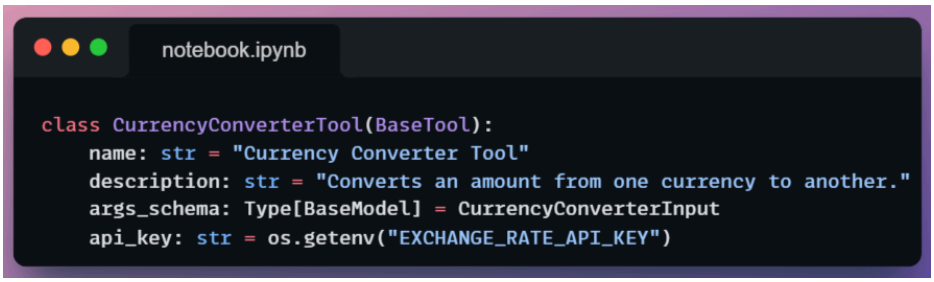
Once that's done, we start with some standard import statements:

```
notebook.ipynb

from dotenv import load_dotenv
load_dotenv()
import os
import requests
from typing import Type
from crewai.tools import BaseTool
from pydantic import BaseModel, Field
```

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

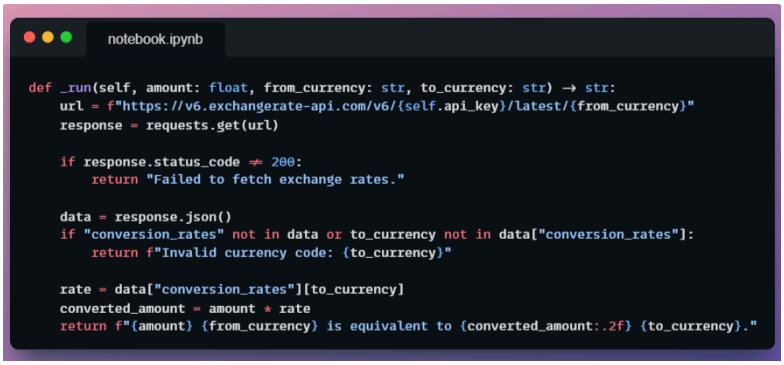
```
notebook.ipynb

class CurrencyConverterInput(BaseModel):
    """Input schema for CurrencyConverterTool."""
    amount: float = Field(..., description="The amount to convert.")
    from_currency: str = Field(..., description="The source currency code (e.g., 'USD').")
    to_currency: str = Field(..., description="The target currency code (e.g., 'EUR').")
```

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



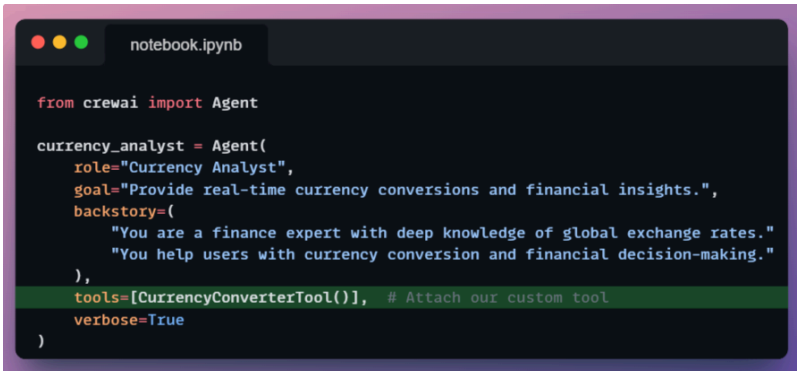
Every tool class should have the _run method which we will execute whenever the Agents wants to make use of it.

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.

notebook.ipynb

```
from crewai import Task

currency_conversion_task = Task(
    description=(
        "Convert {amount} {from_currency} to {to_currency} "
        "using real-time exchange rates."
        "Provide the equivalent amount and "
        "explain any relevant financial context."
    ),
    expected_output=("A detailed response including the "
                     "converted amount and financial insights."),
    agent=currency_analyst
)
```

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

notebook.ipynb

```
from crewai import Crew, Process

crew = Crew(
    agents=[currency_analyst],
    tasks=[currency_conversion_task],
    process=Process.sequential
)

response = crew.kickoff(inputs={"amount": 100,
                                "from_currency": "USD",
                                "to_currency": "EUR"})
```

Printing the response, we get the following output:

```
from IPython.display import Markdown
Markdown(response.raw)
```

✓ 0.0s Python

Converting 100 USD to EUR using real-time exchange rates results in approximately 95.40 EUR.

In the financial context, it's worth noting that exchange rates can fluctuate due to various factors like economic indicators, interest rates, and geopolitical events. As of now, the conversion reflects current market conditions, which are influenced by the latest economic data releases and monetary policies in both the United States and the Eurozone. Given the recent trends, if you're planning a trip to Europe or making an investment, these rates may change, so it's beneficial to monitor them regularly.

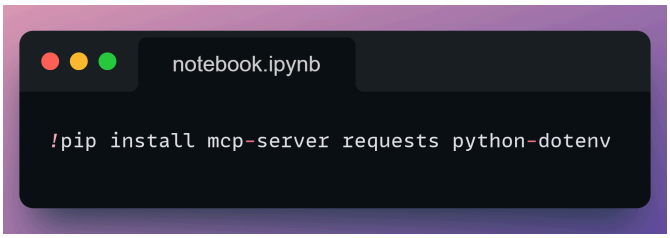
Works as expected!

#3.2) Custom tools via MCP

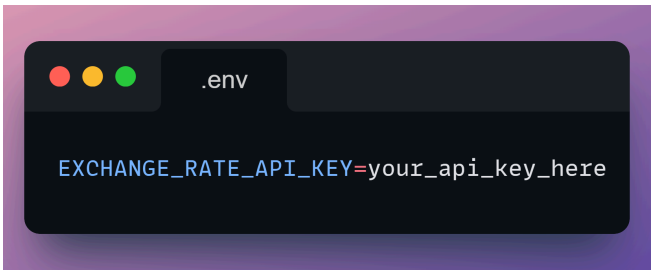
Now, let's take it a step further.

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:

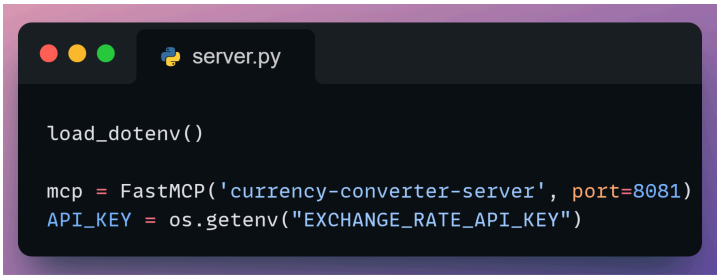


We'll continue using ExchangeRate-API in our .env file:

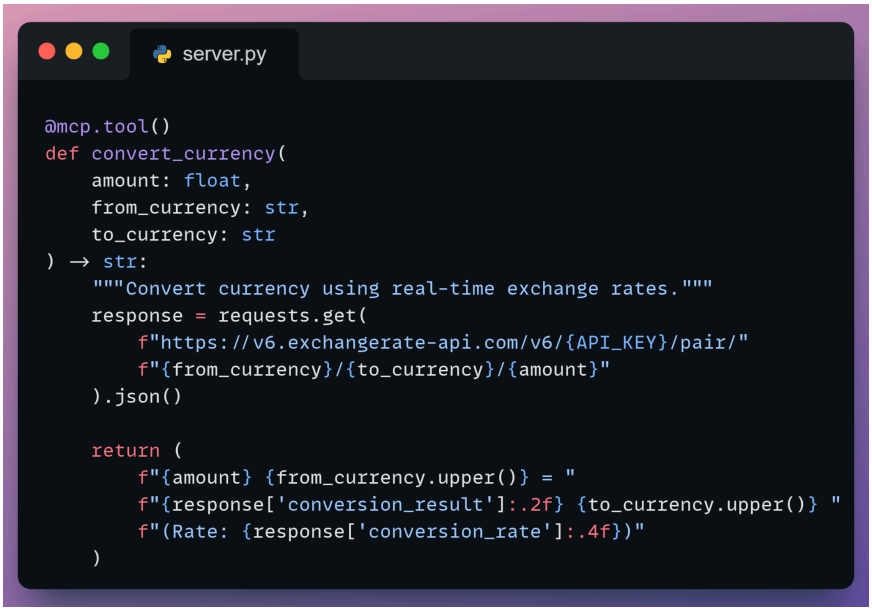


We'll now write a lightweight server.py script that exposes the currency converter tool. We start with the standard imports:

Now, we load environment variables and initialize the server:



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



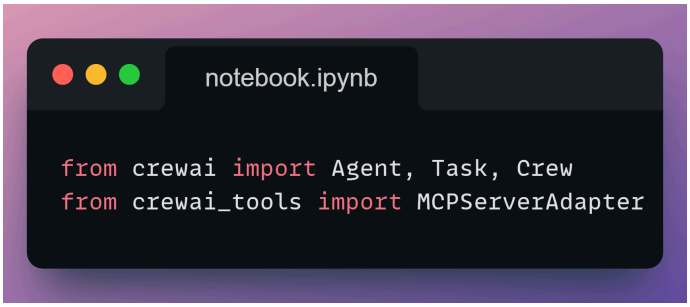
This function takes three inputs—amount, source currency, and target currency—and returns the converted result using the real-time exchange rate API.

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

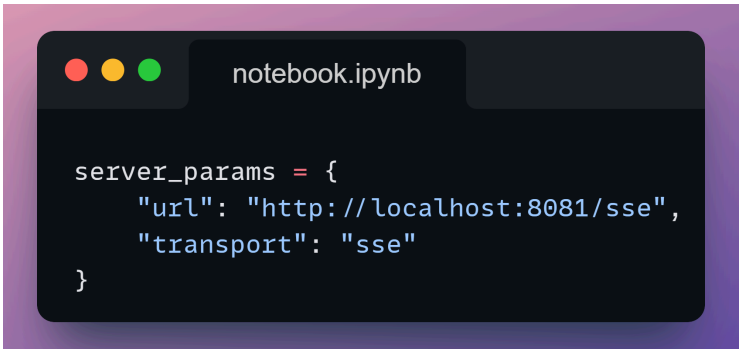
This starts the server and exposes your convert_currency tool at: http://localhost:8081/sse.

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.

```
currency_agent = Agent(
    role="Currency Analyst",
    goal="Convert currency using real-time exchange rates.",
    backstory=(
        "You help users convert between currencies "
        "using up-to-date market data."
    ),
    allow_delegation=False,
    tools=[mcp_tools["convert_currency"]],
)
```

We give the agent a task description:

```
conversion_task = Task(
    description=(
        "Convert {amount} {from_currency} to {to_currency} "
        "using real-time exchange rates."
    ),
    agent=currency_agent,
    expected_output="A formatted result with exchange rate.",
)
```

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

```
crew = Crew(
    agents=[currency_agent], tasks=[conversion_task], verbose=True
)

result = crew.kickoff(inputs={
    "amount": 100, "from_currency": "USD", "to_currency": "INR"
})

print(result)
```

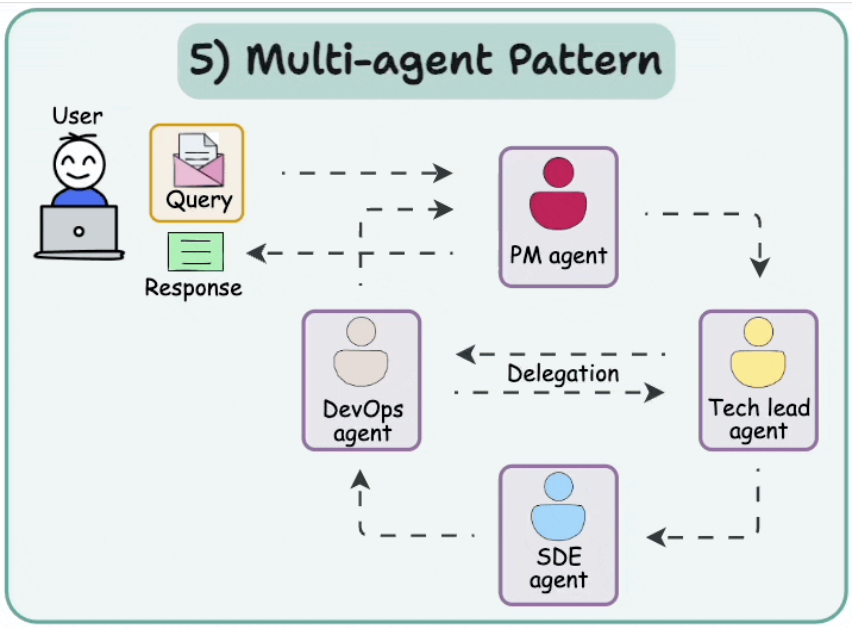
Printing the result, we get the following output:

```
    print(result)

[1]  ✓ 2.8s

...  100 USD = 8734.13 INR (Rate: 87.3413)
```

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

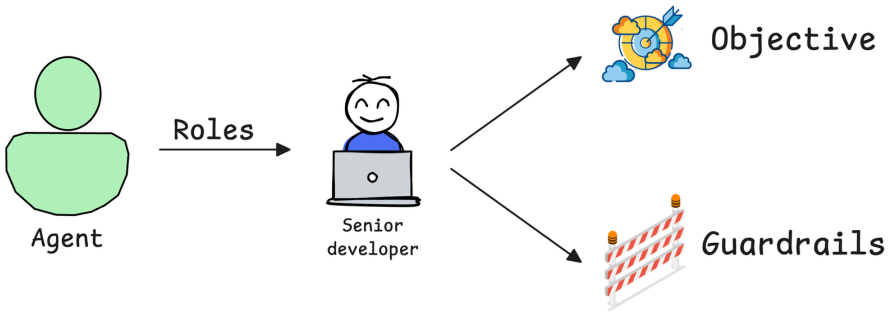
Collaboration leads to smarter, more accurate results.

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:

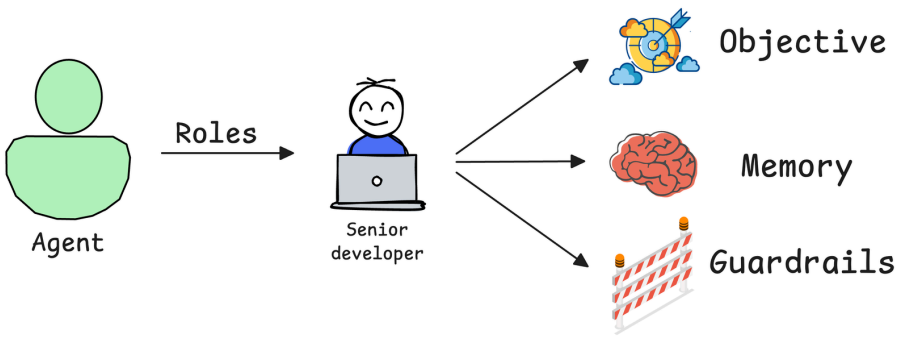
- Limiting tool usage: Prevent an agent from overusing APIs or generating irrelevant queries.
- Setting validation checkpoints: Ensure outputs meet predefined criteria before moving to the next step.
- Establishing fallback mechanisms: If an agent fails to complete a task, another agent or human reviewer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

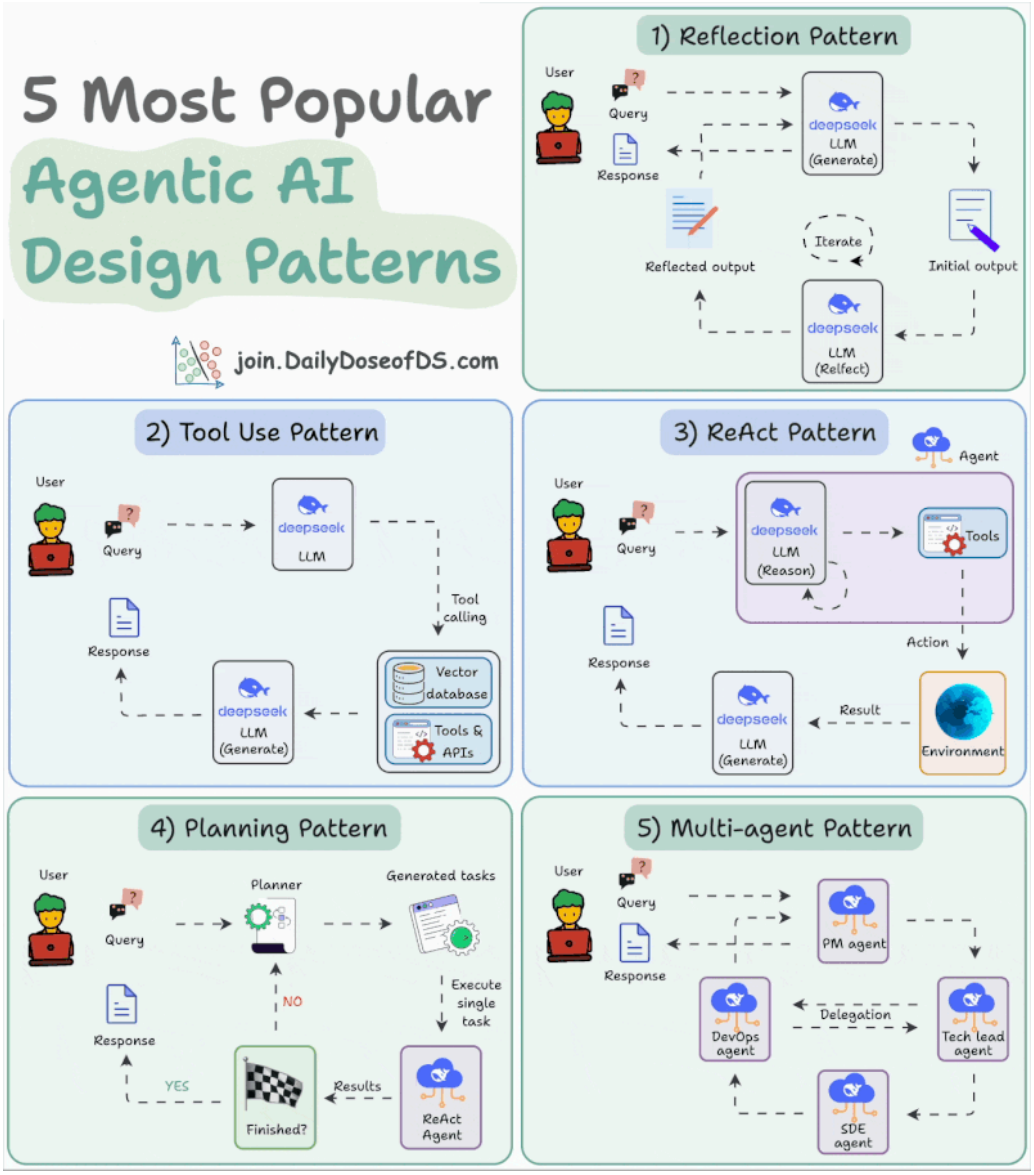
- Short-term memory – Exists only during execution (e.g., recalling recent conversation history).
- Long-term memory – Persists after execution (e.g., remembering user preferences over multiple interactions).
- Entity memory – Stores information about key subjects discussed (e.g., tracking customer details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

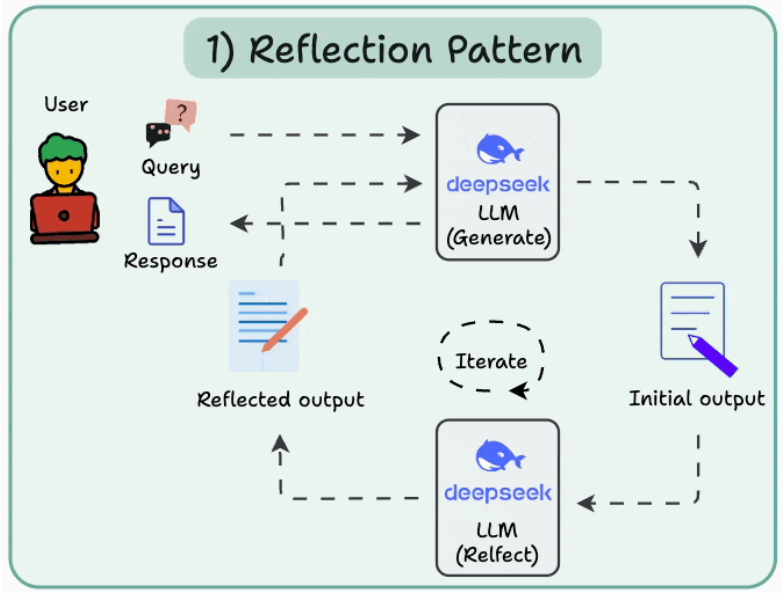
5 Agentic AI Design Patterns

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.

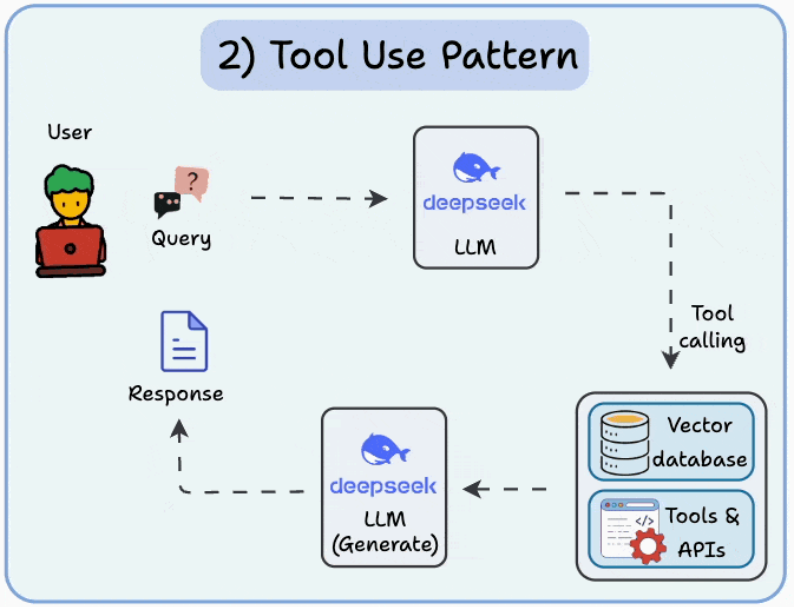


#1) Reflection pattern



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

#2) Tool use pattern

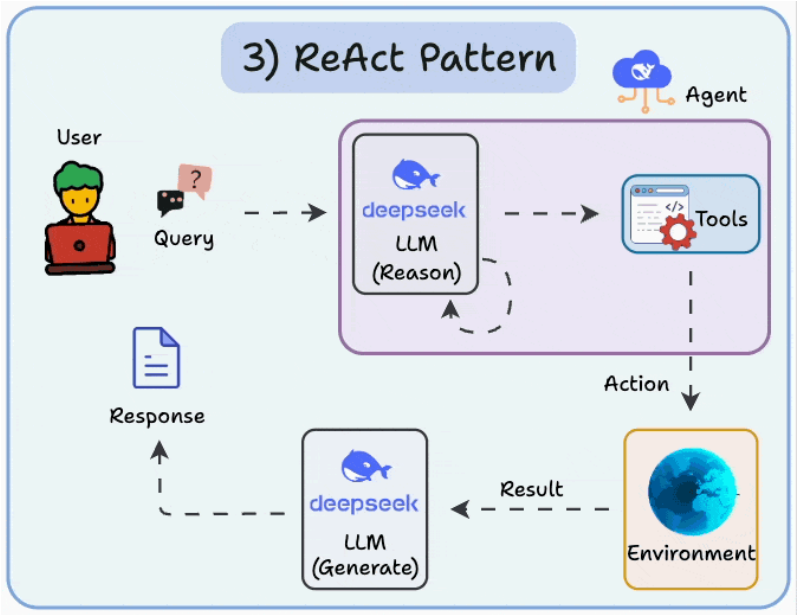


Tools allow LLMs to gather more information by:

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

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

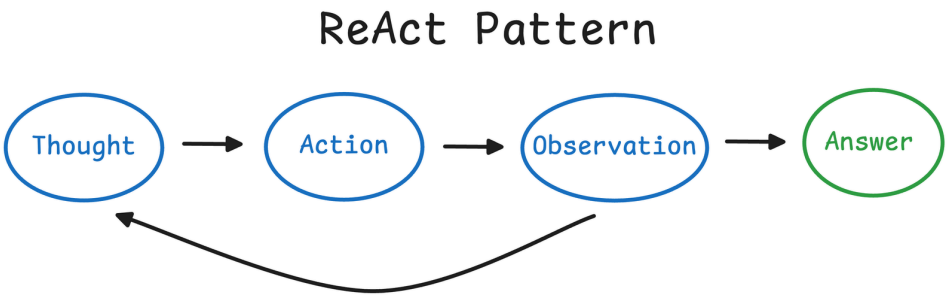
#3) ReAct (Reason and Act) pattern



ReAct combines the above two patterns:

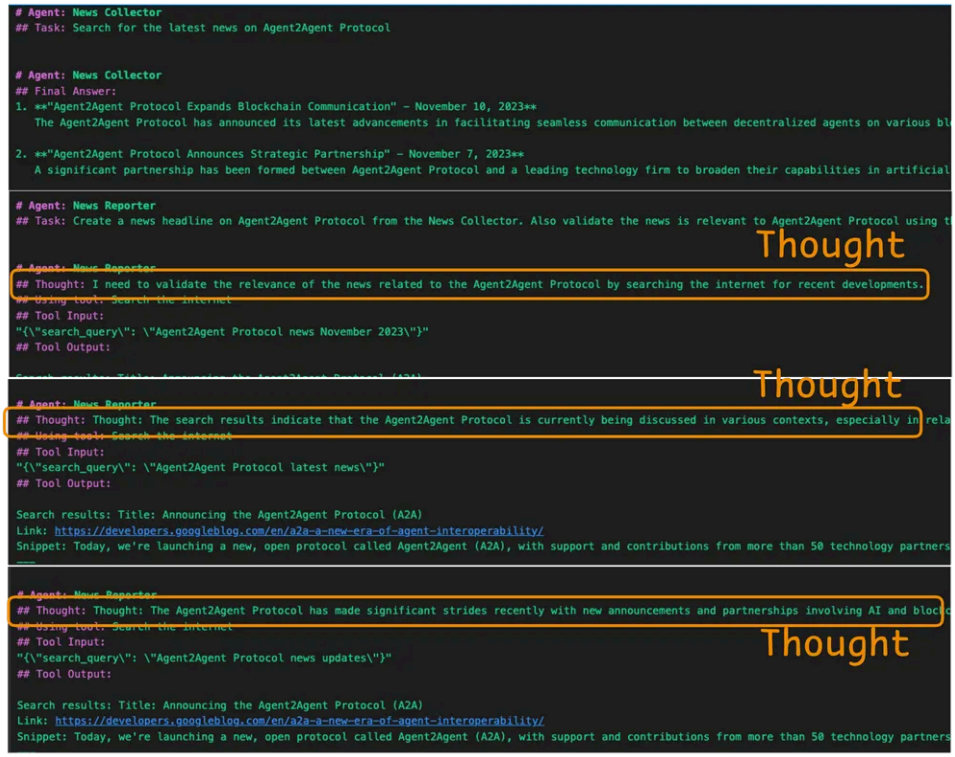
- The Agent reflects on the generated outputs.
- It interacts with the world using tools.

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.

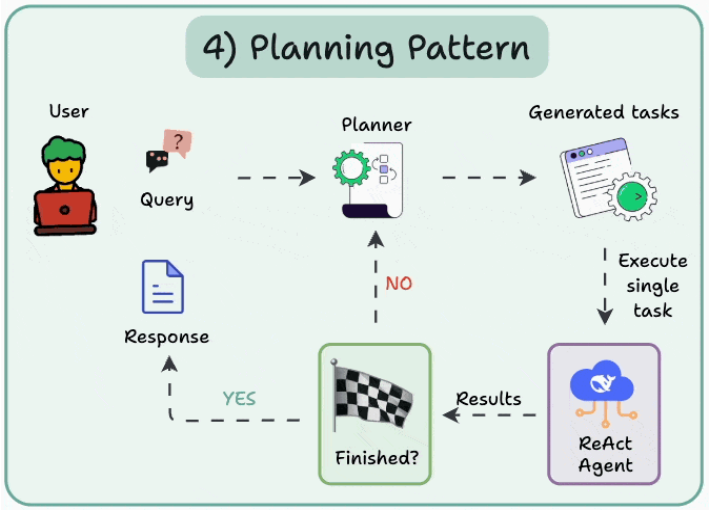
This is the ReAct pattern in action!

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



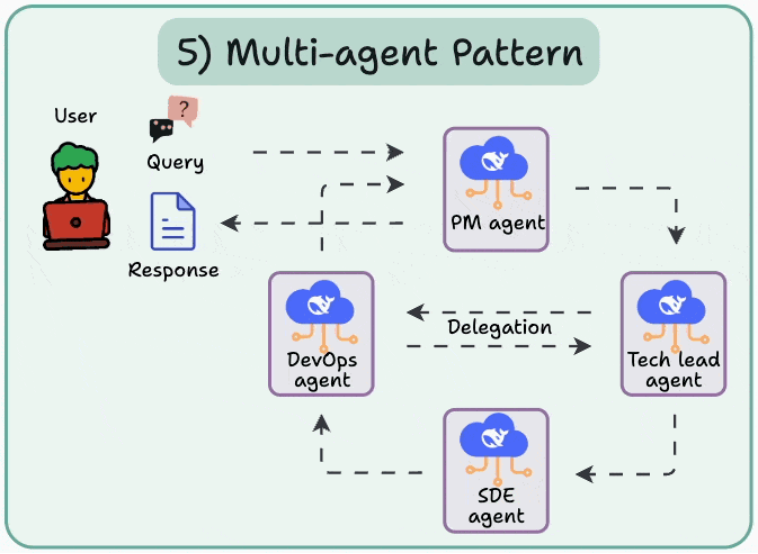
Instead of solving a task in one go, the AI creates a roadmap by:

- Subdividing tasks
- Outlining objectives

This strategic thinking solves tasks more effectively.

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

#5) Multi-Agent pattern



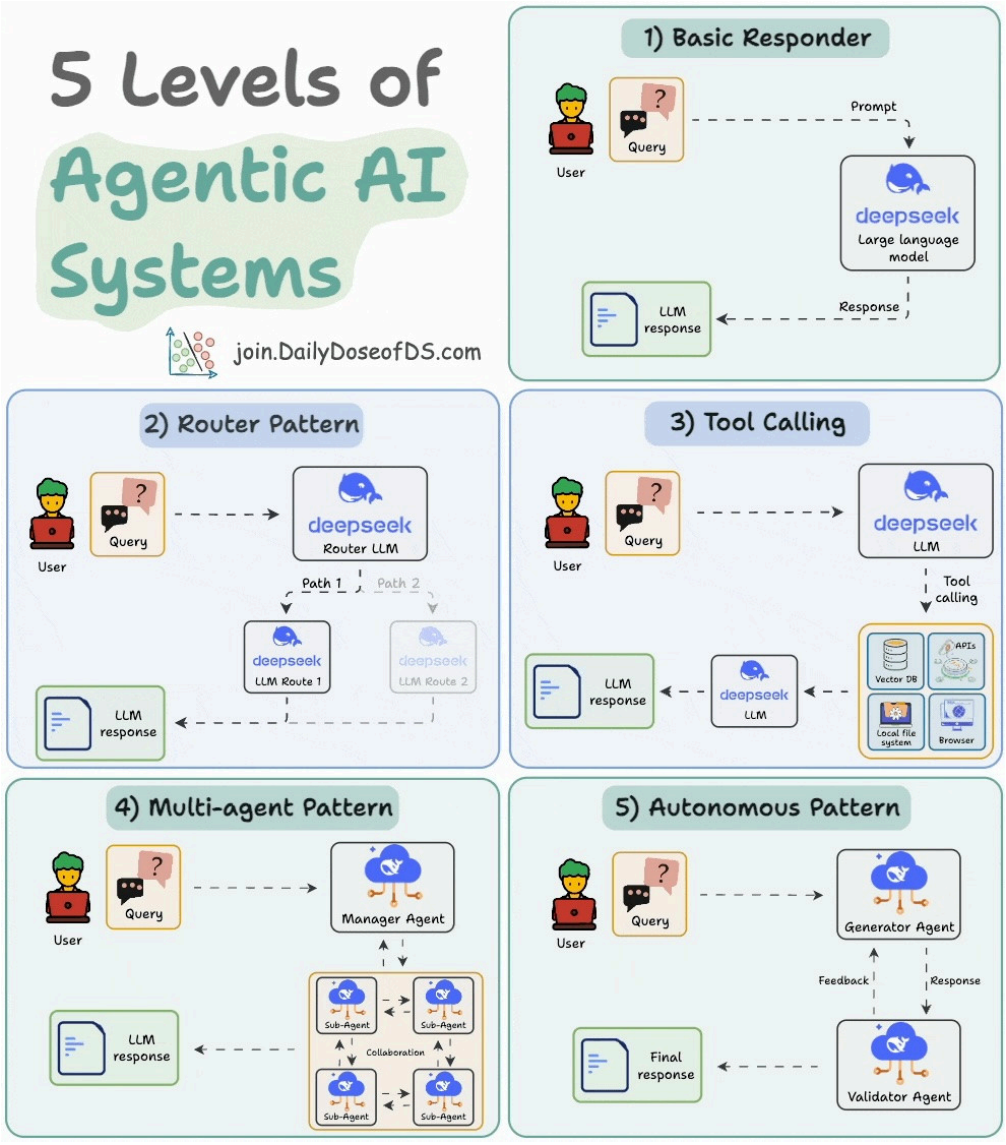
- There are several agents, each with a specific role and task.
- Each agent can also access tools.

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

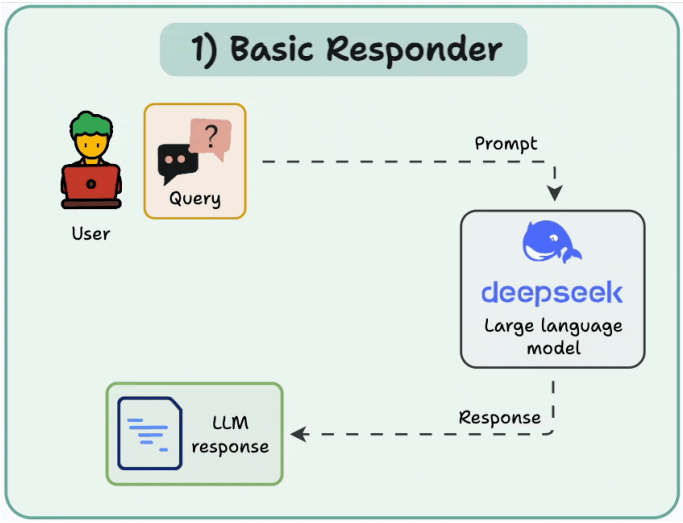
5 Levels of Agentic AI Systems

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.



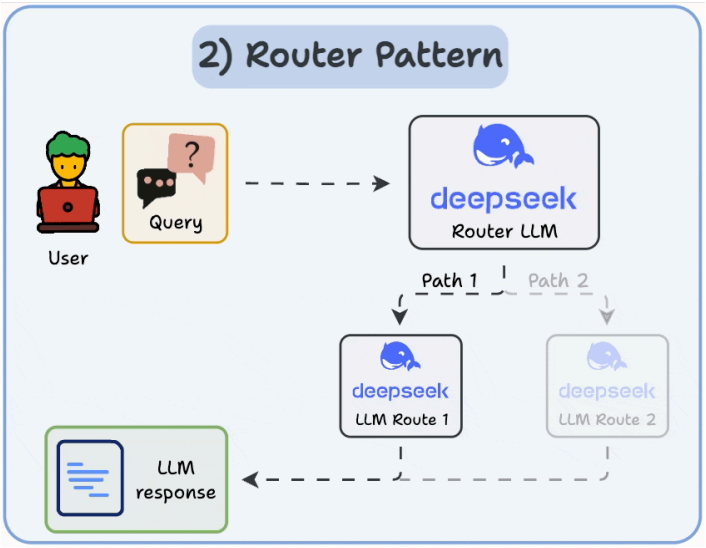
#1) Basic responder



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.

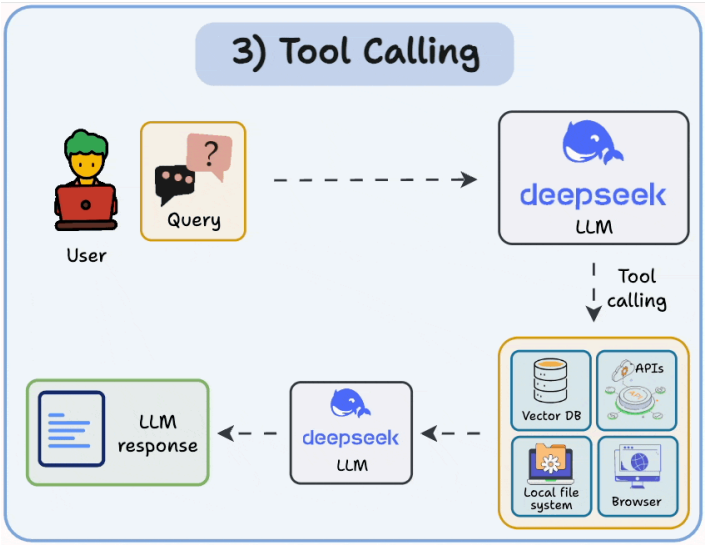
#2) Router pattern



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

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

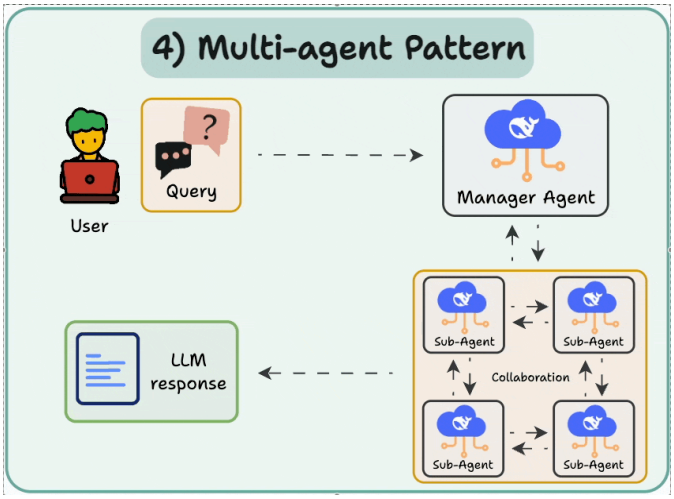
#3) Tool calling



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

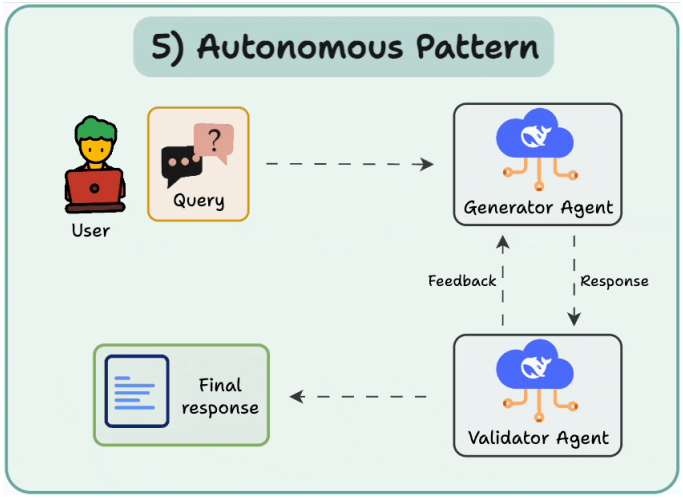


A manager agent coordinates multiple sub-agents and decides the next steps iteratively.

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

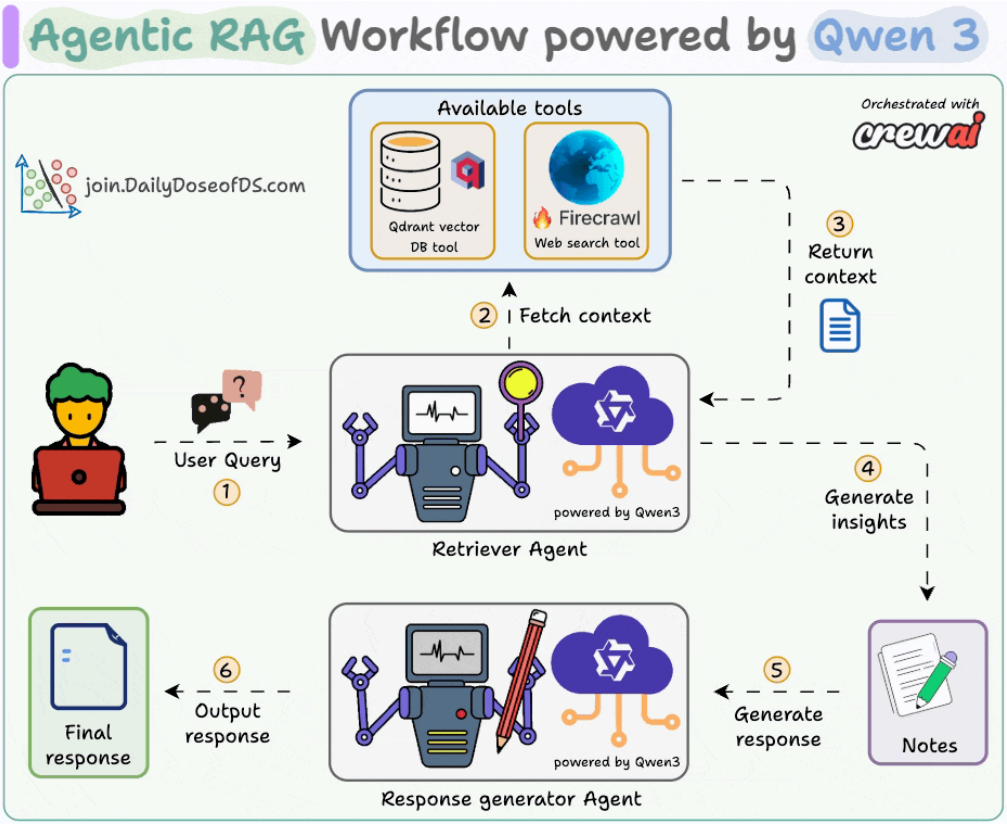


The most advanced pattern, wherein, the LLM generates and executes new code independently, effectively acting as an independent AI developer.

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.

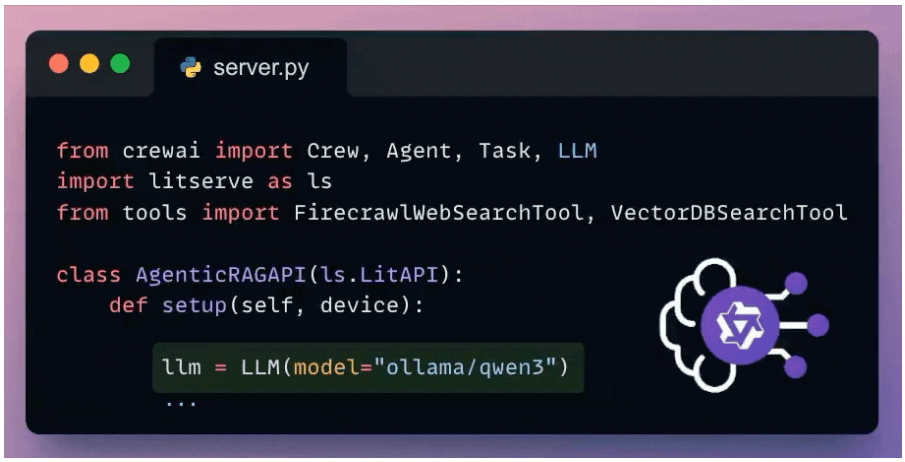
Workflow:

- 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

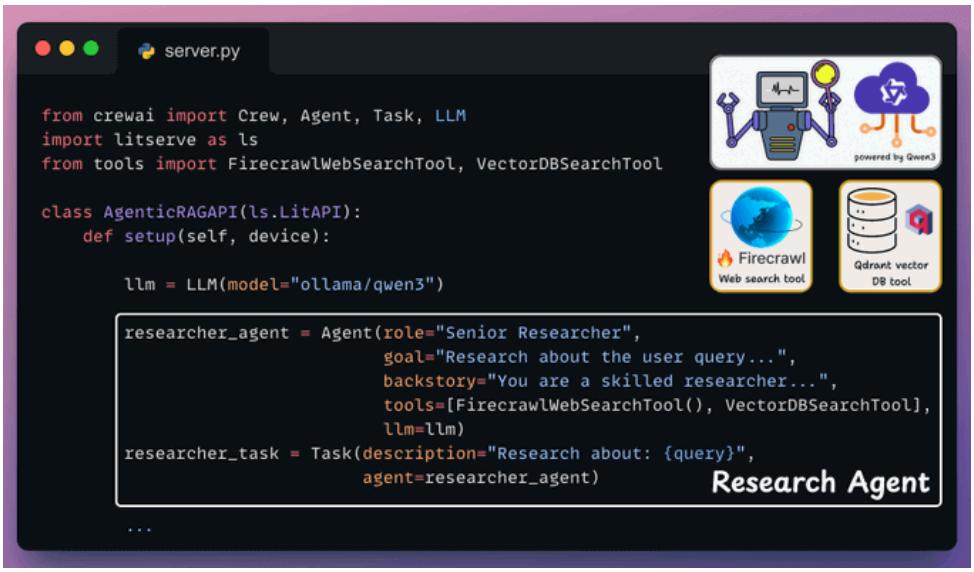
CrewAI seamlessly integrates with all popular LLMs and providers. Here's how we set up a local Qwen 3 via Ollama:



#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.

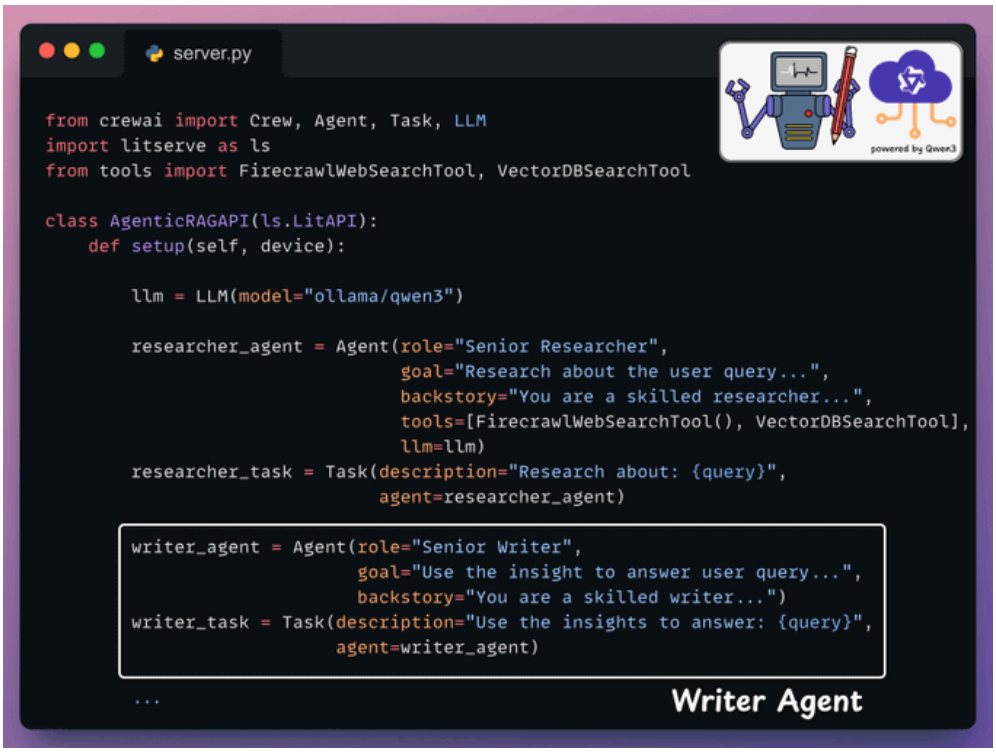
Again, put this in the LitServe setup() method:



#3) Define Writer Agent and Task

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

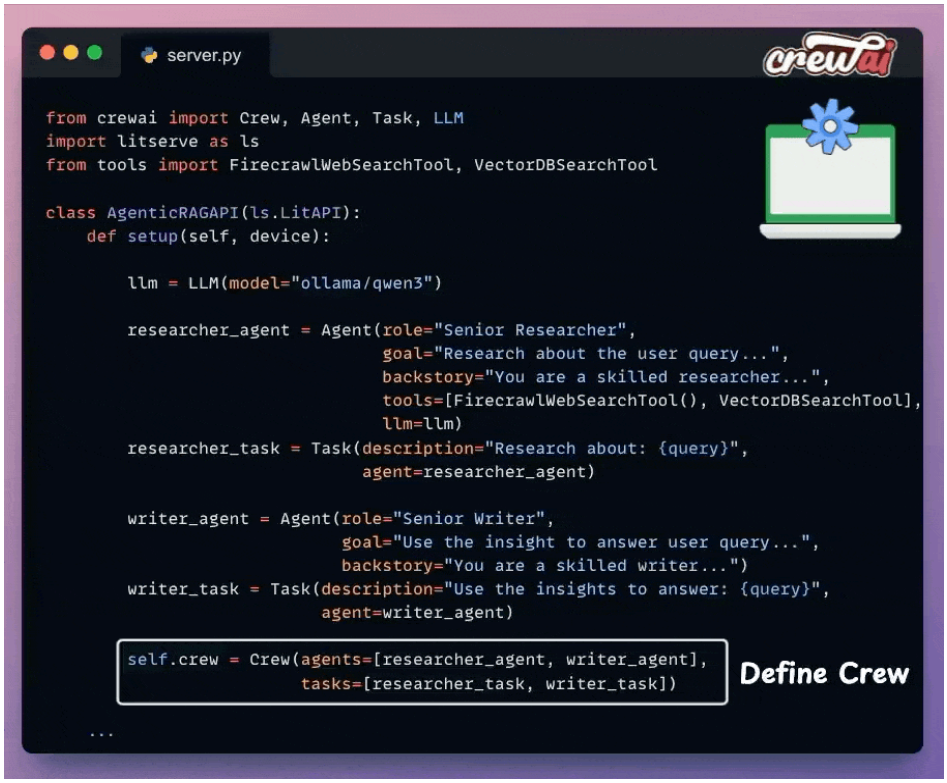
Yet again, we add this in the LitServe setup method:



#4) Set up the Crew

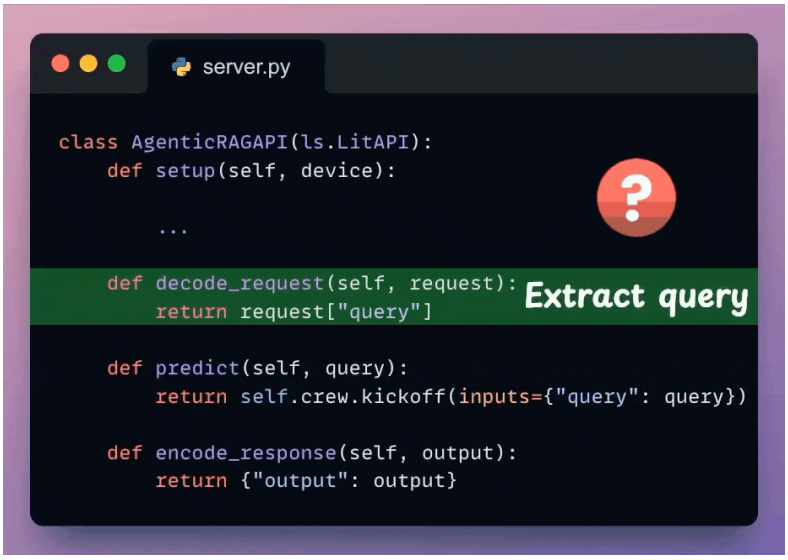
Once we have defined the Agents and their tasks, we orchestrate them into a crew using CrewAI and put that into a setup method.

Check this code:



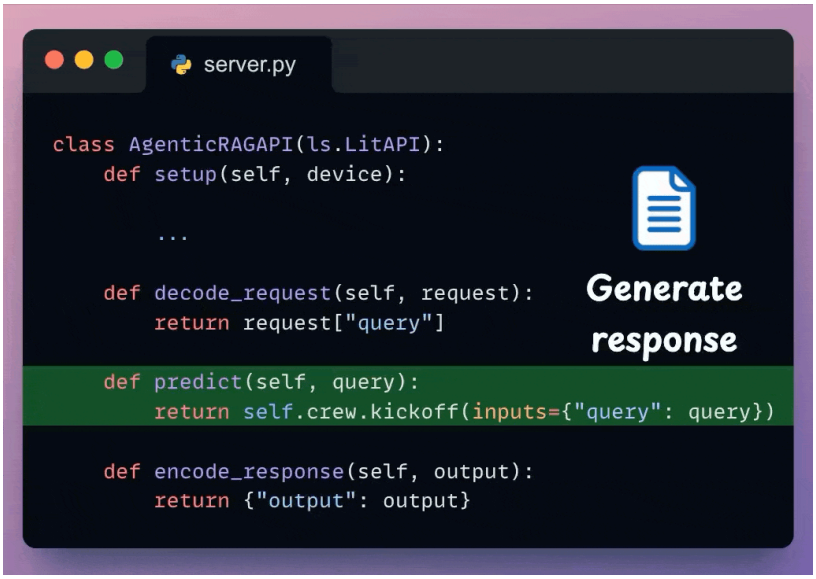
#5) Decode request

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

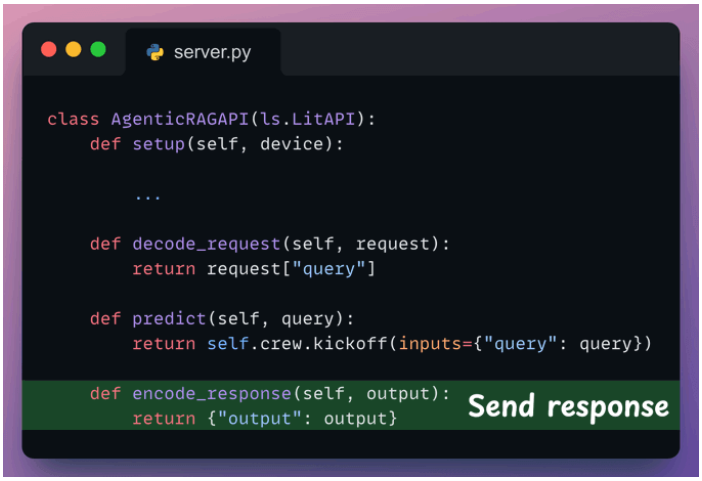
We use the decoded user query and pass it to the Crew defined earlier to generate a response from the model. Check the highlighted code below:



#7) Encode response

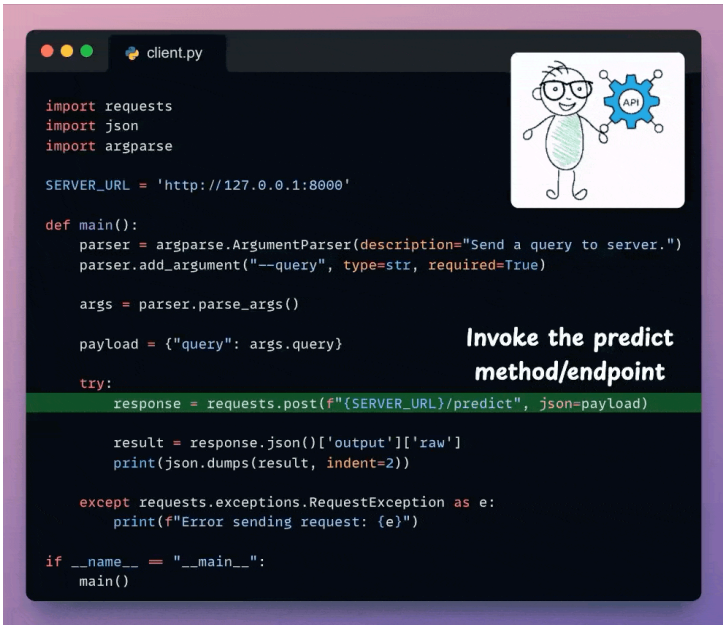
Here, we can post-process the response & send it back to the client.

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:

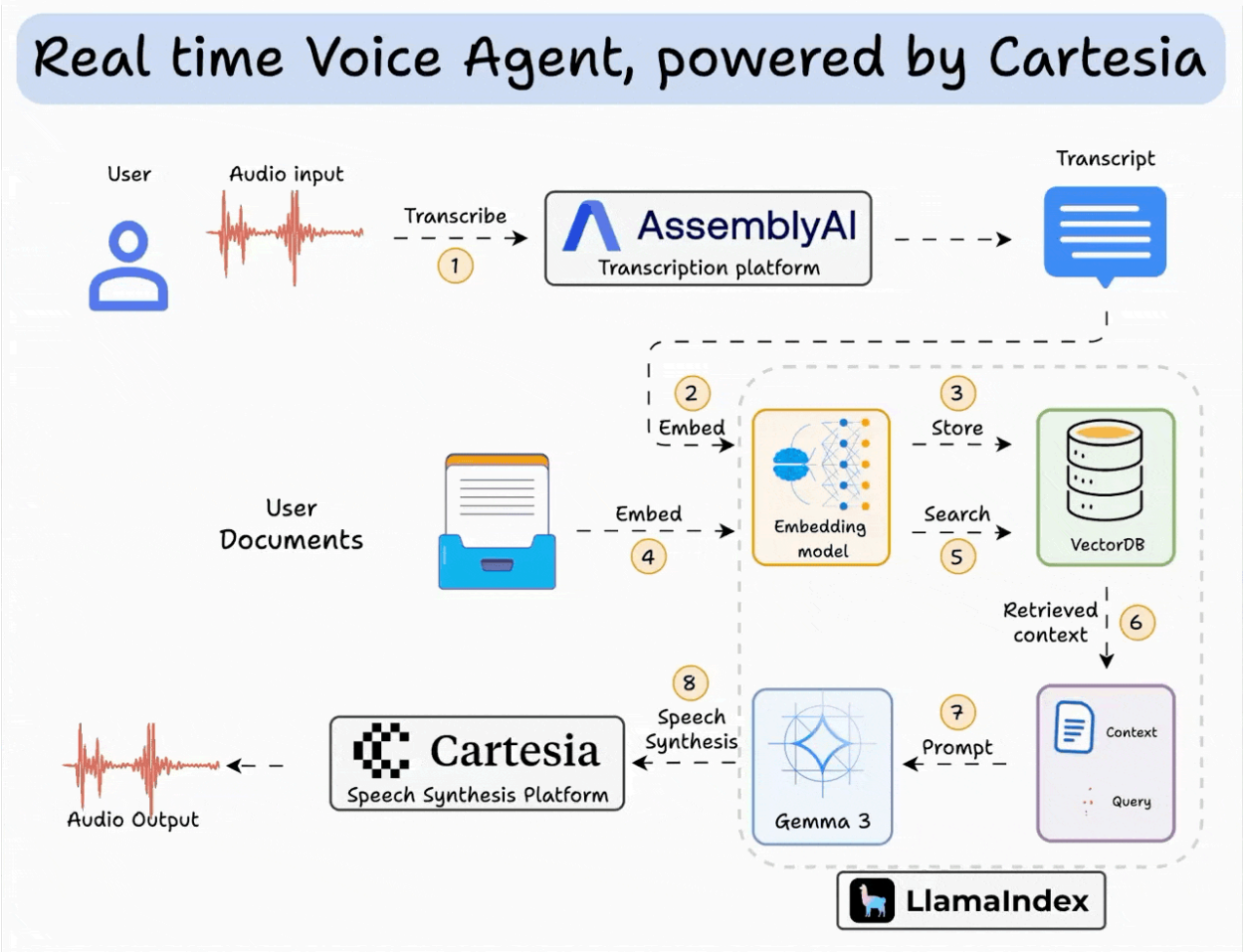


Done! We have deployed our fully private Qwen 3 Agentic RAG using LitServe.

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:

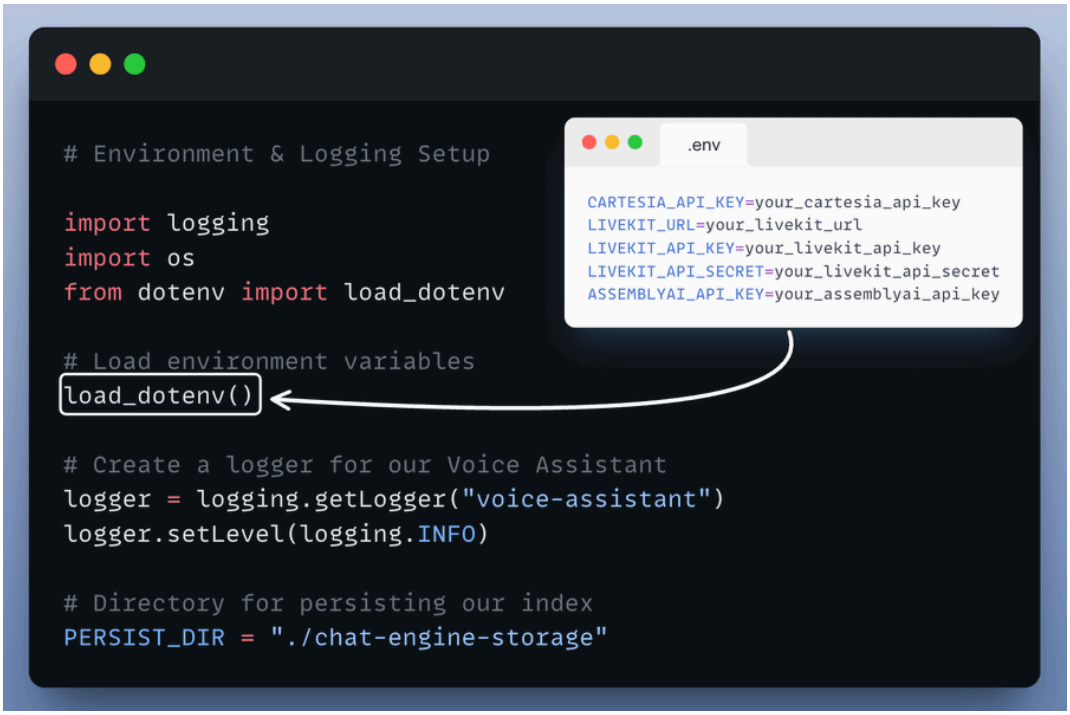
- Listens to real-time audio
- Transcribes it via AssemblyAI
- Uses your docs (via LlamaIndex) to craft an answer
- Speaks that answer back with Cartesia

Let's implement this!

#1) Set up environment and logging

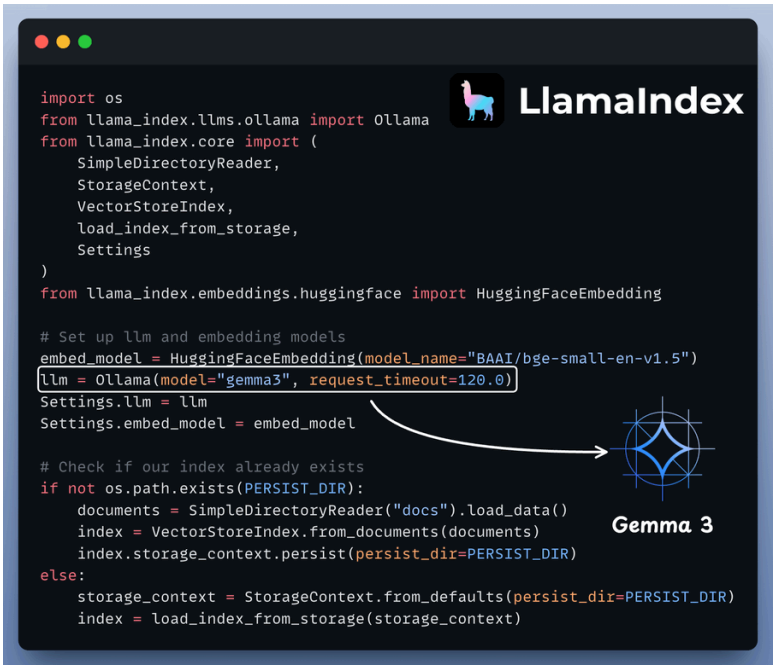
This ensures we can load configurations from .env and keep track of everything in real time.

Check this out:



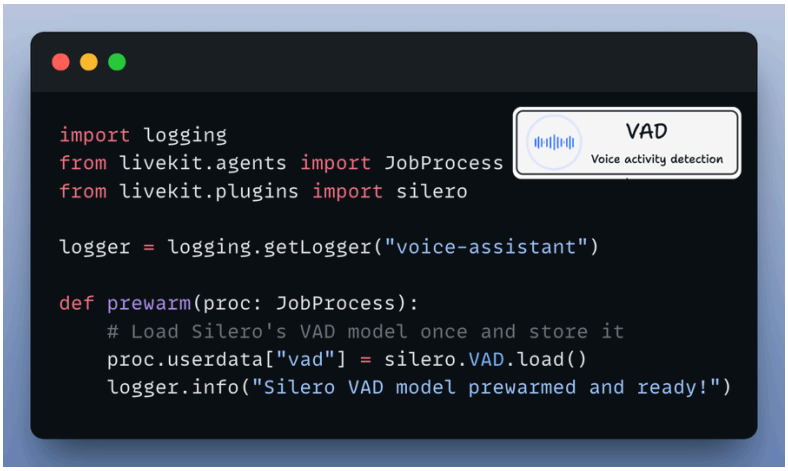
#2) Setup RAG

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:

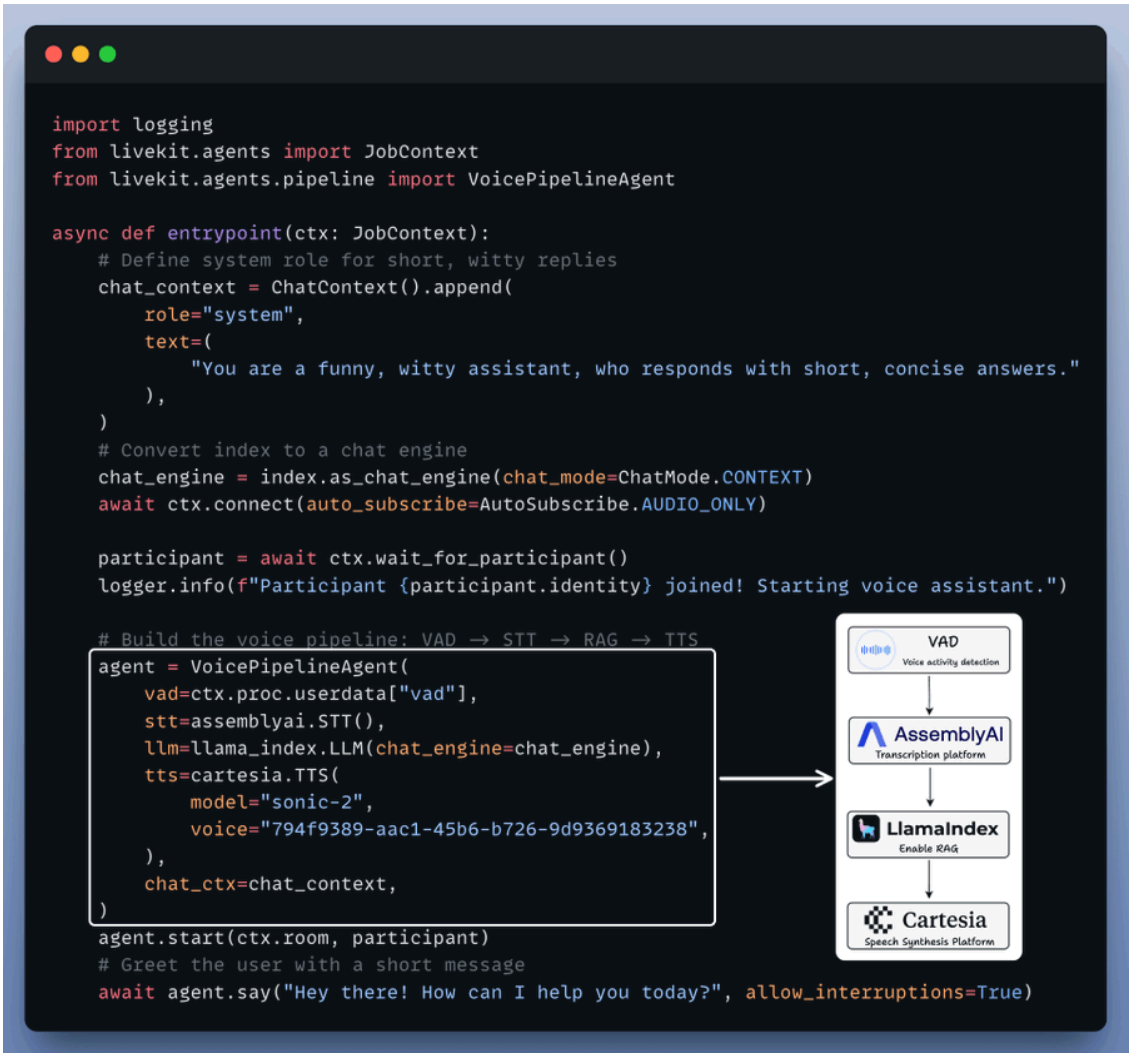


#4) The VoicePipelineAgent and Entry Point

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

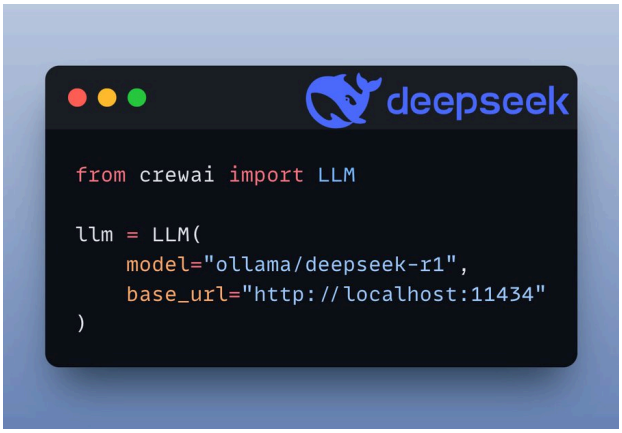
Workflow:

- 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!

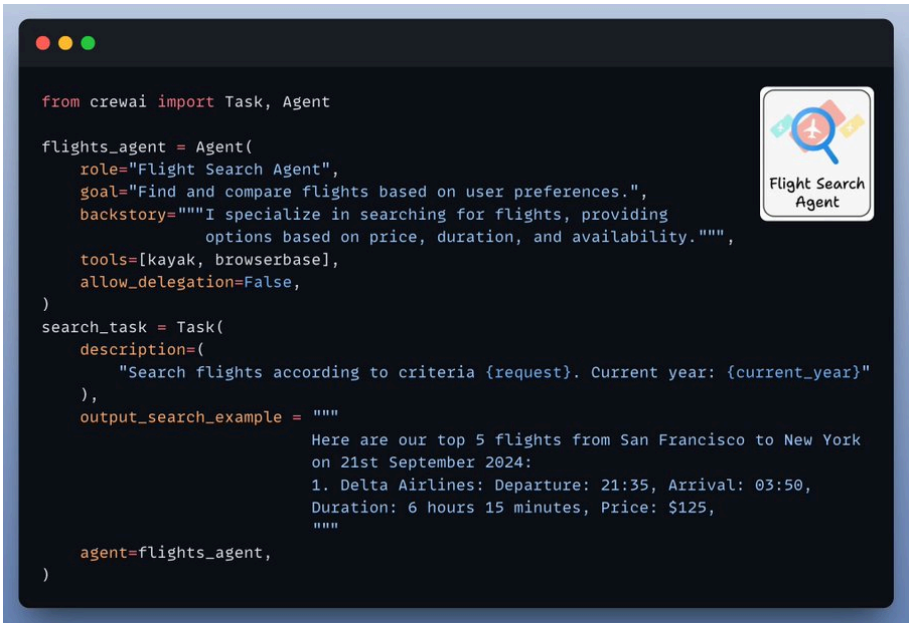
#1) Define LLM

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

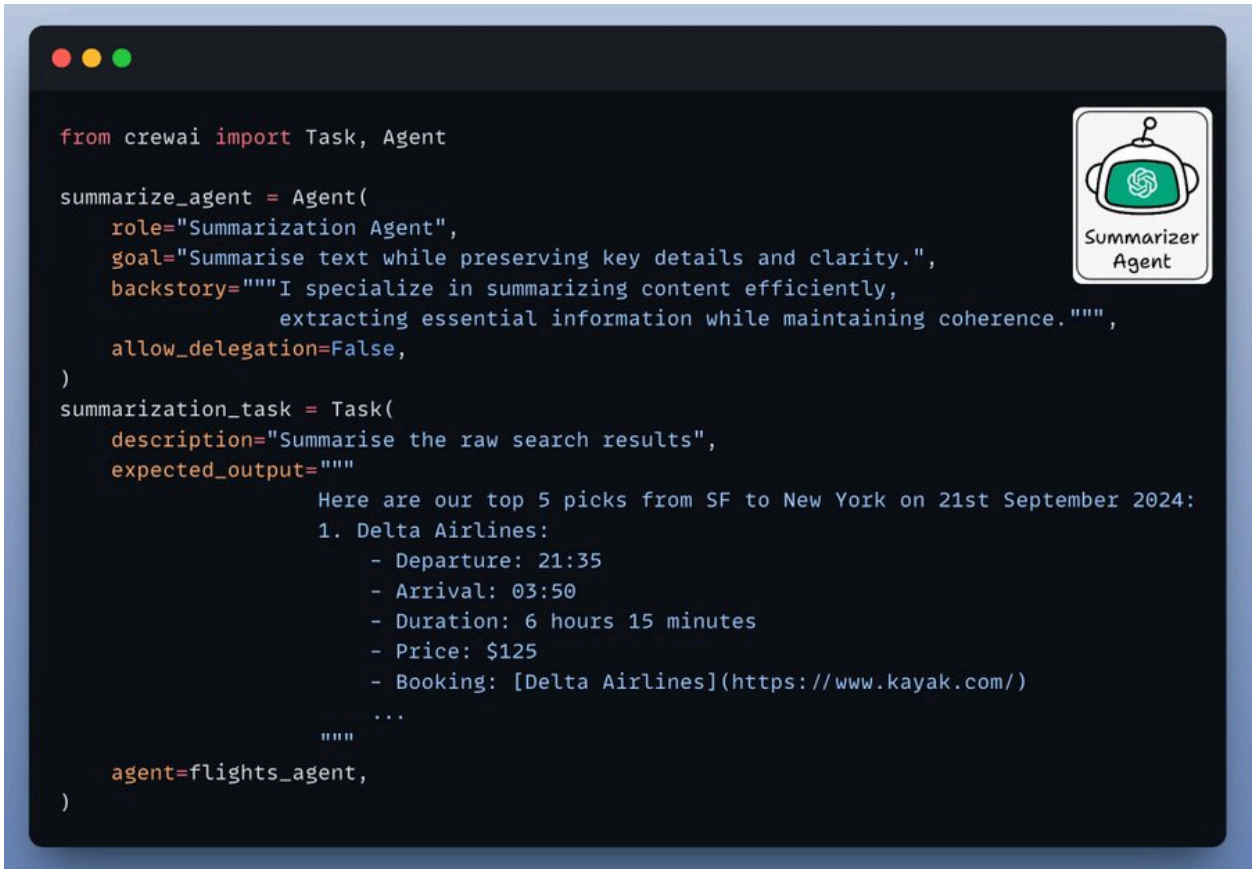
This agent mimics a real human searching for flights by browsing the web, powered by Browserbase's headless-browser tool and can look up flights on sites like Kayak.



#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.

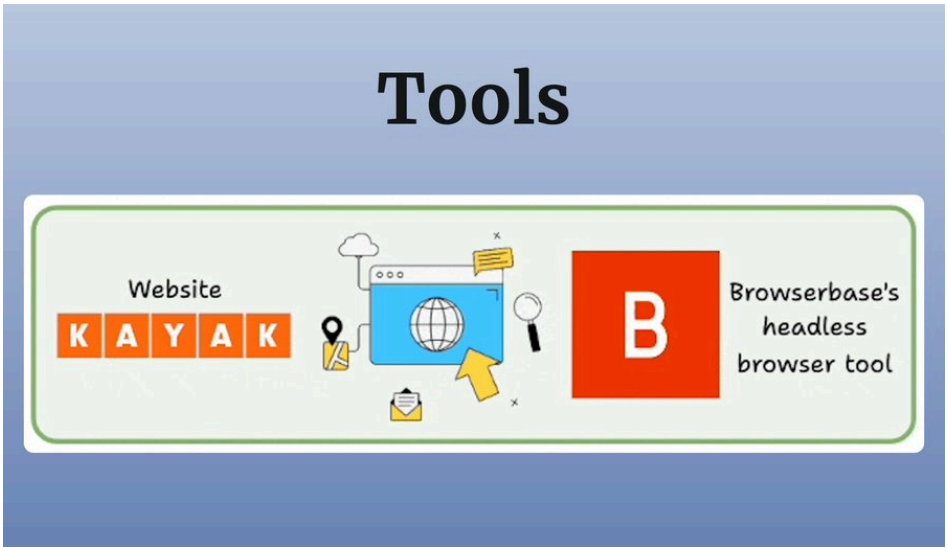



```
from crewai import Task, Agent

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
```