- The right info
- The right tools
- In the right format

This ensures the LLM can effectively complete the task.

#1) Crew flow

We'll follow a top-down approach to understand the code.

Here's an outline of what our flow looks like:

Note that this is one of many blueprints to implement a context engineering workflow. Your pipeline will likely vary based on the use case.

#2) Prepare data for RAG

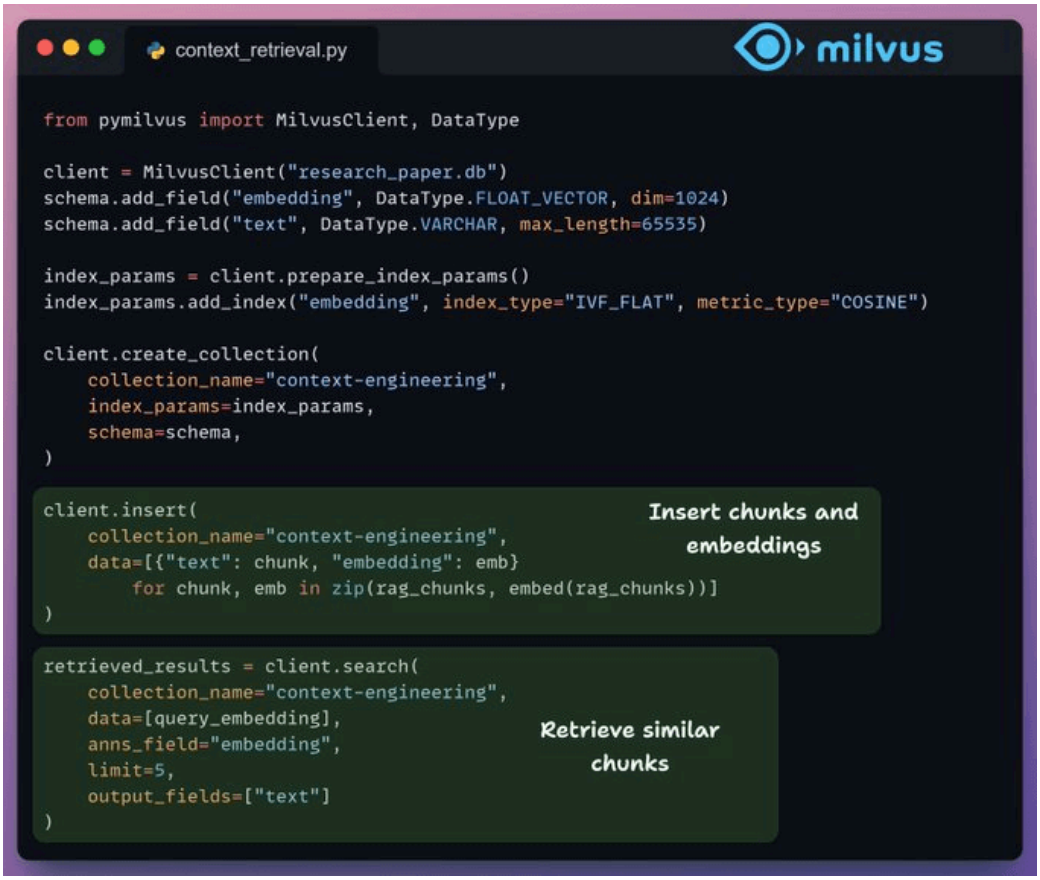
We use Tensorlake to convert the document into RAG-ready markdown chunks for each section.

The extracted data can be directly embedded and stored in a vector DB without further processing.

#3) Indexing and retrieval

Now that we have RAG-ready chunks along with the metadata, it's time to store them in a self-hosted Milvus vector database.

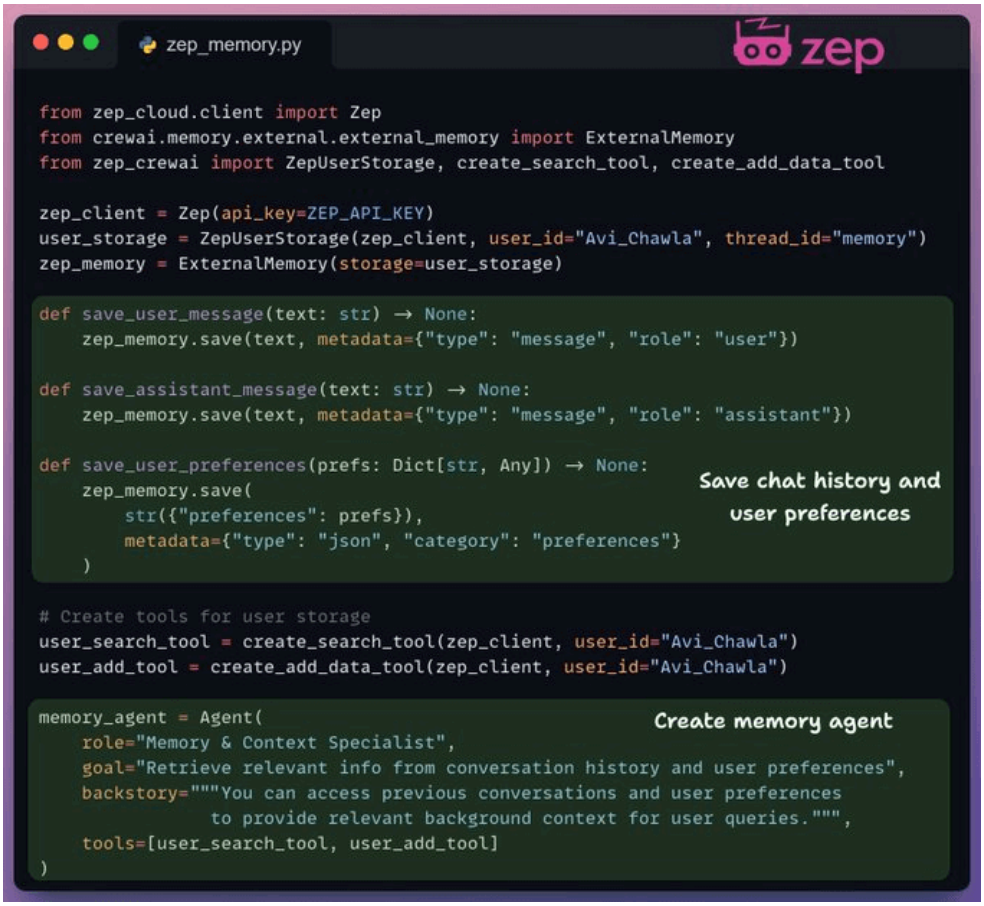
We retrieve the top-k most similar chunks to our query:



#4) Build memory layer

Zep acts as the core memory layer of our workflow. It creates temporal knowledge graphs to organize and retrieve context for each interaction.

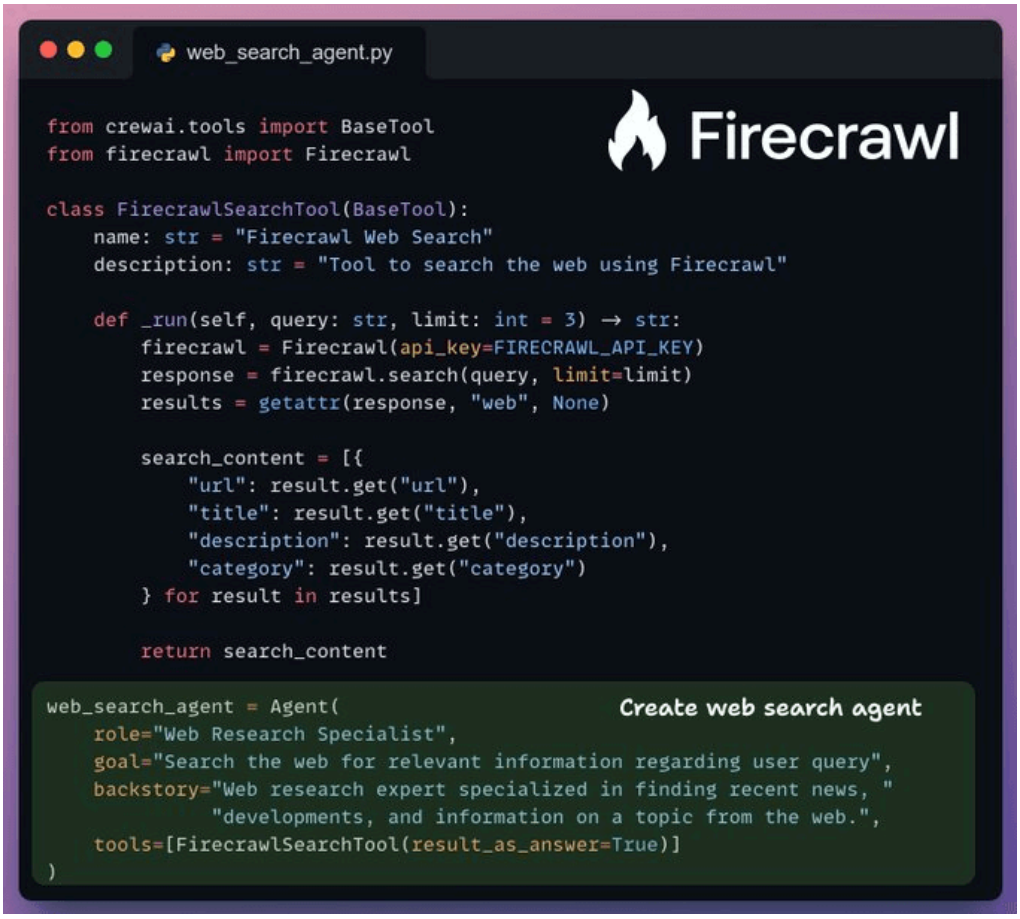
We use it to store and retrieve context from chat history and user data.



#5) Firecrawl web search

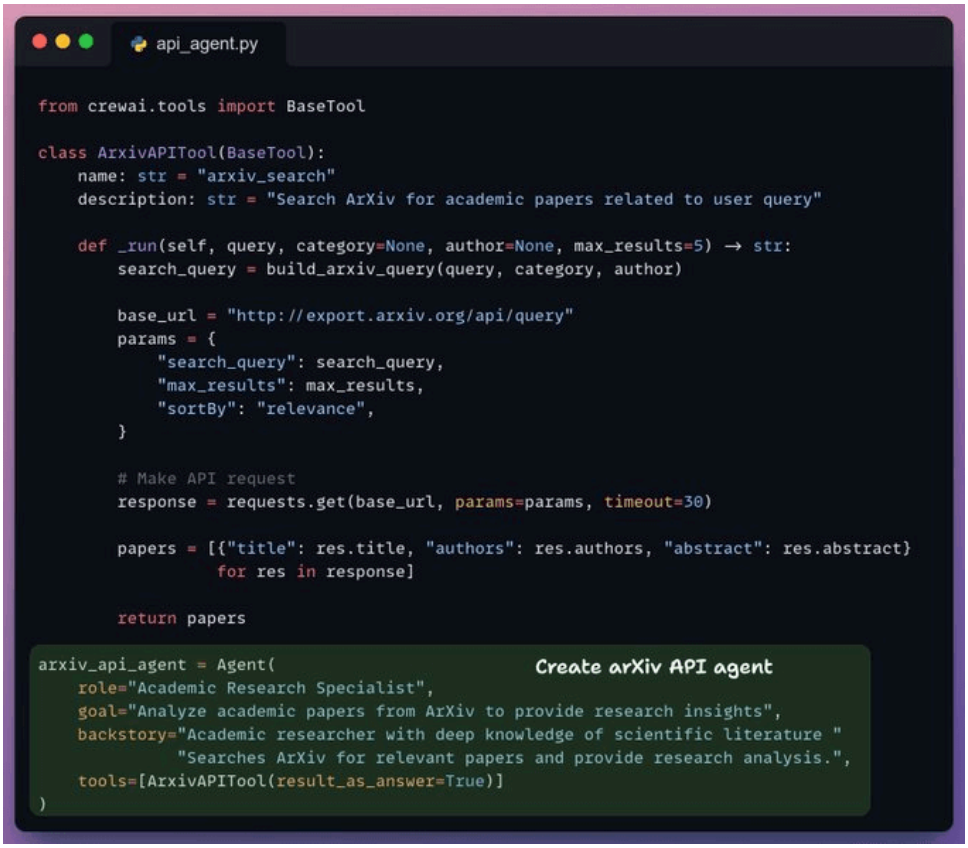
We use Firecrawl web search to fetch the latest news and developments related to the user query.

Firecrawl's v2 endpoint provides 10x faster scraping, semantic crawling, and image search, turning any website into LLM-ready data.



#6) ArXiv API search

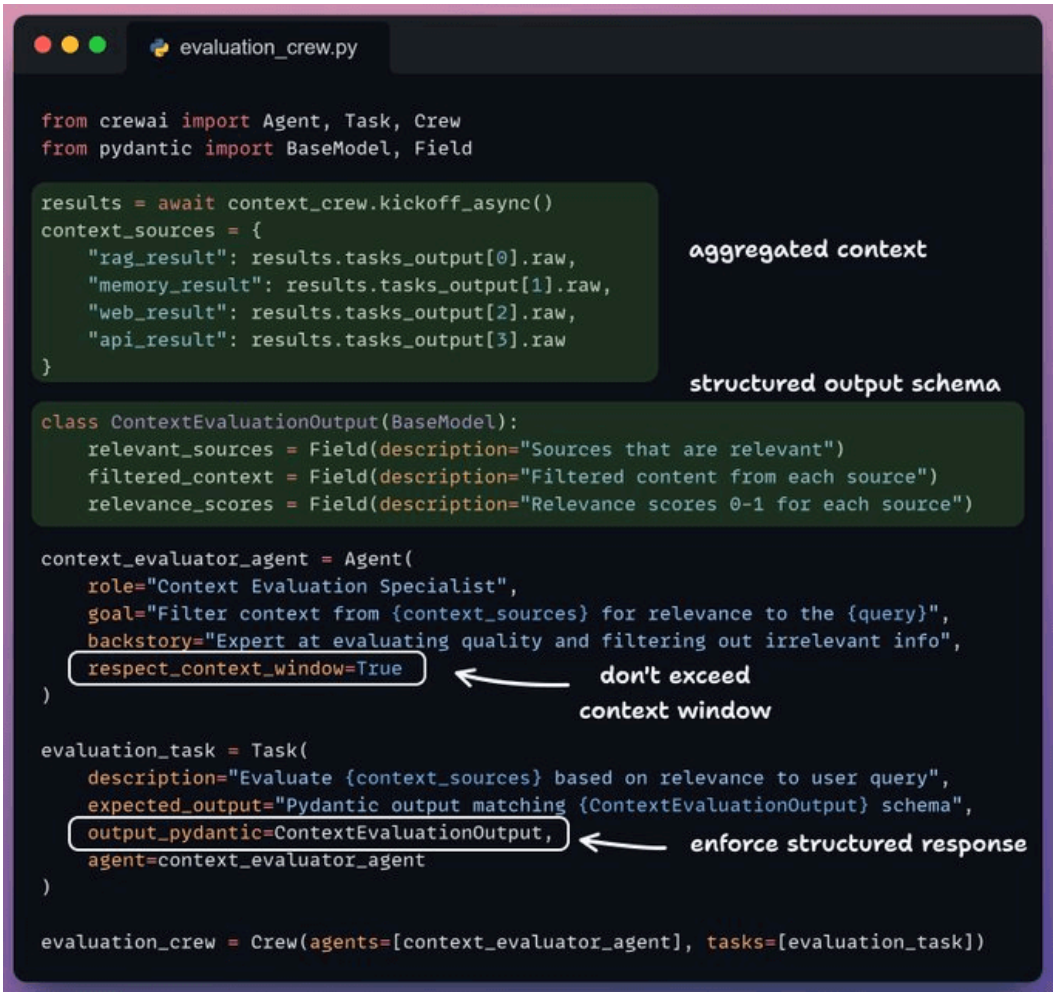
To further support research queries, we use the arXiv API to retrieve relevant results from their data repository based on the user query.



#7) Filter context

Now, we pass our combined context to the context evaluation agent that filters out irrelevant context.

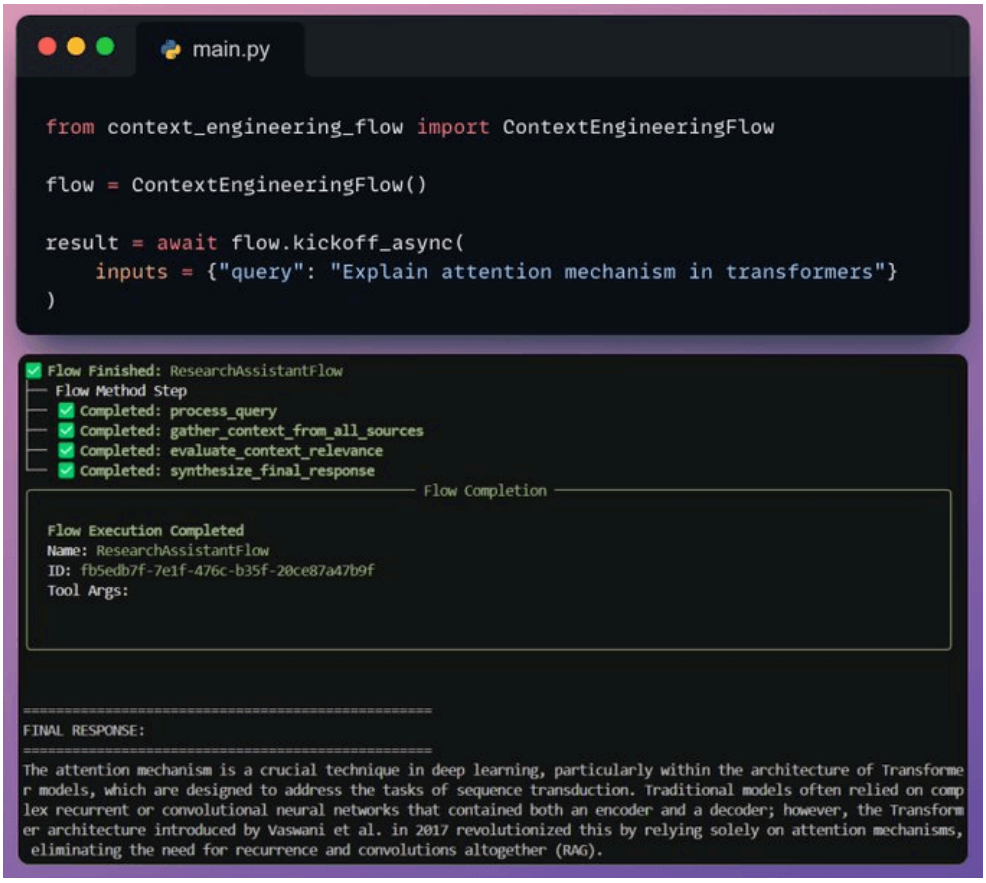
This filtered context is then passed to the synthesizer agent that generates the final response.



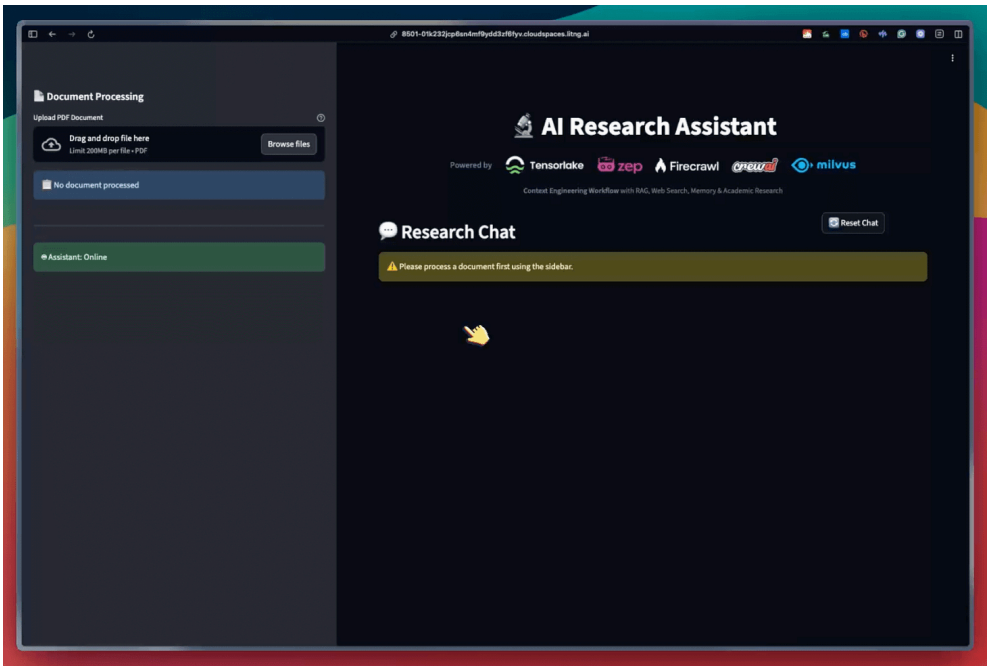
#8) Kick off the workflow

Finally, we kick off our context engineering workflow with a query.

Based on the query, we notice that the RAG tool, powered by Tensorlake, was the most relevant source for the LLM to generate a response.



We also translated this workflow into a streamlit app that:



- Provides citations with links and metadata.
- Provides insights into relevant sources.

The workflow explained above is one of the many blueprints. Your implementation can vary.

Context Engineering in Claude Skills

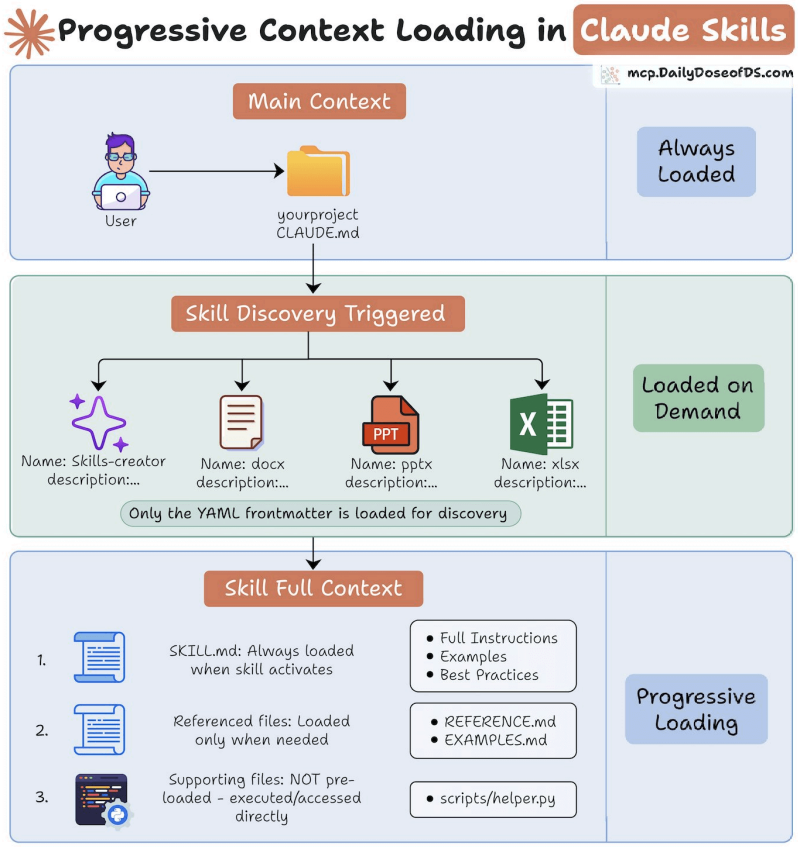
Claude Skills are Anthropic's mechanism for giving agents reusable, persistent abilities without overloading the model's context window.

They solve a practical issue in agent design: LLMs forget everything unless all instructions, examples and edge cases are restated each time.

Skills package this information into small, self-contained units that Claude loads only when they're relevant.

This allows an agent to use hundreds of specialized workflows while keeping its active context lightweight.

To make this scalable, Skills use a three-layer context management system that lets it use 100s of skills without hitting context limits.



Let's understand how it works:

- Layer 1: Main Context - Always loaded, it contains the project configuration.
- Layer 2: Skill Metadata - Comprises only the YAML frontmatter, about 2-3 lines (< 200 tokens).
- Layer 3: Active Skill Context - SKILL.md files and associated documentation are loaded as needed.

Supporting files like scripts and templates aren't pre-loaded but accessed directly when in use, consuming zero tokens.

This architecture supports hundreds of skills without breaching context limits.

Now let's zoom into the main ideas behind Skills, because understanding what they are clarifies why this 3-layer system matters.

Skills as SOPs for Agents

Think of a Skill as a packaged procedure - a complete, reusable workflow that teaches the agent how to perform a task with consistency.

Instead of re-explaining steps, examples, constraints, and edge cases every time, you define the workflow once and reuse it forever.

It's the AI equivalent of an operating manual: structured, repeatable, and self-contained.

Anatomy of a Skill

A skill is simply a folder, but what it contains is carefully designed:

- A skill.md file with two layers of context:
  - YAML Front Matter: a tiny descriptor Claude uses to decide when the skill is relevant.
  - Skill Body: the detailed instructions, workflows, examples, and guidance used during execution.
- Optional supporting files such as scripts, templates or reference docs. These aren't loaded into context, they're fetched only when the agent needs them.

This separation lets Claude stay lightweight until a specific skill is activated.

How Skills Fit Into the Agent Architecture

Skills don't replace Projects, Subagents or MCP - they complement them:

- Projects organize your workspace.
- MCP connects Claude to tools and external services.
- Subagents handle delegated reasoning.
- Skills package the reusable expertise that all of them can rely on.

Each solves a different layer of the agent problem, and skills serve as the procedural knowledge base.

Building Your Own Skills

The creation process is straightforward:

1. Identify a workflow you repeat constantly.
2. Create a skill folder and add a skill.md file.
3. Write the YAML front matter + full markdown instructions.
4. Add any scripts, examples, or supporting resources.
5. Zip the folder and upload it in Claude's capabilities.

Claude Desktop even includes a "Skill Creator" skill that helps generate the structure for you.

Manual RAG Pipeline vs Agentic Context Engineering

Imagine you have data that's spread across several sources (Gmail, Drive, etc.).

How would you build a unified query engine over it?

Devs would typically treat context retrieval like a weekend project.

...and their approach would be: "Embed the data, store in a vector DB and do RAG."

This works beautifully for static sources.

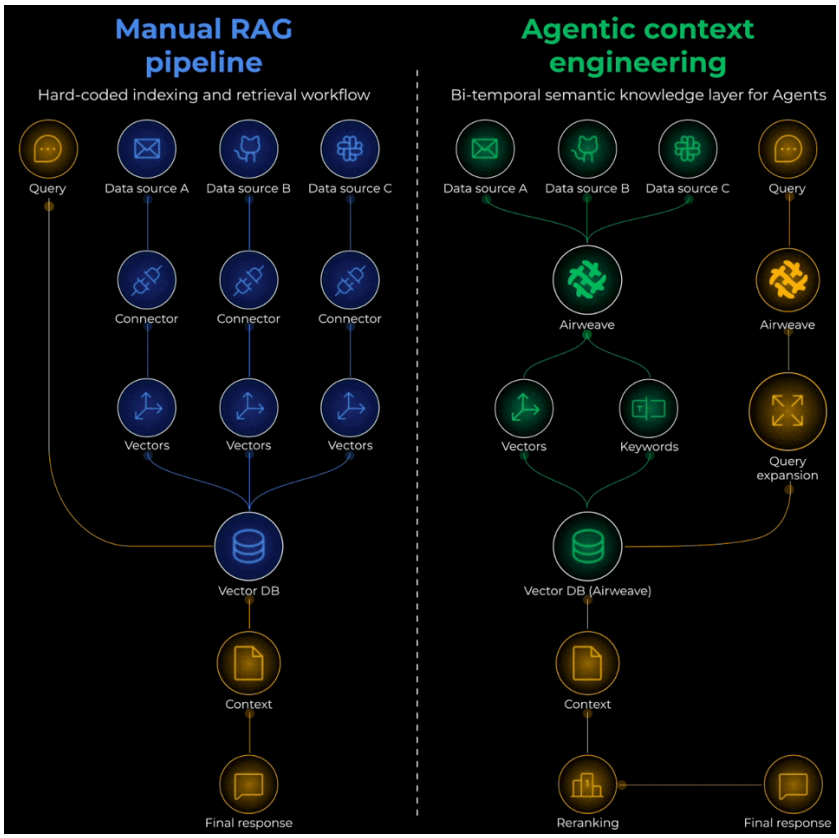
But the problem is that no real-world workflow looks like this.

To understand better, consider this query:

What's blocking the Chicago office project, and when's our next meeting about it?

Answering this single query requires searching across sources like Linear (for blockers), Calendar (for meetings), Gmail (for emails), and Slack (for discussions).

No naive RAG setup can handle this!



To actually solve this problem, you'd need to think of it as building an Agentic context retrieval system with three critical layers:

- Ingestion layer:
    - Connect to apps without auth headaches.
    - Process different data sources properly before embedding (email vs code vs calendar).
    - Detect if a source is updated and refresh embeddings (ideally, without a full refresh).

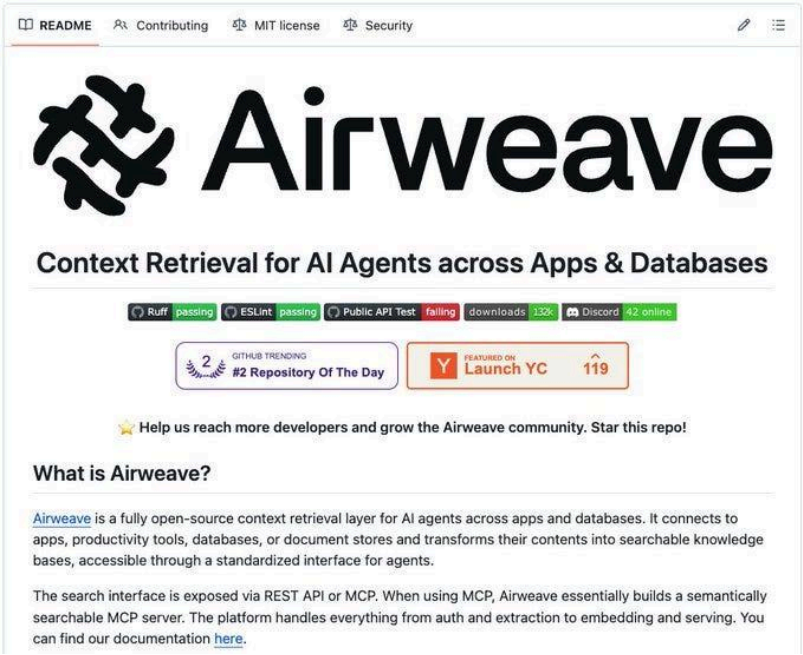
- Retrieval layer:
    - Expand vague queries to infer what users actually want.
    - Direct queries to the correct data sources.
    - Layer multiple search strategies like semantic-based, keyword-based, and graph-based.
    - Ensure retrieving only what users are authorized to see.
    - Weigh old vs. new retrieved info (recent data matters more, but old context still counts).
- Generation layer:
    - Provide a citation-backed LLM response.

That's months of engineering before your first query works.

It's definitely a tough problem to solve...

...but this is precisely how giants like Google (in Vertex AI Search), Microsoft (in M365 products), AWS (in Amazon Q Business), etc., are solving it.

If you want to see it in practice, this approach is actually implemented in Airweave, a recently trending 100% open-source framework that provides the context retrieval layer for AI agents across 30+ apps and databases(as of 3 Dec,2025).



It implements everything we discussed above, like:

- How to handle authentication across apps.
- How to process different data sources.
- How to gather info from multiple tools.
- How to weigh old vs. new info.
- How to detect updates and do real-time sync.
- How to generate perplexity-like citation-backed responses, and more.

For instance, to detect updates and initiate a re-sync, one might do timestamp comparisons.

But this does not tell if the content actually changed (maybe only the permission was updated), and you might still re-embed everything unnecessarily.

Airweave handles this by implementing source-specific hashing techniques like entity-level hashing, file content hashing, cursor-based syncing, etc.

You can see the full implementation on GitHub and try it yourself.

But the core insight applies regardless of the framework you use:

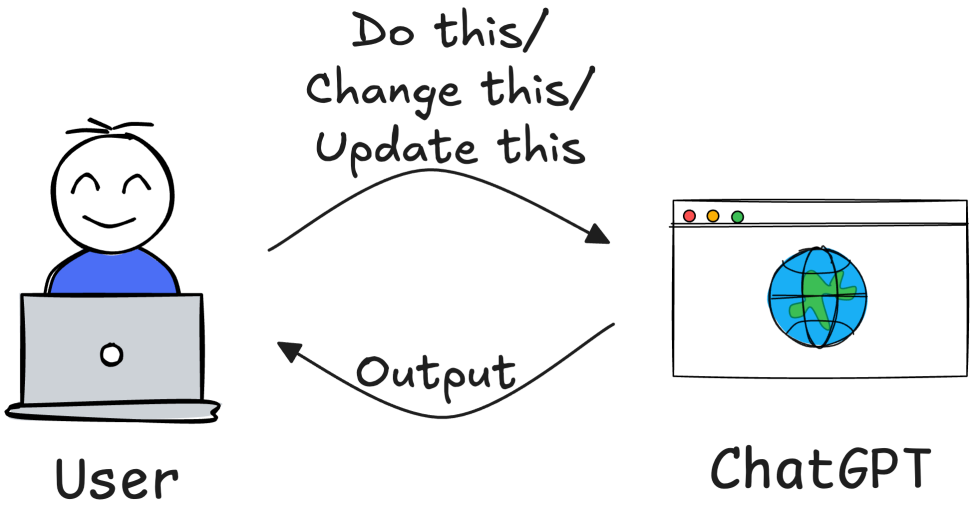
Context retrieval for Agents is an infrastructure problem, not an embedding problem.

You need to build for continuous sync, intelligent chunking, and hybrid search from day one.

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:

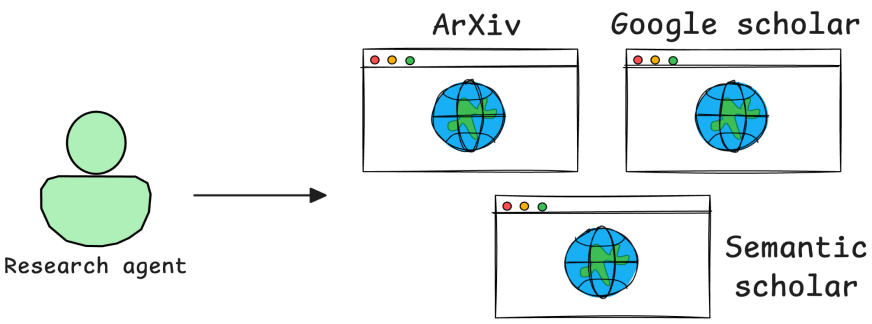


1. Ask for a summary of recent AI research papers.
2. Review the response and realize you need sources.
3. Obtain a list of papers along with citations.
4. Find that some sources are outdated, so you refine your query.
5. Finally, after multiple iterations, you get a useful output.

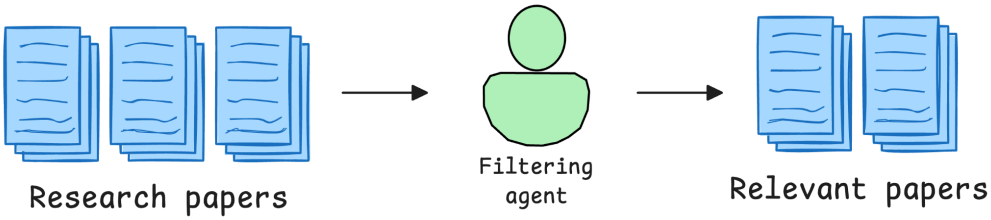
This iterative process takes time and effort, requiring you to act as the decision-maker at every step.

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

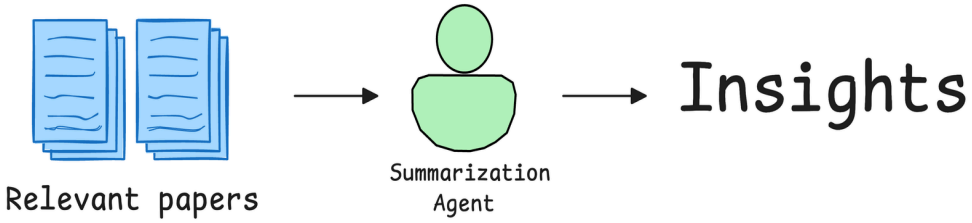
A Research Agent autonomously searches and retrieves relevant AI research papers from arXiv, Semantic Scholar, or Google Scholar.



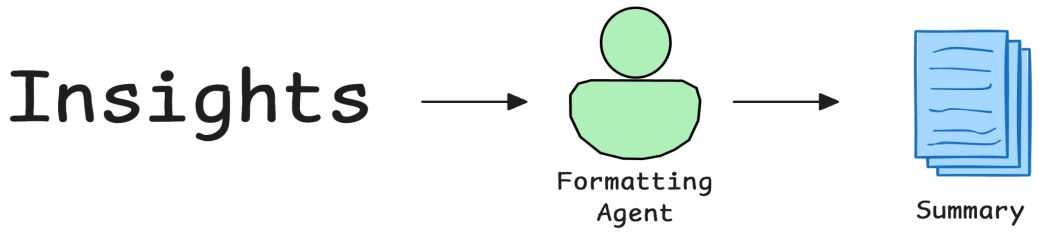
- A Filtering Agent scans the retrieved papers, identifying the most relevant ones based on citation count, publication date, and keywords.



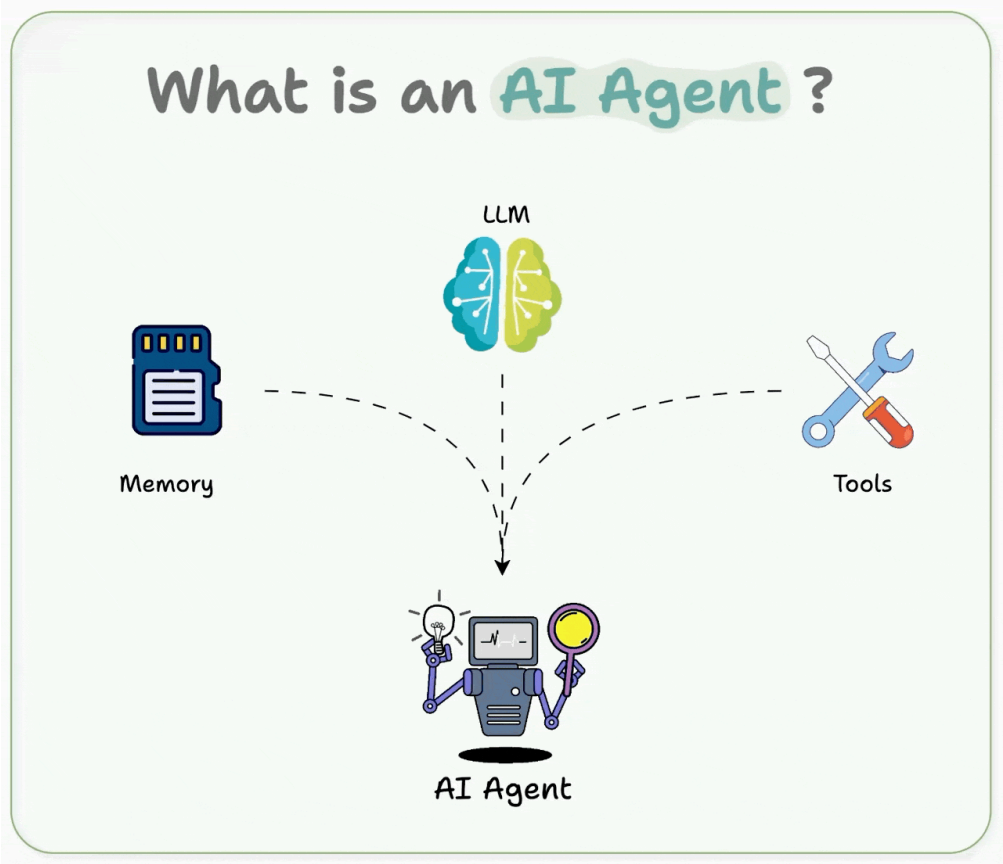
- A Summarization Agent extracts key insights and condenses them into an easy-to-read report.



- 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

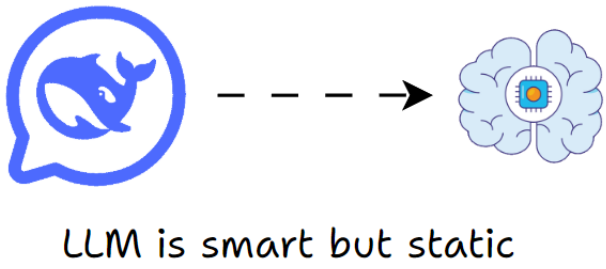
Let's break it down with a simple analogy:

- 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.

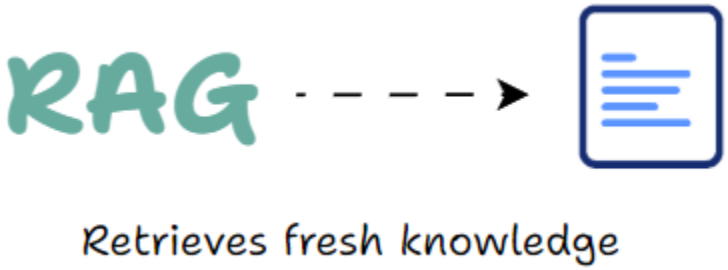
It can reason, generate, summarize but only using what it already knows (i.e., its training data).



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

RAG (Retrieval-Augmented Generation)

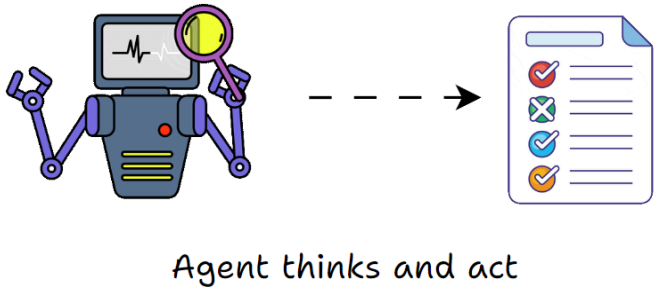
RAG enhances an LLM by retrieving external documents (from a vector DB, search engine, etc.) and feeding them into the LLM as context before generating a response.



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,

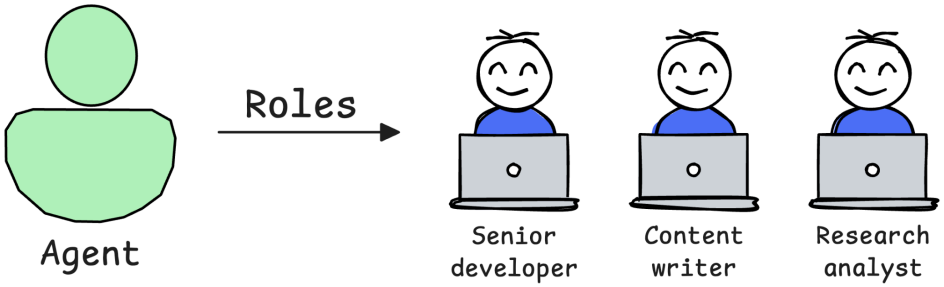
to be effective, they must be built with certain key principles in mind. There are six essential building blocks that make AI agents more reliable, intelligent, and useful in real-world applications:

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.

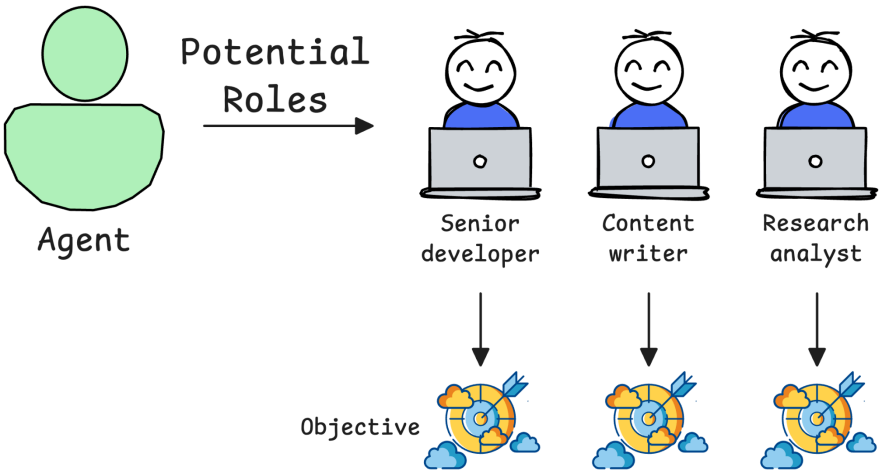
Why?

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.

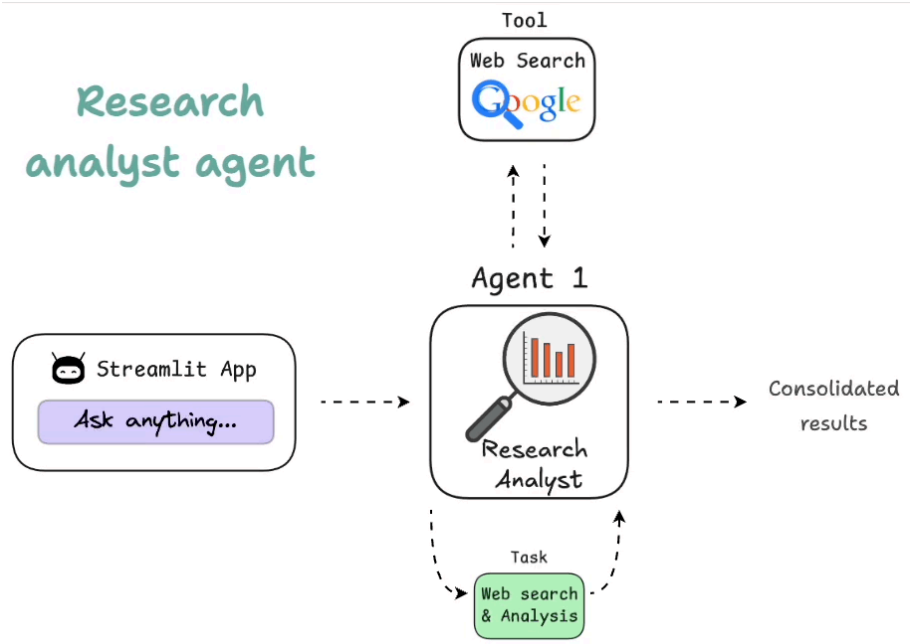
For example, a marketing agent should stick to messaging, tone, and audience not pricing or market analysis.

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.

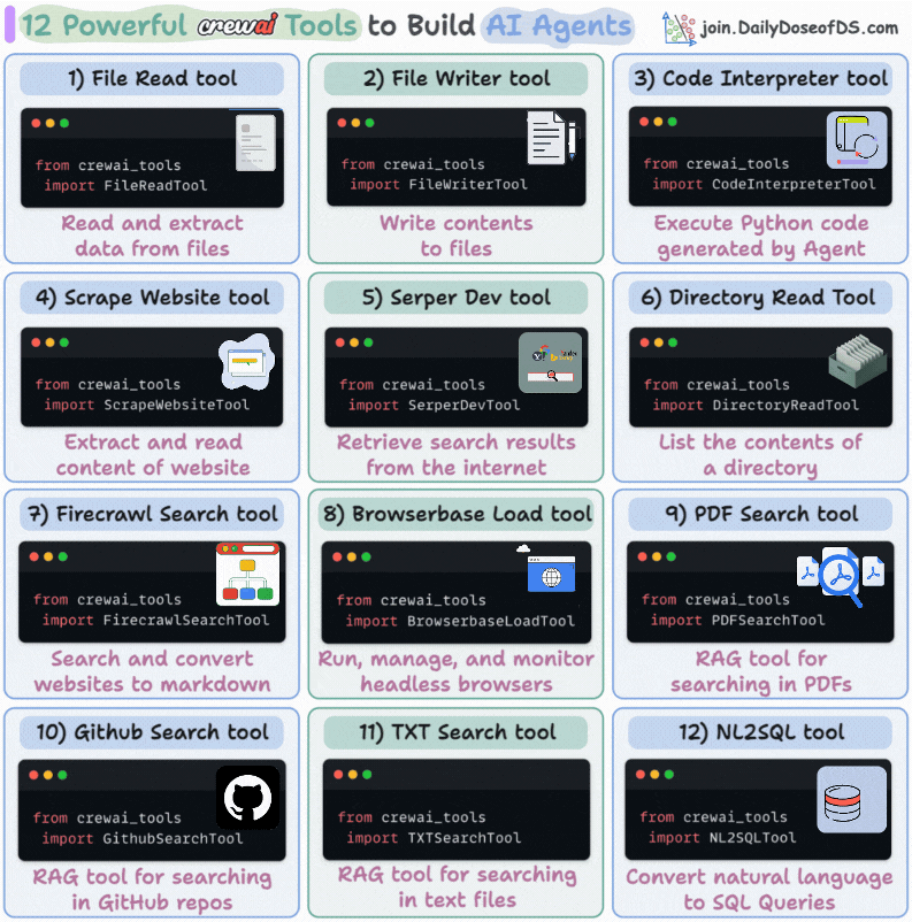
#3.1) Custom tools

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:

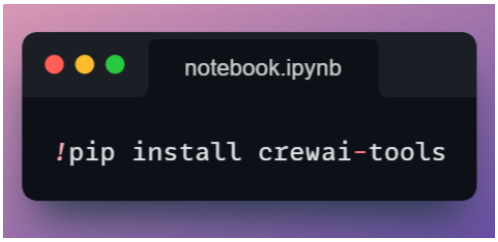


However, you may need to build custom tools at times.

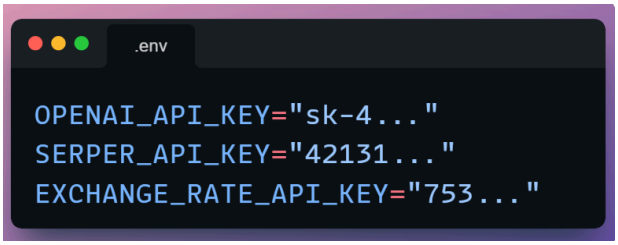
In this example, we're building a real-time currency conversion tool inside CrewAI. Instead of making an LLM guess exchange rates, we integrate a custom tool that fetches live exchange rates from an external API and provides some insights.

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

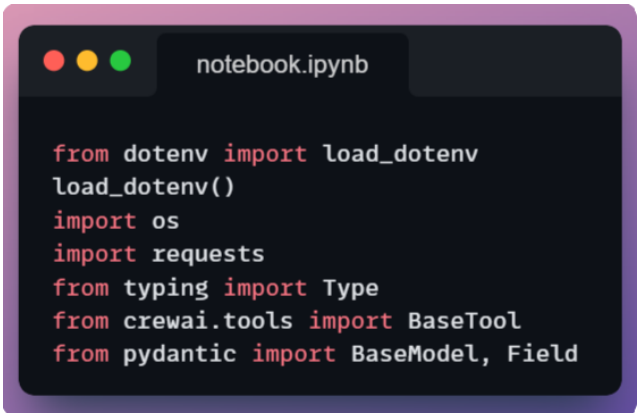
Firstly, make sure the tools package is installed:



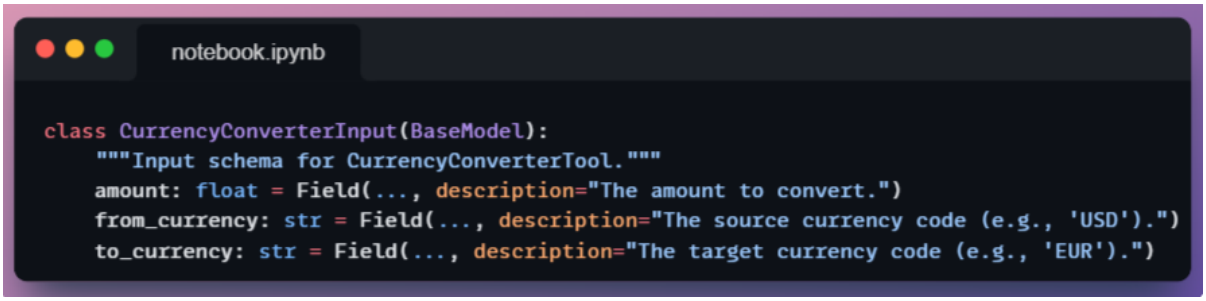
You would also need an API key from here: https://www.exchangerate-api.com/ (it's free). Specify it in the .env file as shown below:



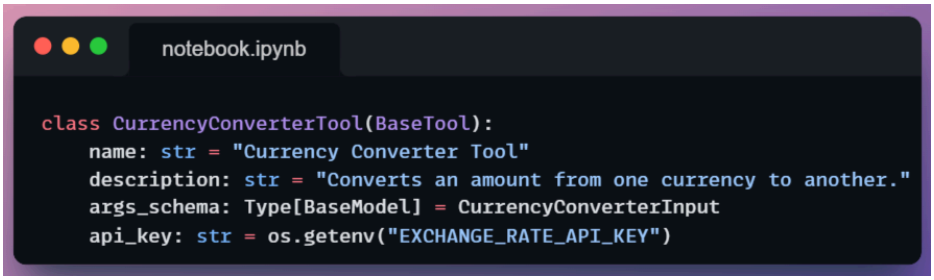
Once that's done, we start with some standard import statements:



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

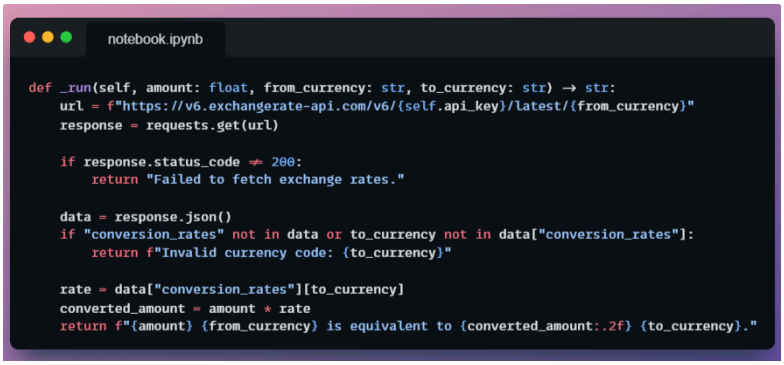


Now, we define the CurrencyConverterTool by inheriting from *BaseTool:*



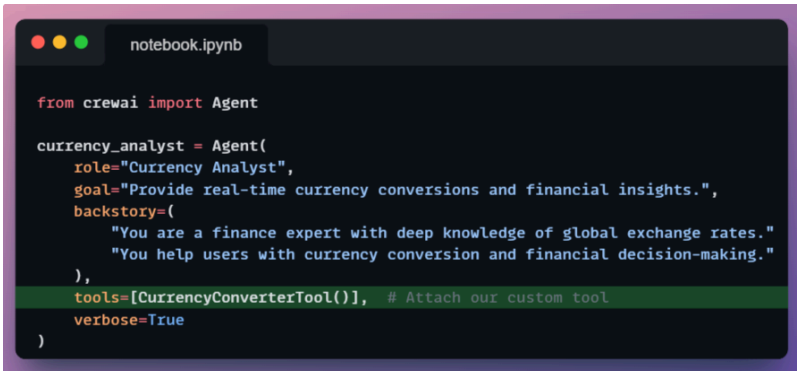
Every tool class should have the _run method which we will execute whenever the Agents wants to make use of it.

For our use case, we implement it as follows:

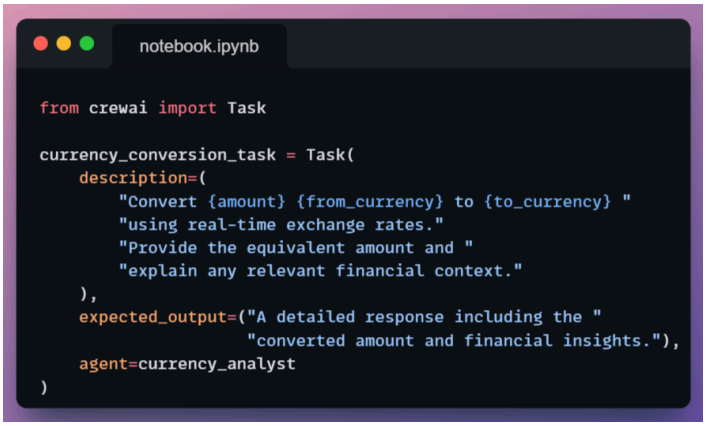


In the above code, we fetch live exchange rates using an API request. We also handle errors if the request fails or the currency code is invalid.

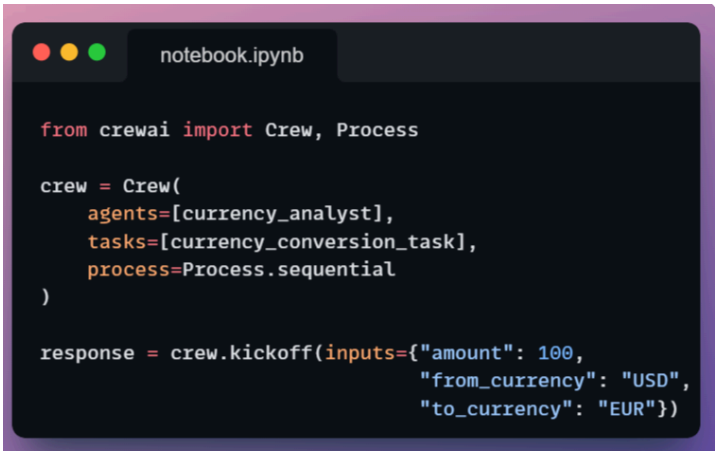
Now, we define an agent that uses the tool for real-time currency analysis and attach our CurrencyConverterTool, allowing the agent to call it directly if needed:



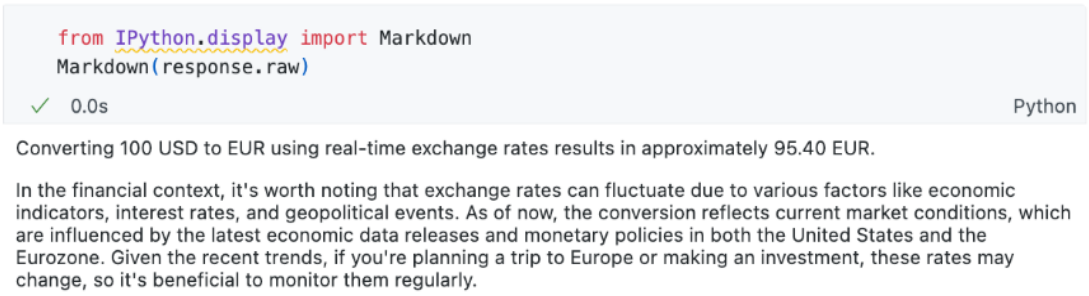
We assign a task to the currency_analyst agent.



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



Printing the response, we get the following output:



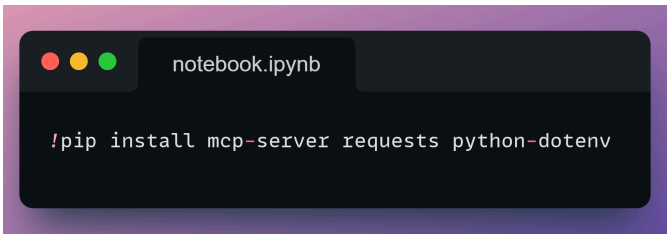
Works as expected!

#3.2) Custom tools via MCP

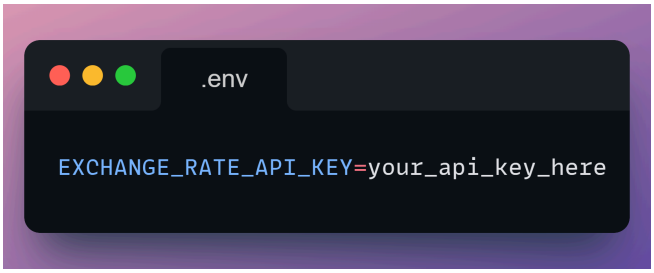
Now, let's take it a step further.

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:

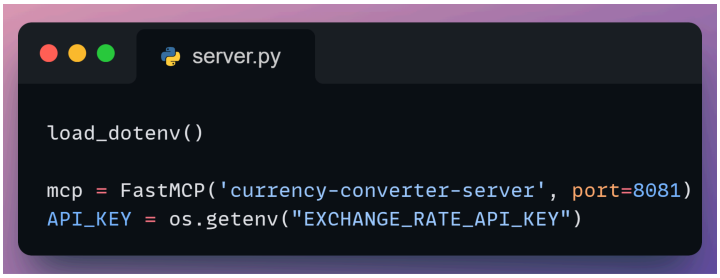


We'll continue using ExchangeRate-API in our .env file:

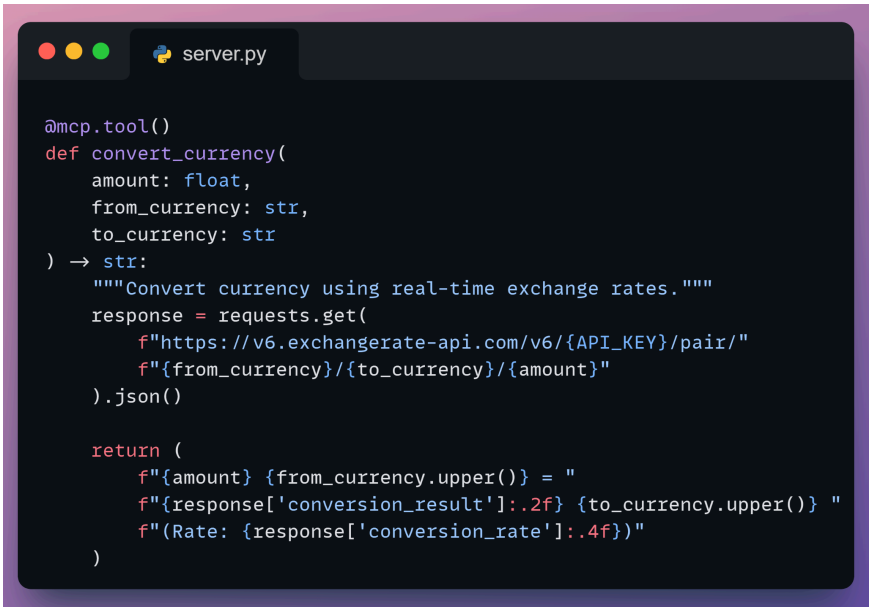


We'll now write a lightweight server.py script that exposes the currency converter tool. We start with the standard imports:

Now, we load environment variables and initialize the server:

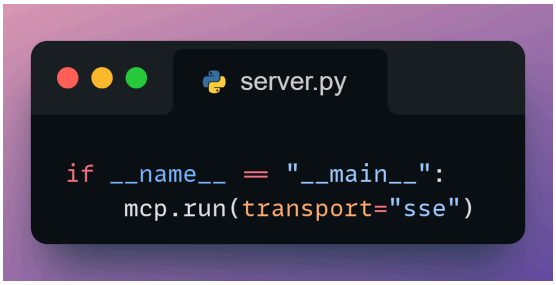


Next, we define the tool logic with @mcp.tool():



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

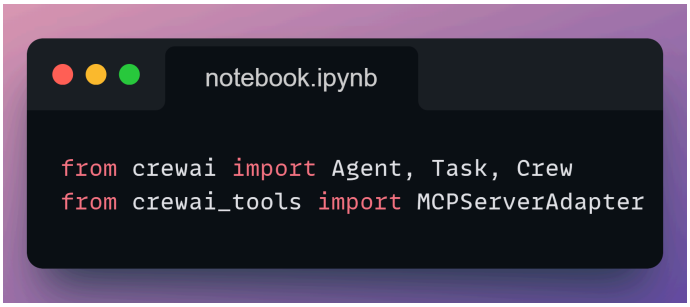
To make the tool accessible, we need to run the MCP server. Add this at the end of your script:



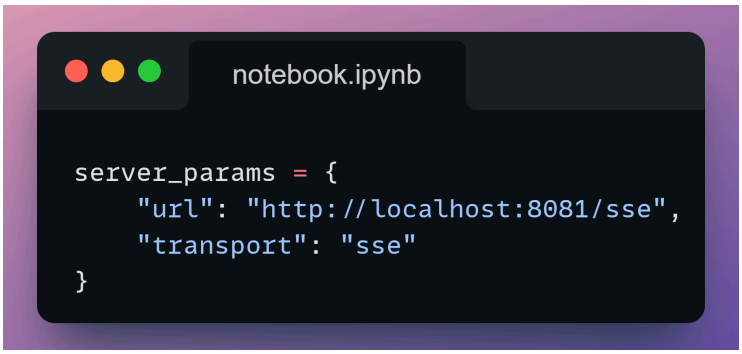
This starts the server and exposes your convert_currency tool at: http://localhost:8081/sse.

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.

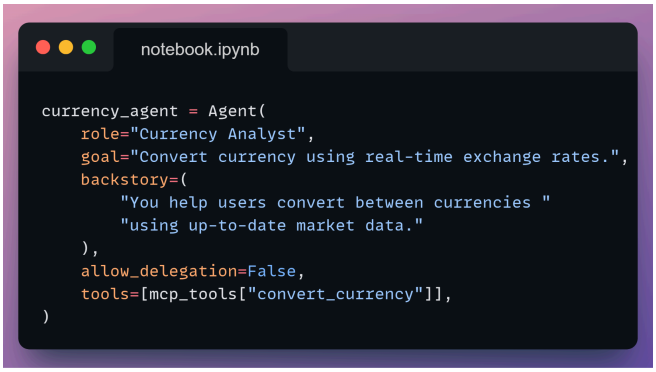


Next, we connect to the MCP tool server. Define the server parameters to connect to your running tool (from server.py).

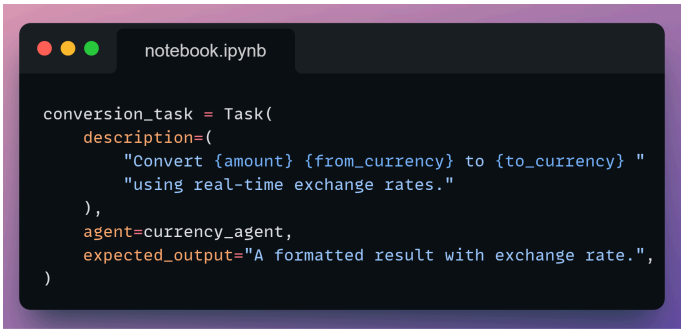


Now, we use the discovered MCP tool in an agent:

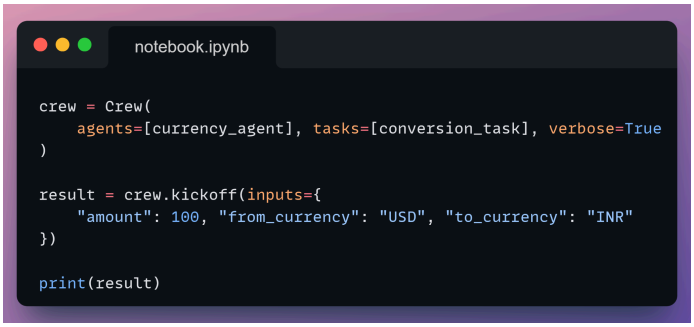
This agent is assigned the convert_currency tool from the remote server. It can now call the tool just like a locally defined one.



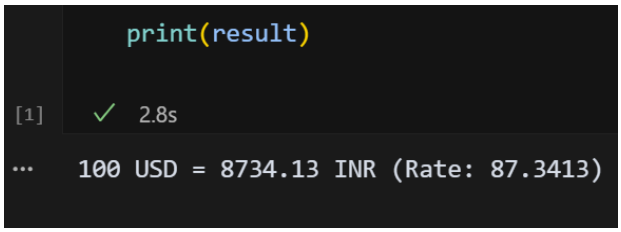
We give the agent a task description:



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



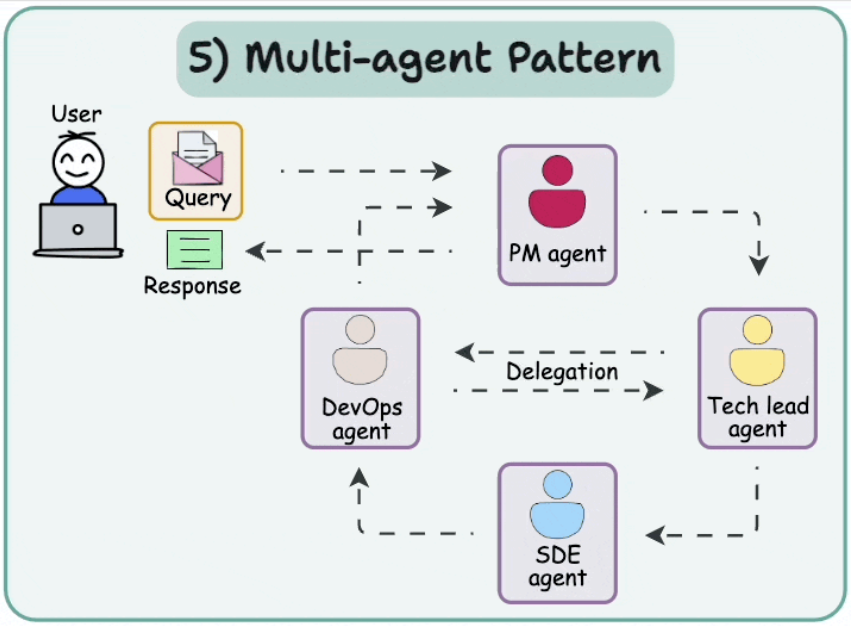
Printing the result, we get the following output:



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.

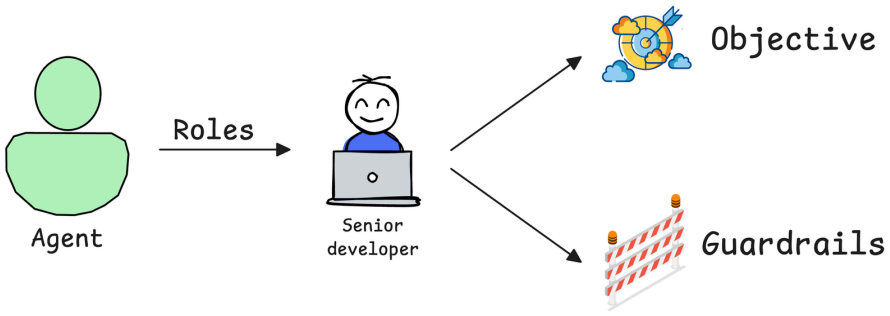
The best practice is to enable agent collaboration by designing workflows where agents can exchange insights and refine their responses together.

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:

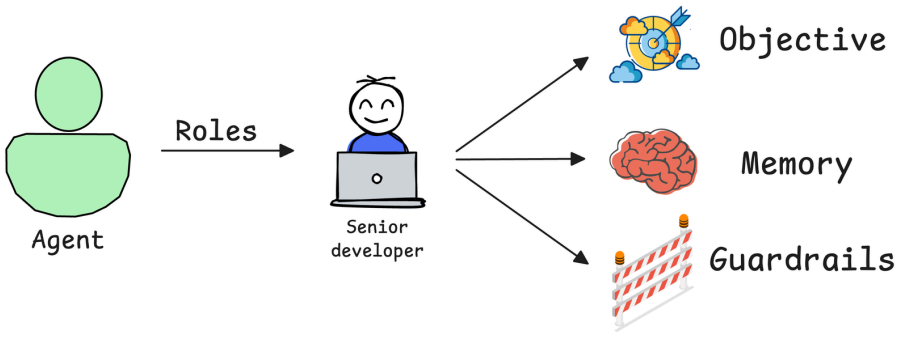
- Limiting tool usage: Prevent an agent from overusing APIs or generating irrelevant queries.
- Setting validation checkpoints: Ensure outputs meet predefined criteria before moving to the next step.
- Establishing fallback mechanisms: If an agent fails to complete a task, another agent or human reviewer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

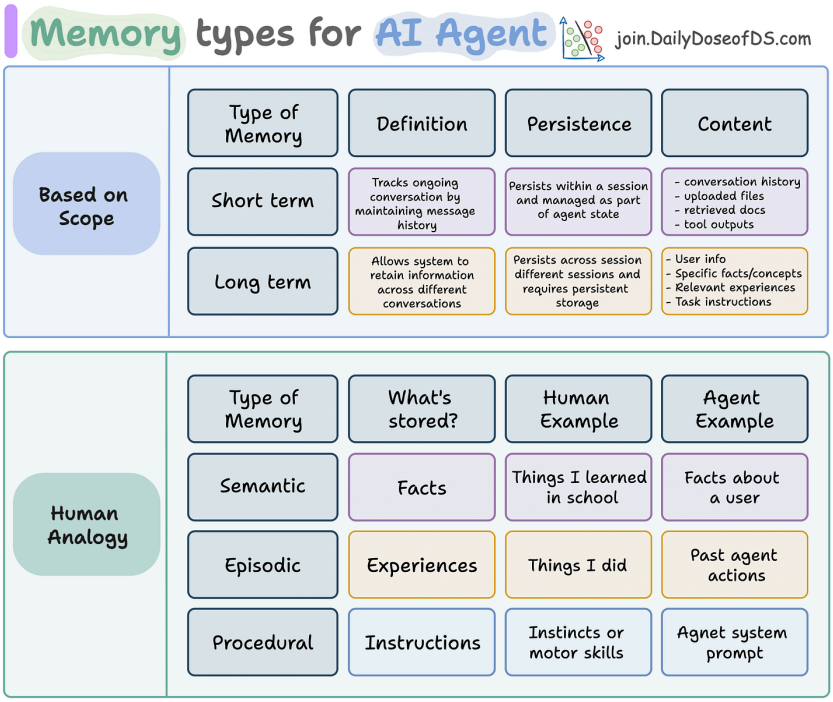
For example, in an AI-powered tutoring system, memory allows the agent to recall past lessons, tailor feedback, and avoid repetition.

Memory Types in AI Agents

Now, let us look at the memory types for AI agents in more detail.

Just like humans, long-term memory in agents can be:

- Semantic → Stores facts and knowledge
- Episodic → Recalls past experiences or task completions
- Procedural → Learns how to do things (think: internalized prompts/instructions)



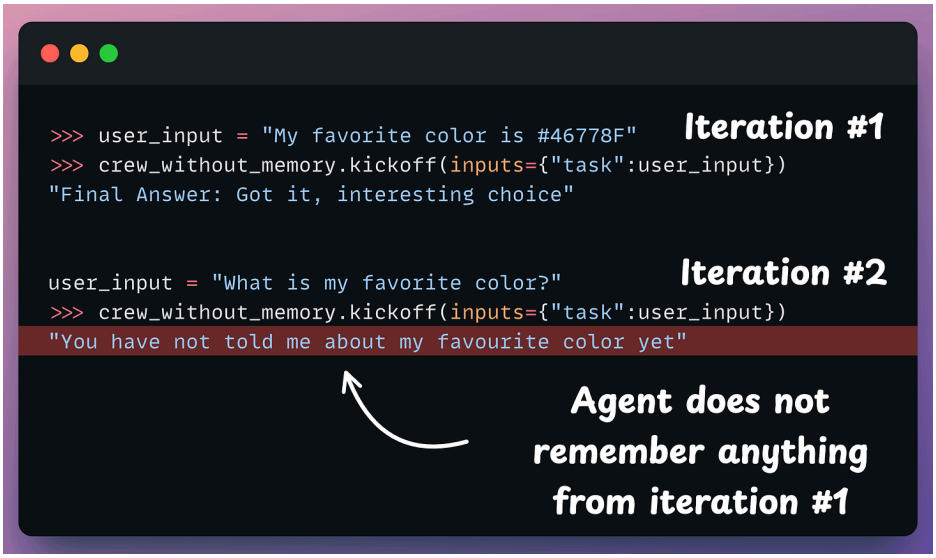
This memory isn't just nice-to-have but it enables agents to learn from past interactions without retraining the model.

This is especially powerful for continual learning: letting agents adapt to new tasks without touching LLM weights.

Importance of Memory for Agentic Systems

Let us now understand why memory is so powerful for Agentic systems?

Consider an Agentic system without Memory (below):



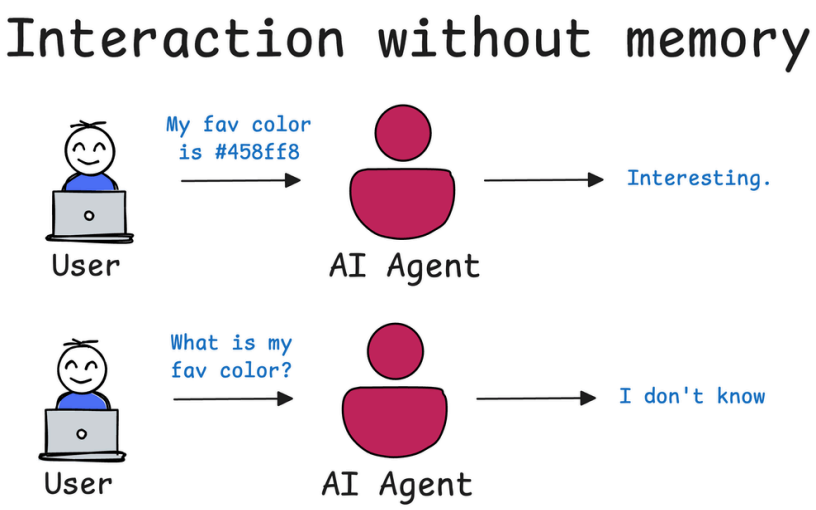
- In iteration #1, the user mentions their favorite color.
- In iteration #2, the Agent knows nothing about iteration #1.

This means the Agent is mostly stateless, and it has no recall abilities.

But now consider an Agentic system built with Memory (below):

- In iteration #1, the user mentions their favorite color.
- In iteration #2, the Agent can recall iteration #1.

Memory matters because if a memory-less Agentic system is deployed in production, every interaction with that Agent will be a blank slate.

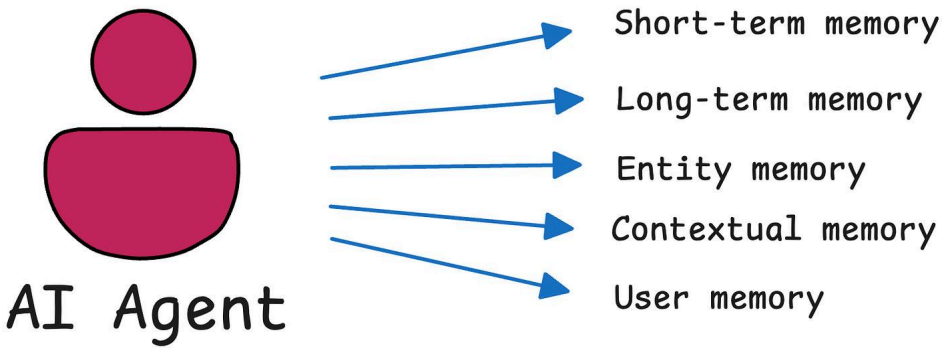


It doesn't matter if the user told the Agent their name five seconds ago, it's forgotten. If the Agent helped troubleshoot an issue in the last session, it won't remember any of it now.

With Memory, your Agent becomes context-aware and practically applicable.

But Memory isn't an abstract concept.

If you dive deeper, it follows a structured and intuitive architecture with several types of Memory.

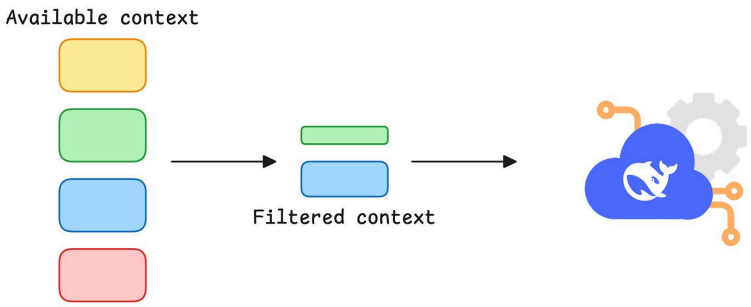


- Short-Term Memory
- Long-Term Memory
- Entity Memory
- Contextual Memory, and

- User Memory

Each serves a unique purpose in helping agents "remember" and utilize past information.

To simulate memory, the system has to manage context explicitly: choosing what to keep, what to discard, and what to retrieve before each new model call.

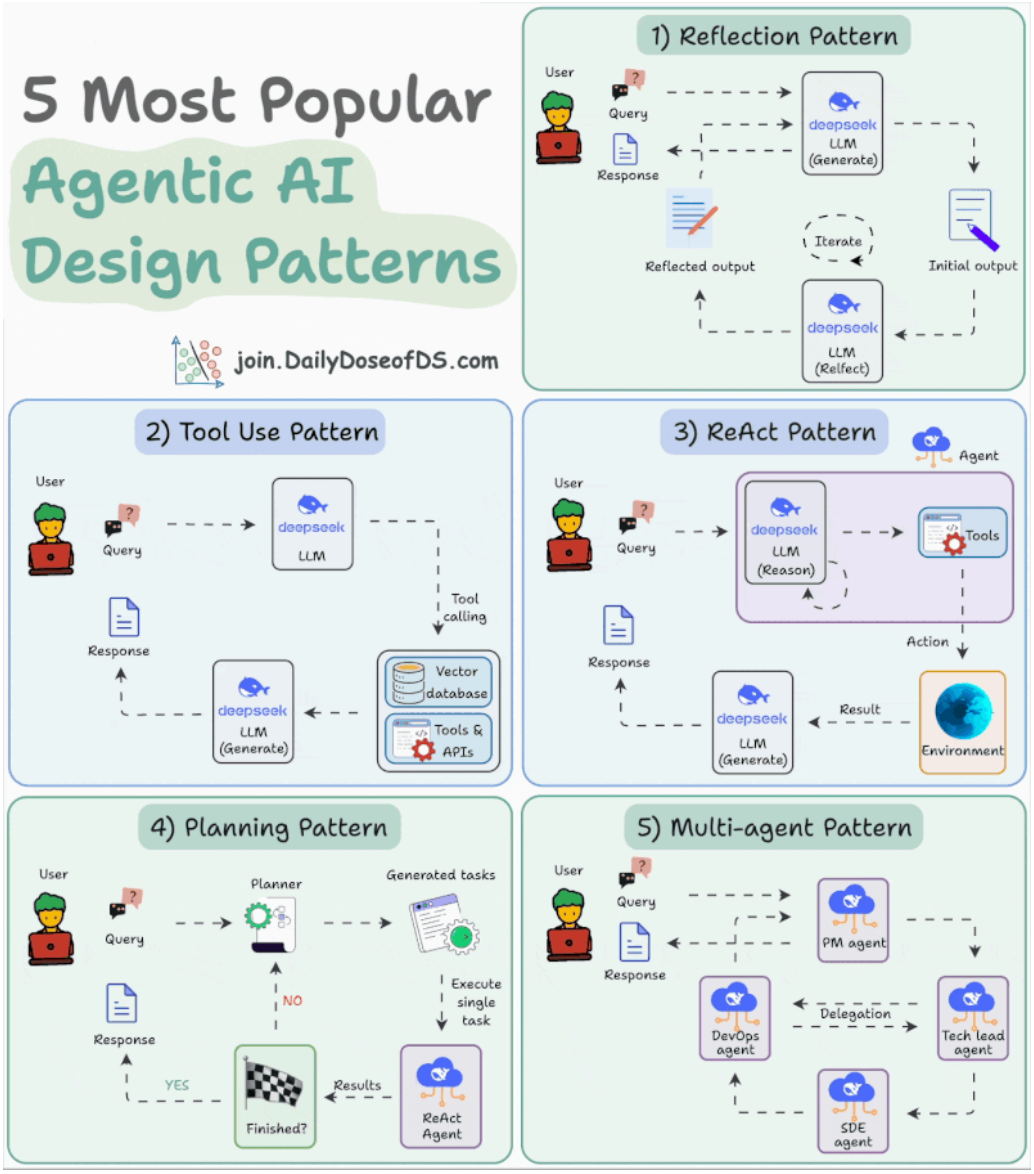


This is why memory is not a property of the model itself. It is a system design problem.

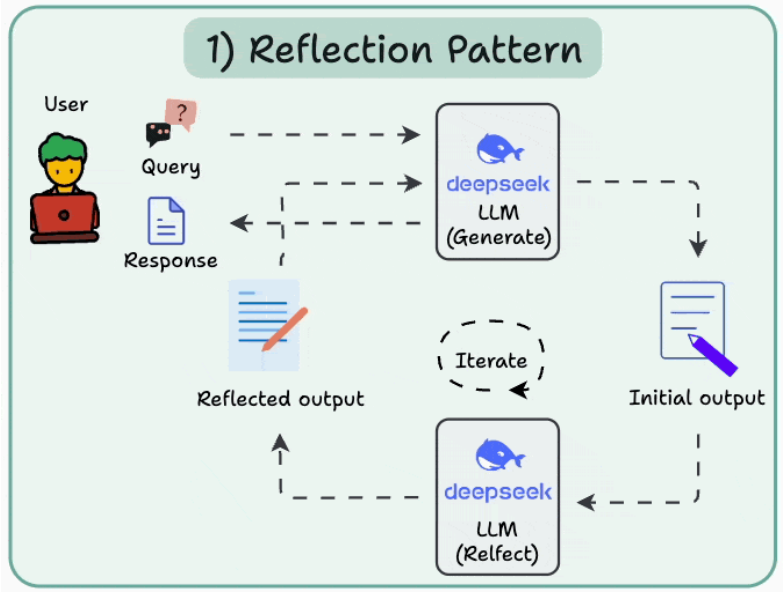
5 Agentic AI Design Patterns

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.

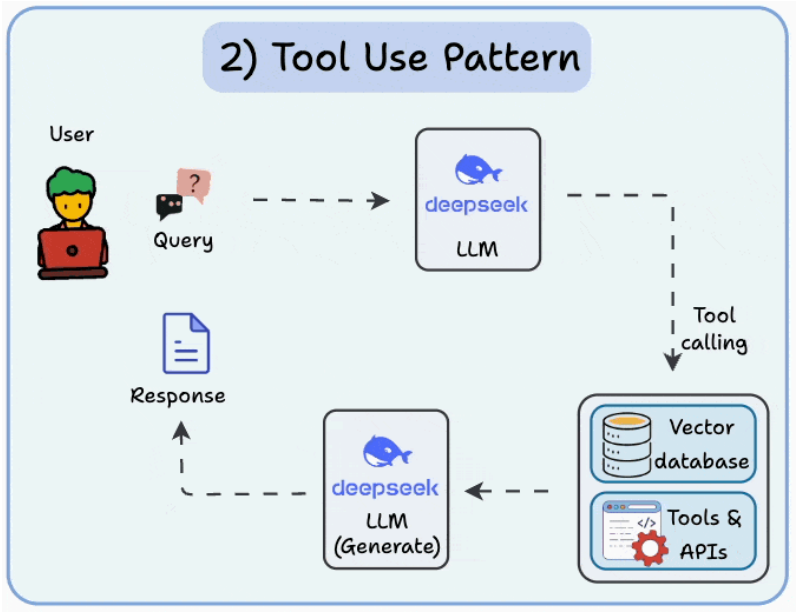


1) Reflection pattern



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

2) Tool use pattern



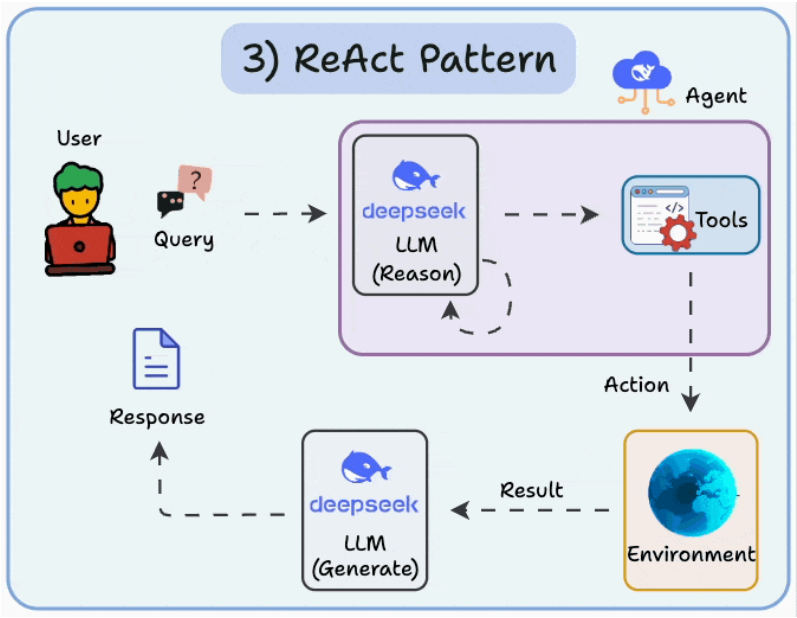
Tools allow LLMs to gather more information by:

- Querying a vector database
- Executing Python scripts

- Invoking APIs, etc.

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

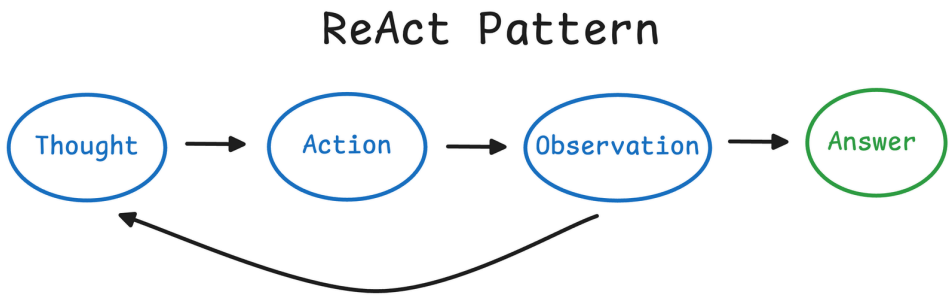
3) ReAct (Reason and Act) pattern



ReAct combines the above two patterns:

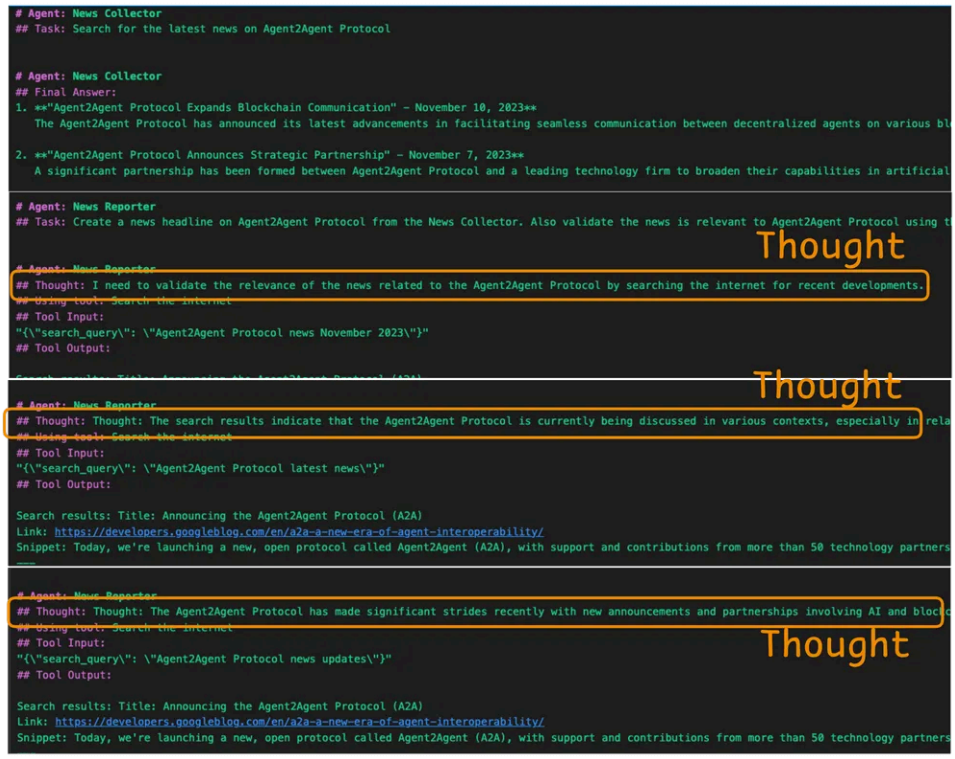
- The Agent reflects on the generated outputs.
- It interacts with the world using tools.

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.

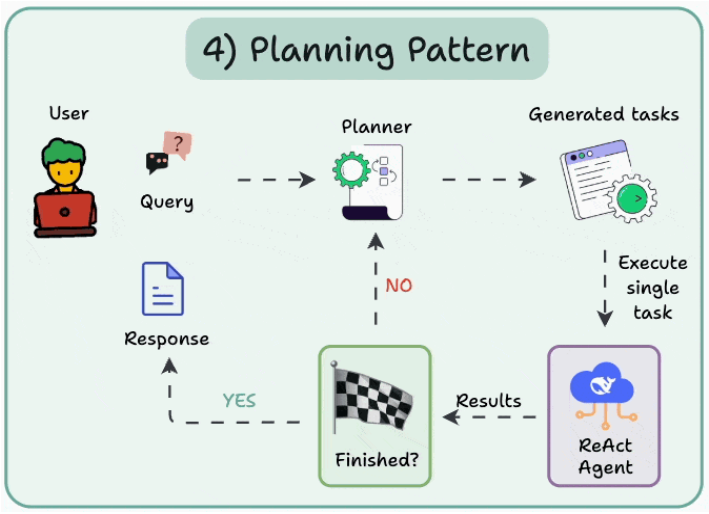
This is the ReAct pattern in action!

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.

4) Planning pattern



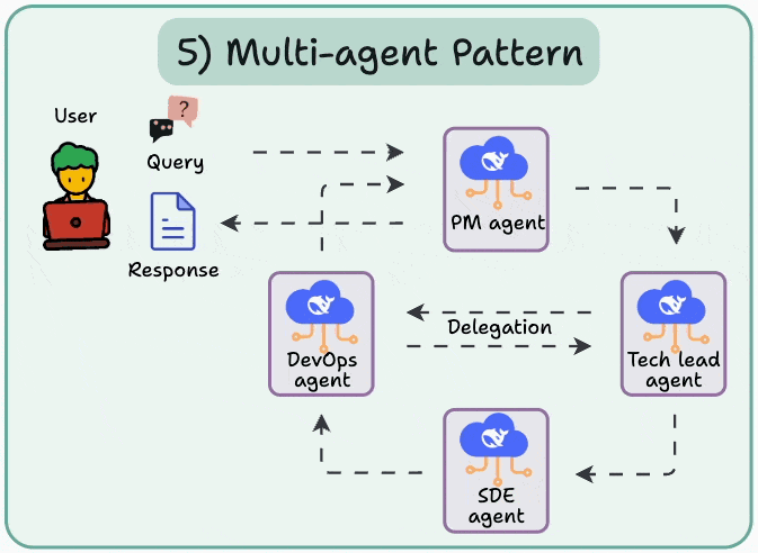
Instead of solving a task in one go, the AI creates a roadmap by:

- 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.

ReAct Implementation from Scratch

Below, we shall implement a ReAct Agent in two ways:

- Manually executing each step for better clarity.
- Without manual intervention to fully automate the Reasoning and Action process.

Let's look at the manual process first.

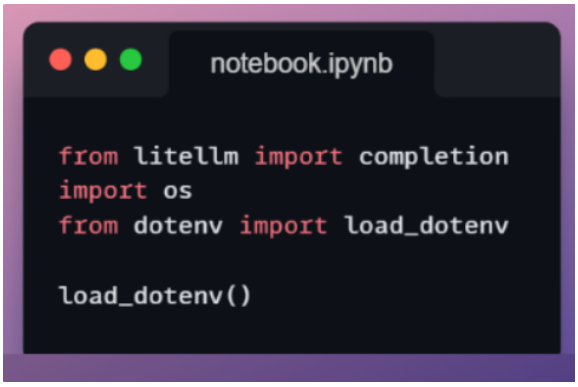
#1) ReAct with manual execution

In this section, we'll implement a lightweight ReAct-style agent from scratch, without using any orchestration framework like CrewAI or LangChain.

We'll manually simulate each round of the agent's reasoning, pausing, acting and observing exactly as a ReAct loop is meant to function.

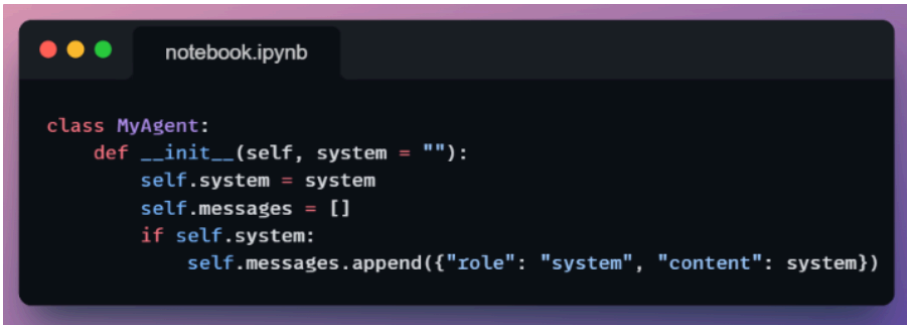
By running the logic cell-by-cell, we will gain full visibility and control over the thinking process, allowing us to debug and validate the agent's behavior at each step.

To begin, we load the environment variables (like your LLM API key) and import completion from LiteLLM (also install it first–pip install litellm), a lightweight wrapper to query LLMs like OpenAI or local models via Ollama.



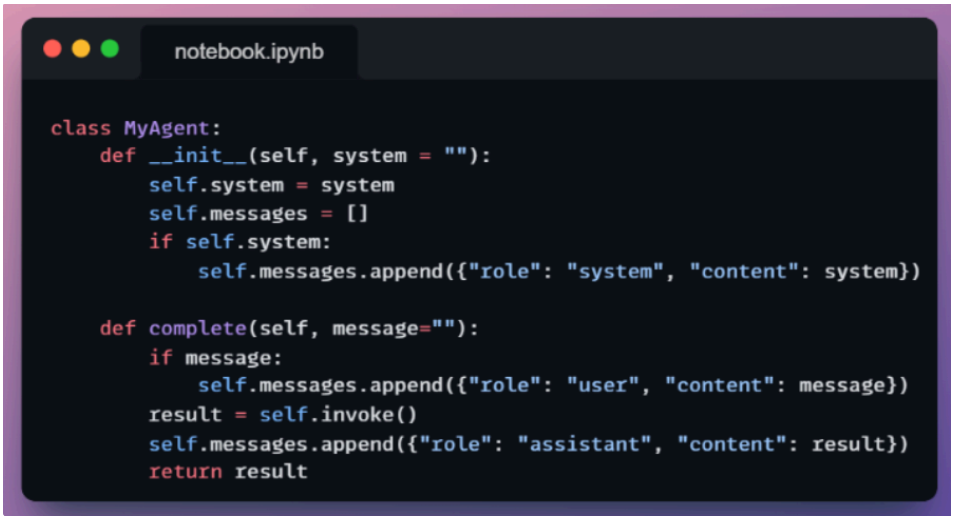
Next, we define a minimal Agent class, which wraps around a conversational LLM and keeps track of its full message history - allowing it to reason step-by-step, access system prompts, remember prior inputs and outputs, and produce multi-turn interactions.

Here's what it looks like:



- ***system*** (str): This is the system prompt that sets the personality and behavioral constraints for the agent. If passed, it becomes the very first message in the conversation just like in OpenAI Chat APIs.
- ***self.messages***: This list acts as the conversation memory. Every interaction, whether it's user input or assistant output is appended to this list. This history is crucial for LLMs to behave coherently across multiple turns.
- If ***system*** is provided, it's added to the message list using the special *"**role**": "**system**"* identifier. This ensures that every completion that follows is conditioned on the system instructions.

Next, we define a ***complete*** method in this class:



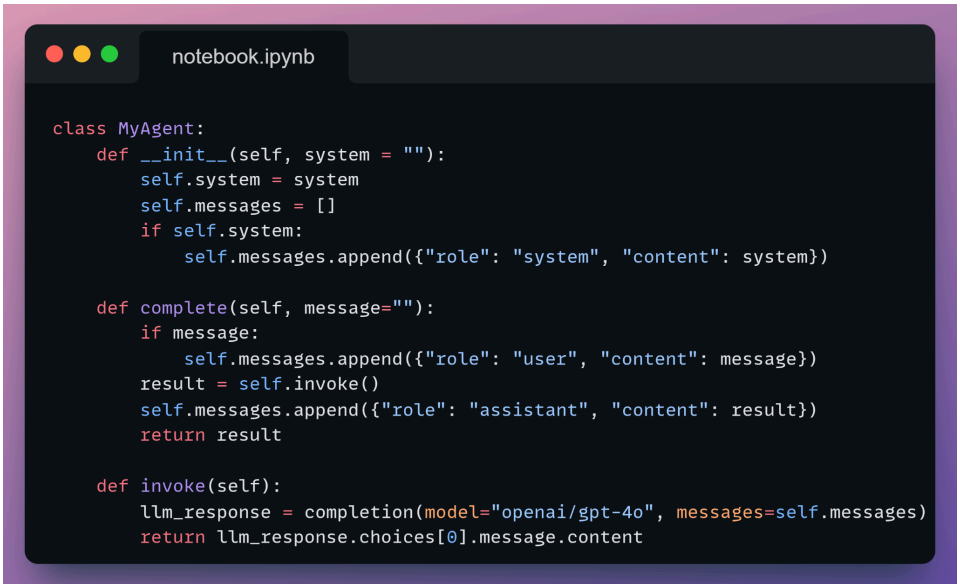
This is the core interface you'll use to interact with your agent.

- If a ***message*** is passed:
  - It gets appended as a *"**user**"* message to ***self.messages***.
  - This simulates the human asking a question or giving instructions.
- Then, ***self.invoke()*** is called (which we will define shortly). This method sends the full conversation history to the LLM.
- The model's reply (stored in ***result***) is then appended to ***self.messages*** as an *"**assistant**"* role.
- Finally, the reply is returned to the caller.

This method does three things in one call:

1. Records the user input.
2. Gets the model's reply.
3. Updates the message history for future turns.

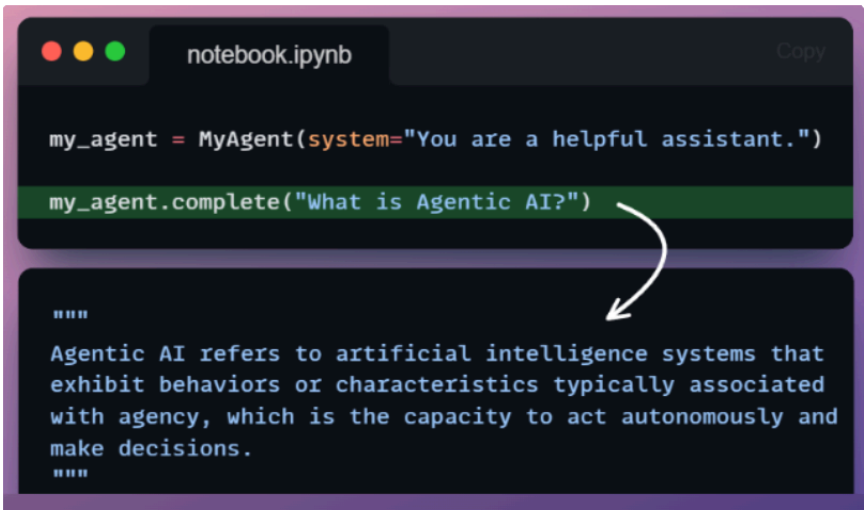
Finally, we have the ***invoke*** method below:



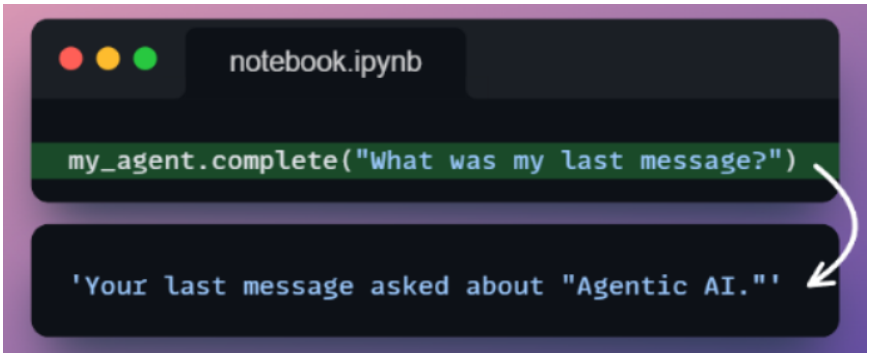
This method handles the actual API call to your LLM provider - in this case, via LiteLLM, using the *"**openai/gpt-4o**"* model.

- ***completion()*** is a wrapper around the chat completion API. It receives the entire message history and returns a response.
- We assume ***completion()*** returns a structure similar to OpenAI's format: a list of choices, where each choice has a ***.message.content*** field.
- We extract and return that content - the assistant's next response.

As a test, we can quickly run a simple interaction below:



At this stage, if we ask it about the previous message, we get the correct output, which shows the assistant has visibility on the previous context:



It correctly remembers and reflects!

Now that our conversational class is setup, we come to the most interesting part, which is defining a ReAct-style prompt.

Before an LLM can behave like an agent, it needs clear instructions - not just on what to answer, but how to go about answering. That's exactly what this ***system_prompt*** does, which is defined below:

notebook.ipynb

```
system_prompt = """
You run in a loop and do JUST ONE thing in a single iteration:

1) "Thought" to describe your thoughts about the input question.
2) "PAUSE" to pause and think about the action to take.
3) "Action" to decide what action to take from the list of actions available to you.
4) "PAUSE" to pause and wait for the result of the action.
5) "Observation" will be the output returned by the action.

At the end of the loop, you produce an Answer.

The actions available to you are:

math:
e.g. math: (14 * 5) / 4
Evaluates mathematical expressions using Python syntax.

lookup_population:
e.g. lookup_population: India
Returns the latest known population of the specified country.

Here's a sample run for your reference:

Question: What is double the population of Japan?

Iteration 1:
Thought: I need to find the population of Japan first.

Iteration 2:
PAUSE

Iteration 3:
Action: lookup_population: Japan

Iteration 4:
PAUSE

(you will now receive an output from the action)

Iteration 5:
Observation: 125,000,000

Iteration 6:
Thought: I now need to multiply it by 2.

Iteration 7:
Action: math: 125000000 * 2

Iteration 8:
PAUSE

(you will now receive an output from the action)

Iteration 9:
Observation: 250000000

Iteration 10:
Answer: Double the population of Japan is 250 million.

Whenever you have the answer, stop the loop and output it to the user.

Now begin solving:
""".strip()
```

This isn't just a prompt. It's a behavioral protocol - defining what structure the agent should follow, how it should reason, and when it should stop.

Let's break it down line by line.

You run in a loop and do JUST ONE thing in a single iteration:

This is the framing sentence. It tells the LLM not to rush toward an answer. Instead, it should proceed step by step, following a defined pattern in a loop - mirroring how a ReAct agent works.

1) "Thought" to describe your thoughts about the input question.

2) "PAUSE" to pause and think about the action to take.

3) "Action" to decide what action to take from the list of actions available to you.

4) "PAUSE" to pause and wait for the result of the action.

5) "Observation" will be the output returned by the action.

Here, we give the LLM a reasoning template. These are the same primitives found in all ReAct-style agents.

Let's break each down:

- Thought: The agent's internal monologue. What is it currently thinking about?
- PAUSE (1): Instead of jumping to action, this forces the model to take a breath - simulating asynchronous steps in a multi-agent environment.
- Action: The agent picks from the list of tools it is given.
- PAUSE (2): Wait again, this time for the actual tool result.
- Observation: This will be injected into the prompt by you (the controller or human), after the tool runs.

By splitting this into explicit parts, we avoid hallucinations and ensure the agent works in a controlled loop.

At the end of the loop, you produce an Answer.

This tells the agent: once it has all the required information - break the loop and give the final answer. No need to keep reasoning indefinitely.

The actions available to you are:

math:

*e.g. math: (14 * 5) / 4*

Evaluates mathematical expressions using Python syntax.

lookup_population:

e.g. lookup_population: India

Returns the latest known population of the specified country.

This is a mini API reference for the agent. We show:

- The name of each tool.
- How to invoke it.
- What kind of output it produces.

This is critical. Without a clear spec, the LLM might:

- Invent non-existent tools.
- Use incorrect syntax.
- Misinterpret what the tool is supposed to do.

By using clear formatting and examples, we teach the model how to interface with tools in a safe, predictable way.

Here's a sample run for your reference:

Question: What is double the population of Japan?

Iteration 1:

Thought: I need to find the population of Japan first.

Iteration 2:

PAUSE

...

Iteration 9:

Observation: 250000000

Iteration 10:

Answer: Double the population of Japan is 250 million.

This worked-out example gives the LLM a pattern to follow. Even more importantly, it provides the developer (you) a way to intervene at each step - injecting tool results or validating whether the flow is working correctly.

With this sample trace:

- The agent knows how to think.
- The agent knows how to act.
- The agent knows when to stop.

Whenever you have the answer, stop the loop and output it to the user.

Now begin solving:

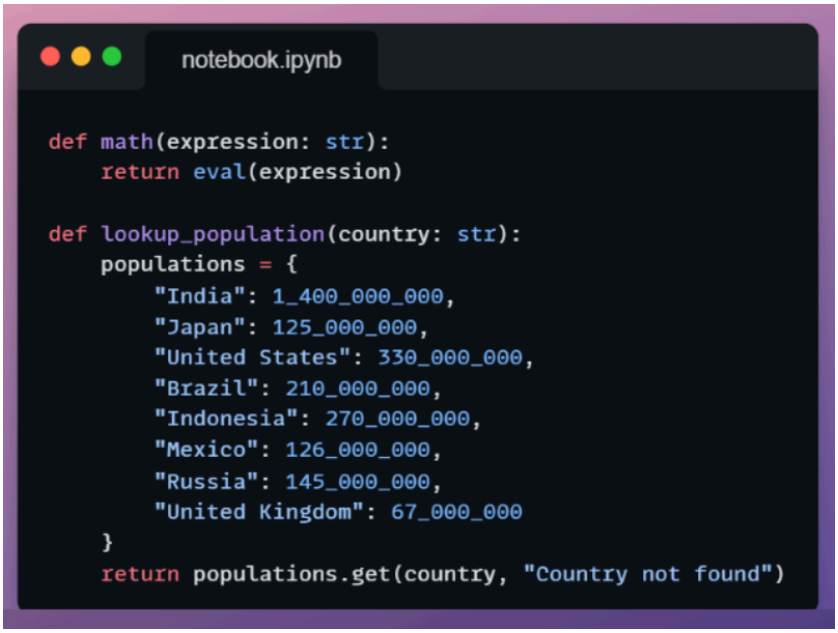
These closing lines are essential.

Without this explicit stop signal, the LLM might continue indefinitely. You're telling it: "When you have all the puzzle pieces, just say the answer and exit the loop."

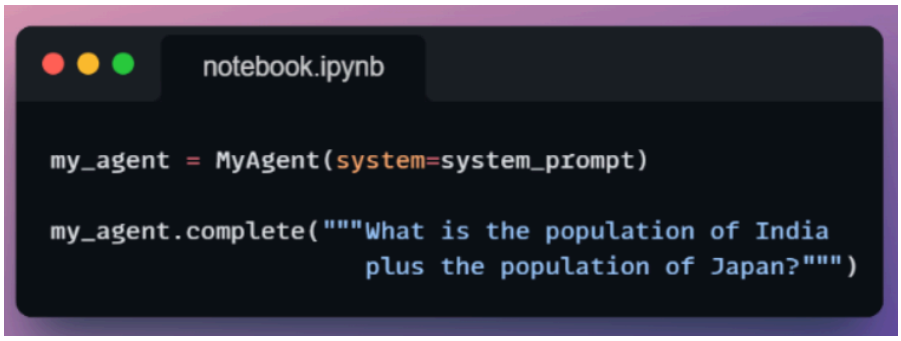
The power of this ***system_prompt*** lies in its structure:

- It models intelligent behavior, not just question answering.
- It imposes strong constraints: think before acting, act within defined bounds, and wait for observations.
- It separates reasoning from execution, mimicking how humans operate.
- It creates a feedback-friendly iteration loop for multi-step problems.

Now that the prompt is defined, we implement the tools.



Finally, we begin a manual ReAct session:

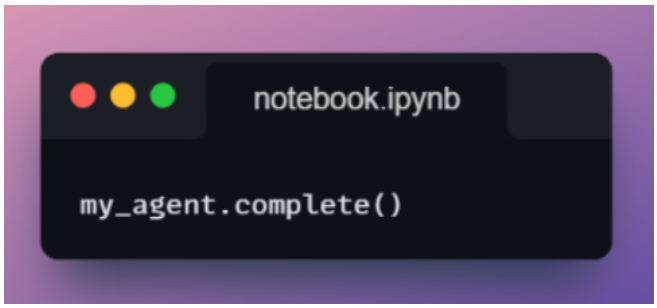


This produces the following output:

Iteration 1:

Thought: I need to find the population of India first.

We, as a user, don't have any input to give at this stage so we just invoke the ***complete()*** method again:

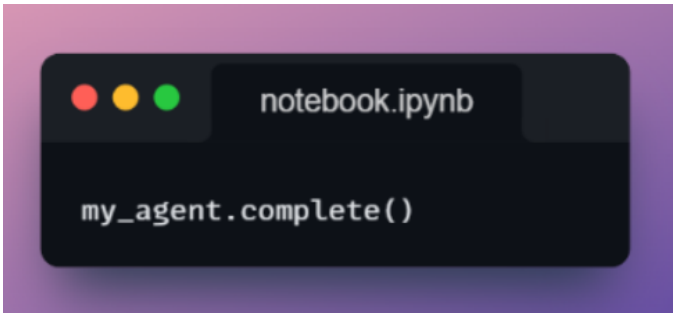


This produces the following output:

Iteration 2:

PAUSE

Yet again, we, as a user, don't have any input to give at this stage so we just invoke the ***complete()*** method again:



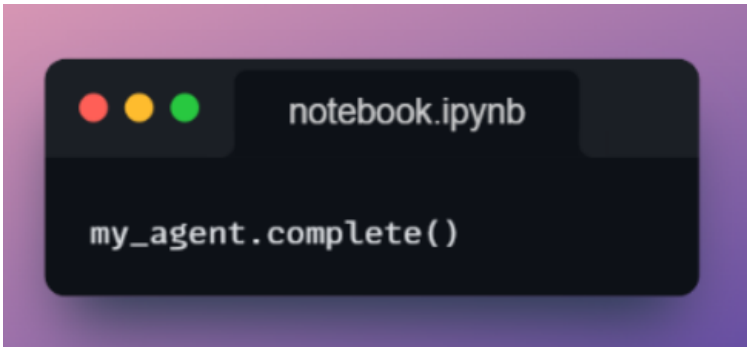
This produces the following output:

Iteration 3:

Action: lookup_population: India

Now it wants to act.

We still don't have any input to give at this stage so we just invoke the ***complete()*** method again:



This produces the following output:

Iteration 4:

PAUSE

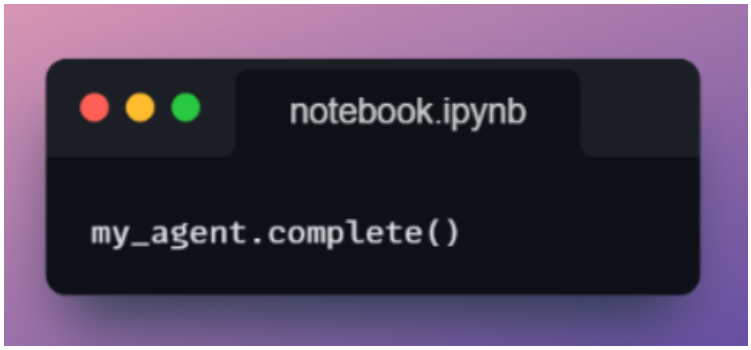
At this stage, it needs to get the tool output in the form of an observation. Here, let's intervene and provide it with the observation:

This produces the following output:

Iteration 5:

Thought: Now I need to find the population of Japan.

We let it continue its execution:

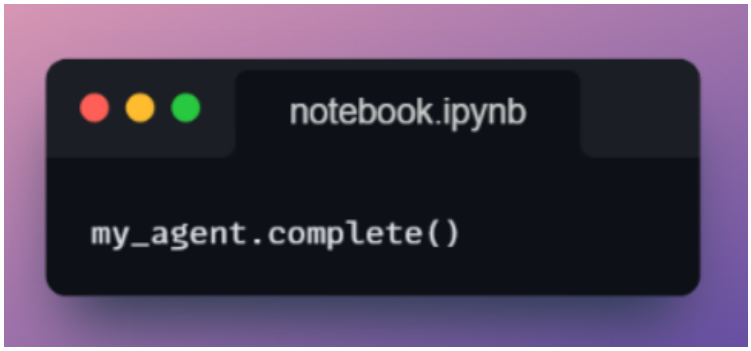


This produces the following output:

Iteration 6:

PAUSE

We again let it continue its execution:



We get the following output:

Iteration 7:

Action: lookup_population: Japan

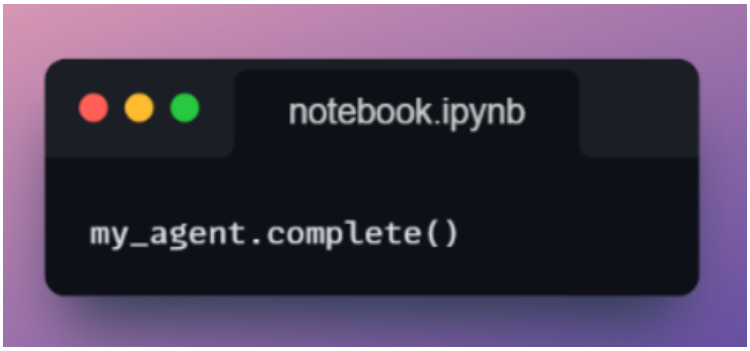
At this stage, it needs to get the tool output in the form of an observation. Here, let's again intervene and provide it with the observation:

This produces the following output:

Iteration 8:

Thought: I now have the populations of both India and Japan. I need to add them together.

We again let it continue its execution:

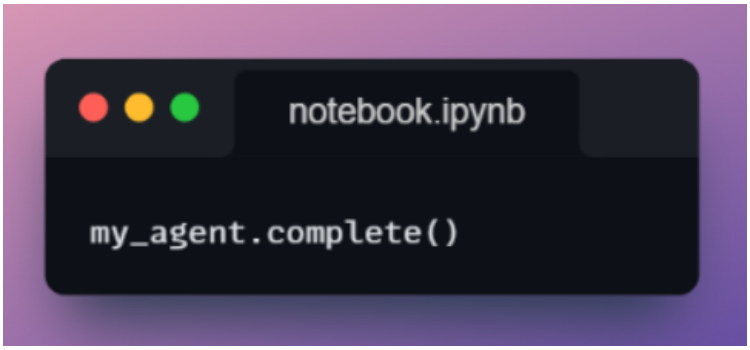


We get the following output:

Iteration 9:

Action: math: 1400000000 + 125000000

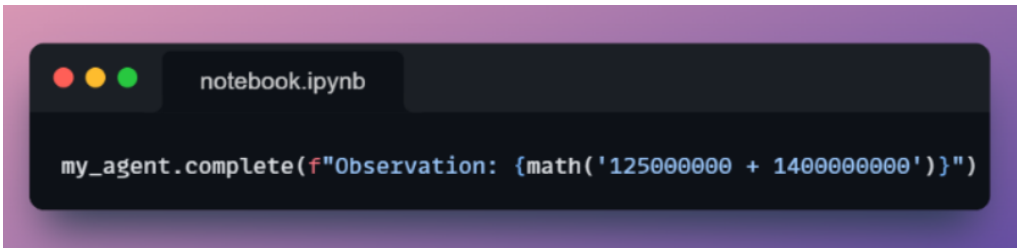
Now we should expect a pause according to the pattern specified:



Iteration 10:

PAUSE

It is again seeking an observation, which is the sum of Japan's population and India's population. To do this, we again manually intervene and provide it with the output:



Finally, in this iteration, we get the following output:

Iteration 11:

Answer: The sum of the population of India and the population of Japan is 1,525,000,000.

Great!!

With this process:

- The LLM thought about what steps to take.
- It chose actions to execute.
- We manually injected tool outputs like real-world observations.

- It looped until it had enough information to generate a final answer.

This gives us an explicit understanding of how reasoning and actions come together in ReAct-style agents.

In the next part, we'll fully automate this - no manual calls required and build a full controller that simulates this entire loop programmatically.

#2) ReAct without manual execution

Now that we have understood how the above ReAct execution went, we can easily automate that to remove our interventions.

In this section, we'll create a controller function that:

- Sends an initial question to the agent,
- Reads its thoughts and actions step-by-step,
- Automatically runs external tools when asked,
- Feeds back observations to the agent,
- And stops the loop once a final answer is found.

This is the entire code that does this:

Let's break down the full loop.

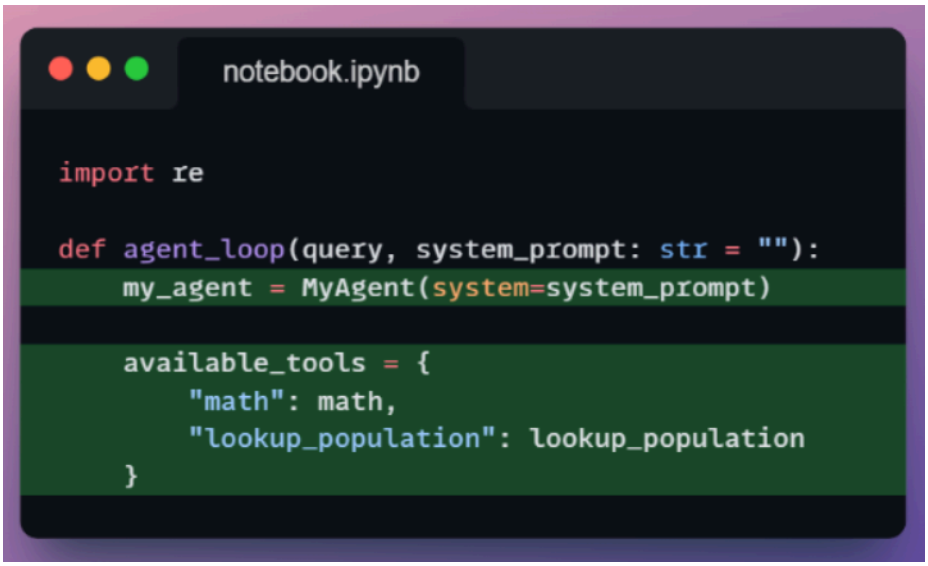
We begin by defining the ***agent_loop()*** function:

It takes:

- ***query***: the user's natural language question.

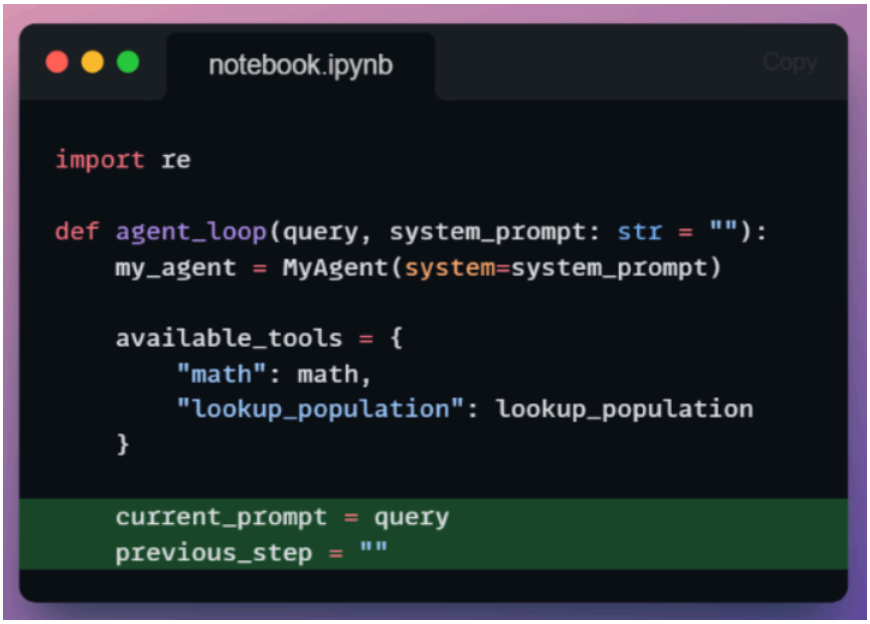
- ***system_prompt***: the same ReAct system prompt we explored earlier (defining the behavior loop).

Next, inside this function, we initialize the Agent and available tools:



- Create a new MyAgent instance, using the structured ReAct prompt.
- Define the dictionary of callable tools available to the agent. These names must match exactly what the agent uses in its ***Action:*** lines.

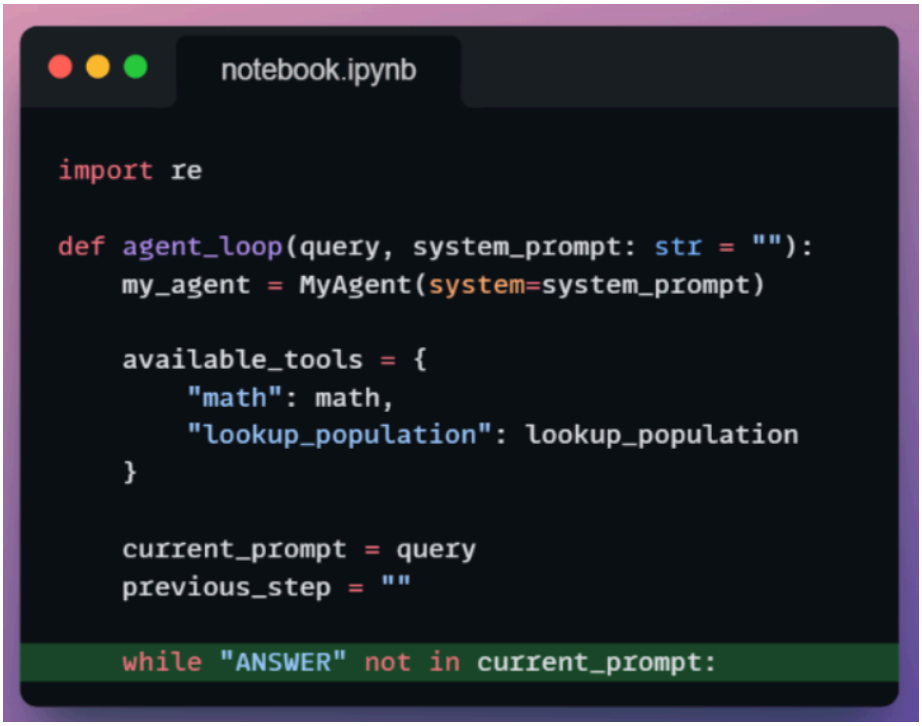
Moving on, we defined some state variables:



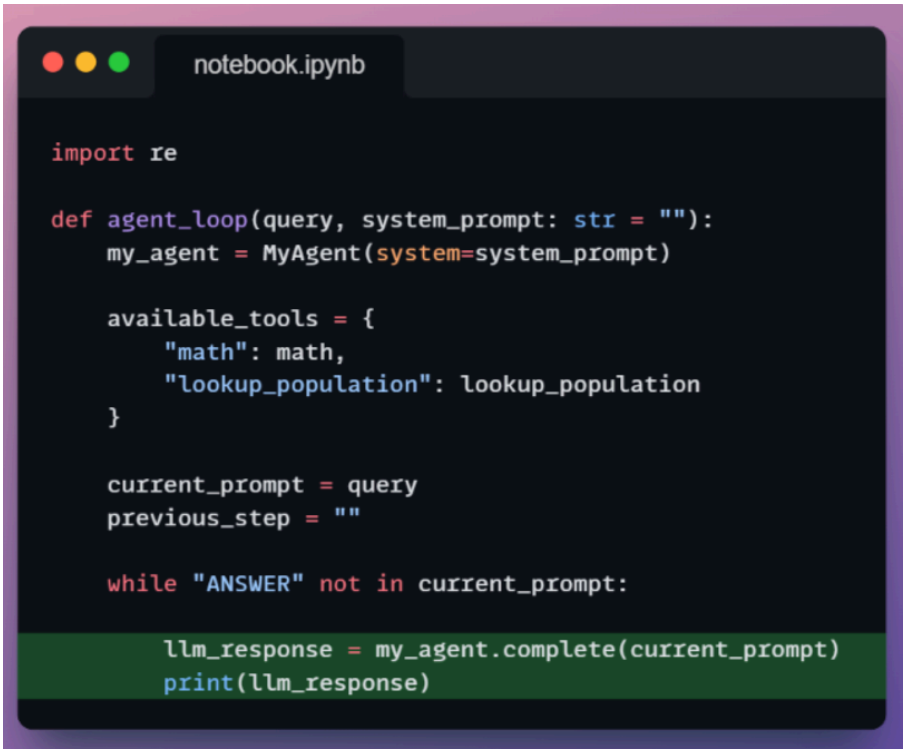
current_prompt stores the next message to be sent to the LLM.

previous_step helps track the last stage (e.g., Thought, Action) for better control flow.

Next, we run the reasoning loop, which continues until the agent produces a final answer. The answer is expected to be marked with ***Answer:*** based on our prompt design:



Next, we feed the ***current_prompt*** into the agent.

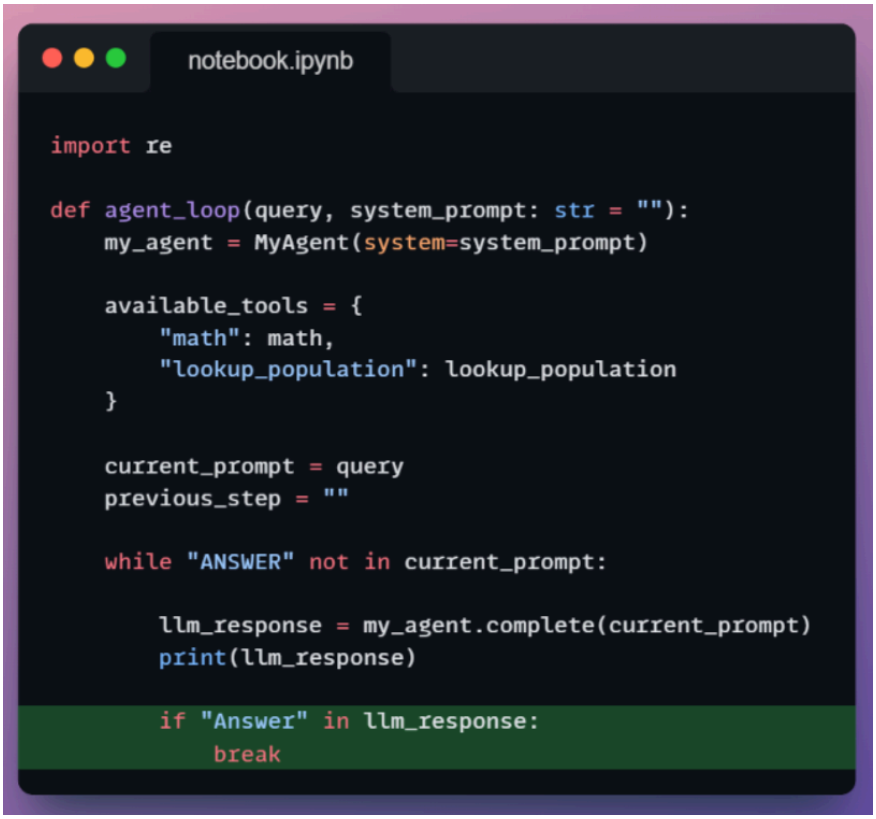


The ***current_prompt*** could be:

- The initial user query,
- A blank string to let the agent continue reasoning,
- An observation from a tool.

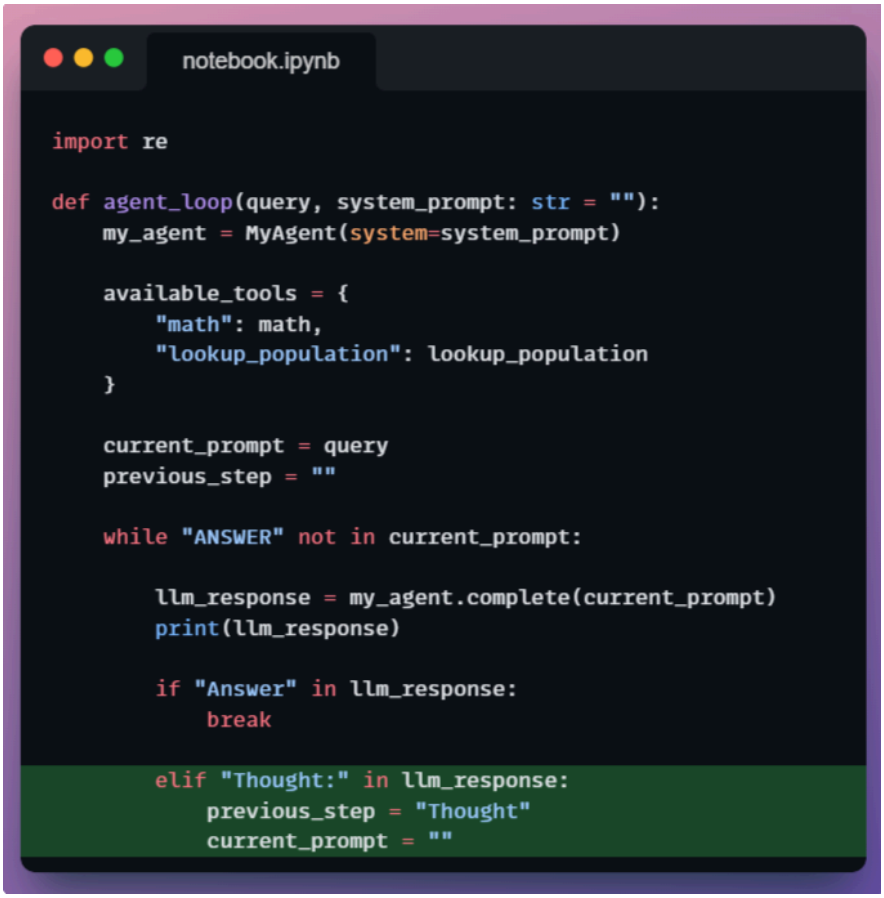
We then print the agent's output, so we can inspect each iteration.

Next, if the agent produces a final answer, we break the loop.



In another case, if the response includes a ***Thought:*** line, we:

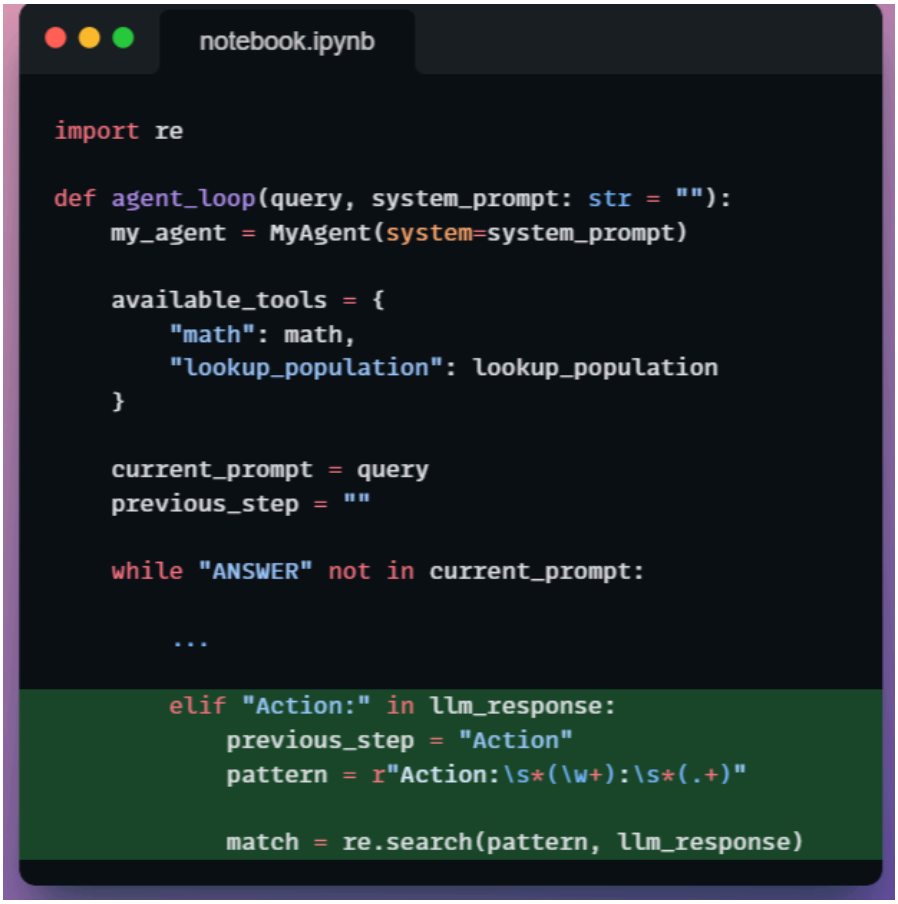
- Record the step type as "Thought".
- Set ***current_prompt*** to an empty string to continue to the next stage (a PAUSE).

- 

Next, we catch the first PAUSE right after the Thought. Nothing else needs to be done here - we just move to the next step.

If we detect an ***Action:*** line, we:

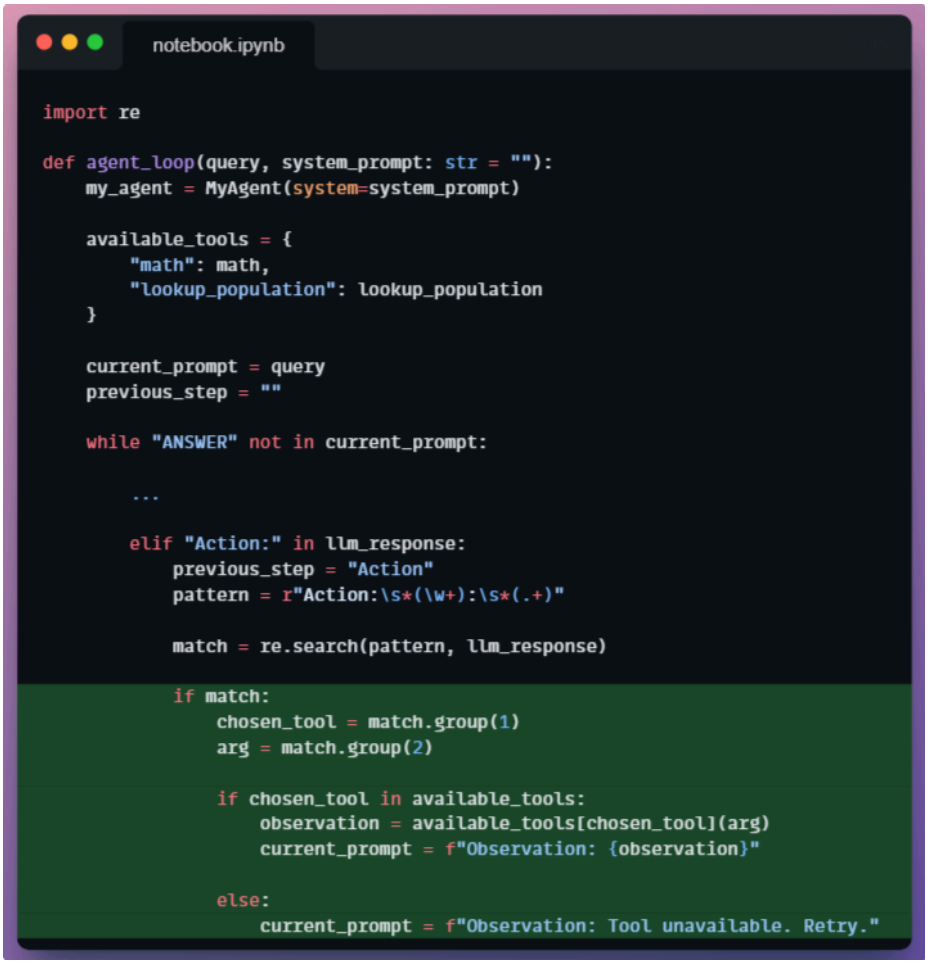
- Note that we're in the action step.
- Use a regex to extract the tool name and its argument.

- 

For example, in: ***Action: lookup_population: India***, the regex pulls out:

- ***lookup_population*** as the tool.
- ***India*** as the argument.

Moving on, we execute the tool and capture the observation:

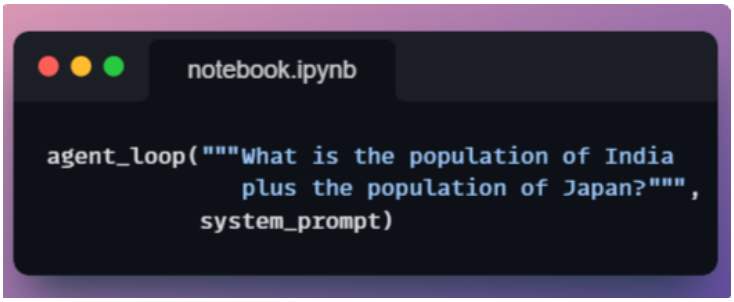


- If the tool name is valid, we call it like a Python function and capture the result.
- We format the output into Observation: ... so the agent can use it in the next step.
- If the tool doesn't exist, we ask the agent to retry.

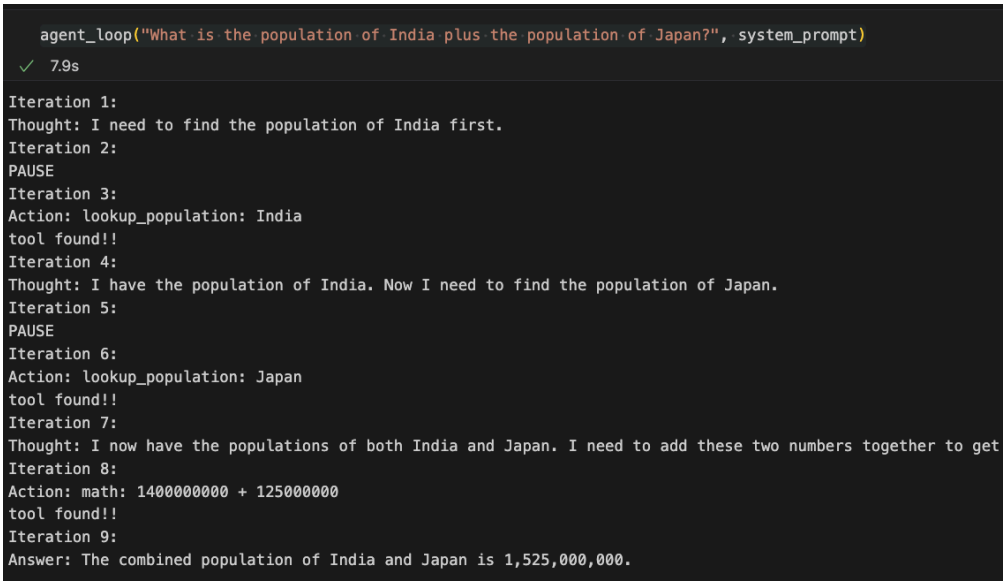
This mimics tool execution + response injection.

Done!

Now we can run this function as follows:



This produces the following output, which is indeed correct:



You now have a fully working ReAct loop without needing any external framework.

Of course, In this implementation, we're using regex matching and hardcoded conditionals to parse the agent's actions and route them to the correct tools.

This approach works well for a tightly controlled setup like this demo. However, it's brittle:

- If the agent slightly deviates from the expected format (e.g., adds extra whitespace, uses different casing, or mislabels an action), the regex could fail to match.

- We're also assuming that the agent will never call a tool that doesn't exist, and that all tools will succeed silently.

In a production-grade system, you'd want to:

- Add more robust parsing (e.g., structured prompts with JSON outputs or function calling).
- Include tool validation, retries, and exception handling.
- Use guardrails or output formatters to constrain what the LLM is allowed to emit.

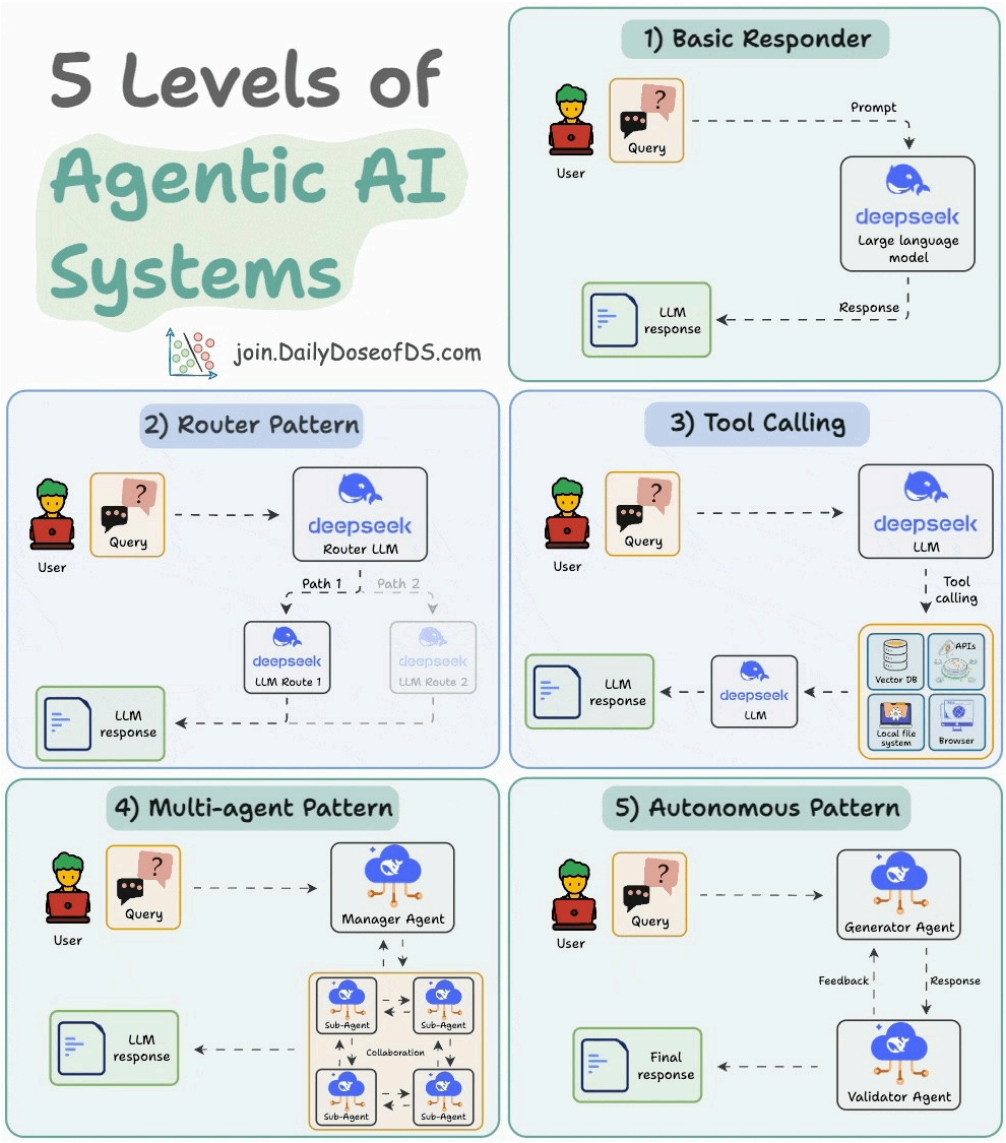
But for the purpose of understanding how ReAct-style loops work under the hood, this is a clean and minimal place to start. It gives you complete transparency into what's happening at each stage of the agent's reasoning and execution process.

This loop demonstrates how a simple agent can think, act, and observe, all powered by your own Python + local LLM stack.

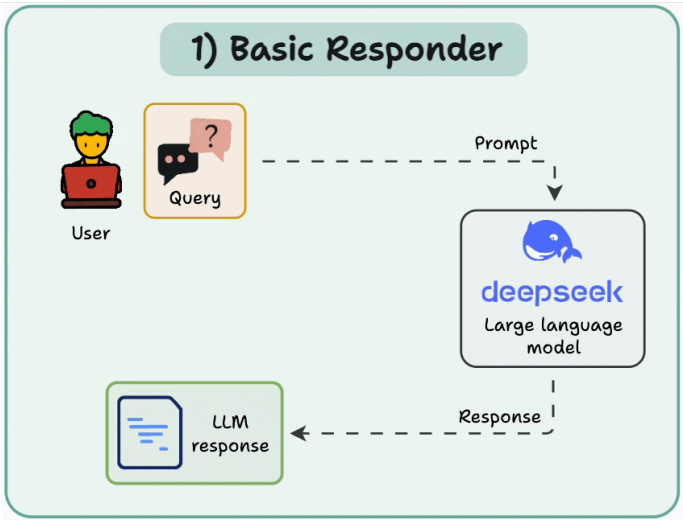
5 Levels of Agentic AI Systems

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.



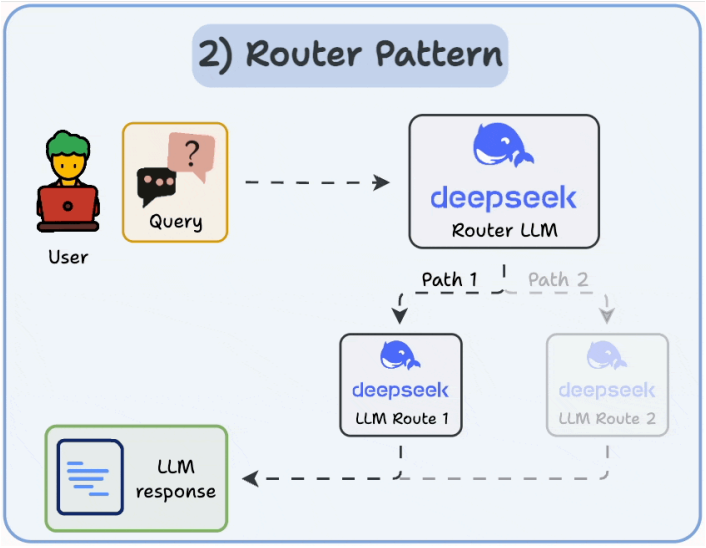
1) Basic responder



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.

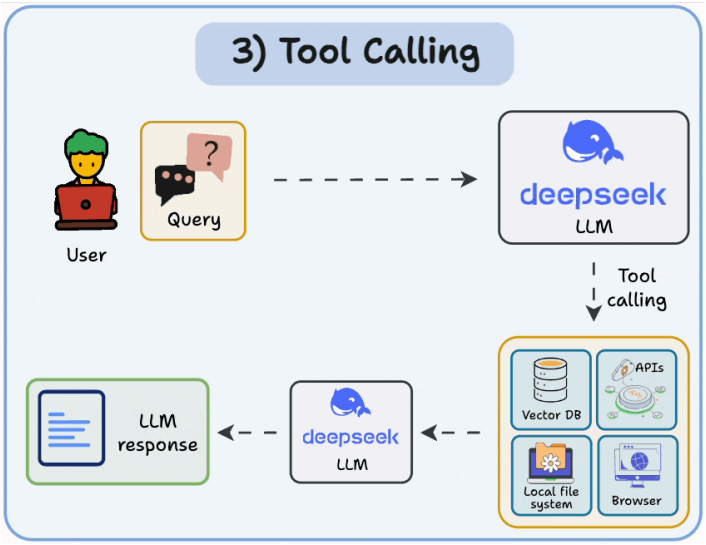
2) Router pattern



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

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

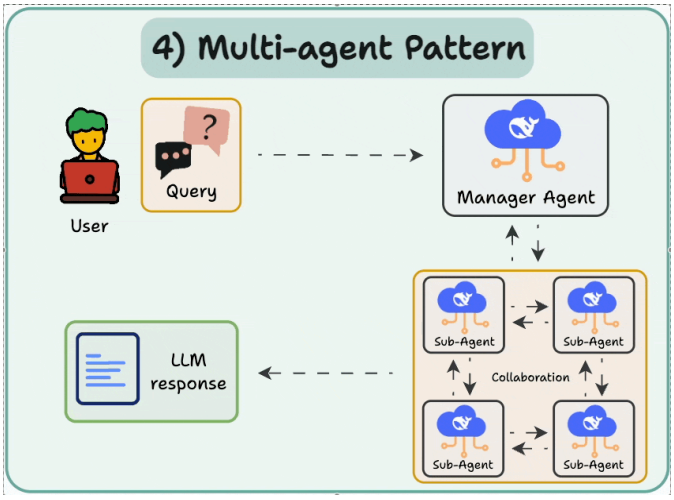
3) Tool calling



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

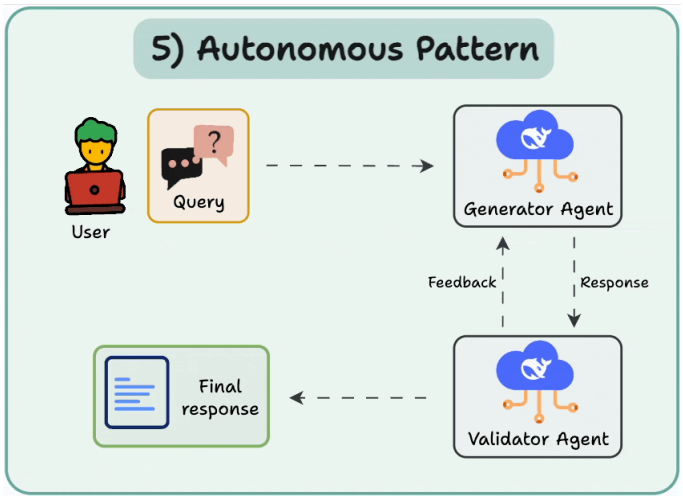


A manager agent coordinates multiple sub-agents and decides the next steps iteratively.

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

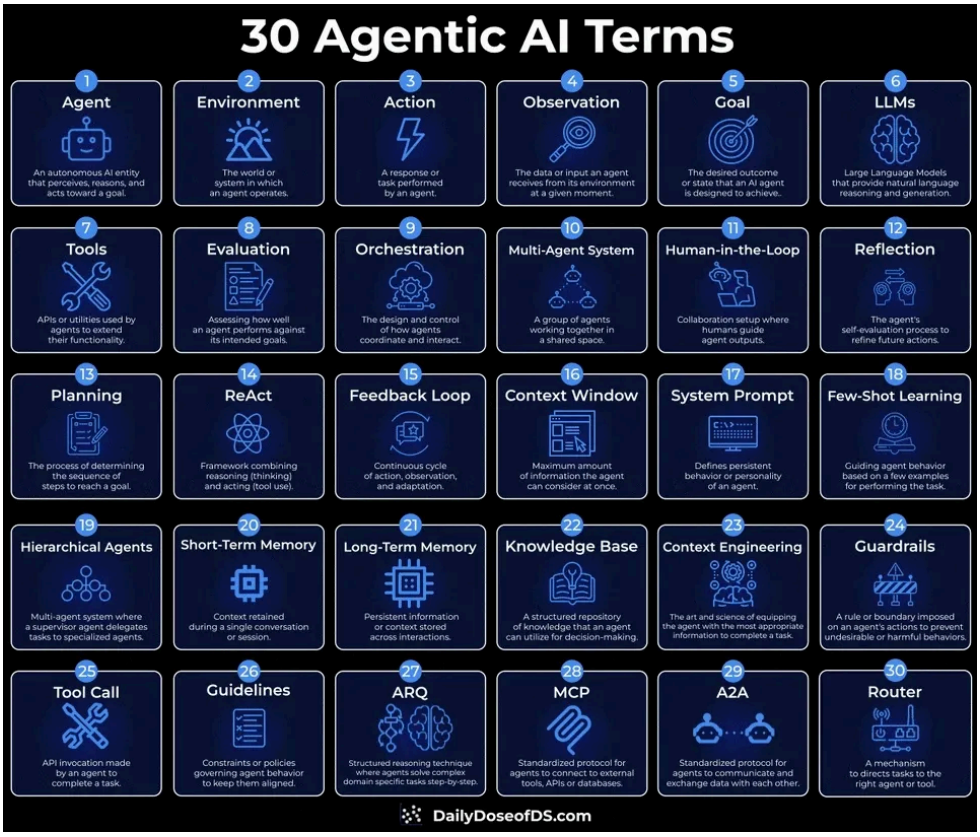


The most advanced pattern, wherein, the LLM generates and executes new code independently, effectively acting as an independent AI developer.

30 Must-Know Agentic AI Terms

We put together a quick visual guide to the 30 most important terms in Agentic AI, covering some of the most essential things you need to understand about how modern AI agents actually think, act, and collaborate.

If you've been exploring agent frameworks like CrewAI, LangGraph, or AutoGen, this glossary will help you connect the dots between key building blocks.



Agent: An autonomous AI entity that perceives, reasons, and acts toward a goal (covered with full implementations here).

Environment: The world or system in which an agent operates and interacts.

Action: A response or task performed by an agent based on its reasoning or goals.

Observation: The data or input an agent receives from its environment at any given moment.

Goal: The desired outcome that an Agent is designed to achieve.

LLMs: Large Language Models that enable agents to reason and generate natural language.

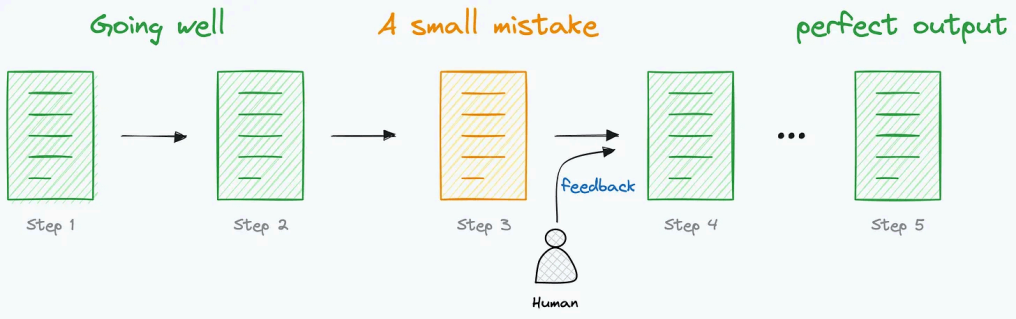
Tools: APIs or utilities agents use to extend their functionality and capabilities to interact with the world.

Evaluation: The process of assessing how well an agent performs against its intended goals (covered here with implementation).

Orchestration: The coordination and control of multiple agents working together to achieve complex tasks.

Multi-agent system: A group of agents collaborating to accomplish a final goal (implemented from scratch in pure Python here).

Human-in-the-loop: A setup where humans intervene or guide the agent's decision-making process.



Reflection: The agent's process of self-assessing its actions to improve future performance.

Planning: Determining the sequence of steps an agent must take to reach its goal (implemented from scratch in pure Python here).

ReAct: A framework where reasoning (thought) and acting (tool use) are combined step by step (implemented from scratch in pure Python here).

Feedback loop: A continuous process of collecting outcomes, observing effects, and adjusting actions.

Context window: The maximum amount of information an agent can consider at once.

System prompt: The persistent background instructions or personality that define an agent's behavior.

Few-shot learning: Teaching an agent new behaviors or tasks with just a few examples.

Hierarchical Agents: A multi-level agent structure where a supervisor agent delegates tasks to sub-agents.

Short-term memory: Temporary context stored during a single session or conversation.

Long-term memory: Persistent context stored across multiple sessions for continuity and learning (covered in detail here with code).

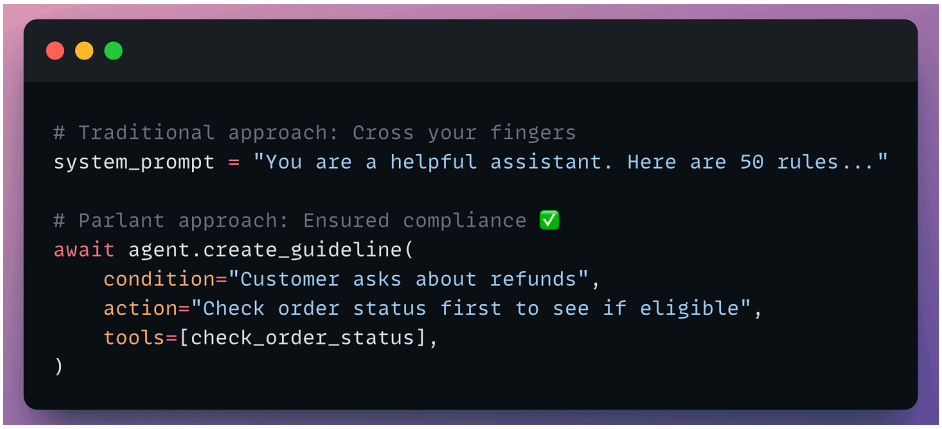
Knowledge base: A structured repository of information that agents can use for reasoning and decision-making (covered in detail here with code).

Context engineering: The practice of shaping what information an agent sees to optimize its output (here's a demo we covered).

Guardrails: Rules or boundaries that prevent an agent from taking harmful or undesired actions (covered with code here).

Tool call: An API invocation made by an agent to perform a specific task.

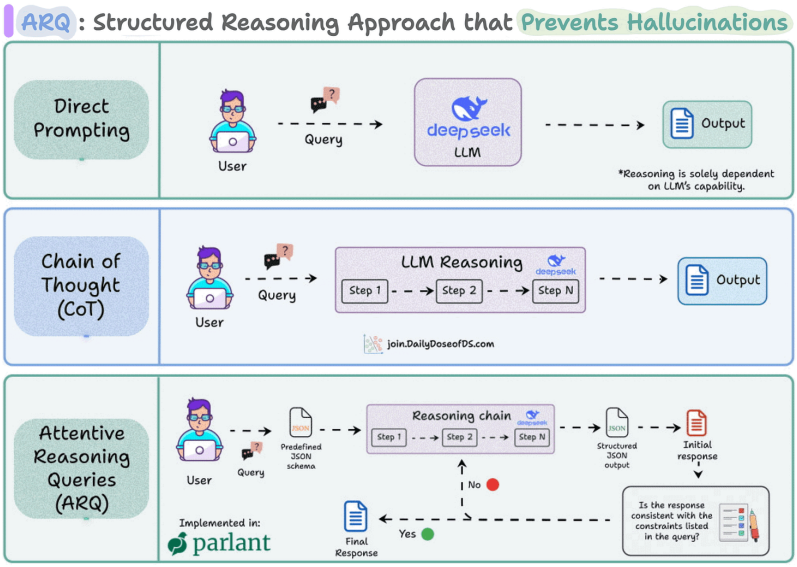
Guidelines: Policies or constraints that keep an agent's behavior aligned with desired outcomes.



```
# Traditional approach: Cross your fingers
system_prompt = "You are a helpful assistant. Here are 50 rules..."

# Parlant approach: Ensured compliance ✅
await agent.create_guideline(
    condition="Customer asks about refunds",
    action="Check order status first to see if eligible",
    tools=[check_order_status],
)
```

ARQ: A new structured reasoning approach where an agent solves complex, domain-specific problems step by step (covered here).



MCP: A standardized way for agents to connect to external tools, APIs, and data sources (learn how to build MCP servers, MCP clients, JSON-RPC, Sampling, Security, Sandboxing in MCPs, and using LangGraph/LlamaIndex/CrewAI/PydanticAI with MCP here).

A2A: Agent-to-Agent protocol enabling agents to communicate and exchange data directly (here's a visual guide).