Mar 30, 2025

A Practical Deep Dive Into Memory for Agentic Systems (Part A)

AI Agents Crash Course—Part 8 (with implementation).

Avi Chawla, Akshay Pachaar

Introduction

So far in this AI Agent crash course series, we've built systems that can:

- Collaborate across multiple crews and tasks.
- Use guardrails, callbacks, and structured outputs.
- Handle multimodal inputs like images.
- Reference outputs of previous tasks.
- Work asynchronously and under human supervision.

We've also explored how to equip agents with Knowledge, allowing them to access internal documentation, structured datasets, or any domain-specific reference material needed to complete their tasks.

However, all of this still leaves one major gap…

So far, our agents have mostly been stateless. They could access tools, reference documents, or perform tasks—but they didn't remember anything unless we passed it into their context explicitly.

What if you want your agents to:

- Recall something a user told them earlier in the conversation.
- Accumulate learnings across different sessions?
- Personalize answers for individual users based on their past behavior?
- Maintain facts about entities (like customers, projects, or teams) across workflows?

Memory solves this!

In this part, we'll explore how to:

- Use Short-Term Memory to retain and retrieve recent interaction context.

- Use Long-Term Memory to accumulate experience across sessions.
- Use Entity Memory to track and recall facts about specific people or objects.
- Use User Memory to personalize interactions.
- Reset, share, and manage memory across sessions and crews.

Everything will be demonstrated using real examples—so you can not only understand how memory works in CrewAI but also learn how to use it to make your agents actually intelligent.

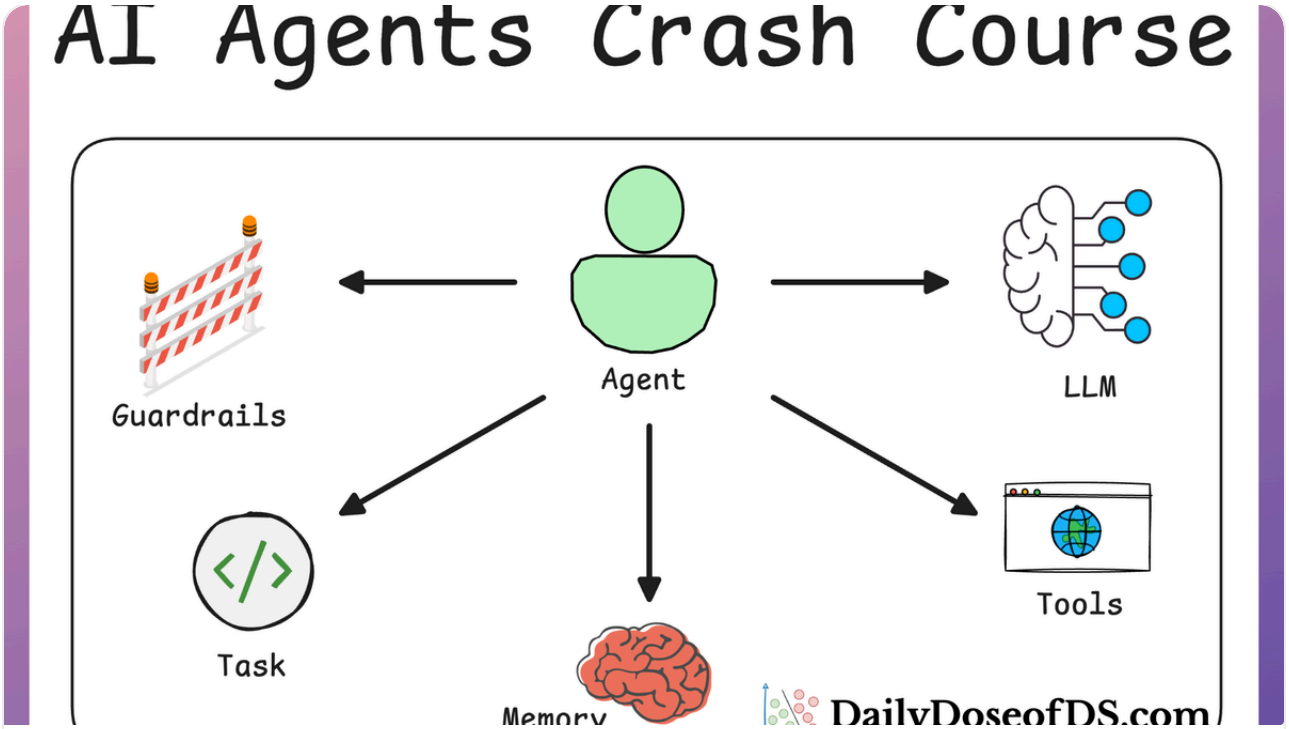
Let’s dive in.

A quick note

In case you are new here...

Over the past seven parts of this crash course, we have progressively built our understanding of Agentic Systems and multi-agent workflows with CrewAI.

- In Part 1, we covered the fundamentals of Agentic systems, understanding how AI agents can act autonomously to perform structured tasks.

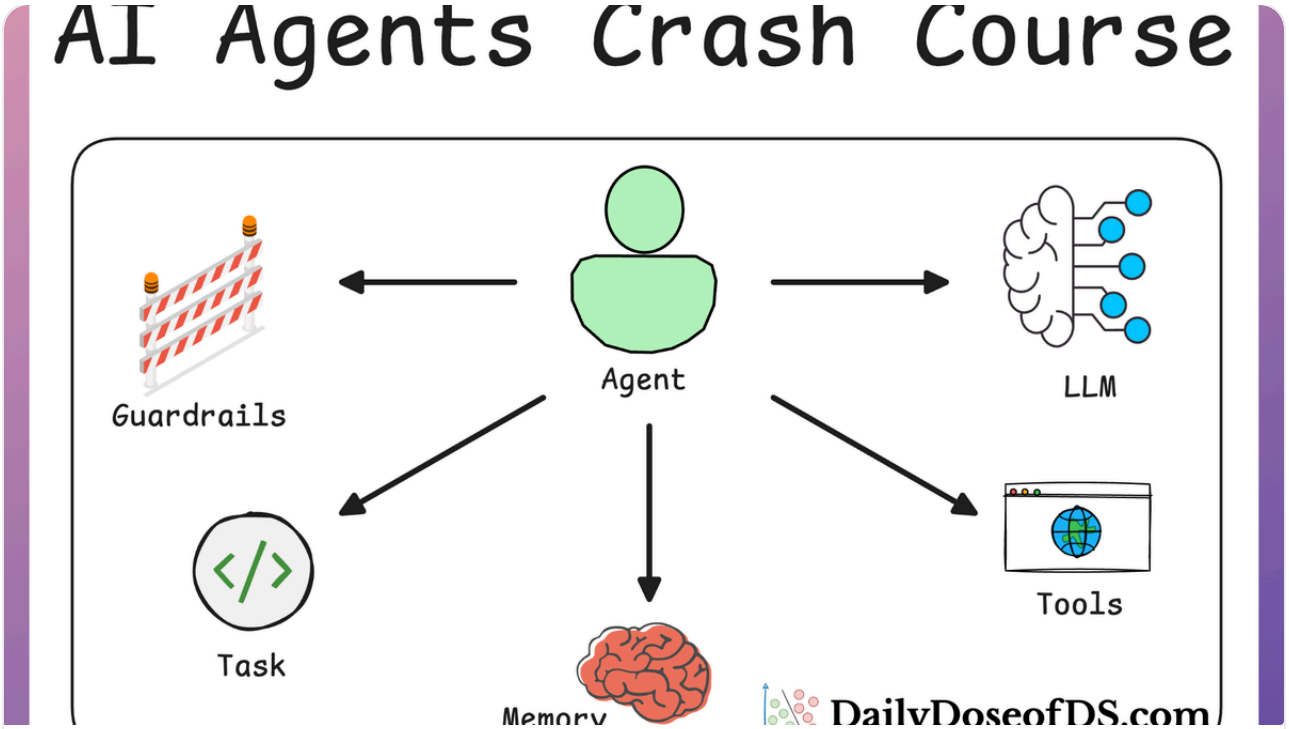


AI Agents Crash Course—Part 1 (With Implementation)

A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

- In Part 2, we explored how to extend Agent capabilities by integrating custom tools, using structured tools, and building modular Crews to compartmentalize responsibilities.

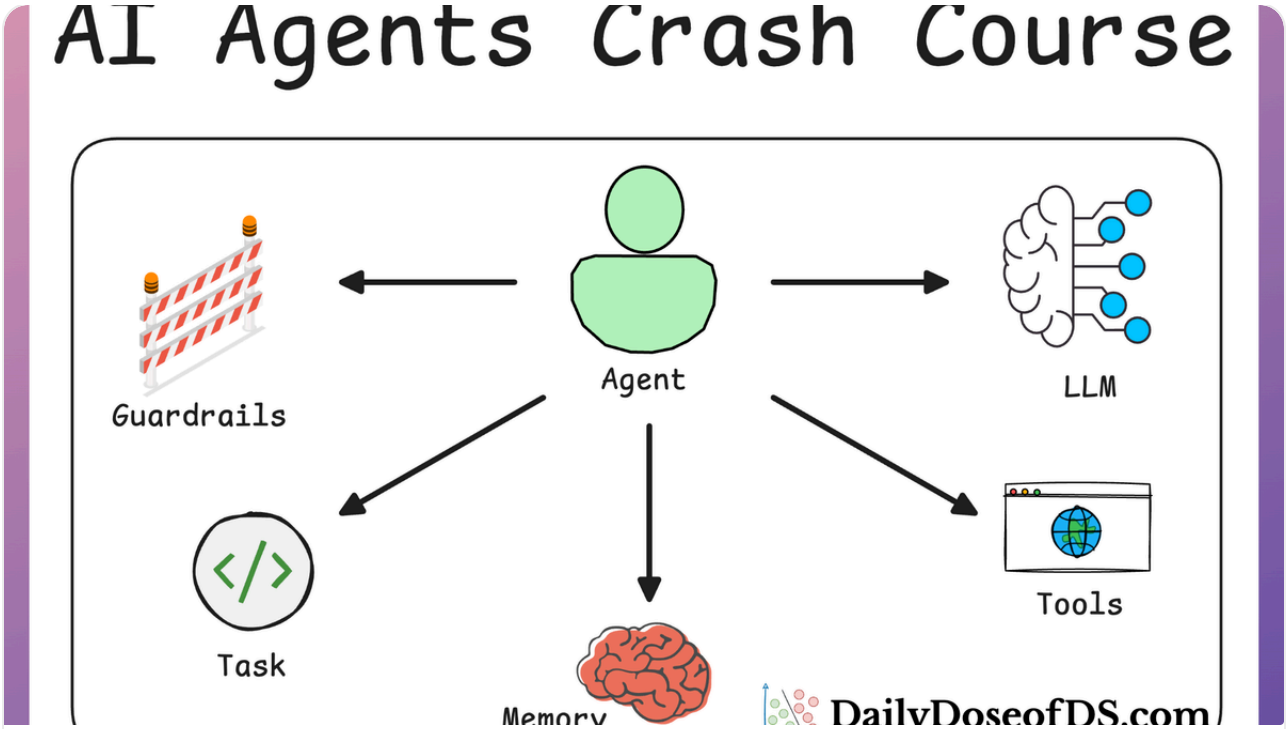


AI Agents Crash Course—Part 2 (With Implementation)

A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

- In Part 3, we focused on Flows, learning about state management, flow control, and integrating a Crew into a Flow. As discussed last time, with Flows, you can create structured, event-driven workflows that seamlessly connect multiple tasks, manage state, and control the flow of execution in your AI applications.

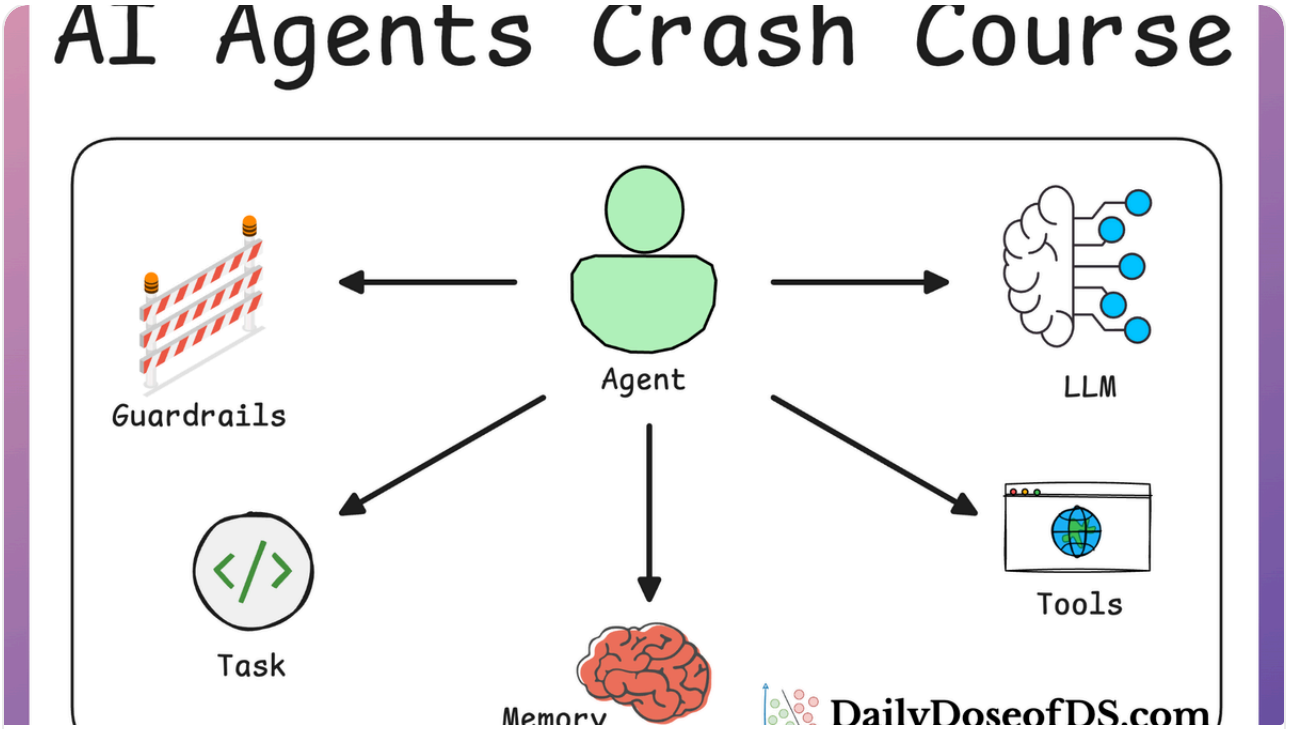


AI Agents Crash Course—Part 3 (With Implementation)

A deep dive into implementing Flows for building robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

- In Part 4, we extended these concepts into real-world multi-agent, multi-crew Flow projects, demonstrating how to automate complex workflows such as content planning and book writing.



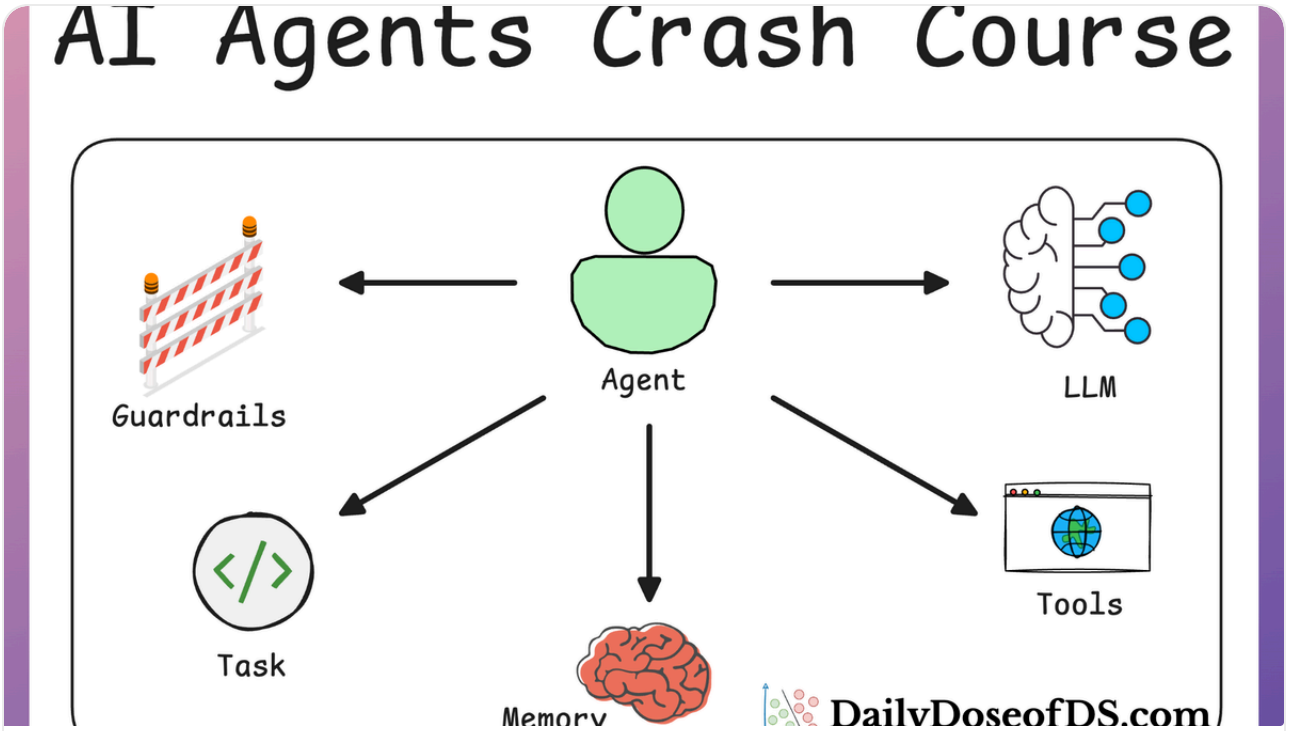
AI Agents Crash Course—Part 4 (With Implementation)

A deep dive into implementing Flows for building robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

- In Part 5 and 6, we moved into advanced techniques that make AI agents more robust, dynamic, and adaptable.
    - Guardrails → Enforcing constraints to ensure agents produce reliable and expected outputs.
    - Referencing other Tasks and their outputs → Allowing agents to dynamically use previous task results.
    - Executing tasks async → Running agent tasks concurrently to optimize performance.
    - Adding callbacks → Allowing post-processing or monitoring of task completions.
    - Introduce human-in-the-loop during execution → Introducing human-in-the-loop mechanisms for validation and control.
    - Hierarchical Agentic processes → Structuring agents into sub-agents and multi-level execution trees for more complex workflows.

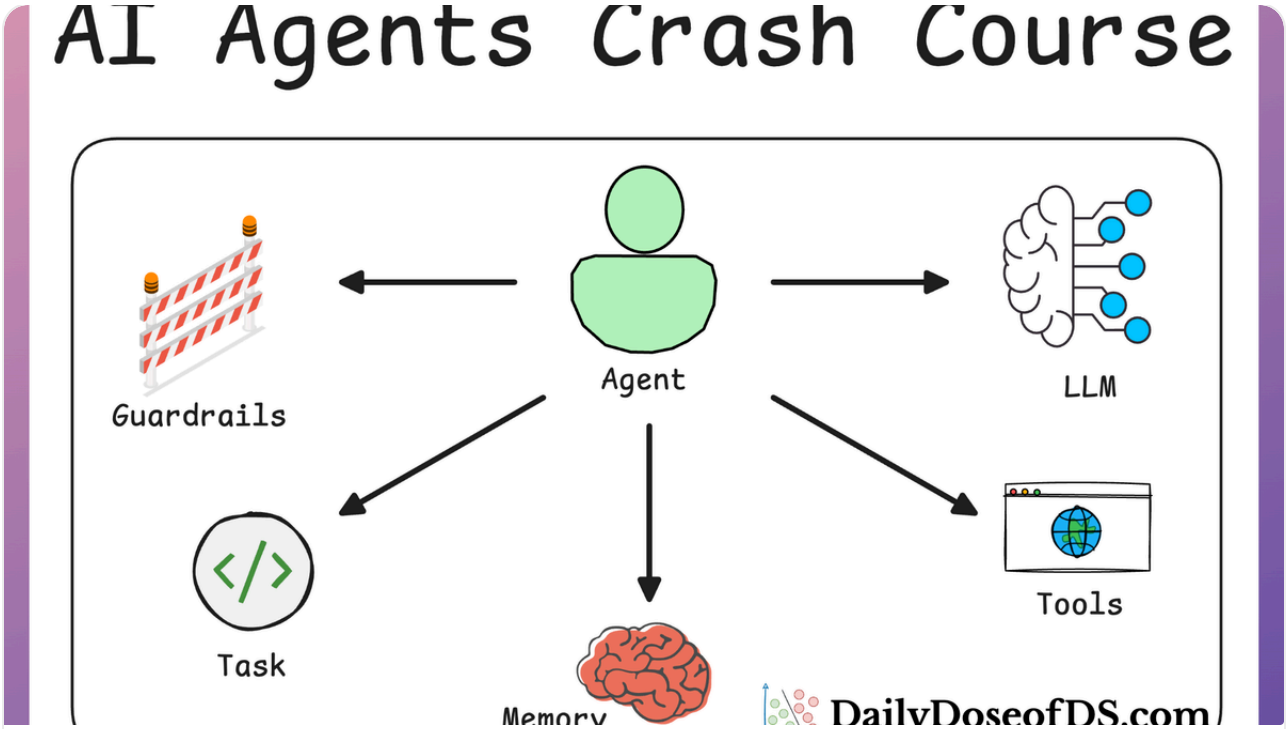
- Multimodal Agents → Extending CrewAI agents to handle text, images, audio, and beyond.
- and more.



AI Agents Crash Course—Part 5 (With Implementation)

A deep dive into advanced techniques to make robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

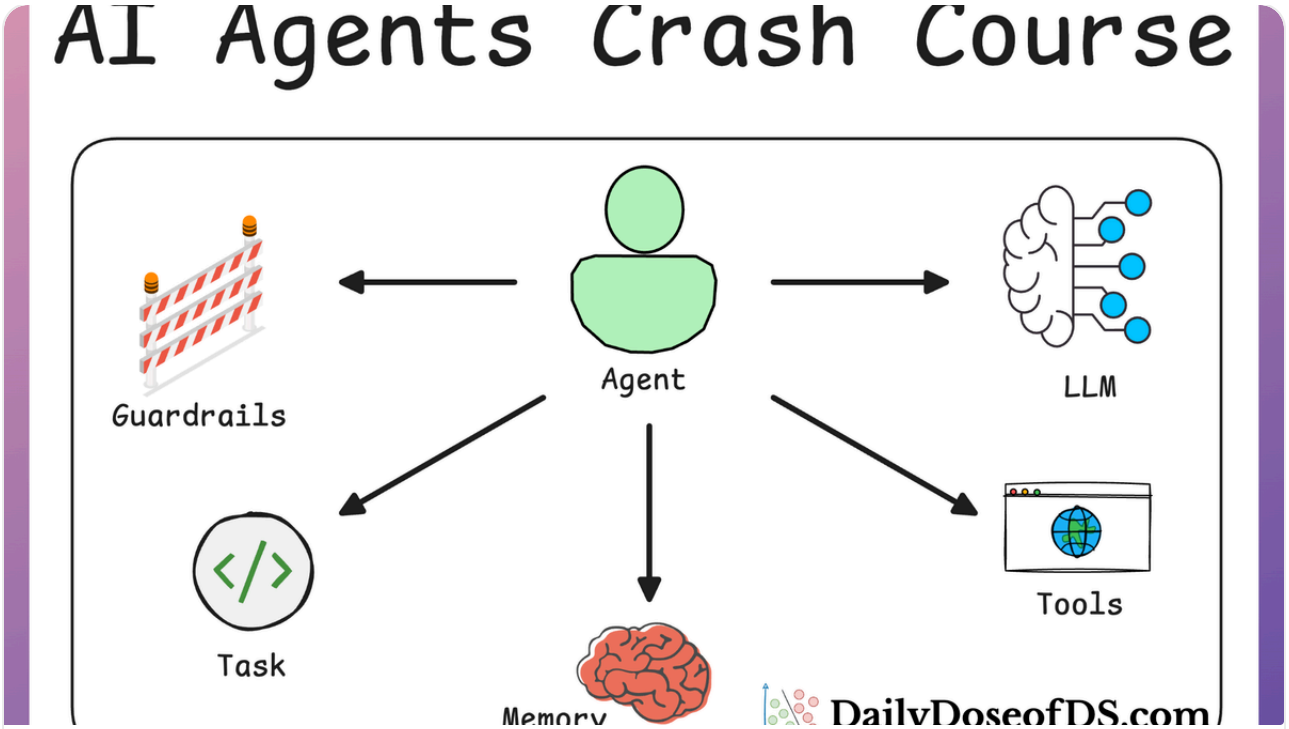


AI Agents Crash Course—Part 6 (With Implementation)

A deep dive into advanced techniques to make robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

- In Part 7, we learned about embedding Knowledge into agentic systems to augment them with external reference material. This was like giving an agent a library of context they can search and use while performing a task.



AI Agents Crash Course—Part 7 (With Implementation)

A deep dive into Knowledge for Agentic systems.

Daily Dose of Data Science • Avi Chawla

What is memory?

In an agentic system like CrewAI, memory is the mechanism that allows an AI agent to remember information from past interactions, ensuring continuity and learning over time.

This differs from an agent's knowledge and tools, which we have already discussed in previous parts.

- Knowledge usually refers to general or static information the agent has access to (like a knowledge base or facts from training data), whereas memory is contextual and dynamic—it's the data an agent stores during its operations (e.g. conversation history, user preferences).
- Tools, on the other hand, let an agent fetch or calculate information on the fly (e.g. web search or calculators) but do not inherently remember those results

for future queries. Memory fills that gap by retaining relevant details the agent can draw upon later, beyond what's in its static knowledge.

But why does memory even matter?

Memory enables agents to maintain continuity across multi-step tasks or conversations, personalize responses based on user-specific details, and adapt by learning from experience.

Think of it this way.

Imagine you have an agentic system deployed in production.

If this system is running without memory, every interaction is 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.

This leads to several problems:

- Users have to repeat themselves constantly.
- Agents lose context between steps of a multi-turn task.
- Personalization becomes impossible.
- Agents can't "learn" from their past experiences, even within the same session.

In contrast, with memory, your agent becomes *context-aware*.

It can remember facts like:

- "You like to eat Pizza without capsicum."
- "This ticket is about login errors in the mobile app."

- “We’ve already generated a draft for this topic.”
- “The user rejected the last suggestion. Don’t propose it again.”

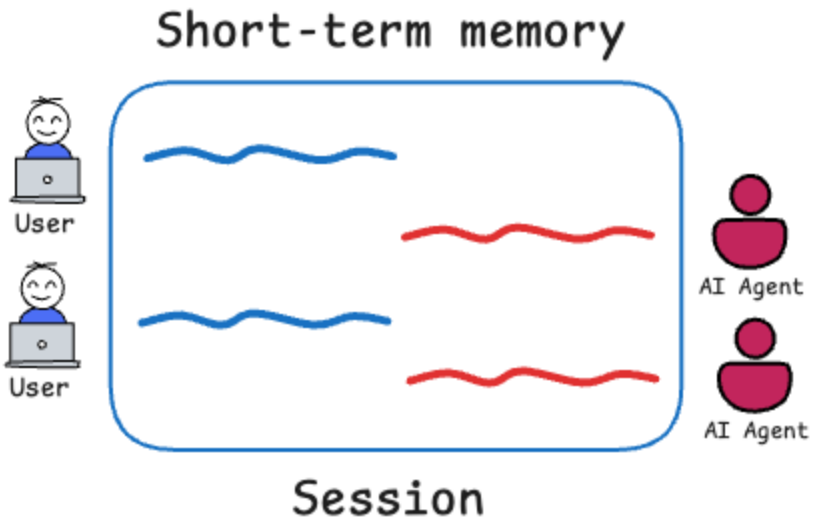
In other words, an agent with memory can recall what you asked or told it earlier, or remember your name and preferences in a subsequent session.

These are several key benefits of integrating a robust memory system into Agents:

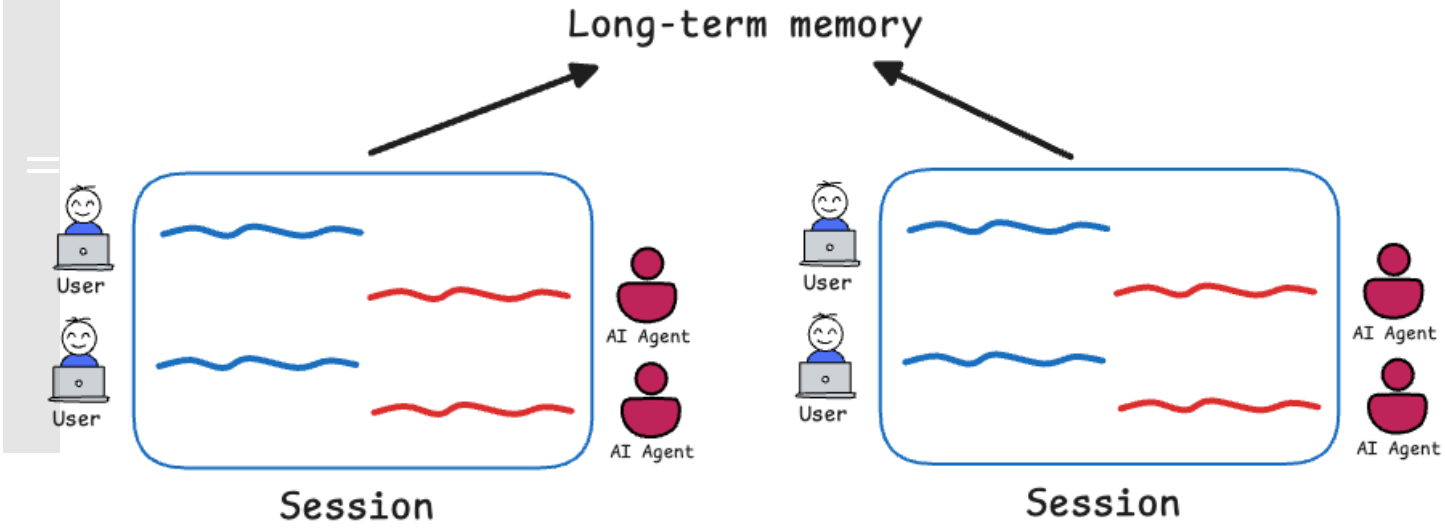
- Context retention: The agent can carry on a coherent dialogue or workflow, referring back to earlier parts of the conversation without re-providing all details. This makes multi-turn interactions more natural and consistent.
- Personalization: The agent can store user-specific information (like a user’s name, past queries, or preferences) and use it to tailor future responses for that particular user.
- Continuous learning: By remembering outcomes and facts from previous runs, the agent accumulates experience. Over time, it can improve decision-making or avoid repeating mistakes by referencing what it learned earlier.
- Collaboration: In a multi-agent “crew,” memory allows one agent to leverage information discovered by another agent in a prior step, improving teamwork efficiency.

Talking specifically about CrewAI, it provides a structured memory architecture with several built-in types of memory (we'll discuss each of them shortly in detail):

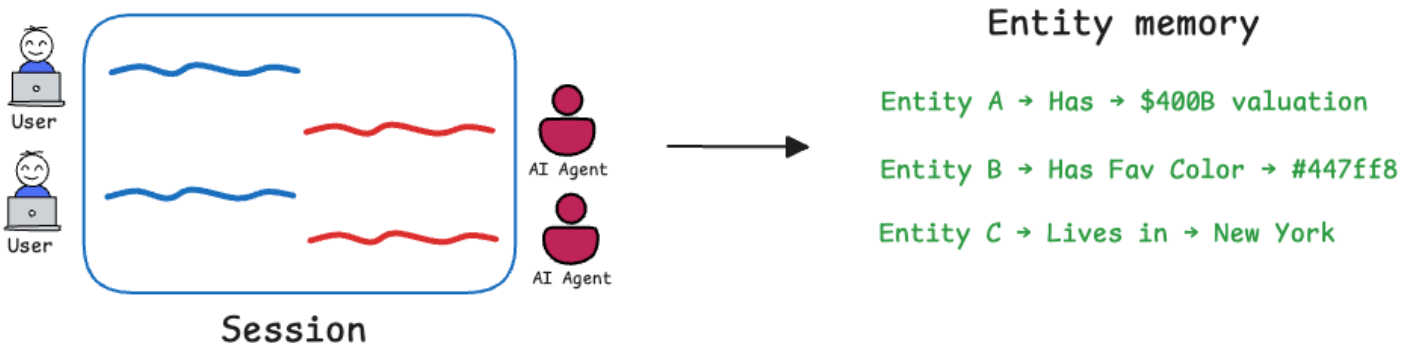
- Short-Term Memory for maintaining immediate context and coherence within a session.



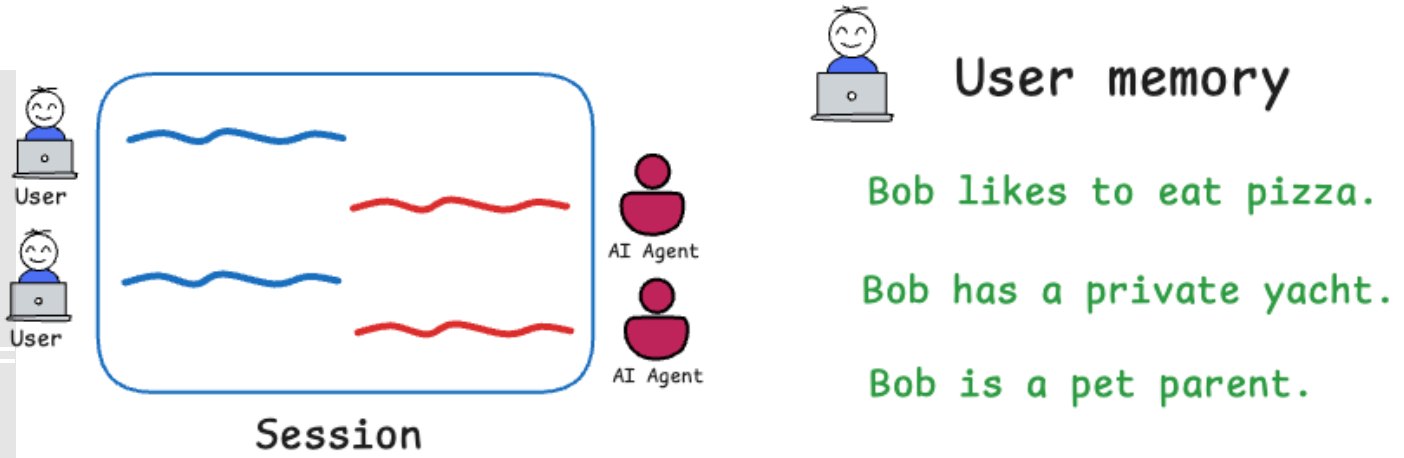
- Long-Term Memory for accumulating knowledge and experience across sessions.



- Entity Memory for tracking information about specific entities and ensuring consistency when those entities are referenced later.



- User Memory for personalization, keeping track of individual users' details so interactions can be tailored.



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

Unlike simply increasing an LLM's context window, these memory systems use efficient storage and retrieval mechanisms to extend an agent's effective recall.

Below, we'll explore each memory type in CrewAI, learn how to enable and use them in your Agents and walk through code examples with step-by-step explanations.

By the end, you should understand how to structure memory-aware agents for production use, and how to choose the right type of memory for your needs.

Installation and setup

💡 Feel free to skip this part if you have followed these instructions before.

Throughout this crash course, we have been using CrewAI, an open-source framework that makes it seamless to orchestrate role-playing, set goals, integrate tools, bring any of the popular LLMs, etc., to build autonomous AI agents.

GitHub - crewAIInc/crewAI: Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling complex tasks.

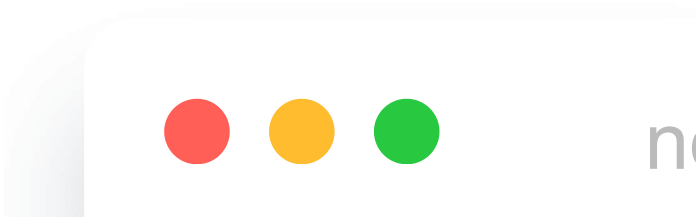
Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling...

GitHub • crewAIInc

To highlight more, CrewAI is a standalone independent framework without any dependencies on Langchain or other agent frameworks.

Let's dive in!

To get started, install CrewAI as follows:

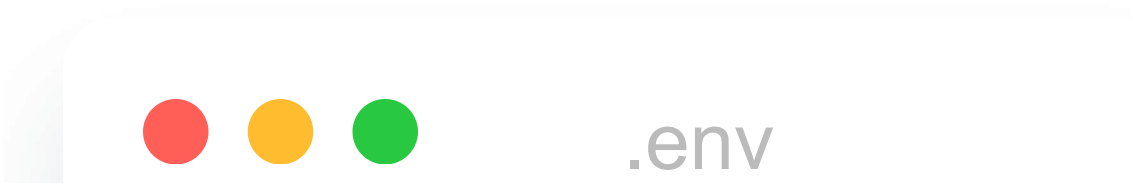


Like the RAG crash course, we shall be using Ollama to serve LLMs locally. That said, CrewAI integrates with several LLM providers like:

- OpenAI
- Gemini
- Groq
- Azure
- Fireworks AI
- Cerebras
- SambaNova
- and many more.

> 💡 If you have an OpenAI API key, we recommend using that since the outputs may not make sense at times with weak LLMs. If you don't have an API key, you can get some credits by creating a dummy account on OpenAI and use that instead. If not, you can continue reading and use Ollama instead but the outputs could be poor in that case.

To set up OpenAI, create a `.env` file in the current directory and specify your OpenAI API key as follows:



Memory demo

Now that we understand the use of memory and what it does, let's start with a simple demo to build practical foundations as well.

> 💡 Make sure you have created a `.env` file with the `OPENAI_API_KEY` specified in it. It will make things much easier and faster for you. Also, add these two lines of code to handle asynchronous operations within a Jupyter Notebook environment, which will allow us to make asynchronous calls smoothly to your Crew Agent.

Once that's done, we start by defining a simple single-agent Crew which acts like a Personal Assistant:

notebook.ipynb

```
from crewai import Crew, Agent, Task, Process

# Define a simple agent (assuming a default LLM agent)
assistant = Agent(role="Personal Assistant",
                  goal="""You are a personal assistant that can
                          help the user with their tasks.""",
                  backstory="""You are a personal assistant that
                               can help the user with their tasks.""",
                  verbose=True)

task = Task(description="Handle this task: {user_task}",
            expected_output="A clear and concise answer to the question.",
            agent=assistant)

# Create a crew with memory enabled
crew = Crew(
    agents=[assistant],
    tasks=[task],
    process=Process.sequential,
    verbose=True
)
```

In the above code:

- We have an Agent, whose goal is to the help the user with their tasks.
- We have a Task, which is to handle the `{user_task}`.
- Lastly, while defining the Crew just like we learned in the previous parts of the crash course.

Next, we kick-off the Crew as follows:

notebook.ipynb

```
user_input = """My favorite color is #46778F and
                my favorite Agent framework is CrewAI."""

result = crew.kickoff(inputs={"user_task": user_input})
```

This produces the following verbose output:

```
# Agent: Personal Assistant
## Task: Handle this task: My favorite color is #46778F and
                my favorite Agent framework is CrewAI.


# Agent: Personal Assistant
## Final Answer:
Your favorite color is #46778F and your favorite Agent framework is CrewAI.
```

Now given that as a user, I have already told about my preference so if I invoke this Crew Agent and ask it about my favorite color, it should be able to answer about it. Let's see if that is the case:

notebook.ipynb

```
user_input = "What is my favorite color?"

result = crew.kickoff(inputs={"user_task": user_input})
```

Running the above code, we get this output:

```
result = crew.kickoff(inputs={"user_task": "What is my favorite color?"})
pprint(result.raw)
```

✓ 1.3s

```
# Agent: Personal Assistant
## Task: Handle this task: What is my favorite color?


# Agent: Personal Assistant
## Final Answer:
Your favorite color is not explicitly stated in our conversation. If you let me know


('Your favorite color is not explicitly stated in our conversation. If you let '
 'me know your favorite color, I can assist you further with related tasks or '
 'inquiries. Please provide your favorite color for a more tailored response!')
```

This shows that one specific execution of a Crew gets to have no visibility on any of the previous executions of the same Crew.

Memory fixes it.

To do this, we define the same simple single-agent Crew, which acts like a Personal Assistant but this time with a small change as highlighted below (it is advised to re-start the notebook before running this code):

notebook.ipynb

```python
from crewai import Crew, Agent, Task, Process

# Define a simple agent (assuming a default LLM agent)
assistant = Agent(role="Personal Assistant",
                  goal="""You are a personal assistant that can
                          help the user with their tasks.""",
                  backstory="""You are a personal assistant that
                               can help the user with their tasks.""",
                  verbose=True)

task = Task(description="Handle this task: {user_task}",
            expected_output="A clear and concise answer to the question.",
            agent=assistant)

# Create a crew with memory enabled
crew = Crew(
    agents=[assistant],
    tasks=[task],
    process=Process.sequential,
    memory=True,
    verbose=True
)
```

Enable memory for Crew

In the above code:

- We have an Agent, whose goal is to the help the user with their tasks.
- We have a Task, which is to handle the `{user_task}`.
- Lastly, while defining the Crew, we have enabled the `memory=True` flag. Technically, this is all we need to do to enable memory for Agents. But the internal details of it are worth learning which we shall discuss in a bit. For now, let's focus on executing this Crew.

Next, we kick-off the Crew as follows:

notebook.ipynb

```
user_input = """My favorite color is #46778F and
                my favorite Agent framework is CrewAI."""

result = crew.kickoff(inputs={"user_task": user_input})
```

```
# Agent: Personal-Assistant
## Task: Handle this task: My favorite color is #46778F and
                my favorite Agent framework is CrewAI.


# Agent: Personal-Assistant
## Final Answer:
Your favorite color is #46778F, and your favorite Agent framework is CrewAI.
```

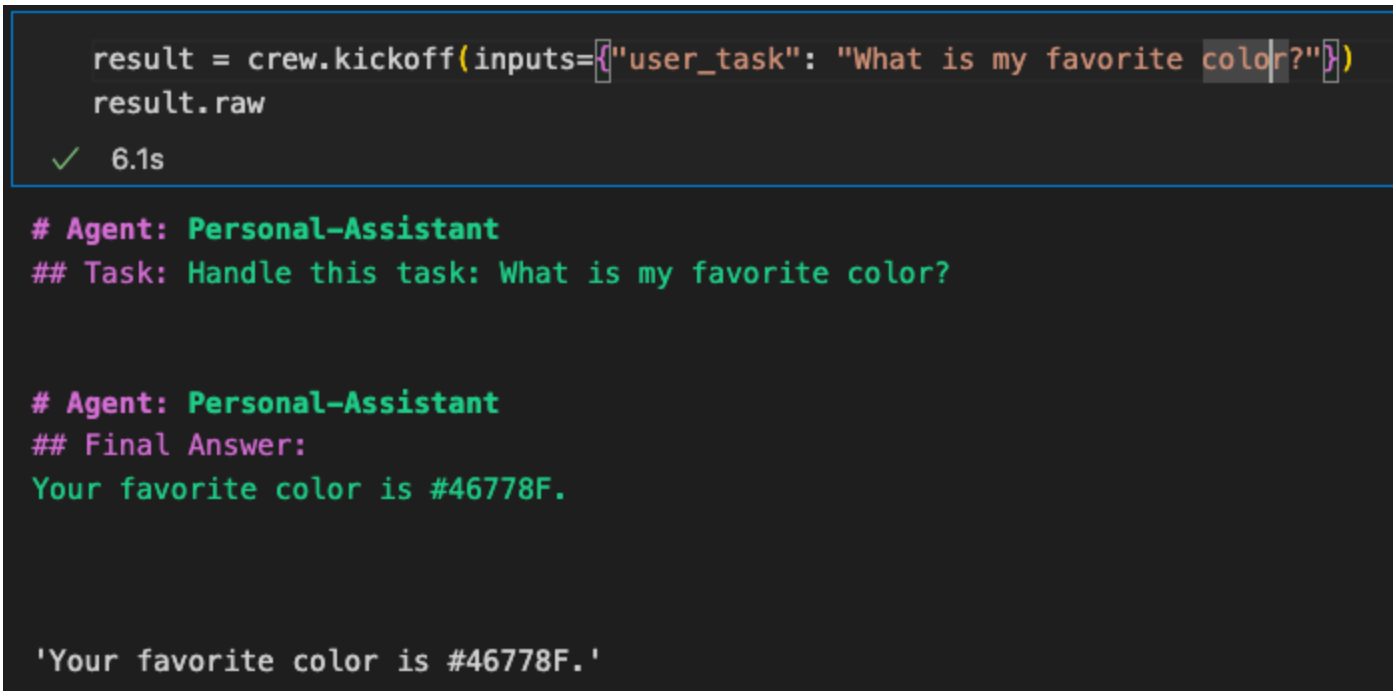
Now given that memory is enable and I have already told about my preference so if I invoke this Crew Agent and ask it about my favorite color, it should be able to answer about it. Let's see if that is the case this time:

notebook.ipynb

```
user_input = "What is my favorite color?"

result = crew.kickoff(inputs={"user_task": user_input})
```

Running the above code, we get this output:



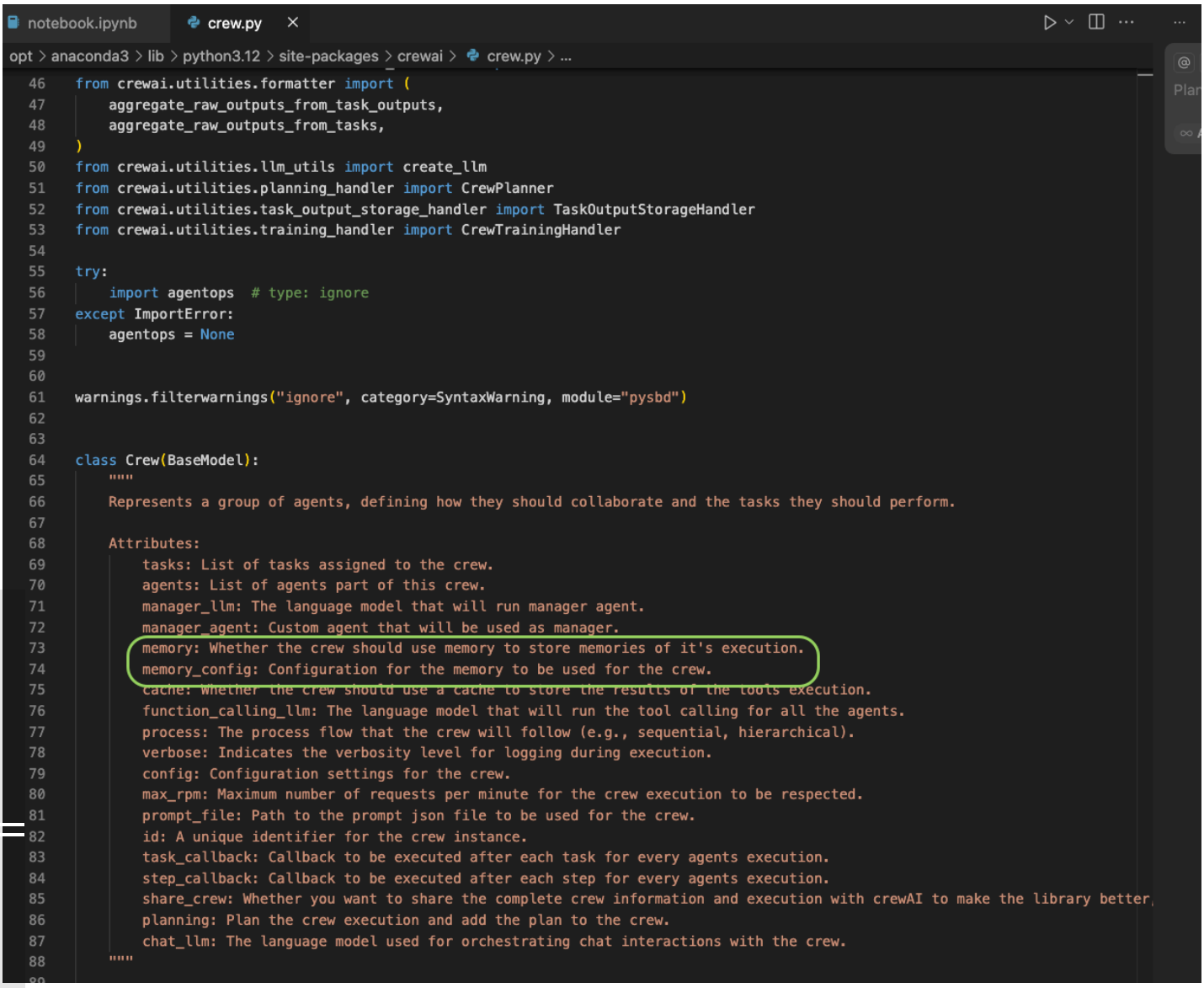
Great! This time, our simple Agentic system is able to access information from previous executions.

And it would be obvious to infer that memory helped it do so.

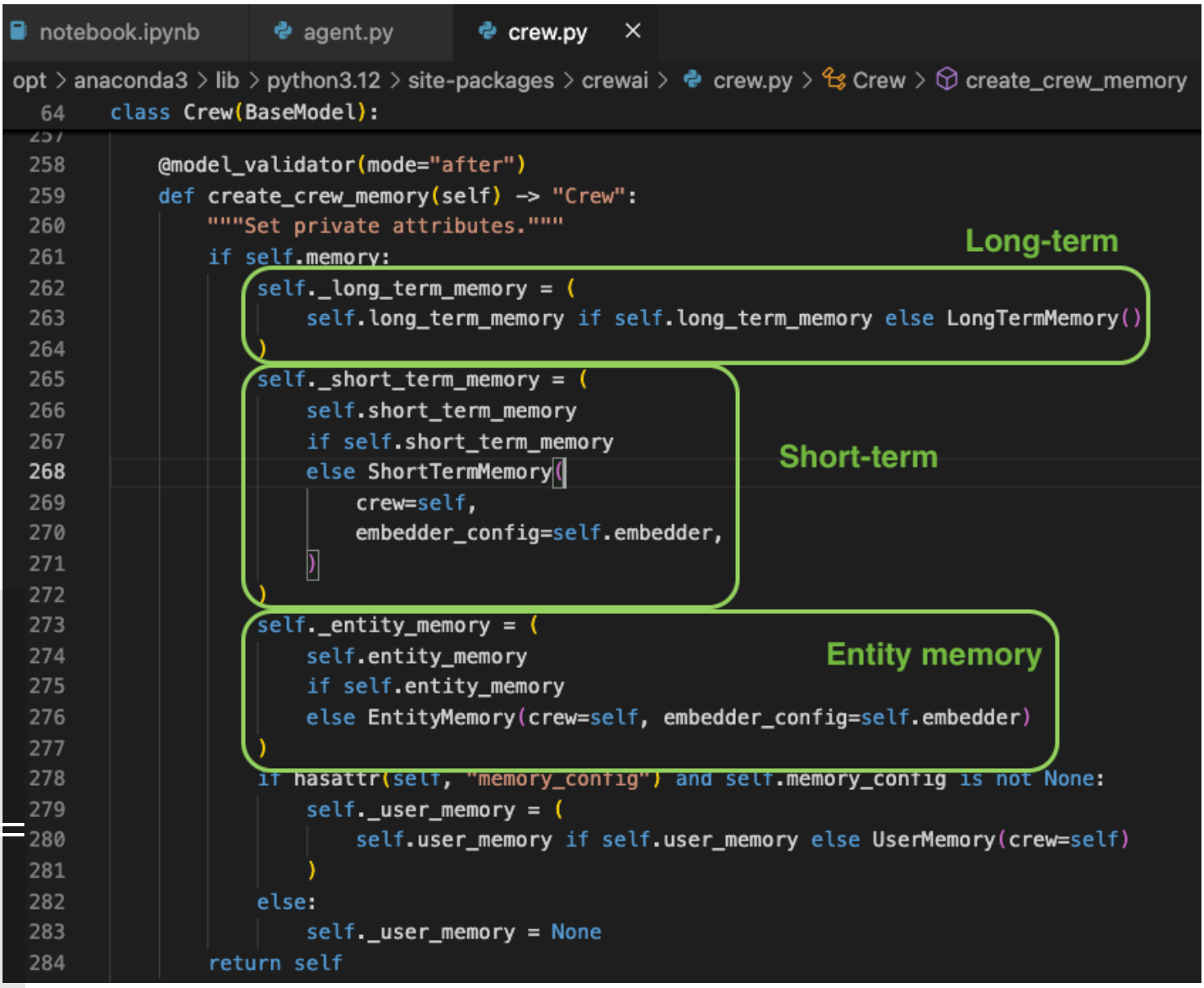
Now, let's dive into technical details and understand what happens under the hood.

To do this, navigate to the source code of the `Crew` class as follows (hover on Crew class, press `Cmd` button and click it):

This will open the crew.py file where the Crew class is implemented and within the docstring, you will find some memory-related attributes, as shown below:



Within this class, you will find the `create_crew_memory` method:



- Line 261: It checks if the `memory` parameter was set to `True` (which is the case in our most recent Crew definition).
- If yes:
  - Line 262-264 (first green block): This defines some sort of long-term memory for the Agent.
  - Line 265-272 (second green block): This defines some sort of short-term memory for the Agent.
  - Line 273-277 (third green block): This defines some sort of entity memory for the Agent.

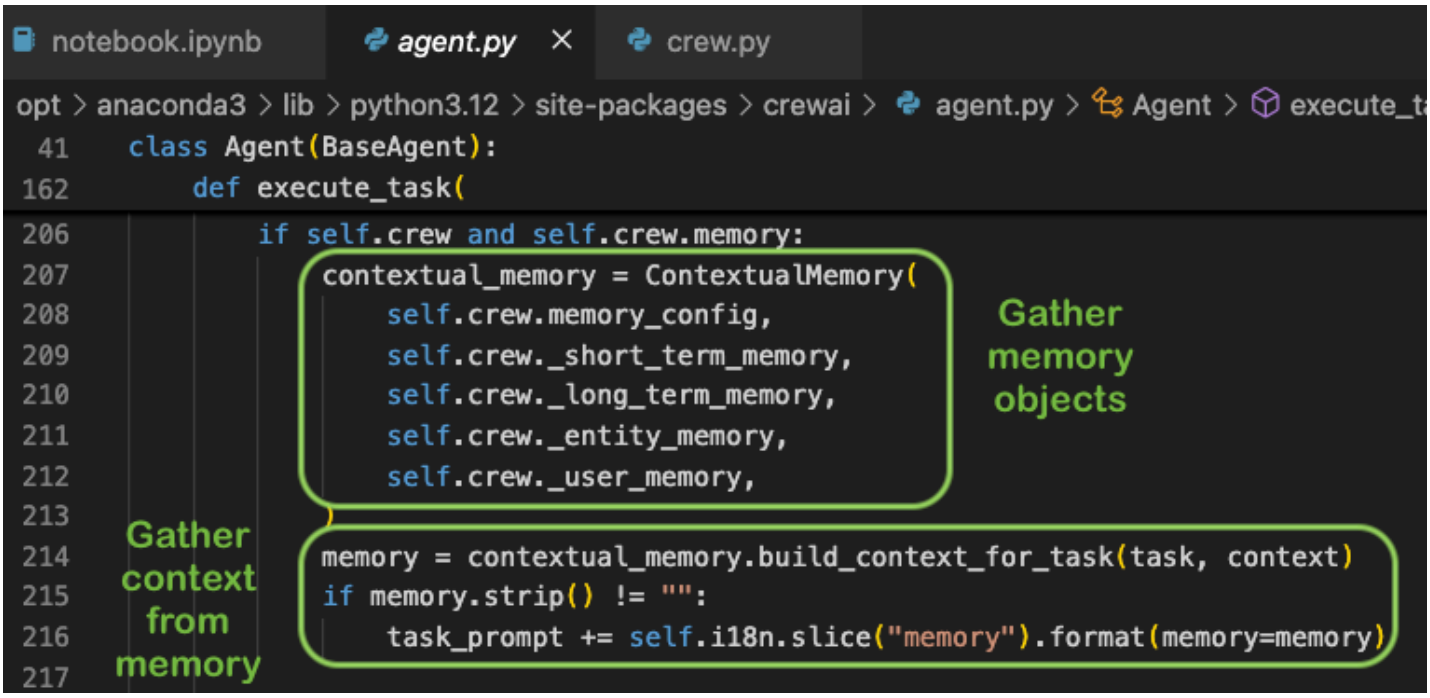
We shall discuss each one of them shortly.

But before we do that, let's keep this a bit abstract and assume there is some sort of long-term memory, short-term memory and entity memory that our Agent can access.

Moving on, just like you navigated to the `Crew` class, navigate to the `Agent` class:

Here, find the `execute_task` method, which, as the name suggests, is responsible for executing a task with the agent.

Within this, you will see a mention of memory:

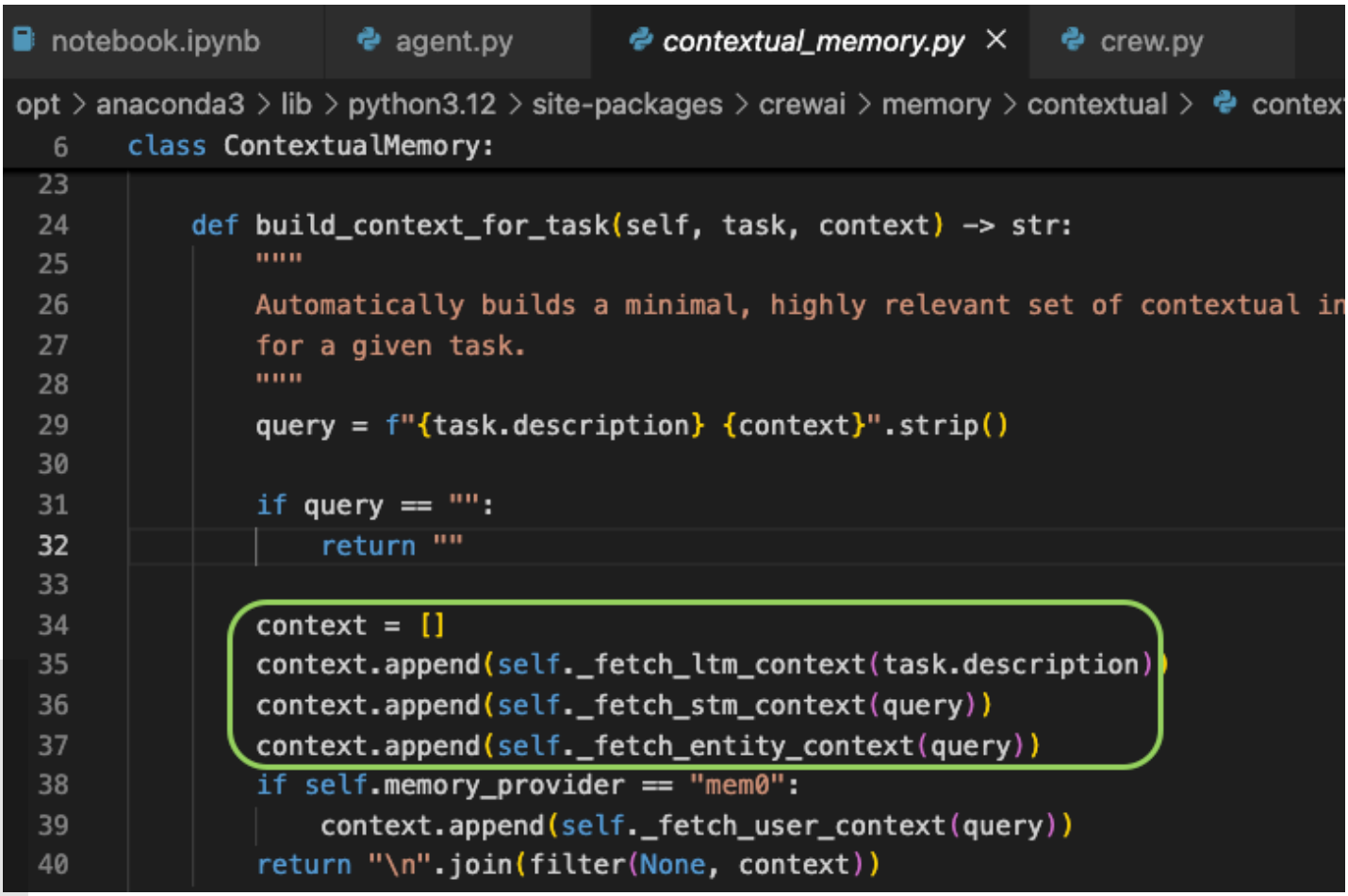


Since the `execute_task` method will only run after a Crew has been defined, the above code shows this:

- Line 207-212: First, retrieve the memory objects of the Crew.
- Line 214: Use the memory objects to invoke the `build_context_for_task` method, which, as the name suggests, finds relevant context for the task using the information in the memory.
- Line 215-216: If the relevant context exists (`if memory`), it is added to the `task_prompt` object.

We can now inspect this further by checking the `build_context_for_task` method.

Follow the same steps as before—hover on the `build_context_for_task` in the `agent.py` file, press `Cmd` and click. This takes us to the implementation of the `build_context_for_task` method:



- Line 35: context is gathered from the long-term memory (`ltm` in the `_fetch_ltm_context` method stands for Long-term Memory). This just uses the task description to query the long-term memory, which shows that long-term memory focuses on broad ideas and context.
- Line 36: context is gathered from the short-term memory (`stm` in the `_fetch_stm_context` method stands for short-term memory). This uses the query, which shows that unlike the long-term memory, short-term memory is much more specific to the query.
- Line 37: context is gathered from the entity memory. This uses the query, which shows that just like short-term memory, entity memory is also much more specific to the query.

Right below the `build_context_for_task` method, you will find the `_fetch_stm_context` method:

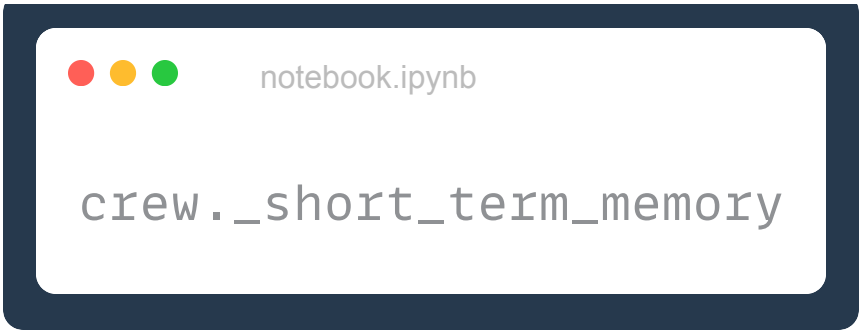
notebook.ipynb | agent.py | contextual_memory.py | crew.py

t > anaconda3 > lib > python3.12 > site-packages > crewai > memory > contextual > contextual_memory.py > Conte

```
6   class ContextualMemory:
42      def _fetch_stm_context(self, query) -> str:
47          stm_results = self.stm.search(query)
48          formatted_results = "\n".join(
49              [
50                  f"- {result['memory'] if self.memory_provider == 'mem0' else result['context']}"
51                  for result in stm_results
52              ]
53          )
54          return f"Recent Insights:\n{formatted_results}" if stm_results else ""
```

- Line 47: It invokes some search method to query the short-term memory.
- Line 48-54: The retrieved results are formatted as a string and returned.

This makes things quite interesting since now, we can invoke the search method on the `_short_term_memory` attribute of our Crew to get the short-term memory object:



This produces the following output:

Let's invoke the `search` method on this short-term memory object:

notebook.ipynb

```
user_input = "What is my favorite color?"

crew._short_term_memory.storage.search(user_input)
```

This produces the following output:

- There are two match results, and we can see some context being retrieved from our earlier message, where we told the Agent about our preference.
- This context is passed to the Agent during the second execution, which makes it easy for the Agent to "remember" what was mentioned earlier and produce a response:

```
result = crew.kickoff(inputs={"user_task": "What is my favorite color?"})
result.raw
```

✓ 6.1s

```
# Agent: Personal-Assistant
## Task: Handle this task: What is my favorite color?


# Agent: Personal-Assistant
## Final Answer:
Your favorite color is #46778F.



'Your favorite color is #46778F.'
```

Now, let's do the same on the `long_term_memory` object, but this time, we need to pass the `task.description` as a parameter as discussed earlier (and shared again in the image below):

notebook.ipynb | agent.py | contextual_memory.py | crew.py

opt > anaconda3 > lib > python3.12 > site-packages > crewai > memory > contextual > contex

```
6   class ContextualMemory:
23
24      def build_context_for_task(self, task, context) -> str:
25          """
26          Automatically builds a minimal, highly relevant set of contextual in
27          for a given task.
28          """
29          query = f"{task.description} {context}".strip()
30
31          if query == "":
32              return ""
33
34          context = []
35          context.append(self._fetch_ltm_context(task.description))
36          context.append(self._fetch_stm_context(query))
37          context.append(self._fetch_entity_context(query))
38          if self.memory_provider == "mem0":
39              context.append(self._fetch_user_context(query))
40          return "\n".join(filter(None, context))
```

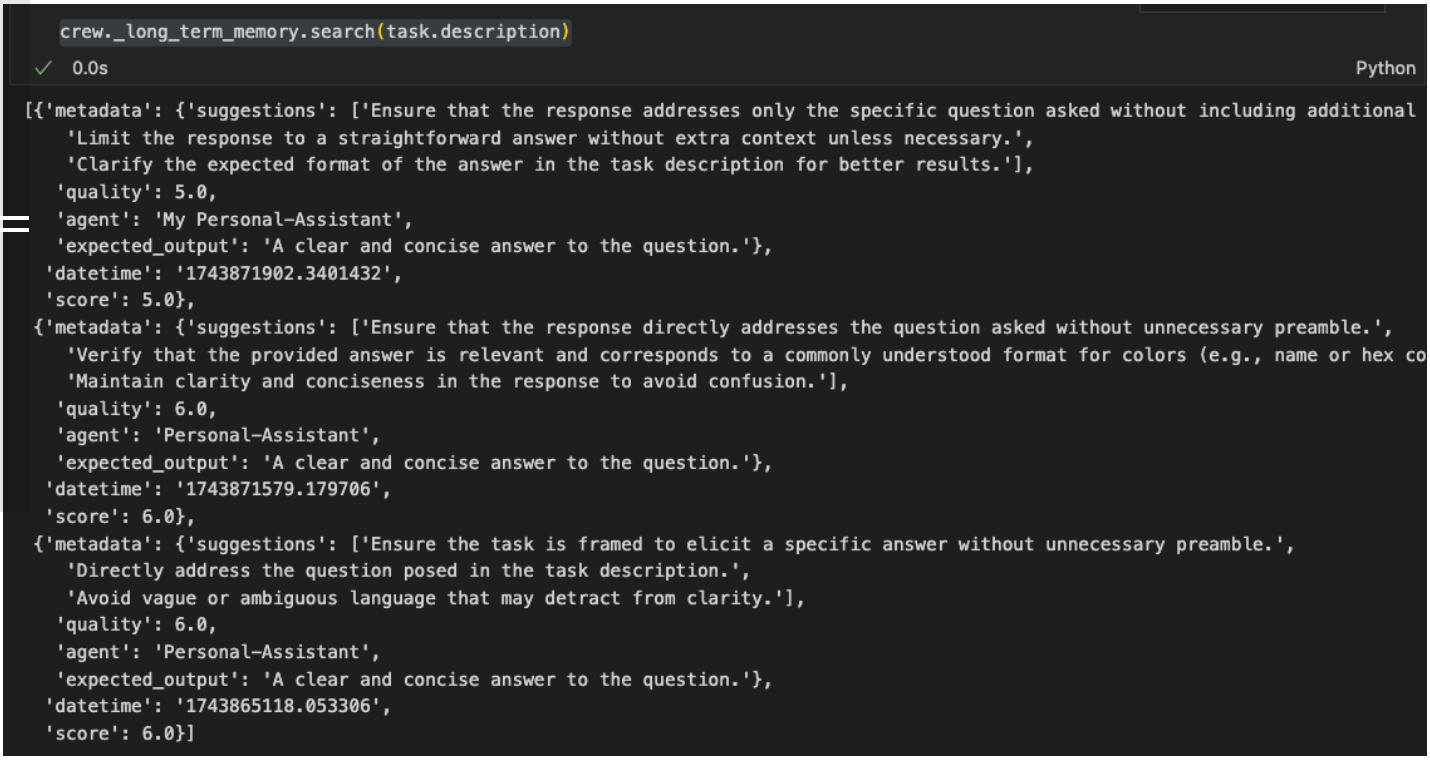
Line 35: Long-term context requires the task description as the argument.

Below, we invoke the search method of the `long_term_memory` object:

notebook.ipynb

```
crew._long_term_memory.search(task.description)
```

And this produces the following output:



```
crew._long_term_memory.search(task.description)
✓ 0.0s                                                                    Python
[{'metadata': {'suggestions': ['Ensure that the response addresses only the specific question asked without including additional
    'Limit the response to a straightforward answer without extra context unless necessary.',
    'Clarify the expected format of the answer in the task description for better results.'],
   'quality': 5.0,
   'agent': 'My Personal-Assistant',
   'expected_output': 'A clear and concise answer to the question.'},
  'datetime': '1743871902.3401432',
  'score': 5.0},
 {'metadata': {'suggestions': ['Ensure that the response directly addresses the question asked without unnecessary preamble.',
    'Verify that the provided answer is relevant and corresponds to a commonly understood format for colors (e.g., name or hex co
    'Maintain clarity and conciseness in the response to avoid confusion.'],
   'quality': 6.0,
   'agent': 'Personal-Assistant',
   'expected_output': 'A clear and concise answer to the question.'},
  'datetime': '1743871579.179706',
  'score': 6.0},
 {'metadata': {'suggestions': ['Ensure the task is framed to elicit a specific answer without unnecessary preamble.',
    'Directly address the question posed in the task description.',
    'Avoid vague or ambiguous language that may detract from clarity.'],
   'quality': 6.0,
   'agent': 'Personal-Assistant',
   'expected_output': 'A clear and concise answer to the question.'},
  'datetime': '1743865118.053306',
  'score': 6.0}]
```

The result we get is a list of memory entries that the agent has previously stored which are relevant to that input. Each item in the list contains:

- `metadata`: A dictionary of structured memory about past interactions. This memory snippet stores not just what the task was about, but also the reflections, feedback, or insights learned from that task. For example:

notebook.ipynb

```
{'metadata':
    {'suggestions': ["""Ensure that the response addresses only
                        the specific question asked without
                        including additional unrelated information.""",
                     """Limit the response to a straightforward
                        answer without extra context unless necessary.""",
                     """Clarify the expected format of the answer
                        in the task description for better results."""],
     'quality': 5.0,
     'agent': 'Personal Assistant',
     'expected_output': 'A clear and concise answer to the question.'
 },
 ...
}
```

- `datetime`: A timestamp indicating when this memory was logged. This is important since depending on the use-case, it may be important to fetch the most recent information.
- `score`: A numeric score representing how well this memory matched your query (`task.description` in this case). The higher the score, the more relevant CrewAI believes this past memory is.

Comparing this to short-term memory, while short-term memory is typically tied to the current session and shorter conversations, long-term memory stores explicit learnings and structured outcomes across runs.

In this case, each memory entry might be from:

- A previous time the agent ran a similar task.
- A reflection stored from a post-task callback.
- A result that was intentionally added to long-term memory.

Finally, let's do the same on the `entity_memory` object. Like the short-term memory example, we need to pass the query as a parameter as discussed earlier (and shared again in the image below):

notebook.ipynb | agent.py | contextual_memory.py | crew.py

opt > anaconda3 > lib > python3.12 > site-packages > crewai > memory > contextual > contex

```
6   class ContextualMemory:
23
24      def build_context_for_task(self, task, context) -> str:
25          """
26          Automatically builds a minimal, highly relevant set of contextual in
27          for a given task.
28          """
29          query = f"{task.description} {context}".strip()
30
31          if query == "":
32              return ""
33
34          context = []
35          context.append(self._fetch_ltm_context(task.description))
36          context.append(self._fetch_stm_context(query))
37          context.append(self._fetch_entity_context(query))
38          if self.memory_provider == "mem0":
39              context.append(self._fetch_user_context(query))
40          return "\n".join(filter(None, context))
```

Line 35: Long-term context requires the task description as the argument.

Below, we invoke the search method of the `entity_memory` object:

notebook.ipynb

```
user_input = "What is my favorite color?"

crew._entity_memory.search(user_input)
```

And this produces the following output:

```
crew._entity_memory.search("What is my favorite color?")
✓ 1.0s
[{'id': '837a6709-c24a-456a-99f3-ec4843447b74',
  'metadata': {'relationships': '- associated with personal preference'},
  'context': 'favorite color(Color): A color that is preferred by someone.',
  'score': 0.7638339235576107},
 {'id': '1bb1713f-575b-4e81-a081-b1a0f7ba3123',
  'metadata': {'relationships': "- Referenced in the description as the user's favorite color."},
  'context': 'favorite color(Color): The color preferred by the user.',
  'score': 0.8393476246969971},
 {'id': 'f980fc11-1d65-4049-8df3-0ff6e25d8918',
  'metadata': {'relationships': '- represents a specific color'},
  'context': '#46778F(Hex Color Code): A specific shade identified by the hex code #46778F.',
  'score': 1.2067459004405603}]
```

Entity memory is CrewAI's way of remembering facts and attributes about specific entities across tasks or sessions.

Think of it like a mini-knowledge graph or fact sheet for every person, object, or concept your agents interact with. While short-term memory helps retain recent interactions, and long-term memory stores reflections or learnings, entity memory is all storing:

- Who the user is
- What they like or dislike
- Known facts about objects, people, or products
- Traits or attributes tied to names, concepts, and identifiers
- and more.

For instance, in the below output, we got a list of relevant entity memories, each with:

```
crew._entity_memory.search("What is my favorite color?")
✓ 1.0s
[{'id': '837a6709-c24a-456a-99f3-ec4843447b74',
  'metadata': {'relationships': '- associated with personal preference'},
  'context': 'favorite color(Color): A color that is preferred by someone.',
  'score': 0.7638339235576107},
 {'id': '1bb1713f-575b-4e81-a081-b1a0f7ba3123',
  'metadata': {'relationships': "- Referenced in the description as the user's favorite color."},
  'context': 'favorite color(Color): The color preferred by the user.',
  'score': 0.8393476246969971},
 {'id': 'f980fc11-1d65-4049-8df3-0ff6e25d8918',
  'metadata': {'relationships': '- represents a specific color'},
  'context': '#46778F(Hex Color Code): A specific shade identified by the hex code #46778F.',
  'score': 1.2067459004405603}]
```

- A unique ID for the stored memory chunk or entity.
- `metadata`, which describes why this memory is relevant. For example:
  - `"associated with personal preference"`
  - `"represents a specific color"`
  - `"referenced as the user's favorite color"`
- `context`, which stores the actual textual content of the memory. In our case, examples include:
  - `"favorite color(Color): A color that is preferred by someone."`
  - `"#46778F (Hex Color Code): A specific shade identified by the hex code #46778F."`
  - This is the real meat of what the agent could refer to in response.
- Finally, a relevance score based on vector similarity—higher scores mean better matches.

Unlike general long-term memory, entity memory is structured around named relationships. For example:

- `"John's birthday is March 15th"`
- `"Product X has a 2-year warranty"`
- `"The user prefers dark mode"`

These relationships can be retrieved, updated, and reasoned over programmatically.

In part 9...

We think it's a good spot to end Part 8 of the AI Agents crash course.

So far, we've explored memory in a practical, hands-on way—by searching and retrieving information from short-term, long-term, and entity memory in CrewAI. You've seen how memory helps agents remember facts, reflect on past actions, and stay context-aware across sessions.

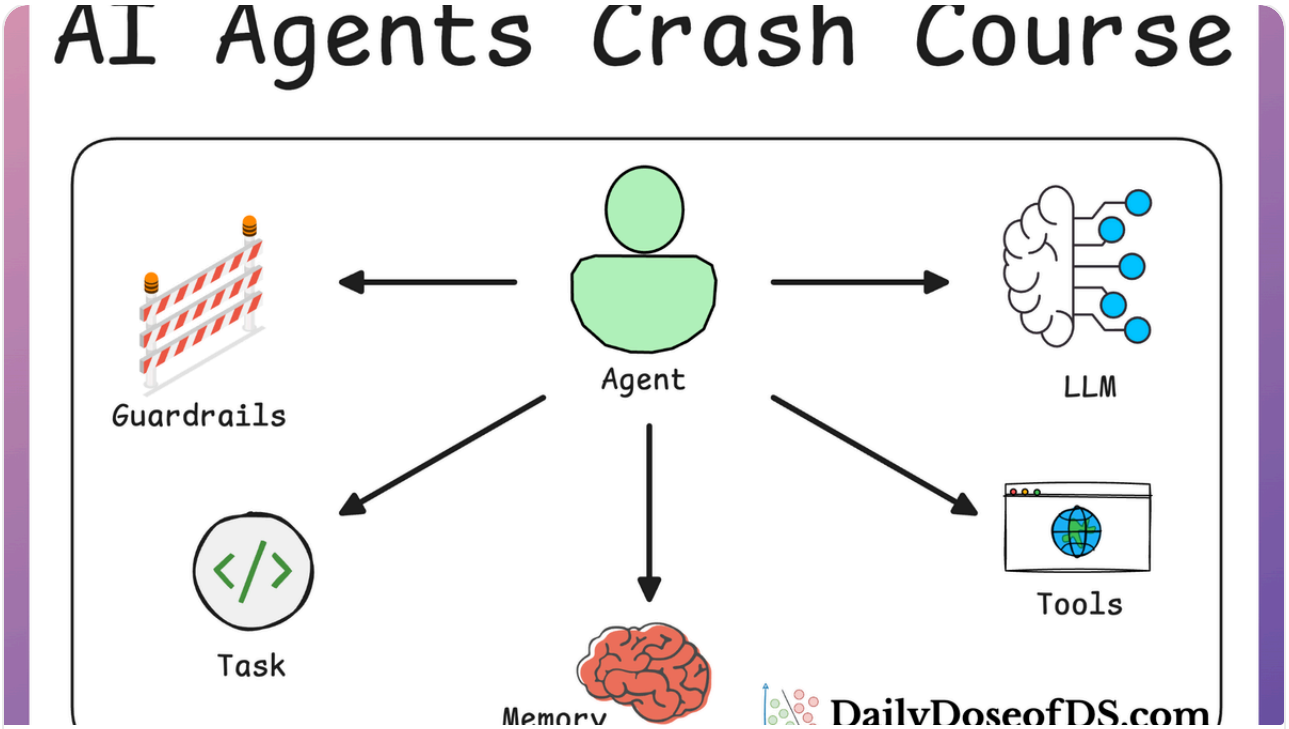
But what we've covered is still just the surface.

In Part 9, we'll go one level deeper. We shall:

- Formalize the 5 types of memories.
- Learn how to customize memory settings.
- Learn how memory works internally—including the vector storage and similarity matching logic behind the scenes.
- Learn how to reset or manage memory, at runtime or across sessions.
- and much more.

By the end of Part 9, you'll have a deep understanding of CrewAI's memory system—not just how to use it, but how to design better workflows because of it.

Read it here:



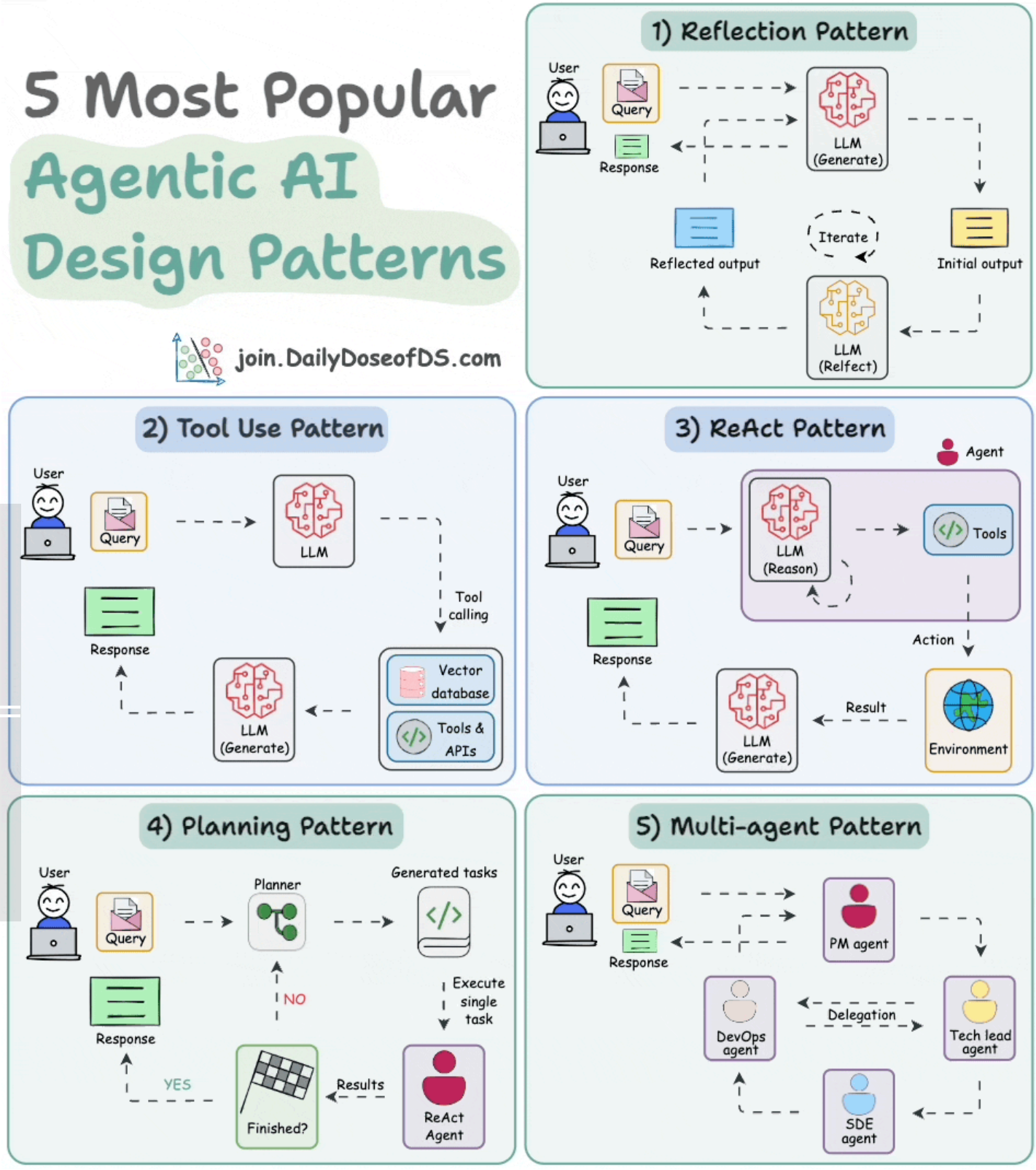
AI Agents Crash Course—Part 9 (With Implementation)

A deep dive into memory for Agentic systems.

Daily Dose of Data Science • Avi Chawla

Moreover, in the upcoming parts, we have several other advanced agentic things planned for you:

- Building production-ready agentic pipelines that scale.
- Agentic RAG (Retrieval-Augmented Generation) – combining RAG with AI agents [we understood this a bit in this part too].
- Optimizing agents for real-world applications in business and automation.
- Building Agents around the Agentic patterns depicted below:



- and many many more.

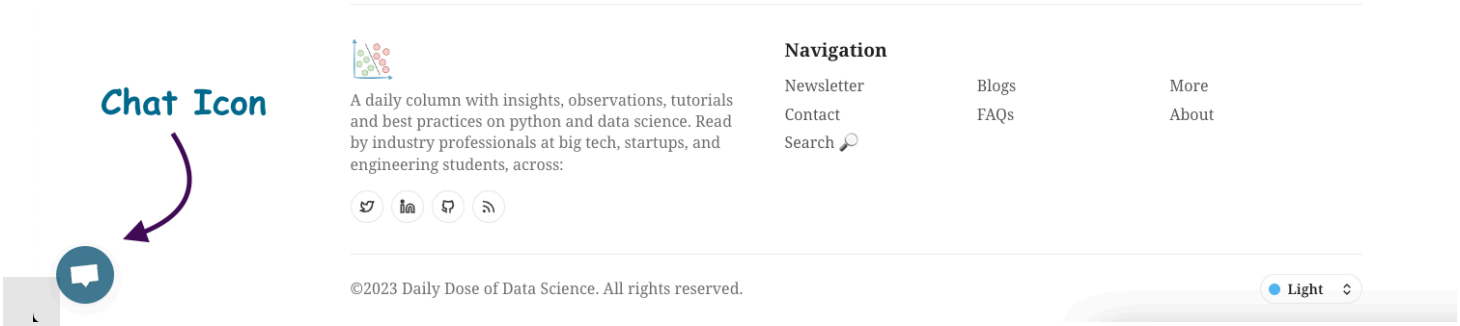
As always, thanks for reading!

Any questions?

Feel free to post them in the comments.

Or

If you wish to connect privately, feel free to initiate a chat here:



Connect via chat

Agents LLMs AI Agent Crash Course

Share this article

Read next

MCP Jun 15, 2025 • 22 min read

The Full MCP Blueprint: Building a Full-Fledged MCP Workflow using Tools, Resources, and Prompts

Model context protocol crash course—Part 4.

Avi Chawla, Akshay Pachaar

Agents Jun 8, 2025 • 20 min read

The Full MCP Blueprint: Building a Custom MCP Client from Scratch

Model context protocol crash course—Part 3.

Avi Chawla, Akshay Pachaar

Agents Jun 1, 2025 • 22 min read

The Full MCP Blueprint: Background, Foundations, Architecture, and Practical Usage (Part B)

Model context protocol crash course—Part 2.

Avi Chawla, Akshay Pachaar