Apr 6, 2025

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

AI Agents Crash Course—Part 9 (with implementation).

Avi Chawla, Akshay Pachaar

Introduction

In Part 8 of this AI Agents crash course, we took a practical look at how memory works in CrewAI-powered systems.

We covered how agents can:

- Track recent interactions using short-term memory.
- Learn from past performance with long-term memory.
- Retain structured facts about specific people or entities with entity memory.

We explored how to query memory with `.search()`, what kind of metadata is returned, and how agents use those memory chunks to answer questions, refine responses, or personalize behavior across tasks and sessions.

But that was just the surface.

While Part 8 gave us a high-level, hands-on understanding of how memory behaves in CrewAI, this part (Part 9) is where we get formal and technical.

In this part, we'll:

- 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.

Let's dive in!

Why memory? A quick recap

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.

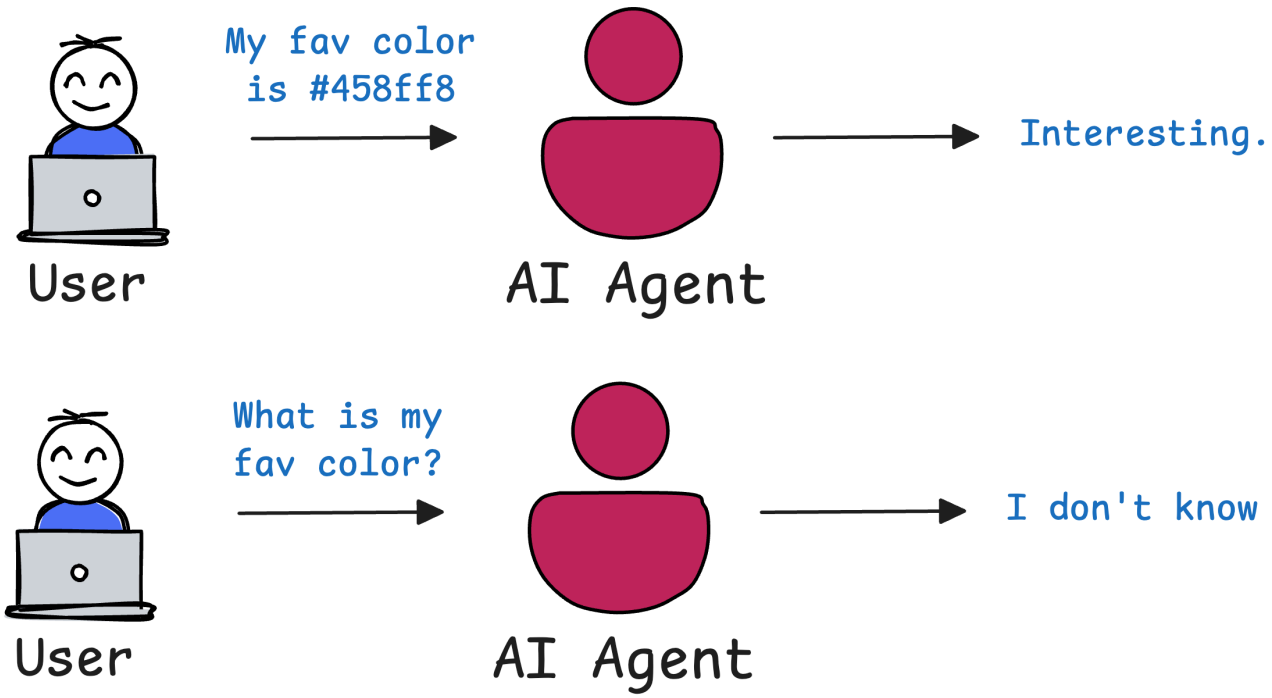
We've also explored how to equip agents with Knowledge, allowing them to access internal documentation, structured datasets, or any domain-specific reference material needed to complete their tasks.

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:

Interaction without memory

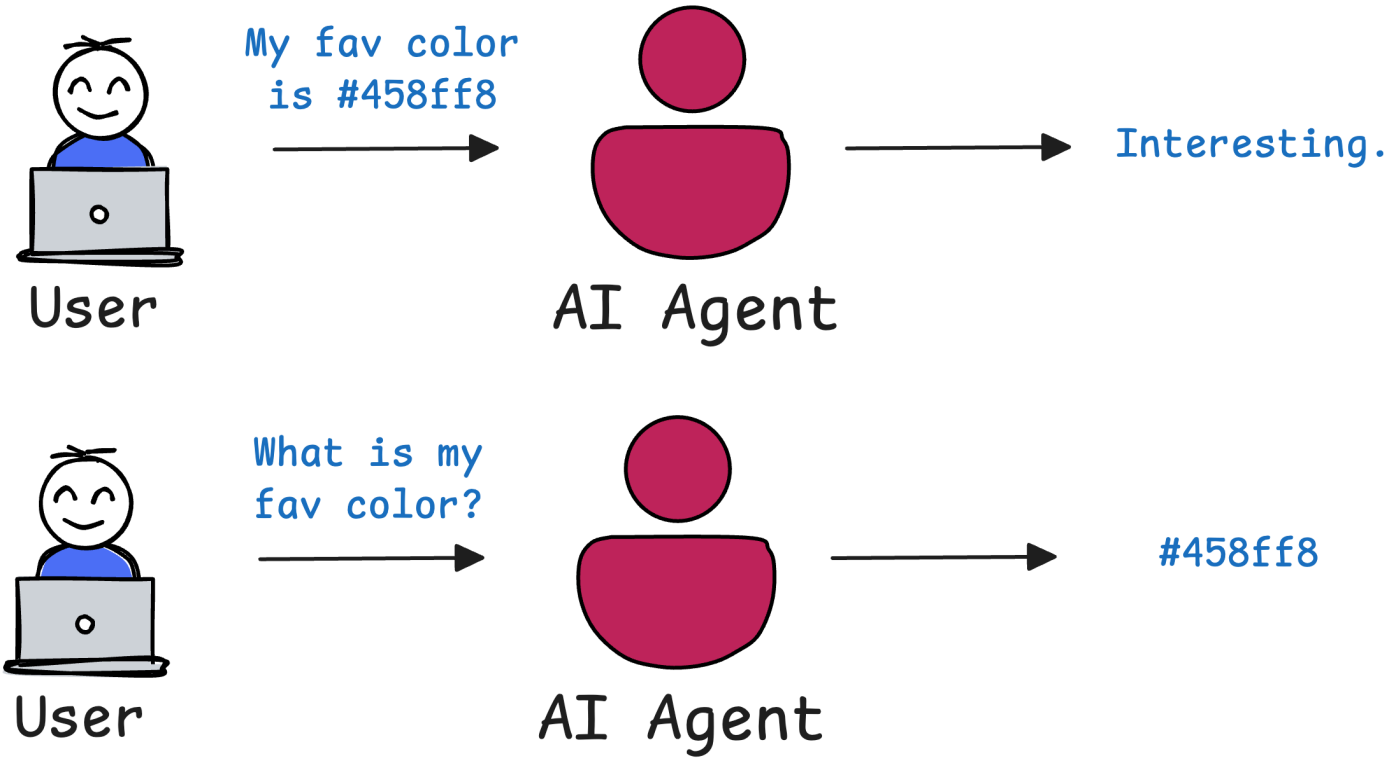


- Recall something a user told them earlier in the conversation.
- Accumulate learnings across different sessions?
- Personalize answers for individual users based on their past behavior.
- Maintain facts about entities (like customers, projects, or teams) across workflows.

Memory solves this.

In 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.

Interaction with memory



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.

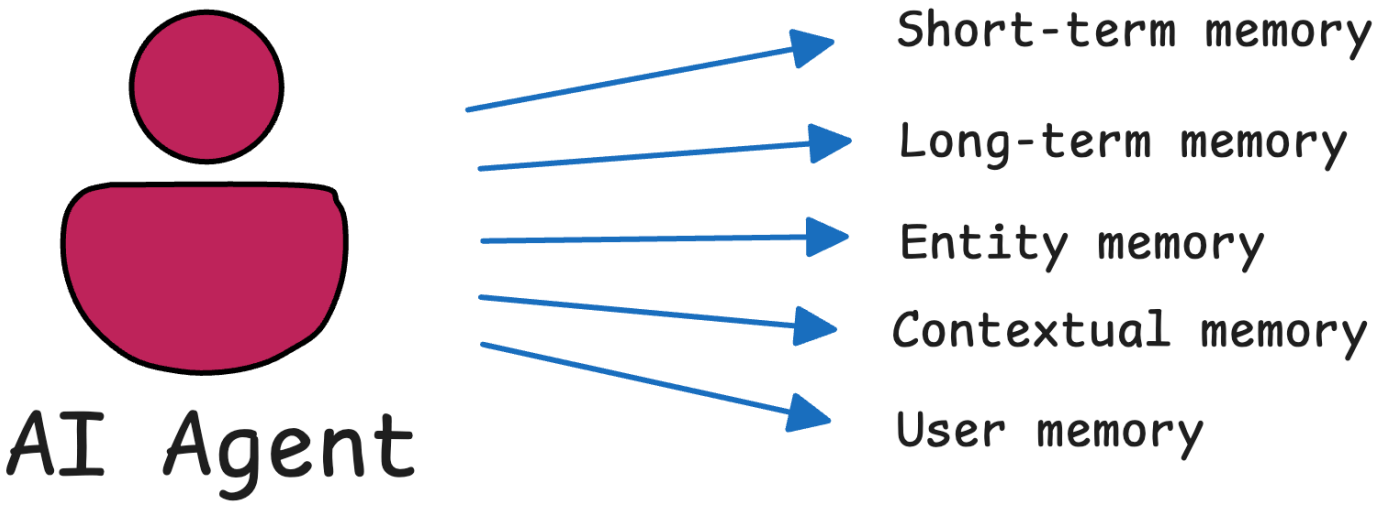
Memory matters because if you have an Agentic system deployed in production and it is running without memory, every interaction is a blank slate.

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.

With memory, your agent becomes context-aware.

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):



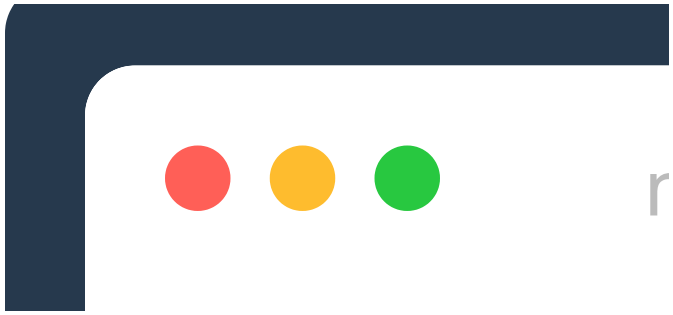
- Short-Term Memory
- Long-Term Memory
- Entity Memory
- Contextual Memory, and
- User Memory

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

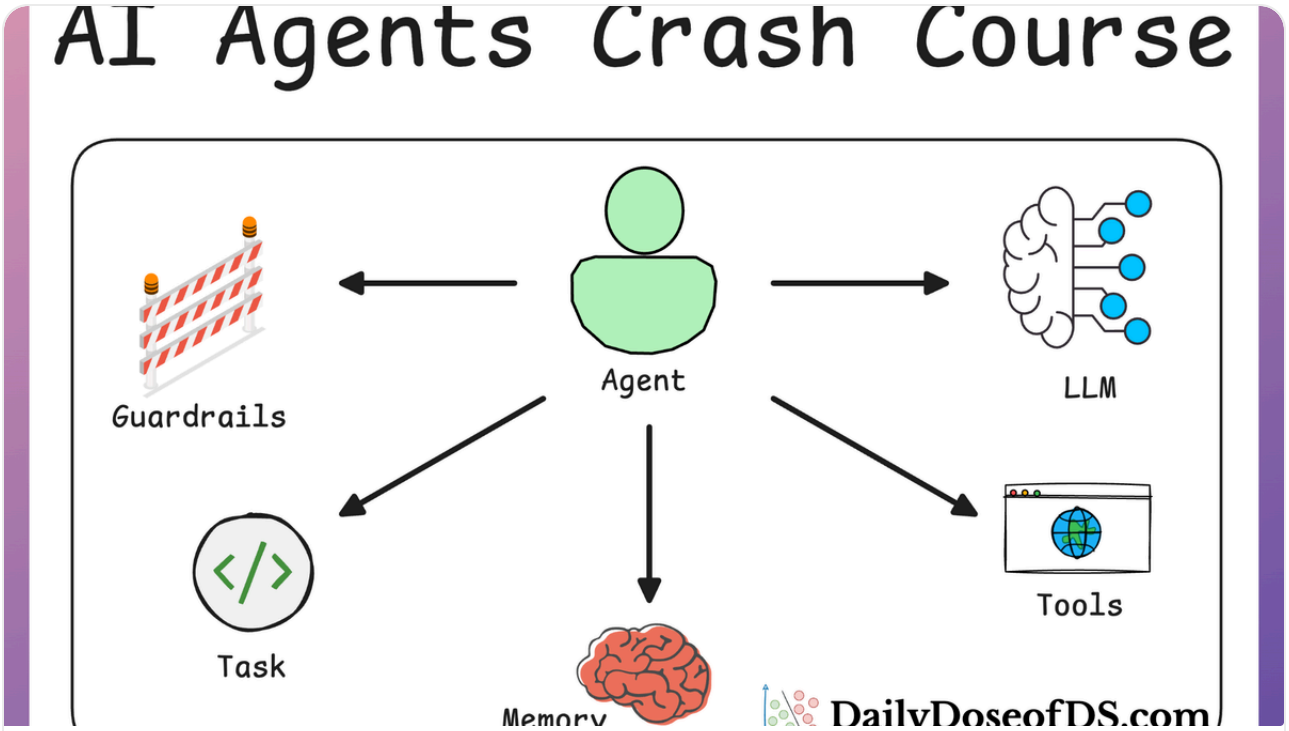
Let's understand them below conceptually and get into the technical details as well.

> 💡 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.



If you haven't read Part 8 yet on Memory, we highly recommend doing so:



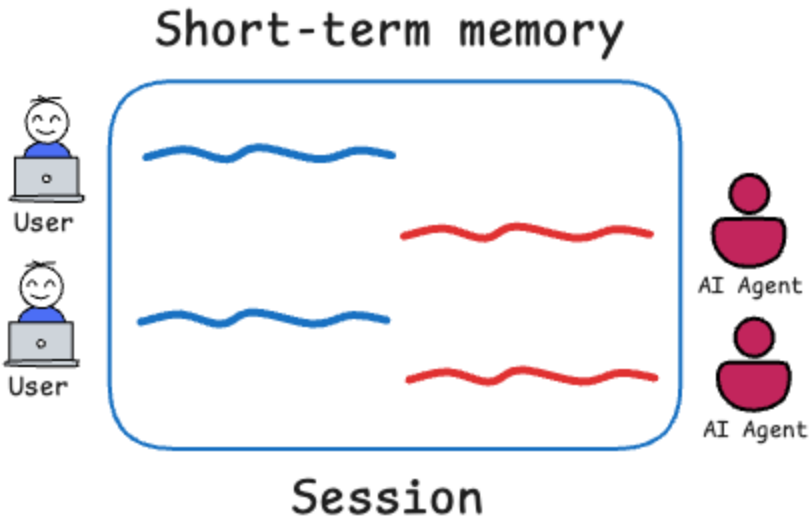
AI Agents Crash Course—Part 8 (With Implementation)

A deep dive into memory for Agentic systems.

Daily Dose of Data Science • Avi Chawla

#1) Short-Term Memory

Short-term memory in CrewAI is the agent's "working memory" for the current session or task sequence. It stores recent interactions and outcomes so the agent can recall information relevant to the ongoing context.



As we also saw in Part 8, under the hood, CrewAI implements short-term memory using a Retrieval-Augmented Generation (RAG) approach:

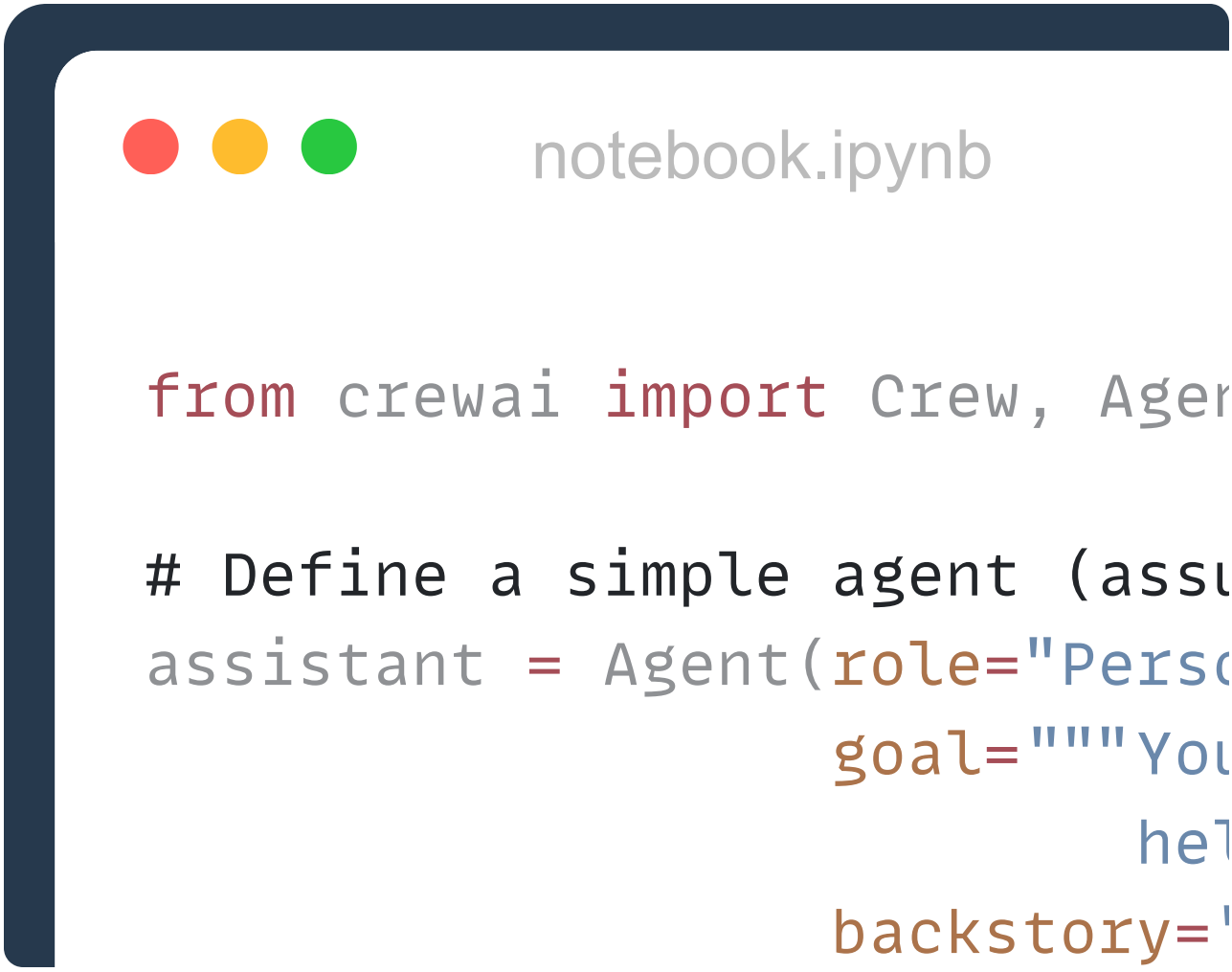
- It embeds the text of recent prompts and results into a vector store (by default using OpenAI embeddings and a local Chroma vector database)
- Retrieves relevant pieces as context for new queries.

This means even if a conversation or task involves many steps, the agent can fetch the most relevant bits from earlier in the session without exceeding the model's immediate context window.

By default, CrewAI's memory system (including short-term memory) is disabled.

To use short-term memory, you need to enable the memory system when configuring your Crew.

This is as simple as setting `memory=True` in the `Crew` configuration.



With `memory=True`, CrewAI will automatically maintain short-term memory during the crew's execution.

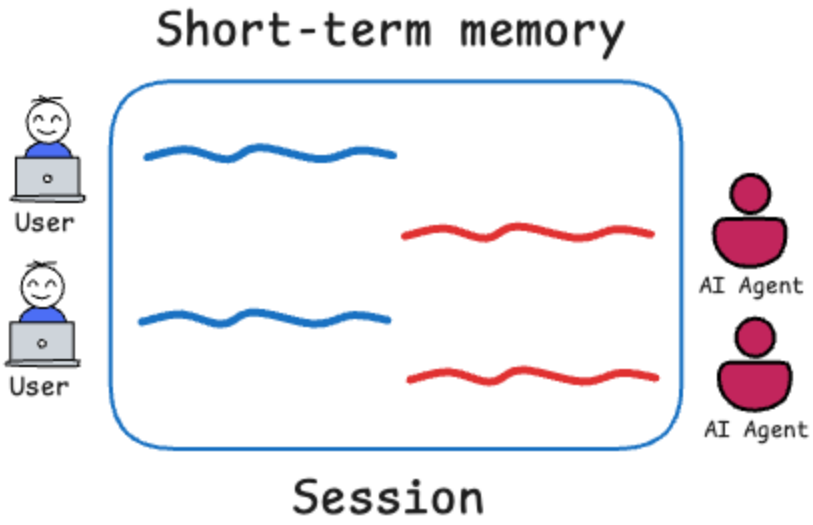
Also, you typically don't need to manage short-term memory manually—it works behind the scenes to store and retrieve recent context.

When is short-term memory active?

Whenever `memory=True` for a crew, short-term memory is active for that crew's agents.

It comes into play especially in multi-turn interactions or multi-step task sequences within that single run (or "session") of the crew.

You can relate this to a single chat session on ChatGPT.



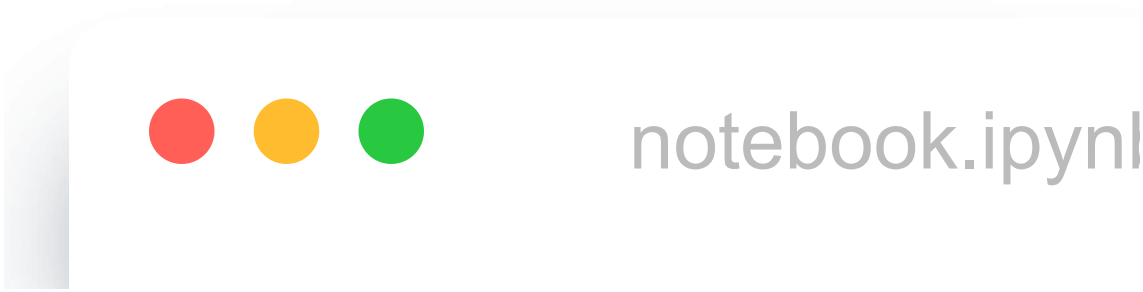
Usually, in ChatGPT, you can recall anything that happened within that particular chat session. However, ChatGPT may struggle to fetch specific stuff that was discussed in some other chat session—unless it is a high-level detail or an entity related information which we shall see shortly.

How to use it within a Crew?

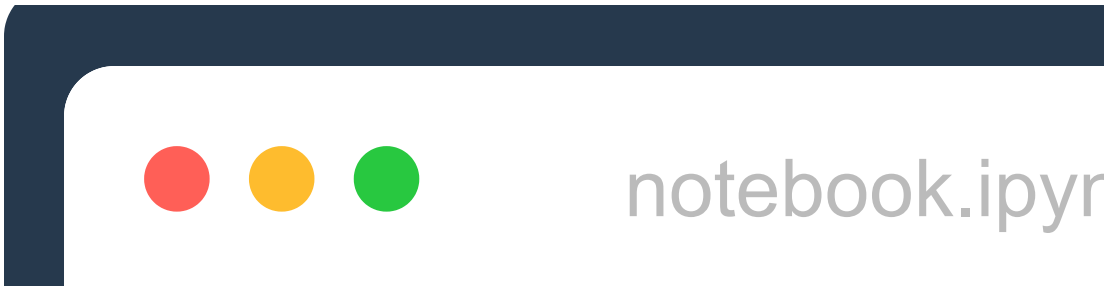
There's no separate API call needed to "add" things to short-term memory—the CrewAI framework will automatically capture each agent's input/output and significant intermediate results into the short-term memory store.

The effect is that an agent can reference earlier parts of the conversation or task.

For example, consider a simple two-step interaction: first the user tells the agent a fact:



...and next the user asks a question that relies on that fact:



With memory enabled, the agent can answer correctly by retrieving the facts from short-term memory.

```
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.'
```

Without memory, the agent might forget the fact if it wasn't explicitly carried in the prompt.

```
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 is critical for continuity in conversations since the agent doesn't ask the user to repeat information and can stay consistent within a single session.

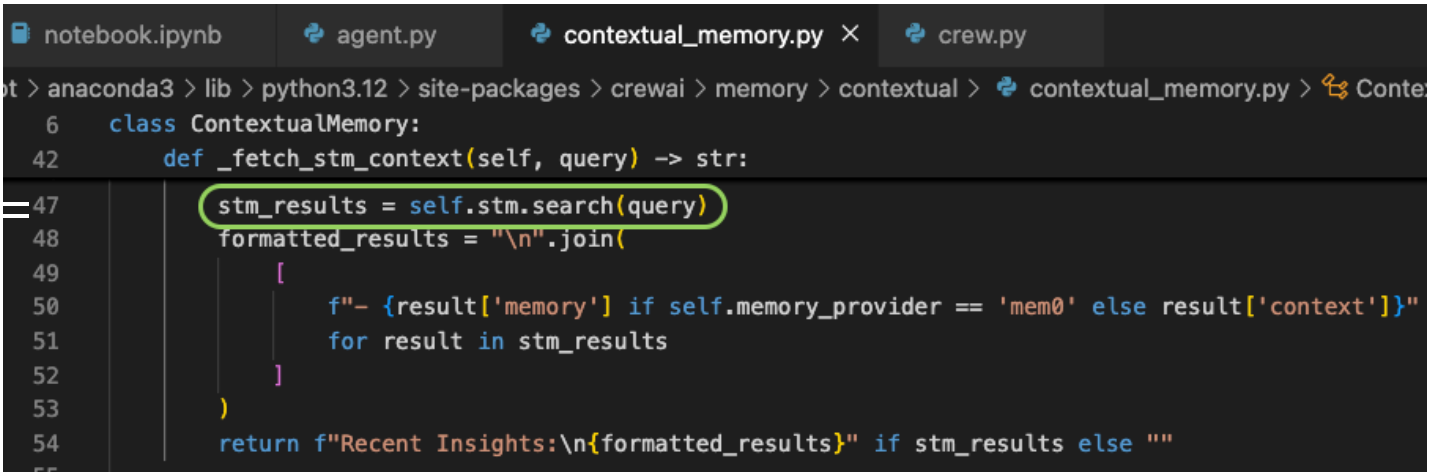
How it works?

Since you might already know RAG, it is easier to understand how this works.

After each task (or each significant interaction), CrewAI's short-term memory module stores a vector representation (using an embedding model) of the content.

> 💡 In our demo, the usage of an embedding model is not that evident since the Crew is using OpenAI internally.

When the Agent gets a new prompt (like the second task above), it uses a similarity search to find relevant pieces of prior context and feeds those into the prompt for the language model. Recall Part 8, where we also noticed how it's done in practice:



All of this happens internally; as a developer using CrewAI, you mostly just enable memory and benefit from more coherent agent behavior.

Sample demo

Let's walk through a complete working example that showcases short-term memory in action.

The setup consists of a personal assistant agent that learns something in one task (e.g., "my favorite color is #46778F") and recalls it in a later task ("what is my favorite color?").

In the code below, we set up a local SQLite-based vector database for storing memories.

Internally:

- CrewAI uses `RAGStorage` as the underlying mechanism.
- Text (prompts, answers) is embedded using a model (by default, OpenAI's embedding model).
- These embeddings are stored in `short_term_memory.db` and queried using similarity search.

Moving on, we define the Agent, its Task and a Crew:

notebook.ipynb

```
agent = Agent(role="A 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=agent)


crew = Crew(
    agents=[agent],
    tasks=[task],
    process=Process.sequential,
    memory=True,
    short_term_memory=short_term_memory
)
crew.kickoff(inputs={"user_task": "My favorite color is #46778F."})
```

use the short-term memory storage

Here's the key:

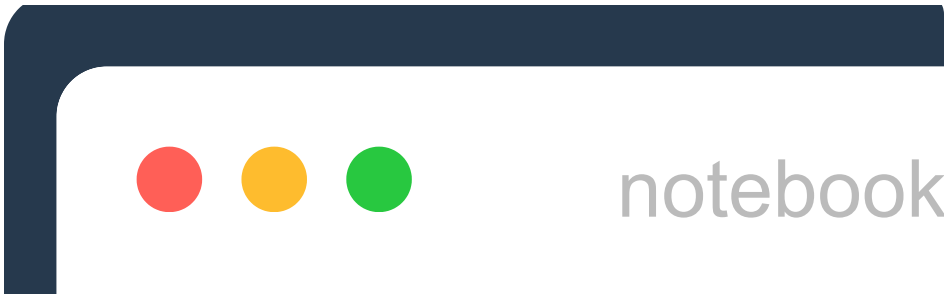
- `memory=True`: Enables memory for the crew.
- `short_term_memory=...`: We pass the specific memory object we created earlier.

As soon as this crew starts running tasks, CrewAI will start storing interaction data in the vector store.

The above Crew execution produces the following output:

```
Overriding of current TracerProvider is not allowed
# Agent: A Personal Assistant
## Task: Handle this task: My favorite color is #46778F.


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

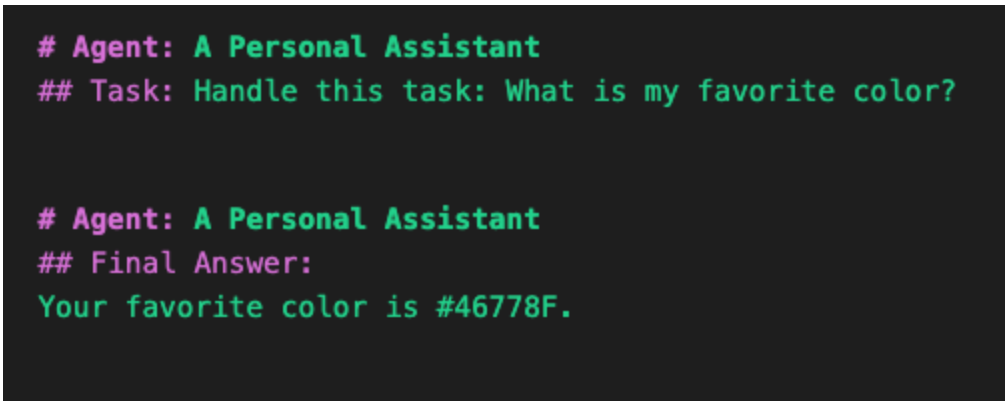
Now the agent is given a new input—but it needs information from the previous interaction:



Because short-term memory is enabled:

- CrewAI performs a similarity search in the vector store.
- The most relevant memory (“My favorite color is #46778F”) is retrieved.
- This is prepended as additional context into the prompt.

This produces the following output:



The model is now able to answer correctly, even though the current prompt doesn't explicitly include the color.

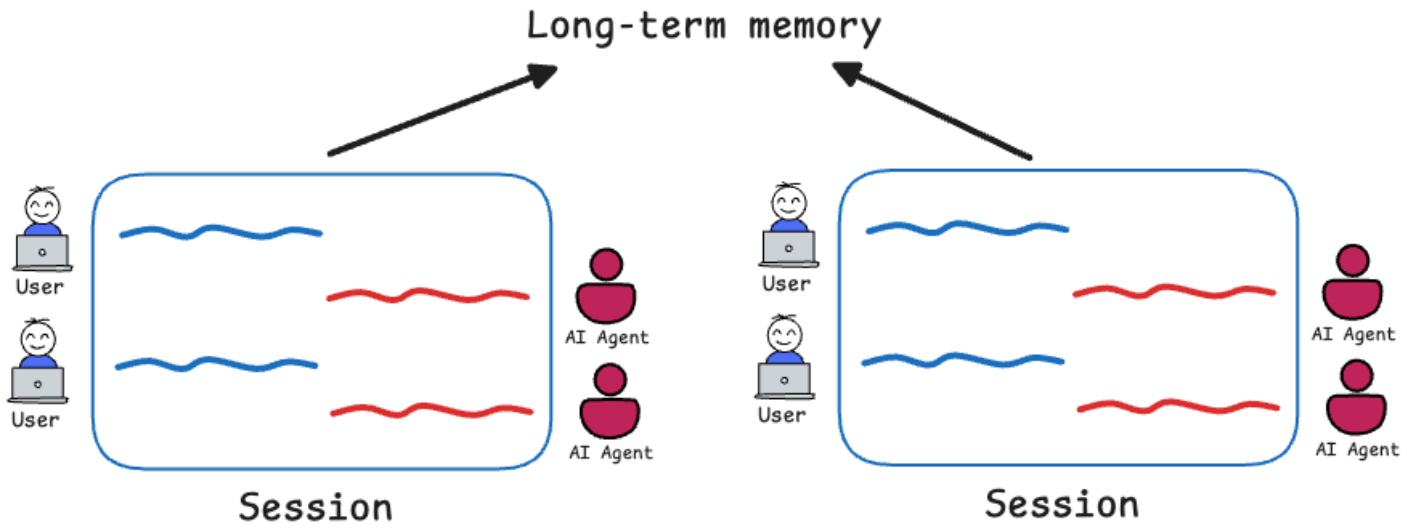
Let’s briefly understand about what happens under the hood:

1. Embedding: After each task, the prompt/response text is embedded using OpenAI’s embedding model (or any configured one).
2. Storage: Those vectors are stored in ChromaDB via the RAGStorage interface.
3. Similarity Search: On a new task, CrewAI retrieves top-K relevant memory chunks using cosine similarity.
4. Context Injection: Retrieved chunks are passed as part of the system prompt to the agent’s LLM.

So essentially, by just enabling memory and configuring a storage backend, you gain powerful RAG-based session memory—critical for agents in multi-turn conversations or complex workflows.

#2) Long-Term Memory

While short-term memory handles the immediate context, long-term memory is about persistence across sessions, and it is used to retrieve high-level information that's related to the task.



You might recall this from the previous part where we noticed that while querying the long-term memory, the implementation used the current task's description to fetch the relevant context (line 35 in the screenshot 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))
```

Being driven by broader information rather than specific information (like with short-term knowledge), the Agent can recall and retrieve high-level insights and learnings from past executions.

To put it another way, while short-term memory is transient (per session), long-term memory is cumulative (across sessions).

Under the hood, CrewAI's default long-term memory uses a lightweight database (SQLite3) to store task results persistently on disk.

When memory is enabled, this happens automatically—the agent will write certain outputs or summaries to the long-term store. We noticed this when we discussed long-term memory in Part 8:

```
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 long-term 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.

- `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.

By reviewing these stored memories, the agent can improve future decisions and avoid repeating past mistakes.

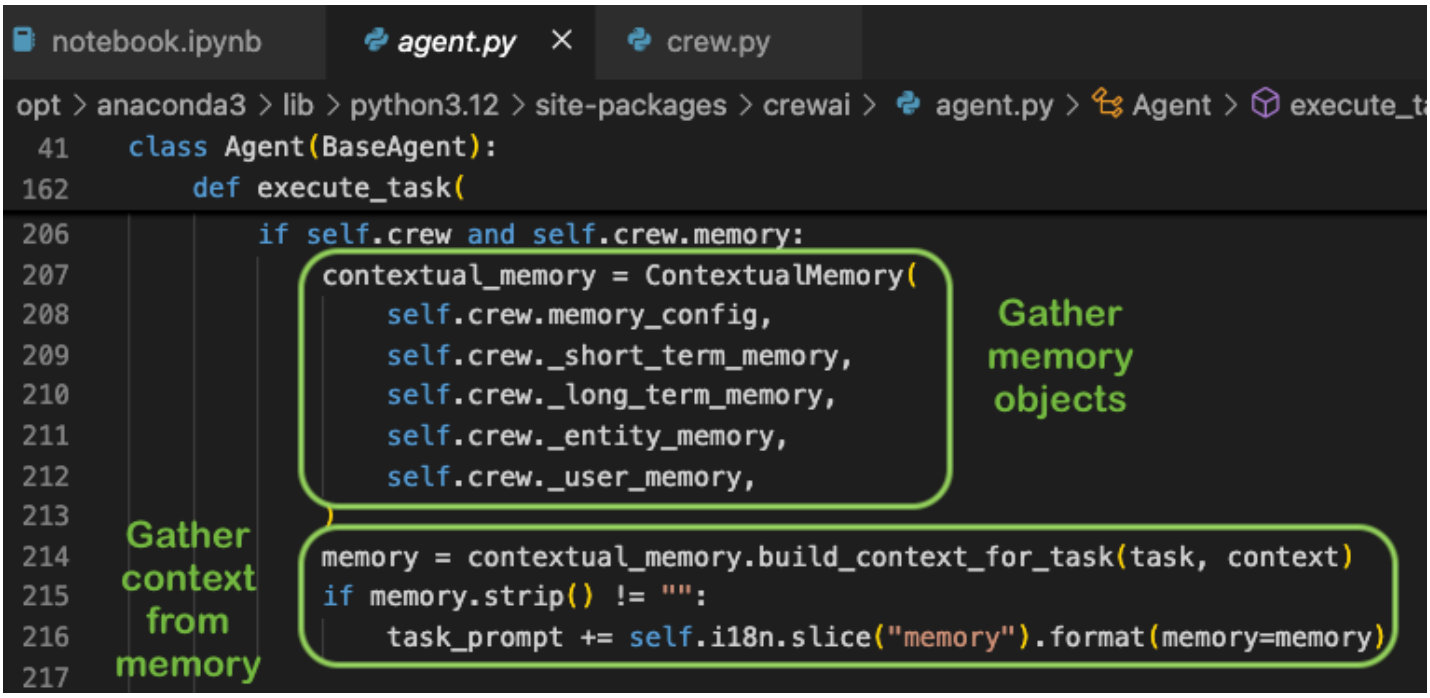
Activating long-term memory

The good news is that when you turn on `memory=True` for a crew, you're enabling both short-term and long-term memory (as well as entity memory) by default.

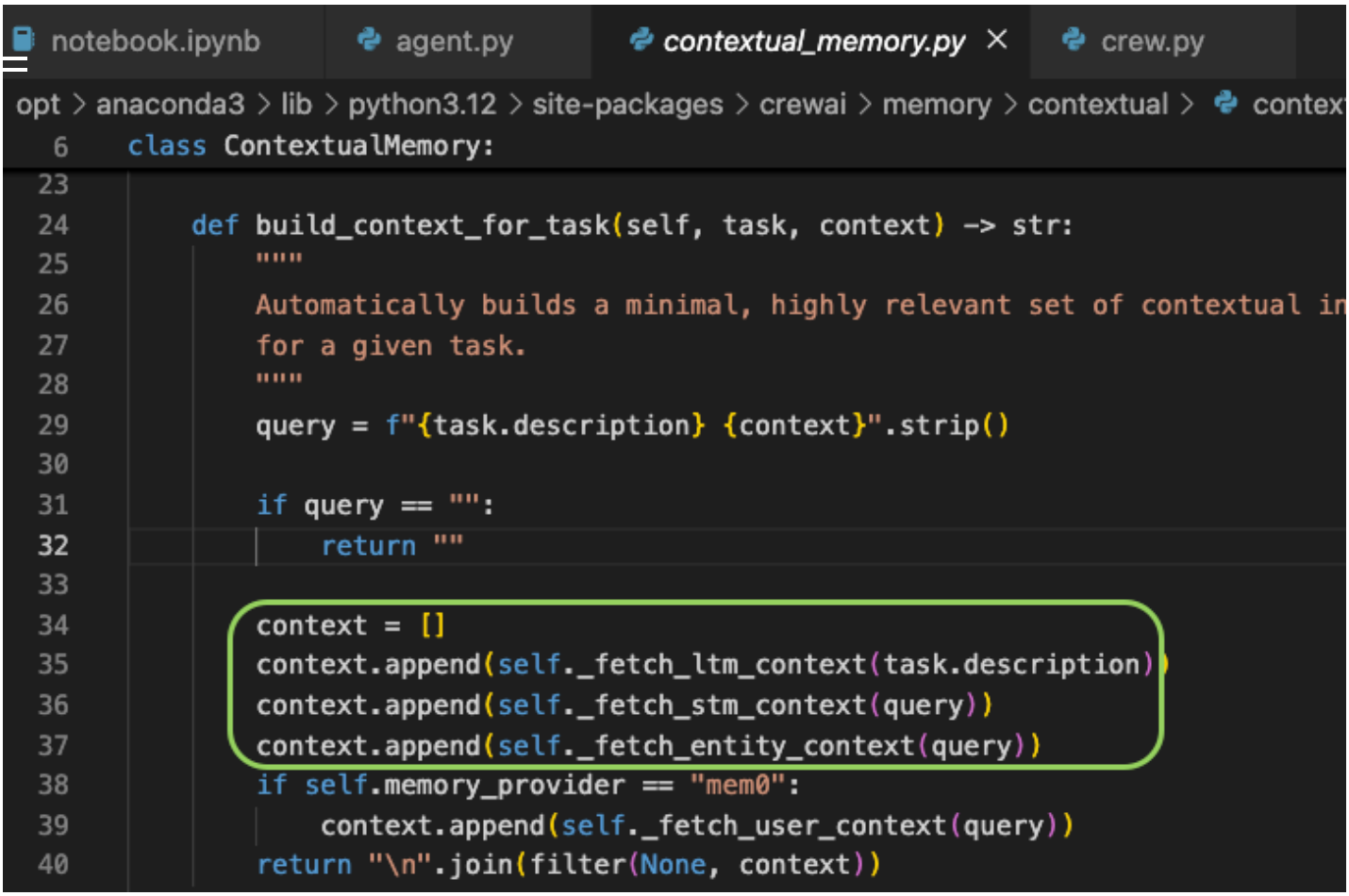
You don't have to explicitly choose one or the other—CrewAI's memory system includes all types working in tandem.

This must have already become clear when we looked at the actual implementation in Part 8, but to recall:

- The `the` method invokes the `build_context_for_task` method, which, as the name suggests, finds relevant context for the task using the information in the memory. This is shown in line 214 below:



- Within the `build_context_for_task` method, you can notice in Lines 35-37 that all short-term, long-term memory and entity method are being queried:



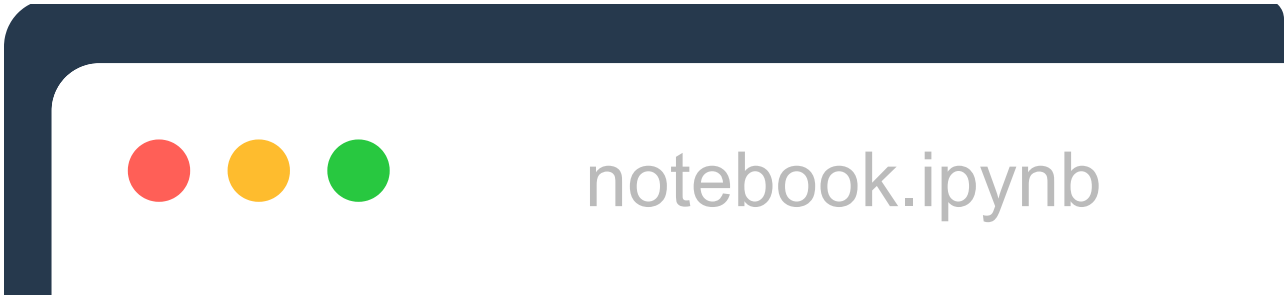
If you want to customize how long-term memory works (for example, specify a custom storage location or database name), you can do so via the `long_term_memory` parameter when creating the Crew.

Sample demo

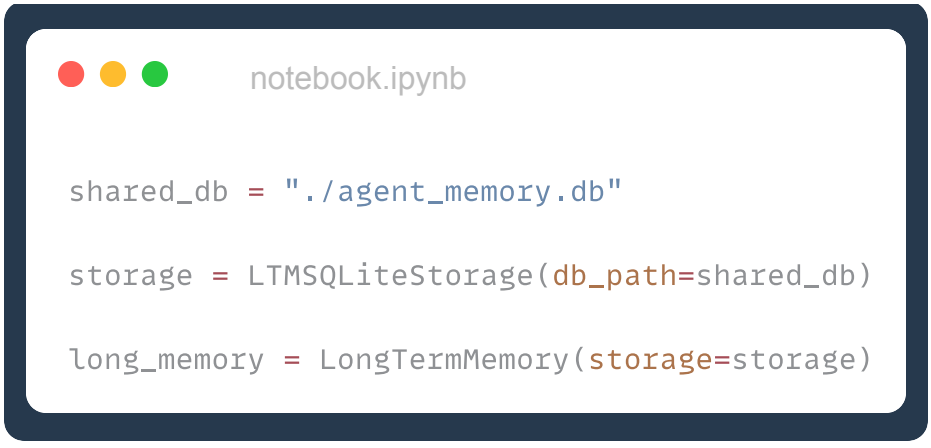
To illustrate long-term memory, let's simulate an agent that learns a fact in one session (in the `its` first Crew) and is asked about it in a new session in another Crew (after the short-term memory would normally be gone).

We will use a custom database path to make the persistence explicit and simulate two separate runs.

To do this, we first start with some import statements:



Next, we create a long-term memory storage that will be shared between sessions:



Moving on, we define the Agent, its Task and the Crew:

notebook.ipynb

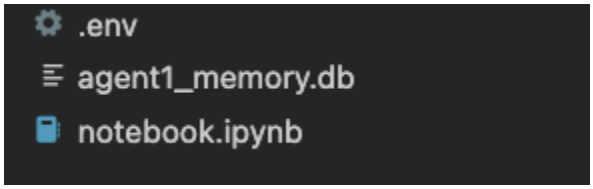
```
agent1 = Agent(role="First Personal Assistant",
               goal="""You are a personal assistant that handles
                       user's tasks.""",
               backstory="""You are a personal assistant that
                            can help the user with their tasks.""",
               verbose=True)

task1 = Task(description="Handle this task: {user_task}",
             expected_output="A clear and concise answer to the question.",
             agent=agent1)

crew1 = Crew(
    agents=[agent1],
    tasks=[task1],
    process=Process.sequential,
    memory=True,
    long_term_memory=long_memory
)
crew1.kickoff(inputs={"user_task": "My favorite color is #46778F."})
```

Enable memory and use the long-term memory storage

This produces the following output:

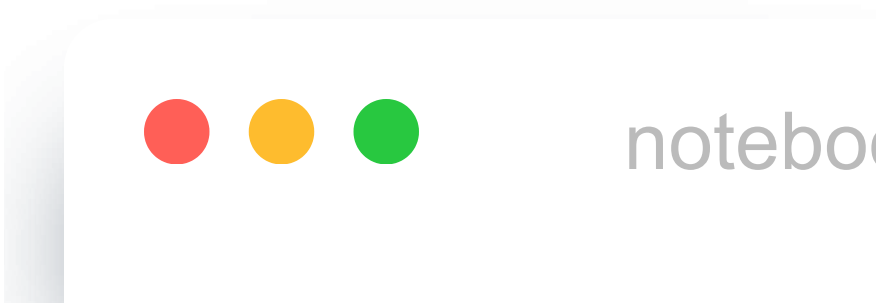
```
Overriding of current TracerProvider is not allowed
# Agent: First Personal Assistant
## Task: Handle this task: What is my favorite color?


# Agent: First Personal Assistant
## Final Answer:
Your favorite color is #46778F, a calming blue-green hue that evokes feelings of tranquility and nature. This color ofter

CrewOutput(raw="Your favorite color is #46778F, a calming blue-green hue that evokes feelings of tranquility and nature.
```

Also, when you will run the above code, you will notice a local SQLite database file created in your current directory—`agent1_memory.db`.



We can see the contents as follows:

- First, connect to the SQLite database file:



- Next, list all the tables inside this database:

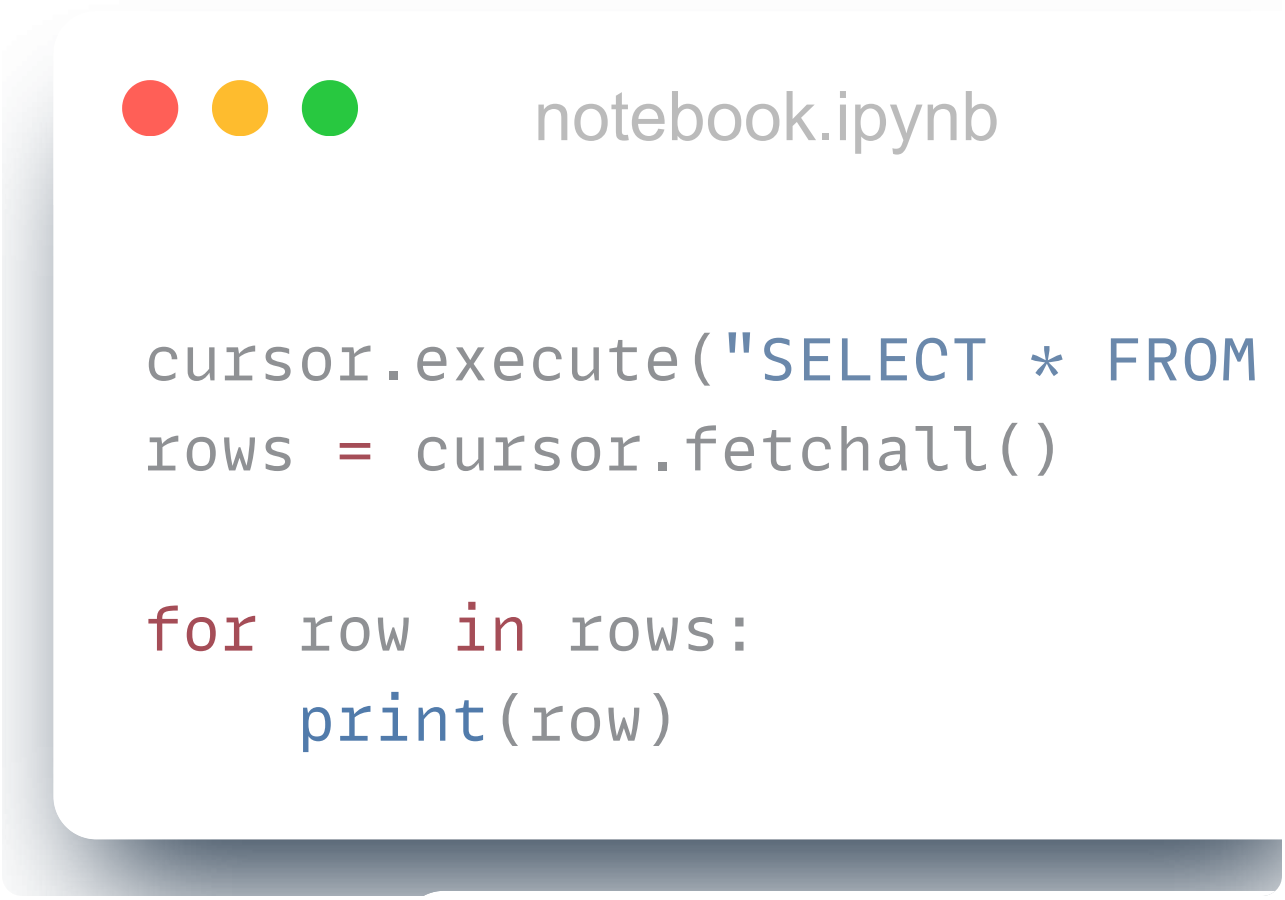
notebook.ipynb

```
cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")

tables = cursor.fetchall()

print("Tables in the database:")
for table in tables:
    print(table[0])
```

Output

```
Tables in the database:
long_term_memories
sqlite_sequence
```

The table name `long_term_memories` appears to be the most relevant, so querying, we get the following output:



The result we get is a list of all long-term memory entries that have been stored so far in this database. Each item in the list contains reflections, feedback, or insights learned from that task and more.

Now to simulate another session, let's define another Crew:

notebook.ipynb

```
task2 = Task(description="Handle this task: {user_task}",
             expected_output="A clear and concise answer to the question.",
             agent=agent1)

crew2 = Crew(
    agents=[agent1],
    tasks=[task2],
    process=Process.sequential,
    memory=True,
    long_term_memory=long_memory
)

crew2.kickoff(inputs={"user_task": "What is my favorite color?"})
```

use the same long-term memory storage

Run the second session

This produces the following output, which shows that we were able to retrieve info provided to one Crew (a session) into a different Crew (another session):

```
Overriding of current TracerProvider is not allowed
# Agent: First Personal Assistant
## Task: Handle this task: What is my favorite color?


# Agent: First Personal Assistant
## Final Answer:
Your favorite color is #46778F, a calming blue-green hue that evokes feelings of
```

Here's what's happening in the above example:

1. We configure a `LongTermMemory` instance backed by an SQLite database file `agent_memory.db` on disk. By passing the same `long_memory` object to both `crew1` and `crew2`, we ensure they share the same long-term memory store. In practice, you could also let CrewAI use the default storage; we explicitly specify one here for clarity.

2. Session 1: We create a crew with a single agent and one task where the user provides a fact about the favorite color. With memory enabled, once this task is processed, the agent's long-term memory will store this information (for

example, it might store a summary or the full statement as something the agent “learned”). The crew then ends. At this point, the short-term memory (conversation context) won’t persist beyond the session, but the long-term memory in `agent_memory.db` now contains the fact about the user's favourite color.

3. Session 2: We create a new Crew (simulating, say, a new conversation or a later time) and ask the agent about user's favorite color. By using the same `long_memory` (pointing to the same database file), this new crew has access to whatever was stored previously. When the agent tries to answer this question, CrewAI will search the long-term memory and find the fact from Session 1. The agent can then retrieve that info and answer this question even though this was a brand new session with no immediate context.

This demonstrates how long-term memory provides continuity across separate runs.

In essence, the agent has remembered a piece of information beyond the lifespan of a single conversation.

Long-term memory is useful for scenarios where you want your agent to gradually build up knowledge.

For instance, if you’re using agents to research and store findings, you can accumulate those findings in long-term memory.

Next time the agent encounters a related query, it can pull from its stored knowledge rather than starting from scratch.

> 💡 Note: In the above example, we reused the same `Agent` instance for simplicity, but that isn’t strictly necessary—the memory is tied to the Crew (and the storage behind it), not the Python object. We could have instantiated a new Agent in Session 2 and as long as we pointed to the same `agent_memory.db`, the new agent would have access to the stored memories. This also underscores how memory can be shared across different crews when explicitly configured (we’ll discuss memory sharing

more soon). If we hadn't provided the `long_term_memory=long_memory` to `crew2`, CrewAI would have created a fresh long-term store (likely a different default file), and the second session would not recall the fact from the first.

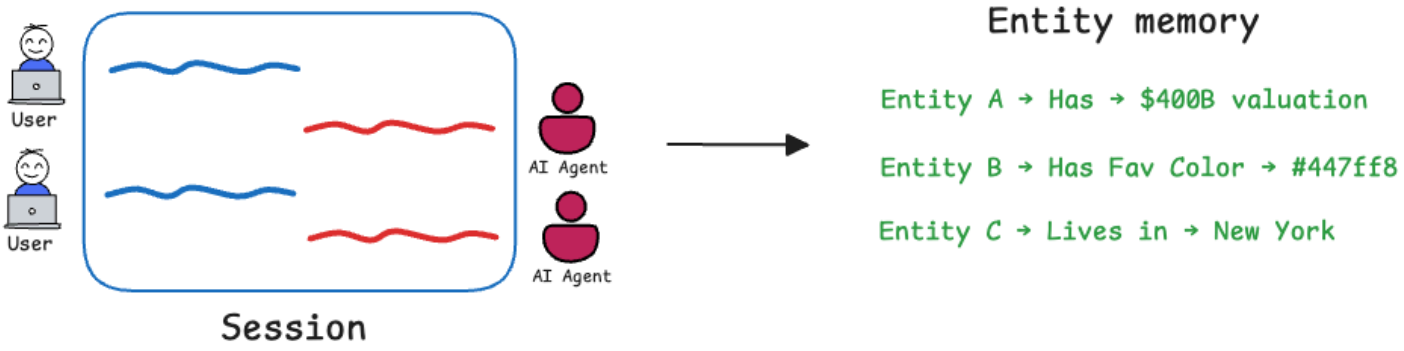
#3) Entity Memory

In complex workflows, agents often encounter various entities.

For instance—people, organizations, products, or other key concepts.

Entity memory in CrewAI is designed to capture and organize information about such entities as they come up.

Rather than storing bits of information in one undifferentiated pool, entity memory creates a sort of profile or knowledge card for each entity.



This way, if an agent deals with multiple entities over time, it can keep details about each entity separate and retrieve them when that entity is relevant.

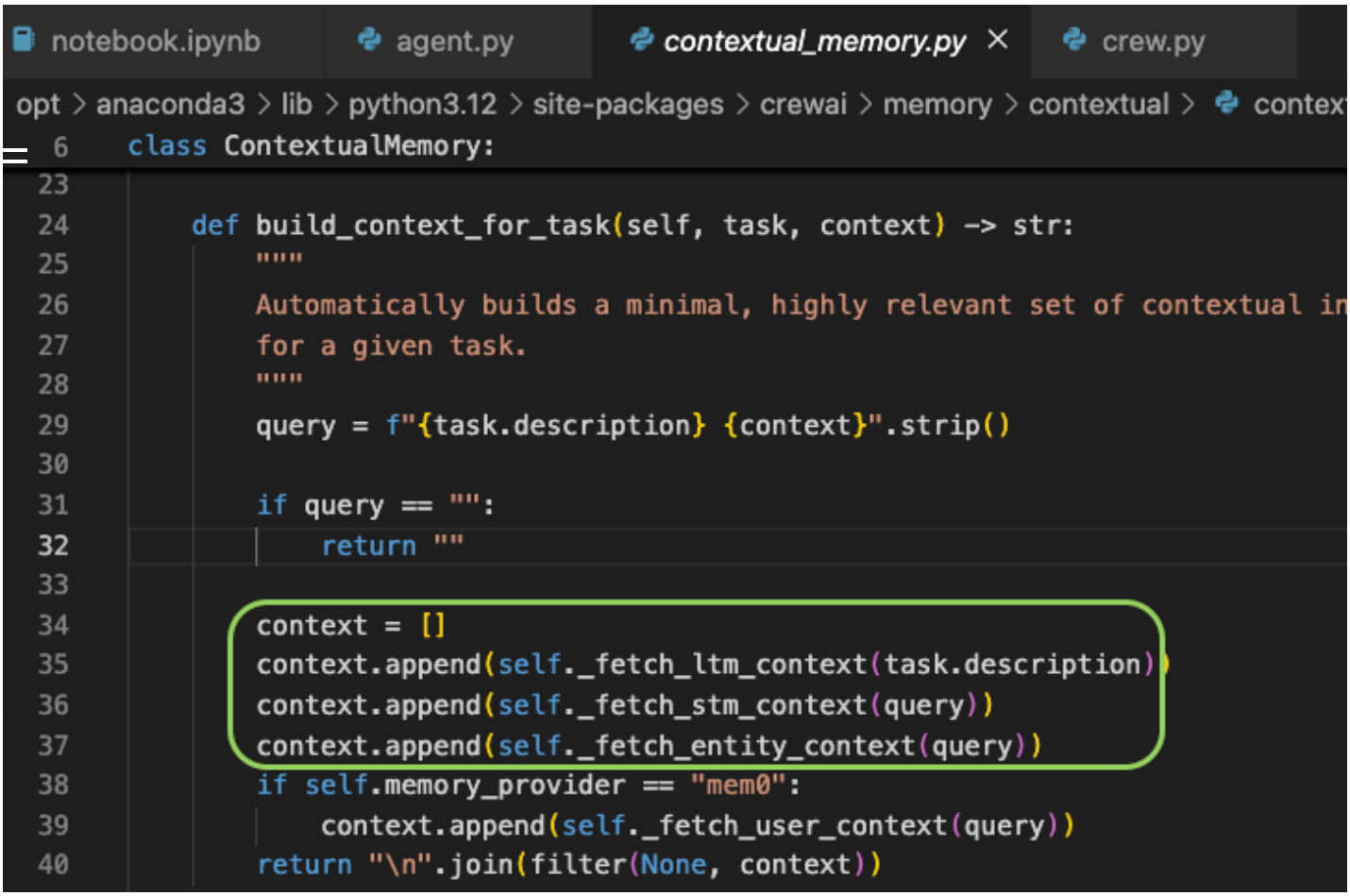
Entity memory is especially useful in scenarios like: a research assistant agent gathering information on multiple companies, a customer support agent handling multiple customers in one session, or any conversation that involves tracking attributes of multiple subjects.

With entity memory, if the agent learned that Acme Corp had a revenue of $5M and Beta Corp had a revenue of $10M, it can store those facts under each company's entity entry.

Later, if the user or task asks "What is Beta Corp's revenue?", the agent can directly pull from Beta Corp's stored info without confusing it with Acme Corp.

CrewAI's implementation of entity memory uses a vector store (RAG) like short-term memory, but it indexes information by or alongside the entity name.

In practice, you don't usually need to manually manage entity memory; just like short-term memory, it works automatically when memory is enabled (again, recall what we saw during the `build_context_for_task` method):



The system may detect entities in the text and store relevant statements about them.

However, to effectively use entity memory, it helps to be aware of it and structure interactions such that the agent has the opportunity to “learn” about entities one at a time.

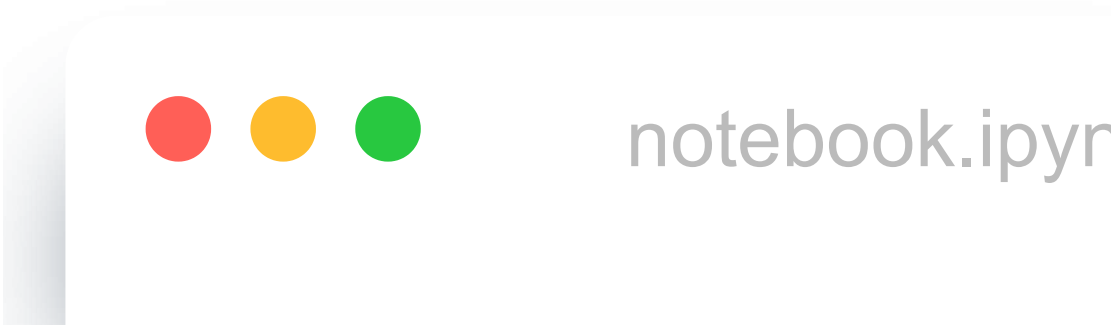
Example flow with entity memory: Suppose we have an agent that will be given information about a person and then asked questions about that person. We’ll use a single session but simulate how entity memory allows focusing on the relevant info.

Like the long-term memory example, we start with some imports:

- `EntityMemory`: This is the key class that handles entity memory in CrewAI. Internally, it wraps a vector-based storage mechanism that stores and retrieves facts or statements per entity.
- `RAGStorage`: A retrieval-augmented generation (RAG) based storage system. In our case, it acts as the backend for entity memory and supports embedding-based similarity search.

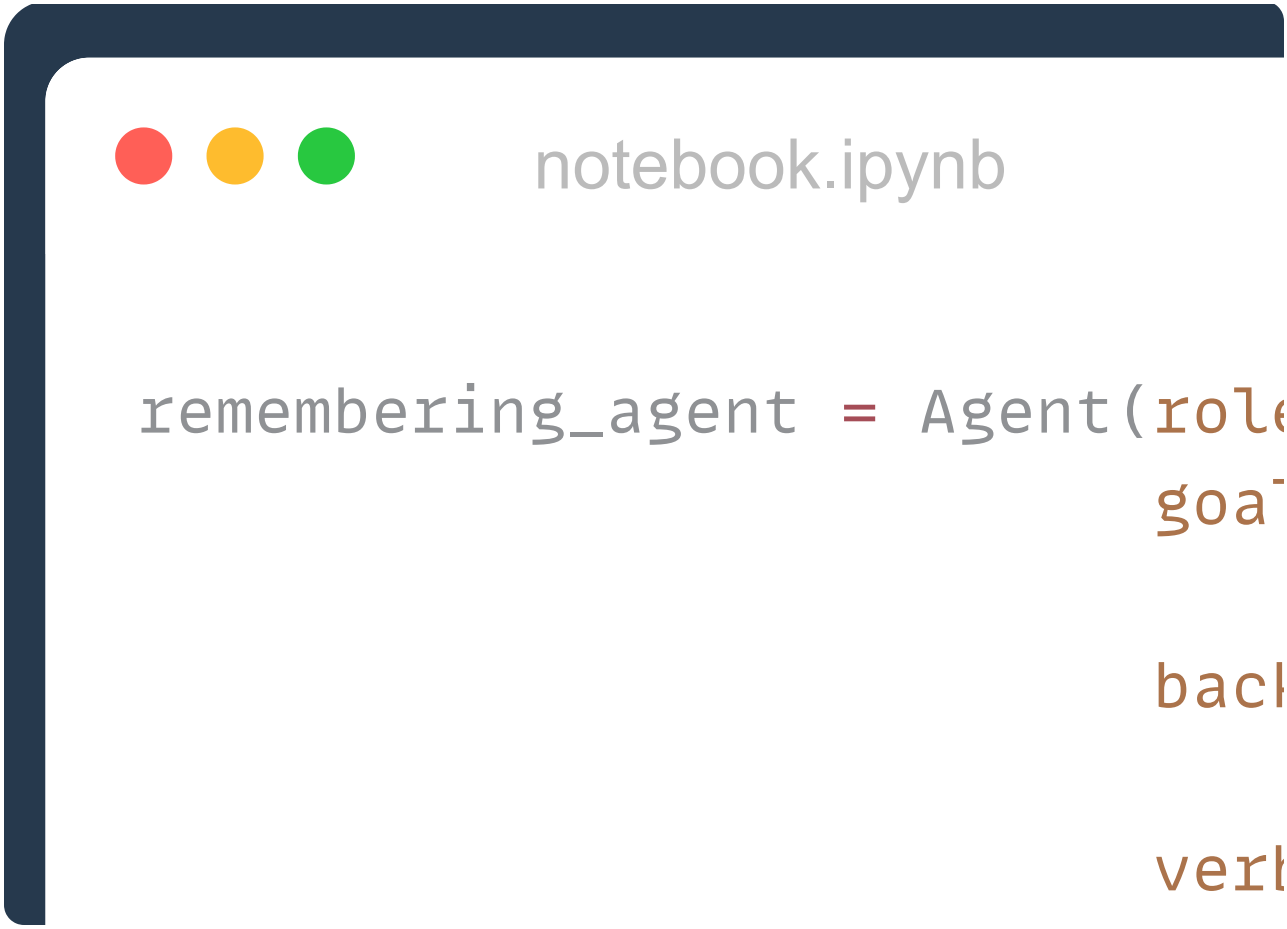
By default, CrewAI uses vector databases like ChromaDB under the hood for `RAGStorage`. This allows storing memory chunks as embeddings and later retrieving them based on semantic similarity.

Next up, create a long-term memory storage that will be shared between sessions:



- Specify a file path for memory storage: `"./agent1_entity_memory.db"`. This is where our memory will live between sessions.
- Instantiate an `EntityMemory` object and tell it to use this file via `RAGStorage`.

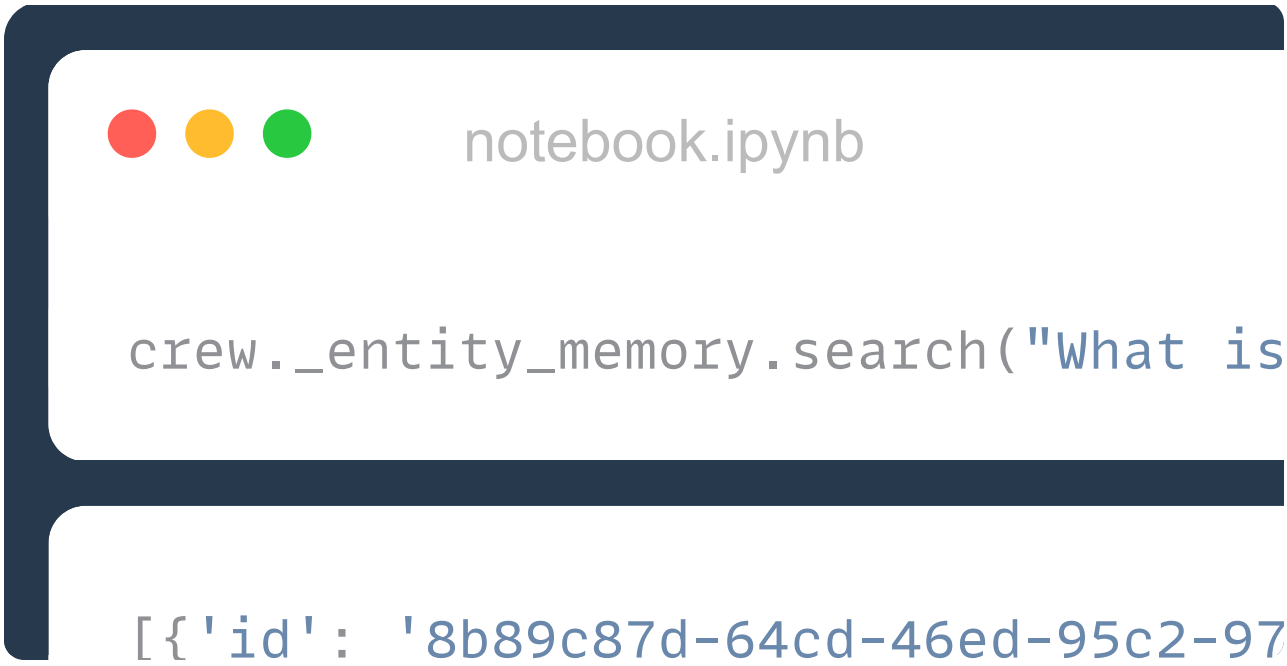
Next, we have a basic `Agent`, a `Task` and a `Crew` instance that doesn't use tools, doesn't delegate—its only job is to respond intelligently while remembering what the user tells it:



How entity memory works behind the scenes:

- Every time the agent processes a statement like: "*My favorite color is #46778F.*"
- CrewAI will extract the entity (here, likely "user" or an implicit "me") and associate the information—*"favorite color is #46778F"*—as a memory tied to that entity.
- Internally, it chunks and embeds this statement using the current LLM's embedding model and stores it using `RAGStorage`.
- Later, if a task queries something like "What is my favorite color?", CrewAI can run a similarity search across memory for that entity to retrieve the most relevant entities—just like how RAG is used for document retrieval.

For instance, when we used the `search` method on the `_entity_memory` object of the crew, we get the following output:



The `.search()` method returns a list of matching memory chunks, ranked by relevance score, and each one is a dictionary with information about:

- The chunk ID
- The metadata, including any relationships or entity tags
- The context, which holds the raw text that was stored

- The score, which indicates how similar the stored memory is to your query

After retrieving these chunks, CrewAI will:

1. Rank them by relevance (via score).
2. Select the most relevant ones (usually top 3–5).
3. Inject them into the context window for the current agent/task execution.

This context contains all the relevant details to produce an accurate response.

In a real production use-case, you might use entity memory to keep track of many kinds of entities.

For instance, a sales agent could maintain memory of different products and their specs, or a story-writing agent could keep a character list with descriptions to maintain consistency.

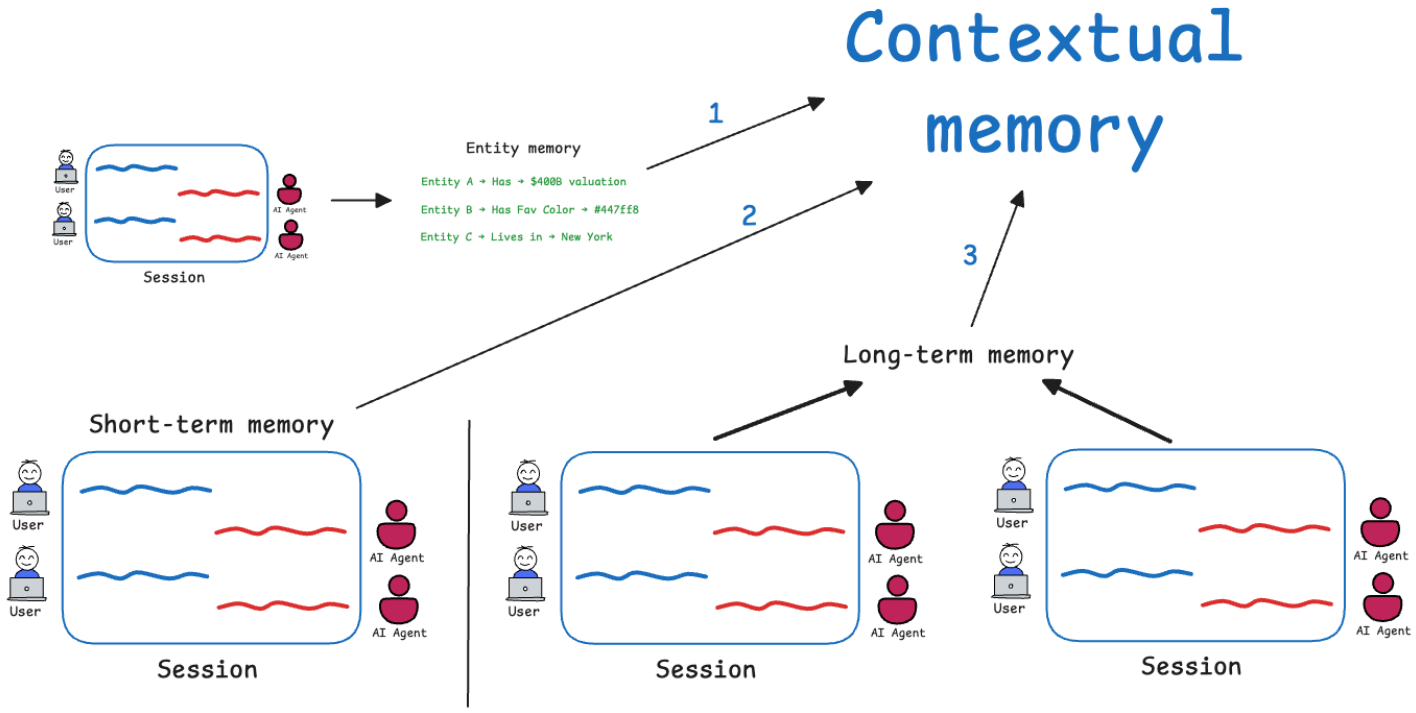
CrewAI handles the heavy lifting of organizing these details. As the developer, you mainly need to ensure memory is on, and perhaps provide clear cues when introducing new entities (as we did by clearly stating facts about Alice and Bob separately).

The system uses the memory to maintain a richer context, leading to deeper understanding of key entities over time.

#4) Contextual Memory

While short-term, long-term, and entity memory each serve a specific purpose, CrewAI introduces a contextual memory mechanism that blends these memory types to present the most relevant information to the agent at any given point in time.

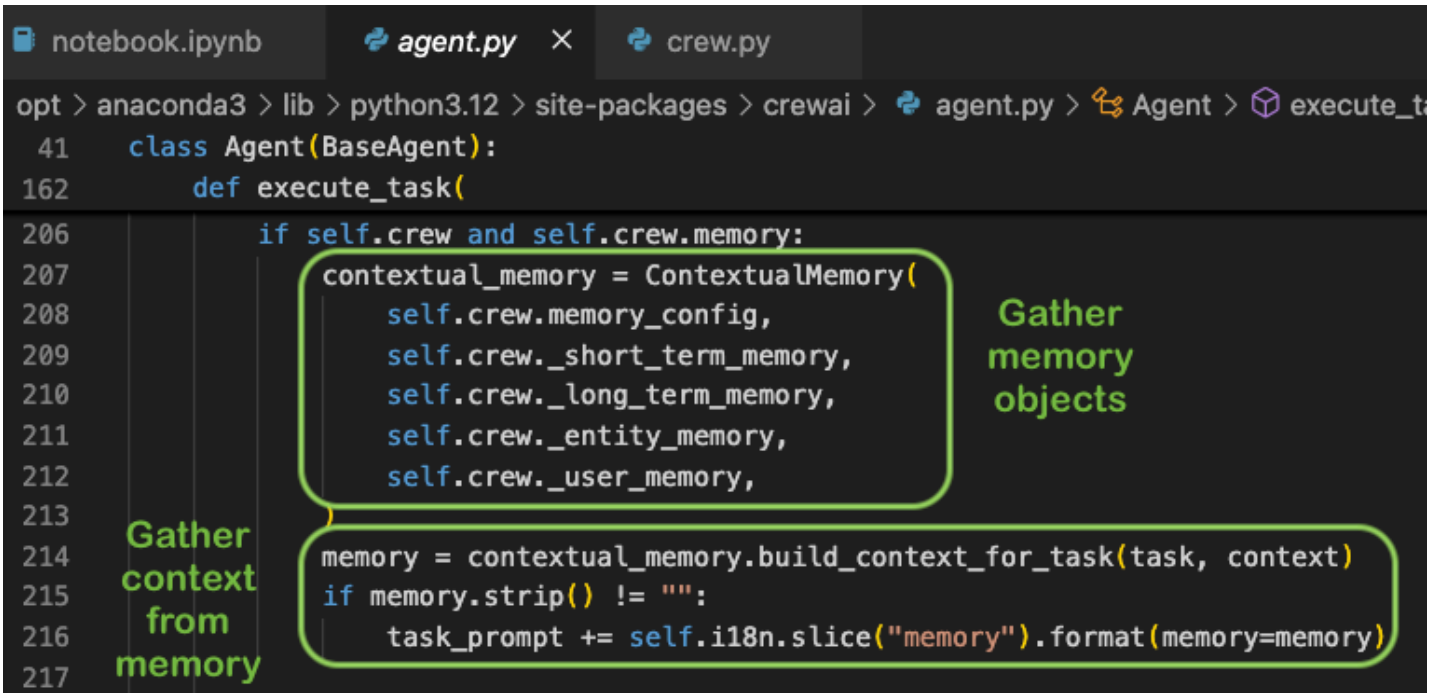
Think of contextual memory as an intelligent retrieval layer. Instead of the agent separately pulling from three different memory stores, CrewAI automatically merges relevant pieces from:



- Short-term memory (recent task interactions),
- Long-term memory (persisted insights from past sessions), and
- Entity memory (facts associated with specific people, places, or things)

This combined context is then injected into the prompt window of the agent's LLM, prioritized by relevance. So regardless of where or when a fact was introduced—this session, a previous one, or within a named entity—the agent receives the most useful and contextually aligned information for its current task.

If you recall the last part, we saw that in the `execute_task` method, which, as the name suggests, is responsible for executing a task with the agent, line 207-212 defines all the memory objects together into a single `contextual_memory` object:



Next, in 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.

Finally, in lines 215-216, if the relevant context exists (`if memory`), it is added to the `task_prompt` object.

So essentially, contextual memory is just a way to declare all types of memories together.

This brings us to the last memory type (which is defined in line 212 below)—User memory.

notebook.ipynb | agent.py | crew.py

opt > anaconda3 > lib > python3.12 > site-packages > crewai > agent.py > Agent > execute_t

```
41    class Agent(BaseAgent):
162       def execute_task(
206           if self.crew and self.crew.memory:
207               contextual_memory = ContextualMemory(
208                   self.crew.memory_config,
209                   self.crew._short_term_memory,
210                   self.crew._long_term_memory,
211                   self.crew._entity_memory,
212                   self.crew._user_memory,
213               )
214               memory = contextual_memory.build_context_for_task(task, context)
215               if memory.strip() != "":
216                   task_prompt += self.i18n.slice("memory").format(memory=memory)
217
```

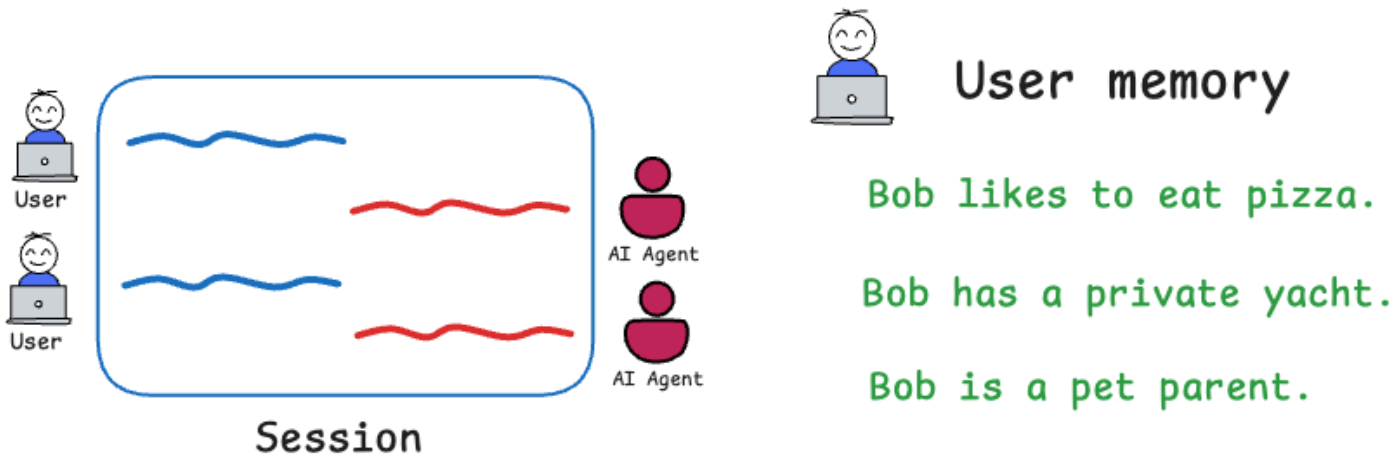
Gather memory objects

Gather context from memory

#5) User Memory

Not all memory is general—often we want memory scoped per user or at a persona level.

User memory in CrewAI is used for personalization: it stores user-specific information and preferences to enhance the user's experience.



For example:

- If user Alice always asks the agent to summarize news about sports, the agent could learn Alice's interest in sports and proactively tailor future interactions

(like highlighting sports news for her).

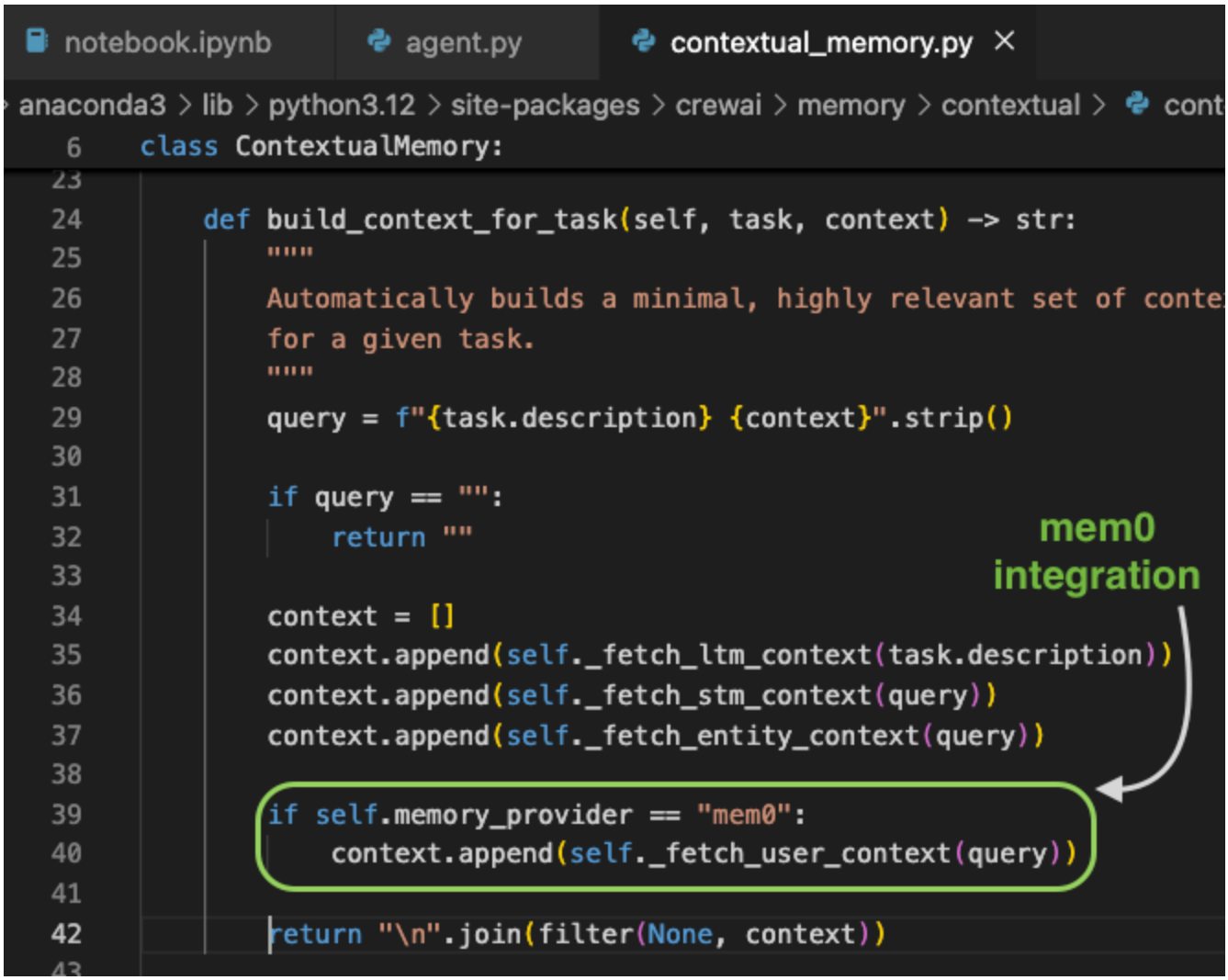
- If another user, Bob, uses the same system, Bob's preferences and history should be remembered separately from Alice's.

User memory allows the agent to distinguish and remember individual users' data.

How do we enable and use user memory in CrewAI?

In practice, user memory is often implemented by associating a `user_id` with the Agent's memory system. CrewAI can integrate with external memory services for this purpose.

One such integration is with **Mem0**, a self-improving memory API that CrewAI supports for user-specific memory storage. If you look closely in the `build_context_for_task` method, this is exactly what you will find when producing a user-specific context:



When configuring a Crew, you can pass a `memory_config` that specifies a memory provider and a user identifier.

This is demonstrated below:

notebook.ipynb

```
from crewai import Crew, Process
from mem0 import MemoryClient

os.environ["MEM0_API_KEY"] = "<your-api-key>"

user_memory = {"provider": "mem0",
               "config": {"user_id": "Paul"},
               "user_memory" : {}}

crew = Crew(agents=my_agents,
            tasks=my_tasks,
            process=Process.sequential,
            memory=True,
            memory_config=user_memory
        )
```

This effectively tells CrewAI to “use this memory service keyed to this particular user.” Then any memories stored will be tied to that user ID, and the agent can retrieve them only when running with the same ID.

While we won't get into the specific details of this, we want you to consider this an assignment. Here's what you need to do:

- Get an API key of Mem0 here: https://mem0.ai/
- Define an Agent, Task and Crew just like we did it earlier.
- In the first task execution, give the Agent a hyper-specific message about you that it should use to generate a response and so that you can see if it's obeying your specific details—some detail about your Pet, the locations you like, etc.
- Next, give it some general tasks that are unrelated to the details you provided above.
- Finally, ask it something related to the hyper-specific task you gave it earlier. For instance, if you mentioned something like "your favourite location is XYZ", ask it to "plan a trip to your favourite location" (here, do not mention "XYZ" so that it can retrieve it from user memory.

Let us know if you face any issues.

Coming back to user memory...

notebook.ipynb

```
from crewai import Crew, Process
from mem0 import MemoryClient

os.environ["MEM0_API_KEY"] = "<your-api-key>"

user_memory = {"provider": "mem0",
               "config": {"user_id": "Paul"},
               "user_memory" : {}}

crew = Crew(agents=my_agents,
            tasks=my_tasks,
            process=Process.sequential,
            memory=True,
            memory_config=user_memory
        )
```

By using user IDs, we isolate memories between users. If we had another user, say Bob, we would run a crew with `user_id: "bob"` and that would maintain a separate memory storage for Bob. Alice's information wouldn't leak into Bob's session, and vice versa.

This is crucial for personalization as well as data privacy. Each user effectively gets their own long-term (and short-term) memory space.

It's worth noting that under the hood, user memory might utilize a combination of short-term and long-term strategies (for example, Mem0 can store immediate conversation context as well as longer-term facts per user).

From a developer's perspective, the main step is providing a consistent user identifier.

In a production scenario, you might use a unique customer ID instead of a name (like above) or username as the `user_id` when constructing the crew for a user session.

CrewAI handles the rest, ensuring that any memory saved is tagged to that user and available next time the same user interacts.

If you cannot or don't want to use an external service like Mem0, you can still implement user-specific memory by managing separate memory stores yourself.

For instance, you could maintain separate long-term memory databases for each user (e.g., `user123.db` for user with ID 123) and choose the path dynamically based on the user logging in.

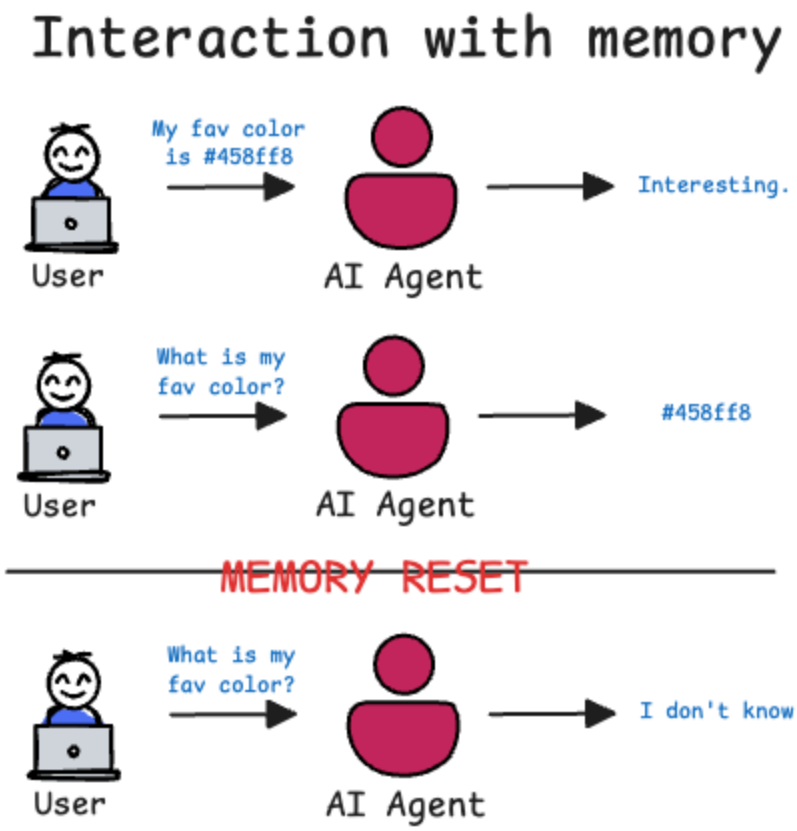
The concept remains the same: key the agent's memories by user so that interactions are personalized.

CrewAI's built-in integration simply streamlines this by using the `memory_config` approach shown above.

Reset memory

As your agents run and accumulate memories, there will be times you might want to reset or clear those memories.

For instance, to wipe the slate clean for a new scenario or to delete sensitive data.



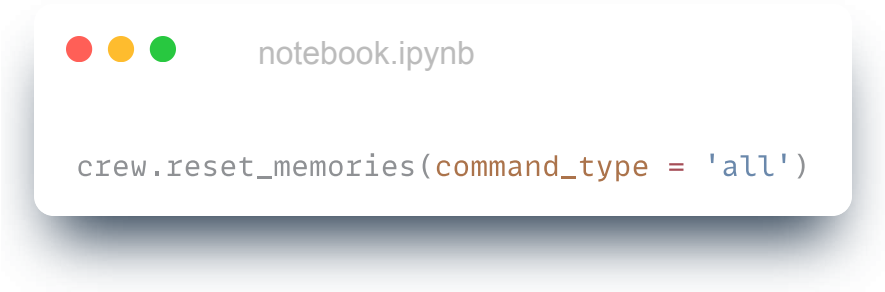
CrewAI provides convenient ways to reset memory either via code or the command-line interface.

You can clear specific types of memory or wipe everything.

For instance, if you only want to clear the short-term memory (recent interactions) but keep long-term learnings, you can do that.

Or you can clear long-term memory if you want the agent to “forget” all accumulated knowledge.

Using the Crew object in code, you can call the `reset_memories` method with a parameter specifying which memory to reset. Here are the options and how to use them:



- `crew.reset_memories(command_type='short')` —Resets the short-term memory for the crew (this clears the recent interaction vector store).
- `crew.reset_memories(command_type='long')` —Resets the long-term memory (this might delete or archive the long-term storage, essentially forgetting all past stored results).
- `crew.reset_memories(command_type='entities')` —Clears the entity memory records.
- `crew.reset_memories(command_type='kickoff_outputs')` —Resets the latest kickoff output of a Task.
- `crew.reset_memories(command_type='all')` —Completely wipes all memory types for that crew (short-term, long-term, entity, etc., all at once).

This ensures that the next time you run the crew (or a new crew with the same storage paths), it starts fresh with no prior memory.

That said, use this with caution, because once cleared, that learned information is gone unless you have it saved elsewhere.

Sample demo

Let's look at it in action.

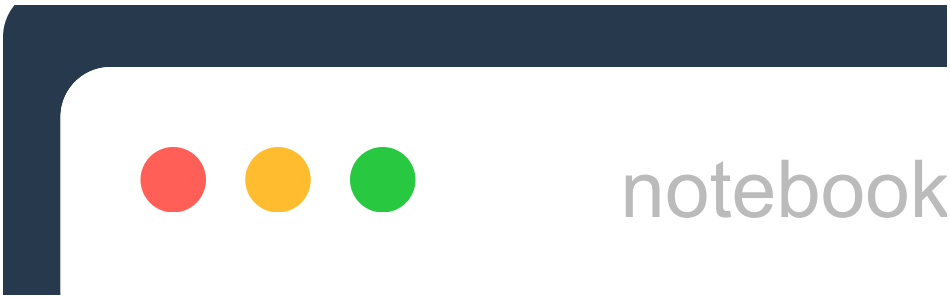
Below, we define an Agent, it's Task and a Crew:

notebook.ipynb

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

# Session 1: Agent learns a fact
remembering_agent = Agent(role="My Personal Remembering Agent",
                          goal="""You are a assistant that can remember
                                  the user's information.""",
                          backstory="""You are a assistant that can
                                       remember the user's information.""",
                          verbose=True)

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

crew = Crew(
    agents=[remembering_agent],
    tasks=[task],
    process=Process.sequential,
    memory=True
)
```

Memory enabled Crew

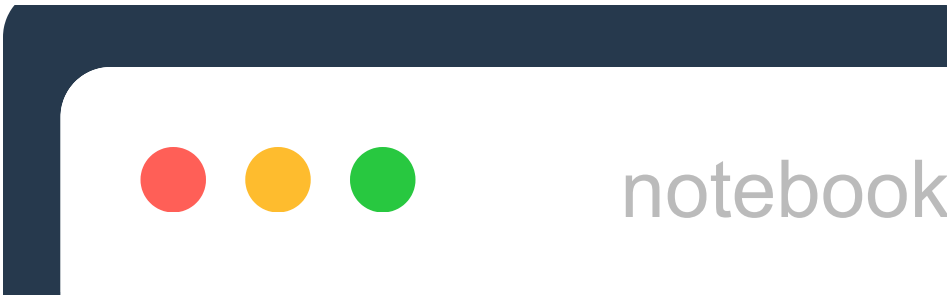
Next, we kickoff the Crew and tell it about our favourite colour:



This produces the following output:

```
Overriding of current TracerProvider is not allowed
# Agent: My Personal Remembering Agent
## Task: Handle this task: My favorite color is #46778F.


# Agent: My Personal Remembering Agent
## Final Answer:
Your favorite color is #46778F.
```

Moving on, let's ask it a question related to this:



As expected, the Agent uses its memory to look through the relevant context and produces a response:

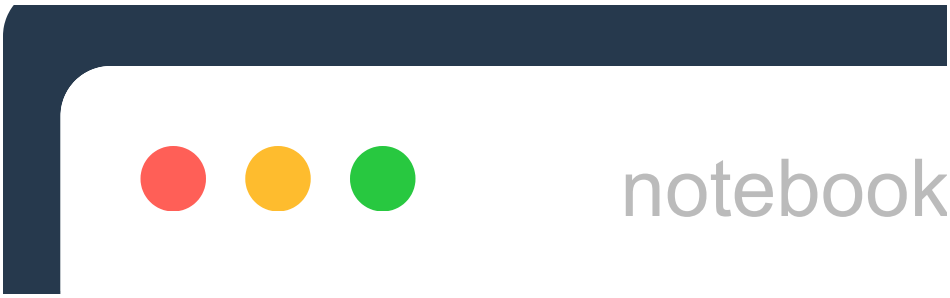
```
# Agent: My Personal Remembering Agent
## Task: Handle this task: What is my favorite color?


# Agent: My Personal Remembering Agent
## Final Answer:
Your favorite color is #46778F.
```

But now, let's reset the memory:

notebook.ipynb

```
crew.reset_memories(command_type = 'all')
```

Now if we ask it the same question again (shown below):



...we get the following output:

```
# Agent: My Personal Remembering Agent
## Task: Handle this task: What is my favorite color?
ERROR:root:Error during short_term save: 'NoneType' object has no attribute 'add'


# Agent: My Personal Remembering Agent
## Final Answer:
Your favorite color is not stored in my memory. Please tell me what your favorite color is so I can remember it
```

Context not found

When would you reset memory?

One case is during development and testing.

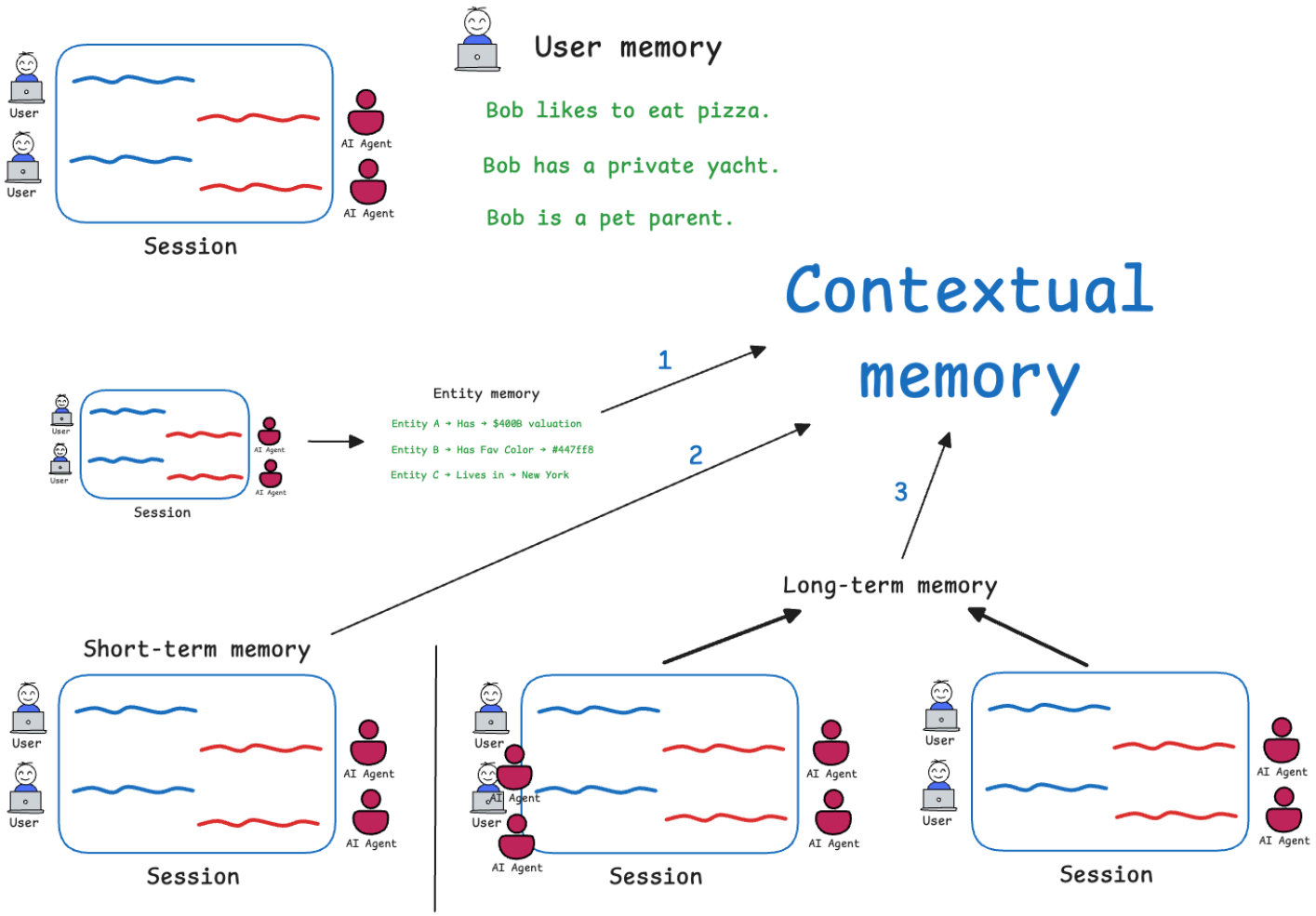
You might want to rerun scenarios from scratch to see how the agent behaves without earlier runs influencing it.

Another case is if an agent has accumulated incorrect or outdated information in long-term memory; you might choose to clear it to avoid confusion.

Also, if you have an application with multiple distinct projects or domains using one CrewAI installation, you might reset memory when switching context to ensure no cross-talk between projects (though using separate memory storage paths is another way to isolate them).

Conclusion

Memory is a cornerstone of creating truly agentic AI systems with CrewAI. In this crash course, we covered the different types of memory CrewAI offers and their uses:



- 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 type of memory addresses a different need, and together they give your agents a powerful ability to remember and learn.

As a developer, you don't have to manage every detail of these memories – much of it is handled by CrewAI once you enable the system.

But knowing how to configure and leverage them lets you build more effective agents.

When deciding which memory types to use, consider your use case: If you need an agent to carry on a complex conversation or multi-step task, short-term (and contextual) memory is essential.

If you want the agent to improve over time or retain information between independent runs, enable long-term memory.

For applications serving many users, integrate user memory so each user's context remains separate and persistent.

And if your task involves many distinct topics or entities, entity memory will help the agent keep facts organized.

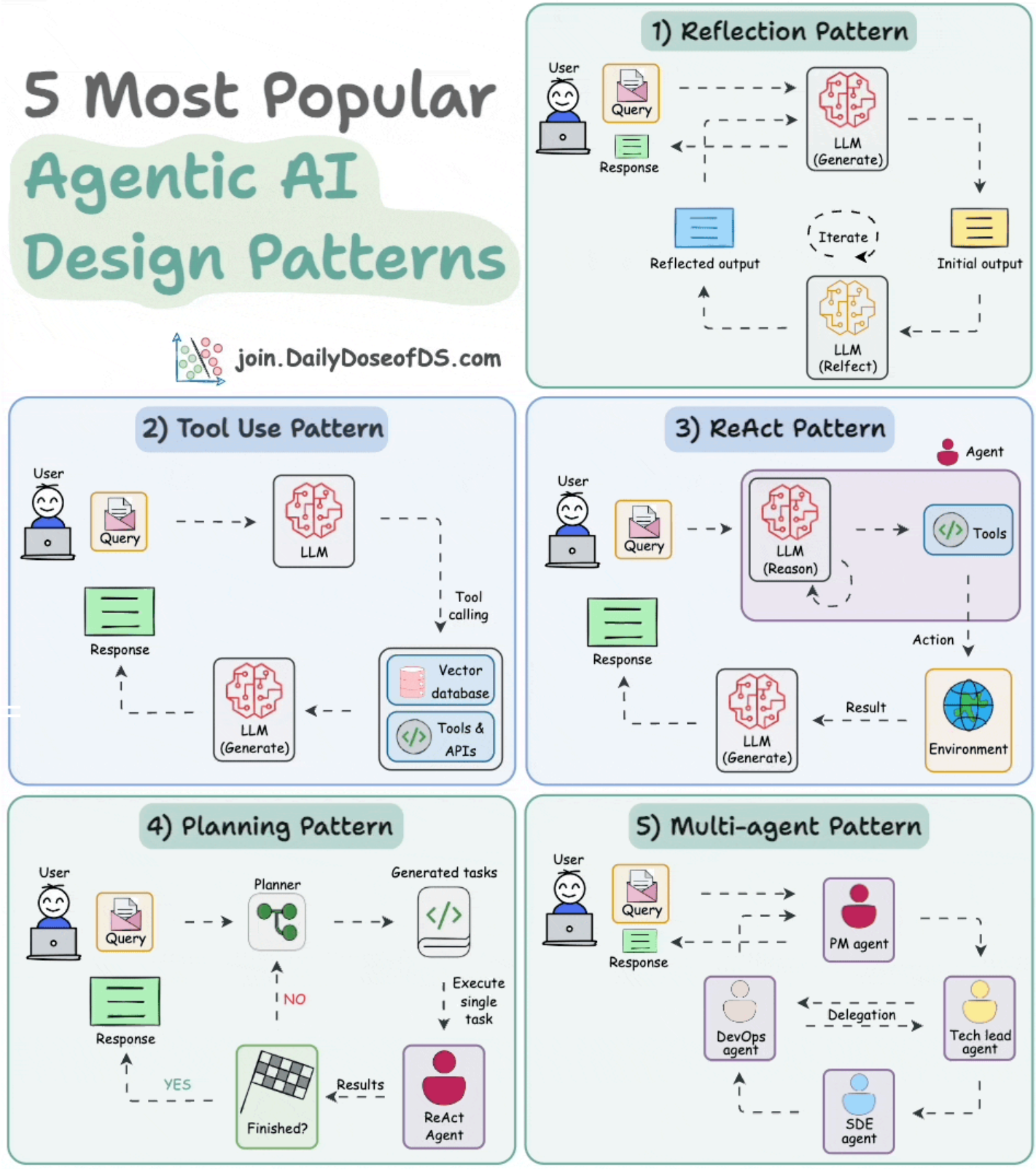
Often, you will simply enable all memory (the default when `memory=True`), and let the agent benefit from the full spectrum.

CrewAI's contextual memory automatically combines short-term, long-term, and entity memories to present the right info at the right time.

With that, we come to an of this article on Memory for AI agents.

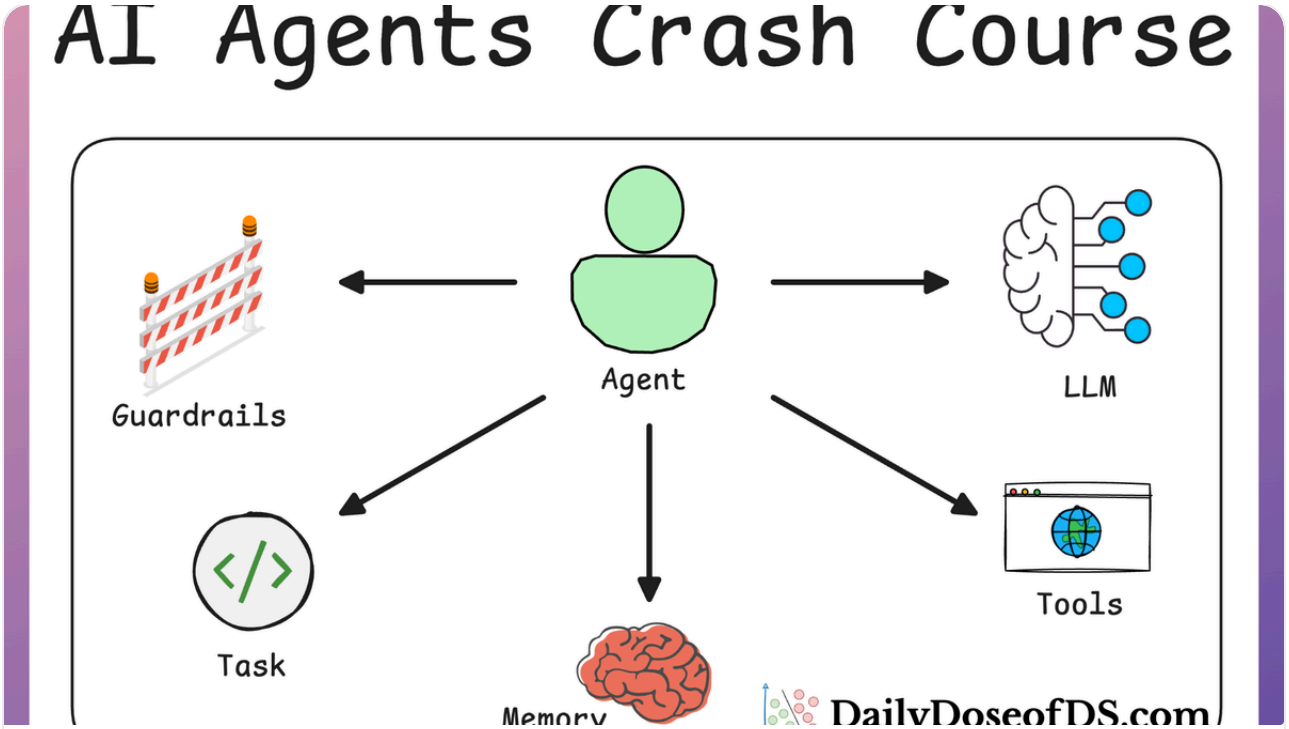
In the upcoming parts, we have several other advanced agentic things planned for you:

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



- and many many more.

Read the next part of this crash course here:



Implementing ReAct Agentic Pattern From Scratch

AI Agents Crash Course—Part 10 (with implementation).

Daily Dose of Data Science • Avi Chawla

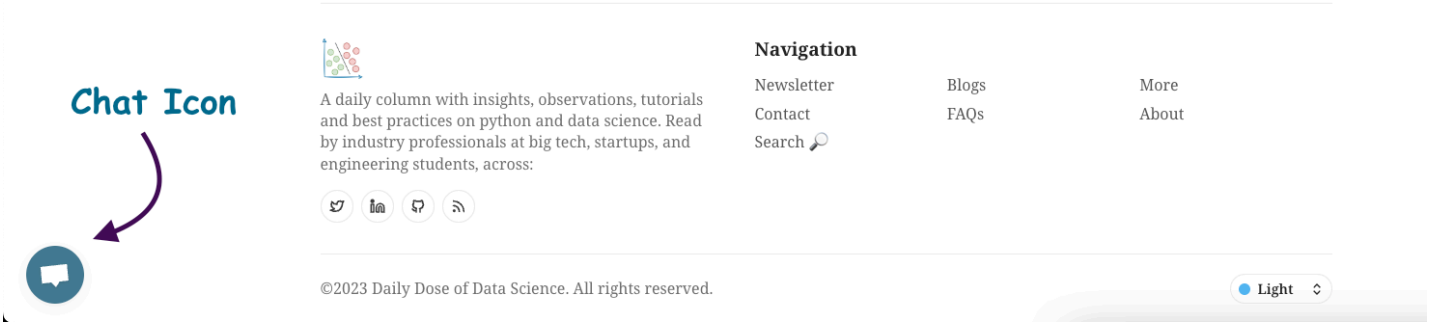
As always, thanks for reading!

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

Navigation