# Can Agents get nostalgic about the past?

Lesson 3: Unlocking Ava's memories

MIGUEL OTERO PEDRIDO
FEB 19, 2025

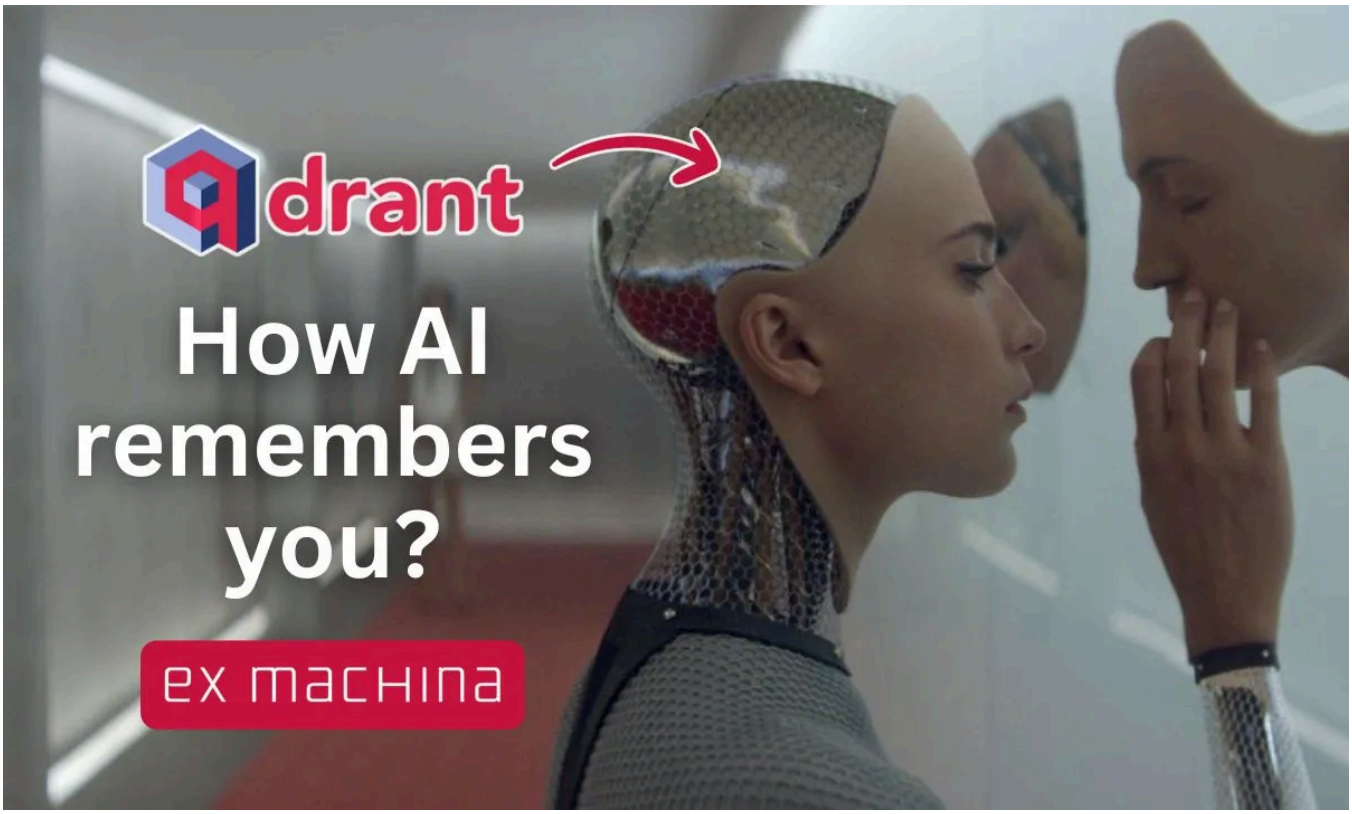


Talking to an Agent **without memory** is like dealing with the main character from **“Memento”** - you say your name, and one message later, it’s asking again ...

Ava was no different. We needed a memory module that made her feel real - one t actually keeps up with the conversation and remembers every relevant detail abou you. Whether you mentioned it two minutes ago or two months ago, she’s got it.

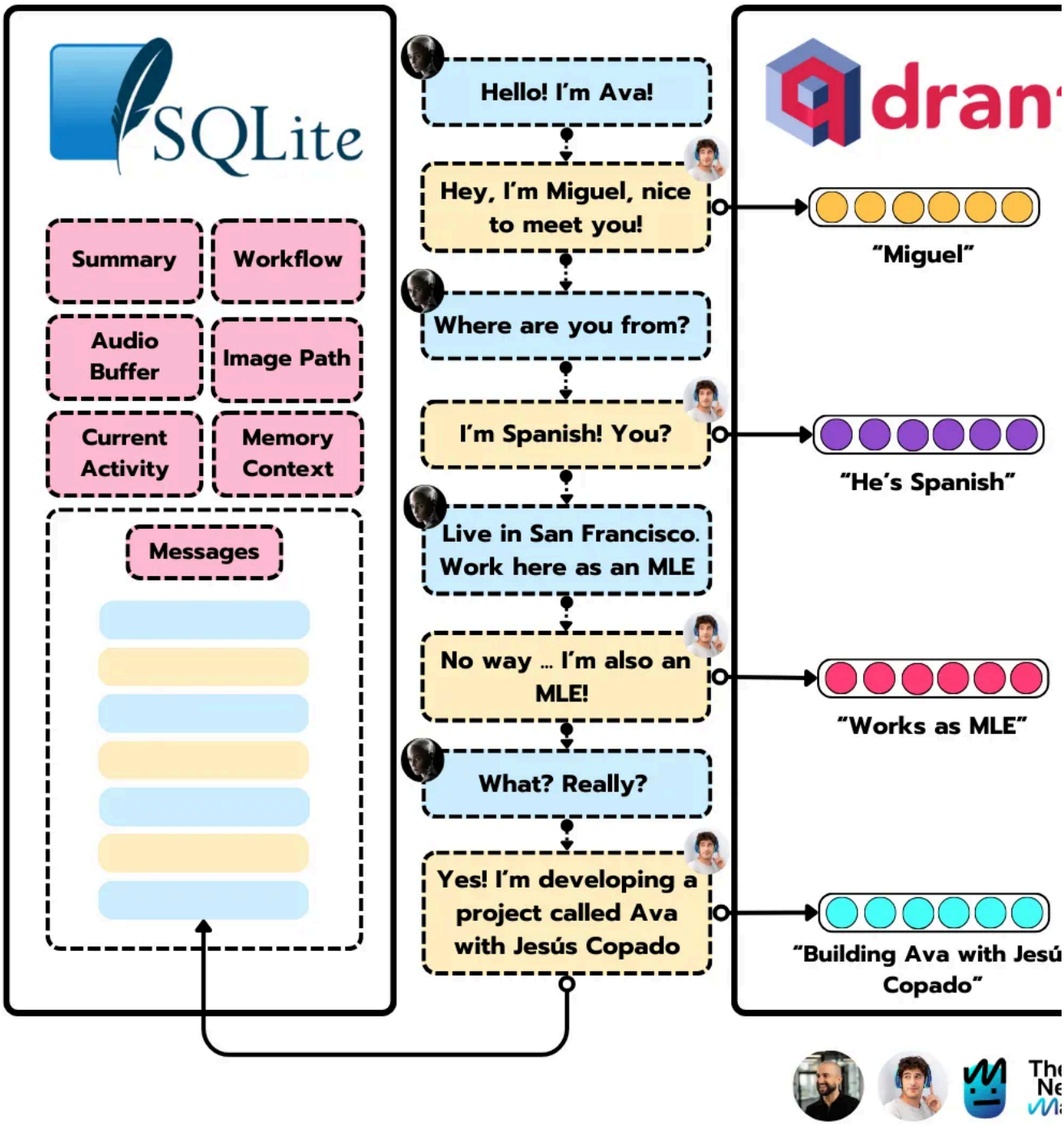
Ready for Lesson 3? Let’s give Ava a memory boost! 👇

> [Check the code here!](#) 👨‍💻



> This is the **third lesson** of **“Ava: The Whatsapp Agent”** course. This lesson bu
> on the theory and code covered in the previous ones, so be sure to check them
> if you haven’t already!
>
> - Lesson One: Project Overview
> - Lesson Two: Dissecting Ava’s brain

# Ava’s Memory Overview




A general overview of the two memory systems: short-term (Sqlite) and long-term (Qdrant)

Let's start with a diagram to give you a big-picture view. As you can see, there are main memory "blocks" - one stored in a SQLite database (left) and the other in a Qdrant collection (right).

If this isn't clear yet, don't worry! We'll go through each memory module in detail i next sections.

## 💠 Short-term memory

The block on the left represents the short-term memory, which is stored in the LangGraph state and then persisted in a SQLite database. LangGraph makes this process simple since it comes with a built-in checkpointer for handling database storage.

In the code, we simply use the **AsyncSqliteSaver** class when compiling the graph This ensures that the LangGraph state checkpoint is continuously saved to SQLite You can see this in action in the code below.

```
...

# Process message through the graph agent
async with AsyncSqliteSaver.from_conn_string(
    settings.SHORT_TERM_MEMORY_DB_PATH
) as short_term_memory:
    graph = graph_builder.compile(checkpointer=short_term_memory)
    await graph.ainvoke(
        {"messages": [HumanMessage(content=content)]},
        {"configurable": {"thread_id": session_id}},
    )

    # Get the workflow type and response from the state
    output_state = await graph.aget_state(
        config={"configurable": {"thread_id": session_id}}
    )

workflow = output_state.values.get("workflow", "conversation")
response_message = output_state.values["messages"][-1].content

....
```

Ava's state is a subclass of LangGraph's MessageState, which means it inherits a **messages** property. This property holds the history of messages exchanged in th conversation - essentially, **that's what we call short-term memory!**

Integrating this short-term memory into the response chain is straightforward. We use LangChain's **MessagesPlaceholder** class, allowing Ava to consider past interactions when generating responses. This keeps the conversation smooth and coherent.

```python
def get_character_response_chain(summary: str = ""):
    model = get_chat_model()
    system_message = CHARACTER_CARD_PROMPT

    if summary:
        system_message += (
            f"\n\nSummary of conversation earlier between Ava and the user: {summary}"
        )

    prompt = ChatPromptTemplate.from_messages(
        [
            ("system", system_message),
            MessagesPlaceholder(variable_name="messages"),
        ]
    )

    return prompt | model | AsteriskRemovalParser()
```

Simple, right? Now, let's get into the interesting part: the **long-term memory.**

## 💠 Long-term memory

Long-term memory isn't just about saving every single message from a conversat far from it 😅. That would be impractical and impossible to scale. Long-term mem works quite differently.

Think about when you meet someone new - you don't remember every word they right? You only retain **key details**, like their **name**, **profession**, **where they're fro** or **shared interests**.

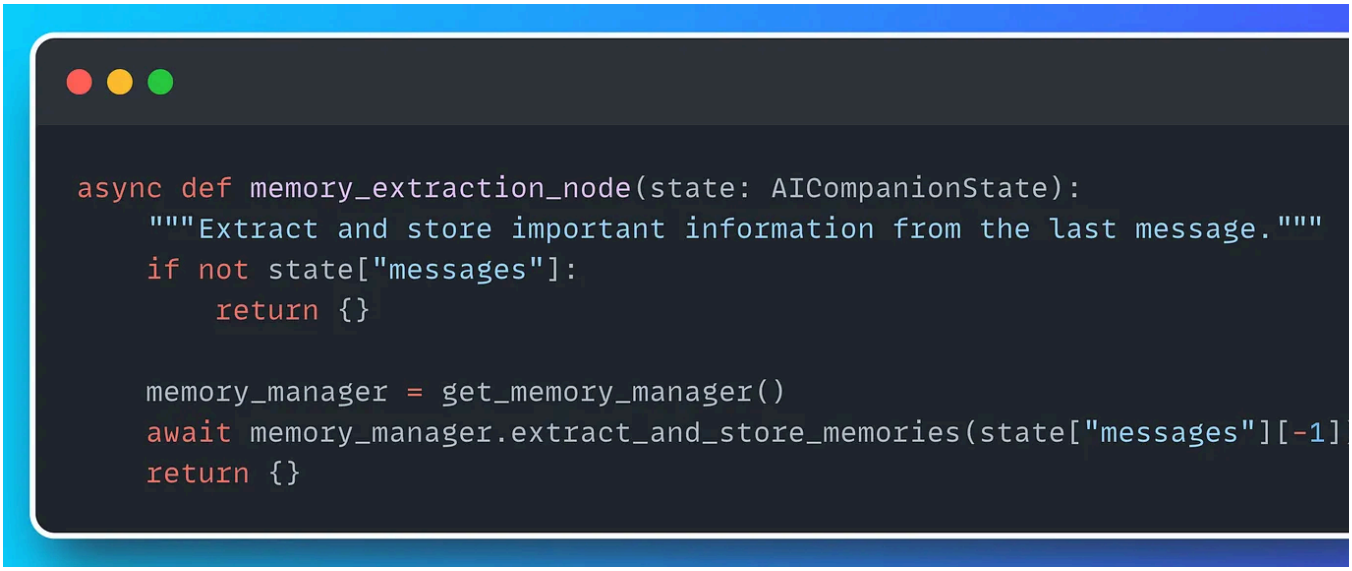
That's exactly what we wanted to replicate with Ava. **How?** 🤔

By using a Vector Database like Qdrant, that lets us store relevant information fror conversations as embeddings. Let's break this down in more detail.

## 🔶 Memory Extraction Node

Remember the other day when we talked about the different nodes in our LangGr workflow? The first one was the **memory_extraction_node**, which is responsible identifying and storing key details from the conversation.

That's the first essential piece we need to get our long-term memory module up a running! 💪



```
async def memory_extraction_node(state: AICompanionState):
    """Extract and store important information from the last message."""
    if not state["messages"]:
        return {}

    memory_manager = get_memory_manager()
    await memory_manager.extract_and_store_memories(state["messages"][-1]
    return {}
```

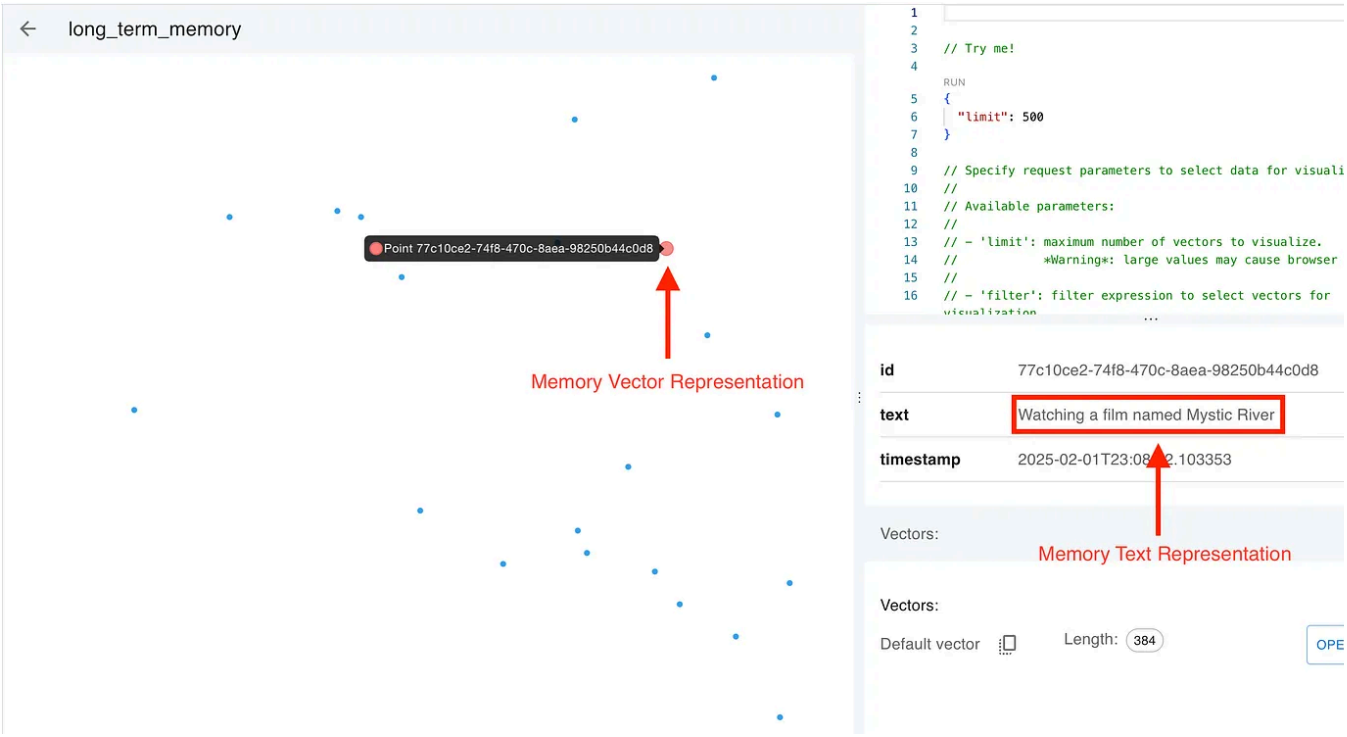
Using our MemoryManager class, this node will retrieve relevant information from the conversation and store it in Qdrant

## 🔶 Qdrant

As the conversation progresses, the **memory_extraction_node** will keep gatheri more and more details about you.

If you check your Qdrant Cloud instance, you'll see the collection gradually filling with "memories".

> Don't know how to configure Qdrant Cloud? There's a detailed guide on setting up here!



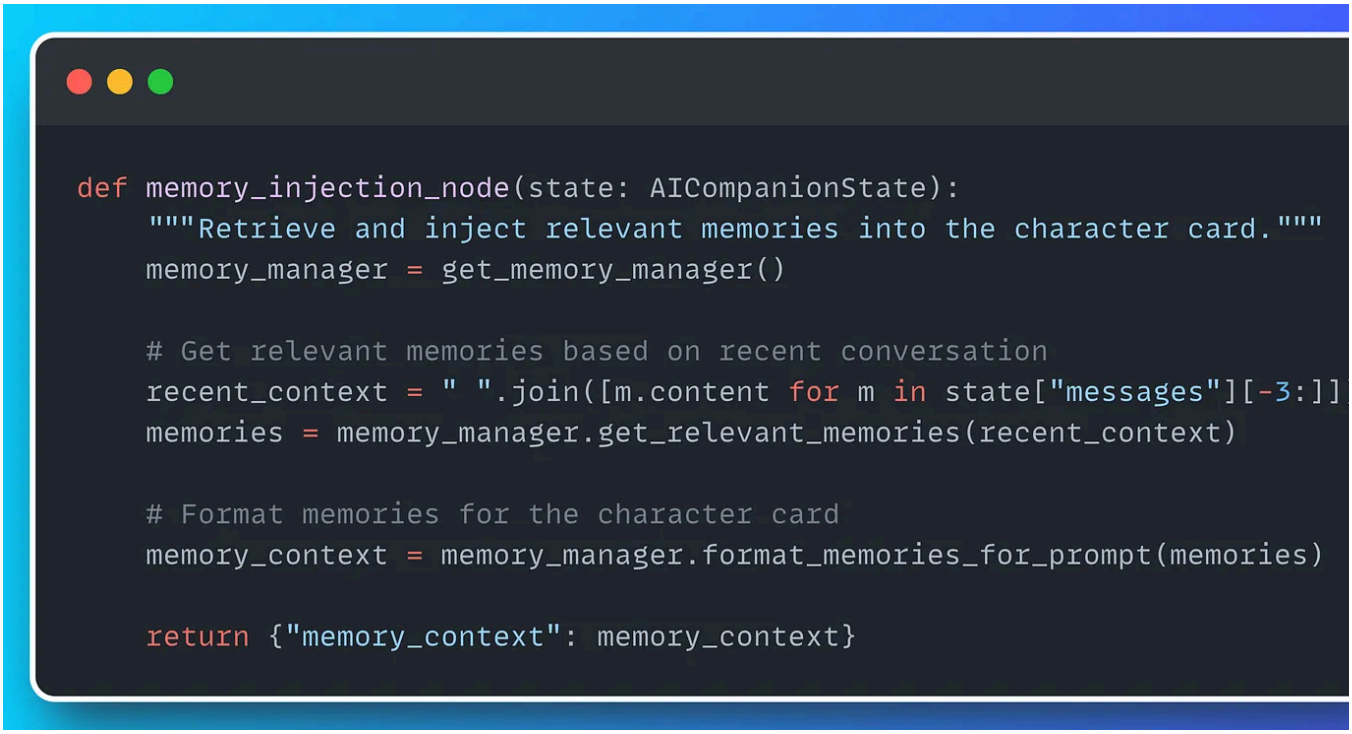

tSNE visualization of Ava's memories

In the example above, you can see how Ava remembers I was watching a film nam "Mystic River" (great movie, by the way). The memories are stored in Qdrant as embeddings by the **memory_extraction_node** using the **all-MiniLM-L6-v2** mod as can be seen in our VectorStore class.

## 🔶 Memory Injection Node

Now that all the memories are stored in Qdrant, how do we let Ava use them in he conversations?

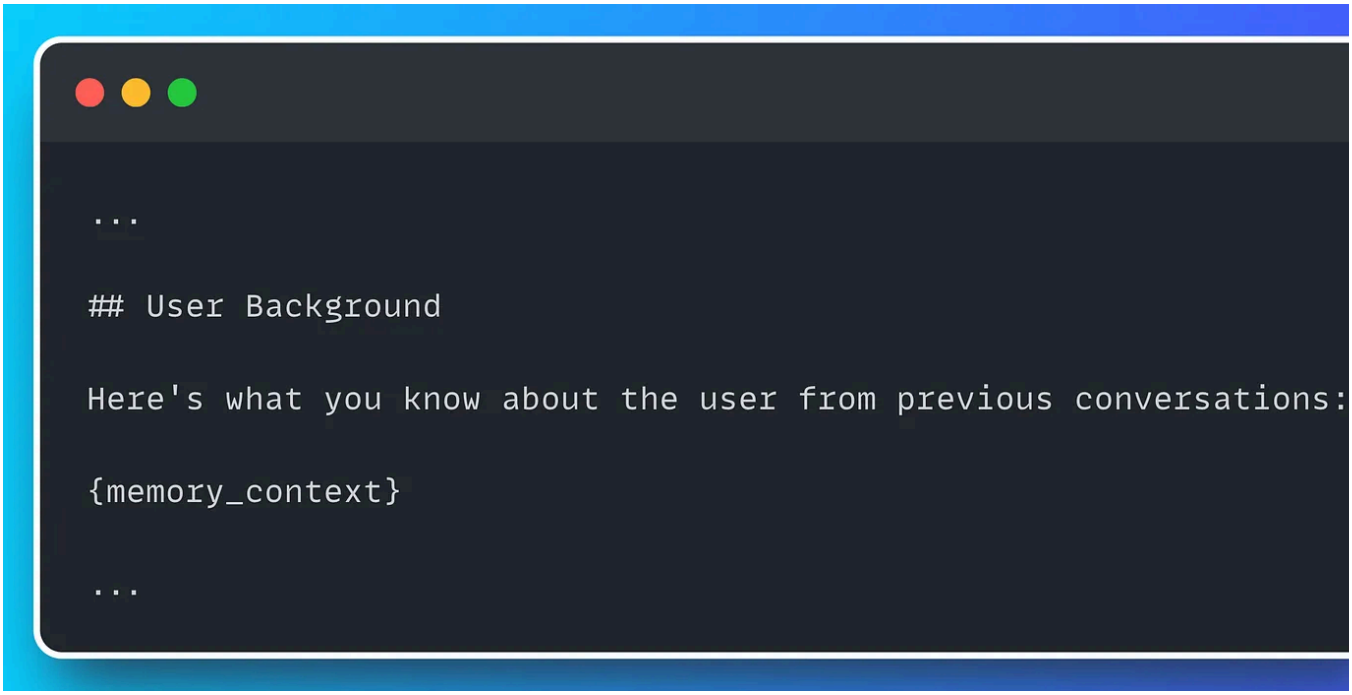
It's simple! We just need one more node: the **memory_injection_node.**



```python
def memory_injection_node(state: AICompanionState):
    """Retrieve and inject relevant memories into the character card."""
    memory_manager = get_memory_manager()

    # Get relevant memories based on recent conversation
    recent_context = " ".join([m.content for m in state["messages"][-3:]])
    memories = memory_manager.get_relevant_memories(recent_context)

    # Format memories for the character card
    memory_context = memory_manager.format_memories_for_prompt(memories)

    return {"memory_context": memory_context}
```

This node uses the MemoryManager class to retrieve relevant memories from Qdr essentially performing a vector search to find the top-k similar embeddings. Then transforms those embeddings (vector representations) into text using the **format_memories_for_prompt** method.

Once that's done, the formatted memories are stored in the **memory_context** property of the graph. This allows them to be parsed into the Character Card Pror the one that defines Ava's personality and behaviour.



```
...

## User Background

Here's what you know about the user from previous conversations:

{memory_context}

...
```

**And that's all for today!**

Just a quick reminder - **Lesson 4** will be available next **Wednesday, February 26**
Plus, don't forget there's a **complementary video lesson** on **Jesús Copado's YouTube channel**.

> We **strongly recommend** checking out **both resources** (**written** lessons and **video** lessons) to **maximize** your **learning experience**! 🙂

**And don't worry. Ava will remember you until next Wednesday!** 🤣

Thanks for reading The Neural Maze! Subscribe for free to receive new posts and support my work.

19 Likes · 2 Restacks

← Previous Next

## Discussion about this post

Comments Restacks

Write a comment...