Build AI Agent Memory From Scratch — Tutoria For Dummies

ZACHARY HUANG
MAR 24, 2025

> *Ever wondered why some chatbots remember your name days later, while others forget you said 5 minutes ago? This guide explains AI memory in super simple terms — no te background needed!*

Have you ever told a chatbot your name, only for it to ask again in the same conversation? Or bee :? I break down how AI r

Great news: it's way discover:

- The shockingly simple trick behind AI memory
- How chatbots keep track of what you've told them
- Why good memory makes the difference between helpful and frustrating AI

For this guide, we'll use **Pocket Flow** — a tiny framework (just 100 lines!) that cuts through all the fancy jargon to show you how AI memory really works. While mos tools hide all the important stuff under complicated code, PocketFlow puts everyt right in front of you so you can actually understand it.

Want to see the working code? *You can check out and run the complete implementatio GitHub: PocketFlow Chat Memory.*

Why Learn Memory with Pocket Flow

Most AI tools are like pre-built furniture with hidden screws and hard-to-read instructions. Pocket Flow is different — it's like a simple DIY starter kit with just **lines of code** that includes only the essential tools you need to build something re and useful!

What makes this perfect for learning:

- **Basic tools only**: Just the essential tools you actually need, not a confusing workshop full of
- **Clear instructio** ori with pictures for
- **Expand at your** the you're comfortable

That's it! Everything else is just details around these core concepts.

The Simple DIY Kit from Pocket Flow

Imagine your AI as a **bustling coffee shop**. PocketFlow provides the following building blocks:

- Nodes are like different **stations** (taking orders, checking recipes, brewing cof archiving receipts).
- Flow is like the **daily routine** that keeps each station running smoothly.
- Shared Store is like your **master order binder**, keeping track of all current orc recipes, and past receipts.

In our coffee shop system:

1. Each station (Node) has three simple jobs:

- **Prep**: Gather what you need (like the right cup or recipe).
- **Exec**: Perform the main task (like brewing the drink).
- **Post**: Update the binder and decide what to do next (like archiving an old rece

2. The shop's routine (Flow) moves between stations based on needs:

- "If we need a recipe, look it up in the binder."
- "If we have too many orders on the counter, move some receipts to the archive

What is Memory in Easy Terms?

Think of **AI memory** like a simple note-taking system:

- **Short-term memory**: “Sticky notes” on your desk for recent info.
- **Long-term memory**: A “filing cabinet” for older info sorted by topic.
- **Retrieval**: Flipping through that cabinet to find what you need.

Just as you’d forget details if you never wrote them down, an AI forgets unless it systematically stores and retrieves information. To handle lots of older messages without losing track, an AI might use *embeddings* or *summaries*. Imagine you have a of vacation photos:

- **Embeddings** are in — so later, if you ph
- **Summaries** work ily the beach in Ma b.

Both methods help the AI skip flipping through every word, either by matching "fingerprints" (embeddings) or by checking short "cliff notes" (summaries) to insta recall the details. In this tutorial, we will focus on embeddings.

Want to see these methods in action? Frameworks like **LangChain** provide:

- Conversation Buffer Window
- Conversation Summary Memory
- Vector Store Retriever Memory

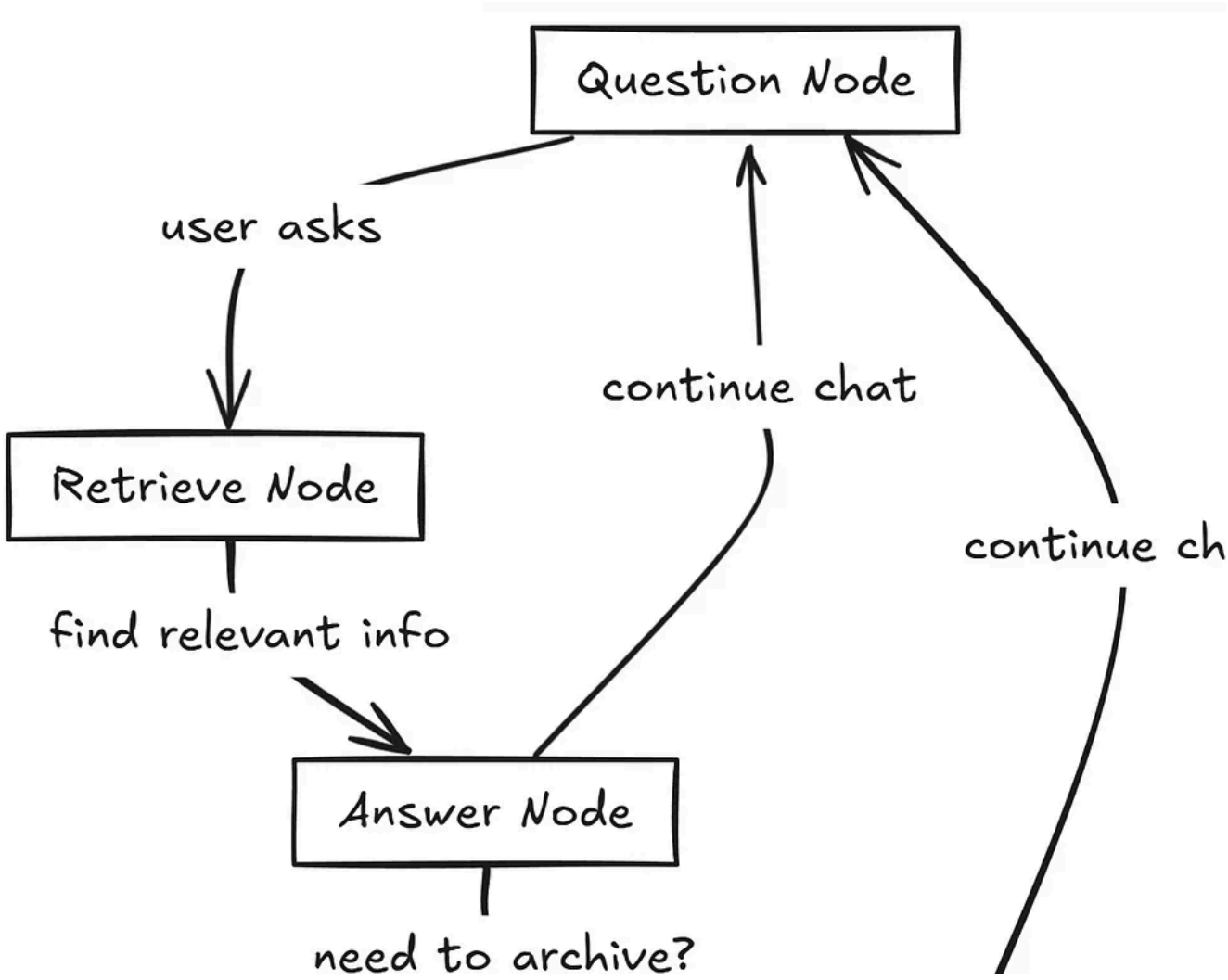
Feeling intimidated? No worries — we'll walk through simple examples with mini runnable code in Poc

How to Dl

Let's break down what we need to build our memory system:

- **Self-Loop Flow**: Think of this as the librarian's daily routine. They listen to questions, check references, answer patrons, and file away old materials — th start over with the next visitor.
- **Shared Store**: Like the library's central information system, with both the ope desk (short-term memory for current questions) and archive room (long-term memory for older topics).

Picture a library customer service desk. It's a continuous loop where information f between you, the librarian, and the archive shelves.



1. **Question Node**: The front desk librarian jots down your question.
2. **Retrieve Node**: The librarian checks the catalog and archived collections.
3. **Answer Node**: The librarian uses the found resources to help you.
4. **Embed Node**: The archivist who labels and files older notes.

No fancy math — just a neat loop that keeps all your conversations at your fingert

Before We Code: Let's Walk Through the Librarian Example

Let's see how our "librarian" handles a real conversation over time:

Day 1: You mention your pet

1. You tell the librarian: "I have a golden retriever named Max who loves to play fetch."
2. The librarian jots this down in a "current notes" binder at the front desk.
3. You chat about a few other things.
4. Eventually, once there are too many notes, the librarian moves some details to archive shelves in the back.
5. To make it easier to find later, the librarian files the note about "golden retrie Max, fetch" under a special topic label (an "embedding").
6. Now it's safely stored in the archive for long-term reference.

A week later: You ask about your dog

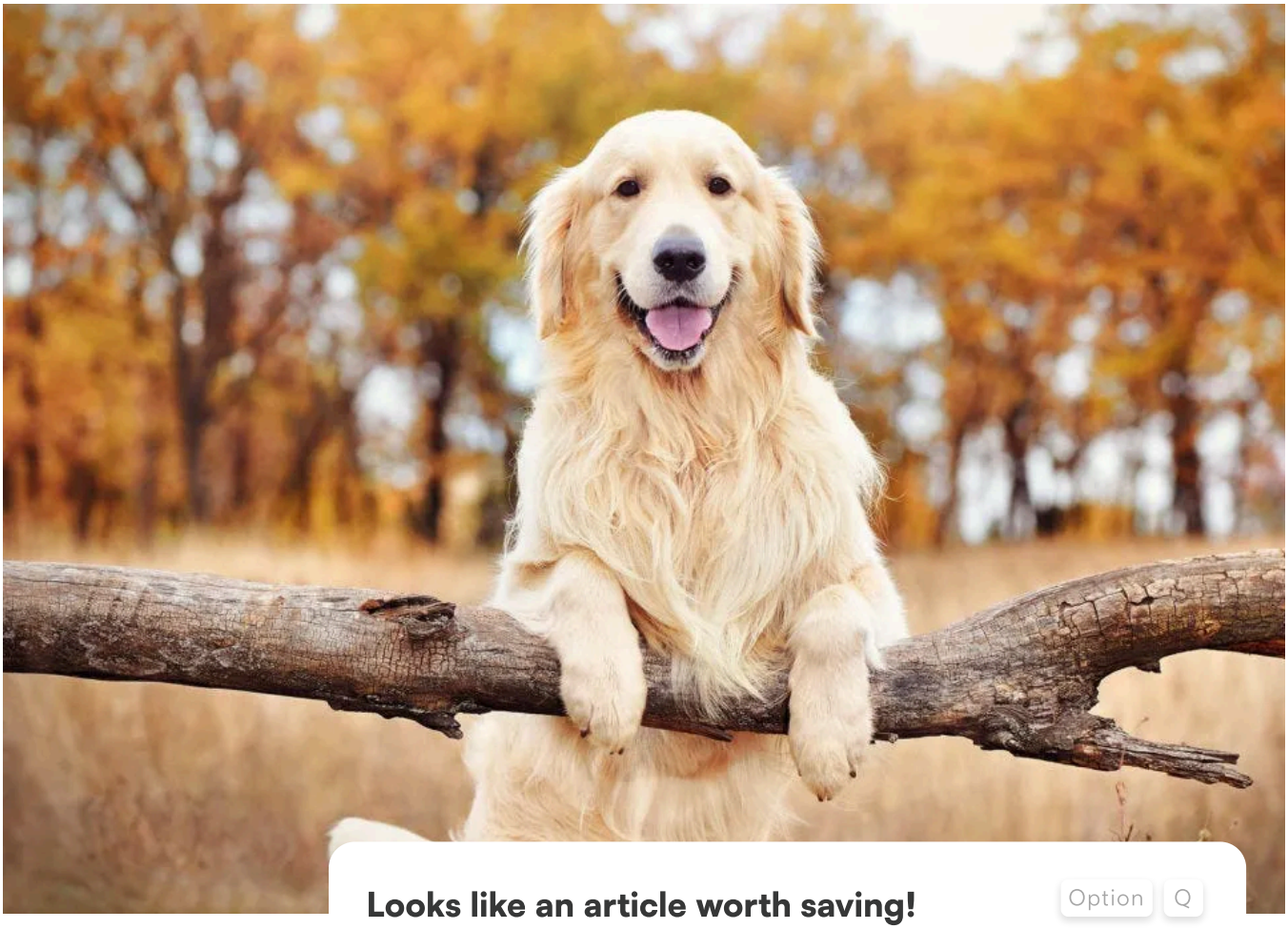
1. You return and a
2. The librarian wr
3. They check the a eve or "Max."
4. They find the original note about your golden retriever, Max.

5. They bring that note to the front desk.
6. The librarian says: “Your dog’s name is Max. He’s a golden retriever who loves play fetch.”

It feels like real remembering because:

- The librarian initially **organized** your info before putting it away.
- They used a topic-based **label** (embedding) for archiving.
- When you asked again, they looked up that label, retrieved the note, and **combined** it with your current question to give a complete answer.

Now let’s build such a librarian-inspired memory system from scratch!



Following along with

memory components at [GitHub: PocketFlow Chat Memory] to explore the details.

Step 1: Set Up the Shared Store

The **Shared Store** is our single "source of truth" for everything your AI needs to remember. It's where we keep both **short-term** details (like the most recent conversation) and **long-term** archives (where older chats get embedded and indexe

```
# Shared store: a simple dictionary to hold everything
shared = {
  # Short-term memory: your "sticky notes" for the current conversatic
  "messages": [],
  # Long-term memory: the "filing cabinet" that stores archived chats
  "vector_index": None,   # a placeholder for an index structure
  "vector_items": []      # a list of archived conversations
}
```

1. **messages**: Acts like short-term memory, holding recent user and assistant messages.
2. **vector_index**: A data structure (like a search index) for retrieving conversatior "topic fingerprint."
3. **vector_items**: A list of older, archived chats plus their embeddings, so they ca pulled back into the conversation.

All our Nodes (Question, Retrieve, Answer, Embed) will read and write from this dictionary, keeping everything in sync. That's the beauty of a single "notebook" fo your AI's memory!

Step 2: De

a. Question Node — Receives user inp and adds it to short-term memory

- **For the `prep`:** Like a librarian opening a fresh notebook page. We create a pla store our conversation if none exists yet.

```
def prep(self, shared):
    if "messages" not in shared:
        shared["messages"] = []
        print("Welcome to the interactive chat!")
    return None
```

- **For the `exec`:** Our librarian asks "How can I help you?" and waits for your question. If you say "exit," we close up shop.

```
def exec(self, _):
    user_input = input("\nYou: ")
    # If user types "exit", we'll stop the conversation
    if user_input.strip().lower() == "exit":
        return None
    return user_input
```

- **For the `post`:** The librarian writes your question in the log book. If you said "exit," we say goodbye. Otherwise, we save your message and move on to chec our records.

```
def post(self, shared, prep_res, exec_res):
    # If exec
    if exec_r
        print
        retur

    # Otherwi
    shared["messages"].append({"role": "user", "content": exec
```

```
        # Then decide the next node to call. Usually a RetrieveNode.
        return "retrieve"
```

b. Retrieve Node — Searches the archives for relevant info

- **For the `prep`:** Our librarian reads your latest question and checks if we have a archives to search. If either is missing, we skip this step.

```
    def prep(self, shared):
        if not shared.get("messages"):
            return None

       # Find the most recent user message
        latest_user_msg = next(
            (msg for msg in reversed(shared["messages"]) if msg["role"
    == "user"),
            None
        )

        # Check if we have a vector index (where archived items are
    stored)
        if "vector_index" not in shared or not shared["vector_index"]:
            return None

        # Return everything we need for the retrieval step
        return {
            "query": latest_user_msg["content"] if latest_user_msg els
    "",
            "vector_index": shared["vector_index"],
            "vectr
        }
```


- **For the `exec`:** O arc the archives for t nt information about "golden retrievers."

```
    def exec(self, inputs):
        if not inputs:
            return None
        query = inputs["query"]
        vector_index = inputs["vector_index"]
        vector_items = inputs["vector_items"]

        # (Pseudo) Create an embedding for the query
        # real code might call an external embedding function
        query_embedding = get_embedding(query)

        # Search the archived items for a match
        best_match, best_distance = search_vectors(vector_index,
    query_embedding)

        # If nothing is found, return None
        if best_match is None:
            return None

        # Return the best matching conversation and distance
        return {
            "conversation": vector_items[best_match],
            "distance": best_distance
        }
```

- **For the `post`**: Our librarian keeps any relevant file they found on the desk. Ei way, we proceed to formulating an answer.

```
    def post(self, shared, prep_res, exec_res):
        if exec_res is None:
            # No relevant info found; we just move on
            shared["retrieved_conversation"] = None
        else:
            # Save the retrieved conversation so the Answer Node can u
    it
            share                                                    n"
        # Continu
        return "a
```

c. Answer Node — Combines new and old info to generate a response

- **For the `prep`:** Our librarian gathers all needed resources: recent conversation notes and any relevant archived files, arranging them for easy reference.

```
def prep(self, shared):
    # Start with the most recent messages (e.g., last few user &
assistant turns)
    recent_messages = shared.get("messages", [])

    # Optionally add retrieved data
    retrieved_conv = shared.get("retrieved_conversation")

    # Combine them into a single context packet
    context = []
    if retrieved_conv:
        # Mark this as a 'system' note indicating it's relevant
background
        context.append({"role": "system", "content": "Relevant
archived info:"})
        context.extend(retrieved_conv)
        context.append({"role": "system", "content": "End of
archived info."})
    context.extend(recent_messages)
    return context
```

- **For the `exec`:** With all information at hand, our librarian crafts a thoughtful response that incorporates both recent and past knowledge.

```
def exec(self
    if not co
        retur

    # Example
    # In prac                                                  el
call
```

```
        response = call_llm(context)
        return response
```

- **For the `post`:** Our librarian writes down their answer, then decides if it's tim archive older notes (if we have more than 6 messages) or just continue helping

```
    def post(self, shared, prep_res, exec_res):
        # If there's no response, end the conversation
        if exec_res is None:
            return None

        # Add the assistant's answer to our short-term memory
        shared["messages"].append({"role": "assistant", "content":
    exec_res})

        # For example, if our messages are piling up, we might archive
    older ones
        if len(shared["messages"]) > 6:
            # We'll show how the 'archive' step works in the next node
            return "embed"

        # Otherwise, loop back to the question node for the next user
    query
        return "question"
```

d. Embed (Archive) Node — Moves older conversations into long-term memory

- **For the `prep`:** If our desk is getting cluttered (more than 6 messages), our libr takes the oldest


```
    def prep(self
        # If we h
        if len(sh
            return None
```

```
        # Extract the oldest user + assistant pair (2 messages)
        oldest_pair = shared["messages"][:2]

        # Remove them from short-term memory
        shared["messages"] = shared["messages"][2:]
        return oldest_pair
```

- **For the `exec`:** Our librarian creates a label for the file by combining the conversation into one document and giving it a special "topic fingerprint" for retrieval later.

```
def exec(self, conversation):
    if not conversation:
        return None

    # Combine the user & assistant messages into one text for
embedding
    user_msg = next((m for m in conversation if m["role"] ==
"user"), {"content": ""})
    assistant_msg = next((m for m in conversation if m["role"] ==
"assistant"), {"content": ""})
    combined_text = f"User: {user_msg['content']}\nAssistant:
{assistant_msg['content']}"

    # Create an embedding (pseudo-code)
    embedding = get_embedding(combined_text)
    return {
        "conversation": conversation,
        "embedding": embedding
    }
```

- **For the `post`:** Our librarian files the labeled conversation in the archives, crea a filing system if question.


```
def post(self
    if not exec_res:
```

```
            # Nothing to archive, just go back to questions
            return "question"

        # Initialize our archive if it's missing
        if "vector_index" not in shared or shared["vector_index"] is
    None:
            shared["vector_index"] = create_index()
            shared["vector_items"] = []

        # Add the embedding to the index
        position = add_vector(shared["vector_index"],
    exec_res["embedding"])

        # Keep track of the archived conversation itself
        shared["vector_items"].append(exec_res["conversation"])
        print(f"✅ Archived a conversation at index position
    {position}.")
        print(f"✅ Total archived: {len(shared['vector_items'])}")

        # Return to the question node to continue the chat
        return "question"
```

Step 3. Connect the Nodes in a Self-Loop Flow

Now let's make the system run by connecting each node into a self-loop flow and choosing a starting point:

```
# Create your node instances
question_node = QuestionNode()
retrieve_node = RetrieveNode()
answer_node = Ans
embed_node = Embe

# Connect the flo
# Main flow path
question_node - '
retrieve_node - '
```

```
# When we need to embed old conversations
answer_node - "embed" >> embed_node

# Loop back for next question
answer_node - "question" >> question_node
embed_node - "question" >> question_node

# Create the flow starting with question node
chat_flow = Flow(start=question_node)

# Set up an empty "notebook"
shared = {}

# Start the conversation loop
chat_flow.run(shared)
```

As soon as we call `chat_flow.run(shared)`, the system:

- Enters the **Question Node** (prep → exec → post).
- Jumps to the node name returned by the **post** step ("retrieve").
- Continues through **Retrieve**, **Answer**, **Embed** as each node's post method dict
- Keeps looping until a node returns `None` (meaning it's time to stop)

All the *Flow* class does is respect these returns — once a node finishes its `post`, *Fl* calls the next node by name. If a node ever returns `None`, the entire conversation e That's our *self-loop* in action:

Question → Retrieve → Answer → Embed → Question → … → None (exit)

That's it! You now ha *ng u*
input, retrieving old *ou*
this code, your AI *ook*

Step 4. Run and Test

Ready to run this code? Here's how:

1. Clone the *GitHub: PocketFlow Chat Memory*
2. Set up your API key:

```
export OPENAI_API_KEY="your-api-key-here"
```

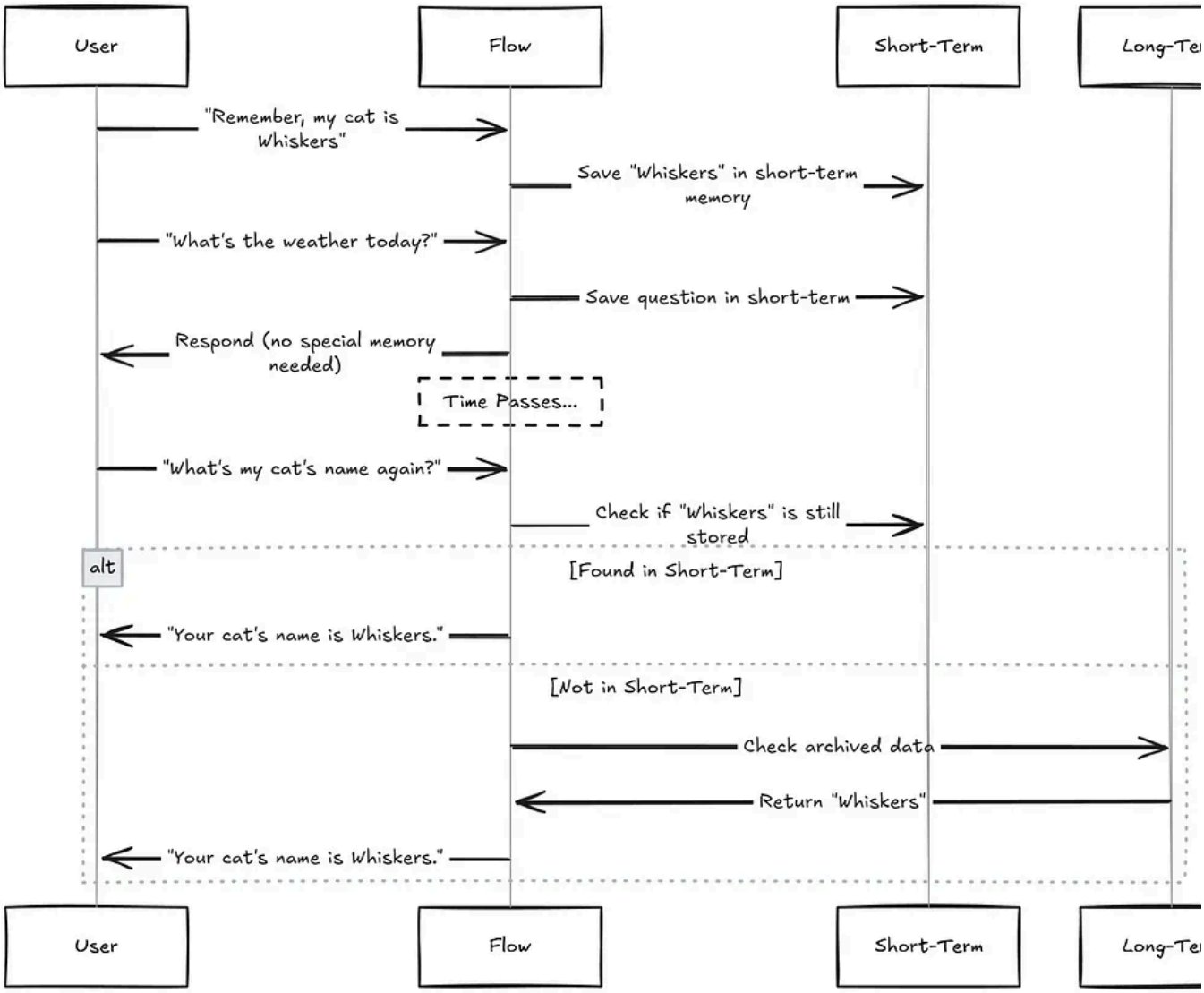
3. Run the application:

```
pip install -r requirements.txt
python main.py
```

Here's a simple example showing how your AI's memory works in practice:

> ***User:*** *"Remember, my cat is Whiskers"*
>
> ***Assistant:*** *"Got it! I'll keep that in mind."*
>
> *(The system saves "Whiskers" to short-term memory.)*
>
> ***User:*** *"What's the weather today?"*
>
> ***Assistant:*** *"I don't have real-time info right now, but you can check a weather app!"*
>
> *(This question doesn't impact memory storage.)*
>
> ...
>
> ***User:*** *"What's my ca*
>
> ***Assistant:*** *"Your cat*
>
> *(The system pulls "Whiskers" from memory.)*

Behind the Scenes (Simplified):



- **Short-Term** stores recent info (like the cat's name).
- **Long-Term** archives older details once short-term gets full.
- **Flow** decides whether to use short-term or check long-term memory when you ask something.
- Even if you ask unrelated questions in between, the system can still recall "Whiskers" when

AI Memor Secret

You've now learned the straightforward secret behind AI memory. No fancy magic just smart organization and looping processes. Here's what makes it tick:

1. **Self-Loop Flow**: A simple cycle that continuously handles your input, finds re past info, generates helpful answers, and archives older messages.
2. **Shared Store**: One organized place to hold both short-term ("sticky notes") an long-term ("filing cabinet") memory.
3. **Memory Retrieval**: Quickly finding relevant conversations from the archives v needed.
4. **Contextual Responses**: Combining new questions and retrieved info to form meaningful answers.

Next time you interact with an AI assistant that seems to "remember" things, you' instantly recognize these simple ideas at work.

Ready to build your own AI system with memory? The complete working code for this tut is available at [GitHub: Pocket Flow Chat Memory](). Clone the repository, run the code, c start experimenting with your own memory-enabled chatbot today!

Comments Restacks

Write a comment...

1 more comment...