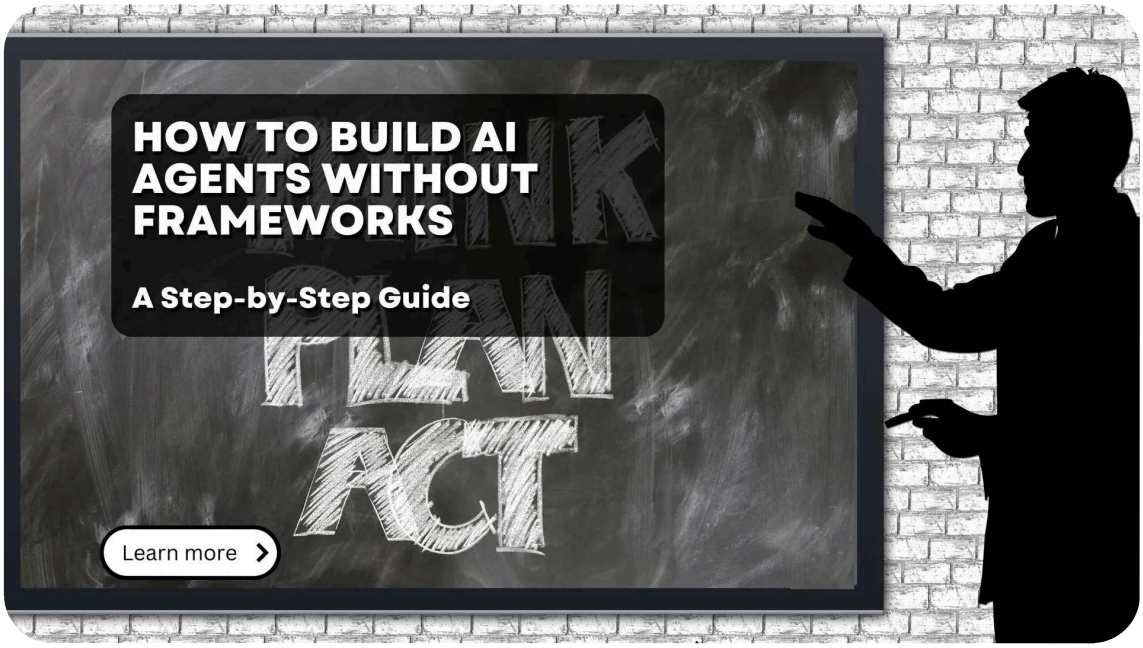
How to Build AI Agents Without Frameworks: A Step-by-Step Guide



AI agents are often overcomplicated. At their core, they're just language models with the ability to use tools and remember context. While frameworks like LangChain or AutoGPT can help you get started quickly, they add layers of abstraction that can make it harder to understand what's actually happening – and harder to customize your agent for specific use cases.

In this tutorial, we'll build an AI agent from scratch using direct API calls to GPT-4o. You'll learn how to create an agent that can query PostgreSQL databases, fetch information from Wikipedia, and maintain conversation context through simple database storage. No frameworks, just clean code and clear concepts.

The goal is to demonstrate that AI agents don't need complex architectures. We'll focus on the essential components: prompt engineering for tool use, storing conversation history in PostgreSQL, and managing the interaction flow between the LLM and its tools.

At the end, you will know exactly, what an agent is, how it works and how to build them. If you then decide to still use a framework - that's fine. However, you will have a much better understanding of what's happening under the hood.

Let's start by understanding what makes something an AI agent, and then build one step by step.

What Are AI Agents?

An AI agent is fundamentally just a language model that can:

- Understand what tools it has available
- Decide when to use these tools
- Remember previous interactions
- Make decisions based on all this information

What Are Tools?

Tools (sometimes called "functions" or "capabilities") are simply functions that the agent can call to interact with the outside world. They're just Python functions with:

- A clear description of what they do
- Defined input parameters
- Structured output

Here's a concrete example of a tool:

```
1 def search_wikipedia(query: str) -> str:
2     """
```

```
    Searches Wikipedia and returns the first paragraph of the most relevant art

    Args:
        query: The search term to look up on Wikipedia

    Returns:
        str: The first paragraph of the most relevant Wikipedia article
    """
    return wikipedia.summary(query, sentences=3)

# This is how we describe the tool to the LLM
wikipedia_tool_description = """
Tool: search_wikipedia
Description: Searches Wikipedia and returns a summary of the topic
Input: A search query string
Example: To learn about Python programming, use: search_wikipedia("Python progr
"""
```

When we give our agent access to this tool, we're essentially telling it: "You can search Wikipedia by calling this function." The LLM then decides, based on the user's question, whether it needs to use this tool to provide a good answer. When we say "the LLM decides," we mean that we prompt the LLM with the available tools and ask it to decide which one to use. The AI model then simply returns a text response with the tool name.

Common examples of tools include:

- Database queries
- Web searches
- API calls
- File operations
- Calculator functions
- Email sending capabilities

To bring both the concepts of agents and tools together, we see that there is no magic involved. It's just AI model interactions and a bunch of python functions.

Bringing this together, a pseudo-code for an AI agent might look like this:

```python
# This is what an agent interaction basically looks like
def agent_interaction(user_input, available_tools, conversation_history):
    # 1. We tell the LLM what it can do
    prompt = f"""You have access to these tools:
    - query_database: Executes SQL queries
    - search_wikipedia: Searches Wikipedia articles

    Previous conversation:
    {conversation_history}

    User question: {user_input}

    Decide if you need to use a tool to answer. If yes, specify which tool and

    # 2. The LLM decides what to do
    llm_response = gpt4_call(prompt)

    # 3. We execute the tool if needed and get the final answer
    if "use query_database" in llm_response:
        # Execute database query and get result
        data = execute_query(extract_query(llm_response))
        return get_final_answer(data)

    return llm_response
```

This is the core concept of an agent. Everything else – complex planning, multiple tool calls, sophisticated memory systems – is just an extension of this basic pattern.

Common misconceptions about agents:

- They don't need complex orchestration frameworks
- They don't need to run autonomously (though they can)
- They don't need sophisticated planning algorithms (though these can help for complex tasks)

The key to building effective agents is not in the framework you use, but in:

- Writing clear prompts that help the LLM understand its capabilities
- Implementing reliable tools that the agent can use

- Maintaining relevant context through some form of memory storage

In the following sections, we'll implement this pattern with real code, building an agent that can actually query databases and search Wikipedia. You'll see that the complexity comes from the specific requirements of your use case, not from the agent architecture itself.

Why Build Agents Without Frameworks?

Frameworks like LangChain and AutoGPT are often recommended for building AI agents. While they can be useful for quick prototypes, there are compelling reasons to avoid them:

Understanding What Really Happens

```
# With a framework:
agent = Agent.from_toolkit(SQLToolkit, WikiToolkit)
result = agent.run("Find users who like pizza")

# Without a framework - what's actually happening:
def process_query(user_input: str):
    prompt = f"""You have these tools:
    1. SQL queries: Execute database queries
    2. Wikipedia search: Look up information

    User question: {user_input}

    Think step by step:
    1. Do I need any tools to answer this?
    2. If yes, which tool and how should I use it?
    """

    response = gpt4_call(prompt)
    if "SQL" in response:
        query = extract_sql(response)
        data = execute_query(query)
    if "Wikipedia" in response:
        data = search_wikipedia(topic)
        return get_final_answer(data)
```

The framework-free version is more explicit. You can see exactly:

- What the LLM is being told
- When tools are being called
- How decisions are being made

Flexibility and Control

Frameworks often make assumptions about:

- How tools should be structured
- What the prompt format should be
- How memory should work

When you need to customize any of these, you often end up fighting against the framework rather than working with it.

Simpler Debugging

When something goes wrong with a framework:

```
# Framework error:
AgentError: Tool execution failed: SQLToolkit.query failed with error:
Invalid chain configuration...

# Without framework:
try:
    data = execute_query(query)
except SQLException as e:
    print(f"SQL error: {e}")  # Clear what went wrong
```

> **Note:** This is more or less the most prominent reason why you should build agents without frameworks. Frameworks abstract away the core of any AI Agent - the LLM interaction. Without adding additional infrastructure, you don't know the exact prompts used, the specific LLM answers replied and the tools used. This makes debugging a nightmare.

Lower Learning Overhead

Building agents requires understanding:

- LLM prompting
- Tool integration
- Memory management

Frameworks add another layer:

- Framework-specific concepts
- Configuration options
- Integration patterns

Why learn both when you only need the fundamentals?

Performance and Resources

Frameworks often:

- Load unnecessary components
- Add network calls
- Include unused features

Direct implementation lets you:

- Load only what you need
- Optimize critical paths
- Control resource usage

> **Note:** Arguably, raw performance is not the most important thing for AI agents, as the AI model interaction in itself is quite slow. So loading some additional modules does not add up that much.

When to Use Frameworks

Frameworks make sense when:

- You're building a quick prototype
- You need pre-built tools
- You're learning agent concepts initially

But for production systems or when you need full control, direct implementation is often better.

Let's move on to building our agent from scratch, where you'll see how straightforward the direct implementation can be.

Building an AI Agent Step by Step

Before we dive into the code, let's understand what we'll be building in this first step. We'll start by defining our tools - the capabilities we want to give our agent.

Step 1: Defining Tools

In this step, we'll create two tools:

1. A PostgreSQL query tool that allows the agent to query your database and search for information stored in the database.
2. A Wikipedia search tool that lets the agent look up information in Wikipedia.

We can see that both tools are quite different, so depending on the users question, the agent should be able to decide to use one of these tools.

When defining tools for OpenAI's function calling, we need three components for each tool:

- A schema that tells the LLM what the tool does and how to use it. This is basically just a json description of the python function. It will get clearer once you see the example.
- The actual implementation of the tool (the Python function)
- Error handling to ensure our agent behaves gracefully when things go wrong

The schema is particularly important because it acts as documentation for the LLM. Think of it as writing API documentation - the better you describe your tools, the better the LLM will use them. Important are the `name` (which should match the function name), `description` (what the tool does) and `parameters` (what function parameter values we need).

For our database tool, we'll:

- Allow only SELECT queries for safety
- Return results as JSON for easy parsing
- Include error handling for database connection issues

For the Wikipedia tool, we'll:

- Limit results to three sentences to keep responses concise
- Handle disambiguation pages and missing articles
- Return structured error messages when searches fail

Here's how we implement these tools:

```python
import json
import psycopg2
import wikipedia
from typing import Dict, List, Any
from openai import OpenAI

# First, we define how the LLM should understand our tools
tools = [
    {
        "type": "function",
        "function": {
            "name": "query_database",
            "description": """Execute a PostgreSQL SELECT query and return the
                            Only SELECT queries are allowed for security reasc
                            Returns data in JSON format.""",
            "parameters": {
                "type": "object",
                "properties": {
                    "query": {
                        "type": "string",
                        "description": """The SQL SELECT query to execute.
                                        Must start with SELECT and should not
```

```
                                        contain any data modification commands
                    }
                },
                "required": ["query"],
                "additionalProperties": False
            },
            "strict": True
        }
    },
    {
        "type": "function",
        "function": {
            "name": "search_wikipedia",
            "description": """Search Wikipedia and return a concise summary.
                            Returns the first three sentences of the most
                            relevant article.""",
            "parameters": {
                "type": "object",
                "properties": {
                    "query": {
                        "type": "string",
                        "description": "The topic to search for on Wikipedia"
                    }
                },
                "required": ["query"],
                "additionalProperties": False
            },
            "strict": True
        }
    }
]

# Now implement the actual tool functions
def query_database(query: str) -> str:
    """
    Execute a PostgreSQL query and return results as a JSON string.
    Includes safety checks and error handling.
    """
    # Security check: Only allow SELECT queries
    if not query.lower().strip().startswith('select'):
        return json.dumps({
            "error": "Only SELECT queries are allowed for security reasons."
        })

    try:
        # You should use environment variables for these in production
        conn = psycopg2.connect(
            "dbname=your_db user=your_user password=your_pass host=localhost"
        )
```

```
72
73          with conn.cursor() as cur:
74              cur.execute(query)
75
76              # Get column names from cursor description
77              columns = [desc[0] for desc in cur.description]
78
79              # Fetch results and convert to list of dictionaries
80              results = []
81              for row in cur.fetchall():
82                  results.append(dict(zip(columns, row)))
83
84              return json.dumps({
85                  "success": True,
86                  "data": str(results),
87                  "row_count": len(results)
88              })
89
90      except psycopg2.Error as e:
91          return json.dumps({
92              "error": f"Database error: {str(e)}"
93          })
94      except Exception as e:
95          return json.dumps({
96              "error": f"Unexpected error: {str(e)}"
97          })
98      finally:
99          if 'conn' in locals():
100             conn.close()
101
102 def search_wikipedia(query: str) -> str:
103     """
104     Search Wikipedia and return a concise summary.
105     Handles disambiguation and missing pages gracefully.
106     """
107     try:
108         # Try to get the most relevant page summary
109         summary = wikipedia.summary(
110             query,
111             sentences=3,
112             auto_suggest=True,
113             redirect=True
114         )
115
116         return json.dumps({
117             "success": True,
118             "summary": summary,
119             "url": wikipedia.page(query).url
120         })
```

```
121
122     except wikipedia.DisambiguationError as e:
123         # Handle multiple matching pages
124         return json.dumps({
125             "error": "Disambiguation error",
126             "options": e.options[:5],  # List first 5 options
127             "message": "Topic is ambiguous. Please be more specific."
128         })
129     except wikipedia.PageError:
130         return json.dumps({
131             "error": "Page not found",
132             "message": f"No Wikipedia article found for: {query}"
133         })
134     except Exception as e:
135         return json.dumps({
136             "error": "Unexpected error",
137             "message": str(e)
138         })
```

Step 2: Creating the Agent Class - Detailed Design

Before implementing the code, let's understand exactly what we're building and how it works:

Core Concept

An AI agent is essentially a loop of:

1. Receiving user input
2. Deciding whether to use tools
3. Using tools if needed
4. Formulating a response

Key Components Design

1. Message Management

- We need to maintain a conversation history as a list of messages
- Each message has a specific role: 'user', 'assistant', or 'tool'
- The history provides context for the LLM to make better decisions

- Think of it like a chat application's state management

2. Tool Execution System

- Works like a command dispatcher:
    1. LLM decides to use a tool
    2. We receive a structured function call
    3. We map this to our actual Python functions
    4. We handle the execution and any errors
    5. We format the result for the LLM

3. Main Processing Loop

```
User Input → LLM Decision → [Tool Use?] → Final Response
                                ↓
                          Execute Tool(s)
                                ↓
                       Feed Results back to LLM
```

Error Handling Strategy

We need three layers of error handling:

1. Tool-level errors (e.g., database connection failed)
2. LLM-level errors (e.g., API issues)
3. General processing errors (e.g., JSON parsing)

Message Flow Example

```
User: "How many users are in our database?"
│
├─► LLM Thinks: "I need to query the database"
│    │
│    ├─► Calls query_database tool
│    │    │
│    │    └─► Returns user count
│    │
│    └─► Formulates final response with data
│
```

```
└─► Final Response: "There are 1,234 users in the database"
```

This design allows for:

- Clear separation of concerns
- Easy debugging and monitoring
- Flexible extension with new tools
- Robust error handling
- Maintainable codebase

Note that we implement the general agent structure as part of a loop. Meaning that we send back the results of our tool calls to the LLM which then decides whether to formulate a final answer or call another tool.

Implementation

```python
from typing import List, Dict, Optional, Any
from openai import OpenAI
import json

class Agent:
    def __init__(self, system_prompt: Optional[str] = None):
        """
        Initialize an AI Agent with optional system prompt.

        Args:
            system_prompt: Initial instructions for the AI
        """
        # Initialize OpenAI client - expects OPENAI_API_KEY in environment
        self.client = OpenAI()

        # Initialize conversation history
        self.messages = []

        # Set up system prompt if provided, otherwise use default
        default_prompt = """You are a helpful AI assistant with access to a da
        and Wikipedia. Follow these rules:
        1. When asked about data, always check the database first
        2. For general knowledge questions, use Wikipedia
        3. If you're unsure about data, query the database to verify
        4. Always mention your source of information
        5. If a tool returns an error, explain the error to the user clearly
        """
```

```
28
29          self.messages.append({
30              "role": "system",
31              "content": system_prompt or default_prompt
32          })
33
34      def execute_tool(self, tool_call: Any) -> str:
35          """
36          Execute a tool based on the LLM's decision.
37
38          Args:
39              tool_call: The function call object from OpenAI's API
40
41          Returns:
42              str: JSON-formatted result of the tool execution
43          """
44          try:
45              function_name = tool_call.function.name
46              function_args = json.loads(tool_call.function.arguments)
47
48              # Execute the appropriate tool. Add more here as needed.
49              if function_name == "query_database":
50                  result = query_database(function_args["query"])
51              elif function_name == "search_wikipedia":
52                  result = search_wikipedia(function_args["query"])
53              else:
54                  result = json.dumps({
55                      "error": f"Unknown tool: {function_name}"
56                  })
57
58              return result
59
60          except json.JSONDecodeError:
61              return json.dumps({
62                  "error": "Failed to parse tool arguments"
63              })
64          except Exception as e:
65              return json.dumps({
66                  "error": f"Tool execution failed: {str(e)}"
67              })
68
69      def process_query(self, user_input: str) -> str:
70          """
71          Process a user query through the AI agent.
72
73          Args:
74              user_input: The user's question or command
75
76          Returns:
```

```
            str: The agent's response
        """
        # Add user input to conversation history
        self.messages.append({
            "role": "user",
            "content": user_input
        })

        try:
            max_iterations = 5
            current_iteration = 0

            while current_iteration < max_iterations:  # Limit to 5 iterations
                current_iteration += 1
                completion = self.client.chat.completions.create(
                    model="gpt-4o",
                    messages=self.messages,
                    tools=tools,  # Global tools list from Step 1
                    tool_choice="auto"  # Let the model decide when to use too
                )

                response_message = completion.choices[0].message

                # If no tool calls, we're done
                if not response_message.tool_calls:
                    self.messages.append(response_message)
                    return response_message.content

                # Add the model's thinking to conversation history
                self.messages.append(response_message)

                # Process all tool calls
                for tool_call in response_message.tool_calls:
                    try:
                        print("Tool call:", tool_call)
                        result = self.execute_tool(tool_call)
                        print("Tool executed......")
                    except Exception as e:
                        print("Execution failed......")
                        result = json.dumps({
                            "error": f"Tool execution failed: {str(e)}"
                        })

                    print(f"Tool result custom: {result}")

                    self.messages.append({
                        "role": "tool",
                        "tool_call_id": tool_call.id,
                        "content": str(result)
```

```
                    })
                    print("Messages:", self.messages)

            # If we've reached max iterations, return a message indicating thi
            max_iterations_message = {
                "role": "assistant",
                "content": "I've reached the maximum number of tool calls (5)
            }
            self.messages.append(max_iterations_message)
            return max_iterations_message["content"]

        except Exception as e:
            error_message = f"Error processing query: {str(e)}"
            self.messages.append({
                "role": "assistant",
                "content": error_message
            })
            return error_message

    def get_conversation_history(self) -> List[Dict[str, str]]:
        """
        Get the current conversation history.

        Returns:
            List[Dict[str, str]]: The conversation history
        """
        return self.messages
```

(Optional) Step 3: Adding Memory to Our Agent

> **Note:** Please consider whether you need memory at all. Does your application really need to store and retrieve conversation histories? More often than not, the answer is no! But if you do need memory, here's how you can add it.

Memory in AI agents is often overcomplicated. Let's understand what we're actually trying to achieve:

What is Agent Memory?

At its core, agent memory is just:

1. Storing conversations and their context
2. Retrieving relevant information when needed
3. Summarizing past interactions when they become too long

Memory Types We'll Implement

1. **Conversation History**
    - Store: User inputs, agent responses, tool usage results
    - Purpose: Keep track of the current conversation flow
2. **Summary Memory**
    - What: Periodic summaries of longer conversations
    - Why: Prevent context window overflow
    - How: Create summaries after X messages or on demand

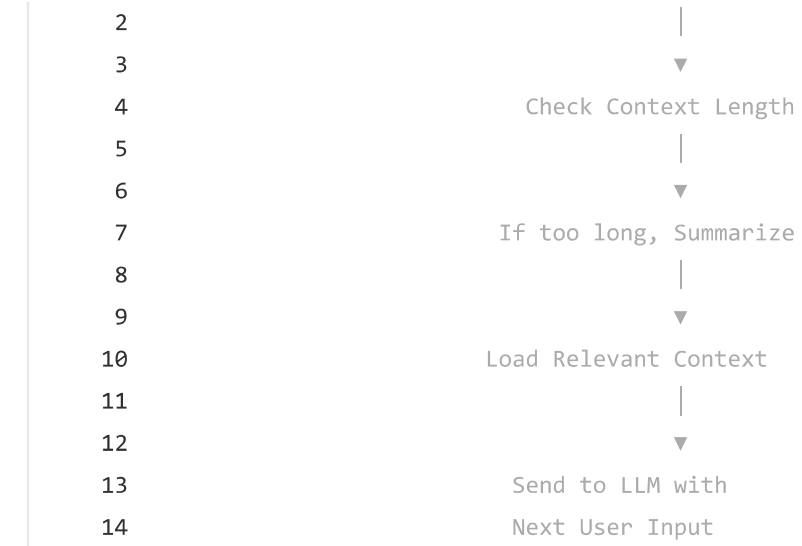
Database Schema

```sql
CREATE TABLE conversations (
    id SERIAL PRIMARY KEY,
    session_id UUID,
    user_input TEXT,
    agent_response TEXT,
    tool_calls JSONB,  -- Store tool usage
    timestamp TIMESTAMPTZ DEFAULT NOW()
);

CREATE TABLE conversation_summaries (
    id SERIAL PRIMARY KEY,
    session_id UUID,
    summary TEXT,
    start_time TIMESTAMPTZ,
    end_time TIMESTAMPTZ,
    message_count INTEGER
);
```

Memory Flow

```
User Input → Store in DB ──────────┐
```



Implementation of Memory for an AI Agent

```python
from datetime import datetime
import uuid
from typing import List, Dict, Optional
from dataclasses import dataclass
import psycopg2
from psycopg2.extras import Json, UUID_adapter

@dataclass
class MemoryConfig:
    """Configuration for memory management"""
    max_messages: int = 20  # When to summarize
    summary_length: int = 2000  # Max summary length in words
    db_connection: str = "dbname=your_db user=your_user password=your_pass"

class AgentMemory:
    def __init__(self, config: Optional[MemoryConfig] = None):
        self.config = config or MemoryConfig()
        self.session_id = uuid.uuid4()
        self.setup_database()

    def setup_database(self):
        """Create necessary database tables if they don't exist"""
        queries = [
            """
            CREATE TABLE IF NOT EXISTS conversations (
                id SERIAL PRIMARY KEY,
                session_id UUID NOT NULL,
                user_input TEXT NOT NULL,
                agent_response TEXT NOT NULL,
                tool_calls JSONB,
                timestamp TIMESTAMPTZ DEFAULT NOW()
```

```
            );
            """,
            """
            CREATE TABLE IF NOT EXISTS conversation_summaries (
                id SERIAL PRIMARY KEY,
                session_id UUID NOT NULL,
                summary TEXT NOT NULL,
                start_time TIMESTAMPTZ NOT NULL,
                end_time TIMESTAMPTZ NOT NULL,
                message_count INTEGER NOT NULL
            );
            """
        ]

        with psycopg2.connect(self.config.db_connection) as conn:
            with conn.cursor() as cur:
                for query in queries:
                    cur.execute(query)

    def store_interaction(self,
                          user_input: str,
                          agent_response: str,
                          tool_calls: Optional[List[Dict]] = None):
        """Store a single interaction in the database"""
        query = """
        INSERT INTO conversations
            (session_id, user_input, agent_response, tool_calls)
        VALUES (%s, %s, %s, %s)
        """

        with psycopg2.connect(self.config.db_connection) as conn:
            with conn.cursor() as cur:
                cur.execute(query, (
                    self.session_id,
                    user_input,
                    agent_response,
                    Json(tool_calls) if tool_calls else None
                ))

    def create_summary(self, messages: List[Dict]) -> str:
        """Create a summary of messages using the LLM"""
        client = OpenAI()

        # Prepare messages for summarization
        summary_prompt = f"""
        Summarize the following conversation in less than {self.config.summary
        Focus on key points, decisions, and important information discovered t

        Conversation:
```

```
        {messages}
        """

        response = client.chat.completions.create(
            model="gpt-4o",
            messages=[{"role": "user", "content": summary_prompt}]
        )

        return response.choices[0].message.content

    def store_summary(self, summary: str, start_time: datetime, end_time: dat
        """Store a conversation summary"""
        query = """
        INSERT INTO conversation_summaries
            (session_id, summary, start_time, end_time, message_count)
        VALUES (%s, %s, %s, %s, %s)
        """

        with psycopg2.connect(self.config.db_connection) as conn:
            with conn.cursor() as cur:
                cur.execute(query, (
                    self.session_id,
                    summary,
                    start_time,
                    end_time,
                    message_count
                ))

    def get_recent_context(self) -> str:
        """Get recent conversations and summaries for context"""
        # First, get the most recent summary
        summary_query = """
        SELECT summary, end_time
        FROM conversation_summaries
        WHERE session_id = %s
        ORDER BY end_time DESC
        LIMIT 1
        """

        # Then get conversations after the summary
        conversations_query = """
        SELECT user_input, agent_response, tool_calls, timestamp
        FROM conversations
        WHERE session_id = %s
        AND timestamp > %s
        ORDER BY timestamp ASC
        """

        with psycopg2.connect(self.config.db_connection) as conn:
```

```
            with conn.cursor() as cur:
                # Get latest summary
                cur.execute(summary_query, (self.session_id,))
                summary_row = cur.fetchone()

                if summary_row:
                    summary, last_summary_time = summary_row
                    # Get conversations after the summary
                    cur.execute(conversations_query, (self.session_id, last_su
                else:
                    # If no summary exists, get recent conversations
                    cur.execute("""
                        SELECT user_input, agent_response, tool_calls, timesta
                        FROM conversations
                        WHERE session_id = %s
                        ORDER BY timestamp DESC
                        LIMIT %s
                    """, (self.session_id, self.config.max_messages))

                conversations = cur.fetchall()

        # Format context
        context = []
        if summary_row:
            context.append(f"Previous conversation summary: {summary}")

        for conv in conversations:
            user_input, agent_response, tool_calls, _ = conv
            context.append(f"User: {user_input}")
            if tool_calls:
                context.append(f"Tool Usage: {tool_calls}")
            context.append(f"Assistant: {agent_response}")

        return "\n".join(context)

    def check_and_summarize(self):
        """Check if we need to summarize and do it if necessary"""
        query = """
        SELECT COUNT(*)
        FROM conversations
        WHERE session_id = %s
        AND timestamp > (
            SELECT COALESCE(MAX(end_time), '1970-01-01'::timestamptz)
            FROM conversation_summaries
            WHERE session_id = %s
        )
        """

        with psycopg2.connect(self.config.db_connection) as conn:
```

```
            with conn.cursor() as cur:
                cur.execute(query, (self.session_id, self.session_id))
                count = cur.fetchone()[0]

                if count >= self.config.max_messages:
                    # Get messages to summarize
                    cur.execute("""
                        SELECT user_input, agent_response, tool_calls, timesta
                        FROM conversations
                        WHERE session_id = %s
                        ORDER BY timestamp ASC
                        LIMIT %s
                    """, (self.session_id, count))

                    messages = cur.fetchall()
                    if messages:
                        # Create and store summary
                        summary = self.create_summary(messages)
                        self.store_summary(
                            summary,
                            messages[0][3],  # start_time
                            messages[-1][3],  # end_time
                            len(messages)
                        )

# Update Agent class to use memory.
class MemoryAgent(Agent):
    def __init__(self, memory_config: Optional[MemoryConfig] = None):
        self.client = OpenAI()
        self.memory = AgentMemory(memory_config)
        self.messages = []

        # Initialize with system prompt
        self.messages.append({
            "role": "system",
            "content": "You are a helpful AI assistant..."
        })

    def process_query(self, user_input: str) -> str:
        try:
            # Check if we need to summarize before adding new messages
            self.memory.check_and_summarize()

            # Get context (including summaries) from memory
            context = self.memory.get_recent_context()

            # Add context to messages if it exists
            if context:
                self.messages = [
```

```
                    self.messages[0],  # Keep system prompt
                    {
                        "role": "system",
                        "content": f"Previous conversation context:\n{context}
                    }
                ]

            # Process the query as before...
            response = super().process_query(user_input)

            # Store the interaction in memory
            tool_calls = None
            if hasattr(self, 'last_tool_calls'):
                tool_calls = self.last_tool_calls

            self.memory.store_interaction(
                user_input=user_input,
                agent_response=response,
                tool_calls=tool_calls
            )

            return response

        except Exception as e:
            error_message = f"Error processing query: {str(e)}"
            self.memory.store_interaction(
                user_input=user_input,
                agent_response=error_message
            )
            return error_message

    def execute_tool(self, tool_call: Any) -> str:
        # Store tool calls for memory
        if not hasattr(self, 'last_tool_calls'):
            self.last_tool_calls = []

        result = super().execute_tool(tool_call)

        # Store tool call information
        self.last_tool_calls.append({
            'tool': tool_call.function.name,
            'arguments': tool_call.function.arguments,
            'result': result
        })

        return result
```

Before we continue...

Interested in how to train your very own Large Language Model?

We prepared a well-researched guide for how to use the latest advancements in Open Source technology to fine-tune your own LLM. This has many advantages like:

- Cost control
- Data privacy
- Excellent performance - adjusted specifically for your intended use

Get your free LLM training guide

Step 4: Using Your AI Agent

After all the setup, using the agent is surprisingly straightforward. Here's how:

```
from your_agent import Agent, MemoryConfig  # assuming we saved our code in yo

# Create an agent
agent = Agent(
    memory_config=MemoryConfig(
        max_messages=10,  # summarize after 10 messages
        db_connection="postgresql://user:pass@localhost/your_db"
    )
)

# Simple question-answer interaction
def chat_with_agent(question: str):
    print(f"\nUser: {question}")
    print(f"Assistant: {agent.process_query(question)}")

if __name__ == "__main__":
```

```
    chat_with_agent("How many users do we have in our database?")
    chat_with_agent("What's the average age of users from New York?")
    chat_with_agent("Compare our user base in California with PostgreSQL's popu
    chat_with_agent("What were the numbers you just mentioned?")
    chat_with_agent("What are the most interesting trends you can find in our u
```

That's it! The agent will:

- Use the database when needed
- Search Wikipedia when appropriate
- Remember previous context
- Create summaries automatically
- Handle all the complexity behind the scenes - but at the same time giving us direct debugging access

The output looks clean and natural:

```
User: How many users do we have in our database?
Assistant: According to the database, we currently have 10,432 registered users

User: What's the average age of users from New York?
Assistant: I've queried the database and found that users from New York have an

User: Compare our user base in California with PostgreSQL's popularity there
Assistant: Let me check both sources. According to our database, we have 2,345 u
```

Enhanced Database Tool with Schema Awareness

Arguably, the above PostgreSQL database tool is a little simple isn't it? From where should the AI model know what database columns are available? You are right with that - we left out this crucial detail for more clarity. We'll modify our database tool to include schema information, which helps the LLM understand what data is available and how to query it correctly:

```python
def get_database_schema() -> str:
    """Retrieve the database schema information"""
    schema_query = """
    SELECT
        t.table_name,
        array_agg(
            c.column_name || ' ' ||
            c.data_type ||
            CASE
                WHEN c.is_nullable = 'NO' THEN ' NOT NULL'
                ELSE ''
            END
        ) as columns
    FROM information_schema.tables t
    JOIN information_schema.columns c
        ON c.table_name = t.table_name
    WHERE t.table_schema = 'public'
        AND t.table_type = 'BASE TABLE'
    GROUP BY t.table_name;
    """

    try:
        with psycopg2.connect("dbname=your_db user=your_user password=your_pas
            with conn.cursor() as cur:
                cur.execute(schema_query)
                schema = cur.fetchall()

                # Format schema information
                schema_str = "Database Schema:\n"
                for table_name, columns in schema:
                    schema_str += f"\n{table_name}\n"
                    for col in columns:
                        schema_str += f"  - {col}\n"

                return schema_str
    except Exception as e:
        return f"Error fetching schema: {str(e)}"

# Updated tool definition with schema information;
# Note that we could also dynamically fetch the schema when needed
tools = [
    {
        "type": "function",
        "function": {
            "name": "query_database",
            "description": """Execute a PostgreSQL SELECT query and return the
            Available tables and their schemas:

```

```
            users
              - id SERIAL PRIMARY KEY
              - email VARCHAR(255) NOT NULL
              - age INTEGER
              - location VARCHAR(100)
              - signup_date DATE NOT NULL
              - last_login TIMESTAMP WITH TIME ZONE
              - job_industry VARCHAR(100)

            user_activity
              - id SERIAL PRIMARY KEY
              - user_id INTEGER REFERENCES users(id)
              - activity_date DATE NOT NULL
              - activity_type VARCHAR(50) NOT NULL
            """,
            "parameters": {
                "type": "object",
                "properties": {
                    "query": {
                        "type": "string",
                        "description": "The SQL SELECT query to execute. Must
                    }
                },
                "required": ["query"],
                "additionalProperties": False
            },
            "strict": True
        }
    }
]

# Updated database query function
def query_database(query: str) -> str:
    """
    Execute a PostgreSQL query with schema awareness
    """
    if not query.lower().strip().startswith('select'):
        return json.dumps({
            "error": "Only SELECT queries are allowed for security reasons.",
            "schema": get_database_schema()  # Return schema for reference
        })

    try:
        with psycopg2.connect("dbname=your_db user=your_user password=your_pas
            with conn.cursor() as cur:
                cur.execute(query)

                # Get column names from cursor description
                columns = [desc[0] for desc in cur.description]
```

```
                # Fetch results and convert to list of dictionaries
                results = []
                for row in cur.fetchall():
                    results.append(dict(zip(columns, row)))

                return json.dumps({
                    "success": True,
                    "data": str(results),
                    "row_count": len(results),
                    "columns": columns
                })

    except Exception as e:
        return json.dumps({
            "error": str(e),
            "schema": get_database_schema()  # Return schema on error for hel
        })

# Usage example:
agent = Agent()

# The agent now knows about the schema and can write better queries
response = agent.process_query(
    "What's the age distribution of users by location and industry?"
)
```

Key improvements:

1. Schema information in tool description
2. Column types and constraints included
3. Schema returned on errors for better error handling
4. Detailed result structure with column names

This helps the LLM:

- Write correct queries on the first try
- Understand available data
- Handle errors better
- Provide more detailed responses

Further Reading

- DSPy: Automatic prompt tuning and creating
- Free and open source ChatGPT alternative: LibreChat
- How to: RAG with Azure AI search

More information on our managed RAG solution?

To Pondhouse AI

More tips and tricks on how to work with AI?

To our Blog

PONDHOUSE DATA OG

We know data.

Location

Pondhouse Data OG
Litscherweg 10/21
6800 Feldkirch
Austria

Phone

+43 650 9820448

Links

Impressum

Data Protection

Article-Archive