REVOLUTION
DATA PLATFORMS

🦜🔗 LangChain

LangChain Integrations

Presented by: Anum Khattak

Overview

- Azure Authentication Methods with LangChain
- LangChain Memories
- LangChain Agent
- **Lab 2:** Document processing and indexing

Azure Authentication Methods with LangChain

Azure Authentication Overview

Microsoft Entra Identity (formerly Azure AD): Comprehensive identity management.

Authentication Methods: OAuth 2.0, OpenID Connect, Managed Identities, Service Principals, Certificate-based Authentication

Register Application in Microsoft Entra Identity

Create an app registration for Langchain

Configure Authentication

OAuth/OpenID Connect: Setup redirect URIs and permissions.

Managed Identity: Enable on Azure hosted service.

Certificate-Based: Configure certificates through Azure Key Vault.

Azure Authentication Methods with LangChain – Cont.

Implement Authentication in LangChain

Tools: Use Microsoft Authentication Library (MSAL).

Process: Handle token acquisition, use, and refresh.

Securely Store Secrets

Azure Key Vault: Manage and access secrets securely, avoiding hard-coded credentials.

Access Azure Services

Authenticated Access: Secure interaction with Azure services under defined permissions.

LangChain Memories

Remember previous interactions or states	Context Preservation Personalization
Key Features of LangChain Memories	Contextual Recall Customizable Memory Size Efficient Resource Management

Types of Memories

ConversationBufferMemory

- Stores all previous chat messages in a buffer

ConversationSummaryMemory

- Summarizes past interactions to reduce memory usage.

ConversationKnowledgeMemory

- Extracts and stores factual knowledge from the conversation.

Types of Memories

VectorStoreRetrieverMemory

- Uses embeddings to store and retrieve memory.

EntityMemory

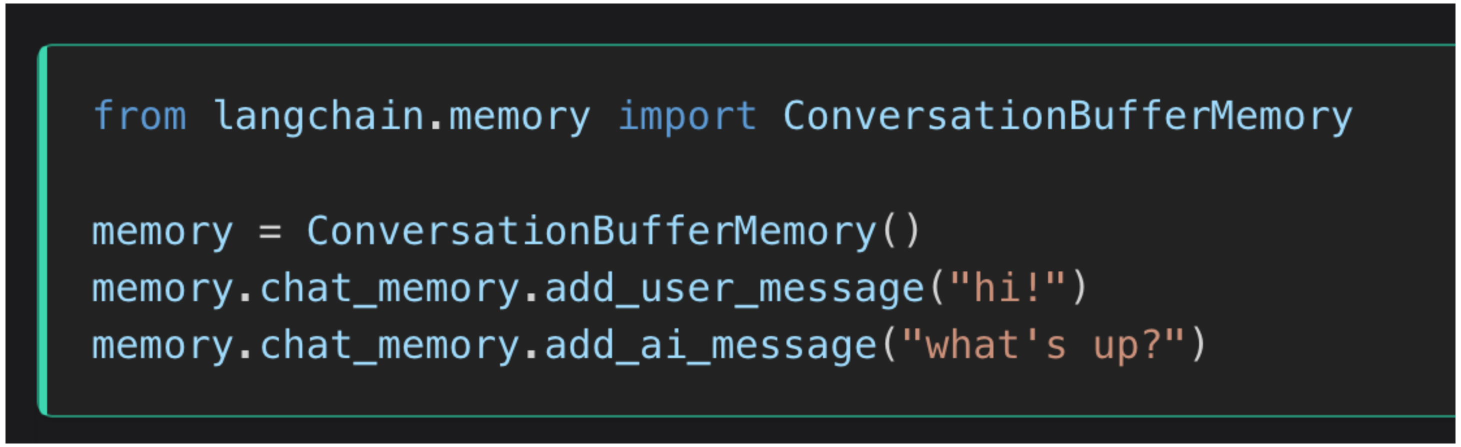
- Tracks information about specific entities mentioned during the conversation.

CombinedMemory

- Combines different memory types for a more customized experience.

Example (Basic Interaction): ConversationBufferMemory

Stores all previous chat messages in a buffer, passes those into the prompt template

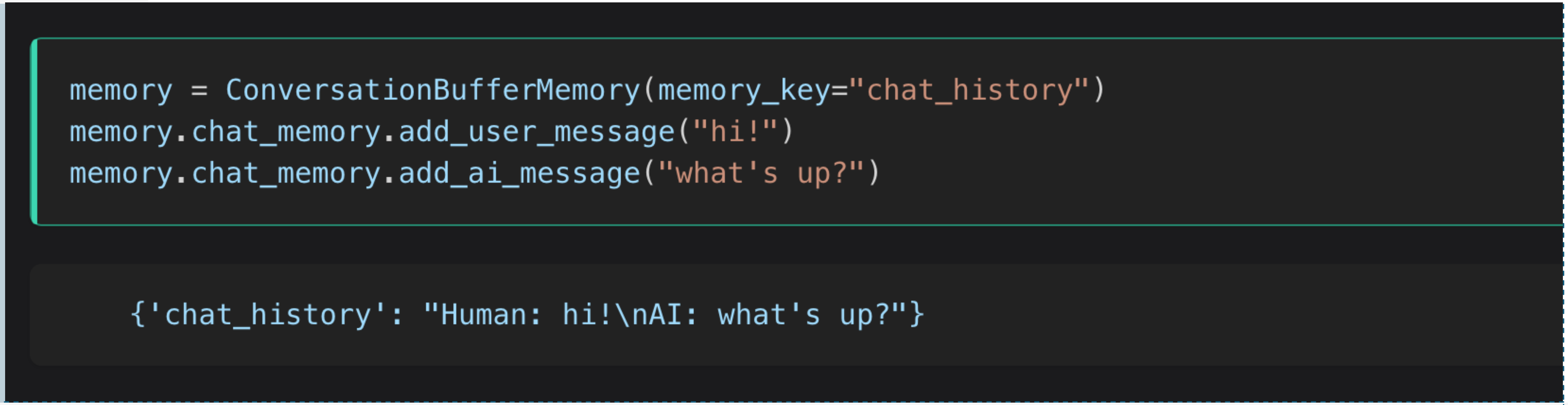


```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("what's up?")
```

What are few key concepts to understand?

1. What variables get returned from memory?
   - See these variables: memory.load_memory_variables({})
   - Example: Single key: {'history': "Human: hi!\nAI: what's up?"}



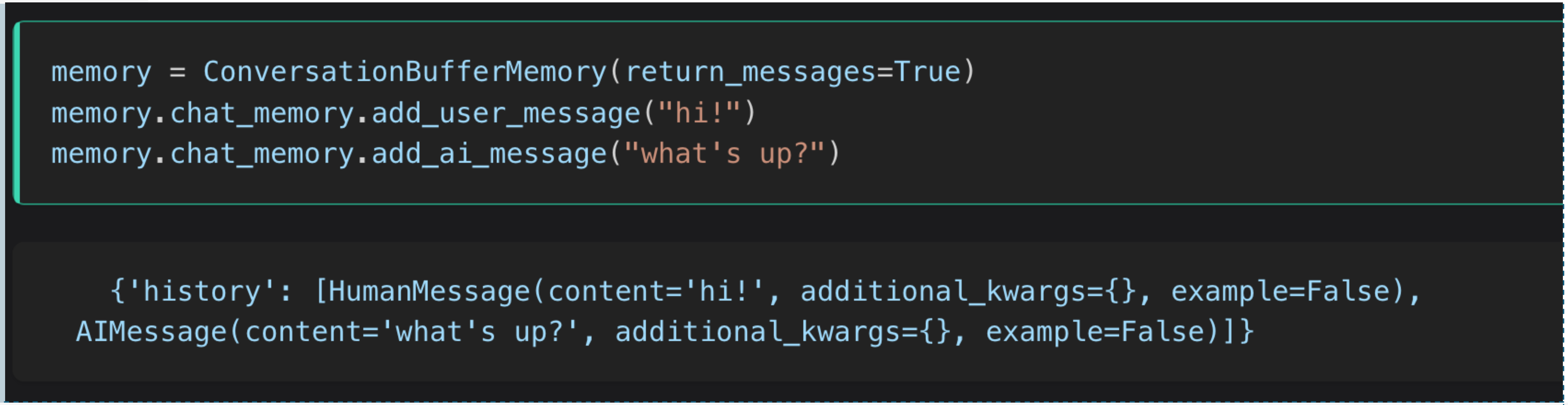
```
memory = ConversationBufferMemory(memory_key="chat_history")
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("what's up?")
```

```
    {'chat_history': "Human: hi!\nAI: what's up?"}
```

What are few key concepts to understand?

2. Whether memory is a string or a list of messages?

- By default, they are returned as a single string. In order to return as a list of messages, you can set return_messages=True



```
memory = ConversationBufferMemory(return_messages=True)
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("what's up?")
```

```
    {'history': [HumanMessage(content='hi!', additional_kwargs={}, example=False),
  AIMessage(content='what's up?', additional_kwargs={}, example=False)]}
```

What are few key concepts to understand?

3. What keys are saved to memory?

- Chains often deal with multiple input/output keys.
- To know which keys to save to chat history, use the input_key and output_key parameters in the memory types. (default: None)
- When there are multiple input/output keys, you must explicitly specify which one to use for saving.

LangChain Agent

An agent can be a tool or a function that performs tasks based on user input

Types of Agents

Tool Calling, OpenAI Tools, OpenAI Functions, XML, Structured Chats, JSON Chat, ReAct, Self Ask With Search

Dimension of Agent Types

Intended Model Type

Determines whether the agent is for Chat Models (message input/output) or LLMs (string input/output).

Supports Chat History

Determines whether the agent supports an ongoing conversation (chat history)

Supports Multi-Input Tools

Determines if agents support tools requiring multiple inputs.

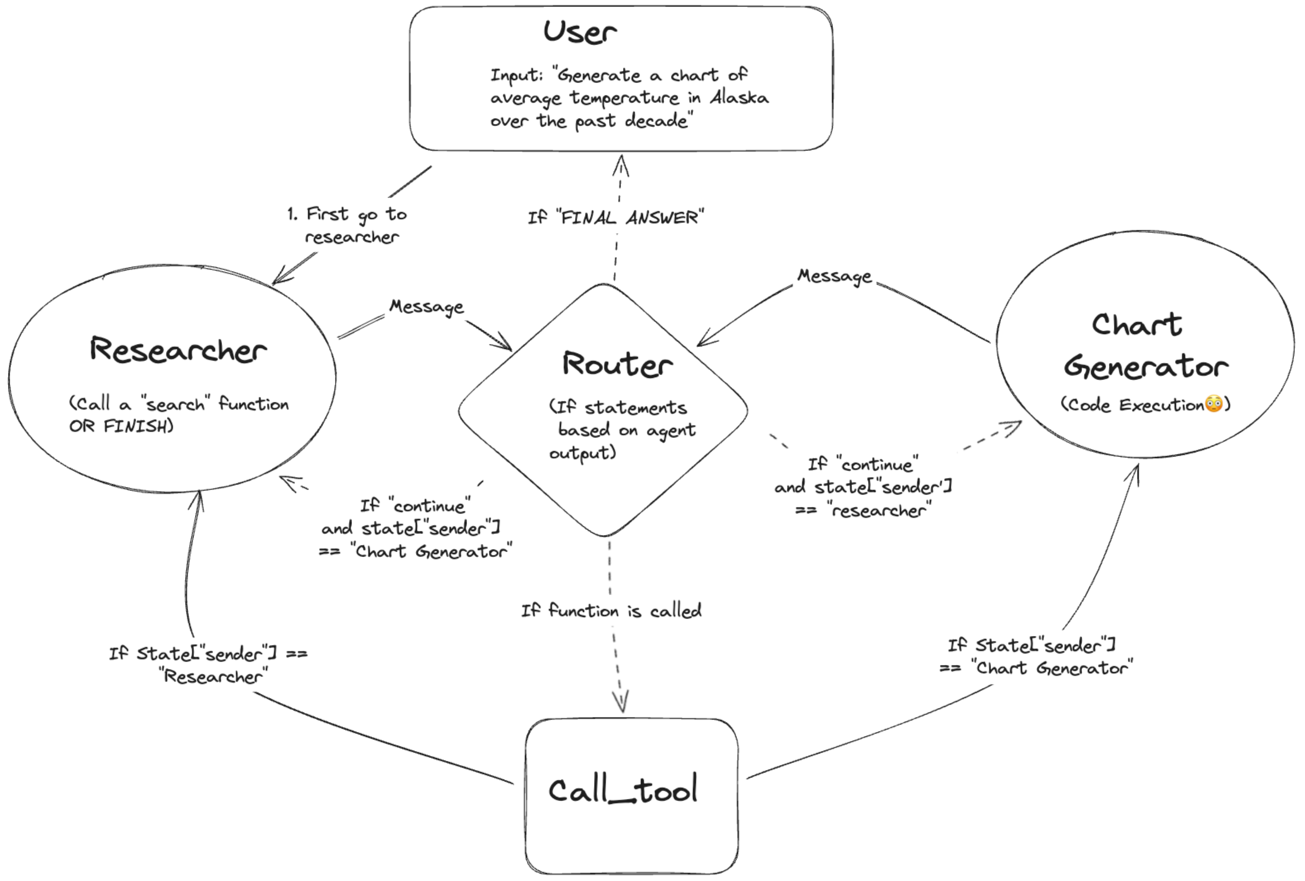
Dimension of Agent Types

Supports Parallel Function Calling

Indicates whether agents can call multiple tools at the same time, speeding up processes when possible.

Required Model Parameters

Specifies if additional parameters are needed (e.g., OpenAI function calling). If no parameters are required, all actions are managed through prompting.


User
Input: "Generate a chart of average temperature in Alaska over the past decade"
1. First go to researcher
If "FINAL ANSWER"
Message
Message
Researcher
(Call a "search" function OR FINISH)
Router
(If statements based on agent output)
Chart Generator
(Code Execution😳)
If "continue" and state["sender"] == "Chart Generator"
If "continue" and state["sender"] == "researcher"
If function is called
If State["sender"] == "Researcher"
If State["sender"] == "Chart Generator"
Call_tool

Multi-agent Collaboration Example

Image source: https://blog.langchain.dev/langgraph-multi-agent-workflows/

LangSmith

Facilitates the tracing and evaluation of language model applications and intelligent agents (aiding the transition from prototype to production)

Example Walkthrough:
https://python.langchain.com/v0.1/docs/langsmith/walkthrough/

Lab 2: Document Processing and Indexing

Helpful Resources

- LangChain Cookbook: https://github.com/langchain-ai/langchain/tree/master/cookbook

- LangSmith Cookbook: https://github.com/langchain-ai/langsmith-cookbook/tree/main

- Documentations:

Introduction:
https://python.langchain.com/docs/introduction/
https://api.python.langchain.com/en/latest/langchain_api_reference.html
https://docs.smith.langchain.com/
https://www.langchain.com/langsmith
https://python.langchain.com/v0.1/docs/langsmith/walkthrough/