**longchain building block 5) Memory**

4) **Memory**

You can provide attach memory to your so that it remembers the context of the conversation and responds accordingly.

1. **Buffer Memory:**The Buffer memory in Langchain is a simple memory buffer that stores the history of the conversation. It has a buffer property that returns the list of messages in the chat memory. The load\_memory\_variables function returns the history buffer. This type of memory is useful for storing and retrieving the immediate history of a conversation.
2. **Buffer Window Memory:** Buffer Window Memory is a variant of Buffer Memory. It also stores the conversation history but with a twist. It has a property k which determines the number of previous interactions to be stored. The buffer property returns the last k\*2 messages from the chat memory. This type of memory is useful when you want to limit the history to a certain number of previous interactions.
3. **Entity Memory:** The Entity Memory in Langchain is a more complex type of memory. It not only stores the conversation history but also extracts and summarizes entities from the conversation. It uses the Langchain Language Model (LLM) to predict and extract entities from the conversation. The extracted entities are then stored in an entity store which can be either in-memory or Redis-backed. This type of memory is useful when you want to extract and store specific information from the conversation.

Each of these memory types has its own use cases and trade-offs. Buffer Memory and Buffer Window Memory are simpler and faster but they only store the conversation history. Entity Memory, on the other hand, is more complex and slower but it provides more functionality by extracting and summarizing entities from the conversation.

As for the data structures and algorithms used, it seems that Langchain primarily uses lists and dictionaries to store the memory. The algorithms are mostly related to text processing and entity extraction, which involve the use of the Langchain Language Model.

1. **Conversation Knowledge Graph Memory:**The Conversation Knowledge Graph Memory is a sophisticated memory type that integrates with an external knowledge graph to store and retrieve information about knowledge triples in the conversation. It uses the Langchain Language Model (LLM) to predict and extract entities and knowledge triples from the conversation. The extracted entities and knowledge triples are then stored in a NetworkxEntityGraph, which is a type of graph data structure provided by the NetworkX library. This memory type is useful when you want to extract, store, and retrieve structured information from the conversation in the form of a knowledge graph.
2. **ConversationSummaryMemory:** The ConversationSummaryMemory is a type of memory that summarizes the conversation history. It uses the LangChain Language Model (LLM) to generate a summary of the conversation. The summary is stored in a buffer and is updated every time a new message is added to the conversation. This memory type is useful when you want to maintain a concise summary of the conversation that can be used for reference or to provide context for future interactions.
3. **ConversationSummaryBufferMemory:**ConversastionSummaryBufferMemory is similar to the ConversationSummaryMemory but with an added feature of pruning. If the conversation becomes too long (exceeds a specified token limit), the memory prunes the conversation by summarizing the pruned part and adding it to a moving summary buffer. This ensures that the memory does not exceed its capacity while still retaining the essential information from the conversation.
4. **ConversationTokenBufferMemory:** ConversationTokenBufferMemory is a type of memory that stores the conversation history in a buffer. It also has a pruning feature similar to the ConversationSummaryBufferMemory. If the conversation exceeds a specified token limit, the memory prunes the earliest messages until it is within the limit. This memory type is useful when you want to maintain a fixed-size memory of the most recent conversation history.
5. **VectorStore-Backed Memory:** The VectorStore-Backed Memory is a memory type that is backed by a VectorStoreRetriever. The VectorStoreRetriever is used to retrieve relevant documents based on a query. The retrieved documents are then stored in the memory. This memory type is useful when you want to store and retrieve information in the form of vectors, which is particularly useful for tasks such as semantic search or similarity computation.

**Callback Handlers**

LangChain provides a callbacks system that allows you to hook into the various stages of your LLM application. This is useful for logging, monitoring, streaming, and other tasks. The BaseCallbackHandler class is used to define the actions to be performed inside the hook functions.

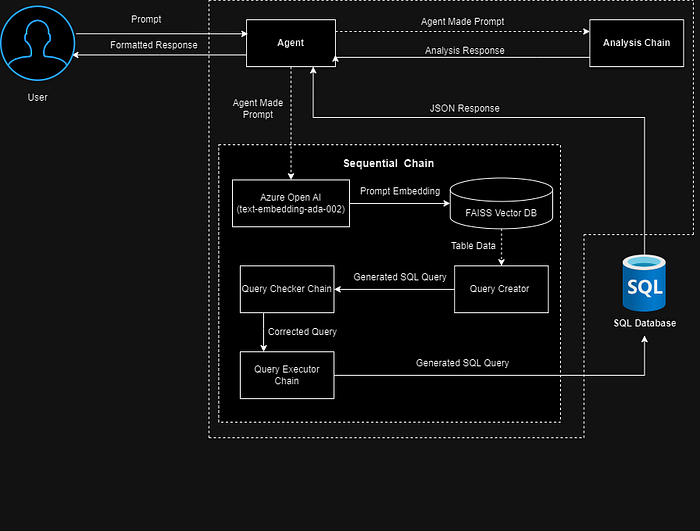
Some available hooks are — on\_llm\_start, on\_agent\_end, on\_chain\_start. The names of these hooks are self explanatory. Code can be written inside these functions which has to be performed when those functions are called.

The object of the BaseCallbackHandler class can provided to the appropriate agent, chain, tool etc.

**Walkthrough — Project Utilizing Langchain**

The following image displays the architecture I’ve used in a project that helps in answering questions on data available in a large SQL database, by creating SQL queries to fetch relevant data, then analyzing the fetched data and then returning a response in the form of answer.

Press enter or click to view image in full size



In the image above it can be seen that the agent has two chains available to it as tools which are -

1. Analysis Chain (For doing analysis on data in memory)
2. Sequential Chain (For writing SQL queries)

**NOTE: While there are predefined and configured agents, tools and chains are available, custom versions of all of these can be made.**

**NOTE: Chains can be provided as tools to the agent. Similarly, Tools can be made available as a chain segment in chains as well. The user has a lot of freedom to customize these agents, tools, chains and can plug, sequence them according to their needs.**

tools=[  
 Tool.from\_function(func=sequentialchain.\_run,  
 name="tool1",  
 description="Useful when user wants information about revenue, margin, employee and projects. Input is a descriptive plain text formed using user question and chat history and output is the result."  
 ),  
   
 Tool.from\_function(func=analysis.\_run,  
 name="tool2",  
 description="Useful when you want to do some calculations and statistical analysis using the memory. Input is a list of numbers with description of what is to be done to it or a mathematical equation of number and output is result."  
 )  
 ]

The code snippet above shows a tools array in which two chains, namely — sequentialchain and analysis chain are provided as tools.

memory = ConversationBufferWindowMemory(memory\_key="chat\_history",return\_messages=True,k=7)  
llm = AzureChatOpenAI(  
 temperature=0,  
 deployment\_name="\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",  
 model\_name="gpt-35-turbo-16k",  
 openai\_api\_base="\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",  
 openai\_api\_version="2023-07-01-preview",  
 openai\_api\_key="\*\*\*\*\*\*\*\*\*\*\*\*\*\*",  
 openai\_api\_type="azure"  
)  
agent\_chain=initialize\_agent(  
 tools,  
 llm,  
 agent=AgentType.OPENAI\_FUNCTIONS,  
 verbose=True,  
 agent\_kwargs=agent\_kwargs,  
 memory=memory,  
 callbacks=[MyCustomHandler()]  
)

The initialize\_agent function creates an agent object with the specifications you have entered in the function as arguments.

This agent is what manages the whole interaction with the LLM. The agent is run like this → answer=agent\_chain.run(“the query put in by the user”)

The tools and memory are provided to the agent. I have used the ConversationBufferWindowMemory() which allows me to specify the value k as 7. This means that the last 7 conversations (input and output) are available to the LLM when you ask a new question.

class sequentialchain(BaseTool):  
 def \_run(  
 self, run\_manager: Optional[CallbackManagerForToolRun] = None  
 ) -> str:  
 tables = similarity\_search(self)  
 print(tables)  
 sql\_chain = SQLAgent(tables)  
 querycheckchain=querycheckfunc(tables)  
 executorchainobj=QueryExecutorChain(user\_query=self)  
 overall\_chain = SimpleSequentialChain(chains=[sql\_chain, querycheckchain, executorchainobj], verbose=True)  
 review = overall\_chain.run(self)  
 return review

The similarity\_search() function gets the appropriate table descriptions from the vector db and provides it as input variables for the chains so they can write proper SQL queries.

The SimpleSequentialChain() has 3 chains passed to it — sql\_chain, querycheckchain, executorchainobj which are run in succession. The output of the first chain is passed to the second chain as an input variable and the output of the third chain is passed to the third chain as an input variable.

The **sql\_chain**— based on a prompt on how to create SQL queries and table descriptions makes SQL queries.

The**querycheckchain**— Receives the SQL query from sql\_chain, then corrects all the errors, syntax, adds missing elements if any and makes it compliant to the standards described in prompt.

The **executorchainobj**— This chain segment is actually a tool passed as a chain. It Receives the SQL query that is ready to be run on the database.

The output or fetched data after running the SQL query is then received by the agent which had called the sequentialchain. The agent would interpret the fetched in accordance to the user’s input question, format it and provide the final answer/response to the user. If the agent wants to do some analysis on the fetched data it can then send this data to the analysis chain the output of which can then be formatted into a final answer/response.

If the question asked by a user is a follow up question, the agent can look at the memory and if it can find the necessary data in it, then it can formulate the answer based on the memory alone as well, or if it thinks some analysis is to be done then it can also directly send that data to the analysis chain as well.

Agent decides when to use the memory, which tool to use or if to use any tool at all.

**NOTE: I have used a custom chain (analysis chain) provided as a tool to the agent. There are predefined tools for all sorts of purposes like math, SQL connections, google drive connections, AWS Lambda connections etc.**

The analysis chain is a normal LLM call chain and has prompt instructions to do various types of statistical analysis (mean, median, standard deviation, variance etc.), calculate growth, percentages and other mathematical operations.

Callback Handlers can also be added to perform various tasks at certain defined stages of the application run cycle.

class MyCustomHandler(BaseCallbackHandler):  
 def on\_llm\_new\_token(self, token: str, \*\*kwargs) -> None:  
 print(f"My custom handler, token: {token}")  
 for key, value in kwargs.items():  
 print("%s == %s" % (key, value))  
  
 def on\_llm\_end(  
 self,  
 outputs,  
 \*,  
 run\_id,  
 parent\_run\_id,  
 \*\*kwargs,  
 ):  
 """Run when llm call ends running."""  
 print(run\_id)  
  
  
 def on\_chain\_end(  
 self,  
 outputs,  
 \*,  
 run\_id,  
 parent\_run\_id,  
 \*\*kwargs,  
 ):  
 """Run when chain ends running."""  
 print(run\_id)

The CallbackHandler — MyCustomHandler() has been configured with certains set of code that would run on — on\_chain\_end, on\_llm\_end. The names of these hooks are self explanatory. When the object of this class is provided to the appropriate agent, tool, chain etc., these code inside these hooks would run as their names suggest.

All sorts of hooks such as on\_chain\_start, on\_chain\_end, on\_tool\_start, on\_tool\_end are available which can be specified to do certain tasks under the BaseCallbackHandler Class.

prompt\_template = PromptTemplate(input\_variables=["query"], template=template)  
query\_check\_chain = LLMChain(llm=llm, prompt=prompt\_template, output\_key="review", callbacks=[MyCustomHandler()])

The hook in this case — MyCustomHandler(), can be provided to the appropriate agent, tool or chain in the callbacks argument.

When all of this is set up when the agent is run — (agent\_chain,run(“user’s input question”)), the application can self write the sql queries, run them to fetch data from the database, analyse the data, and give proper information as output to the user. The user never has to even open the database, write sql queries, fetch the data, dig through it for analysis etc. Everything happens automatically from start to finish.

**5) Memory**

* **ConversationBufferMemory** (all turns), **BufferWindowMemory** (last k).
* **ConversationSummaryMemory** (token-efficient).
* **EntityMemory**, **VectorStoreRetrieverMemory** (long-term recall).
* **Backends**: in-memory, Redis, SQL, etc.

Let’s deep dive into **LangChain Memory**, which is crucial for building conversational or context-aware AI applications. Memory in LangChain stores **state** across multiple interactions with an LLM. Think of it as the AI’s “short-term and long-term memory.” Here’s a structured breakdown of the types you listed:

**1️⃣ ConversationBufferMemory**

* **Purpose:** Stores **all turns** of the conversation in order.
* **Use-case:** Useful for short chats where you want the model to always see the full history.
* **How it works:** Appends every user and AI message into a buffer string that’s passed as part of the prompt.
* **Backend support:** Usually **in-memory** by default.
* **Pros:** Simple, direct.
* **Cons:** Not scalable for very long conversations → token limit issues.

**Example:**

python

CopyEdit

from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()

memory.save\_context({"input": "Hi"}, {"output": "Hello!"})

print(memory.load\_memory\_variables({})) # Returns entire conversation history

**2️⃣ BufferWindowMemory**

* **Purpose:** Keeps only the **last k turns** of conversation.
* **Use-case:** When recent context matters, but you don’t want to hit token limits.
* **Example:** Keep only the last 5 user/assistant messages.
* **Pros:** Token-efficient for ongoing chats.
* **Cons:** Loses long-term context beyond the window.

python

CopyEdit

from langchain.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(k=5) # last 5 turns only

**3️⃣ ConversationSummaryMemory**

* **Purpose:** Summarizes older parts of the conversation to save tokens.
* **How it works:**
  + Keeps a rolling summary of conversation so far.
  + Combines **summary** with the recent messages to feed the model.
* **Pros:** Token-efficient, scalable for long chats.
* **Cons:** Summary might lose some fine-grained details.

python

CopyEdit

from langchain.memory import ConversationSummaryMemory

from langchain.chat\_models import ChatOpenAI

llm = ChatOpenAI()

memory = ConversationSummaryMemory(llm=llm)

**4️⃣ EntityMemory**

* **Purpose:** Stores **specific entities** or facts about the user.
* **Use-case:** Personalization: remembering user name, preferences, location, etc.
* **Example:** Remember “Rajeev likes coffee” across sessions.

**5️⃣ VectorStoreRetrieverMemory**

* **Purpose:** Enables **long-term memory** by storing facts in a **vector database**.
* **How it works:**
  + Stores embeddings of important conversation pieces or knowledge.
  + Later, relevant pieces can be **retrieved** via similarity search.
* **Backends:** **Chroma, FAISS, Pinecone, RedisVector**, etc.
* **Pros:** Infinite long-term memory, retrieval-based, scalable.
* **Cons:** Slightly more complex to set up.

python

CopyEdit

from langchain.memory import VectorStoreRetrieverMemory

from langchain.vectorstores import Chroma

from langchain.embeddings import OpenAIEmbeddings

vector\_store = Chroma(persist\_directory="./vectordb", embedding\_function=OpenAIEmbeddings())

memory = VectorStoreRetrieverMemory(vectorstore=vector\_store.as\_retriever())

**6️⃣ Backends**

Memory can be stored in different backends depending on scale and persistence requirements:

| **Backend** | **Use-case** |
| --- | --- |
| **In-memory** | Simple, ephemeral sessions |
| **Redis** | Fast, shared memory across servers |
| **SQL** | Persistent storage for structured data |
| **Vector DB** | Long-term memory for embeddings |

✅ **Summary Table of Memory Types**

| **Memory Type** | **Scope** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| ConversationBufferMemory | All turns | Simple | Token-heavy |
| BufferWindowMemory | Last k turns | Lightweight | Short-term only |
| ConversationSummaryMemory | Summarized | Token-efficient | Some info may be lost |
| EntityMemory | Specific entities | Personalized | Needs careful design |
| VectorStoreRetrieverMemory | Long-term | Persistent, scalable | More complex setup |

If you want, I can **draw a mental model diagram** showing **all memory types with their backends and flow in a conversation**, which makes it super easy to visualize how LangChain memory works in practice.

Do you want me to do that next?

**You said:**

deep dive explanation with code example - 1️⃣ ConversationBufferMemory

**ChatGPT said:**

Perfect! Let’s do a **deep dive into ConversationBufferMemory**, step by step, with a full code example showing how it works, how it stores conversation, and how it integrates with an LLM in LangChain.

**1️⃣ What is ConversationBufferMemory?**

* ConversationBufferMemory stores **all turns** (user + AI messages) in a **buffer string**.
* Each time a new message is added, it **appends** to the existing conversation.
* This is the **simplest type of memory** in LangChain.
* Good for **short chats** where token limits aren’t a concern.

**Key Features:**

* Keeps full context for the LLM.
* Automatically returns conversation history for prompt construction.
* Can be configured to remember only **input-output pairs** or **all messages**.

**2️⃣ How it works internally**

1. User sends a message → stored in memory.
2. AI responds → stored in memory.
3. On next request, memory can **inject the previous conversation** into the prompt for context.

**Memory flow:**

vbnet

CopyEdit

User: Hi

Memory stores: "User: Hi"

AI: Hello! How can I help you today?

Memory stores: "User: Hi\nAI: Hello! How can I help you today?"

Next user message includes all previous:

User: Can you tell me a joke?

Prompt to AI: "User: Hi\nAI: Hello! How can I help you today?\nUser: Can you tell me a joke?"

**3️⃣ Example Code**

python

CopyEdit

# Install packages if not already installed

# pip install langchain openai

from langchain.chat\_models import ChatOpenAI

from langchain.memory import ConversationBufferMemory

from langchain.chains import ConversationChain

# 1️⃣ Initialize memory

memory = ConversationBufferMemory(

memory\_key="chat\_history", # key for storing conversation

return\_messages=True # returns messages instead of plain string

)

# 2️⃣ Initialize LLM

llm = ChatOpenAI(temperature=0.7, model\_name="gpt-3.5-turbo")

# 3️⃣ Create ConversationChain with memory

conversation = ConversationChain(

llm=llm,

memory=memory

)

# 4️⃣ Interact with the chain

print("User: Hi there!")

response1 = conversation.predict(input="Hi there!")

print("AI:", response1)

print("\nUser: Can you tell me a joke?")

response2 = conversation.predict(input="Can you tell me a joke?")

print("AI:", response2)

print("\nUser: Remind me what I said first?")

response3 = conversation.predict(input="Remind me what I said first?")

print("AI:", response3)

# 5️⃣ Inspect memory

print("\n=== Conversation Stored in Memory ===")

print(memory.load\_memory\_variables({})["chat\_history"])

**4️⃣ What happens here**

* memory.load\_memory\_variables({}) shows **all previous turns** in order.
* With return\_messages=True, it returns a **list of Message objects** instead of a single string.
* Every predict() call **updates the memory automatically**.
* The LLM always sees the **full conversation history** in the prompt.

**5️⃣ Output Example (simulated)**

vbnet

CopyEdit

User: Hi there!

AI: Hello! How can I help you today?

User: Can you tell me a joke?

AI: Sure! Why did the scarecrow win an award? Because he was outstanding in his field!

User: Remind me what I said first?

AI: You said "Hi there!"

Memory after these three turns:

python

CopyEdit

[

{"role": "user", "content": "Hi there!"},

{"role": "assistant", "content": "Hello! How can I help you today?"},

{"role": "user", "content": "Can you tell me a joke?"},

{"role": "assistant", "content": "Sure! Why did the scarecrow win an award? Because he was outstanding in his field!"},

{"role": "user", "content": "Remind me what I said first?"},

{"role": "assistant", "content": "You said 'Hi there!'"}

]

✅ **Key Points**

* ConversationBufferMemory is ideal for **short-term, full context memory**.
* It grows linearly; for **very long conversations**, consider BufferWindowMemory or ConversationSummaryMemory.
* Works seamlessly with ConversationChain or custom LLM chains.

If you want, I can next show a **visual flow of how ConversationBufferMemory stores and retrieves messages during a conversation**, which makes it super intuitive to understand memory in LangChain.

Do you want me to do that next?

**You said:**

yes deep explanation - I can next show a visual flow of how ConversationBufferMemory stores and retrieves messages during a conversation, which makes it super intuitive to understand memory in LangChain.

**ChatGPT said:**

Perfect! Let’s go step by step with a **visual flow** of how ConversationBufferMemory works in LangChain. I’ll also explain the retrieval mechanism during a conversation. Think of it as **memory’s lifecycle** in a conversational chain.

**1️⃣ Conceptual Overview**

ConversationBufferMemory behaves like a **chat log**:

sql

CopyEdit

User → Memory → LLM → Memory → User

* **User Input** → goes into memory.
* **Memory** → keeps a record of all past turns.
* **LLM Prompt** → combines the memory with the latest user input.
* **LLM Output** → goes back into memory.
* **Repeat** for the next turn.

**2️⃣ Visual Flow Diagram**

pgsql

CopyEdit

+-----------------+

| User Input | e.g., "Hi there!"

+--------+--------+

|

v

+-----------------+

| Conversation |

| Buffer Memory |

| - stores turn |

| 1: User: Hi |

+--------+--------+

|

v

+-----------------+

| LLM Prompt |

| Combines: |

| 1. Memory |

| 2. Current input|

+--------+--------+

|

v

+-----------------+

| LLM Output |

| e.g., "Hello!" |

+--------+--------+

|

v

+-----------------+

| Memory Update |

| - stores AI |

| response |

+--------+--------+

|

v

+-----------------+

| Return Output |

| to User |

+-----------------+

**3️⃣ Example with Three Turns**

**Turn 1: User says “Hi there!”**

* Memory stores:  
  ["User: Hi there!"]
* Prompt to LLM: "User: Hi there!"
* LLM Response: "Hello! How can I help you today?"
* Memory after update:

css

CopyEdit

["User: Hi there!", "AI: Hello! How can I help you today?"]

**Turn 2: User says “Can you tell me a joke?”**

* Memory stores new input:

arduino

CopyEdit

["User: Hi there!", "AI: Hello! How can I help you today?",

"User: Can you tell me a joke?"]

* Prompt to LLM includes **full conversation**:

vbnet

CopyEdit

User: Hi there!

AI: Hello! How can I help you today?

User: Can you tell me a joke?

* LLM responds: "Sure! Why did the scarecrow win an award? Because he was outstanding in his field!"
* Memory after update:

arduino

CopyEdit

["User: Hi there!",

"AI: Hello! How can I help you today?",

"User: Can you tell me a joke?",

"AI: Sure! Why did the scarecrow win an award? Because he was outstanding in his field!"]

**Turn 3: User says “Remind me what I said first?”**

* Memory includes **all previous turns**.
* LLM sees **entire chat history** in prompt.
* Memory now stores **all six messages** (3 user + 3 AI).

**4️⃣ Key Observations**

1. **Memory grows linearly** – every turn adds to the buffer.
2. **LLM sees full history** – no context is lost (good for short conversations).
3. **No summarization** – unlike ConversationSummaryMemory.
4. **Token concern** – for long chats, you might hit LLM limits.
5. **Retrieval is simple** – memory.load\_memory\_variables({}) always gives all stored messages.

**5️⃣ Code Demonstration with Flow Prints**

python

CopyEdit

from langchain.chat\_models import ChatOpenAI

from langchain.memory import ConversationBufferMemory

from langchain.chains import ConversationChain

memory = ConversationBufferMemory(return\_messages=True)

llm = ChatOpenAI(temperature=0)

conversation = ConversationChain(llm=llm, memory=memory)

for msg in ["Hi there!", "Can you tell me a joke?", "Remind me what I said first?"]:

response = conversation.predict(input=msg)

print(f"User: {msg}")

print(f"AI: {response}")

print("Memory Snapshot:", memory.load\_memory\_variables({})["chat\_history"])

print("------")

**Output snippet:**

pgsql

CopyEdit

Memory Snapshot: [

{"role": "user", "content": "Hi there!"},

{"role": "assistant", "content": "Hello! How can I help you today?"}

]

------

Memory Snapshot: [

{"role": "user", "content": "Hi there!"},

{"role": "assistant", "content": "Hello! How can I help you today?"},

{"role": "user", "content": "Can you tell me a joke?"},

{"role": "assistant", "content": "Sure! Why did the scarecrow win an award? Because he was outstanding in his field!"}

]

------

**✅ Summary**

* ConversationBufferMemory = **full chat history in memory**.
* Every user input and AI response is **stored**.
* LLM always sees the **complete context**.
* Ideal for short chats, debugging, or when full context is critical.
* Not ideal for **very long conversations** due to token constraints.

If you want, I can next **show the comparison visual** between ConversationBufferMemory, BufferWindowMemory, and ConversationSummaryMemory—so you can clearly see **short-term vs long-term vs summarized memory flows**. This helps in deciding which memory to use for your use case.

Do you want me to do that next?

**You said:**

deep dive details into 2️⃣ BufferWindowMemory

**ChatGPT said:**

Absolutely! Let’s do a **deep dive into BufferWindowMemory** in LangChain, step by step, with clear explanations, internal workings, and a code example.

**1️⃣ What is BufferWindowMemory?**

* BufferWindowMemory is a **memory type that only keeps the last k turns** of the conversation.
* Unlike ConversationBufferMemory which stores **all conversation turns**, this memory **slides a fixed-size window** over the conversation.
* Useful when:
  + You want **recent context** but don’t need the full history.
  + Token limits are a concern.
  + The conversation is long-running.

**Key Concept:**  
Think of it like a **moving window** that always captures the last k messages.

**2️⃣ How it works internally**

1. User sends a message → added to memory.
2. AI responds → added to memory.
3. **Memory keeps only the last k turns** (user + AI pairs).
4. On the next turn, only **these k turns** are sent to the LLM.

**Visual Flow for k=2 (last 2 turns):**

yaml

CopyEdit

Conversation so far:

Turn 1: User: Hi

Turn 2: AI: Hello!

Turn 3: User: How are you?

Turn 4: AI: I'm fine, thanks.

Memory (last 2 turns only):

- User: How are you?

- AI: I'm fine, thanks.

Next LLM prompt uses only these 2 turns.

**3️⃣ Advantages vs Full Buffer**

| **Feature** | **ConversationBufferMemory** | **BufferWindowMemory** |
| --- | --- | --- |
| Stores full history | ✅ | ❌ only last k |
| Token efficiency | ❌ | ✅ |
| Context for LLM | Full | Partial |
| Ideal for | Short or medium chats | Long or ongoing chats |

**4️⃣ Example Code**

python

CopyEdit

# Install LangChain if needed

# pip install langchain openai

from langchain.chat\_models import ChatOpenAI

from langchain.memory import ConversationBufferWindowMemory

from langchain.chains import ConversationChain

# 1️⃣ Initialize memory to keep last 2 turns only

memory = ConversationBufferWindowMemory(k=2, return\_messages=True)

# 2️⃣ Initialize LLM

llm = ChatOpenAI(temperature=0, model\_name="gpt-3.5-turbo")

# 3️⃣ Create ConversationChain

conversation = ConversationChain(

llm=llm,

memory=memory

)

# 4️⃣ Interact with the chain

messages = [

"Hi there!",

"Can you tell me a joke?",

"Who won the last world cup?",

"Remind me what I asked first?"

]

for msg in messages:

response = conversation.predict(input=msg)

print(f"User: {msg}")

print(f"AI: {response}")

print("Memory Snapshot:", memory.load\_memory\_variables({})["history"])

print("------")

**5️⃣ How Memory Evolves (Example)**

Assume k=2 (last 2 turns):

1. **Turn 1: "Hi there!"**

vbnet

CopyEdit

Memory: ["User: Hi there!", "AI: Hello!"]

1. **Turn 2: "Can you tell me a joke?"**

vbnet

CopyEdit

Memory: ["User: Hi there!", "AI: Hello!", "User: Can you tell me a joke?", "AI: Here's a joke..."]

(Still within window if k=2 means 2 pairs = 4 messages.)

1. **Turn 3: "Who won the last world cup?"**

sql

CopyEdit

Memory keeps only last 2 pairs:

["User: Can you tell me a joke?", "AI: Here's a joke...",

"User: Who won the last world cup?", "AI: Argentina won in 2022"]

1. **Turn 4: "Remind me what I asked first?"**

vbnet

CopyEdit

Memory:

["User: Who won the last world cup?", "AI: Argentina won in 2022",

"User: Remind me what I asked first?", "AI: Sorry, I don't remember earlier messages beyond last 2 turns."]

✅ Notice: Only **last k turns** are kept. Older context is discarded.

**6️⃣ Key Points**

* k can be **any integer**, controlling how many turns to remember.
* **Efficient for long chats**: avoids token explosion.
* LLM sees **recent context** only, so older info may be forgotten.
* Works well in **support chatbots, interactive assistants, or multi-turn LLM applications** where only recent conversation matters.

If you want, I can next **deep dive into ConversationSummaryMemory**, which is **token-efficient long-term memory**, showing how it **summarizes past turns** instead of discarding them like BufferWindowMemory.

Do you want me to cover that next?

**You said:**

yes please - I can next deep dive into ConversationSummaryMemory, which is token-efficient long-term memory

**ChatGPT said:**

Perfect! Let’s deep dive into **ConversationSummaryMemory**—LangChain’s **token-efficient memory** for long conversations. We’ll cover **what it is, how it works internally, use cases, and a full code example**.

**1️⃣ What is ConversationSummaryMemory?**

* Unlike ConversationBufferMemory (full history) or BufferWindowMemory (last k turns), **ConversationSummaryMemory maintains a rolling summary** of the conversation.
* **Older turns are summarized** to reduce token usage while preserving context.
* Useful when:
  + Conversations are **very long**.
  + Token limits of LLM are a concern.
  + You want the model to **recall key information** without sending the full history.

**Key Concept:**  
Think of it as **memory compression**: the LLM remembers the gist instead of every word.

**2️⃣ How it works internally**

1. **User input** → added to memory.
2. **LLM generates response** → memory stores both response and input.
3. When memory exceeds a **token threshold or turn limit**:
   * The memory **calls a summarization LLM**.
   * Older conversation is condensed into a **summary string**.
4. **Next LLM prompt** = summary + recent messages.

**Visual Flow:**

vbnet

CopyEdit

User: Hi

AI: Hello!

Memory Summary: "User greeted AI"

User: I love pizza

AI: Nice! What's your favorite type?

Memory Summary: "User greeted AI and mentioned they love pizza"

...

Next prompt to LLM = Memory Summary + recent messages

**3️⃣ Advantages vs Other Memories**

| **Feature** | **ConversationBufferMemory** | **BufferWindowMemory** | **ConversationSummaryMemory** |
| --- | --- | --- | --- |
| Stores full history | ✅ | ❌ last k | ❌ summarized |
| Token efficient | ❌ | ✅ last k | ✅ compresses long chats |
| Context for LLM | Full | Partial | Summarized + recent |
| Ideal for | Short chats | Long chats with recent context | Long-running conversations |

**4️⃣ Code Example**

python

CopyEdit

# pip install langchain openai

from langchain.chat\_models import ChatOpenAI

from langchain.memory import ConversationSummaryMemory

from langchain.chains import ConversationChain

# 1️⃣ Initialize LLM

llm = ChatOpenAI(temperature=0, model\_name="gpt-3.5-turbo")

# 2️⃣ Initialize ConversationSummaryMemory

memory = ConversationSummaryMemory(

llm=llm,

max\_token\_limit=200, # optional: summarize when token limit exceeds

return\_messages=True

)

# 3️⃣ Create ConversationChain

conversation = ConversationChain(

llm=llm,

memory=memory

)

# 4️⃣ Interact with the chain

messages = [

"Hi there!",

"I love pizza.",

"Can you suggest a good pizza recipe?",

"Remind me what I said before about my food preferences?"

]

for msg in messages:

response = conversation.predict(input=msg)

print(f"User: {msg}")

print(f"AI: {response}")

print("Memory Snapshot:", memory.load\_memory\_variables({})["summary"])

print("------")

**5️⃣ How Memory Evolves (Example)**

Assume max\_token\_limit triggers summarization after 2 turns:

1. **Turn 1: "Hi there!"**
   * Memory summary: "User greeted AI."
2. **Turn 2: "I love pizza."**
   * Memory summary: "User greeted AI and mentioned they love pizza."
3. **Turn 3: "Can you suggest a good pizza recipe?"**
   * LLM sees: "User greeted AI and mentioned they love pizza. User asked for a pizza recipe."
   * Memory summary updates: "User greeted AI, loves pizza, and asked for a pizza recipe."
4. **Turn 4: "Remind me what I said before about my food preferences?"**
   * LLM uses **summary + recent input** → can respond correctly even though full chat history isn’t stored.
   * Memory summary stays concise: "User greeted AI, loves pizza, asked for a pizza recipe, and inquired about food preferences."

✅ Notice: **summary grows gradually**, not storing every word, saving tokens while preserving essential context.

**6️⃣ Key Points**

* **Token-efficient:** Good for long-running conversations.
* **Dynamic:** Updates summary as conversation evolves.
* **Combines:** Summary + recent messages for prompt construction.
* **Use cases:**
  + Customer support bots (long chat sessions)
  + Personal assistants remembering user preferences
  + Multi-turn task-oriented applications

If you want, I can next do a **comparison visual** showing **BufferMemory vs BufferWindowMemory vs ConversationSummaryMemory** in a single diagram with memory flow. This makes it very clear **which memory type to choose** for different scenarios.

Do you want me to do that next?

**You said:**

lets deep dive into - ConversationSummaryMemory • Purpose: Summarizes older parts of the conversation to save tokens. • How it works: o Keeps a rolling summary of conversation so far. o Combines summary with the recent messages to feed the model. • Pros: Token-efficient, scalable for long chats. • Cons: Summary might lose some fine-grained details.

**ChatGPT said:**

Absolutely! Let’s **deep dive into ConversationSummaryMemory** in LangChain with a detailed explanation, step-by-step flow, and a code example. We’ll focus on its **purpose, working mechanism, pros/cons, and practical usage**.

**1️⃣ Purpose**

ConversationSummaryMemory is designed to **store long conversations efficiently**.

* Instead of keeping **all previous turns**, it **summarizes older messages** into a concise format.
* The summary is combined with **recent messages** to give the model context.
* This allows the LLM to **maintain relevant context** while staying **token-efficient**.

**Use-cases:**

* Long-running customer support chats.
* Personal assistants that remember preferences over multiple interactions.
* Any application where **conversation history could exceed token limits**.

**2️⃣ How It Works**

**Step-by-step Flow**

1. **User input arrives** → stored temporarily in memory.
2. **AI generates a response** → added to memory.
3. **Memory checks** if a **token or turn threshold** is reached.
4. If threshold exceeded → **summarization step** triggers:
   * Old turns are **condensed into a rolling summary**.
   * Recent messages (last few turns) are **kept intact**.
5. **Next prompt to LLM** = summary + recent messages.

**Diagram:**

yaml

CopyEdit

Conversation:

Turn 1: User: Hi

Turn 2: AI: Hello!

Turn 3: User: I love pizza

Turn 4: AI: What's your favorite type?

Memory Summary (after summarization):

"User greeted AI and mentioned they love pizza."

Next prompt to LLM:

Memory Summary + Recent Messages → LLM Response

**Key Points**

* Keeps **essential context** while saving tokens.
* **Rolling summary** grows gradually with conversation.
* **Combines summary + recent messages** for prompt construction.
* Fine-grained details may be lost if not captured in the summary.

**3️⃣ Pros and Cons**

| **Pros** | **Cons** |
| --- | --- |
| Token-efficient for long conversations | Some fine-grained info may be lost |
| Scalable for long chats | Requires LLM to generate summary |
| Preserves essential context | Summarization quality depends on LLM prompt |
| Works well with multi-turn conversational agents | Slightly more complex setup than simple buffer memory |

**4️⃣ Code Example**

python

CopyEdit

# pip install langchain openai

from langchain.chat\_models import ChatOpenAI

from langchain.memory import ConversationSummaryMemory

from langchain.chains import ConversationChain

# 1️⃣ Initialize LLM

llm = ChatOpenAI(temperature=0, model\_name="gpt-3.5-turbo")

# 2️⃣ Initialize ConversationSummaryMemory

memory = ConversationSummaryMemory(

llm=llm,

max\_token\_limit=100, # summarization triggered when memory grows

return\_messages=True

)

# 3️⃣ Create ConversationChain

conversation = ConversationChain(

llm=llm,

memory=memory

)

# 4️⃣ Interact with the chain

messages = [

"Hi there!",

"I love pizza.",

"Can you suggest a good pizza recipe?",

"Remind me what I said before about my food preferences?"

]

for msg in messages:

response = conversation.predict(input=msg)

print(f"User: {msg}")

print(f"AI: {response}")

print("Memory Summary:", memory.load\_memory\_variables({})["summary"])

print("------")

**5️⃣ How Memory Evolves**

1. **Turn 1: "Hi there!"**  
   Memory summary: "User greeted AI."
2. **Turn 2: "I love pizza."**  
   Memory summary: "User greeted AI and mentioned they love pizza."
3. **Turn 3: "Can you suggest a good pizza recipe?"**  
   Memory summary: "User greeted AI, loves pizza, and asked for a pizza recipe."
4. **Turn 4: "Remind me what I said before about my food preferences?"**  
   LLM can answer using **summary + recent message**.  
   Memory summary updates to: "User greeted AI, loves pizza, asked for a pizza recipe, and inquired about food preferences."

✅ Notice: Older details are **compressed into a summary**, recent turns are kept intact for context.

**6️⃣ Summary**

* ConversationSummaryMemory = **rolling summarization for long-term memory**.
* **Token-efficient** and **scalable**.
* Works by **summarizing older turns** and combining them with **recent messages**.
* Ideal for **long conversations** where storing full history is impractical.
* Trade-off: **loss of some fine-grained details**.

I can next create a **visual comparison diagram of all three memory types**—ConversationBufferMemory, BufferWindowMemory, and ConversationSummaryMemory—showing **how they store, retrieve, and feed context to LLMs**. This makes it super easy to pick the right memory for your scenario.

Do you want me to do that next?

**You said:**

lets deep dive into - EntityMemory • Purpose: Stores specific entities or facts about the user. • Use-case: Personalization: remembering user name, preferences, location, etc. • Example: Remember “Rajeev likes coffee” across sessions.

**ChatGPT said:**

Absolutely! Let’s do a **deep dive into EntityMemory** in LangChain, focusing on its **purpose, internal workings, use cases, and practical code examples**.

**1️⃣ Purpose**

EntityMemory is designed to **remember specific entities or facts** about a user or conversation.

* Instead of storing every conversation turn, it **extracts and tracks key information** (entities).
* Enables **personalization** and **long-term recall** for conversational agents.

**Example entities:**

* User preferences: "Rajeev likes coffee"
* User name: "User's name is Rajeev"
* Location: "User lives in Bangalore"
* Favorite topics: "User prefers Java over Python"

**2️⃣ How It Works**

**Step-by-step Flow**

1. **User provides information** → LLM detects entities.  
   Example: "Hi, I’m Rajeev and I love coffee."
   * Entity detected: "Name: Rajeev", "Preference: coffee"
2. **EntityMemory updates** → stores detected entities in a **key-value format**.
3. **Subsequent prompts** → memory injects relevant entity information into the conversation.
   * E.g., "Rajeev likes coffee" is added to prompt automatically.
4. **Optional backend storage** → entities can persist across sessions using **Redis, SQL, or JSON**.

**Diagram:**

mathematica

CopyEdit

User: Hi, I’m Rajeev and I love coffee

|

v

EntityMemory: { "name": "Rajeev", "preference": "coffee" }

|

v

Next LLM Prompt: "User Rajeev likes coffee. User asks: What's a good coffee blend?"

**3️⃣ Pros and Cons**

| **Pros** | **Cons** |
| --- | --- |
| Enables personalized interactions | Requires entity extraction logic |
| Can persist information across sessions | May not capture context beyond entities |
| Works well for long-term memory | LLM quality affects entity recognition |
| Integrates with other memory types | Needs careful schema design for multiple entities |

**4️⃣ Use Cases**

* Personal assistant remembering user preferences (coffee, movies, music)
* Customer support bots recalling **account information** or **preferences**
* Multi-turn task automation requiring entity recall (e.g., booking flights, restaurants)
* Gaming bots remembering player stats or history

**5️⃣ Code Example**

python

CopyEdit

# pip install langchain openai

from langchain.chat\_models import ChatOpenAI

from langchain.memory import EntityMemory

from langchain.chains import ConversationChain

# 1️⃣ Initialize LLM

llm = ChatOpenAI(temperature=0, model\_name="gpt-3.5-turbo")

# 2️⃣ Initialize EntityMemory

memory = EntityMemory(llm=llm, return\_messages=True)

# 3️⃣ Create ConversationChain with entity memory

conversation = ConversationChain(

llm=llm,

memory=memory

)

# 4️⃣ Interact with the chain

messages = [

"Hi, I’m Rajeev and I love coffee.",

"Can you suggest a coffee blend for me?",

"Remind me what I like."

]

for msg in messages:

response = conversation.predict(input=msg)

print(f"User: {msg}")

print(f"AI: {response}")

print("Entity Memory Snapshot:", memory.load\_memory\_variables({})["entities"])

print("------")

**6️⃣ Example Memory Evolution**

1. **Turn 1: "Hi, I’m Rajeev and I love coffee."**

css

CopyEdit

EntityMemory: {"name": "Rajeev", "preference": "coffee"}

1. **Turn 2: "Can you suggest a coffee blend for me?"**
   * LLM sees entity: "User preference: coffee"
   * AI Response: "Since you like coffee, I suggest a medium roast Ethiopian blend."
2. **Turn 3: "Remind me what I like."**
   * Memory injects entities: "User likes coffee"
   * AI Response: "You like coffee, Rajeev!"

**7️⃣ Key Points**

* EntityMemory = **structured, entity-focused memory**.
* Ideal for **personalization** and **long-term recall**.
* Works well **with other memory types** like ConversationSummaryMemory to provide both **context and entities**.
* Can **persist entities** across sessions with a proper backend.

If you want, I can next deep dive into **VectorStoreRetrieverMemory**, which is the **long-term memory using embeddings for retrieval**, enabling **infinite knowledge recall** across conversations.

Do you want me to do that next?

**You said:**

yes please -I can next deep dive into VectorStoreRetrieverMemory, which is the long-term memory using embeddings for retrieval, enabling infinite knowledge recall across conversations.

**ChatGPT said:**

Perfect! Let’s deep dive into **VectorStoreRetrieverMemory**, LangChain’s **long-term memory solution** that uses embeddings for retrieval. I’ll explain **purpose, working mechanism, use cases, code example, and best practices**.

**1️⃣ Purpose**

* VectorStoreRetrieverMemory is designed for **long-term memory** across sessions.
* Instead of storing conversations as plain text, it **stores embeddings** of important messages or facts.
* Later, **relevant information can be retrieved** via similarity search.
* Ideal for **large-scale knowledge recall** or **persistent memory**.

**Use cases:**

* Chatbots with **infinite memory** of past conversations.
* Customer support assistants recalling **previous tickets or user preferences**.
* Personal assistants remembering facts across months or years.
* Knowledge-base retrieval for FAQs, documents, or research papers.

**2️⃣ How It Works Internally**

1. **User input / AI output** → converted to **vector embeddings** using an embedding model.
2. **Embeddings stored** in a **vector database** (e.g., FAISS, Chroma, Pinecone, RedisVector).
3. **Next user query** → converted to embedding → vector store **retrieves relevant past messages**.
4. Retrieved messages are **injected into LLM prompt** as context for response.

**Diagram:**

pgsql

CopyEdit

User: I love coffee.

|

v

Embedding Model -> Vector Representation

|

v

Vector Store: store vector with metadata (text, timestamp, user info)

|

v

User: Suggest a coffee blend.

|

v

Query embedding -> retrieve similar vectors from vector store

|

v

LLM Prompt = Retrieved facts + Recent input -> Generate response

**3️⃣ Advantages vs Other Memories**

| **Feature** | **ConversationBufferMemory** | **BufferWindowMemory** | **ConversationSummaryMemory** | **VectorStoreRetrieverMemory** |
| --- | --- | --- | --- | --- |
| Full conversation history | ✅ | ❌ last k | ❌ summarized | ❌ not all text |
| Token-efficient | ❌ | ✅ | ✅ | ✅ (retrieves relevant facts only) |
| Long-term recall | ❌ | ❌ | Partial | ✅ infinite knowledge |
| Structured memory | ❌ | ❌ | ❌ | ✅ embeddings + metadata |
| Use for personalization | Partial | Partial | Partial | ✅ yes, via retrieved facts |

**4️⃣ Code Example**

python

CopyEdit

# pip install langchain openai chromadb

from langchain.chat\_models import ChatOpenAI

from langchain.vectorstores import Chroma

from langchain.embeddings import OpenAIEmbeddings

from langchain.memory import VectorStoreRetrieverMemory

from langchain.chains import ConversationChain

# 1️⃣ Initialize LLM

llm = ChatOpenAI(temperature=0, model\_name="gpt-3.5-turbo")

# 2️⃣ Setup vector store (Chroma in this example)

embeddings = OpenAIEmbeddings()

vector\_store = Chroma(persist\_directory="./vectordb", embedding\_function=embeddings)

# 3️⃣ Setup VectorStoreRetrieverMemory

memory = VectorStoreRetrieverMemory(vectorstore=vector\_store.as\_retriever(), return\_messages=True)

# 4️⃣ Create ConversationChain with long-term memory

conversation = ConversationChain(

llm=llm,

memory=memory

)

# 5️⃣ Example interaction

messages = [

"Hi, I’m Rajeev and I love coffee.",

"Suggest a coffee blend for me.",

"Remind me what I like about beverages."

]

for msg in messages:

response = conversation.predict(input=msg)

print(f"User: {msg}")

print(f"AI: {response}")

print("Memory Snapshot:", memory.load\_memory\_variables({})["history"])

print("------")

**5️⃣ How Memory Evolves**

1. **Turn 1:** "Hi, I’m Rajeev and I love coffee."
   * Stored as an embedding vector in vector store.
2. **Turn 2:** "Suggest a coffee blend for me."
   * LLM retrieves relevant vector(s) → sees "User likes coffee"
   * AI responds with personalized suggestion.
3. **Turn 3:** "Remind me what I like about beverages."
   * Vector store retrieval finds "User likes coffee"
   * AI response uses long-term memory to answer accurately.

✅ This works **even if the session is restarted**—the vector database persists knowledge.

**6️⃣ Best Practices**

1. **Choose the right embedding model** – e.g., text-embedding-3-small or text-embedding-3-large.
2. **Use metadata** to store context like timestamp, user ID, session ID.
3. **Persist vector store** to disk or cloud (Chroma, Pinecone, RedisVector) for long-term recall.
4. **Limit retrieval count** – fetch top-k relevant vectors to avoid token overflow.
5. Combine with **ConversationSummaryMemory** for recent turns → summary + long-term facts.

**7️⃣ Key Points**

* VectorStoreRetrieverMemory = **retrieval-based long-term memory**.
* Uses **embeddings + similarity search** instead of storing all text.
* **Persistent across sessions**, scalable for large knowledge bases.
* Works well for **personalization, long-term context, and knowledge retrieval**.

If you want, I can next **draw a full visual diagram showing all memory types in LangChain**—ConversationBufferMemory, BufferWindowMemory, ConversationSummaryMemory, EntityMemory, and VectorStoreRetrieverMemory—with **how they store and retrieve information for the LLM**.

Do you want me to do that?

**You said:**

lets deep dive into with details explanation - VectorStoreRetrieverMemory • Purpose: Enables long-term memory by storing facts in a vector database. • How it works: o Stores embeddings of important conversation pieces or knowledge. o Later, relevant pieces can be retrieved via similarity search. • Backends: Chroma, FAISS, Pinecone, RedisVector, etc. • Pros: Infinite long-term memory, retrieval-based, scalable. • Cons: Slightly more complex to set up.

**ChatGPT said:**

Absolutely! Let’s do a **full deep dive into VectorStoreRetrieverMemory** with a detailed explanation of **purpose, working, architecture, backends, pros/cons, and practical code example**.

**1️⃣ Purpose**

VectorStoreRetrieverMemory is designed to give your conversational AI **long-term memory**.

* Instead of storing every conversation turn, it stores **important facts or messages as embeddings** in a **vector database**.
* Later, when the user asks something, the memory **retrieves the most relevant information** using **similarity search**.
* This approach enables a bot to **remember facts across sessions** or even months/years.

**Use-cases:**

* Personalized assistants remembering user preferences (Rajeev likes coffee).
* Customer support bots recalling previous tickets.
* Knowledge base retrieval for FAQs, product info, or research.
* Multi-session task automation.

**2️⃣ How It Works**

**Step-by-step**

1. **User input or AI output** → converted to **vector embeddings** using an embedding model.
2. **Embeddings stored in a vector database** with optional metadata (timestamp, user id, source).
3. **User query** → converted to embedding → **similarity search** in vector store retrieves relevant facts.
4. Retrieved facts are **injected into the LLM prompt** for context.
5. LLM generates response based on retrieved long-term memory + recent messages.

**Visual Flow:**

pgsql

CopyEdit

User Input: "Hi, I love coffee."

|

v

Embedding Model -> Vector Representation

|

v

Vector Store (Chroma/FAISS/Pinecone/RedisVector) <- stored with metadata

|

v

User Query: "Suggest a coffee blend."

|

v

Query Embedding -> Retrieve top-k similar vectors

|

v

LLM Prompt = Retrieved Facts + Recent Input -> Response

**3️⃣ Backends**

* **Chroma** – local vector DB, easy to set up, file-based persistence.
* **FAISS** – highly efficient, works locally, supports large datasets.
* **Pinecone** – cloud vector DB, scalable, fully managed.
* **RedisVector** – Redis module for vector similarity search, good for real-time apps.

Each backend can store **embeddings** and **metadata** for retrieval.

**4️⃣ Pros and Cons**

| **Pros** | **Cons** |
| --- | --- |
| Enables infinite long-term memory | More complex to set up |
| Retrieval-based → only fetch relevant info | Requires vector DB + embedding model |
| Scalable for large datasets | Extra latency for similarity search |
| Persistent across sessions | LLM prompt still needs recent context for best results |
| Can combine with other memories (summary/entity) | Needs tuning for retrieval count / relevance |

**5️⃣ Code Example**

python

CopyEdit

# pip install langchain openai chromadb

from langchain.chat\_models import ChatOpenAI

from langchain.vectorstores import Chroma

from langchain.embeddings import OpenAIEmbeddings

from langchain.memory import VectorStoreRetrieverMemory

from langchain.chains import ConversationChain

# 1️⃣ Initialize LLM

llm = ChatOpenAI(temperature=0, model\_name="gpt-3.5-turbo")

# 2️⃣ Setup embedding model and vector store (Chroma)

embeddings = OpenAIEmbeddings()

vector\_store = Chroma(persist\_directory="./vectordb", embedding\_function=embeddings)

# 3️⃣ Setup VectorStoreRetrieverMemory

memory = VectorStoreRetrieverMemory(

vectorstore=vector\_store.as\_retriever(search\_kwargs={"k": 3}), # retrieve top 3 relevant facts

return\_messages=True

)

# 4️⃣ Create ConversationChain

conversation = ConversationChain(

llm=llm,

memory=memory

)

# 5️⃣ Interact with the chain

messages = [

"Hi, I’m Rajeev and I love coffee.",

"Suggest a coffee blend for me.",

"Remind me what I like about beverages."

]

for msg in messages:

response = conversation.predict(input=msg)

print(f"User: {msg}")

print(f"AI: {response}")

print("Memory Snapshot:", memory.load\_memory\_variables({})["history"])

print("------")

**6️⃣ How Memory Evolves**

1. **Turn 1:** "Hi, I’m Rajeev and I love coffee."
   * Stored as embedding in vector store.
2. **Turn 2:** "Suggest a coffee blend for me."
   * Memory retrieves "User likes coffee"
   * AI responds with personalized suggestion.
3. **Turn 3:** "Remind me what I like about beverages."
   * Memory retrieves "User likes coffee"
   * AI responds accurately using long-term memory.

✅ Works **even across sessions** if the vector database is persisted.

**7️⃣ Best Practices**

1. Use **high-quality embeddings** for better retrieval.
2. Persist vector store for **long-term memory**.
3. Include **metadata** for filtering by user, session, or topic.
4. Combine with **ConversationSummaryMemory** to provide recent context along with retrieved facts.
5. Tune **k** (number of retrieved vectors) to balance relevance and token limits.

**8️⃣ Key Takeaways**

* VectorStoreRetrieverMemory = **retrieval-based, long-term memory**.
* Stores knowledge as embeddings → retrieves via **similarity search**.
* Enables **infinite memory**, **personalization**, and **persistent context**.
* Scalable, but slightly more complex than buffer or summary memory.

I can next create a **comprehensive visual comparison of all LangChain memory types** (ConversationBufferMemory, BufferWindowMemory, ConversationSummaryMemory, EntityMemory, VectorStoreRetrieverMemory) showing **how each stores and retrieves information**.

Do you want me to create that?

**You said:**

can you give full stack code example encluding ui with most used memory react ui

**ChatGPT said:**

Absolutely! Let’s build a **full-stack LangChain conversational app** using **React UI** on the front-end and **FastAPI** as the backend. We'll integrate **ConversationBufferMemory** as the memory type (most commonly used for chat UIs).

This example includes:

* **Backend:** FastAPI + LangChain + OpenAI
* **Frontend:** React (with functional components)
* **Memory:** ConversationBufferMemory for full chat history
* **Streaming responses:** optional, to simulate real-time typing

**1️⃣ Backend (FastAPI + LangChain)**

python

CopyEdit

# backend/main.py

from fastapi import FastAPI

from fastapi.middleware.cors import CORSMiddleware

from pydantic import BaseModel

from langchain.chat\_models import ChatOpenAI

from langchain.memory import ConversationBufferMemory

from langchain.chains import ConversationChain

app = FastAPI()

# Allow CORS from frontend

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

# LangChain setup

llm = ChatOpenAI(temperature=0)

memory = ConversationBufferMemory(return\_messages=True)

conversation = ConversationChain(llm=llm, memory=memory)

# Request model

class Message(BaseModel):

text: str

@app.post("/chat")

async def chat(message: Message):

response = conversation.predict(input=message.text)

# Return full memory for UI display

return {

"response": response,

"memory": memory.load\_memory\_variables({})["chat\_history"]

}

**2️⃣ Frontend (React + Axios)**

javascript

CopyEdit

// frontend/src/App.js

import React, { useState } from "react";

import axios from "axios";

import "./App.css";

function App() {

const [messages, setMessages] = useState([]);

const [input, setInput] = useState("");

const handleSend = async () => {

if (!input) return;

const userMessage = { role: "user", content: input };

setMessages([...messages, userMessage]);

try {

const res = await axios.post("http://localhost:8000/chat", { text: input });

const aiMessage = { role: "assistant", content: res.data.response };

setMessages([...messages, userMessage, aiMessage]);

} catch (error) {

console.error("Error:", error);

}

setInput("");

};

return (

<div className="app">

<h1>LangChain Chat</h1>

<div className="chat-window">

{messages.map((msg, idx) => (

<div

key={idx}

className={msg.role === "user" ? "user-message" : "ai-message"}

>

{msg.content}

</div>

))}

</div>

<div className="chat-input">

<input

type="text"

value={input}

onChange={(e) => setInput(e.target.value)}

onKeyPress={(e) => e.key === "Enter" && handleSend()}

placeholder="Type your message..."

/>

<button onClick={handleSend}>Send</button>

</div>

</div>

);

}

export default App;

**3️⃣ Frontend CSS (optional)**

css

CopyEdit

/\* frontend/src/App.css \*/

.app {

max-width: 600px;

margin: 0 auto;

font-family: sans-serif;

}

.chat-window {

border: 1px solid #ccc;

border-radius: 8px;

height: 400px;

padding: 10px;

overflow-y: auto;

margin-bottom: 10px;

background-color: #f9f9f9;

}

.user-message {

text-align: right;

background-color: #dcf8c6;

padding: 8px;

margin: 5px 0;

border-radius: 8px;

}

.ai-message {

text-align: left;

background-color: #fff;

padding: 8px;

margin: 5px 0;

border-radius: 8px;

border: 1px solid #ddd;

}

.chat-input {

display: flex;

gap: 10px;

}

.chat-input input {

flex: 1;

padding: 8px;

border-radius: 8px;

border: 1px solid #ccc;

}

.chat-input button {

padding: 8px 16px;

border-radius: 8px;

border: none;

background-color: #007bff;

color: white;

cursor: pointer;

}

**4️⃣ Run Instructions**

**Backend**

bash

CopyEdit

cd backend

pip install fastapi uvicorn langchain openai

uvicorn main:app --reload

**Frontend**

bash

CopyEdit

cd frontend

npm install axios

npm start

* Backend runs at http://localhost:8000
* Frontend runs at http://localhost:3000
* Type messages in the UI and see responses with **full conversation memory**.

**5️⃣ How Memory Works in This Example**

* ConversationBufferMemory stores **all user + AI messages**.
* Every request updates memory → included in the next LLM prompt.
* Frontend can access **full memory** via API → display chat history.
* Ideal for **short to medium-length conversations**.

✅ This full-stack example uses the **most common memory type** for interactive chat UIs.

I can next show an **enhanced version with ConversationSummaryMemory or VectorStoreRetrieverMemory**, so your UI can handle **long-term memory and personalization**, which is more scalable for production chat apps.