Advanced Generative AI: Building LLM Applications

LangChain for LLM Application Development: Part 2

Quick Recap

Lorem ipsum dolor sit amet, adipiscing elit, sed do incididunt ut labore magna aliqua.?

- The significance of the interaction between the user, the model, and the output within the LangChain framework. How does this interaction contribute to the overall effectiveness and usability of LangChain-based systems? Provide examples to illustrate your points.
- The functionality and significance of the CharacterTextSplitter module within the LangChain framework. How does this module contribute to the efficiency and effectiveness of text processing in LangChain? Provide examples to illustrate your points.

Engage and Think

ABC Corp, a company that makes movies and shows, wants to make their stuff more interesting and exciting. They care about making sure everyone feels included, being creative, and making sure what they make is fair and okay. ABC wants to be the best in entertainment.

How can they use GenAI technology to make interesting stories, fun things to do, and different kinds of people in their shows, and make sure everything is fair and right, so everyone likes it more?

Learning Objectives

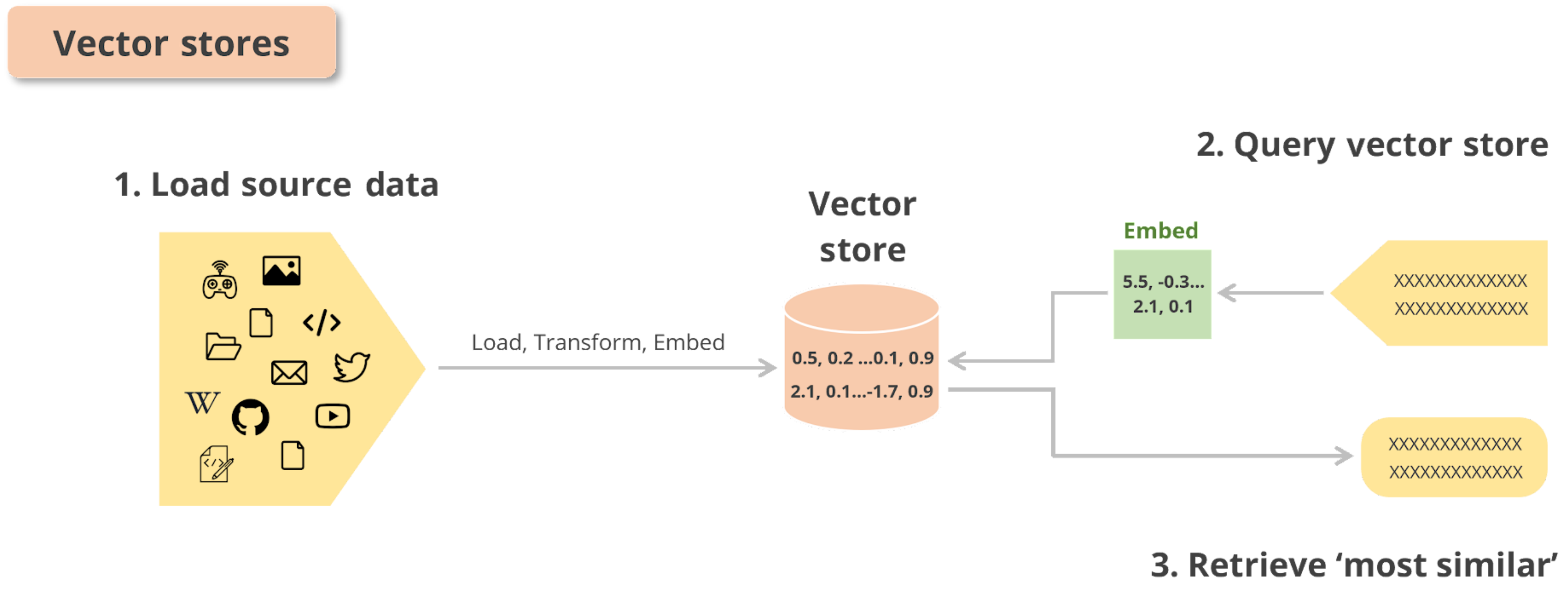
By the end of this lesson, you will be able to:

- Apply VectorStores for storing and retrieving diverse embeddings efficiently
- Explain the fundamental concepts and principles behind LangChain Retriever
- Develop a comprehensive understanding of LangChain Chains and their role as the backbone of conversational interfaces
- Infer the diverse functionalities of LangChain Agents and their role in leveraging external tools and resources to perform specific actions within LangChain

Introduction to VectorStore

Introduction to VectorStore

In LLM, the VectorStore acts as a storage space for embeddings. It helps quickly find and work with data representations, improving the model's performance.



Source: https://python.langchain.com/assets/images/vector_stores-125d1675d58cfb46ce9054c9019fea72.jpg

Introduction to VectorStore

The VectorStore in LangChain is an abstract class that represents a store of vectors.

It is utilized for adding vectors and documents to the store, removing items from the store, and conducting searches within the store.

It takes care of storing embedded data and performing vector searches for you.

There are different implementations of VectorStore based on different technologies such as ChromaDB, Faiss, pgVector database, and pinecone.

VectorStore: FAISS

FAISS (Facebook AI Similarity Search) is a high-performance library built for efficient similarity search at scale. The following are the key features:

Ease of integration	Designed to connects with LangChain for robust search applications
Scalability	Designed to handle large datasets with high-dimensional vectors
Speed	Optimized for rapid nearest neighbor searches

Use case: Ideal for production environments requiring large-scale data handling and low-latency retrieval.

VectorStore: Pinecone

Pinecone is a fully managed, cloud-native vector database that ensures high availability and global indexing. The following are the key features:

Ease of integration	Offers seamless connectivity with LangChain for enterprise applications
Scalability	Provides robust, cloud-managed performance for large-scale deployments
Global reach	Delivers low-latency, distributed vector search across geographies

Use case: Best suited for applications that demand high performance, scalability, and minimal operational overhead in a cloud environment.

VectorStore: ChromaDB

ChromaDB is a lightweight, user-friendly vector store designed for local storage of embeddings. The following are the key features:

Ease of use	Simple setup and integration with LangChain.
Local deployment	Ideal for rapid prototyping and smaller-scale projects.
Cost-effective	Negligible cloud costs for local embedding management.

Use case: Perfect for moderate data volumes and quick similarity searches in development environments.

How VectorStore Works?

Here's how the process of storing and searching over unstructured data works in the context of LangChain's VectorStore:

Embedding data

Embeddings are used to transform unstructured data into structured formats. Storing these resultant vectors is used for handling such data efficiently

Storing vectors

The VectorStore in LangChain takes care of storing the embedded data.

Embedding query

At the time of querying, the unstructured query is transformed into an embedded form.

How VectorStore Works?

Retrieving vector

The VectorStore retrieves the embedding vectors that are similar to the embedded query.

Vector search

VectorStore conducts a vector search, finding the most similar vectors to a given one.

Demo: Loaders, text splitters, embeddings, VectorStores

Duration: 5 minutes

By working with these files, you will gain hands-on experience in loading documents, splitting them into manageable chunks, embedding them into a numerical space, and storing these embeddings for efficient similarity searches. These are key steps in many natural language processing pipelines, including document classification, information retrieval, and semantic search.

Data: state_of_union.txt, michael_resume.pdf

Note

Please download the solution document from the Reference Material Section and follow the Jupyter Notebook for step-by-step execution.

Quick Check

What is NOT a function of the VectorStore in LangChain?

A. It represents a store of vectors.

B. It's used to combine vectors and documents, remove them from the collection, and look for things in the collection.

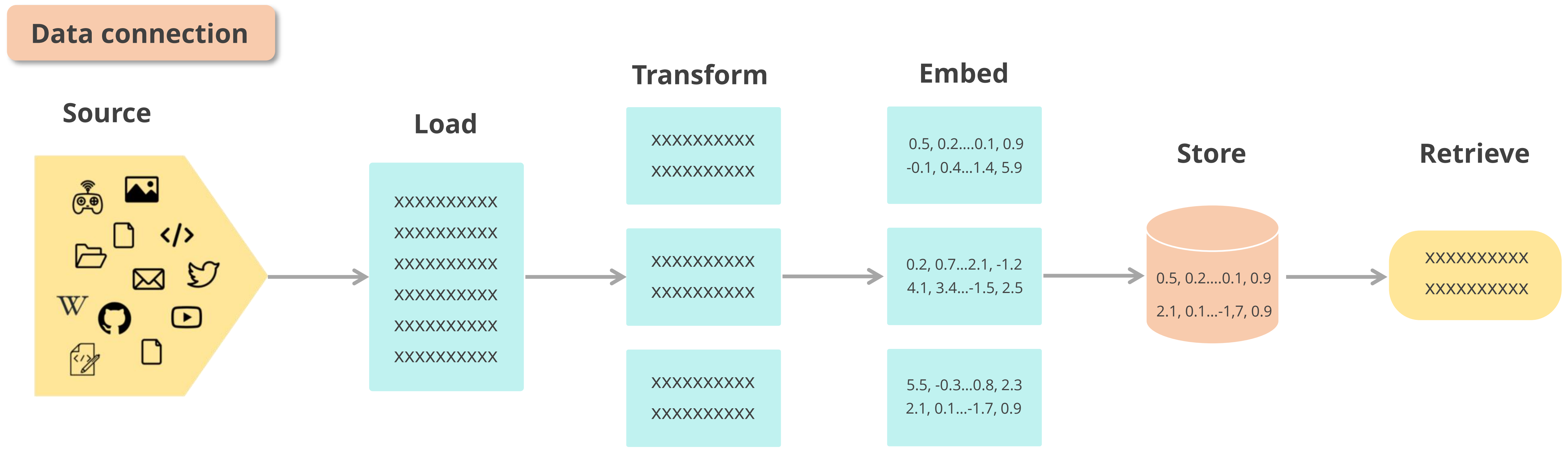
C. It takes care of storing embedded data and performing vector searches.

D. It is used to perform syntax checking and error handling in the code.

LangChain Retriever

LangChain Retriever

LangChain retrievers are tools in LangChain that fetch information from external sources to enhance the language model's knowledge and capabilities.



Source: https://python.langchain.com/assets/images/data_connection-95ff2033a8faa5f3ba41376c0f6dd32a.jpg

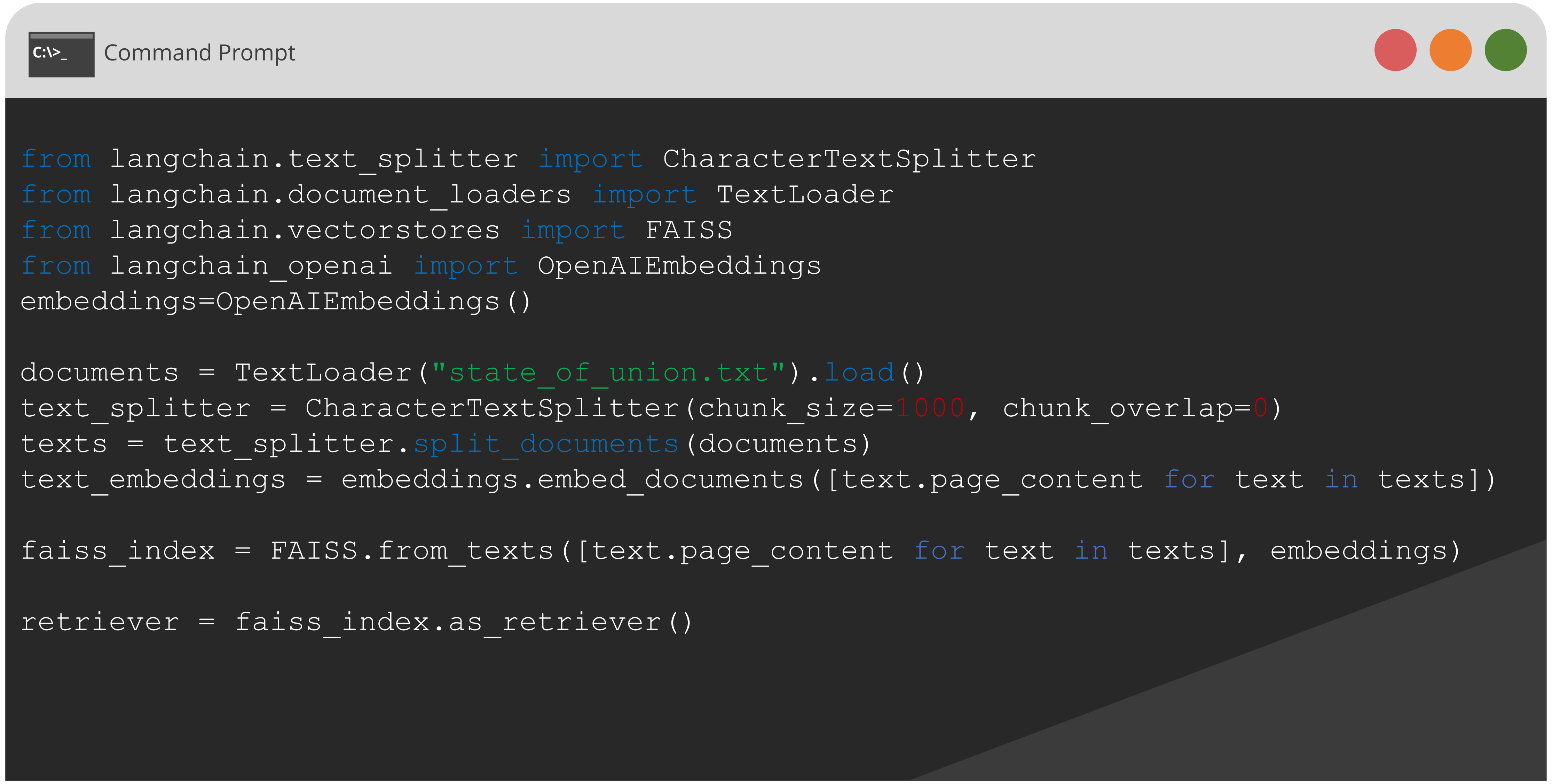
LangChain Retriever

LangChain retrievers fetch documents with unstructured queries, offering versatility beyond vector stores, which primarily store and retrieve.

While vector stores can form the backbone of a retriever, there are other types of retrievers as well.

Retrievers take a string query as input and return a list of document objects as output.

LangChain Retriever: Example



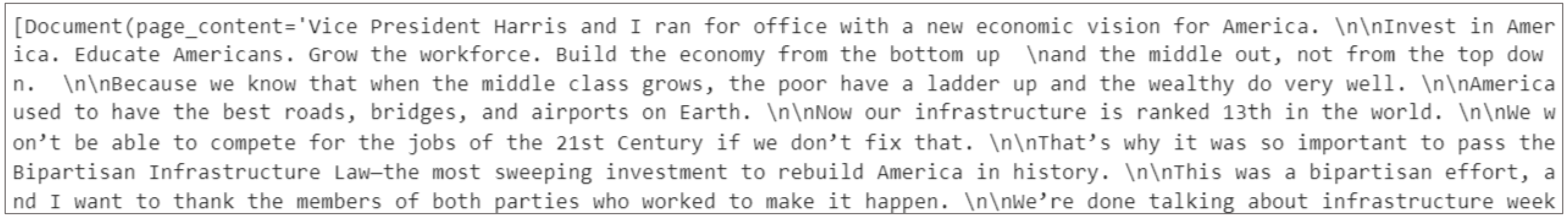
```
from langchain.text_splitter import CharacterTextSplitter
from langchain.document_loaders import TextLoader
from langchain.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
embeddings=OpenAIEmbeddings()

documents = TextLoader("state_of_union.txt").load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
text_embeddings = embeddings.embed_documents([text.page_content for text in texts])

faiss_index = FAISS.from_texts([text.page_content for text in texts], embeddings)

retriever = faiss_index.as_retriever()
```

LangChain Retriever: Example

Command Prompt

```
docs = db.get_relevant_documents(
    "What did the president say about Ketanji Brown Jackson"
)
print(docs)
```

Output:



```
[Document(page_content='Vice President Harris and I ran for office with a new economic vision for America. \n\nInvest in Amer
ica. Educate Americans. Grow the workforce. Build the economy from the bottom up  \nand the middle out, not from the top dow
n.  \n\nBecause we know that when the middle class grows, the poor have a ladder up and the wealthy do very well. \n\nAmerica
used to have the best roads, bridges, and airports on Earth. \n\nNow our infrastructure is ranked 13th in the world. \n\nWe w
on’t be able to compete for the jobs of the 21st Century if we don’t fix that. \n\nThat’s why it was so important to pass the
Bipartisan Infrastructure Law—the most sweeping investment to rebuild America in history. \n\nThis was a bipartisan effort, a
nd I want to thank the members of both parties who worked to make it happen. \n\nWe’re done talking about infrastructure week
```

Quick Check

What is a distinguishing feature of LangChain retrievers in comparison to vector stores?

A. LangChain retrievers use structured queries for document retrieval.

B. LangChain retrievers primarily store and retrieve vectors.

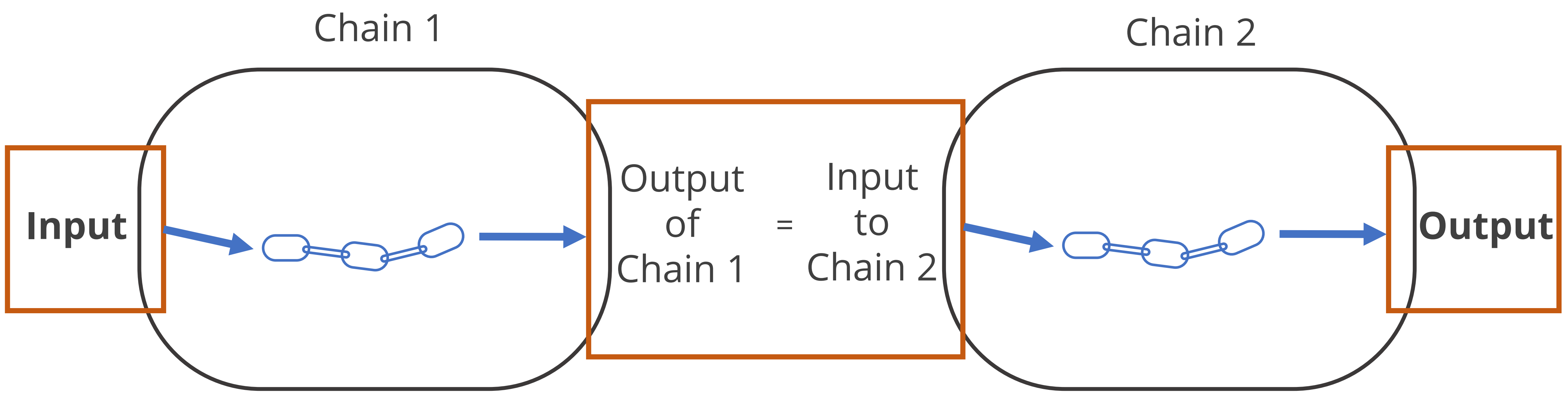
C. LangChain retrievers fetch documents with unstructured queries.

D. LangChain retrievers are specifically designed for storing large datasets.

LangChain Chains

LangChain Chains

Using an LLM alone is suitable for basic tasks, but complex applications may require chaining LLMs or other components.



Source: https://av-eks-lekhak.s3.amazonaws.com/media/__sized__/article_images/5_83DdPYD-thumbnail_webp-600x300.webp

LangChain Chains

LangChain offers two high-level frameworks for the chaining of components:

Chain interface

This is the legacy approach for chaining components.

LangChain expression language (LCEL)

This is the updated approach recommended for chain composition in new applications.

Note: While LCEL is suggested for new applications, LangChain still supports built-in chains, with both frameworks documented for reference.

Foundational LangChain Chain

In applications of LLMs, chaining typically involves integrating a prompt template with the LLM and, if necessary, employing an output parser for further refinement.

An LLMChain, commonly used in LangChain, boosts language model functionality and is integrated into other chains and agents.

An LLMChain consists of a PromptTemplate and a language model(which could be either an LLM or a chat model).

The LLMChain formats the PromptTemplate using input and memory key values, then passes it to the LLM for output.

Simple LLMChain: Example

Command Prompt

```
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

# Change the prompt template
prompt_template = "What would be an innovative name for a startup focusing on
{product}?"

llm = OpenAI(temperature=0)
llm_chain = LLMChain(llm=llm,
prompt=PromptTemplate.from_template(prompt_template))
Output=llm_chain("sustainable energy solutions")
print(output)
```

Output:

```
{'product': 'sustainable energy solutions', 'text': '\n\nEcoPowerTech'}
```

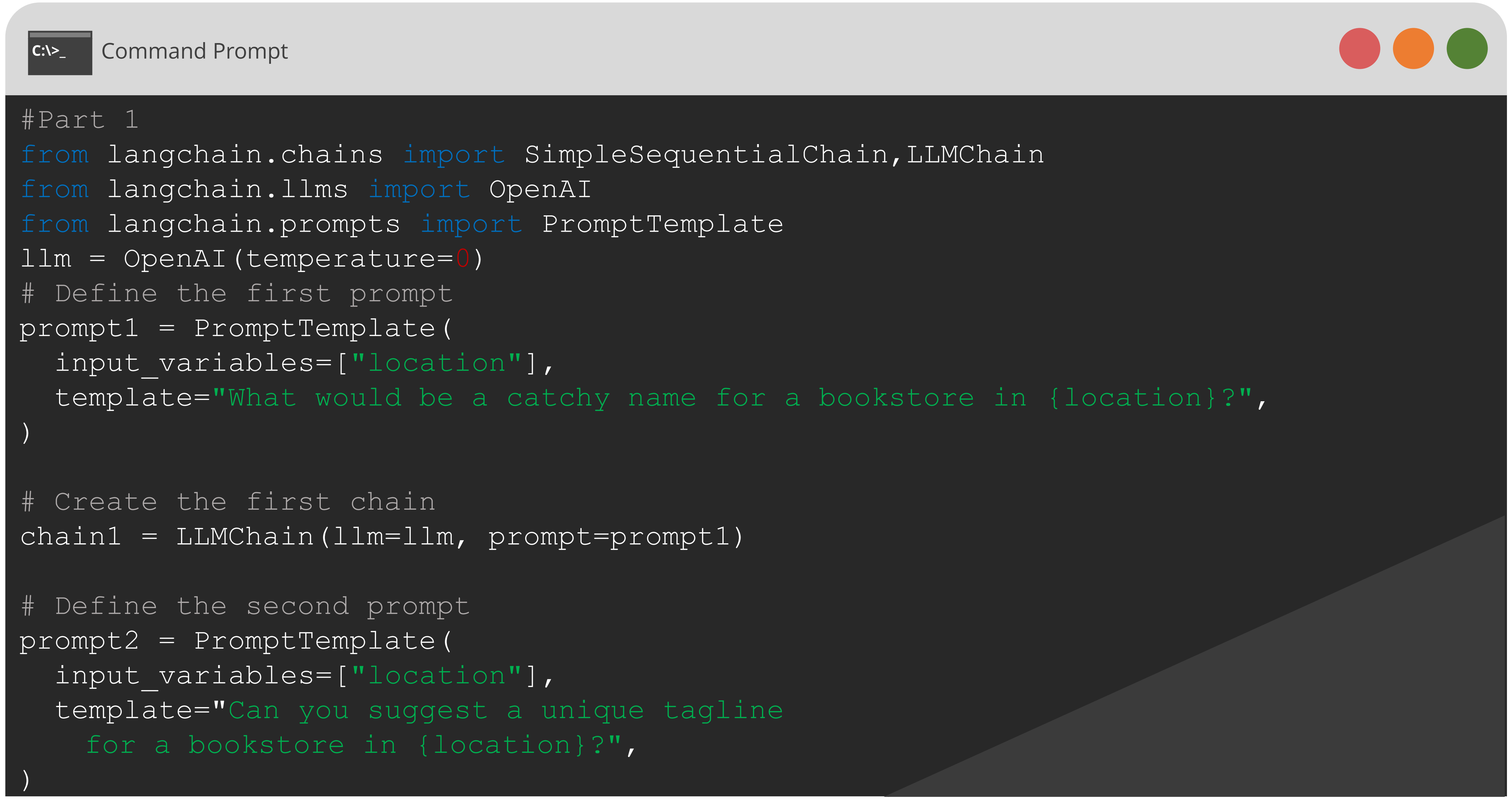
Sequential Chain in LangChain

A sequential chain in LangChain allows you to execute multiple chains in a sequence, much like a well-orchestrated symphony.

The SimpleSequentialChain object from the langchain.chains module enables you to create such a sequential chain.

To execute a SimpleSequentialChain, list the desired chains and use the **run()** method.

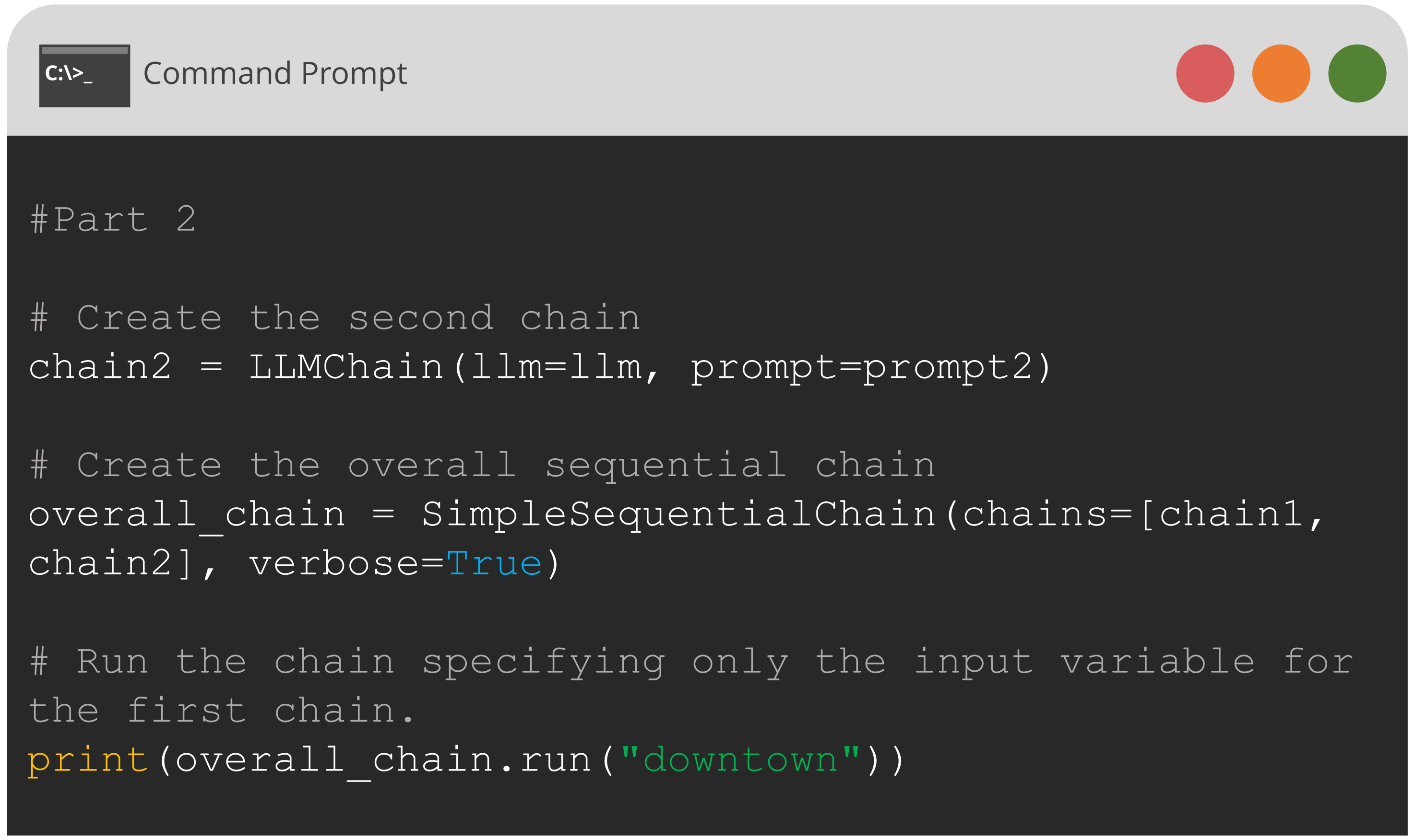
Sequential Chain: Example



Command Prompt

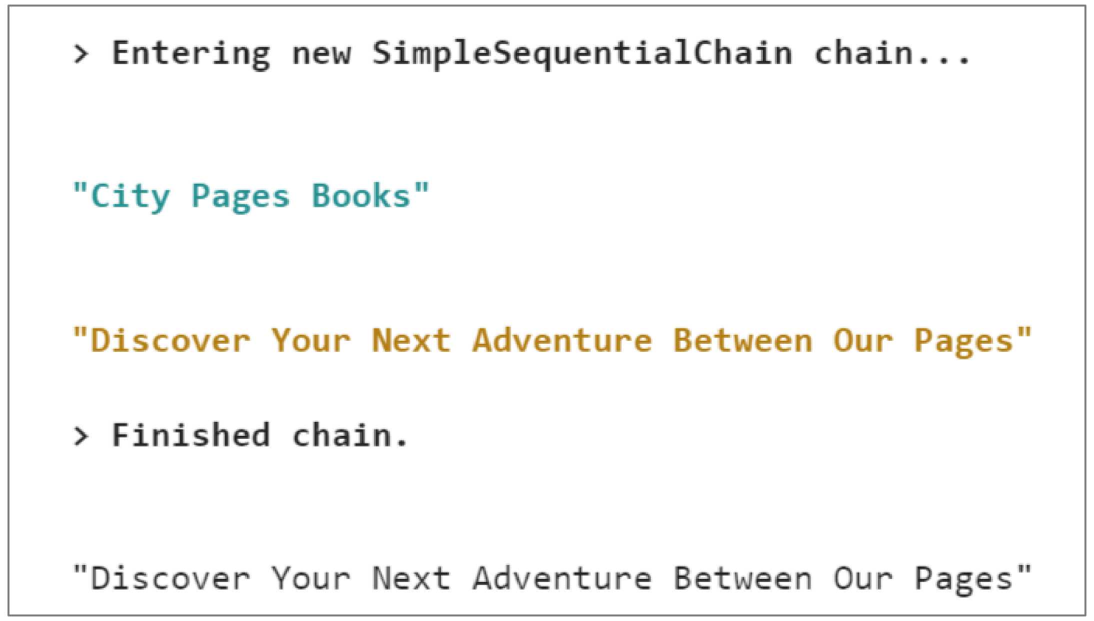
```
#Part 1
from langchain.chains import SimpleSequentialChain,LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
llm = OpenAI(temperature=0)
# Define the first prompt
prompt1 = PromptTemplate(
  input_variables=["location"],
  template="What would be a catchy name for a bookstore in {location}?",
)

# Create the first chain
chain1 = LLMChain(llm=llm, prompt=prompt1)

# Define the second prompt
prompt2 = PromptTemplate(
  input_variables=["location"],
  template="Can you suggest a unique tagline
    for a bookstore in {location}?",
)
```

Sequential Chain: Example



```
#Part 2

# Create the second chain
chain2 = LLMChain(llm=llm, prompt=prompt2)

# Create the overall sequential chain
overall_chain = SimpleSequentialChain(chains=[chain1,
chain2], verbose=True)

# Run the chain specifying only the input variable for
the first chain.
print(overall_chain.run("downtown"))
```

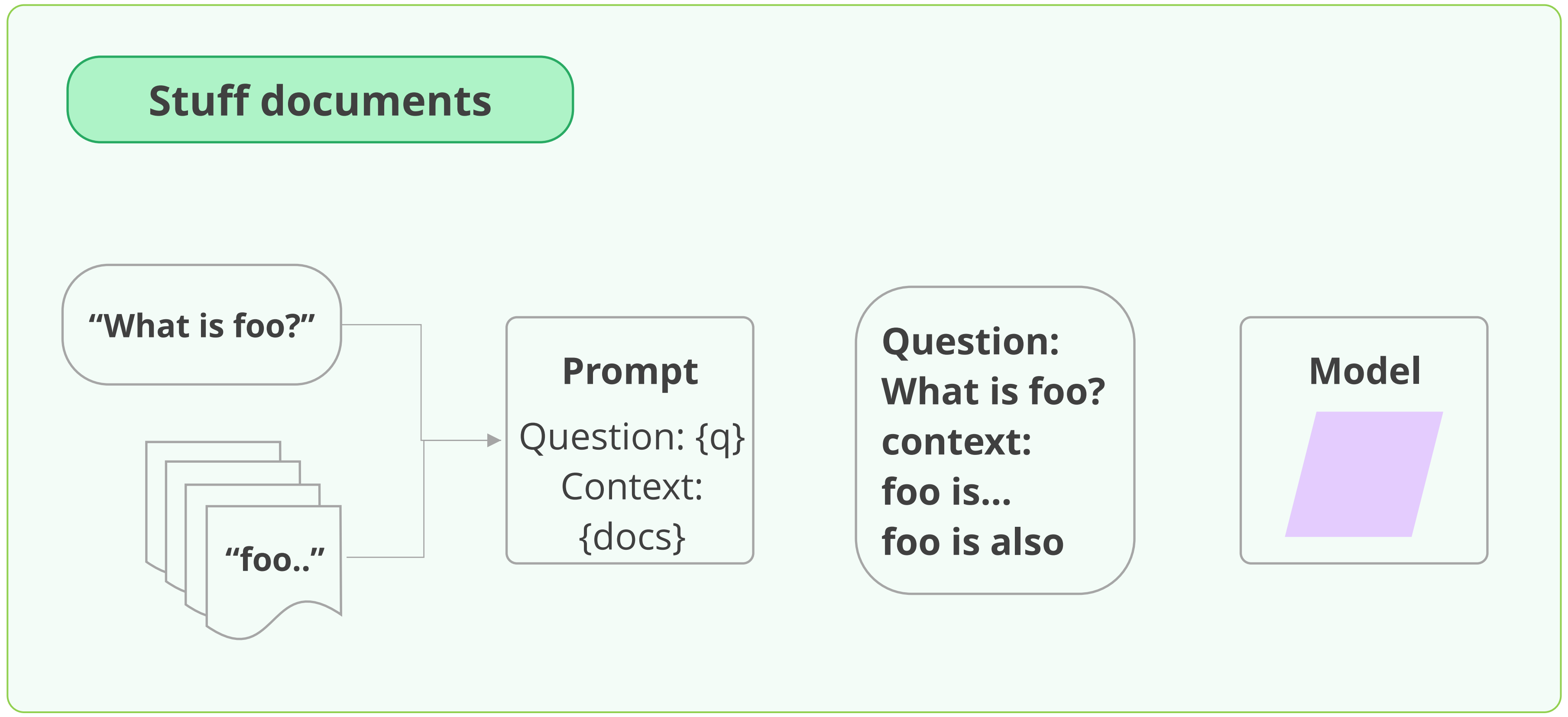
Output:



```
> Entering new SimpleSequentialChain chain...


"City Pages Books"


"Discover Your Next Adventure Between Our Pages"

> Finished chain.


"Discover Your Next Adventure Between Our Pages"
```

Stuff Chain

The stuff chain in LLM generates creative content based on specific prompts, assisting in brainstorming and idea generation with language models.



Source: https://python.langchain.com/assets/images/stuff-f51054532840dfb3cbdf86670b48ac7f.jpg

Stuff Chain

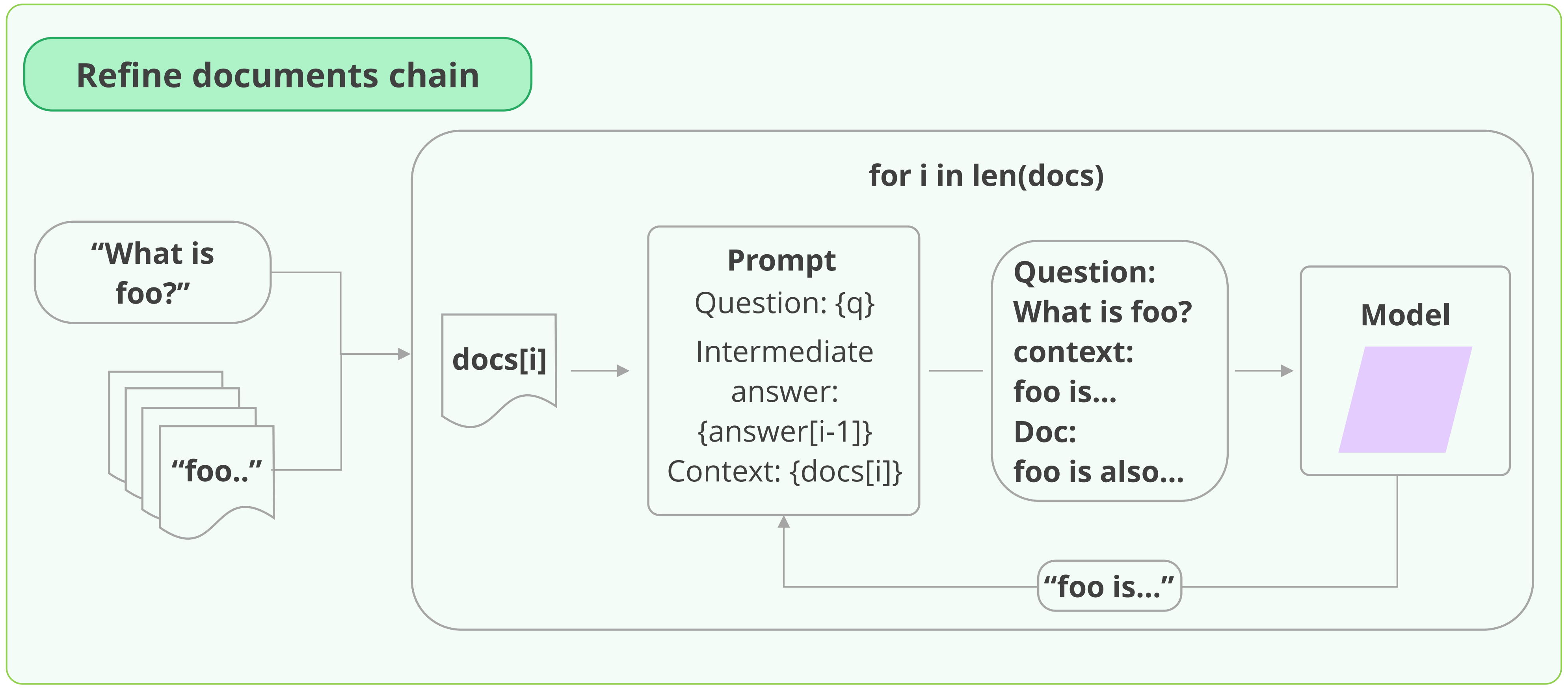
Also known as the **Stuff documents chain**, it's a simple way to generate documents by filling or stuffing them with content.

It operates by taking a list of documents, incorporating them all into a prompt, and then passing that prompt to an LLM.

This chain is particularly effective for applications where the documents are relatively small and only a few are typically passed in for most calls.

Refine Chain

The refine chain in LLM improves outputs by further processing and polishing language model-generated content, enhancing its quality and coherence.



Source: https://python.langchain.com/assets/images/refine-42297d920f42e9988a3e53982f8e83d6.jpg

Refine Chain

The **refine documents** chain acts like a careful craftsman, improving its output by revisiting input documents and updating answers iteratively.

Like a craftsman focusing on one piece, the refine chain sends one document at a time to the LLM, ideal for tasks needing analysis of numerous documents.

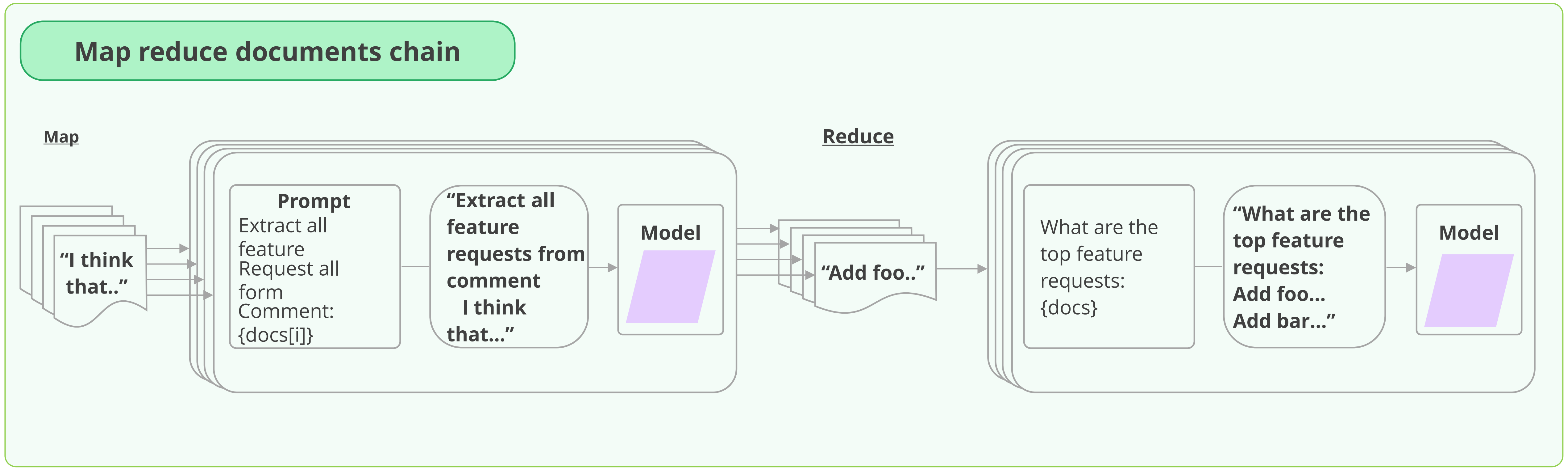
Refine Chain

However, like any craftsman, it has its trade-offs. This chain will make far more LLM calls than, for example, the Stuff documents chain.

Refining tasks with lots of document references or requiring detailed information from many documents can be tough for the refine chain.

Map Reduce Chain

The map reduce chain in LLM simplifies complex tasks by dividing them into smaller pieces, processing each independently, and then combining the results.


Map reduce documents chain
Map
Reduce
"I think that.."
Prompt
Extract all feature Request all form Comment: {docs[i]}
"Extract all feature requests from comment I think that..."
Model
"Add foo.."
What are the top feature requests: {docs}
"What are the top feature requests: Add foo... Add bar..."
Model

Map Reduce Chain

Imagine the **map reduce** documents chain as a skilled artisan working on a complex piece of artwork.

First, like an artist creating each part, the **map reduce** chain uses an LLM chain on each document separately. This is the **map step**.

Map Reduce Chain

Next, in the reduce step, much like an artisan assembling all the individual pieces to form a single masterpiece, all the newly crafted documents are passed to a separate combined documents chain to create a single, cohesive output.

The **map reduce chain** handles large artwork by compressing mapped documents, ensuring they fit in the combined documents chain.

Demo: LangChain Sequential Chain

Duration: 10 minutes

Showcase how LangChain can create dynamic and context-specific dialogues, aiding in the preparation for bank exams by simulating realistic banking scenarios.

Objective: In this demo, we explore LangChain's sequential chain using a fictional TV show, **Banking Chronicles,** to set up chat models, define chains, and generate dialogue.
The sequential chain will run with a given theme, in this case, **Investment Strategies**, to generate a conversation.

Note

Please download the solution document from the Reference Material Section and follow the Jupyter Notebook for step-by-step execution.

Quick Check

What is NOT true about the LangChain chains?

A. LangChain offers two high-level frameworks for chaining components:

B. The structure of an LLMChain includes a PromptTemplate and a language model

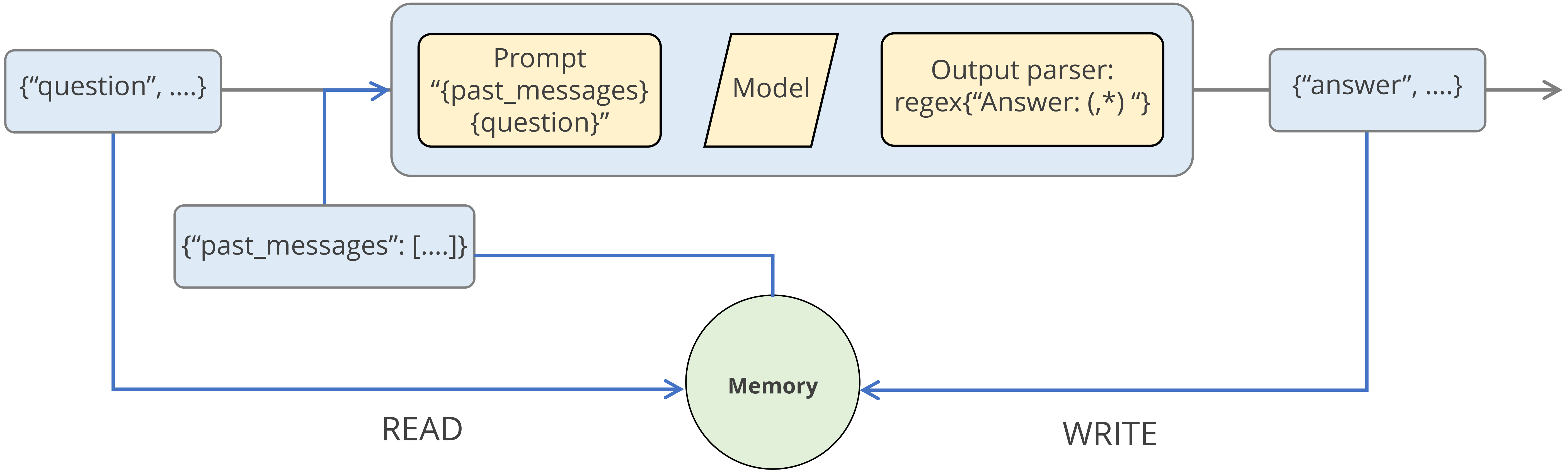
C. The LLMChain operates by formatting the prompt template

D. The LLMChain operates by passing the formatted string to the LLM, and the output from the LLM is passed to another LLM for further processing.

LangChain Memory

LangChain Memory

LangChain Memory stores data between LLM executions. Acting as a temporary storage for information, preferences, or context in chains.



LangChain Memory

In LangChain, memory plays a crucial role in conversational interfaces, allowing systems to reference past interactions.

This is achieved through the storage and querying of information, with two primary actions: reading and writing.

The memory system interacts with a chain twice during a run, augmenting user inputs and storing the inputs and outputs for future reference.

Building Memory into a System

Embedding memory in LangChain enhances chat interactions by organizing and recalling information effectively, creating a more efficient system.

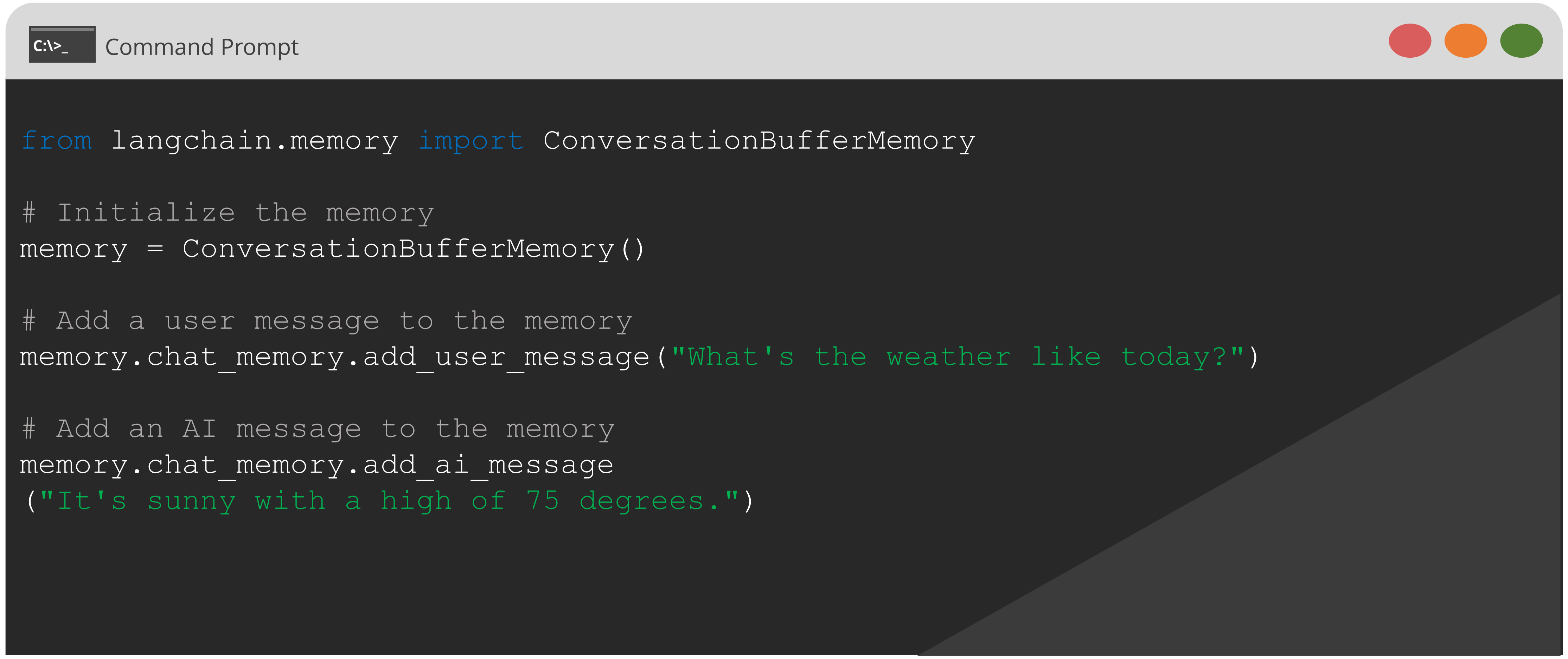
Storing chat messages

LangChain memory stores chat messages in lists and databases, ensuring recording and reference.

Querying chat messages

LangChain organizes stored chat messages, allowing efficient retrieval of relevant information during conversations.

LangChain Memory: Example



```
from langchain.memory import ConversationBufferMemory

# Initialize the memory
memory = ConversationBufferMemory()

# Add a user message to the memory
memory.chat_memory.add_user_message("What's the weather like today?")

# Add an AI message to the memory
memory.chat_memory.add_ai_message
("It's sunny with a high of 75 degrees.")
```

Memory Types in LangChain

Adaptive case management (ACM)

Conversation buffer window memory

Conversation entity memory

Conversation summary memory

VectorStoreRetrieverMemory

Demo: Langchain Memory

Duration: 10 minutes

Description: In this demo, you will learn how to use different types of memory in LangChain. Each type of memory serves a unique purpose and can be used in different scenarios depending on the requirements of your conversation model.

Note

Please download the solution document from the Reference Material Section and follow the Jupyter Notebook for step-by-step execution.

Quick Check

What role does memory play in conversational interfaces within LangChain?

A. Memory serves as a computational resource for processing user inputs.

B. Memory facilitates real-time decision-making in conversational systems.

C. Memory allows systems to reference past interactions through reading and writing.

D. Memory is primarily used for storing visual information in conversational interfaces.

LangChain Agents

LangChain Agents

Agents make use of external tools to perform specific actions. LangChain provides many out-of-the-box agent tools.

LangChain agents are like the conductors of an orchestra, coordinating various components to create a harmonious output.

LLMs utilize tools to access external sources like Google, Wikipedia, YouTube, and Python REPL databases, to solve complex problems.

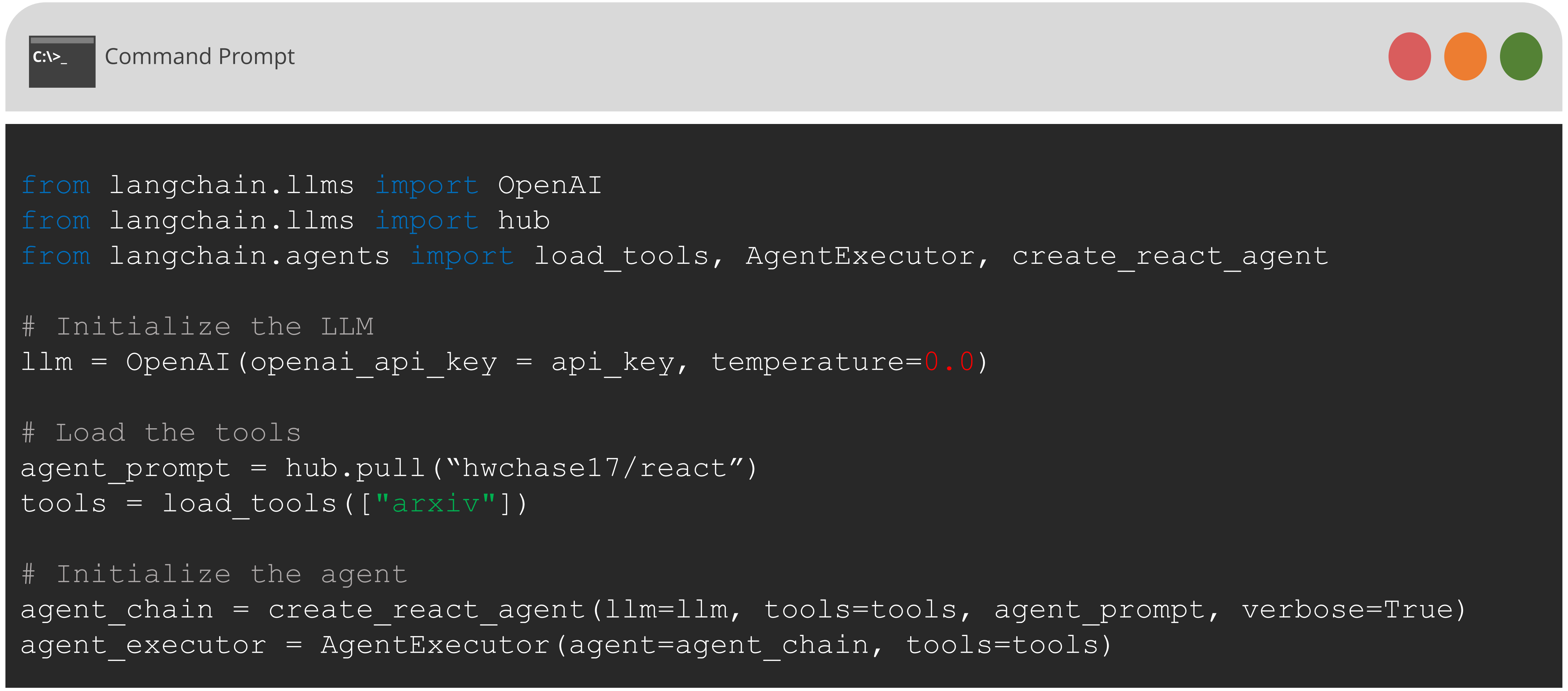
Steps to Implement Agent

They involve an LLM to perform a series of steps:

Decide	Based on the user input or its previous outputs, the agent decides which action to perform.
Perform	The agent performs the chosen action.
Observe	The agent observes the output of the action.
Repeat	The agent repeats the first three steps until it completes the task defined in the user input to the best of its abilities.

Example: Creating an Agent that Accesses Arxiv

Create an agent accessing Arxiv, a well-known research paper portal, and ask it to provide information about a paper.



```
from langchain.llms import OpenAI
from langchain.llms import hub
from langchain.agents import load_tools, AgentExecutor, create_react_agent

# Initialize the LLM
llm = OpenAI(openai_api_key = api_key, temperature=0.0)

# Load the tools
agent_prompt = hub.pull(“hwchase17/react”)
tools = load_tools(["arxiv"])

# Initialize the agent
agent_chain = create_react_agent(llm=llm, tools=tools, agent_prompt, verbose=True)
agent_executor = AgentExecutor(agent=agent_chain, tools=tools)
```

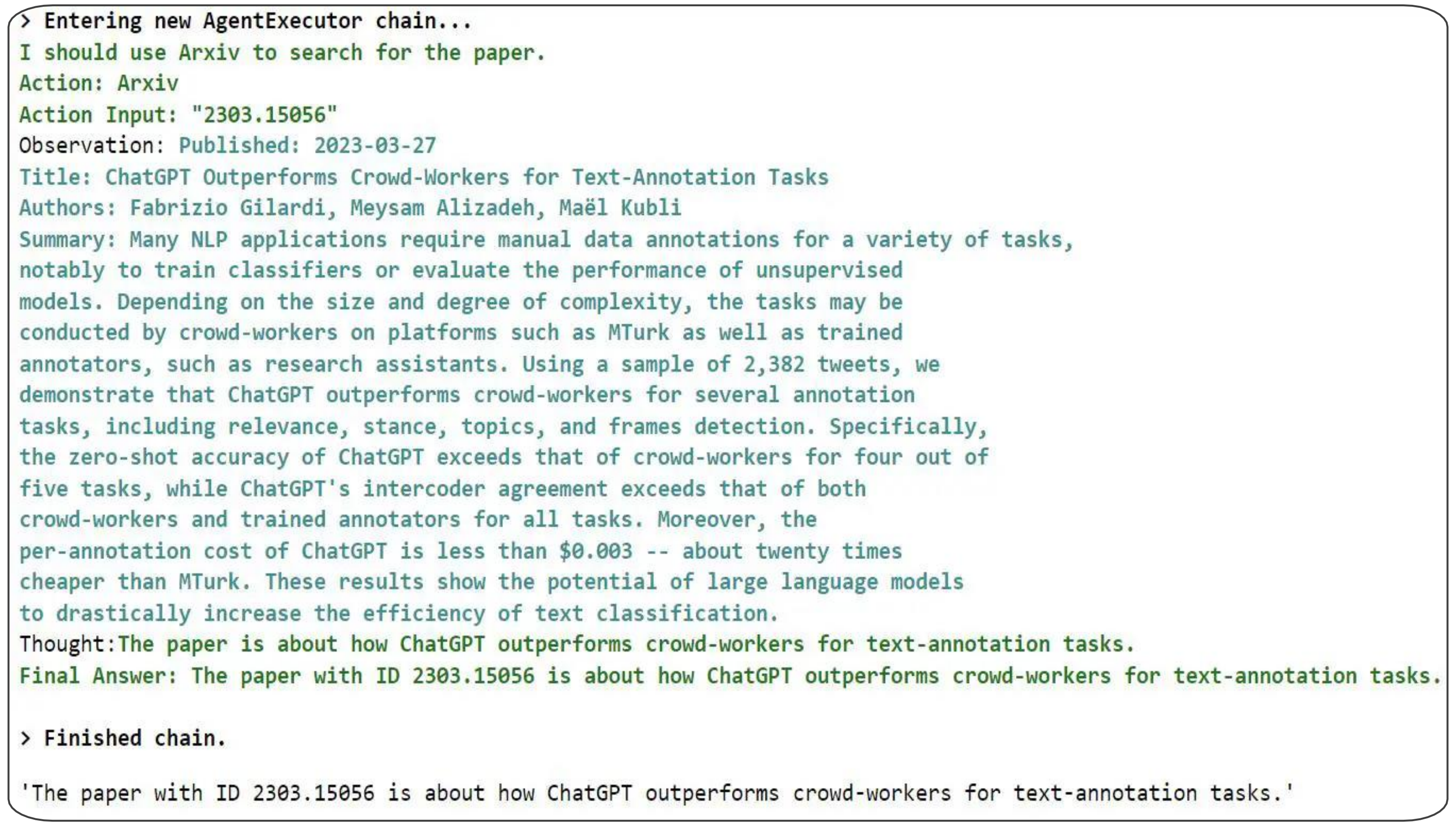
Note

In this example, start by loading the Arxiv tool. Next, initialize an agent with the tool, the LLM, and the agent type. Set **verbose = True** to view the decision-making process of the agent.

Initialize the Agent

The initialize_agent() method returns an object which can be executed like a chain via the run() method.

```
# Run the agent chain
agent_chain.run("Can you tell me about the paper
2303.15056?")
#See the output right in the image.
```



```
> Entering new AgentExecutor chain...
I should use Arxiv to search for the paper.
Action: Arxiv
Action Input: "2303.15056"
Observation: Published: 2023-03-27
Title: ChatGPT Outperforms Crowd-Workers for Text-Annotation Tasks
Authors: Fabrizio Gilardi, Meysam Alizadeh, Maël Kubli
Summary: Many NLP applications require manual data annotations for a variety of tasks,
notably to train classifiers or evaluate the performance of unsupervised
models. Depending on the size and degree of complexity, the tasks may be
conducted by crowd-workers on platforms such as MTurk as well as trained
annotators, such as research assistants. Using a sample of 2,382 tweets, we
demonstrate that ChatGPT outperforms crowd-workers for several annotation
tasks, including relevance, stance, topics, and frames detection. Specifically,
the zero-shot accuracy of ChatGPT exceeds that of crowd-workers for four out of
five tasks, while ChatGPT's intercoder agreement exceeds that of both
crowd-workers and trained annotators for all tasks. Moreover, the
per-annotation cost of ChatGPT is less than $0.003 -- about twenty times
cheaper than MTurk. These results show the potential of large language models
to drastically increase the efficiency of text classification.
Thought:The paper is about how ChatGPT outperforms crowd-workers for text-annotation tasks.
Final Answer: The paper with ID 2303.15056 is about how ChatGPT outperforms crowd-workers for text-annotation tasks.

> Finished chain.

'The paper with ID 2303.15056 is about how ChatGPT outperforms crowd-workers for text-annotation tasks.'
```

Demo: Langchain Agents

Duration: 5 minutes

Description: In this demo, you will learn how to use LangChain Agents. You will understand how to create and initialize an agent, load tools, and use the agent for questions and answers.

Note

Please download the solution document from the Reference Material Section and follow the Jupyter Notebook for step-by-step execution.

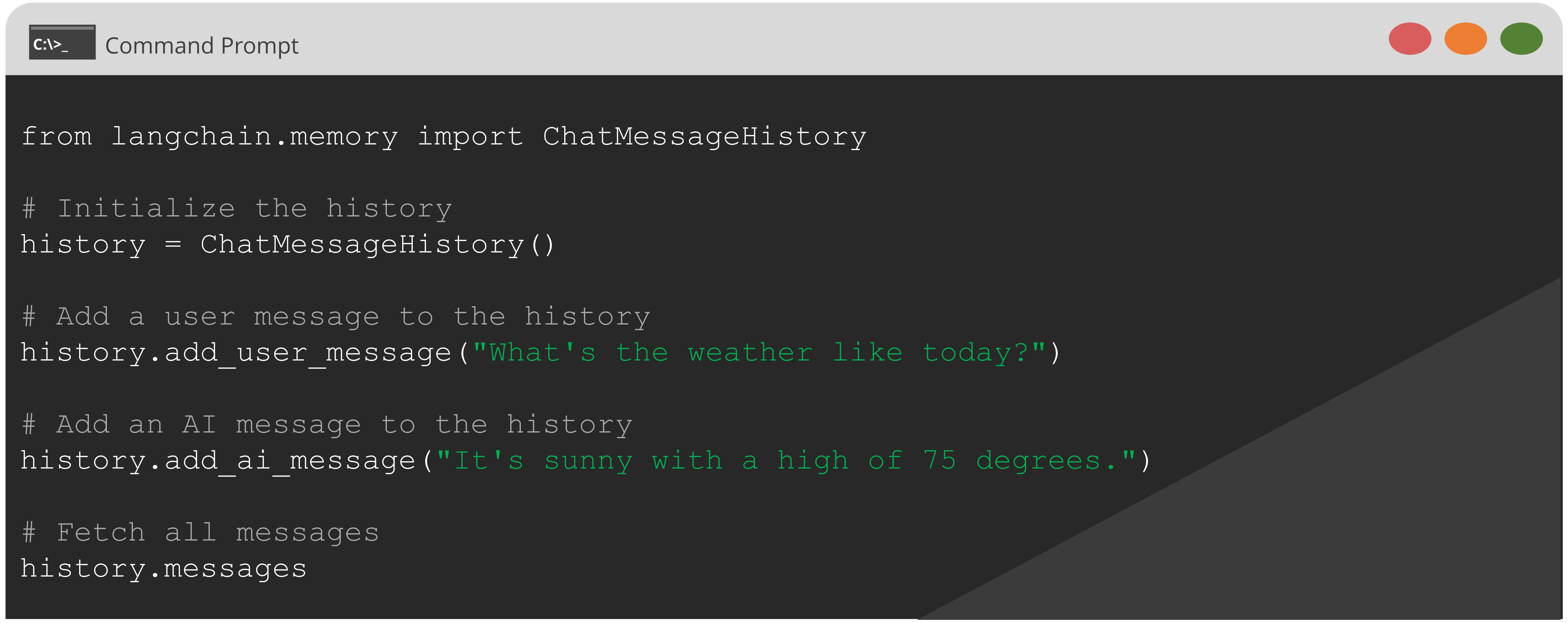
Chat Messages

The ChatMessageHistory class is a fundamental utility class that underpins most memory modules in LangChain.

Think of it as a handy organizer that neatly stores HumanMessages and AIMessages and provides easy access to them.

This class can be particularly useful if you're managing memory outside of a chain.

Chat Messages: Example



```python
from langchain.memory import ChatMessageHistory

# Initialize the history
history = ChatMessageHistory()

# Add a user message to the history
history.add_user_message("What's the weather like today?")

# Add an AI message to the history
history.add_ai_message("It's sunny with a high of 75 degrees.")

# Fetch all messages
history.messages
```

Output:

```
[HumanMessage(content="What's the weather like today?", additional_kwargs={}), AIMessage(content="It's sunny
with a high of 75 degrees.", additional_kwargs={})]
```

Demo: Running Local Falcon LLM

Duration: 10 minutes

This demo will walk you through the process of setting up a local Falcon LLM using LangChain's PromptTemplate and ConversationChain functionalities.

Note

Please download the solution document from the Reference Material Section and follow the Jupyter Notebook for step-by-step execution.

Quick Check

In LangChain, what role do agents play in coordinating various components to achieve a harmonious output?

A. Agents serve as external tools for specific actions.

B. Agents are responsible for fine-tuning LLMs.

C. Agents act as conductors, coordinating various components like an orchestra.

D. Agents access sources like Google, Wikipedia, and YouTube.

Guided Practice

Overview **Duration: 20 minutes**

In this activity, you will walk through the process of using LangChain, an open-source framework that enables the development of applications with large language models (LLMs) like OpenAI's GPT-4. Your goal is to create a chatbot that can answer questions about the BCG 2022 Annual Sustainability Report, a document detailing BCG's effort to address global challenges.

Note

Please download the solution document from the Reference Material Section and follow the Jupyter Notebook for step-by-step execution.

Key Takeaways

- For simple tasks, an LLM suffices, but complex applications may require chaining LLMs or other components.
- In LangChain, embedding memory enhances chat interactions by organizing and recalling information effectively, creating a more efficient system.
- The memory system interacts with a chain twice during a run, augmenting user inputs and storing the inputs and outputs for future reference.
- The ChatMessageHistory class is a basic utility in LangChain, supporting many memory modules.

Q&A