2025

Lesson 09

LangChain And LLM

LangChain

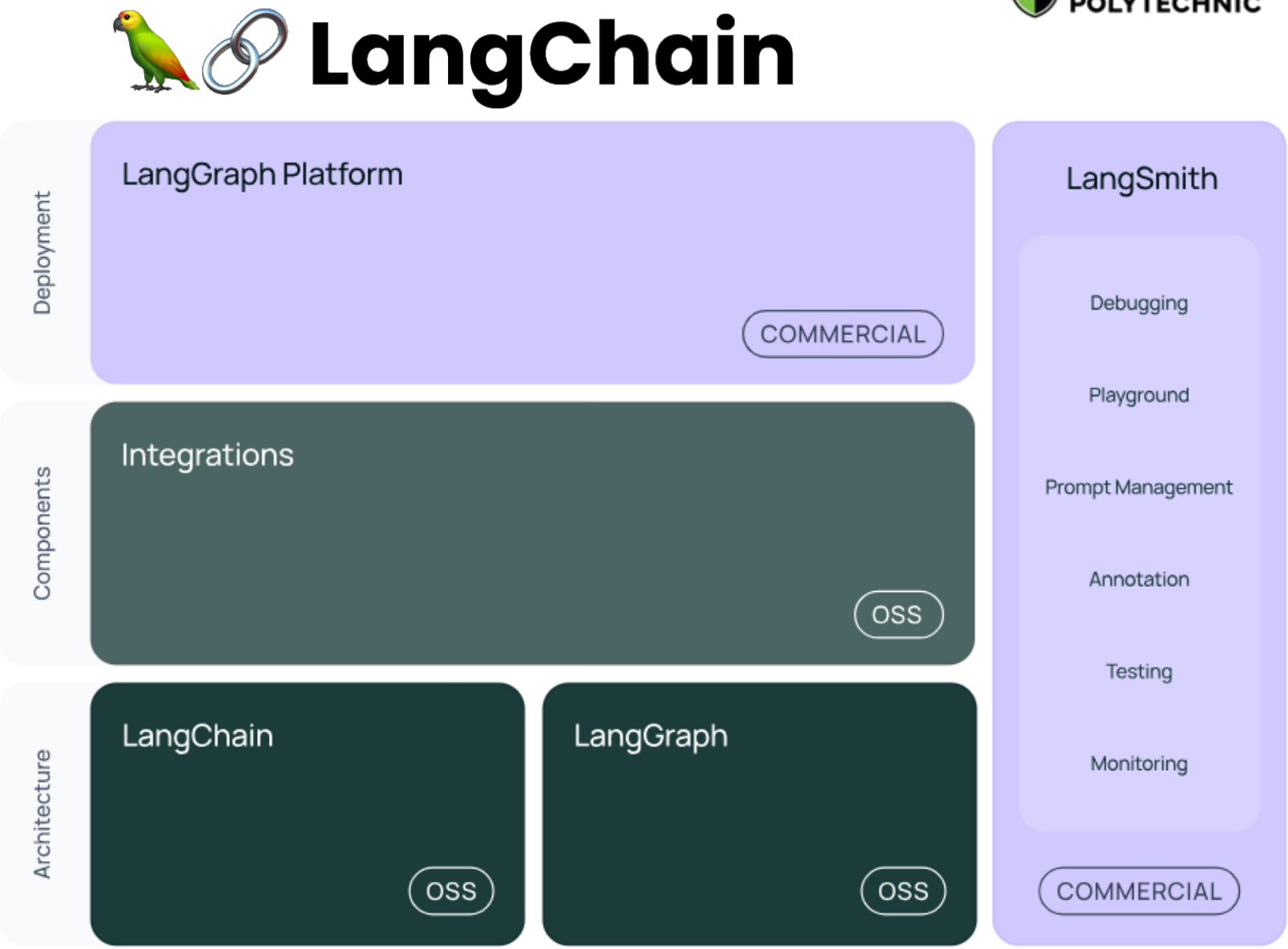
Introduction

LangChain is a framework for developing applications powered by large language models. It enables applications that are context-aware and rely on a language model to reason.

Use LangGraph to build stateful agents with first-class streaming and human-in-the-loop support.

Use LangSmith to inspect, monitor and evaluate your chains, so that you can continuously optimize and deploy with confidence.

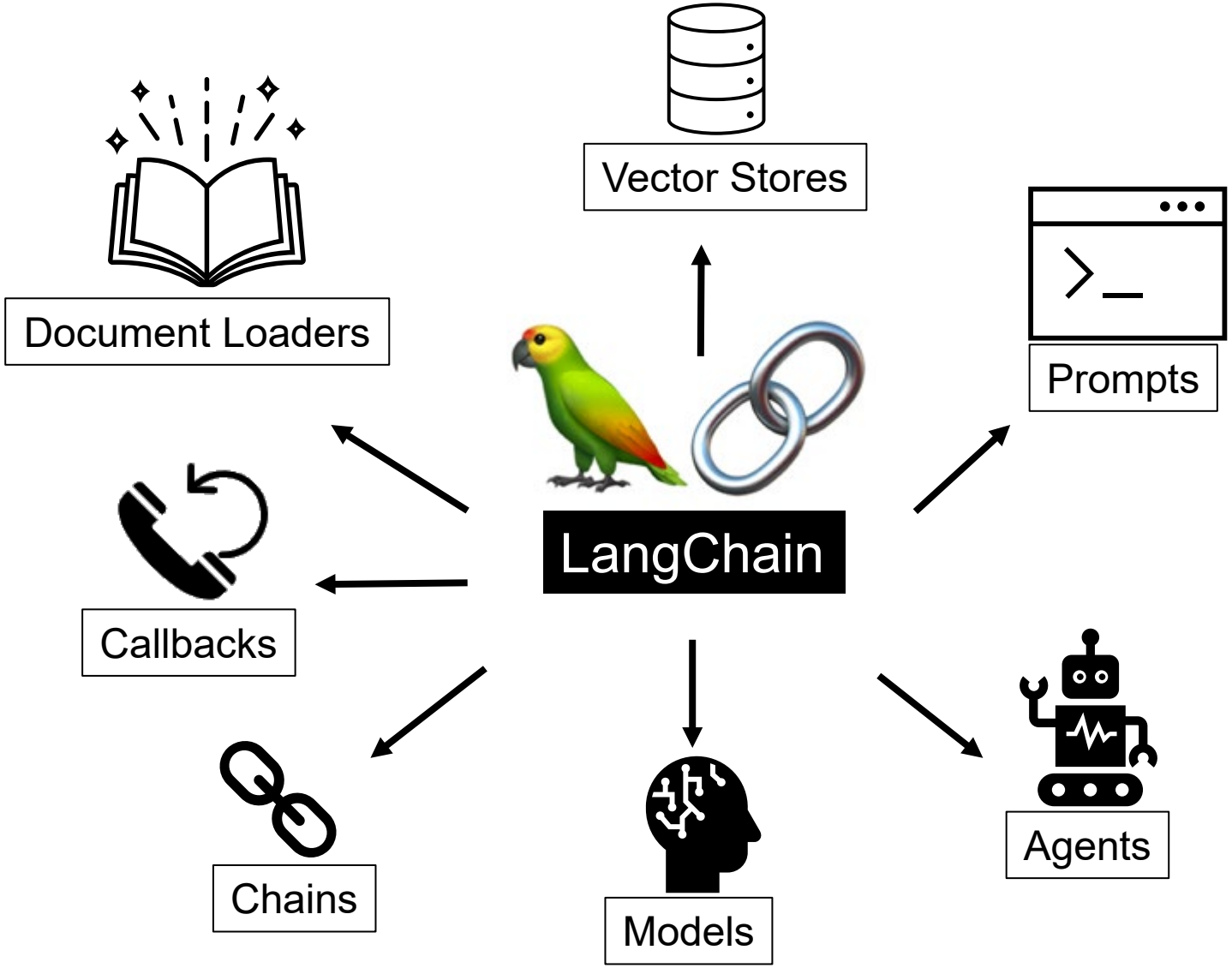
Turn your LangGraph applications into production-ready APIs and Assistants with LangGraph Platform



Source: https://python.langchain.com/docs/introduction/

Components of LangChain

- In LangChain, components are the building blocks used to construct powerful applications that interact with language models (LLMs).
- These components provide modular, reusable, and customizable functionalities, making it easy to design workflows or pipelines for specific tasks.



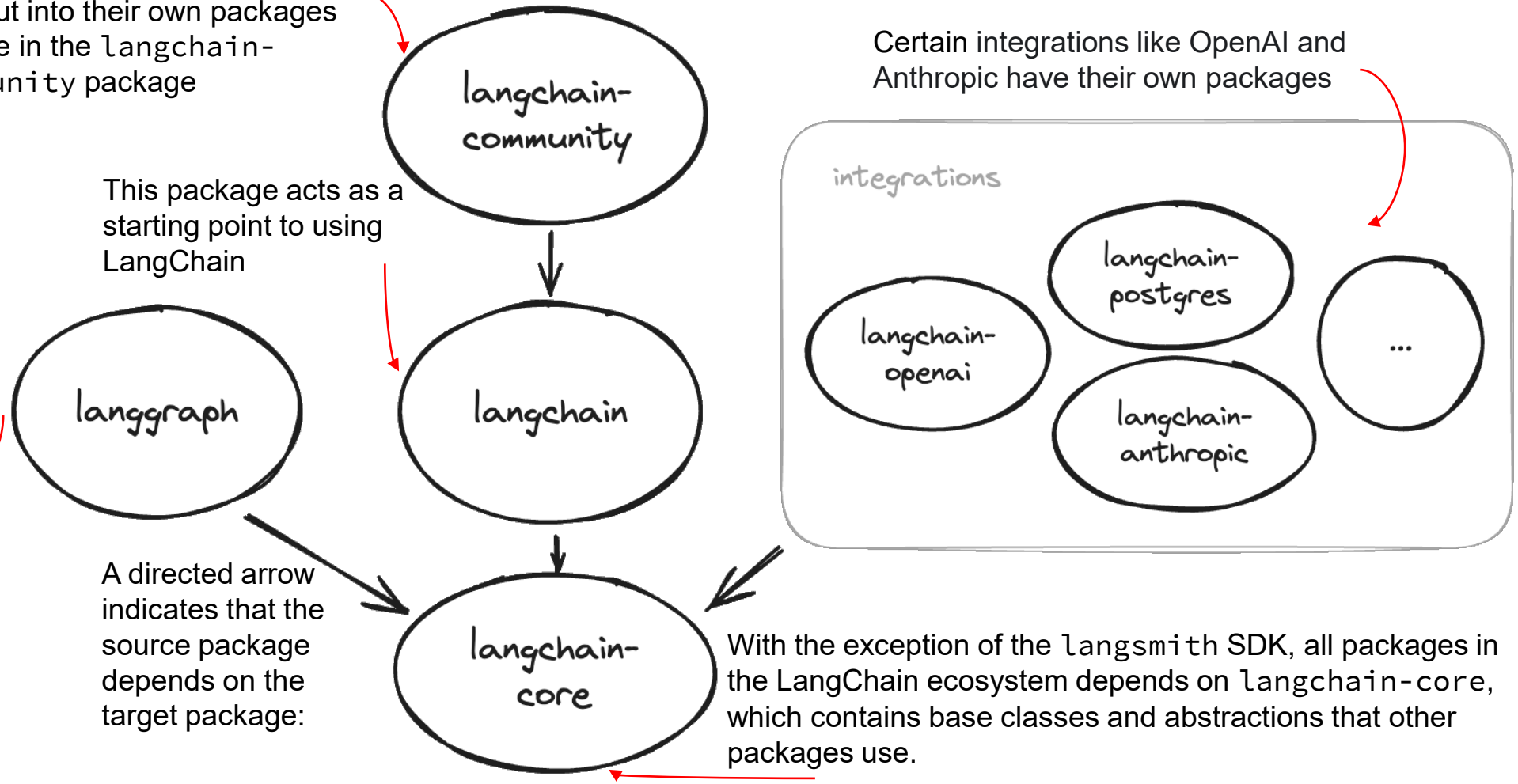
LangChain Ecosystem Packages

Any integrations that haven't been split out into their own packages will live in the `langchain-community` package

Certain integrations like OpenAI and Anthropic have their own packages

This package acts as a starting point to using LangChain

Note: A directed arrow (➔) indicates that the source package depends on the target package



`Langgraph` is a library for building stateful, multi-actor applications with LLMs

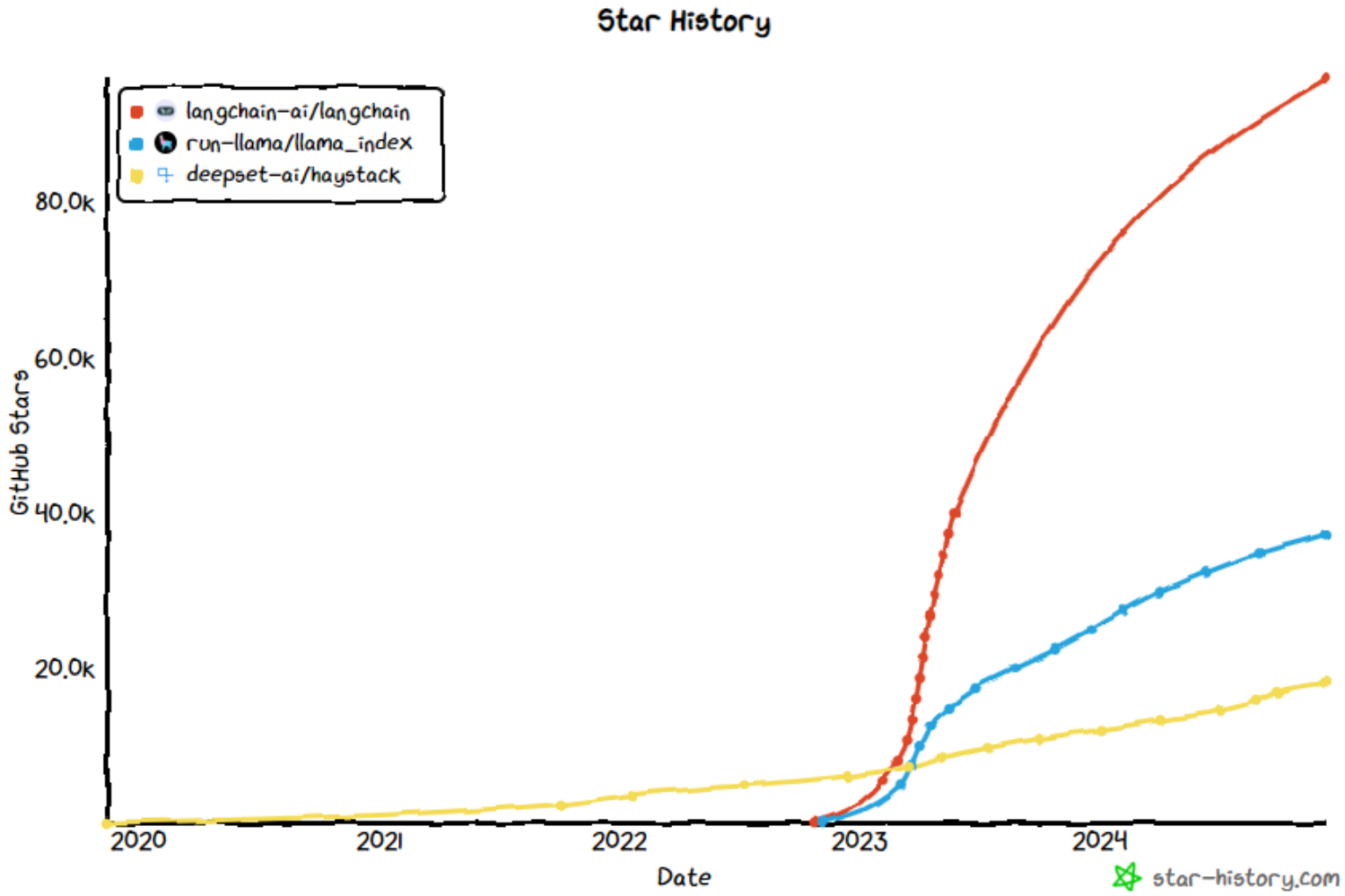
A directed arrow indicates that the source package depends on the target package:

With the exception of the `langsmith` SDK, all packages in the LangChain ecosystem depends on `langchain-core`, which contains base classes and abstractions that other packages use.

Source: https://python.langchain.com/docs/how_to/installation/

Why LangChain?

- LangChain gains popularity in a short period of time.
- Benefits:
  - Increased flexibility
  - Improved performance
  - Enhanced reliability
  - Open source



LangChain API Documentation

Best Friend Forever

python.langchain.com/api_reference/

API Reference | Search Ctrl + K | Docs

Section Navigation

Base packages
- Core
- Langchain
- Text Splitters
- Community
- Experimental

Integrations
- AI21
- Anthropic
- AstraDB
- AWS
- Azure Dynamic Sessions
- Box
- Cerebras
- Chroma
- Cohere
- Couchbase
- Databricks
- Elasticsearch
- Exa
- Fireworks
- Google Community
- Google GenAI
- Groq
- Huggingface
- IBM
- Milvus

LangChain Python API Reference

On this page
- Base packages
- Integrations

LangChain Python API Reference

Welcome to the LangChain Python API reference. This is a reference for all `langchain-x` packages.

For user guides see https://python.langchain.com.

For the legacy API reference hosted on ReadTheDocs see https://api.python.langchain.com/.

Base packages

Core	Langchain	Text Splitters
langchain-core: 0.3.24	langchain: 0.3.11	langchain-text-splitters: 0.3.2
Community	**Experimental**	
langchain-community: 0.3.11	langchain-experimental: 0.3.3	

Integrations

OpenAI	Anthropic	AWS
langchain-openai 0.2.12	langchain-anthropic 0.3.0	langchain-aws 0.2.9
Huggingface	**MistralAI**	**AI21**
langchain-huggingface 0.1.2	langchain-mistralai 0.2.3	langchain-ai21 1.0.1

https://python.langchain.com/api_reference/

Best Practice for API Key(s) Handling

- OpenAI's recommended that the API key(s) be handled with extreme care.
- Never deploy your key in client-side environments like browsers or mobile apps.
- Never commit your key to your repository (for example: GitHub).
- Load the API key via a text file or use environment variable in place of your API key.
- Monitor your account usage and rotate your keys often or when needed.

Source: https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety

Getting Started with LangChain

- LangChain requires integrations with various model providers, data stores, APIs and other third-party components.
- You must provide relevant API keys for LangChain to function. Some methods to achieve this are:
  - Setting up key as an environment variable

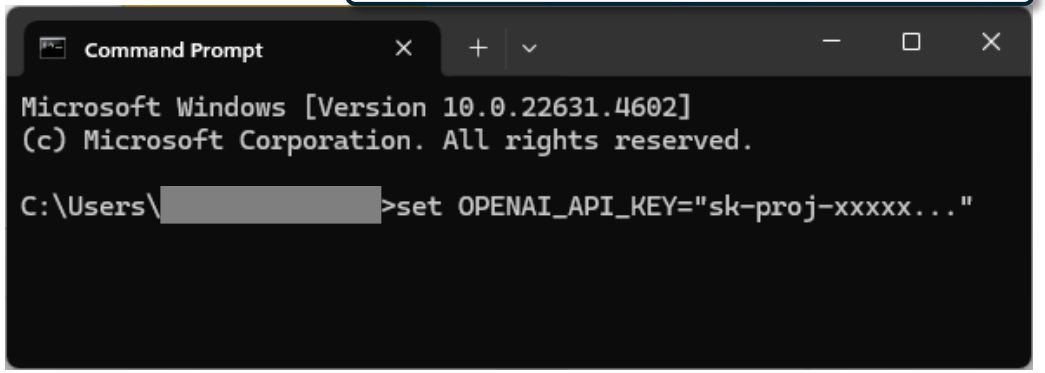
Enter the API key manually

```
import getpass
import os

# setup the OpenAI API Key

# get OpenAI API key ready and enter it when ask
os.environ["OPENAI_API_KEY"] = getpass.getpass()
```

Export the API key at Command Prompt



  - Load the key to an environment variable via a text file

Load the API key from a file

```
import os
from dotenv import load_dotenv

# load OPENAI API key
load_dotenv('env.txt')
openai_api_key = os.getenv('OPENAI_API_KEY')
```

Make sure the method you choose does not publicly expose the API key

A Simple LLM Application

Explain

```
# load langchain libraries
from langchain_openai import ChatOpenAI
from langchain.schema import HumanMessage

chat_model = ChatOpenAI(
    # don't need this if the OpenAI API Key is stored in the environment variable
    #openai_api_key="sk-proj-xxxxxxxxx",
    model_name='gpt-4o-mini'
)

# setup message prompt
text = "What date is Singapore National Day?"
messages = [HumanMessage(content=text)]

# note that Chat Model takes in message objects as input and generate message object as output

response = chat_model.invoke(messages)
print(response.content)
```

Import the libraries

Setup chat model with model ‘gpt-4o-mini’

Setup the human message “What date is Singapore National Day’

Output
Singapore National Day is celebrated on August 9th each year. It commemorates the country's independence from Malaysia in 1965.

Prompt Template

Prompt Template

- Translate user input and parameters into instructions for an LLM.
- Take as input a dictionary where each key represent a variable in the prompt template to fill in.
- Is a string template we can pass variables to in order to generate the final prompt string.
- LangChain documentation ➔ “A prompt template refers to a reproducible way to generate a prompt”.

A Simple LLM Application With Prompt Template

Explain

```
system_template = "You are a helpful assistant that translates {input_language} to
{output_language}."

human_template = "{text}"
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", system_template),
    ("human", human_template),
])

# trsnslate English to French
messages = chat_prompt.format_messages(
    input_language="English",
    output_language="French",
    text="I love programming."
)

response = chat_model.invoke(messages).content
print(response)
```

-Setup system and human templates
-Create the chat prompt template

Create the prompt and populate the values

Output
J'adore la programmation.

LLM API Provider Interface

- LangChain provides integration for different types of model.
- LLM: the model takes a text string as input and returns a text string.
- Chat models: the model takes a list of chat messages as input and return a chat message. Specifically it has 3 major components:
  - HumanMessage – the prompt given by the user.
  - SystemMessage – A context that can be passed to the ChatModel about its role.
  - AIMessage – the response given by LLM.

Message Types

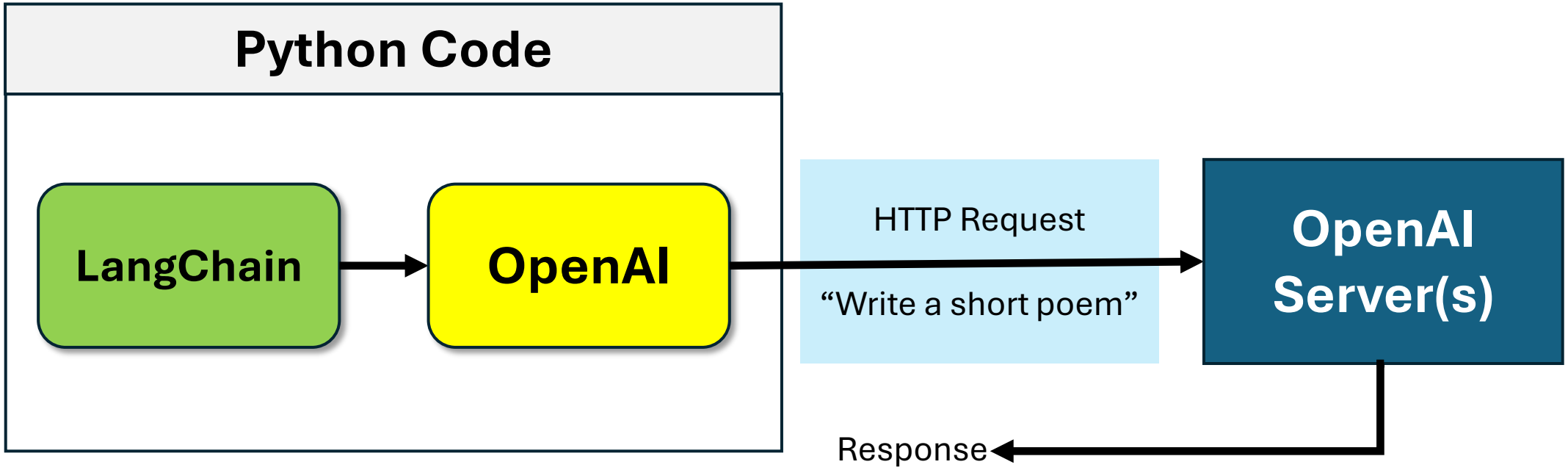
Role	Purpose	Usage Pattern

System	• Helpful background context that guides AI
	• Set the behaviour or context for the conversation. It defines how the AI should respond and what role it should play
		{
		"role": "system", "content": "You are a helpful assistant that specializes in programming and technology."
		}

Human (User)	• Message representing the input from the user. It is what the AI receives as a query or prompt to generate a response	{
		"role": "human", "content": "Can you explain the difference between a list and a tuple in Python?"
		}

AI (Assistant)	• The response generated by the model. It addressed the user's query based on the context provided by the system message	{
		"role": "assistant", "content": "Sure! In Python, a list is a mutable sequence, meaning you can modify its elements after creation, while a tuple is immutable, meaning it cannot be changed once defined. Tuples are generally used for fixed collections of items."
		}

How it work?



*The LLM by OpenAI does not reside in your local machine.
*LangChain also works with open source LLM model like Llama from Meta. You can host the LLM model locally.

LangChain Expression Language

LangChain Expression Language (LCEL)

- LangChain provides a declarative way to compose chains that is more intuitive and productive than directly write code.
- The different components of LCEL are placed in a sequence which is separated by a pipe symbol (|).
- The chain or LCEL is executed from left to right.
  - For example: chain = prompt | model | output_parser
  - The `prompt` output is piped to the `LLM model`. The output of the `LLM model` is then piped to `output_parser` which extracts the text in the output.
- LCEL is a method to create arbitrary custom chains. It is built on the `Runnable` protocol.

Prompt Template & LCEL

Explain

```
from langchain_core.output_parsers.string import StrOutputParser

llm = ChatOpenAI(
    model_name='gpt-4o-mini',
    temperature=0.7,
)

output_parser = StrOutputParser()

human_template = "Write {lines} sentences about {topic}."
prompt = ChatPromptTemplate.from_template(human_template)

lines_topic_dict = {
    "lines" : "3",
    "topic": "Sir Stamford Raffles"
}

lcel_chain_02 = prompt | llm | output_parser

lcel_chain_02.invoke(lines_topic_dict)
```

Parse the output of a language model into a string format

Create the prompt template

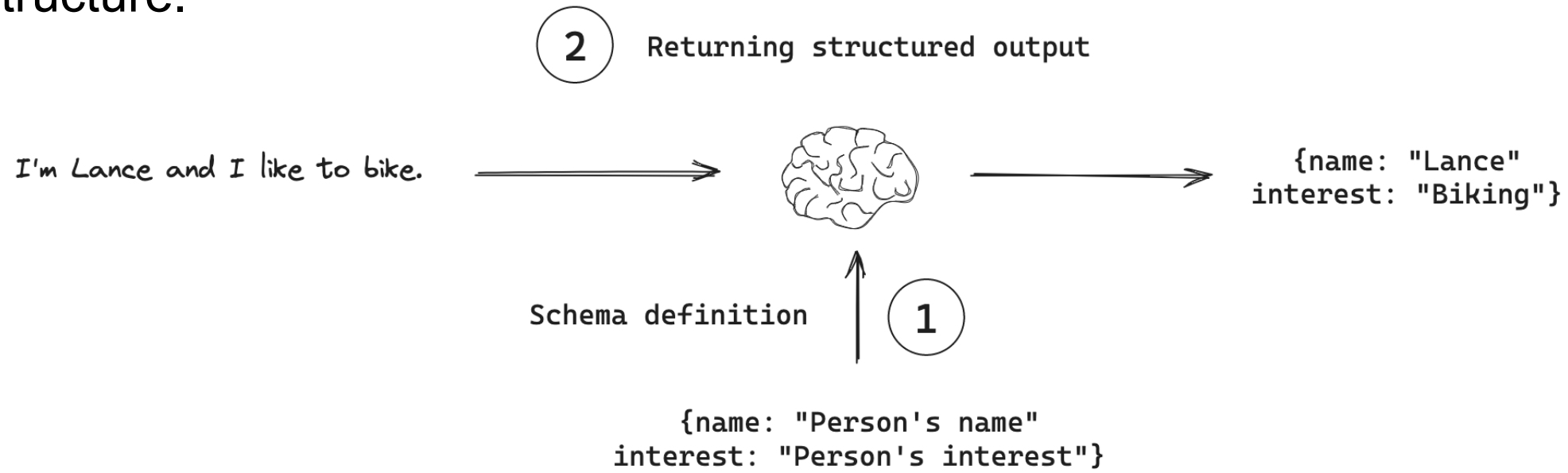
Setup the template parameter dictionary

Use LCEL notation to pipe from prompt to output parser

Output
Sir Stamford Raffles was a British statesman …

Structured Output

- LLM model responds to user directly in natural language.
- There are scenarios where the outputs needs to be in some structured format.
- In LangChain, models can be instructed to output a particular output structure.



Source: https://python.langchain.com/docs/concepts/structured_outputs/

Persistence

Memory

- LLMs are stateless – each incoming query is processed independently of the other interactions i.e. in other words, LLMs don't save anything.
- Memory allows a LLM to remember previous interactions with the user.
- There are many applications where remembering previous interactions is important such as chatbots. Memory persistence allows us to do that.
- There are few pros and cons using memory persistence.

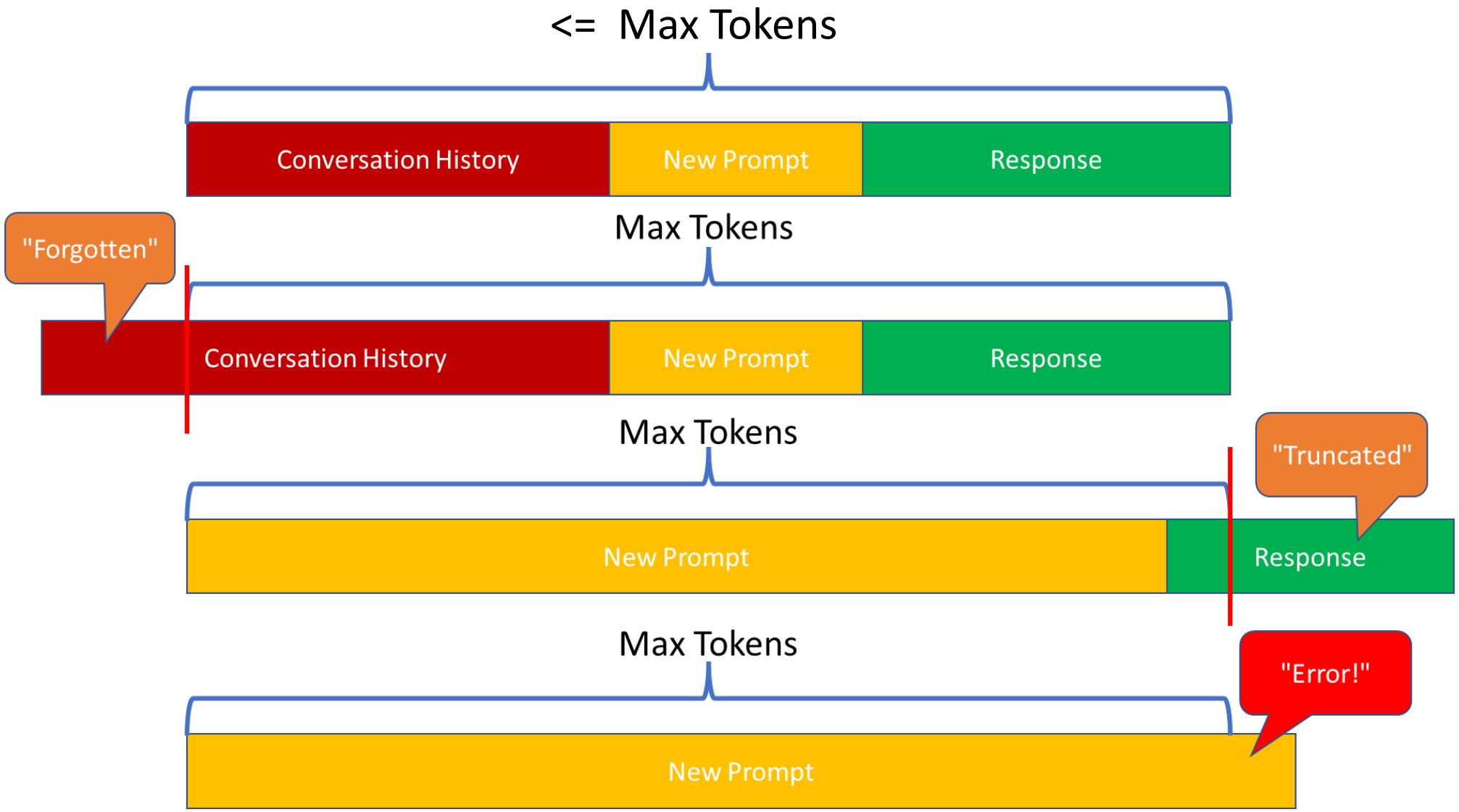
Pros	Cons
Maximum information due to storing everything during conversation exchanges	Increase in token counts will slow down response times and higher costs
API is simple and intuitive	Limited by LLM context window limit 4096 token for text-davinci-003 and gpt-3.5-turbo

Source: https://python.langchain.com/api_reference/langchain/memory.html#

Memory Persistence

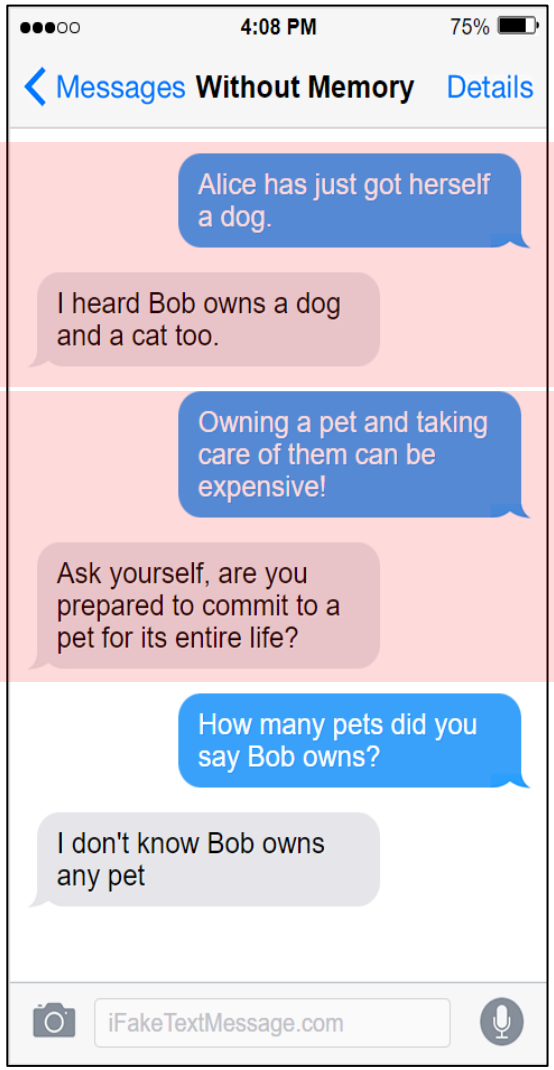
- The basic implementation is to simply stores the conversation history.
- Additional processing may be required in some situations when the conversation history is too large to fit in the content window of the model.
- One method is to summarize each conversation before storing it.
- Starting from v0.3 release of LangChain, LangChain is recommending user to take advantage of LangGraph persistence to incorporate `memory` into new LangChain applications.

Context Window (Recap)

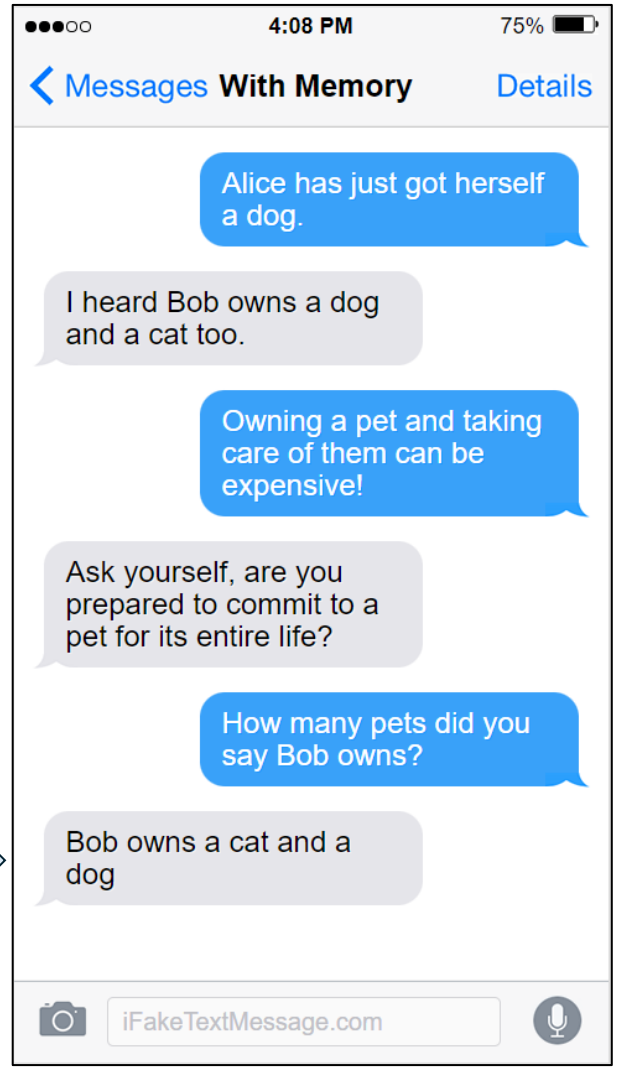


Conversation History

Without memory persistence



With memory persistence



Activity

Activity

- In this activity, we will be creating chat model that takes in a sequences of message and returns chat messages as outputs.
- LangChain does not host any of the chat models. It depends on third party LLM model providers.
- Prompt template serves as a flexible framework for creating dynamic and reusable prompts.
- LCEL is the preferred way to define workflows.
- LangChain Parsers are components to process or transform the output generated by an LLM.
- LLMs are inherently stateless. To enable a more seamless and context-aware user experience, memory persistence is crucial when working with LLMs.

20 mins

Reference

- LangChain
  https://www.langchain.com/
- LangChain Documentation
  https://python.langchain.com/docs/introduction/
- LangChain Tutorial
  https://python.langchain.com/docs/tutorials/llm_chain/
- LangGraph
  https://langchain-ai.github.io/langgraph/tutorials/introduction/

Thank you!