**Advanced LangChain Training Guide**

**Table of Contents**

1. [Introduction](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#introduction)
2. [Prerequisites](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#prerequisites)
3. [Setting Up Your Environment](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#setting-up-your-environment)
4. [Working with OpenAI API](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#working-with-openai-api)
5. [Building Advanced Chains](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#building-advanced-chains)
6. [Implementing Memory](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#implementing-memory)
7. [Integrating Tools](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#integrating-tools)
8. [RAG and Vector Databases](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#rag-and-vector-databases)
9. [Creating Powerful Agents](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#creating-powerful-agents)
10. [Advanced Use Cases](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#advanced-use-cases)
11. [Performance Optimization](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#performance-optimization)
12. [Resources and Next Steps](https://claude.ai/chat/ae0c5540-4618-4125-9d78-575d7abd3b1a#resources-and-next-steps)

**Introduction**

LangChain has emerged as a powerful framework for developing applications powered by language models. This advanced training guide will help you master key components of the LangChain ecosystem including integration with OpenAI, building sophisticated chains, implementing memory for contextual conversations, leveraging tools, creating retrieval-augmented generation (RAG) systems with vector databases, and developing autonomous agents.

**Prerequisites**

Before starting this training, you should have:

* Intermediate Python programming knowledge
* Basic understanding of LLMs and their capabilities
* Familiarity with basic LangChain concepts
* OpenAI API key or access to another LLM provider
* Virtual environment management skills (venv, conda, etc.)

**Setting Up Your Environment**

Let's begin by setting up a proper development environment:

# Create a virtual environment

python -m venv langchain-advanced

source langchain-advanced/bin/activate # On Windows: langchain-advanced\Scripts\activate

# Install necessary packages

pip install langchain openai chromadb faiss-cpu tiktoken langchainhub langchain-openai

pip install langchain-community langchain-experimental langserve

# Set your OpenAI API key

export OPENAI\_API\_KEY="your-api-key-here" # On Windows: set OPENAI\_API\_KEY=your-api-key-here

Create a project structure:

langchain-advanced-project/

├── data/

│ └── documents/

├── notebooks/

├── src/

│ ├── agents/

│ ├── chains/

│ ├── memory/

│ ├── rag/

│ └── tools/

├── tests/

├── app.py

├── requirements.txt

└── README.md

**Working with OpenAI API**

LangChain provides streamlined integration with the OpenAI API through the langchain-openai package.

**Basic Model Integration**

from langchain\_openai import ChatOpenAI

from langchain\_core.messages import HumanMessage, SystemMessage

# Initialize the model

llm = ChatOpenAI(

model="gpt-4-turbo",

temperature=0.7,

max\_tokens=2000

)

# Simple invocation

messages = [

SystemMessage(content="You are a helpful AI assistant specialized in Python programming."),

HumanMessage(content="Write a function that calculates the Fibonacci sequence.")

]

response = llm.invoke(messages)

print(response.content)

**Streaming Responses**

from langchain\_core.callbacks import StreamingStdOutCallbackHandler

# Initialize with streaming

streaming\_llm = ChatOpenAI(

model="gpt-4",

temperature=0.7,

streaming=True,

callbacks=[StreamingStdOutCallbackHandler()]

)

# Messages will stream to stdout

response = streaming\_llm.invoke(messages)

**Token Usage Tracking**

from langchain\_core.callbacks import get\_openai\_callback

with get\_openai\_callback() as cb:

response = llm.invoke(messages)

print(f"Total Tokens: {cb.total\_tokens}")

print(f"Prompt Tokens: {cb.prompt\_tokens}")

print(f"Completion Tokens: {cb.completion\_tokens}")

print(f"Total Cost (USD): ${cb.total\_cost}")

**Function Calling**

from langchain\_openai import ChatOpenAI

from langchain\_core.messages import HumanMessage

import json

# Define functions

tools = [

{

"type": "function",

"function": {

"name": "get\_weather",

"description": "Get the current weather in a given location",

"parameters": {

"type": "object",

"properties": {

"location": {

"type": "string",

"description": "The city and state, e.g. San Francisco, CA"

},

"unit": {

"type": "string",

"enum": ["celsius", "fahrenheit"]

}

},

"required": ["location"]

}

}

}

]

# Initialize model with tools

llm = ChatOpenAI(model="gpt-4-turbo").bind(tools=tools)

# Invoke with a message that will trigger function calling

message = HumanMessage(content="What's the weather like in Boston right now?")

response = llm.invoke([message])

# Print the function call info

print(json.dumps(response.additional\_kwargs, indent=2))

**Building Advanced Chains**

Chains in LangChain enable the composition of multiple components into powerful pipelines.

**Sequential Chains**

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_openai import ChatOpenAI

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_core.runnables import RunnablePassthrough

# First chain to generate product features

feature\_prompt = ChatPromptTemplate.from\_template(

"Generate 5 innovative features for a {product} targeting {audience}."

)

feature\_chain = feature\_prompt | ChatOpenAI(model="gpt-3.5-turbo") | StrOutputParser()

# Second chain to create marketing copy based on features

marketing\_prompt = ChatPromptTemplate.from\_template(

"Create compelling marketing copy for a {product} with the following features:\n{features}"

)

marketing\_chain = marketing\_prompt | ChatOpenAI(model="gpt-4") | StrOutputParser()

# Connect chains using RunnablePassthrough

full\_chain = {

"product": RunnablePassthrough(),

"audience": lambda x: "young professionals",

"features": lambda x: feature\_chain.invoke({"product": x, "audience": "young professionals"})

} | marketing\_chain

# Invoke the full chain

result = full\_chain.invoke("smart water bottle")

print(result)

**Router Chain**

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_core.output\_parsers import JsonOutputParser

from langchain\_openai import ChatOpenAI

from langchain.chains.router import MultiRouteChain

from langchain.chains.router.llm\_router import LLMRouterChain, RouterOutputParser

from langchain\_core.runnables import RunnableBranch

# Create destination chains

math\_chain = ChatPromptTemplate.from\_template(

"You are a math expert. Solve this problem: {input}"

) | ChatOpenAI(model="gpt-4") | StrOutputParser()

programming\_chain = ChatPromptTemplate.from\_template(

"You are a programming expert. Answer this question: {input}"

) | ChatOpenAI(model="gpt-4") | StrOutputParser()

history\_chain = ChatPromptTemplate.from\_template(

"You are a history expert. Answer this question: {input}"

) | ChatOpenAI(model="gpt-3.5-turbo") | StrOutputParser()

general\_chain = ChatPromptTemplate.from\_template(

"Answer this general knowledge question: {input}"

) | ChatOpenAI(model="gpt-3.5-turbo") | StrOutputParser()

# Create router prompt

router\_prompt = ChatPromptTemplate.from\_template(

"""Given the user's input, classify it as either 'math', 'programming', 'history', or 'general'.

User input: {input}

Output a JSON with a single key 'destination' and the value being one of: 'math', 'programming', 'history', 'general'."""

)

# Create router chain

router\_chain = router\_prompt | ChatOpenAI(model="gpt-3.5-turbo") | JsonOutputParser()

# Create branch

chain = RunnableBranch(

(lambda x: x["destination"] == "math", math\_chain),

(lambda x: x["destination"] == "programming", programming\_chain),

(lambda x: x["destination"] == "history", history\_chain),

general\_chain

)

# Full pipeline

full\_chain = {

"destination": lambda x: router\_chain.invoke({"input": x}),

"input": lambda x: x

} | chain

# Test

result = full\_chain.invoke("Calculate the derivative of f(x) = x^3 + 2x^2 - 4x + 7")

print(result)

**Batch Processing with Chains**

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_openai import ChatOpenAI

from langchain\_core.output\_parsers import StrOutputParser

# Create a simple chain

prompt = ChatPromptTemplate.from\_template("Summarize the following text in one sentence: {text}")

chain = prompt | ChatOpenAI(model="gpt-3.5-turbo") | StrOutputParser()

# Sample texts to process

texts = [

"The quick brown fox jumps over the lazy dog.",

"To be or not to be, that is the question.",

"It was the best of times, it was the worst of times."

]

# Process in batch

results = chain.batch([{"text": t} for t in texts])

for text, summary in zip(texts, results):

print(f"Original: {text}")

print(f"Summary: {summary}")

print("-" \* 50)

**Implementing Memory**

Memory mechanisms allow chains and agents to maintain context across interactions.

**Conversation Buffer Memory**

from langchain.memory import ConversationBufferMemory

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_openai import ChatOpenAI

from langchain\_core.runnables import RunnablePassthrough

from langchain\_core.output\_parsers import StrOutputParser

# Initialize memory

memory = ConversationBufferMemory(return\_messages=True)

# Add some messages to memory

memory.chat\_memory.add\_user\_message("Hello, my name is John.")

memory.chat\_memory.add\_ai\_message("Hi John! How can I help you today?")

# Create the chain with memory

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful AI assistant."),

("placeholder", "{history}"),

("human", "{input}")

])

chain = {

"history": lambda \_: memory.load\_memory\_variables({})["history"],

"input": RunnablePassthrough()

} | prompt | ChatOpenAI() | StrOutputParser()

# Run the chain

response = chain.invoke("What's my name?")

print(response)

# Save the new exchange to memory

memory.chat\_memory.add\_user\_message("What's my name?")

memory.chat\_memory.add\_ai\_message(response)

# Run again with updated memory

response = chain.invoke("Do you remember what we talked about?")

print(response)

**Window Memory for Long Conversations**

from langchain.memory import ConversationBufferWindowMemory

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_openai import ChatOpenAI

from langchain\_core.runnables import RunnablePassthrough

from langchain\_core.output\_parsers import StrOutputParser

# Initialize window memory (keeps only last k interactions)

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

# Simulate a longer conversation

conversation = [

("user", "My name is Sarah."),

("ai", "Nice to meet you, Sarah!"),

("user", "I'm planning a trip to Japan."),

("ai", "That sounds exciting! When are you planning to visit Japan?"),

("user", "In cherry blossom season."),

("ai", "Cherry blossom season in Japan typically runs from late March to early April. It's a beautiful time to visit!"),

]

# Add the conversation to memory

for role, content in conversation:

if role == "user":

memory.chat\_memory.add\_user\_message(content)

else:

memory.chat\_memory.add\_ai\_message(content)

# Create chain with window memory

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful travel assistant."),

("placeholder", "{history}"),

("human", "{input}")

])

chain = {

"history": lambda \_: memory.load\_memory\_variables({})["history"],

"input": RunnablePassthrough()

} | prompt | ChatOpenAI() | StrOutputParser()

# Test with a question about earlier conversation

response = chain.invoke("What's my name?")

print("Response when asking about name (which should be outside window):", response)

# Test with question about recent conversation

response = chain.invoke("What season did I mention for my Japan trip?")

print("Response about recent topic:", response)

**Summary Memory for Long-Term Context**

from langchain.memory import ConversationSummaryMemory

from langchain\_openai import ChatOpenAI

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_core.runnables import RunnablePassthrough

from langchain\_core.output\_parsers import StrOutputParser

# Initialize summary memory with an LLM for summarization

memory = ConversationSummaryMemory(

llm=ChatOpenAI(model="gpt-3.5-turbo"),

return\_messages=True

)

# Simulate a detailed conversation about a project

memory.chat\_memory.add\_user\_message("I'm working on a machine learning project to predict customer churn.")

memory.chat\_memory.add\_ai\_message("That's an interesting project! What kind of data do you have?")

memory.chat\_memory.add\_user\_message("I have historical customer data including usage patterns, support tickets, and billing information.")

memory.chat\_memory.add\_ai\_message("Great! Have you identified key features that might be indicators of churn?")

memory.chat\_memory.add\_user\_message("Yes, we've found that declining usage in the 30 days before renewal and unresolved support tickets are strong indicators.")

memory.chat\_memory.add\_ai\_message("Those are good signals. Have you considered using a gradient boosting algorithm like XGBoost for this classification task?")

memory.chat\_memory.add\_user\_message("We're currently comparing XGBoost, Random Forest, and a deep learning approach.")

memory.chat\_memory.add\_ai\_message("That's a solid approach. For churn prediction, you might also want to consider adding features that capture the change in usage patterns over time, not just absolute values.")

# Create chain with summary memory

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful AI assistant specializing in machine learning."),

("placeholder", "{history}"),

("human", "{input}")

])

chain = {

"history": lambda \_: memory.load\_memory\_variables({})["history"],

"input": RunnablePassthrough()

} | prompt | ChatOpenAI(model="gpt-4") | StrOutputParser()

# Test with a question about the project

response = chain.invoke("What model approaches were we considering for the project?")

print(response)

# Check the generated summary

print("\nGenerated conversation summary:")

print(memory.predict\_new\_summary(memory.chat\_memory.messages, ""))

**Entity Memory for Key Information**

from langchain.memory import ConversationEntityMemory

from langchain\_openai import ChatOpenAI

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_core.runnables import RunnablePassthrough

from langchain\_core.output\_parsers import StrOutputParser

# Initialize entity memory

llm = ChatOpenAI(model="gpt-3.5-turbo")

memory = ConversationEntityMemory(llm=llm)

# Simulate a conversation with entity information

memory.save\_context(

{"input": "My name is Alex and I work at TechCorp as a senior developer."},

{"output": "Nice to meet you, Alex! How long have you been at TechCorp?"}

)

memory.save\_context(

{"input": "I've been there for 5 years, mainly focusing on cloud architecture."},

{"output": "That's impressive! Cloud architecture is a critical area. Are you working on any specific projects?"}

)

memory.save\_context(

{"input": "Yes, I'm leading Project Nebula, which is a new distributed database system."},

{"output": "Project Nebula sounds fascinating! What technologies are you using for this distributed database system?"}

)

# Create chain with entity memory

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful AI assistant with perfect recall of our conversation."),

("human", "{input}"),

("human", "Relevant entities from our conversation: {entities}")

])

chain = {

"entities": lambda x: memory.load\_memory\_variables({})["entities"],

"input": RunnablePassthrough()

} | prompt | llm | StrOutputParser()

# Test the chain with questions about specific entities

response = chain.invoke("What is my name and where do I work?")

print(response)

response = chain.invoke("What project am I working on?")

print(response)

# Check stored entities

print("\nStored entity information:")

for entity, info in memory.entity\_store.store.items():

print(f"{entity}: {info}")

**Integrating Tools**

Tools allow LangChain agents to interact with external services and APIs.

**Building Basic Tools**

from langchain\_core.tools import tool

from langchain\_openai import ChatOpenAI

from langchain.agents import create\_openai\_tools\_agent

from langchain.agents import AgentExecutor

from langchain\_core.prompts import ChatPromptTemplate

import requests

import json

from datetime import datetime

# Create a weather tool

@tool

def get\_weather(location: str, unit: str = "fahrenheit") -> str:

"""Get the current weather in a given location."""

# In a real application, you'd call a weather API here

# For this example, we'll return mock data

weather\_data = {

"New York": {"temp": 72, "condition": "Sunny"},

"London": {"temp": 18, "condition": "Rainy"},

"Tokyo": {"temp": 28, "condition": "Partly Cloudy"},

"Sydney": {"temp": 25, "condition": "Clear"}

}

if location in weather\_data:

data = weather\_data[location]

temp = data["temp"]

# Convert temperature if necessary

if unit.lower() == "celsius" and location in ["New York"]:

temp = round((temp - 32) \* 5/9)

elif unit.lower() == "fahrenheit" and location not in ["New York"]:

temp = round((temp \* 9/5) + 32)

return f"The current weather in {location} is {temp}°{'C' if unit.lower() == 'celsius' else 'F'} and {data['condition'].lower()}."

else:

return f"Weather data for {location} is not available."

# Create a date tool

@tool

def get\_current\_date() -> str:

"""Get the current date and time."""

now = datetime.now()

return f"The current date and time is {now.strftime('%Y-%m-%d %H:%M:%S')}"

# Create a calculator tool

@tool

def calculate(expression: str) -> str:

"""Calculate the result of a mathematical expression."""

try:

# Be careful with eval() in production!

# Use a safer alternative in real applications

result = eval(expression)

return f"The result of {expression} is {result}"

except Exception as e:

return f"Error calculating {expression}: {str(e)}"

# Create an agent with these tools

tools = [get\_weather, get\_current\_date, calculate]

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful assistant with access to tools. Use them when necessary."),

("human", "{input}"),

])

llm = ChatOpenAI(model="gpt-4-turbo")

agent = create\_openai\_tools\_agent(llm, tools, prompt)

agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# Test the agent

response = agent\_executor.invoke({"input": "What's the weather like in Tokyo in celsius, and what's 345 \* 27?"})

print(response["output"])

**Web Browsing Tool**

from langchain\_core.tools import tool

from langchain\_openai import ChatOpenAI

from langchain.agents import create\_openai\_tools\_agent

from langchain.agents import AgentExecutor

from langchain\_core.prompts import ChatPromptTemplate

import requests

from bs4 import BeautifulSoup

@tool

def search\_web(query: str) -> str:

"""Search the web for information on a specific topic."""

# In a real scenario, you would use a real search API

# For this example, let's simulate a search result

search\_results = {

"climate change": "Climate change is the long-term alteration of temperature and weather patterns. Recent research shows accelerating polar ice melt and rising sea levels.",

"artificial intelligence": "Artificial intelligence (AI) is intelligence demonstrated by machines, as opposed to natural intelligence displayed by animals including humans.",

"python programming": "Python is a high-level, general-purpose programming language known for its readability and versatility.",

}

# Check if we have a result for this query

for key in search\_results:

if key in query.lower():

return search\_results[key]

return "No specific information found on that topic."

@tool

def fetch\_webpage\_content(url: str) -> str:

"""Fetch and extract text content from a webpage."""

try:

# In a production environment, handle this more robustly

# This is a simplified example

headers = {

"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36"

}

response = requests.get(url, headers=headers, timeout=10)

response.raise\_for\_status()

soup = BeautifulSoup(response.text, 'html.parser')

# Remove script and style elements

for script in soup(["script", "style"]):

script.extract()

text = soup.get\_text()

lines = (line.strip() for line in text.splitlines())

chunks = (phrase.strip() for line in lines for phrase in line.split(" "))

text = '\n'.join(chunk for chunk in chunks if chunk)

# Limit to first 500 characters for this example

return text[:500] + "..." if len(text) > 500 else text

except Exception as e:

return f"Error fetching {url}: {str(e)}"

# Create an agent with web tools

web\_tools = [search\_web, fetch\_webpage\_content]

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful research assistant with access to web search and content extraction tools."),

("human", "{input}"),

])

llm = ChatOpenAI(model="gpt-4-turbo")

web\_agent = create\_openai\_tools\_agent(llm, web\_tools, prompt)

web\_agent\_executor = AgentExecutor(agent=web\_agent, tools=web\_tools, verbose=True)

# Test the web agent

response = web\_agent\_executor.invoke({"input": "What can you tell me about climate change?"})

print(response["output"])

**Database Tool**

from langchain\_core.tools import tool

from langchain\_openai import ChatOpenAI

from langchain.agents import create\_openai\_tools\_agent

from langchain.agents import AgentExecutor

from langchain\_core.prompts import ChatPromptTemplate

import sqlite3

import pandas as pd

# Create a simple in-memory SQLite database

conn = sqlite3.connect(':memory:')

cursor = conn.cursor()

# Create a sample table

cursor.execute('''

CREATE TABLE employees (

id INTEGER PRIMARY KEY,

name TEXT,

department TEXT,

salary REAL,

hire\_date TEXT

)

''')

# Insert sample data

employees = [

(1, 'John Smith', 'Engineering', 85000, '2020-03-15'),

(2, 'Sarah Johnson', 'Marketing', 78000, '2019-11-01'),

(3, 'Michael Brown', 'Engineering', 92000, '2018-06-23'),

(4, 'Emma Davis', 'HR', 65000, '2021-09-12'),

(5, 'James Wilson', 'Marketing', 81000, '2020-01-28'),

(6, 'Linda Miller', 'Engineering', 89000, '2017-04-10'),

(7, 'Robert Taylor', 'Finance', 110000, '2016-08-17'),

(8, 'Patricia Anderson', 'HR', 68000, '2022-02-05'),

]

cursor.executemany('INSERT INTO employees VALUES (?, ?, ?, ?, ?)', employees)

conn.commit()

@tool

def run\_sql\_query(query: str) -> str:

"""Run an SQL query on the employee database and return results."""

try:

result = pd.read\_sql\_query(query, conn)

return result.to\_string()

except Exception as e:

return f"Error executing SQL query: {str(e)}"

@tool

def get\_table\_schema() -> str:

"""Get the schema of available tables in the database."""

try:

cursor.execute("SELECT name FROM sqlite\_master WHERE type='table';")

tables = cursor.fetchall()

schema\_info = ""

for table in tables:

table\_name = table[0]

cursor.execute(f"PRAGMA table\_info({table\_name});")

columns = cursor.fetchall()

schema\_info += f"Table: {table\_name}\n"

schema\_info += "Columns:\n"

for col in columns:

schema\_info += f" - {col[1]} ({col[2]})\n"

schema\_info += "\n"

return schema\_info

except Exception as e:

return f"Error getting schema: {str(e)}"

# Create an agent with database tools

db\_tools = [run\_sql\_query, get\_table\_schema]

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful database assistant that can run SQL queries and provide information about database schema."),

("human", "{input}"),

])

llm = ChatOpenAI(model="gpt-4-turbo")

db\_agent = create\_openai\_tools\_agent(llm, db\_tools, prompt)

db\_agent\_executor = AgentExecutor(agent=db\_agent, tools=db\_tools, verbose=True)

# Test the database agent

response = db\_agent\_executor.invoke({"input": "What tables are available and what's their structure?"})

print(response["output"])

response = db\_agent\_executor.invoke({"input": "What is the average salary by department?"})

print(response["output"])

**Custom API Tool**

from langchain\_core.tools import tool

from langchain\_openai import ChatOpenAI

from langchain.agents import create\_openai\_tools\_agent

from langchain.agents import AgentExecutor

from langchain\_core.prompts import ChatPromptTemplate

import json

import requests

@tool

def get\_exchange\_rate(base\_currency: str, target\_currency: str) -> str:

"""Get the current exchange rate between two currencies."""

# In a real application, you would use an actual currency API

# For this example, we'll use simulated data

exchange\_rates = {

"USD": {"EUR": 0.92, "GBP": 0.78, "JPY": 147.8, "CAD": 1.35},

"EUR": {"USD": 1.09, "GBP": 0.85, "JPY": 160.6, "CAD": 1.47},

"GBP": {"USD": 1.28, "EUR": 1.18, "JPY": 189.5, "CAD": 1.73},

"JPY": {"USD": 0.0068, "EUR": 0.0062, "GBP": 0.0053, "CAD": 0.0091},

"CAD": {"USD": 0.74, "EUR": 0.68, "GBP": 0.58, "JPY": 109.5}

}

base = base\_currency.upper()

target = target\_currency.upper()

if base not in exchange\_rates:

return f"Exchange rate data for {base} is not available."

if target not in exchange\_rates[base]:

return f"Exchange rate for {base} to {target} is not available."

rate = exchange\_rates[base][target]

return f"The current exchange rate from {base} to {target} is {rate}."

@tool

def get\_stock\_price(symbol: str) -> str:

"""Get the current stock price for a given ticker symbol."""

# Simulated stock data

stock\_prices = {

"AAPL": 172.62,

"MSFT": 389.47,

"GOOG": 147.03,

"AMZN": 178.22,

"META": 474.09,

"TSLA": 177.50,

"NVDA": 892.05

}

symbol = symbol.upper()

if symbol in stock\_prices:

return f"The current stock price of {symbol} is ${stock\_prices[symbol]:.2f}."

else:

return f"Stock price data for {symbol} is not available."

# Create an agent with financial tools

financial\_tools = [get\_exchange\_rate, get\_stock\_price]

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful financial assistant that can provide exchange rates and stock prices."),

("human", "{input}"),

])

llm = ChatOpenAI(model="gpt-4-turbo")

financial\_agent = create\_openai\_tools\_agent(llm, financial\_tools, prompt)

financial\_agent\_executor = AgentExecutor(agent=financial\_agent, tools=financial\_tools, verbose=True)

# Test the financial agent

response = financial\_agent\_executor.invoke({"input": "What's the current exchange rate from USD to EUR, and what's the stock price of NVIDIA?"})

print(response["output"])

**RAG and Vector Databases**

Retrieval Augmented Generation (RAG) with vector databases enables LLMs to access and use external knowledge.

**Setting Up a Vector Store**

import os

from langchain\_community.vectorstores import Chroma

from langchain\_openai import OpenAIEmbeddings

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

# Sample documents for our knowledge base

documents = [

"LangChain is a framework for developing applications powered by language models. It enables applications that are context-aware and reason about their environments.",

"Vector databases are specialized databases that store data as high-dimensional vectors and enable efficient similarity search.",

"Retrieval Augmented Generation (RAG) is a technique that combines retrieval-based and generation-based approaches for natural language processing tasks.",

"Large Language Models (LLMs) are AI systems trained on vast amounts of text data that can generate human-like text based on input prompts.",

"OpenAI's GPT models are a family of large language models that can generate human-like text based on the input they receive.",

"Embeddings are vector representations of text or other data that capture semantic meaning, allowing for operations like similarity comparison.",

"Prompt engineering is the process of designing and refining input prompts to get the desired output from language models.",

"Chain of Thought (CoT) prompting is a technique that encourages language models to break down complex reasoning tasks into intermediate steps.",

"Few-shot learning refers to the ability of a model to learn tasks from just a few examples, as opposed to requiring large datasets.",

"Fine-tuning is the process of further training a pre-trained model on a specific dataset to adapt it to particular tasks or domains."

]

# Create a directory for the database

os.makedirs("data/vectordb", exist\_ok=True)

# Create text splitter

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=1000,

chunk\_overlap=200

)

# Split documents into chunks

doc\_chunks = text\_splitter.create\_documents([doc for doc in documents])

# Initialize embeddings

embeddings = OpenAIEmbeddings()

# Create vector store

vectordb = Chroma.from\_documents(

documents=doc\_chunks,

embedding=embeddings,

persist\_directory="data/vectordb"

)

# Basic similarity search

query = "What is RAG?"

docs = vectordb.similarity\_search(query, k=2)

print(f"Retrieved {len(docs)} documents for query: '{query}'")

for i, doc in enumerate(docs):

print(f"Document {i+1}: {doc.page\_content[:100]}...")

**Building a Basic RAG Pipeline**

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_openai import ChatOpenAI

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_core.runnables import RunnablePassthrough

# Initialize LLM

llm = ChatOpenAI(model="gpt-4-turbo")

# Create a retriever from the vector store

retriever = vectordb.as\_retriever(search\_kwargs={"k": 3})

# Create a prompt template

template = """Answer the question based on the following context:

Context:

{context}

Question: {question}

Answer:"""

prompt = ChatPromptTemplate.from\_template(template)

# Create the RAG chain

rag\_chain = (

{"context": retriever, "question": RunnablePassthrough()}

| prompt

| llm

| StrOutputParser()

)

# Test the RAG chain

question = "Explain how RAG works and its benefits."

answer = rag\_chain.invoke(question)

print(f"Question: {question}")

print(f"Answer: {answer}")

**Advanced RAG with Metadata Filtering**

from langchain\_community.vectorstores import Chroma

from langchain\_openai import OpenAIEmbeddings

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

# Create a more structured knowledge base with metadata

documents\_with\_metadata = [

{"content": "LangChain is a framework for developing applications powered by language models. It enables applications that are context-aware and reason about their environments.", "metadata": {"category": "frameworks", "topic": "langchain"}},

{"content": "Vector databases are specialized databases that store data as high-dimensional vectors and enable efficient similarity search.", "metadata": {"category": "databases", "topic": "vector\_db"}},

{"content": "Retrieval Augmented Generation (RAG) is a technique that combines retrieval-based and generation-based approaches for natural language processing tasks.", "metadata": {"category": "techniques", "topic": "rag"}},

{"content": "Large Language Models (LLMs) are AI systems trained on vast amounts of text data that can generate human-like text based on input prompts.", "metadata": {"category": "models", "topic": "llm"}},

{"content": "OpenAI's GPT models are a family of large language models that can generate human-like text based on the input they receive.", "metadata": {"category": "models", "topic": "gpt"}},

{"content": "Embeddings are vector representations of text or other data that capture semantic meaning, allowing for operations like similarity comparison.", "metadata": {"category": "techniques", "topic": "embeddings"}},

{"content": "Prompt engineering is the process of designing and refining input prompts to get the desired output from language models.", "metadata": {"category": "techniques", "topic": "prompting"}},

{"content": "Chain of Thought (CoT) prompting is a technique that encourages language models to break down complex reasoning tasks into intermediate steps.", "metadata": {"category": "techniques", "topic": "cot"}},

{"content": "Few-shot learning refers to the ability of a model to learn tasks from just a few examples, as opposed to requiring large datasets.", "metadata": {"category": "techniques", "topic": "few\_shot"}},

{"content": "Fine-tuning is the process of further training a pre-trained model on a specific dataset to adapt it to particular tasks or domains.", "metadata": {"category": "techniques", "topic": "fine\_tuning"}}

]

# Create documents with metadata

structured\_docs = [

Document(page\_content=item["content"], metadata=item["metadata"])

for item in documents\_with\_metadata

]

# Create a new vector store with metadata

os.makedirs("data/vectordb\_meta", exist\_ok=True)

vectordb\_meta = Chroma.from\_documents(

documents=structured\_docs,

embedding=embeddings,

persist\_directory="data/vectordb\_meta"

)

# Create a filtered retriever

def get\_filtered\_retriever(category=None, topic=None):

filter\_dict = {}

if category:

filter\_dict["category"] = category

if topic:

filter\_dict["topic"] = topic

return vectordb\_meta.as\_retriever(

search\_kwargs={"k": 3, "filter": filter\_dict if filter\_dict else None}

)

# Example of retrieving only documents about techniques

techniques\_retriever = get\_filtered\_retriever(category="techniques")

techniques\_results = techniques\_retriever.invoke("What are different techniques used with LLMs?")

print("Techniques-only results:")

for doc in techniques\_results:

print(f"- {doc.page\_content[:100]}... [Topic: {doc.metadata['topic']}]")

# Create a specialized RAG chain with filtering

def create\_filtered\_rag\_chain(category=None, topic=None):

filtered\_retriever = get\_filtered\_retriever(category, topic)

template = """Answer the question based on the following context, which contains information about {filter\_desc}:

Context:

{context}

Question: {question}

Answer:"""

filter\_desc = ""

if category and topic:

filter\_desc = f"{topic} (category: {category})"

elif category:

filter\_desc = f"the {category} category"

elif topic:

filter\_desc = f"the topic of {topic}"

else:

filter\_desc = "various AI topics"

prompt = ChatPromptTemplate.from\_template(template)

return (

{"context": filtered\_retriever,

"question": RunnablePassthrough(),

"filter\_desc": lambda \_: filter\_desc}

| prompt

| llm

| StrOutputParser()

)

# Test the filtered RAG chain

techniques\_rag = create\_filtered\_rag\_chain(category="techniques")

answer = techniques\_rag.invoke("How can I improve LLM outputs?")

print(f"\nQuestion: How can I improve LLM outputs?")

print(f"Answer (filtered to techniques only): {answer}")

**Hybrid Search with Reranking**

from langchain\_community.retrievers import BM25Retriever

from langchain.retrievers import EnsembleRetriever

from langchain\_openai import OpenAIEmbeddings

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

# Create a larger corpus for hybrid search demonstration

additional\_docs = [

"Hybrid search combines multiple retrieval methods, such as keyword-based and semantic search, to improve results.",

"BM25 is a popular retrieval algorithm that ranks documents based on the frequency of query terms.",

"Cross-encoders are transformer models that take both the query and document as input to compute relevance scores.",

"Reranking is the process of taking an initial set of retrieved documents and reordering them based on additional criteria.",

"Bi-encoders encode queries and documents separately into vector representations for efficient similarity search.",

"Dense retrieval uses dense vector representations to capture semantic meaning for information retrieval.",

"Sparse retrieval methods like TF-IDF and BM25 focus on term frequency and are effective for keyword matching.",

"Semantic search aims to understand the intent and context of a query rather than just matching keywords.",

"Query expansion involves reformulating the original query to improve retrieval performance.",

"Embedding models are neural networks trained to map text to vector representations that capture semantic meaning."

]

# Combine all documents

all\_docs = documents + additional\_docs

# Split documents

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=1000,

chunk\_overlap=200

)

doc\_chunks = text\_splitter.create\_documents(all\_docs)

# Create vector store

os.makedirs("data/hybrid\_db", exist\_ok=True)

hybrid\_vectordb = Chroma.from\_documents(

documents=doc\_chunks,

embedding=embeddings,

persist\_directory="data/hybrid\_db"

)

# Create a BM25 (keyword) retriever

bm25\_retriever = BM25Retriever.from\_documents(doc\_chunks)

bm25\_retriever.k = 3 # Set to retrieve top 3 documents

# Create a vector retriever

vector\_retriever = hybrid\_vectordb.as\_retriever(search\_kwargs={"k": 3})

# Create an ensemble retriever

ensemble\_retriever = EnsembleRetriever(

retrievers=[bm25\_retriever, vector\_retriever],

weights=[0.5, 0.5]

)

# Build a simple reranking function

def rerank\_documents(query, docs, top\_n=3):

"""Rerank documents based on cosine similarity to query"""

query\_embedding = embeddings.embed\_query(query)

# Get embeddings for all documents

doc\_embeddings = [embeddings.embed\_query(doc.page\_content) for doc in docs]

# Calculate cosine similarity

from sklearn.metrics.pairwise import cosine\_similarity

import numpy as np

# Reshape query embedding for sklearn

query\_embedding = np.array(query\_embedding).reshape(1, -1)

doc\_embeddings = np.array(doc\_embeddings)

# Calculate similarities

similarities = cosine\_similarity(query\_embedding, doc\_embeddings)[0]

# Create pairs of (similarity, document)

scored\_docs = list(zip(similarities, docs))

# Sort by similarity (highest first)

scored\_docs.sort(key=lambda x: x[0], reverse=True)

# Return top N documents

return [doc for \_, doc in scored\_docs[:top\_n]]

# Create reranking retriever

def hybrid\_retriever\_with\_reranking(query):

# First get documents from ensemble retriever

initial\_docs = ensemble\_retriever.invoke(query)

# Rerank the documents

reranked\_docs = rerank\_documents(query, initial\_docs)

return reranked\_docs

# Test the hybrid search with reranking

query = "What is hybrid search and how does it compare to vector search?"

retrieved\_docs = hybrid\_retriever\_with\_reranking(query)

print(f"Retrieved {len(retrieved\_docs)} documents for query: '{query}'")

for i, doc in enumerate(retrieved\_docs):

print(f"Document {i+1}: {doc.page\_content}")

# Create a RAG chain with hybrid retrieval and reranking

hybrid\_rag\_template = """Answer the question based on the following context:

Context:

{context}

Question: {question}

Answer:"""

hybrid\_rag\_prompt = ChatPromptTemplate.from\_template(hybrid\_rag\_template)

def hybrid\_retriever\_chain(query):

docs = hybrid\_retriever\_with\_reranking(query)

context = "\n\n".join([doc.page\_content for doc in docs])

return {"context": context, "question": query}

hybrid\_rag\_chain = (

RunnablePassthrough()

| hybrid\_retriever\_chain

| hybrid\_rag\_prompt

| llm

| StrOutputParser()

)

# Test the hybrid RAG chain

hybrid\_answer = hybrid\_rag\_chain.invoke("What are the differences between dense and sparse retrieval?")

print(f"\nQuestion: What are the differences between dense and sparse retrieval?")

print(f"Answer: {hybrid\_answer}")

**Multi-Query RAG**

from langchain\_openai import ChatOpenAI

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_core.output\_parsers import CommaSeparatedListOutputParser

# Create a query generator using LLM

def generate\_queries(original\_query, num\_queries=3):

"""Generate multiple queries from an original query"""

query\_gen\_prompt = ChatPromptTemplate.from\_template(

"""You are an expert at generating alternative search queries.

Given an original query, generate {num} different search queries that would help retrieve relevant information.

The queries should be different from each other and cover various aspects of the original query.

Original Query: {query}

Output the queries as a comma-separated list."""

)

query\_gen\_chain = (

query\_gen\_prompt

| ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)

| CommaSeparatedListOutputParser()

)

queries = query\_gen\_chain.invoke({"query": original\_query, "num": num\_queries})

# Add the original query to the list

queries.append(original\_query)

return queries

# Create multi-query retriever

def multi\_query\_retriever(query, retriever, num\_queries=3):

"""Retrieve documents using multiple queries generated from the original query"""

# Generate multiple queries

queries = generate\_queries(query, num\_queries)

print(f"Generated queries: {queries}")

# Retrieve documents for each query

all\_docs = []

for q in queries:

docs = retriever.invoke(q)

all\_docs.extend(docs)

# Remove duplicates based on content

unique\_docs = []

seen\_content = set()

for doc in all\_docs:

if doc.page\_content not in seen\_content:

unique\_docs.append(doc)

seen\_content.add(doc.page\_content)

# Rerank documents

reranked\_docs = rerank\_documents(query, unique\_docs)

return reranked\_docs

# Create a RAG chain with multi-query retrieval

multi\_query\_rag\_template = """Answer the question based on the following context:

Context:

{context}

Question: {question}

Answer:"""

multi\_query\_rag\_prompt = ChatPromptTemplate.from\_template(multi\_query\_rag\_template)

def multi\_query\_retriever\_chain(query):

docs = multi\_query\_retriever(query, vector\_retriever, num\_queries=2)

context = "\n\n".join([doc.page\_content for doc in docs])

return {"context": context, "question": query}

multi\_query\_rag\_chain = (

RunnablePassthrough()

| multi\_query\_retriever\_chain

| multi\_query\_rag\_prompt

| llm

| StrOutputParser()

)

# Test the multi-query RAG chain

multi\_query\_answer = multi\_query\_rag\_chain.invoke("What are embeddings and how are they used in AI?")

print(f"\nQuestion: What are embeddings and how are they used in AI?")

print(f"Answer: {multi\_query\_answer}")

**RAG with Self-Query**

from langchain.retrievers.self\_query.base import SelfQueryRetriever

from langchain\_core.pydantic\_v1 import BaseModel, Field

from langchain\_openai import OpenAIEmbeddings

from langchain.chains.query\_constructor.base import AttributeInfo

# Define the metadata schema

class DocumentMetadata(BaseModel):

category: str = Field(description="The category of the document (e.g., frameworks, databases, techniques, models)")

topic: str = Field(description="The specific topic of the document (e.g., langchain, vector\_db, rag, llm)")

# Define metadata attributes

metadata\_attributes = [

AttributeInfo(

name="category",

description="The category of the document",

type="string",

),

AttributeInfo(

name="topic",

description="The specific topic of the document",

type="string",

),

]

# Create a self-query retriever

self\_query\_retriever = SelfQueryRetriever.from\_llm(

llm=ChatOpenAI(model="gpt-4-turbo"),

vectorstore=vectordb\_meta,

document\_contents="Information about various AI and language model concepts",

metadata\_field\_info=metadata\_attributes,

)

# Test the self-query retriever

query = "Find information about techniques related to prompt engineering"

results = self\_query\_retriever.invoke(query)

print(f"Self-query results for: '{query}'")

for doc in results:

print(f"- {doc.page\_content[:100]}... [Category: {doc.metadata['category']}, Topic: {doc.metadata['topic']}]")

# Create RAG chain with self-query

self\_query\_rag\_template = """Answer the question based on the following context:

Context:

{context}

Question: {question}

Answer:"""

self\_query\_rag\_prompt = ChatPromptTemplate.from\_template(self\_query\_rag\_template)

def self\_query\_retriever\_chain(query):

docs = self\_query\_retriever.invoke(query)

context = "\n\n".join([doc.page\_content for doc in docs])

return {"context": context, "question": query}

self\_query\_rag\_chain = (

RunnablePassthrough()

| self\_query\_retriever\_chain

| self\_query\_rag\_prompt

| llm

| StrOutputParser()

)

# Test the self-query RAG chain

self\_query\_answer = self\_query\_rag\_chain.invoke("What techniques are used for fine-tuning language models?")

print(f"\nQuestion: What techniques are used for fine-tuning language models?")

print(f"Answer: {self\_query\_answer}")

**Creating Powerful Agents**

Agents can use LLMs to determine which actions to take and in what order to achieve a goal.

**Building a Basic Agent**

from langchain.agents import create\_openai\_functions\_agent, AgentExecutor

from langchain\_openai import ChatOpenAI

from langchain\_core.tools import tool

from langchain\_core.prompts import ChatPromptTemplate

# Define some basic tools

@tool

def search\_weather(location: str) -> str:

"""Search for the current weather in a specific location."""

# Simulated weather data

weather\_data = {

"New York": "72°F, Sunny",

"London": "65°F, Cloudy",

"Tokyo": "80°F, Clear",

"Sydney": "70°F, Partly Cloudy"

}

if location in weather\_data:

return f"The weather in {location} is currently {weather\_data[location]}."

else:

return f"Weather data for {location} is not available."

@tool

def calculate(expression: str) -> str:

"""Calculate the result of a mathematical expression."""

try:

result = eval(expression)

return f"The result of {expression} is {result}"

except Exception as e:

return f"Error calculating {expression}: {str(e)}"

# Create the agent

llm = ChatOpenAI(model="gpt-4-turbo")

tools = [search\_weather, calculate]

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful assistant who can use tools to answer questions."),

("human", "{input}"),

])

agent = create\_openai\_functions\_agent(llm, tools, prompt)

agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# Test the agent

response = agent\_executor.invoke({"input": "What's the weather in Tokyo, and what's 25 \* 16?"})

print(response["output"])

**Agent with Memory**

from langchain.agents import create\_openai\_functions\_agent, AgentExecutor

from langchain\_openai import ChatOpenAI

from langchain\_core.tools import tool

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder

from langchain.memory import ConversationBufferMemory

# Define tools

@tool

def search\_weather(location: str) -> str:

"""Search for the current weather in a specific location."""

# Simulated weather data

weather\_data = {

"New York": "72°F, Sunny",

"London": "65°F, Cloudy",

"Tokyo": "80°F, Clear",

"Sydney": "70°F, Partly Cloudy"

}

if location in weather\_data:

return f"The weather in {location} is currently {weather\_data[location]}."

else:

return f"Weather data for {location} is not available."

@tool

def search\_restaurants(location: str) -> str:

"""Search for popular restaurants in a specific location."""

# Simulated restaurant data

restaurant\_data = {

"New York": ["The Gramercy Tavern", "Le Bernardin", "Peter Luger Steak House"],

"London": ["The Ivy", "Dishoom", "Gordon Ramsay Restaurant"],

"Tokyo": ["Sukiyabashi Jiro", "Sushi Saito", "Den"],

"Sydney": ["Quay", "Tetsuya's", "Sixpenny"]

}

if location in restaurant\_data:

restaurants = ", ".join(restaurant\_data[location])

return f"Popular restaurants in {location} include: {restaurants}."

else:

return f"Restaurant data for {location} is not available."

# Create memory

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

# Create the agent with memory

llm = ChatOpenAI(model="gpt-4-turbo")

tools = [search\_weather, search\_restaurants]

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful travel assistant who remembers previous conversations."),

MessagesPlaceholder(variable\_name="chat\_history"),

("human", "{input}"),

])

agent = create\_openai\_functions\_agent(llm, tools, prompt)

agent\_executor = AgentExecutor(

agent=agent,

tools=tools,

memory=memory,

verbose=True

)

# Test the agent with memory

response = agent\_executor.invoke({"input": "What's the weather in Tokyo?"})

print("Response 1:", response["output"])

response = agent\_executor.invoke({"input": "Can you recommend some restaurants there?"})

print("Response 2:", response["output"])

response = agent\_executor.invoke({"input": "What was the weather there again?"})

print("Response 3:", response["output"])

**Multi-Step Planning Agent**

from langchain.agents import create\_openai\_functions\_agent, AgentExecutor

from langchain\_openai import ChatOpenAI

from langchain\_core.tools import tool

from langchain\_core.prompts import ChatPromptTemplate

from typing import List, Dict, Any

# Task tracking tools

tasks = {}

task\_id\_counter = 0

@tool

def create\_task(description: str) -> str:

"""Create a new task with the given description."""

global task\_id\_counter

task\_id = task\_id\_counter

task\_id\_counter += 1

tasks[task\_id] = {"description": description, "completed": False}

return f"Task created with ID {task\_id}: {description}"

@tool

def list\_tasks() -> str:

"""List all tasks and their status."""

if not tasks:

return "No tasks found."

task\_list = []

for task\_id, task in tasks.items():

status = "✓" if task["completed"] else "□"

task\_list.append(f"Task {task\_id}: {status} {task['description']}")

return "\n".join(task\_list)

@tool

def complete\_task(task\_id: int) -> str:

"""Mark a task as completed."""

if task\_id not in tasks:

return f"Task {task\_id} not found."

tasks[task\_id]["completed"] = True

return f"Task {task\_id} marked as completed: {tasks[task\_id]['description']}"

@tool

def search\_flight(origin: str, destination: str) -> str:

"""Search for flights between two cities."""

# Simulated flight data

flight\_data = {

("New York", "London"): ["BA123 - $800", "VS456 - $750", "AA789 - $830"],

("London", "Tokyo"): ["JL321 - $1200", "BA654 - $1150", "NH987 - $1300"],

("Tokyo", "Sydney"): ["QF123 - $900", "JL456 - $950", "NH789 - $920"]

}

if (origin, destination) in flight\_data:

flights = flight\_data[(origin, destination)]

return f"Available flights from {origin} to {destination}:\n" + "\n".join(flights)

else:

return f"No direct flights found from {origin} to {destination}."

@tool

def search\_hotel(location: str) -> str:

"""Search for hotels in a specific location."""

# Simulated hotel data

hotel\_data = {

"New York": ["Plaza Hotel - $400/night", "Marriott - $300/night", "Hilton - $350/night"],

"London": ["The Savoy - £350/night", "Ritz London - £400/night", "Claridge's - £380/night"],

"Tokyo": ["Park Hyatt - ¥40,000/night", "Mandarin Oriental - ¥35,000/night", "Imperial Hotel - ¥30,000/night"],

"Sydney": ["Four Seasons - A$400/night", "Shangri-La - A$350/night", "Hilton - A$300/night"]

}

if location in hotel\_data:

hotels = hotel\_data[location]

return f"Available hotels in {location}:\n" + "\n".join(hotels)

else:

return f"No hotel data available for {location}."

# Create the agent with planning capabilities

llm = ChatOpenAI(model="gpt-4-turbo")

planning\_tools = [create\_task, list\_tasks, complete\_task, search\_flight, search\_hotel]

planning\_prompt = ChatPromptTemplate.from\_messages([

("system", """You are a travel planning assistant that helps users plan trips step by step.

When asked about planning a trip, break it down into individual tasks and use the task management tools.

Always think about the logical sequence of tasks needed for planning a trip.

Use the search tools to find relevant information about flights and hotels.

After each response, consider what next steps might be helpful for the user."""),

("human", "{input}"),

])

planning\_agent = create\_openai\_functions\_agent(llm, planning\_tools, planning\_prompt)

planning\_executor = AgentExecutor(

agent=planning\_agent,

tools=planning\_tools,

verbose=True,

max\_iterations=10 # Limit the number of iterations for safety

)

# Test the planning agent

response = planning\_executor.invoke({"input": "I want to plan a trip from New York to Tokyo next month."})

print(response["output"])

# Continue planning

response = planning\_executor.invoke({"input": "What's next in the planning process?"})

print(response["output"])

**Tool-Calling Agent**

from langchain.agents.format\_scratchpad import format\_to\_openai\_function\_messages

from langchain.agents.output\_parsers import OpenAIFunctionsAgentOutputParser

from langchain\_openai import ChatOpenAI

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder

from langchain\_core.messages import AIMessage, HumanMessage

# Enhanced tools with more specific parameters

@tool

def search\_flights(origin: str, destination: str, date: str) -> str:

"""

Search for flights between two cities on a specific date.

Args:

origin: The departure city (e.g., "New York", "London")

destination: The arrival city (e.g., "Tokyo", "Paris")

date: The date of travel in YYYY-MM-DD format

"""

# Simulated flight data

flight\_data = {

("New York", "London", "2025-05-15"): ["BA123 - $800 (10:00 AM - 10:00 PM)", "VS456 - $750 (2:00 PM - 2:00 AM)"],

("London", "Tokyo", "2025-05-20"): ["JL321 - $1200 (9:00 AM - 6:00 AM)", "BA654 - $1150 (11:00 AM - 8:00 AM)"],

("Tokyo", "Sydney", "2025-05-25"): ["QF123 - $900 (1:00 PM - 10:00 PM)", "JL456 - $950 (3:00 PM - 11:00 PM)"]

}

key = (origin, destination, date)

if key in flight\_data:

flights = flight\_data[key]

return f"Available flights from {origin} to {destination} on {date}:\n" + "\n".join(flights)

else:

return f"No flights found from {origin} to {destination} on {date}."

@tool

def book\_hotel(location: str, check\_in: str, check\_out: str, guests: int) -> str:

"""

Book a hotel in a specific location.

Args:

location: The city where the hotel is located

check\_in: Check-in date in YYYY-MM-DD format

check\_out: Check-out date in YYYY-MM-DD format

guests: Number of guests

"""

# Simulated hotel booking response

return f"Hotel booked in {location} for {guests} guests from {check\_in} to {check\_out}. Confirmation #HOTEL12345."

@tool

def rent\_car(location: str, pick\_up: str, drop\_off: str, car\_type: str) -> str:

"""

Rent a car in a specific location.

Args:

location: The city where the car will be rented

pick\_up: Pick-up date in YYYY-MM-DD format

drop\_off: Drop-off date in YYYY-MM-DD format

car\_type: Type of car (e.g., "economy", "luxury", "SUV")

"""

# Simulated car rental response

return f"{car\_type.capitalize()} car rented in {location} from {pick\_up} to