Lecture 1: Introduction to LangChain & Agentic Thinking

🎯 Learning Objectives

- Understand what LangChain is and its role in the LLM ecosystem.
- Define agentic AI and understand its principles.
- Explore basic use cases of LangChain.

📄 What is LangChain?

LangChain is a framework designed to help developers build applications powered by large language models (LLMs). It simplifies and structures interactions between LLMs and various components like prompts, memory, APIs, and external tools.

🧠 What is Agentic AI?

Agentic AI refers to systems that exhibit autonomy, make decisions, and use tools or resources to accomplish tasks. These agents:

- Understand tasks.
- Reason through intermediate steps.
- Take actions using tools or APIs.
- Learn or adapt through memory.

LangChain facilitates building such systems by managing complexity and orchestration.

🛠️ LangChain Architecture Overview

- **LLMs**: Interfaces to models like GPT-4, Claude, or Mistral.
- **Prompts**: Templates and input formatting.
- **Chains**: A sequence of calls (e.g., prompt → LLM → output).
- **Memory**: Retaining conversation context.
- **Agents**: LLMs that choose actions (tools, reasoning steps).
- **Tools**: External functionality (calculators, APIs, DBs).
- **Callbacks**: Monitoring and logging actions.

📌 Use Cases

- Chatbots with memory
- Question answering over documents
- Autonomous agents that browse the web
- Workflow automation (multi-step tasks)

✍️ Hands-On: Your First LangChain App

Install LangChain:

```
pip install langchain openai
```

Set up a basic chatbot:

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain

llm = ChatOpenAI(temperature=0)

prompt = ChatPromptTemplate.from_template("You are a helpful assistant. Answer:
{question}")
chain = LLMChain(llm=llm, prompt=prompt)

response = chain.run({"question": "What is LangChain?"})
print(response)
```

☑ Summary

- LangChain provides abstractions over LLMs to build intelligent, agentic apps.
- You learned the core ideas behind LangChain and agentic AI.
- You built a simple chatbot using `LLMChain`.

🛍 Assignment

1. Modify the chatbot to take a user's name and personalize responses.
2. Try using different `temperature` values in `ChatOpenAI`.
3. Read the LangChain docs to preview what's coming next.

Lecture 2: LangChain Primitives – Models, Prompts, and Chains

---

🎯 Learning Objectives

- Understand and use core LangChain primitives: LLMs, Prompts, and Chains.
- Learn how to structure inputs and outputs for LLM-based applications.
- Build and run an `LLMChain` with custom prompts.

Core Primitives

LangChain simplifies working with LLMs by introducing key abstractions:

LLMs

These are interfaces to large language models (e.g., OpenAI, Anthropic).

```
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(temperature=0.7)
```

Prompts

A `PromptTemplate` defines how inputs are formatted for the model.

```
from langchain.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_template("Translate the following to French:
{text}")
```

Chains

`LLMChain` connects prompts and models together.

```
from langchain.chains import LLMChain

chain = LLMChain(llm=llm, prompt=prompt)
response = chain.run({"text": "Hello, how are you?"})
print(response)
```

How It Works

LLMChain Flow:

1. Takes input variables (`text`)
2. Formats the input using `PromptTemplate`
3. Sends to the `ChatOpenAI` model
4. Returns formatted output

Why Use Chains?

Chains allow you to:

- Encapsulate multi-step logic.
- Add memory and history.
- Build modular, testable components.

✨ Customizing Prompts

Prompts can include multiple variables:

```
prompt = ChatPromptTemplate.from_template("Write a {tone} email to {person} about
{topic}.")
inputs = {
    "tone": "formal",
    "person": "Dr. Smith",
    "topic": "the project deadline"
}
response = chain.run(inputs)
```

🩹 Hands-On Exercise

Goal: Build a multi-use translator using `LLMChain`.

```
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI()

prompt = ChatPromptTemplate.from_template("Translate this sentence to {language}:
{sentence}")
chain = LLMChain(llm=llm, prompt=prompt)

response = chain.run({"language": "Spanish", "sentence": "Good morning!"})
print(response)
```

🧠 Summary

- You learned the three key LangChain primitives: LLMs, Prompts, and Chains.
- You built custom chains using formatted prompts.
- You understand how to modularize language model interactions.

📚 Assignment

1. Modify your prompt to include tone/style of translation.
2. Create an `LLMChain` that turns a list of bullet points into a formal paragraph.

3. Experiment with chaining multiple `LLMChain` steps.

Lecture 3: Memory in LangChain

🎯 Learning Objectives

- Understand the concept of memory in LangChain.
- Use built-in memory classes to maintain conversation state.
- Learn when and how to use memory effectively in applications.

🧠 What is Memory in LangChain?

Memory allows your application to **retain state across calls**, making conversations feel more natural and context-aware. It stores previous inputs, outputs, or summaries of interactions.

🧩 Types of Memory

1. ConversationBufferMemory

Stores a **raw buffer** of messages from the conversation.

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
```

2. ConversationSummaryMemory

Stores a **summary** of the conversation using an LLM.

```
from langchain.memory import ConversationSummaryMemory
from langchain.chat_models import ChatOpenAI

memory = ConversationSummaryMemory(llm=ChatOpenAI())
```

3. ConversationBufferWindowMemory

Keeps a **window of recent messages**, useful when token limits matter.

```
from langchain.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(k=3)
```

Using Memory with Chains

You can plug memory into a `ConversationChain`:

```
from langchain.chains import ConversationChain
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI()
memory = ConversationBufferMemory()

chain = ConversationChain(llm=llm, memory=memory, verbose=True)

response = chain.run("Hello, my name is Alice.")
response = chain.run("What is my name?")
print(response)
```

Behind the Scenes

Memory stores:

- `chat_history`: a string of previous user/AI messages.
- Automatically adds history to the prompt input.
- Can be serialized to disk or external DBs for persistence.

Hands-On Exercise

Goal: Create a conversational bot that remembers user preferences.

```
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI()
memory = ConversationBufferMemory()

chain = ConversationChain(llm=llm, memory=memory)

print(chain.run("Hi, I'm planning a trip to Japan."))
print(chain.run("What did I say about my trip?"))
```

Advanced: Custom Memory Classes

You can implement your own memory by subclassing `BaseMemory`.

Use cases:

- Store data in Redis or Pinecone.
- Track multiple sessions.
- Add vector search to long-term memory.

☑ Summary

- Memory adds context to conversations.
- LangChain offers several memory types for different use cases.
- You can easily attach memory to chains or agents.

Assignment

1. Try out `ConversationSummaryMemory` and compare results.
2. Use `ConversationBufferWindowMemory` to limit the context window.
3. Add memory to a translation or QA bot you previously built.

Lecture 4: Document Q&A and Retrieval-Augmented Generation (RAG)

Learning Objectives

- Understand the concept of Retrieval-Augmented Generation (RAG).
- Use LangChain to build a Document Q&A system.
- Learn how to split documents, embed them, and query using vector stores.

What is RAG?

Retrieval-Augmented Generation (RAG) is a technique that enhances LLM responses by providing relevant external knowledge from documents.

Instead of relying only on the model's internal knowledge, we:

1. Split and index documents.
2. Embed them in a vector store.
3. Retrieve the most relevant chunks for a given query.
4. Feed them into the LLM as context.

Core Components

1. Document Loaders

To load files (PDF, TXT, CSV, etc.).

```
from langchain.document_loaders import TextLoader

loader = TextLoader("example.txt")
documents = loader.load()
```

2. Text Splitters

Break long documents into manageable chunks.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
docs = text_splitter.split_documents(documents)
```

3. Embeddings and Vector Stores

Convert text chunks into embeddings and store them in a searchable format.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma

embedding = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(docs, embedding=embedding)
```

🔍 RetrievalQA Chain

Combines a retriever and an LLM to answer questions based on your documents.

```
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

retriever = vectorstore.as_retriever()
qa_chain = RetrievalQA.from_chain_type(llm=ChatOpenAI(), retriever=retriever)

response = qa_chain.run("What is the document about?")
print(response)
```

🧪 Hands-On Exercise

Goal: Create a basic Q&A system over your own text file.

```python
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

# Load and split the document
loader = TextLoader("example.txt")
documents = loader.load()
splitter = RecursiveCharacterTextSplitter(chunk_size=300, chunk_overlap=30)
docs = splitter.split_documents(documents)

# Embed and store
embedding = OpenAIEmbeddings()
vectordb = Chroma.from_documents(docs, embedding=embedding)

# QA chain
retriever = vectordb.as_retriever()
qa = RetrievalQA.from_chain_type(llm=ChatOpenAI(), retriever=retriever)

# Ask questions
print(qa.run("Summarize the document."))
```

🧠 Summary

- RAG allows you to build LLM apps that consult external data.
- LangChain simplifies the document loading, splitting, embedding, and querying process.
- You can now build powerful, context-aware Q&A systems.

📚 Assignment

1. Load a longer document (e.g., PDF or Markdown) and ask detailed questions.
2. Use `FAISS` or `Pinecone` instead of Chroma.
3. Try different chunk sizes and observe retrieval quality.

Lecture 5: Tools and Agents in LangChain

🎯 Learning Objectives

- Understand what agents are and how they work in LangChain.
- Learn about the `Tool` abstraction and how to build agents that use them.
- Build a basic agent that uses tools to answer questions.

🧠 What is an Agent?

Agents use an LLM to **decide what actions to take**, using tools to complete tasks. This allows dynamic decision-making rather than following a static chain.

Agents can:

- Choose from multiple tools.
- Perform reasoning (e.g., ReAct: Reason + Act).
- Chain multiple steps to reach a solution.

🛠️ Tools

Tools are simple Python functions exposed to the agent.

```
from langchain.agents import tool

@tool
def add(a: int, b: int) -> int:
    "Adds two numbers"
    return a + b
```

🚀 Setting Up an Agent

LangChain provides different agent types. A common starting point is the **Zero-Shot ReAct Agent**.

```
from langchain.agents import initialize_agent, AgentType
from langchain.chat_models import ChatOpenAI
from langchain.agents import load_tools

llm = ChatOpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm=llm)

agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

response = agent.run("What is the square root of the population of France?")
print(response)
```

🔧 Custom Tool Example

```
from langchain.tools import Tool

def get_weather(location: str) -> str:
    return f"The weather in {location} is sunny."

weather_tool = Tool(
    name="WeatherTool",
    func=get_weather,
    description="Returns the weather for a given location."
)
```

🤖 Agent Decision Loop

Typical agent behavior (ReAct):

1. Observes the input question.
2. Chooses a tool.
3. Takes action using the tool.
4. Observes result and repeats if needed.
5. Returns final answer.

🧪 Hands-On Exercise

Goal: Build an agent with a calculator and custom weather tool.

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent, AgentType
from langchain.tools import Tool

def get_weather(city: str) -> str:
    return f"The weather in {city} is 25°C and clear."

weather_tool = Tool(
    name="WeatherTool",
    func=get_weather,
    description="Returns current weather for a city"
)

llm = ChatOpenAI()
tools = [weather_tool]
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

print(agent.run("What is the weather in Tokyo?"))
```

☑ Summary

- Agents allow dynamic decision-making using LLMs and tools.
- Tools extend what the LLM can do (math, search, custom APIs).
- You built an agent using built-in and custom tools.

Assignment

1. Create a new tool that fetches current time or date.
2. Build an agent that combines search and weather tools.
3. Explore how agents behave with ambiguous or multi-step tasks.

Lecture 6: Building Custom Tools and Toolkits in LangChain

Learning Objectives

- Understand how to build and register your own tools.
- Learn how to group tools into toolkits.
- Use LangChain Expression Language (LCEL) for tool orchestration.

What is a Tool?

A **Tool** in LangChain is a callable function with a name and description, allowing LLMs to choose and execute actions during runtime.

Each tool should:

- Accept a single string input (or use custom parsing).
- Return a string output.
- Be stateless (for simplicity and reliability).

Defining a Custom Tool

```
from langchain.tools import Tool

def search_knowledge_base(query: str) -> str:
    return f"Search results for: {query}"

search_tool = Tool(
    name="KnowledgeBaseSearch",
    func=search_knowledge_base,
```

```
    description="Searches the internal knowledge base for relevant information."
)
```

🧰 Grouping Tools into Toolkits

A **Toolkit** is a collection of tools, typically related to a domain (e.g., calendar, finance, etc.).

```
from langchain.agents import Tool, AgentExecutor
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent, AgentType

tools = [search_tool, ...]  # Add other tools as needed

llm = ChatOpenAI()
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)
```

🔀 LangChain Expression Language (LCEL)

LCEL is a declarative way to compose chains and tools using pipe syntax.

```
from langchain_core.runnables import RunnableLambda

def reverse_string(s: str) -> str:
    return s[::-1]

reverse_tool = RunnableLambda(reverse_string)
result = reverse_tool.invoke("Hello")
print(result)  # Output: "olleH"
```

You can also chain operations:

```
from langchain_core.runnables import RunnablePassthrough

chain = RunnablePassthrough.assign(reversed=reverse_tool)
output = chain.invoke("LangChain")
print(output)  # Output: {"reversed": "niahCgnaL"}
```

Hands-On Exercise

Goal: Create a custom date and greeting toolkit.

```
from datetime import datetime

def get_date(_: str) -> str:
    return f"Today's date is {datetime.now().strftime('%Y-%m-%d')}."

def greet(name: str) -> str:
    return f"Hello, {name}!"

date_tool = Tool(name="GetDate", func=get_date, description="Returns today's
date.")
greet_tool = Tool(name="GreetUser", func=greet, description="Greets the user by
name.")

tools = [date_tool, greet_tool]

llm = ChatOpenAI()
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
verbose=True)

print(agent.run("What is today's date and say hi to Alice."))
```

Summary

- Tools can be built from any callable function.
- Toolkits help organize related tools.
- LCEL enables structured, pipeline-based tool orchestration.

Assignment

1. Create a toolkit with at least 3 custom tools (e.g., calculator, time zone converter, emoji generator).
2. Use LCEL to build a processing pipeline from user input to formatted output.
3. Experiment with adding error handling inside tools.

Lecture 7: LangGraph for Multi-Step, Multi-Agent Workflows

Learning Objectives

- Understand the purpose of LangGraph.
- Build stateful, branching, multi-step flows using LangGraph.
- Model decision logic and multi-agent workflows.

🔁 What is LangGraph?

LangGraph is an extension of LangChain for building **state machines and agent workflows** using a graph-based approach.

Use cases:

- Multi-agent collaboration
- Conditional branching
- Step-by-step workflows
- Long-lived processes

🔧 Core Concepts

- **Nodes**: Steps in the graph (LLM calls, tools, decisions).
- **Edges**: Transitions between nodes based on output or logic.
- **State**: Persisted data passed along the graph.
- **GraphExecutor**: Orchestrates execution through the graph.

✨ Example: Simple Decision Flow

```
from langgraph.graph import StateGraph
from langchain_core.runnables import RunnableLambda

def decide_route(state):
    if "error" in state["input"].lower():
        return "handle_error"
    return "process_input"

def process_input(state):
    return {"result": f"Processed: {state['input']}"}

def handle_error(state):
    return {"result": "An error occurred."}

builder = StateGraph()
builder.add_node("router", RunnableLambda(decide_route))
builder.add_node("process_input", RunnableLambda(process_input))
builder.add_node("handle_error", RunnableLambda(handle_error))

builder.set_entry_point("router")
builder.add_conditional_edges("router", lambda x: x, {
    "process_input": "process_input",
    "handle_error": "handle_error"
})
builder.set_finish_point("process_input")
builder.set_finish_point("handle_error")
```

```
graph = builder.compile()
result = graph.invoke({"input": "show me data"})
print(result)
```

Multi-Agent Collaboration

LangGraph allows multiple agents to pass state and decisions between each other.

Example: Debate Between Agents

1. **Agent A** makes a statement.
2. **Agent B** responds.
3. **Moderator** summarizes or decides the winner.

Each step is a node in the graph.

Benefits of LangGraph

- Built-in support for persistence
- Easily model loops and retries
- Clear logic structure for debugging
- Support for long-running applications

Hands-On Exercise

Goal: Build a 2-agent debate with a moderator.

```
from langchain_core.runnables import RunnableLambda
from langgraph.graph import StateGraph

def agent_a(state):
    return {"a": "Cats are better than dogs."}

def agent_b(state):
    return {"b": "Dogs are better because they are loyal."}

def moderator(state):
    return {"result": f"Agent A said: {state['a']}, Agent B replied:
{state['b']}."}

graph = StateGraph()
graph.add_node("AgentA", RunnableLambda(agent_a))
graph.add_node("AgentB", RunnableLambda(agent_b))
graph.add_node("Moderator", RunnableLambda(moderator))

graph.set_entry_point("AgentA")
graph.add_edge("AgentA", "AgentB")
```

```
graph.add_edge("AgentB", "Moderator")
graph.set_finish_point("Moderator")

compiled = graph.compile()
result = compiled.invoke({})
print(result)
```

☑ Summary

- LangGraph enables structured, stateful, branching workflows.
- You can model decision logic and multi-agent processes.
- Reusable for long-running, collaborative, or dynamic tasks.

🎑 Assignment

1. Extend the debate graph to include a second round.
2. Create a customer support flow with nodes: intake → categorize → respond → follow-up.
3. Add error handling logic and retry mechanism to a node.

Lecture 8: Web Research & Browsing Agents in LangChain

🎯 Learning Objectives

- Enable agents to use the internet for live research.
- Use web-based tools like SerpAPI and RequestsTool.
- Build agents that search, summarize, and reason with web content.

🌍 Why Web Access?

LLMs are limited by their training data. Adding web tools allows your agent to:

- Access real-time information (news, weather, stock prices).
- Explore unknown or rare knowledge.
- Conduct live research or summarization.

🔧 Web Tools in LangChain

1. SerpAPI Tool

Provides access to Google search results.

```
from langchain.tools import SerpAPIWrapper
from langchain.agents import Tool

search = SerpAPIWrapper()
search_tool = Tool(
    name="Search",
    func=search.run,
    description="Useful for answering questions about current events or unknown
facts."
)
```

⚠ Requires SerpAPI key: `SERPAPI_API_KEY=your_api_key`

---

2. Requests Tool

Performs web scraping or API calls.

```
from langchain.tools import RequestsGetTool

requests_tool = RequestsGetTool()
```

Using Web Tools with Agents

```
from langchain.agents import initialize_agent, AgentType
from langchain.chat_models import ChatOpenAI

tools = [search_tool, requests_tool]
llm = ChatOpenAI()

agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

response = agent.run("What is the latest news about AI regulations?")
print(response)
```

Tool Comparison

Tool	Use Case	Notes
SerpAPI	Search & summary	Needs API key

Tool	Use Case	Notes
RequestsTool	Raw page scraping/API call	Might require HTML parsing
BingSearchTool	Microsoft-powered search	Alternative to SerpAPI

🧪 Hands-On Exercise

Goal: Build a research agent for summarizing news.

```
from langchain.tools import SerpAPIWrapper, Tool
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent, AgentType

search = SerpAPIWrapper()
search_tool = Tool(
    name="Search",
    func=search.run,
    description="Searches the web for real-time data."
)

llm = ChatOpenAI()
agent = initialize_agent([search_tool], llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

question = "Summarize today's most important tech headline."
print(agent.run(question))
```

☑️ Summary

- LangChain agents can browse the web using tools.
- SerpAPI and RequestsTool extend agent capabilities beyond static knowledge.
- You built a research-capable agent.

📚 Assignment

1. Use RequestsTool to fetch and summarize the contents of a news article.
2. Create a "daily briefing" agent: fetch headlines, weather, and stock prices.
3. Add error handling when web search fails or returns nothing.

Lecture 9: LangChain in Production – Logging, Callbacks, and LangSmith

---

🎯 Learning Objectives

- Understand how to monitor and debug LangChain applications.
- Use callbacks and tracing for logging and analytics.
- Explore LangSmith for visualizing chain and agent runs.

🚀 Why Production Observability?

When deploying LangChain apps, you need visibility into:

- Prompt inputs/outputs
- Tool usage and agent steps
- Token usage and costs
- Latency and errors

LangChain provides **callbacks** and **LangSmith** for this purpose.

🔧 Callbacks in LangChain

Callbacks let you hook into chain/agent execution events.

Built-in Callbacks

- `StdOutCallbackHandler`: Prints steps to stdout.
- `TracingCallbackHandler`: Sends data to LangSmith for visualization.
- Custom callback handlers for logging to files, databases, or APM tools.

Example: Using StdOutCallbackHandler

```
from langchain.callbacks import StdOutCallbackHandler
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
from langchain.prompts import ChatPromptTemplate

llm = ChatOpenAI(callbacks=[StdOutCallbackHandler()])
prompt = ChatPromptTemplate.from_template("What is the capital of {country}?")
chain = LLMChain(llm=llm, prompt=prompt)

response = chain.run({"country": "Germany"})
print(response)
```

📊 LangSmith: Cloud Platform for Debugging & Monitoring

LangSmith is a hosted tool that helps:

- Log all runs and traces.
- Visualize chain and agent paths.
- Track prompt usage, latency, token cost.

- Compare versions of chains/prompts.

Setup LangSmith

1. Sign up at https://smith.langchain.com
2. Set environment variables:

```
export LANGCHAIN_API_KEY="your-key"
export LANGCHAIN_TRACING_V2="true"
export LANGCHAIN_PROJECT="your-project-name"
```

3. Enable tracing in your app and run as normal.

Visualizing Traces

Each chain/agent run shows:

- Input → Output
- Intermediate steps
- Tool invocations
- Errors and retries

LangSmith provides UI for:

- Filtering and searching logs
- Sharing trace URLs with team
- Analyzing performance over time

🩹 Hands-On Exercise

Goal: Track token usage and visualize chain runs.

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
from langchain.prompts import ChatPromptTemplate
import os

# Enable LangSmith
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = "your-api-key"
os.environ["LANGCHAIN_PROJECT"] = "LangChainLectureDemo"

llm = ChatOpenAI()
prompt = ChatPromptTemplate.from_template("Explain the concept of {topic} to a 5-
year-old.")
chain = LLMChain(llm=llm, prompt=prompt)
```

```
print(chain.run({"topic": "black holes"}))
```

☑ Summary

- Use callbacks to log and debug LangChain apps.
- LangSmith provides deep tracing and analysis tools.
- In production, observability is critical for reliability and cost tracking.

Assignment

1. Use LangSmith to track 3 different LLM chains and compare traces.
2. Add error handling to a tool and log the exception flow.
3. Explore using LangSmith for team collaboration and debugging.

Lecture 10: Capstone – Building an End-to-End Autonomous Agent

---

Learning Objectives

- Combine all LangChain concepts: LLMs, prompts, memory, tools, agents, RAG, and tracing.
- Build a complete, functional AI assistant.
- Understand deployment considerations and next steps.

Capstone Architecture Overview

Your final project will involve:

- LLM for natural language understanding and response
- Tools for accessing external functionality (e.g., search, weather, calculator)
- Memory for conversation context
- Retrieval-based question answering (RAG)
- Agent orchestration
- LangSmith tracing for monitoring

Tech Stack Summary

Component	LangChain Feature
Core logic	`LLMChain`, `AgentExecutor`
Context	`ConversationBufferMemory`
Knowledge base	`Chroma` + `OpenAIEmbeddings`

Component	LangChain Feature
Tools	Custom and built-in tools
Logging	`LangSmith` and Callbacks

🧠 Planning the Assistant

Features:

- Remembers the user's name and preferences
- Answers questions using documents
- Searches the web for unknown facts
- Can perform basic calculations
- Logs all activity to LangSmith

🛠️ Putting It All Together (Simplified Version)

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent, AgentType
from langchain.tools import Tool
from langchain.memory import ConversationBufferMemory
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains import RetrievalQA
from langchain.callbacks import StdOutCallbackHandler

# Load documents
docs = TextLoader("guide.txt").load()
splits = RecursiveCharacterTextSplitter(chunk_size=300,
chunk_overlap=30).split_documents(docs)
db = Chroma.from_documents(splits, OpenAIEmbeddings())

# Setup RAG tool
retriever = db.as_retriever()
qa_chain = RetrievalQA.from_chain_type(llm=ChatOpenAI(), retriever=retriever)
qa_tool = Tool(name="DocumentQA", func=qa_chain.run, description="Answers
questions using internal documents")

# Custom tools
def get_time(_: str) -> str:
    from datetime import datetime
    return f"The current time is {datetime.now().strftime('%H:%M:%S')}."
time_tool = Tool(name="TimeTool", func=get_time, description="Returns current
time.")

# LLM and memory
llm = ChatOpenAI(callbacks=[StdOutCallbackHandler()])
```

```
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)

# Initialize agent
tools = [qa_tool, time_tool]
agent = initialize_agent(tools, llm,
agent=AgentType.CONVERSATIONAL_REACT_DESCRIPTION, memory=memory, verbose=True)

# Run interaction
print(agent.run("Hi, my name is Sam. What's the time and what can you tell me
about LangChain?"))
```

🩹 Capstone Project Options

Choose one:

1. **Personal AI Assistant**
   Remembers your name, checks weather, gives updates from documents.

2. **Research Agent**
   Summarizes web results, reads PDFs, writes notes.

3. **Customer Support Agent**
   Answers queries from internal docs, escalates difficult questions, logs interactions.

🚀 Deployment Tips

- Wrap your app in a FastAPI or Streamlit interface.
- Use `dotenv` for managing API keys.
- Monitor runs with LangSmith.
- Add retry logic for tool failures.
- Use vector store persistence for long-term knowledge.

☑ Summary

- You've integrated every core LangChain concept.
- Built and ran a full-featured, intelligent assistant.
- You're ready to build production-grade AI applications.

🎓 Final Assignment

1. Complete your capstone project with 3+ tools, memory, and RAG.
2. Log all activity with LangSmith.
3. Prepare a short demo video or README that explains your app.

Thank you for completing the LangChain for Agentic AI course! 🎉