**Workshop: Advanced LangChain for Data Scientists**

A multi-day hands-on course covering LangChain’s advanced features – OpenAI API integration, chain composition, memory, tools, RAG with FAISS, and agents. Each day mixes deep technical discussion with live coding. Data scientists will build and run example code, and by the end implement an end-to-end project combining LangChain, OpenAI, FAISS, and agents.

**References:** LangChain documentation and tutorials​[python.langchain.com](https://python.langchain.com/docs/integrations/llms/openai/#:~:text=To%20access%20OpenAI%20models%20you%27ll,openai%60%20integration%20package)​[python.langchain.com](https://python.langchain.com/api_reference/langchain/chains/langchain.chains.llm.LLMChain.html#:~:text=from%20langchain_core,prompt%3Dprompt)​[comet.com](https://www.comet.com/site/blog/memory-in-langchain-a-deep-dive-into-persistent-context/#:~:text=In%20LangChain%2C%20the%20Memory%20module,information%20to%20make%20better%20decisions)​[python.langchain.com](https://python.langchain.com/docs/how_to/custom_tools/#:~:text=%40tool%20decorator)​[python.langchain.com](https://python.langchain.com/docs/integrations/tools/#:~:text=Tools%20are%20utilities%20designed%20to,be%20passed%20back%20to%20models)​[python.langchain.com](https://python.langchain.com/docs/tutorials/rag/#:~:text=Build%20a%20Retrieval%20Augmented%20Generation,App%3A%20Part%201)​[python.langchain.com](https://python.langchain.com/docs/tutorials/rag/#:~:text=Indexing)​[python.langchain.com](https://python.langchain.com/docs/tutorials/rag/#:~:text=Retrieval%20and%20generation)​[python.langchain.com](https://python.langchain.com/v0.1/docs/modules/agents/#:~:text=The%20core%20idea%20of%20agents,take%20and%20in%20which%20order)​[ibm.com](https://www.ibm.com/think/tutorials/llm-agent-orchestration-with-langchain-and-granite#:~:text=LLM%20agent%20orchestration%20refers%20to,the%20adaptability%20of%20these%20systems)​[blog.langchain.dev](https://blog.langchain.dev/how-to-think-about-agent-frameworks/#:~:text=,agent%20abstractions%20built%20on%20top) provide detailed explanations of concepts and example code. These resources underpin the hands-on examples below.

**Day 1: LangChain Basics & OpenAI Integration**

* **Overview:** Introduction to LangChain’s abstractions (chains, runnables) and how to plug in OpenAI models. Explain the difference between a simple LLM call and a chain of operations.
* **OpenAI API Setup:** Sign up for OpenAI, set OPENAI\_API\_KEY, and install the integration with pip install langchain-openai​[python.langchain.com](https://python.langchain.com/docs/integrations/llms/openai/#:~:text=To%20access%20OpenAI%20models%20you%27ll,openai%60%20integration%20package). For example:

from langchain\_openai import OpenAI

llm = OpenAI(model\_name="gpt-3.5-turbo") # uses OPENAI\_API\_KEY

response = llm.invoke("Hello, how are you?")

print(response) # ChatGPT-style response

As noted in the LangChain docs, you must obtain an API key and configure it before calling any OpenAI model​[python.langchain.com](https://python.langchain.com/docs/integrations/llms/openai/#:~:text=To%20access%20OpenAI%20models%20you%27ll,openai%60%20integration%20package).

* **Prompt Templates & Chaining:** Introduce PromptTemplate and chaining syntax. In LangChain you can pipe a template into an LLM, e.g.:

from langchain\_core.prompts import PromptTemplate

from langchain\_openai import OpenAI

prompt = PromptTemplate.from\_template("Translate '{text}' to French.")

chain = prompt | OpenAI(model\_name="gpt-3.5-turbo")

result = chain.invoke({"text": "Hello, world!"})

print(result)

This creates a chain where the user-provided "text" is inserted into the prompt before calling the LLM​[python.langchain.com](https://python.langchain.com/docs/integrations/llms/openai/#:~:text=from%20langchain_core). The example above demonstrates LangChain’s pipeline style: a PromptTemplate piped into an OpenAI LLM​[python.langchain.com](https://python.langchain.com/docs/integrations/llms/openai/#:~:text=from%20langchain_core). You may also use LLMChain (the older API) for similar effect​[python.langchain.com](https://python.langchain.com/api_reference/langchain/chains/langchain.chains.llm.LLMChain.html#:~:text=from%20langchain_core,prompt%3Dprompt).

* **Key Concepts:** Data scientists should understand that LangChain chains are runnable pipelines: each step (prompt, LLM, parser) is a component. For instance, using LLMChain:

from langchain.chains import LLMChain

from langchain\_community.llms import OpenAI

from langchain\_core.prompts import PromptTemplate

prompt = PromptTemplate(input\_variables=["adjective"], template="Tell me a {adjective} joke")

chain = LLMChain(llm=OpenAI(), prompt=prompt) # Creates a chain of one prompt + LLM&#8203;:contentReference[oaicite:5]{index=5}

print(chain.invoke({"adjective": "funny"}))

This snippet (from LangChain docs) shows constructing a chain with a prompt and an OpenAI LLM​[python.langchain.com](https://python.langchain.com/api_reference/langchain/chains/langchain.chains.llm.LLMChain.html#:~:text=from%20langchain_core,prompt%3Dprompt). Unlike a simple llm.invoke(), a chain can combine multiple steps and carry intermediate state.

* **Hands-On Exercise:**
  + *Exercise:* Build a simple translation chain. Participants will create a PromptTemplate for translating English to another language and pipe it into OpenAI.
  + *Steps:*
    1. Set your OpenAI API key and install langchain-openai.
    2. Write a PromptTemplate that inserts user text into a translation prompt.
    3. Create an OpenAI LLM object and chain it with the template.
    4. Invoke the chain on sample texts and inspect the outputs.

**Day 2: Chains with Memory & Custom Tools**

* **Memory in LangChain:** Explore how to maintain context across chain calls. LangChain’s memory module persists state between calls of a chain or agent​[comet.com](https://www.comet.com/site/blog/memory-in-langchain-a-deep-dive-into-persistent-context/#:~:text=In%20LangChain%2C%20the%20Memory%20module,information%20to%20make%20better%20decisions). For example, one can attach a ConversationBufferMemory or ChatMessageHistory to a chain so that previous messages are stored and included in future prompts.
  + *Concept:* *Memory is a class that gets called at the start and end of each chain execution* – it loads past variables into the chain input and saves new outputs for next time​[python.langchain.com](https://python.langchain.com/api_reference/langchain/chains/langchain.chains.llm.LLMChain.html#:~:text=Optional%20memory%20object,memory%20docs%20for%20the%20full). This lets the model “remember” facts or conversation history. For example, using a ChatMessageHistory to store messages:

from langchain.memory import ChatMessageHistory

history = ChatMessageHistory()

history.add\_user\_message("Hello, who are you?")

history.add\_ai\_message("I am an AI model.")

print(history.messages) # Shows the stored chat history

Here the conversation history is preserved in history.messages, which could be fed back into a prompt. Tools like ConversationBufferMemory automatically accumulate all exchanges​[python.langchain.com](https://python.langchain.com/api_reference/langchain/memory/langchain.memory.buffer.ConversationBufferMemory.html#:~:text=A%20basic%20memory%20implementation%20that,simply%20stores%20the%20conversation%20history).

* + *Key Concepts:* LangChain provides several memory types: e.g. ConversationBufferMemory (stores full chat history), EntityMemory (remembers specific facts), and SummaryMemory (keeps a condensed summary)​[comet.com](https://www.comet.com/site/blog/memory-in-langchain-a-deep-dive-into-persistent-context/#:~:text=In%20LangChain%2C%20the%20Memory%20module,information%20to%20make%20better%20decisions)​[python.langchain.com](https://python.langchain.com/api_reference/langchain/memory/langchain.memory.buffer.ConversationBufferMemory.html#:~:text=A%20basic%20memory%20implementation%20that,simply%20stores%20the%20conversation%20history). Using memory is critical for chatbots or agents that must carry context or user preferences between steps.
  + *Hands-On Exercise:*
    - *Exercise:* Build a chat chain with memory. Use ConversationBufferMemory so that the LLM always sees the full previous exchange.
    - *Steps:*
      1. Create a conversational prompt template that includes a placeholder for past messages (e.g. using ChatPromptTemplate).
      2. Instantiate ConversationBufferMemory and attach it to your chain or LLM.
      3. Run a multi-turn conversation (ask a question, then another follow-up) and verify that the LLM’s second answer uses context from the first turn.
      4. Inspect the memory buffer to confirm it stored prior messages.
* **Tool Creation:** Learn how to extend LangChain with custom tools. A *tool* is simply a function or object that the LLM can call; it has a name, description, and (optionally) a JSON schema for arguments​[python.langchain.com](https://python.langchain.com/docs/how_to/custom_tools/#:~:text=Attribute%20Type%20Description%20name%20str,result%20direcly%20to%20the%20user)​[python.langchain.com](https://python.langchain.com/docs/integrations/tools/#:~:text=Tools%20are%20utilities%20designed%20to,be%20passed%20back%20to%20models). LangChain can wrap plain Python functions as tools. For example:

from langchain\_core.tools import tool

@tool

def multiply(a: int, b: int) -> int:

"""Multiply two numbers."""

return a \* b

This decorator makes multiply into a LangChain tool, using the function name and docstring as metadata​[python.langchain.com](https://python.langchain.com/docs/how_to/custom_tools/#:~:text=%40tool%20decorator). The LLM can then be prompted to use multiply(a, b) to get a numeric result. Under the hood, tools can also be built by subclassing BaseTool or using StructuredTool.from\_function for more control​[python.langchain.com](https://python.langchain.com/docs/how_to/custom_tools/#:~:text=LangChain%20supports%20the%20creation%20of,tools%20from). Well-defined names and descriptions help the LLM choose the right tool​[python.langchain.com](https://python.langchain.com/docs/how_to/custom_tools/#:~:text=Models%20will%20perform%20better%20if,names%2C%20descriptions%20and%20JSON%20schemas).

* + *Integration:* Tools are most powerful when used by agents. They let LLM agents interact with APIs, perform calculations, or query data sources. For now, focus on creating a tool; we will use it with an agent on Day 4.
  + *Hands-On Exercise:*
    - *Exercise:* Create and test a simple custom tool.
    - *Steps:*
      1. Write a Python function (e.g. a calculator or weather lookup) and decorate it with @tool.
      2. Inspect the generated name, description, and argument schema of the tool.
      3. Call the tool directly to verify it runs.
      4. (Optional) Use LangChain’s toolkit to register your tool and experiment with an LLM prompt asking to use that tool (see Agent section tomorrow).

**Day 3: Retrieval-Augmented Generation (RAG) & FAISS Vector Stores**

* **RAG Overview:** Introduction to Retrieval-Augmented Generation (RAG). In RAG-based systems, the LLM is supplemented by a retrieval component: given a user query, the system first retrieves relevant documents from a pre-built index and then feeds those to the LLM to generate an answer​[python.langchain.com](https://python.langchain.com/docs/tutorials/rag/#:~:text=Build%20a%20Retrieval%20Augmented%20Generation,App%3A%20Part%201). As LangChain’s tutorial explains, Q&A chatbots often use RAG to “build a simple Q&A application over a text data source”​[python.langchain.com](https://python.langchain.com/docs/tutorials/rag/#:~:text=Build%20a%20Retrieval%20Augmented%20Generation,App%3A%20Part%201). This improves accuracy on domain-specific questions by providing contextual facts.

**Indexing Pipeline:** A RAG system has two main phases. The first is *indexing*: ingesting and embedding the source documents. A typical pipeline is:

* 1. **Load** raw documents (PDFs, text files, etc.).
  2. **Split** large texts into chunks suitable for LLM context windows.
  3. **Embed** each chunk into a vector (using an embedding model, e.g. OpenAIEmbeddings).
  4. **Store** these vectors in a vector database (e.g. FAISS).

*Figure: RAG Indexing Pipeline (load, split, embed, store).* These steps prepare your data for fast retrieval​[python.langchain.com](https://python.langchain.com/docs/tutorials/rag/#:~:text=Indexing). For example, in code:

from langchain.document\_loaders import TextLoader

from langchain.text\_splitter import CharacterTextSplitter

from langchain.embeddings import OpenAIEmbeddings

from langchain.vectorstores import FAISS

# (1) Load and split documents

docs = TextLoader("data/articles.txt").load()

splitter = CharacterTextSplitter(chunk\_size=500, chunk\_overlap=50)

chunks = splitter.split\_documents(docs)

# (2) Embed and (3) store in FAISS

embeddings = OpenAIEmbeddings()

vector\_store = FAISS.from\_documents(chunks, embeddings)

Here vector\_store is a FAISS index containing embeddings of all chunks. FAISS (Facebook AI Similarity Search) is a high-performance library for vector similarity search​[python.langchain.com](https://python.langchain.com/docs/integrations/vectorstores/faiss/#:~:text=,See%20The%20FAISS%20Library%20paper).

* **Retrieval & Generation:** At query time, RAG works like this:
  1. **Retrieve:** Given a user question, use the vector store to find the most similar document chunks (typically via cosine similarity).
  2. **Generate:** Construct a prompt that includes both the original question and the retrieved context, then pass it to the LLM to generate the answer.

*Figure: RAG Retrieval & Generation.* The query goes through the Retriever, relevant docs are fed into the prompt, and the LLM produces the final answer​[python.langchain.com](https://python.langchain.com/docs/tutorials/rag/#:~:text=Retrieval%20and%20generation). In LangChain you can build this with a chain like:

from langchain.chains import RetrievalQA

from langchain\_openai import OpenAI

retriever = vector\_store.as\_retriever()

qa\_chain = RetrievalQA.from\_chain\_type(

llm=OpenAI(model\_name="gpt-3.5-turbo"),

chain\_type="stuff",

retriever=retriever

)

answer = qa\_chain.run("What is LangChain used for?")

print(answer)

The RetrievalQA chain will automatically query the FAISS index and append top documents to the prompt. The system then generates a response that incorporates those documents. LangChain’s RAG tutorial notes that this *“combines the power of information retrieval with advanced text generation”*​[medium.com](https://medium.com/@alexrodriguesj/retrieval-augmented-generation-rag-with-langchain-and-faiss-a3997f95b551#:~:text=Building%20Retrieval).

* **Hands-On Exercise:**
  1. *Exercise:* Build a mini RAG QA system.
  2. *Steps:*
     1. Prepare a small corpus (e.g. a few Wikipedia pages saved as text).
     2. Split and embed the corpus into a FAISS vector store (using code above).
     3. Write a query (e.g. “Explain X concept”) and use vector\_store.similarity\_search(query) to fetch relevant chunks.
     4. Feed those chunks and the query into an LLM (e.g. via RetrievalQA or manually via a PromptTemplate) and get an answer.
     5. Verify that the answer correctly references the retrieved documents.
     6. (Optional) Try conversational RAG by using ConversationalRetrievalChain to handle follow-up questions.

**Day 4: Agents, Orchestration, and Capstone Project**

* **Agents 101:** LangChain *agents* allow the LLM to decide which tools/actions to use. Unlike a fixed chain, an agent uses the LLM as a reasoning engine to choose the next action​[python.langchain.com](https://python.langchain.com/v0.1/docs/modules/agents/#:~:text=The%20core%20idea%20of%20agents,take%20and%20in%20which%20order). In practice, you supply an agent with a list of tools (search, calculators, custom tools, etc.), and the agent will iteratively query the LLM to decide which tool to call. For example:

from langchain.agents import AgentType, initialize\_agent, load\_tools

from langchain\_openai import OpenAI

llm = OpenAI(model\_name="gpt-3.5-turbo")

tools = load\_tools(["llm-math", "serpapi"], llm=llm) # built-in tools

agent = initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION)

response = agent.run("What is 17 times 19, and who is the current president of the US?")

print(response)

In this example, the agent may use the “math” tool to compute 17×19 and the “search” tool to find the president. According to LangChain docs, agents structure the workflow: *“an agent works in the following manner: … a language model is used as a reasoning engine to determine which actions to take and in which order.”*​[python.langchain.com](https://python.langchain.com/v0.1/docs/modules/agents/#:~:text=The%20core%20idea%20of%20agents,take%20and%20in%20which%20order).

* **Orchestration & Multi-Agent:** Discuss advanced scenarios. Complex workflows might involve multiple agents or multi-step reasoning. LangChain’s new *LangGraph* is an orchestration framework that allows declarative pipelines and can manage multi-agent systems​[blog.langchain.dev](https://blog.langchain.dev/how-to-think-about-agent-frameworks/" \l ":~:text=,agent%20abstractions%20built%20on%20top" \t "_blank). Generally, *LLM agent orchestration* refers to managing and coordinating the LLM and tools to perform tasks​[ibm.com](https://www.ibm.com/think/tutorials/llm-agent-orchestration-with-langchain-and-granite#:~:text=LLM%20agent%20orchestration%20refers%20to,the%20adaptability%20of%20these%20systems). For instance, IBM’s guide describes it as structuring a workflow where an AI agent “acts as the central decision-maker” and calls various APIs or tools as needed​[ibm.com](https://www.ibm.com/think/tutorials/llm-agent-orchestration-with-langchain-and-granite#:~:text=LLM%20agent%20orchestration%20refers%20to,the%20adaptability%20of%20these%20systems). This day’s content covers the conceptual background; deep orchestration (LangGraph, multi-agent design) is left as optional reading.
* **Capstone Project – Intelligent QA Agent:** In the afternoon, participants will build a comprehensive project that uses everything learned: a conversational agent capable of answering questions with context and tools. **Project outline:**
  1. **Data Setup:** Use the FAISS index from Day 3 (e.g. a knowledge base of documents).
  2. **Tools:** Include a retrieval tool (wrap the FAISS retriever as a LangChain tool or use RetrievalQA), plus at least one custom tool (e.g. a calculator or web API tool via LangChain integration).
  3. **Agent:** Initialize an agent with these tools and a ChatOpenAI LLM. Give it a system prompt instructing it to use tools for relevant subtasks.
  4. **Conversation Memory:** Attach memory so the agent remembers the dialog context between turns.
  5. **Demo:** Interact with your agent by asking complex questions (e.g. requiring document lookup + computation). Observe how it chooses tools and forms its answers.

*Example skeleton code (conceptual):*

from langchain\_openai import ChatOpenAI

from langchain.agents import initialize\_agent, AgentType

from langchain.vectorstores import FAISS

from langchain.embeddings import OpenAIEmbeddings

from langchain.tools import VectorStoreQAWithSourcesTool, tool

from langchain.memory import ConversationBufferMemory

# Build FAISS vector store (as before)

docs = [...] # load your document list

vector\_store = FAISS.from\_documents(docs, OpenAIEmbeddings())

retriever = vector\_store.as\_retriever()

# Define a retrieval-based tool

retrieval\_tool = VectorStoreQAWithSourcesTool(

name="DocQA", retriever=retriever,

description="Answer questions from the documents"

)

# Define another custom tool

@tool

def calculate(expr: str) -> str:

return str(eval(expr)) # simple math

llm = ChatOpenAI(model\_name="gpt-4")

tools = [retrieval\_tool, calculate]

agent = initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION)

memory = ConversationBufferMemory()

agent.agent.llm\_chain.memory = memory

# Run the agent

print(agent.run("Calculate 17\*19 and then find which document discusses Fibonacci."))

This setup uses a FAISS-backed tool and a math tool. The agent will call calculate for the multiplication and use DocQA (backed by FAISS) to search the knowledge base. Throughout, conversation memory retains context across the interaction.

* **Hands-On Capstone:**  
  Participants will implement this end-to-end agent, test it on queries, and refine prompts/tool descriptions. By completion, they will have built a production-ready style system that leverages **LangChain chains, memory, tools, RAG (FAISS), OpenAI LLMs, and agent orchestration**. This practical project ties together all workshop topics and demonstrates advanced LangChain in action.