LangChain, LCEL, RAG & Memory

Architecting Intelligent Applications: A Deep Dive

Presentation Agenda

Part 1: LangChain Core

- Framework Overview
- Prompt Templates & Loaders
- Memory & State Management

Part 2: RAG Architecture

- Retrieval Augmented Generation
- Vector Stores & Retrieval

Part 3: LCEL

- LangChain Expression Language
- Runnables & Chains

Part 4: Deployment

- LangServe APIs
- Observability with LangSmith

Part 1: LangChain Core

Foundations, Prompts, and Components

What is LangChain?

The Orchestration Layer

LangChain is a comprehensive orchestration framework designed to simplify the development of applications powered by Large Language Models (LLMs).

It acts as the middleware that connects your raw LLM (like GPT-4) to the real world—connecting it to your data, your APIs, and managing the state of the conversation.

Key Capabilities

- ✓ **Context-awareness:** Connecting LLMs to private data (PDFs, databases) to ground responses in reality.
- ✓ **Reasoning:** enabling the model to determine a sequence of actions (Agents) rather than just generating text.
- ✓ **Standardization:** Switching between Model Providers with minimal code changes.

Core Components & Modules

LangChain provides modular building blocks that can be used individually or combined.

Model I/O

The input/output layer. This includes **Prompt Templates** (managing dynamic inputs), **Chat Models** (interfaces for LLMs), and **Output Parsers** (structuring the raw text response into JSON or other formats).

Chains

The "glue" of the framework. Chains allow you to link multiple components together. For example, a chain might take user input, format it with a Prompt Template, and then pass it to an LLM.

Agents

Dynamic decision making. Unlike Chains (hardcoded sequences), Agents use the LLM as a reasoning engine to determine which "Tools" (Search, Calculator, API) to use and in what order.

Prompt Templates

Structuring Input

LLMs require carefully crafted prompts to work effectively. Hardcoding strings is brittle and unscalable.

Prompt Templates allow you to parameterize your inputs. They treat prompts as functions that take input variables (like user queries or context) and return a formatted string.

This abstraction makes it easy to manage complex prompt engineering, version prompts, and swap them out without changing application logic.

Code Example

```python
from langchain.prompts import PromptTemplate

# Define a template with variables
template = PromptTemplate.from_template(
  "Explain {topic} to a {audience}."
)

# Format the prompt with inputs
formatted = template.format(
  topic="quantum mechanics",
  audience="5 year old"
)
# Result: "Explain quantum mechanics to a 5
year old."
```

Document Loaders

Standardizing Data Ingestion

Before you can use data with an LLM, you need to load it. Document Loaders handle the heavy lifting of reading files from various sources (PDFs, CSVs, Notion, Slack, Web) and standardizing them.

They convert raw data into a standard Document object, which contains:

- page_content: The text content.
- metadata: Source URL, page number, author, etc.

Code Example

```python
from langchain_community.document_loaders
import PyPDFLoader

# Initialize the loader with file path
loader = PyPDFLoader("./annual_report.pdf")

# Load and parse the document
docs = loader.load()

print(docs[0].page_content)
print(docs[0].metadata)
# {'source': '...', 'page': 0}
```

Memory & State Management

The Stateless Challenge

LLMs are inherently stateless. If you send a request "My name is Bob", and then immediately send "What is my name?", the model will not know unless you provide the previous context again.

LangChain's **Memory** module handles storing, retrieving, and formatting this conversation history so it can be injected into the prompt automatically.

Memory Strategies

- **ConversationBuffer:** Stores the raw text of all previous inputs and outputs. Good for short conversations.
- **ConversationSummary:** Uses an LLM to "summarize" the conversation as it happens. Useful for long-running chats.
- **ConversationWindow:** Keeps a sliding window of the last ***k*** interactions.

Implementing Memory

Memory is typically used within a Chain. The system manages two keys: input and history.

Code Example

```python
from langchain.memory import
ConversationBufferMemory


# Initialize memory to store chat history
memory = ConversationBufferMemory(
    memory_key = "chat_history",
    return_messages= True
)


# Manually adding context (usually automated by
Chains)
memory.chat_memory.add_user_message("Hi, I'm Bob")
memory.chat_memory.add_ai_message("Hello Bob!")
```

How it works

1. User sends a new message.
2. The Chain reads the chat_history from Memory.
3. The Chain formats the Prompt: [History] + [New Input].
4. The LLM generates a response.
5. The Chain saves the new Input/Response pair back to Memory.

Part 2: RAG Architecture

Retrieval Augmented Generation

What is RAG?

Bridging the Knowledge Gap

Retrieval Augmented Generation (RAG) is a technique to optimize the output of an LLM so it references an authoritative knowledge base outside of its training data before generating a response.

Retrieval

The system searches for documents relevant to the user's query (e.g., "Company Vacation Policy") using a Vector Store.

Generation

The system passes the retrieved documents ***along with*** the user's question to the LLM to generate a grounded answer.

The RAG Pipeline Stages

A standard RAG pipeline consists of distinct ETL stages followed by retrieval.

1. **Loading:** Importing data
2. **Splitting:** Breaking large documents into smaller chunks (e.g., 1000 characters) to fit context windows.
3. **Embedding:** Converting text chunks into numerical vectors.
4. **Storing:** Saving these vectors in a Database (Pinecone, Chroma).
5. **Retrieval:** Finding the chunks most mathematically similar to the user's question.

Data Ingestion & Loaders

Document Loaders

LangChain simplifies the "Extract" phase with **Document Loaders**. These handle the messy work of parsing different file formats.

Loaders standardize everything into a Document object, which contains:

- page_content: The actual text.
- metadata: Source info, page numbers, titles, etc.

Supported Sources

```python
from langchain_community.document_loaders
import PyPDFLoader, CSVLoader, WebBaseLoader

# Load PDF
pdf_loader = PyPDFLoader("report.pdf")

# Load Website
web_loader =
WebBaseLoader("https://example.com")

docs = web_loader.load()
```

Text Splitting Strategies

Why split text?

You cannot feed a 100-page PDF into an LLM prompt; it exceeds the token limit. We must break documents into "chunks".

Recursive Character Splitter

The most common strategy. It attempts to split by paragraphs `\n\n` first, then sentences `\n`, then spaces. This tries to keep semantically related text together.

The Importance of Overlap

We usually define a chunk_overlap. If we split a sentence in half at the end of a chunk, the meaning is lost. Overlap ensures that the end of one chunk is repeated at the start of the next.

```python
from langchain_text_splitters import
RecursiveCharacterTextSplitter


# 1. Initialize the splitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100,
)


# 2. Define sample text
text = """
LangChain is a framework for developing applications
powered by language models. It provides tools for
document loading, splitting, and vector storage.
The RecursiveCharacterTextSplitter is often used for
RAG applications.
"""


# 3. Split the text into strings
chunks = text_splitter.split_text(text)
```

Embeddings

From Text to Numbers

- ✓ Embeddings are a way to represent text as a high-dimensional vector (a list of numbers, e.g., 1536 dimensions)
- ✓ In this vector space, concepts that are similar are located close together (Semantic Similarity).
- ✓ This allows us to search for "concepts" rather than just matching keywords

```
from sentence_transformers import SentenceTransformer

# 1. Load Model
model = SentenceTransformer('all-MiniLM-L6-v2’)

# 2. Input Data
text = "Vector databases are efficient."

# 3. Generate Embedding
vector = model.encode(text)

# Result: A dense float array
print(vector.shape) # (384,)
```

Vector Store & Retrieval

Storage & Retrieval

- **Vector Stores** are specialized databases designed to store and index vectors.
- They use algorithms like HNSW to perform Approximate Nearest Neighbor (ANN) searches incredibly fast, even with millions of records
- e.g. Chroma, Pinecone, FAISS, Weaviate)

- In LangChain, a **Retriever** is an interface that returns Documents given an unstructured query
- It wraps the Vector Store
- We use mmr (Maximal Marginal Relevance) to fetch documents that are similar to the query but also distinct from each other to provide diverse context.

Code Example

```
from langchain_community.vectorstores import
Chroma

# 1. Index the document chunks
db = Chroma.from_documents(chunks, embeddings)

# 2. Create a retriever interface
retriever = db.as_retriever(
  search_type="mmr",
  search_kwargs={"k": 4}
)

# 3. Retrieve relevant docs
docs = retriever.invoke("How does RAG work?")
```

Part 3: LCEL

LangChain Expression Language

The Pipe Operator (|)

Declarative Workflow

- LCEL uses a syntax similar to Unix pipes. It signifies taking the output of the left component and passing it as input to the right component
- This creates extremely readable code that reads like a workflow diagram and handles streaming and async operations out of the box.

Visualizing the Flow

```
chain = prompt | model | output_parser
```

1. **Prompt:** Takes user input -> PromptValue.
2. **Model:** Takes PromptValue -> ChatMessage.
3. **Parser:** Takes ChatMessage -> String.

The Runnable Interface

A Standard Protocol

To make the pipe syntax work, every component in LangChain (Prompts, Models, Retrievers, Output Parsers) implements the **Runnable** protocol. This ensures they all speak the same language.

Standard Methods:

- .invoke(input): Synchronous call.
- .ainvoke(input): Asynchronous call.
- .stream(input): Stream back response chunks.
- .batch(inputs): Run a list of inputs in parallel.

RunnablePassthrough:

A utility that passes input unchanged to the next step. Useful when you need to use the user's input in multiple places (e.g., in the prompt AND in the retrieval step).

Building a Chain with LCEL

We create a Chain to pass the user input via RunnablePassThrough() which then pipes to the prompt, model & parser

Step-by-Step Execution

1. **Create** the Chain piping the user input to prompt to model to parser. RunnablePassThrough(), passes the user input provided through chain.invoke()
2. **Prompt:** Populates the template with the retrieved context and question.
3. **Model:** Generates the answer
4. **Parser:** Parses the json output to get the answer

```python
# 1. Build the chain
# RunnablePassthrough() takes the string from .invoke()
# and assigns it to 'topic'
chain = (
    {"topic": RunnablePassthrough()}
    | prompt
    | model
    | parser
)


# 2. Invoke the chain
# The string "bears" is passed through to the 'topic’
# key in the prompt
result = chain.invoke("bears")
print(result)
```

Part 4: Deployment

Moving from Prototype to Production

LangServe

Instant REST APIs

Deploying LLM apps is hard. You need to handle streaming, retries, and types. **LangServe** wraps your LCEL chain and exposes it as a production-ready REST API.

It is built on top of **FastAPI** and uses **Pydantic** for data validation. If your chain expects a string, LangServe ensures the API rejects integers.

Implementation

```
from langserve import add_routes
from fastapi import FastAPI

app = FastAPI()

add_routes(
  app,
  chain,
  path="/my-chain"
)
```

Observability & Playground

The Black Box Problem

When chains get complex, it's hard to know where they failed. Did the retriever fail to find the doc? Did the LLM hallucinate?

LangSmith

A platform for debugging and monitoring. By setting one environment variable, every step of your LCEL chain is logged. You can see exactly what input the retriever got and what it outputted.

/playground

LangServe automatically generates a /playground endpoint. This is a UI that lets you interact with your API, configure inputs, and view the streaming output without writing any frontend code.