Assignment 10 — Solutions

LangChain & LCEL

Part C — Building GenAI Applications · 18 questions with answers

Q1. What is LangChain and how does it help build LLM applications? What are its main components? ★

Answer

LangChain is an open-source framework (Python/JS) for building applications powered by LLMs. It provides standard interfaces and ready-made integrations so you can connect LLMs with prompts, data, tools and memory, and switch providers easily.

Main components:

- **Models** – chat models / LLMs (OpenAI, Anthropic, Gemini, Ollama, Hugging Face) and embedding models.
- **Prompts** – PromptTemplate, ChatPromptTemplate, few-shot templates.
- **Output parsers** – convert text into strings, JSON, Pydantic objects.
- **Chains / LCEL** – compose steps with the `|` operator.
- **Document loaders, text splitters, vector stores, retrievers** – for RAG.
- **Memory / message history.**
- **Tools and agents** – let the LLM call functions/APIs (agents now built mainly with **LangGraph**).
- Ecosystem: **LangSmith** (tracing, evaluation), LangGraph (stateful agents), LangServe.

Q2. Initialize an LLM / chat model (OpenAI or open-source) and ask one question. [CODE]

Answer

```
# pip install langchain langchain-openai
import os
from langchain_openai import ChatOpenAI
os.environ["OPENAI_API_KEY"] = "sk-..."          # better: load from .env

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
response = llm.invoke("What is Retrieval-Augmented Generation in one sentence?")
print(response.content)

# Open-source alternative running locally with Ollama
# from langchain_ollama import ChatOllama
# llm = ChatOllama(model="llama3.1")
```

Q3. Create a prompt template with two input variables. [CODE]

Answer

```
from langchain_core.prompts import PromptTemplate, ChatPromptTemplate

prompt = PromptTemplate.from_template(
    "Explain {topic} to a {audience} in 3 bullet points.")
print(prompt.format(topic="vector databases", audience="10-year-old"))

# Chat version with a system message
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert teacher."),
    ("human", "Explain {topic} to a {audience}.")])
print(llm.invoke(chat_prompt.invoke({"topic": "LoRA", "audience": "manager"})).content)
```

Q4. What is LCEL (LangChain Expression Language)? Build a chain: prompt | model | output_parser. (Why did LCEL replace the legacy LLMChain?) ★

Answer

LCEL (LangChain Expression Language) is a declarative way to compose components with the **pipe operator `|`**. Every component is a **Runnable** with the same interface (invoke, stream, batch, ainvoke), so the output of one step flows into the next.

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

prompt = ChatPromptTemplate.from_template("Write a one-line tagline for a {product}.")
model = ChatOpenAI(model="gpt-4o-mini")
chain = prompt | model | StrOutputParser()
print(chain.invoke({"product": "AI-powered fraud detection system"}))
```

Why LCEL replaced LLMChain: LLMChain was a rigid class for one prompt + one LLM and is now deprecated. LCEL chains automatically get **streaming, async, batch/parallel execution, retries/fallbacks, and LangSmith tracing**, are easier to compose and customize, and work with any Runnable.

Q5. What is the difference between invoke, stream and batch? ★

Answer

- **invoke(input)** – runs the chain on **one input** and returns the **complete output** when finished.
- **stream(input)** – runs on one input and **yields output chunks (tokens) as they are generated** → lower perceived latency for chat UIs.
- **batch([inputs])** – runs on a **list of inputs in parallel** (configurable `max_concurrency`) and returns a list of outputs → efficient bulk processing.

(Async versions: ainvoke, astream, abatch.)

```
chain.invoke({"product": "phone"})
for chunk in chain.stream({"product": "phone"}):
    print(chunk, end="", flush=True)
chain.batch([{"product": "phone"}, {"product": "laptop"}], config={"max_concurrency": 5})
```

Q6. What are output parsers? Name a few (StrOutputParser, JsonOutputParser, PydanticOutputParser) and their purposes.

Answer

Output parsers convert the raw model output (an AIMessage / text) into a **usable structure**, and can also inject **format instructions** into the prompt.

- **StrOutputParser** – extracts plain text content from the message.
- **JsonOutputParser** – parses JSON (optionally against a Pydantic schema); supports streaming partial JSON.
- **PydanticOutputParser** – parses and **validates** into a Pydantic object.
- **CommaSeparatedListOutputParser** – returns a Python list.
- **DatetimeOutputParser, EnumOutputParser, XMLOutputParser.**
- **OutputFixingParser / RetryOutputParser** – ask the LLM to fix malformed output.

For models with native tool calling, `llm.with_structured_output(Schema)` is the preferred modern approach.

Q7. What is parallelism in LCEL (RunnableParallel)? What are RunnablePassthrough and RunnableLambda?

Answer

- **RunnableParallel** – runs several runnables **at the same time on the same input** and returns a dict of their outputs. (A plain dict in a chain is converted to RunnableParallel automatically.)
- **RunnablePassthrough** – passes the input through **unchanged** (often used to forward the user's question alongside retrieved context); `.assign()` adds new keys.
- **RunnableLambda** – wraps any **Python function** as a runnable.

```
from langchain_core.runnables import RunnableParallel, RunnablePassthrough, RunnableLambda

joke = ChatPromptTemplate.from_template("Tell a joke about {topic}") | llm | StrOutputParser()
poem = ChatPromptTemplate.from_template("Write a 2-line poem about {topic}") | llm |
    StrOutputParser()
both = RunnableParallel(joke=joke, poem=poem)
print(both.invoke({"topic": "data science"}))

# Typical RAG usage
rag_inputs = {"context": retriever | RunnableLambda(lambda docs: "\n\n".join(d.page_content for d
    in docs)),
              "question": RunnablePassthrough()}
```

Q8. What is a Document in LangChain? What are document loaders? Name a few.

Answer

- **Document** – LangChain's basic unit of text: an object with **`page_content`** (the text) and **`metadata`** (dict – source file, page number, URL, author, etc.). Metadata is used for citations and filtering.

```
from langchain_core.documents import Document
doc = Document(page_content="LoRA reduces trainable parameters.", metadata={"source": "notes.pdf",
    "page": 3})
```

- **Document loaders** read data from a source and return a list of Documents.

Examples: **TextLoader**, **PyPDFLoader** / PyMuPDFLoader, **CSVLoader**, **DirectoryLoader**, **WebBaseLoader** (web pages), Docx2txtLoader / UnstructuredWordDocumentLoader, UnstructuredExcelLoader, JSONLoader, YoutubeLoader, NotionDBLoader, GoogleDriveLoader, S3 loaders.

Q9. Load a .txt, a .pdf and a .csv file as LangChain documents. [CODE]

Answer

```
# pip install langchain-community pypdf
from langchain_community.document_loaders import TextLoader, PyPDFLoader, CSVLoader

txt_docs = TextLoader("notes.txt", encoding="utf-8").load()
print(len(txt_docs), txt_docs[0].metadata)        # 1 Document

pdf_docs = PyPDFLoader("report.pdf").load()        # one Document per page
print(len(pdf_docs), pdf_docs[0].metadata)        # {'source': 'report.pdf', 'page': 0}

csv_docs = CSVLoader("customers.csv").load()       # one Document per row
print(csv_docs[0].page_content)                    # "name: Ravi\ncity: Pune ..."
```

Q10. Load all .pdf files and all .txt files in a folder (DirectoryLoader). [CODE]

Answer

```
from langchain_community.document_loaders import DirectoryLoader, PyPDFLoader, TextLoader

# All PDFs in a folder (including sub-folders)
pdf_loader = DirectoryLoader("data/", glob="**/*.pdf", loader_cls=PyPDFLoader,
                             show_progress=True)
pdf_docs = pdf_loader.load()

# All text files
txt_loader = DirectoryLoader("data/", glob="**/*.txt", loader_cls=TextLoader,
                             loader_kwargs={"encoding": "utf-8"})
txt_docs = txt_loader.load()
print(len(pdf_docs), len(txt_docs))

# Alternative for PDFs: PyPDFDirectoryLoader("data/")
```

Q11. How do you split documents? Explain RecursiveCharacterTextSplitter, chunk_size and chunk_overlap. Split a document into 1000-character chunks. ★

Answer

RecursiveCharacterTextSplitter (the recommended default) tries to split on a hierarchy of separators – paragraphs "\n\n", then lines "\n", then spaces " ", then characters – so chunks keep natural boundaries while staying under the size limit.

- **chunk_size** – maximum size of each chunk (characters by default, or tokens with `from_tiktoken_encoder`).
- **chunk_overlap** – number of characters repeated between consecutive chunks so that sentences/ideas cut at a boundary are not lost (typically 10–20% of chunk_size).

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
chunks = splitter.split_documents(pdf_docs)       # or split_text(long_string)
print(len(chunks), len(chunks[0].page_content))   # each chunk <= 1000 characters
```

Other splitters: CharacterTextSplitter, TokenTextSplitter, MarkdownHeaderTextSplitter, HTMLHeaderTextSplitter, language-aware code splitters, SemanticChunker.

Q12. How do you summarize a document larger than the context window? Explain stuff, map_reduce and refine. ★

Answer

When a document is larger than the context window, split it into chunks and use one of these strategies:

- **Stuff** – put **all chunks into one prompt** and summarize once. Simplest, one LLM call, best quality – but only works if everything fits in the context window.
- **Map-Reduce** – **Map:** summarize each chunk **independently (in parallel)**. **Reduce:** combine the chunk summaries into a final summary (recursively if still too long). Scales to very large documents and is fast; may lose connections between chunks.
- **Refine** – summarize the first chunk, then **iteratively update the running summary** with each next chunk ("here is the existing summary, refine it with this new text"). Keeps context flowing, but is **sequential (slow)** and later chunks can dominate.

```
from langchain_classic.chains.summarize import load_summarize_chain   # langchain<1.0: from
    langchain.chains.summarize
chain = load_summarize_chain(llm, chain_type="map_reduce")
print(chain.invoke({"input_documents": chunks})["output_text"])
```

(Today you can also use LangGraph map-reduce, or a long-context model with "stuff" when the document fits.)

Q13. How do you make an LLM remember previous conversations? ★

Answer

LLMs are **stateless** – each API call is independent. To "remember", the application must **store the conversation and send relevant history back with each new request**.

Approaches:

- Send the **full message history** (short chats).
- Keep only the **last k messages** (window) to control tokens.
- **Summarize** older messages and keep the summary + recent messages.
- Store long-term facts in a **vector store or database** and retrieve relevant memories (long-term memory).
- In LangChain: `RunnableWithMessageHistory` with a chat-history store (in-memory, Redis, SQL); in LangGraph: **checkpointers** keyed by `thread_id`.

Q14. Explain the memory types: buffer, buffer-window (what is 'k'?), summary and summary-buffer memory. Write code for one of them.

Answer

- **ConversationBufferMemory** – stores the **entire conversation verbatim**. Simple, but tokens grow without limit.
- **ConversationBufferWindowMemory** – keeps only the **last k interactions**; **k = number of recent human–AI exchanges** kept (k=2 → last 2 question–answer pairs). Older messages are dropped.
- **ConversationSummaryMemory** – uses an LLM to keep a **running summary** of the whole conversation instead of raw messages. Good for long chats; costs extra LLM calls and loses detail.
- **ConversationSummaryBufferMemory** – hybrid: keeps **recent messages verbatim** and **summarizes older ones** once the buffer exceeds `max_token_limit`.

```
# Legacy API, still asked in interviews. In LangChain 1.x it lives in the
# langchain-classic package (pip install langchain-classic); in langchain<1.0 use
```

```
# "from langchain.chains ..." and "from langchain.memory ..." instead.
from langchain_classic.chains import ConversationChain
from langchain_classic.memory import (ConversationBufferMemory, ConversationBufferWindowMemory,
                              ConversationSummaryMemory, ConversationSummaryBufferMemory)

memory = ConversationBufferWindowMemory(k=2)
# memory = ConversationBufferMemory()
# memory = ConversationSummaryMemory(llm=llm)
# memory = ConversationSummaryBufferMemory(llm=llm, max_token_limit=200)
conv = ConversationChain(llm=llm, memory=memory, verbose=True)
conv.invoke({"input": "Hi, I'm Ravi and I work on fraud detection."})
conv.invoke({"input": "I like LangGraph."})
conv.invoke({"input": "What is my name?"})
print(memory.load_memory_variables({}))
```

Q15. How is conversation memory handled in current LangChain/LangGraph (message history, checkpointers)?

Answer

The legacy Memory classes are deprecated. Current approaches:

1. RunnableWithMessageHistory (LCEL):

```
from langchain_core.chat_history import InMemoryChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages([("system", "You are helpful."),
    MessagesPlaceholder("history"), ("human", "{input}")])
store = {}
def get_history(session_id):
    return store.setdefault(session_id, InMemoryChatMessageHistory())
chat = RunnableWithMessageHistory(prompt | llm, get_history,
                                  input_messages_key="input", history_messages_key="history")
chat.invoke({"input": "My name is Ravi"}, config={"configurable": {"session_id": "u1"}})
```

2. LangGraph checkpointers (recommended for agents):

```
from langgraph.checkpoint.memory import MemorySaver
from langgraph.prebuilt import create_react_agent
agent = create_react_agent(llm, tools=[], checkpointer=MemorySaver())
cfg = {"configurable": {"thread_id": "user-1"}}
agent.invoke({"messages": [("user", "Hi, I'm Ravi")]}, cfg)
agent.invoke({"messages": [("user", "What's my name?")]}, cfg)   # remembers
```

Use SQLite/Postgres/Redis checkpointers in production; trim or summarize messages to control tokens; LangGraph **Store** for long-term, cross-thread memory.

Q16. How do you use the Hugging Face pipeline inside LangChain?

Answer

```
# pip install langchain-huggingface transformers
from langchain_huggingface import HuggingFacePipeline, ChatHuggingFace

llm = HuggingFacePipeline.from_model_id(
    model_id="Qwen/Qwen2.5-0.5B-Instruct",
    task="text-generation",
    pipeline_kwargs={"max_new_tokens": 128, "temperature": 0.1, "do_sample": True})
print(llm.invoke("What is LangChain?"))

chat = ChatHuggingFace(llm=llm)        # applies the model's chat template
chain = prompt | chat | StrOutputParser()

# Or wrap an existing transformers pipeline
from transformers import pipeline
pipe = pipeline("text-generation", model="gpt2", max_new_tokens=50)
llm2 = HuggingFacePipeline(pipeline=pipe)

# Hosted inference instead of local: HuggingFaceEndpoint(repo_id=..., huggingfacehub_api_token=...)
```

Q17. Use one chain as a tool inside another chain or agent. [CODE]

Answer

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.tools import tool

summary_chain = (ChatPromptTemplate.from_template("Summarize in 2 lines:\n{text}")
                 | llm | StrOutputParser())

# Option 1: convert the runnable directly
summarize_tool = summary_chain.as_tool(
    name="summarizer", description="Summarizes a long piece of text.")

# Option 2: wrap it in a @tool function
@tool
def summarize(text: str) -> str:
    '''Summarize a long piece of text in two lines.'''
    return summary_chain.invoke({"text": text})

from langgraph.prebuilt import create_react_agent
agent = create_react_agent(llm, tools=[summarize])
agent.invoke({"messages": [("user", "Summarize this: <long text>")]})
```

The agent sees the tool's name and description and decides when to call the chain.

Q18. How do you handle long conversation memory without exceeding the context window or inflating token cost? ★ · Asked: Wipro

Answer

- **Sliding window:** send only the last k turns (buffer-window memory).
- **Summarization:** keep a running summary of older turns + the recent turns verbatim (summary-buffer memory); re-summarize when a token budget is exceeded.
- **Token-based trimming:** `trim_messages` with a max-token budget, always keeping the system prompt.
- **Long-term memory store:** extract important facts (user preferences, decisions) into a database/vector store and **retrieve only the relevant memories** for each new question.
- **Structured state:** keep key fields (customer_id, current order, intent) in application state instead of re-sending the whole chat.
- **Prompt caching** for the stable prefix to cut cost.

- Persist history per session (Redis/Postgres, LangGraph checkpointer) with TTL and privacy controls.