Top 40 **LangChain** Interview Questions

With Answers

1. What are the core components of LangChain?

Think of LangChain as a **framework to build LLM apps**(chatbots,RAG,agents, etc.).

Its main building blocks are:

1. **Models**
   - These are your **LLMs / Chat models** (OpenAI, Anthropic, etc.).
   - LangChain doesn't create its own models; it **connects** to them.
   - Examples: `ChatOpenAI`, `OpenAI`, local models, etc.
2. **Prompts**
   - A **prompt** is the text you send to the model.
   - LangChain helps you create **prompt templates** with variables (e.g., `{user_input}`).
3. **Chains**
   - A **chain** is a **fixed pipeline**:

     Prompt → Model → (Optional) Output Parser → Final Answer.
   - You decide the **exact order** of steps.
4. **Memory**
   - Memory is how LangChain **remembers previous messages** in a conversation.
   - It adds past chat history to the prompt automatically.
5. **Tools**
   - Tools are **functions the model can call**, like:

    - Search Google
    - Query a database
    - Call an API
    - Do math, etc.
  - Tools are mainly used by **agents**.

6. **Agents**
   - Agents are like "**smart controllers**" over an LLM.
   - The agent decides:
     - What to do next
     - Which tool to use
     - When to stop and respond to the user.

7. **Documents, Text Splitters, and Indexes**
   - Documents: text you want your LLM to use (PDFs, web pages, etc.).
   - Text splitters: break big documents into **smaller chunks**.
   - Indexes / Vector stores: store document embeddings for **search** (RAG).

8. **Output Parsers**
   - Help you turn raw model text (strings) into **structured data**, like:
     - JSON
     - Python dict
     - Custom objects

Tiny example: a very simple LangChain pipeline

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

# 1. Model
```

```
model = ChatOpenAI(model="gpt-4o-mini")

# 2. Prompt template
prompt = ChatPromptTemplate.from_template(
"Explain {topic} in simple terms for a beginner."
)

# 3. Output parser (just returns a string)
parser = StrOutputParser()

# 4. Chain = Prompt → Model → Parser
chain = prompt | model | parser

# 5. Run the chain
answer = chain.invoke({"topic": "LangChain"})
print(answer)
```

This uses **models, prompts, chains, and an output parser** — four core components.

---

2. What is the difference between a chain and an agent in LangChain?

Chain

- A **chain** is like a **fixed recipe**.
- Steps are pre-defined and **do not change** during runtime.
- Example flow:
  1. Take user input
  2. Fill a prompt template
  3. Call the model
  4. Return the result

You, the developer, decide **every step**.

Agent

- An **agent** is like a **smart assistant** that can:
  - Decide **what to do next**
  - Choose **which tool to call**
  - Use **multiple tools step by step**
- The decision-making is done by the **LLM itself**, guided by a system prompt.

You decide **what tools are available**.

The **agent decides** how to use them.

Simple comparison

Feature	Chain	Agent
Flow	Fixed	Dynamic
Who decides next step?	You (developer)	LLM (agent "brain")
Tools	Usually none or fixed	Can choose from multiple tools
Complexity	Simple to build, predictable	Powerful but more complex

---

Mini examples

Chain example (fixed steps):

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

model = ChatOpenAI(model="gpt-4o-mini")

prompt = ChatPromptTemplate.from_template(
    "Translate this sentence into Hindi:\n\n{sentence}"
)

chain = prompt | model | StrOutputParser()
```

```
print(chain.invoke({"sentence": "I love learning LangChain."}))
```

This chain will **always** just translate text. Nothing more, nothing less.

Agent example (dynamic tool use):

High-level idea (conceptual, not full runnable code):

```
from langchain_openai import ChatOpenAI
from langchain.tools import tool
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.prompts import ChatPromptTemplate

# 1. Define a tool
@tool
def add(a: int, b: int) → int:
"""Add two integers and return the sum."""
    return a + b

tools = [add]

# 2. Model
model = ChatOpenAI(model="gpt-4o-mini")

# 3. Agent prompt (simplified)
prompt = ChatPromptTemplate.from_template(
"You are a helpful assistant that can use tools to solve problems."
)

# 4. Create agent and executor
agent = create_tool_calling_agent(model, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools)

# 5. Ask a question; agent decides whether to call add()
```

```
result = agent_executor.invoke({"input": "What is 12 + 30?"})
pr int(resul t["output"])
```

Here, the **agent** reads the question, decides to call the `add` tool, gets the result, and then replies.

---

3. How does LangChain handle memory?

Memory in LangChain is how it **remembers previous interactions** and passes them back to the model.

Why do we need memory?

Without memory, this can happen:

1. You: “My name is Chandra.””
2. You: “What is my name?”
3. Model (without memory): “You did not tell me your name.””

With memory, the model **gets the full conversation**, so it can answer:

“Your name is Chandra.””

Types of memory (conceptually)

Common memory styles:

1. **Conversation buffer memory**
    - Stores **all previous messages** as plain text.
    - Good for small/medium conversations.
2. **Summary memory**
    - Stores a **summary** of older messages.
    - Useful when conversations are long (to avoid very long prompts).
3. **Combined / hybrid memory**
    - Mix of raw recent messages + summary of older ones.

Simple example: Conversation with memory

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

# Model
model = ChatOpenAI(model="gpt-4o-mini")

# Memory: keeps all previous messages in a buffer
memory = ConversationBufferMemory(return_messages=True)

# Conversation chain with memory
conversation = ConversationChain(
 llm=model,
 memory=memory,
    verbose=True # prints internal steps
)

# Turn 1
print(conversation.predict(input="Hi, my name is Chandra."))

# Turn 2
print(conversation.predict(input="What is my name?"))
```

Because of `ConversationBufferMemory`, the second call includes the context of the first message, so the model can remember your name.

---

4. What are indexes in LangChain, and how are they used?

In LangChain/RAG context, **indexes** are usually **vector stores** or similar data structures that help the model **find relevant information** from documents.

Simple idea

1. You have many documents (PDFs, articles, notes).
2. You convert them into **embeddings** (vectors).
3. You store them in a **vector database** (FAISS, Chroma, Pinecone, etc.).
4. When a user asks a question:
    - You convert the question into an embedding.
    - You search the index for the **most similar chunks**.
    - You give those chunks + the question to the model.

This is the core idea of **RAG (Retrieval-Augmented Generation).**

Example: building a small index with a list of texts

```
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_community.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains import RetrievalQA

# 1. Documents
texts = [
"LangChain is a framework for building LLM applications.",
"Retrieval-Augmented Generation (RAG) combines a retriever with a genera
tor.",
"Vector stores help us search similar text using embeddings."
]

# 2. Split text (for longer docs, this matters more)
splitter = RecursiveCharacterTextSplitter(chunk_size=100, chunk_overlap=10)
docs = splitter.create_documents(texts)

# 3. Create embeddings and index (FAISS)
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(docs, embeddings)
```

```
# 4. Create retriever
retriever = vectorstore.as_retriever()

# 5. Model
llm = ChatOpenAI(model="gpt-4o-mini")

# 6. RetrievalQA chain (uses retriever + LLM)
qa_chain = RetrievalQA.from_chain_type(
llm=llm,
    retriever=retriever,
 return_source_documents=True
)

# 7. Ask a question
result = qa_chain({"query": "What is LangChain used for?"})
pr int(resul t["resul t"])
```

Here, **FAISS** is the **index** (vector store) that helps find relevant chunks for the question.

---

5. What is the purpose of prompt templates in LangChain?

If you manually write a prompt every time, it’s easy to:

- Repeat yourself
- Make mistakes
- Forget to include important instructions

Prompt templates solve this.

What is a prompt template?

- A **prompt with placeholders** (variables) that you can fill in.
- Example: `"Explain {topic} to a 10-year-old."`
- You only change `{topic}`; the rest remains consistent.

Benefits

- **Reusability**: One template, many usages.
- **Consistency**: Same format every time.
- **Safety**: You can lock in system instructions.
- **Easier to maintain**: You can update wording in one place.

Example: Using `ChatPromptTemplate`

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

# 1. Prompt template
prompt = ChatPromptTemplate.from_template(
    """
    You are a friendly coding tutor.

    Explain the following concept to a beginner:
    Concept: {concept}

 Use a simple example.
    """
)

# 2. Model + parser
model = ChatOpenAI(model="gpt-4o-mini")
parser = StrOutputParser()

# 3. Chain
chain = prompt | model | parser

# 4. Invoke with a specific concept
print(chain.invoke({"concept": "what is an API"}))
```

You can now reuse the same `prompt` for **any concept** by just changing the `concept` var iable.

---

6. What are tools in LangChain, and how do agents use them?

What are tools?

Tools are **functions that the LLM is allowed to call** during reasoning.

Examples of tools:

- A function to:
  - Search the web
  - Query a database
  - Look up a document
  - Do math (`calculator`)
  - Call an external API (weather, stocks, etc.)

Each tool has:

- A **name**
- A **description** (tells the model when to use it)
- A **function** to execute

How agents use tools

An **agent**:

1. Reads the user’s question.
2. Decides if it needs a tool.
3. If yes:
   - Picks a tool
   - Calls it with parameters

4. Looks at the result.
5. Maybe calls more tools.
6. Finally, gives a **final answer**.

All this decision logic is done by the **LLM**, guided by a system prompt that explains how to use tools.

Simple example: a calculator tool

```
from langchain_openai import ChatOpenAI
from langchain.tools import tool
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.prompts import ChatPromptTemplate

# 1. Define a tool as a Python function
@tool
def multiply(a: int, b: int) → int:
"""Multiply two integers and return the result."""
    return a * b

tools = [multiply]

# 2. Model
model = ChatOpenAI(model="gpt-4o-mini")

# 3. Agent prompt
prompt = ChatPromptTemplate.from_template(
    """
 You are a helpful assistant. You can use tools to solve problems.
    If the user asks for any calculation, use the appropriate tool.
 When done, explain the answer in simple language.
    """
)

# 4. Create the agent and executor
```

```
agent = create_tool_calling_agent(model, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools)

# 5. Ask a question that requires calculation
result = agent_executor.invoke({"input": "If a box has 24 apples and I buy 5 su
ch boxes, how many apples do I have?"})
pr int(resul t["output"])
```

The agent should **decide** to call `multiply(24, 5)` and then explain the result.

---

7. How does LangChain support multi-turn conversations?

Multi-turn conversation = **chat that remembers previous turns.**

LangChain supports this using:

1. **Chat models** (like `ChatOpenAI`)
2. **Memory** (conversation history)
3. Optional: **chains or agents** wrapped with memory.

A simple conversation with memory

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

# 1. Model
model = ChatOpenAI(model="gpt-4o-mini")

# 2. Memory: keeps all previous messages
memory = ConversationBufferMemory(return_messages=True)

# 3. Conversation chain
conversation = ConversationChain(
llm=model,
```

```
 memory=memory
)

# Turn 1
print("User: Hi, I am Chandra.")
print("Bot:", conversation.predict(input="Hi, I am Chandra."))

# Turn 2
print("\nUser: What is my name?")
print("Bot:", conversation.predict(input="What is my name?"))
```

Because of memory, the model knows your name in the second turn.

Multi-turn conversations with agents

You can also combine:

- **Agents** (for tool usage)
- **Memory** (for remembering chat history)

Conceptually:

```
from langchain_openai import ChatOpenAI
from langchain.memory import ConversationBufferMemory
from langchain.agents import AgentExecutor, create_tool_calling_agent
from langchain.prompts import ChatPromptTemplate

# tools = [...] # Your tools
# model = ChatOpenAI(...)
# memory = ConversationBufferMemory(return_messages=True)

prompt = ChatPromptTemplate.from_template(
    """
    You are a helpful assistant that can use tools.
    Use the conversation history to keep context:
    {chat_history}
```

```
    User: {input}
    """
)

# agent = create_tool_calling_agent(model, tools, prompt)

# agent_executor = AgentExecutor(agent=agent, tools=tools, memory=memor
y)
# Now each call to agent_executor.invoke(...) will remember previous messag
es
```

Now the agent can:

- Remember what was discussed earlier.
- Use tools over multiple turns.
- Behave more like a human assistant.

If you want, next we can:

- Take **each concept** (e.g., memory or indexes) and go **deeper with more code**, or
- Build a **small RAG example** or **simple agent** step by step.

---

8. How do you configure an LLM in LangChain for text generation?

In LangChain, an **LLM object** is your main way to talk to a model like OpenAI, etc.

Basic steps:

1. **Install provider package** (example: OpenAI)
2. **Set your API key** (usually via environment variable)
3. **Create an LLM / Chat model instance**
4. **Use it to generate text**

Example: configuring a chat model for text generation

```
from langchain_openai import ChatOpenAI

# 1. Create the model
llm = ChatOpenAI(
 model="gpt-4o-mini", # which model to use
 temperature=0.7,         # creativity level (0 = strict, 1 = creative)
 max_tokens=256           # max length of the generated answer
)

# 2. Call the model directly
response = llm.invoke("Explain LangChain in very simple words.")
pr int(response.content)
```

Key parameters:

- `model`: which LLM to use (e.g., `"gpt-4o-mini"`, `"gpt-4.1"`, etc.)
- `temperature`: controls randomness
  - `0.0` → more factual, stable
  - `0.8` → more creative, varied
- `max_tokens`: max length of the response

You can then plug this `llm` into **chains, agents, RAG pipelines**, etc.

---

9. Explain the role of callbacks in LangChain for monitoring.

Callbacks are like "hooks" that let you **observe what's happening inside LangChain**:

- When an LLM is called
- When a tool is used
- When a chain starts/ends
- When tokens stream, etc.

You can use callbacks to:

- Log inputs and outputs
- Measure latency / response time
- Stream tokens in real time (like a typing effect)
- Debug complex chains/agents

Simple example: printing tokens as they are generated

```
from langchain.callbacks.base import BaseCallbackHandler
from langchain_openai import ChatOpenAI

class PrintTokensHandler(BaseCallbackHandler):
 def on_llm_new_token(self, token: str, **kwargs) → None:
        # This runs every time a new token is produced
        print(token, end="", flush=True)

# Create the callback handler
handler = PrintTokensHandler()

# Attach it to the model
llm = ChatOpenAI(
 model="gpt-4o-mini",
 temperature=0.2,
    callbacks=[handler] # register callback here
)

# Call the model
_ = llm.invoke("Tell me a short story about a programmer learning LangChai
n.")
```

You'll see the story **streaming token by token** in the terminal.

You can also write callbacks to:

- Save logs to a file

- Send metrics to an observability tool
- Record full trace of a chain/agent for debugging

---

10. How do you use LangChain to switch between different LLM providers?

One big advantage of LangChain: it gives you a **common interface** to talk to **different providers**.

General idea:

1. You write your **app logic** (chains, prompts, etc.) using a generic "LLM" or "ChatModel"."
2. You only swap **which class you import / instantiate**.

Example: Switching from OpenAI to another provider (conceptually)

Using OpenAI:

```
from langchain_openai import ChatOpenAI

def get_tutor_llm():
    return ChatOpenAI(
        model="gpt-4o-mini",
        temperature=0.4
    )

llm = get_tutor_llm()
print(llm.invoke("Explain what an API is.").content)
```

Suppose you want to switch to another provider (e.g. Anthropic, Groq, etc.)

```
# from langchain_anthropic import ChatAnthropic # (example)
# or from langchain_groq import ChatGroq          # (example)
```

```
def get_tutor_llm():
# Just change this implementation
    return ChatAnthropic(
        model="claude-3-opus-20240229",
        temperature=0.4
    )

llm = get_tutor_llm()
print(llm.invoke("Explain what an API is.").content)
```

Everything else in your app can remain the same if you design it to accept `llm` as a parameter.

Tips for easy switching

- Wrap model creation in a **function** or **config file**.
- Pass `llm` into chains/agents instead of hardcoding it.
- You can use env variables like `PROVIDER=openai` / `PROVIDER=anthropic` and choose based on that.

---

11. What is a chain in LangChain, and how is it used in NLP?

A **chain** is a **pipeline of steps** that processes data and calls models.

In NLP, you often need more than just "send prompt → get answer":"

- Preprocess user input
- Insert it into a prompt
- Call LLM
- Post-process/format the output

A chain lets you **combine these steps** in a clear, reusable way.

Modern way: using the `|` (pipe) operator

Typical NLP chain:

1. **PromptTemplate** →
2. **LLM / Chat model** →
3. **Output parser**

Example: Simple explanation chain

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

# 1. Prompt template
prompt = ChatPromptTemplate.from_template(
 "Explain the concept of {topic} in simple language with a small example."
)

# 2. Model
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.5)

# 3. Output parser
parser = StrOutputParser()

# 4. Chain = prompt → model → parser
chain = prompt | llm | parser

# 5. Use the chain
result = chain.invoke({"topic": "tokenization in NLP"})
pr int(resul t)
```

How it is used in NLP:

- Question answering
- Summar ization
- Translation

- Paraphrasing
- Grammar correction
- Classification, etc.

You just change the **prompt** and plug in the appropriate model.

---

12. What is the difference between LLMChain and SequentialChain?

> Note: These are from the “classic” LangChain chains API.
>
> The newer style uses composable runnables (`prompt | llm | parser`), but LLMChain/SequentialChain are still useful concepts.

LLMChain

- A **single-step** chain:
  - Takes input variables
  - Fills a **prompt template**
  - Calls an LLM
  - Returns the result

Think: **“one prompt → one LLM call”**.

SequentialChain

- A chain that **runs multiple chains one after another**.
- Output of one chain can become input to the next.
- Good for **multi-step workflows**, like:
  1. Generate an outline.
  2. Expand it into a full article.
  3. Summarize the article.

Table view

Feature	LLMChain	SequentialChain
Steps	Single	Multiple, ordered
Data flow	Input → Prompt → LLM → Output	Output of step i → Input of step i+1
Use cases	Simple tasks	Pipelines / multi-stage NLP workflows

13. How do you create a sequential chain in LangChain?

Let’sbuild a simple2-steppipeline:

1. Chain 1: **Generate a blog outline**
2. Chain 2: **Write the blog content from that outline**

Using classic `LLMChain` + `SimpleSequentialChain`

```
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.6)

# Step 1: Outline generator
outline_prompt = PromptTemplate(
    input_variables=["topic"],
 template="Create a short blog outline about: {topic}"
)
outline_chain = LLMChain(llm=llm, prompt=outline_prompt)

# Step 2: Content writer
content_prompt = PromptTemplate(
    input_variables=["outline"],
 template="""
Write a detailed blog post based on this outline:
    {outline}
    """
```

```
)
content_chain = LLMChain(llm=llm, prompt=content_prompt)

# Sequential chain (output of step 1 → input of step 2)
overall_chain = SimpleSequentialChain(
    chains=[outline_chain, content_chain],
    verbose=True
)

# Run the sequential chain
topic = "Benefits of learning LangChain for developers"
final_blog = overall_chain.run(topic)
print(final_blog)
```

What happens:

- You call `overall_chain.run(topic)`
- First chain uses `topic` → returns `outline`
- Second chain uses that `outline` → returns `blog`

Newer runnable style (conceptually)

You can also do something like:

```
# overall_chain = first_chain | second_chain
```

Where each part can be a runnable. But the above `SimpleSequentialChain` example gives you the **clear "step by step" idea**, which is good for beginners.

---

14. How do you pass inputs to a LangChain chain?

It depends on the type of chain and how many input variables it expects.

Case 1: Single input variable

For many simple chains (like `SimpleSequentialChain`), you can use:

```
result = chain.run("Your input here")
```

This works when:

- The chain expects **only one input**, and
- It knows the input variable name internally.

Case 2: Multiple named variables

If your prompt has multiple variables, for example:

```
prompt = PromptTemplate(
    input_variables=["language", "topic"],
 template="Explain {topic} in {language}."
)
```

Then you usually call:

```
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.4)
chain = LLMChain(llm=llm, prompt=prompt)

result = chain.invoke({
"language": "Hindi",
"topic": "REST APIs"
})
print(result["text"]) # older LLMChain returns dict with "text"
```

With the **newer runnable pipeline style** (`prompt | llm | parser`):

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

```
prompt = ChatPromptTemplate.from_template(
"Explain {topic} in {language} with a tiny example."
)

llm = ChatOpenAI(model="gpt-4o-mini")
parser = StrOutputParser()

chain = prompt | llm | parser

# Here invoke gets a dict of inputs
answer = chain.invoke({
"language": "simple English",
   "topic": "JSON"
})
print(answer)
```

Q15. How do you configure an LLM in LangChain for text generation?

To generate text using LangChain, you must first configure an **LLM (Large Language Model)**.

LangChain allows you to use many providers (OpenAI, Anthropic, Groq, etc.), but the setup steps are similar:

Steps to configure an LLM

1. Install the provider's integration package
2. Set the API key
3. Create the LLM / Chat model instance
4. Use it with `.invoke()` for text generation

Beginner-friendly Example (OpenAI)

```
from langchain_openai import ChatOpenAI
```

```
llm = ChatOpenAI(
model="gpt-4o-mini",
temperature=0.7, # creativity
 max_tokens=200        # length of output
)

response = llm.invoke("Write a short note about LangChain.")
pr int(response.content)
```

Important configuration parameters

- **model** → Which AI model to use
- **temperature** → Controls creativity
- **max_tokens** → Length of response
- **top_p/top_k** → Controls randomness (optional)

Q16. Explain the role of callbacks in LangChain for monitoring.

Callbacks are like **event listeners** in LangChain.

They allow you to **track and monitor everything happening** during:

- LLM calls
- Tool usage
- Chains execution
- Agent decision steps
- Token streaming

Why are callbacks useful?

- Debugging
- Logging model inputs/outputs
- Tracking how long requests take

- Streaming tokens (typing effect)
- Observability dashboards
- Monitoring cost usage (in some integrations)

Simple Callback Example: Print tokens as they stream

```
from langchain.callbacks.base import BaseCallbackHandler
from langchain_openai import ChatOpenAI

class PrintTokens(BaseCallbackHandler):
def on_llm_new_token(self, token, **kwargs):
        print(token, end="", flush=True)

llm = ChatOpenAI(
model="gpt-4o-mini",
    callbacks=[PrintTokens()]
)

llm.invoke("Tell a short story about a robot learning coding.")
```

You will see the answer printed **token-by-token**, like a typing animation.

Q17. How do you use LangChain to switch between different LLM providers?

LangChain makes switching LLMs extremely easy because all models share a **common interface**.

General strategy

1. Define a function that returns an LLM model
2. Change only that function when switching providers
3. Keep the rest of your application the same

Example: Using OpenAI

```
from langchain_openai import ChatOpenAI

def get_model():
return ChatOpenAI(model="gpt-4o-mini")
```

Switch to Anthropic (example)

```
from langchain_anthropic import ChatAnthropic

def get_model():
return ChatAnthropic(model="claude-3-opus-20240229")
```

Switch to Groq (example)

```
from langchain_groq import ChatGroq

def get_model():
 return ChatGroq(model="mixtral-8×7b")
```

No changes needed in your chains or prompts.

Only the **model creation function** changes → this is the power of LangChain.

Q18. What is a chain in LangChain, and how is it used in NLP?

A **Chain** is a **pipeline of steps** for processing text.

Typical NLP workflow:

1. Receive input
2. Insert it into a prompt
3. Call an LLM
4. Parse the output

A chain lets you combine these steps.

Modern chain using the pipe operator (`|`)

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

prompt = ChatPromptTemplate.from_template(
"Explain {topic} in simple language with an example."
)

llm = ChatOpenAI(model="gpt-4o-mini")
parser = StrOutputParser()

chain = prompt | llm | parser

result = chain.invoke({"topic": "embeddings"})
pr int(resul t)
```

Chains are used for many NLP tasks

- Translation
- Summar ization
- Text classification
- Paraphrasing
- Question answering
- Blog/article generation

You simply adjust the **prompt template** to change the behavior.

Q19. What is the difference between LLMChain and SequentialChain?

Feature	LLMChain	SequentialChain
Steps	One step	Multiple steps
Input/Output	Single prompt → LLM	Output of step 1 → step 2 → …
Best for	Simple tasks	Multi-step workflows

LLMChain (single-step)

```
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate

llm = ChatOpenAI(model="gpt-4o-mini")

prompt = PromptTemplate(
    input_variables=["concept"],
 template="Explain {concept} in beginner-friendly words."
)

chain = LLMChain(llm=llm, prompt=prompt)

chain.run("reactive programming")
```

SequentialChain (multi-step)

Usedwheneachstepdependsontheprevious one.

Example:

1. Generate an outline
2. Convert outline → full article

Q20. How do you create a Sequential Chain in LangChain?

Let’s build a simple 2-step pipeline:

Goal:

✓ Step 1 → Generate outline

✓ Step 2 → Expand into blog

Code:

```
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.6)

outline_prompt = PromptTemplate(
    input_variables=["topic"],
 template="Write a short outline for a blog on: {topic}"
)
outline_chain = LLMChain(llm=llm, prompt=outline_prompt)

content_prompt = PromptTemplate(
    input_variables=["outline"],
 template="Expand this outline into a full blog:\n{outline}"
)
content_chain = LLMChain(llm=llm, prompt=content_prompt)

# Sequential chain: step1 → step2
overall_chain = SimpleSequentialChain(
    chains=[outline_chain, content_chain],
    verbose=True
)

print(overall_chain.run("Benefits of learning LangChain"))
```

The output of the first chain becomes the input of the second.

Q21. How do you pass inputs to a LangChain chain?

Different chains accept inputs differently.

Case 1: Single-input chains

```
result = chain.run("Hello AI")
```

This works when the chain expects **one variable**.

Case 2: Multiple variables

If your prompt has:

```
template = "Explain {concept} in {language}."
```

You pass:

```
result = chain.invoke({
    "concept": "API",
 "language": "simple English"
})
pr int(resul t["tex t"])
```

Case 3: Runnable chains using `|`

```
result = (prompt | llm | parser).invoke({
    "topic": "JSON",
    "language": "English"
})
```

Q22. What is the role of output parsers in LangChain chains?

When an LLM replies, it usually sends back **plain text**.

But in real apps, you often need the output in a **specific format**, such as:

- Just a **string** with no extra quotes
- A **list** (e.g., bullet points)
- A **JSON object** (e.g., `{ "title": "...", "tags": [...] }`)
- A **Python dictionary** or custom object

Output parsers in LangChain help you:

1. **Tell the model how to format the output** (using instructions)
2. **Convert the model's text into structured data** in Python

Some common parsers:

- `StrOutputParser` → returns a **simple string**
- `JsonOutputParser` → expects **valid JSON** and parses it
- Other structured parsers (Pydantic-based, etc.)

Simple example: Using `StrOutputParser`

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

model = ChatOpenAI(model="gpt-4o-mini")

prompt = ChatPromptTemplate.from_template(
    "Explain {topic} in one short paragraph for a beginner."
)

parser = StrOutputParser()

chain = prompt | model | parser

result = chain.invoke({"topic": "APIs"})
print(result) # this is a plain Python string
```

Here, the **output parser** makes sure your chain returns a clean **string**, not a complex object.

Q23. How do you implement a chain with multiple prompts in LangChain?

A “chain with multiple prompts” usually means:

- Step 1: Use **Prompt A** → LLM
- Step 2: Take output from step 1, use it in **Prompt B** → LLM
- (Optionally more steps...)

You can do this using:

- Classic `LLMChain` + `SimpleSequentialChain`, or
- Modern runnable style (`|`) with multiple prompt steps.

Example: Two-step chain using classic API

Goal:

1. Generate an outline
2. Turn that outline into a blog

```
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.6)

# Prompt 1 – create outline
outline_prompt = PromptTemplate(
    input_variables=["topic"],
 template="Create a 3-point outline for a blog about: {topic}"
)
outline_chain = LLMChain(llm=llm, prompt=outline_prompt)
```

```
# Prompt 2 – expand to blog
content_prompt = PromptTemplate(
    input_variables=["outline"],
 template="Write a blog article based on this outline:\n{outline}"
)
content_chain = LLMChain(llm=llm, prompt=content_prompt)

# Chain them sequentially
overall_chain = SimpleSequentialChain(
    chains=[outline_chain, content_chain],
    verbose=True
)

blog = overall_chain.run("Why beginners should learn LangChain")
pr int(blog)
```

Each step has its **own prompt**, and both are part of the same multi-prompt chain.

Q24. Write a function to chain text generation and parsing.

Let's build a function that:

1. Uses an LLM to **generate JSON text**
2. Uses an **output parser** to parse that JSON into Python

Goal:

Given a topic, generate **3 FAQs** in structured JSON.

```
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain_core.output_parsers import JsonOutputParser

model = ChatOpenAI(model="gpt-4o-mini")
parser = JsonOutputParser()
```

```
prompt = ChatPromptTemplate.from_template(
    """
    You are an FAQ generator.

    For the topic: "{topic}"

 Return exactly this JSON format:
    {{
      "faqs": [
        {{  "question":  "string",  "answer":  "string"  }},
        {{  "question":  "string",  "answer":  "string"  }},
        {{ "question": "string", "answer": "string" }}
      ]
    }}
    """
)

chain = prompt | model | parser

def generate_faqs(topic: str):
    """
    Generates structured FAQs for a given topic using LangChain
 and returns them as a Python dict.
    """
    result = chain.invoke({"topic": topic})
    return result # this is a Python dict thanks to JsonOutputParser

# Example usage
faqs_data = generate_faqs("LangChain basics")
pr int(faqs_data["faqs"][0]["question"])
pr int(faqs_data["faqs"][0]["answer"])
```

This function shows **text generation + parsing** combined into a reusable function.

Q25. What is memory in LangChain, and how is it used in NLP?

Innormal LLMcalls, eachrequest is **stateless**:

The model doesn't remember what you said earlier unless you send the previous messages again.

Memory in LangChain is a helper that:

- Stores **conversation history** (messages)
- Automatically **injects that history** into the prompt for future calls

Why is this important in NLP?

For **multi-turn conversations**, you want the model to:

- Remember your name
- Remember your goals Refer to
- earlier questions Continue
- discussions naturally

Without memory:

> You: My name is Neha.
>
> You: What is my name?
>
> Model: I don't know.

With memory:

> You: My name is Neha.
>
> You: What is my name?
>
> Model: Your name is Neha.

LangChain provides different memory types:

- **ConversationBufferMemory** → stores all messages
- **ConversationSummaryMemory** → stores a summary

- **Combined** approaches for long chats

Q26. How do you add memory to a LangChain chain?

Simplest way: use `ConversationChain` with a memory object.

Example: Add memory with `ConversationBufferMemory`

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

llm = ChatOpenAI(model="gpt-4o-mini")

memory = ConversationBufferMemory(return_messages=True)

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

print(conversation.predict(input="Hi, my name is Chandra."))
print(conversation.predict(input="What is my name?"))
```

Here:

- `memory` stores messages
- Each call to `conversation.predict()` automatically **includes previous chat history** in the prompt.

You can also add memory to more complex chains/agents, but this is the most beginner-friendly starting point.

Q27. How do you retrieve memory from a LangChain conversation?

Once you're using memory, you might want to:

- See what is stored inside
- Use it somewhere else
- Debug the conversation

You can do this using methods on the memory object:

Using `ConversationBufferMemory`

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

llm = ChatOpenAI(model="gpt-4o-mini")
memory = ConversationBufferMemory(return_messages=True)

conversation = ConversationChain(llm=llm, memory=memory)

conversation.predict(input="Hi, I'm learning LangChain.")
conversation.predict(input="Can you remind me what I'm learning?")
conversation.predict(input="Explain it in one sentence.")

# 1. See stored variables
print(memory.load_memory_variables({}))
# Example key: {"history": "<full formatted history>"}

# 2. Access raw messages
for msg in memory.chat_memory.messages:
 print(type(msg), " → ", msg.content)
```

- `load_memory_variables({})` returns a dict (usually with `"history"`).
- `chat_memory.messages` gives you individual message objects (human/AI).

Q28. What is the role of memory keys in LangChain?

Memory keys control:

1. **What name is used to store the history**
2. **How that history is injected into the prompt**

Key properties:

- `memory_key` → the key under which chat history is saved/returned
- `input_key` → the name of the main user input variable
- `output_key` → the name for model output (in some chains)

Example: if your prompt template expects `{chat_history}` as a variable, your memory should use `memory_key="chat_history"` so everything lines up.

Example with custom memory key

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

llm = ChatOpenAI(model="gpt-4o-mini")

memory = ConversationBufferMemory(
 memory_key="chat_history", # this key will hold the history
 return_messages=True
)

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

conversation.predict(input="Hi, I'm learning LangChain.")
print(memory.load_memory_variables({}))
```

If later your prompt template uses `{chat_history}`, the keys will match.

In short:

Memory keys are the **names** used to connect:

- Memory → chain
- Chain → prompt

So that the conversation history appears in the right place.

Q29. What is retrieval-augmented generation (RAG) in LangChain?

Retrieval-Augmented Generation (RAG) is a pattern where:

1. You **retrieve** relevant information from external data (PDFs, docs, DB, etc.)
2. You **feed that information + the user question** to the LLM
3. The LLM generates an answer using this context

This solves a big problem:

LLMs **don't know your private data** (company docs, PDFs, notes).

RAG lets you "attach" your own knowledge to the model at **query time**, without retraining.

RAG Workflow in simple steps

1. Load your documents
2. Split them into chunks
3. Convert chunks into embeddings
4. Store them in a **vector store** (index)
5. At query time:
    - Convert the question into an embedding
    - Retrieve similar chunks
    - Pass those chunks + question to the LLM

LangChain has helpers for **all** these steps.

Q30. How do you create a vector store in LangChain?

A **vector store** stores **embeddings** of text chunks and lets you do **similarity search**.

Example using **FAISS** (in-memory vector store):

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Step 1: Your raw texts
texts = [
 "LangChain helps you build LLM-powered applications.",
 "Retrieval-Augmented Generation combines retrieval and generation.",
 "Vector stores allow similarity search over text."
]

# Step 2: Split into chunks (more useful for large docs)
splitter = RecursiveCharacterTextSplitter(chunk_size=100, chunk_overlap=10)
docs = splitter.create_documents(texts)

# Step 3: Create embeddings
embeddings = OpenAIEmbeddings()

# Step 4: Create vector store
vectorstore = FAISS.from_documents(docs, embeddings)

# Step 5: Test a similarity search
query = "How can I search documents using embeddings?"
results = vectorstore.similarity_search(query, k=2)

for doc in results:
    print("Chunk:", doc.page_content)
```

This `vectorstore` can now be plugged into a retriever and then into a **RAG chain**.

Q31. What is the role of embeddings in LangChain retrieval?

Embeddingsare numericalrepresentations(vectors)oftextsuchthat:

- Similar text → similar vectors
- Different text → distant vectors

In retrieval/RAG, embeddings are used to:

1. Convert each document chunk into a vector
2. Store those vectors in a **vector store**
3. Convert the user’s question into a vector
4. Compare this query vector to document vectors
5. Retrieve the **most similar** chunks

Without embeddings, you’d be limited to simple keyword search.

With embeddings, you get **semantic search**, e.g.:

- Query: “How can I talk to a database using LangChain?”
- Retrieved chunk: “LangChain supports tools that can run SQL queries on relational databases.””

Even though the words are not identical, the **meaning** is similar, so their embeddings are close.

Tiny example of embedding usage

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS

embeddings = OpenAIEmbeddings()

texts = ["apple fruit", "apple company", "banana", "grapes"]
vectorstore = FAISS.from_texts(texts, embeddings)

query = "iPhone maker"
```

```
results = vectorstore.similarity_search(query, k=1)
print(results[0].page_content) # likely "apple company"
```

The model understands that **“iPhone maker” ≈ “apple company”** via embeddings.

Q32. How do you use LangChain to query a vector store?

Once you’ve created a **vector store** (like FAISS, Chroma, etc.), you typically:

1. Turn it into a **retriever**
2. Use `.invoke()` / `.get_relevant_documents()` to query it
3. (Optionally) plug it into a **RAG chain**

Example: Query a FAISS vector store directly

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS

# 1. Build a vector store from sample texts
texts = [
"LangChain is a framework to build LLM-powered apps.",
"RAG combines document retrieval with generation.",
"Embeddings help with semantic search over text."
]

embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_texts(texts, embeddings)

# 2. Query it
query = "How can I search documents semantically?"
results = vectorstore.similarity_search(query, k=2)
```

```
for i, doc in enumerate(results, start=1):
    print(f"Result {i}: {doc.page_content}")
```

Using it as a retriever

```
retriever = vectorstore.as_retriever(search_kwargs={"k": 2})

docs = retriever.invoke("What does LangChain do?")
for d in docs:
    print("-", d.page_content)
```

The retriever is what you'll usually connect to a **RAG pipeline**.

Q33. Write a function to create a LangChain RAG pipeline.

Let's build a **small reusable RAG pipeline**:

- Input: user question
- Steps:
  1. Use retriever to get relevant docs
  2. Pass docs + question to an LLM
  3. Return answer

We'll use LCEL (`|` pipes).

```
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

def create_rag_pipeline(texts):
    """
```

```
 Creates a simple RAG pipeline:
- builds vector store from `texts`
    - returns a chain that answers questions using those texts
    """
    # 1. Split texts into chunks
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=300,
        chunk_overlap=50
    )
 docs = splitter.create_documents(texts)

    # 2. Build vector store
 embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(docs, embeddings)

    # 3. Retriver
    retriever = vectorstore.as_retriever()

 # 4. LLM
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.2)

 # 5. Prompt for RAG
prompt = ChatPromptTemplate.from_template(
        """
        You are a helpful assistant. Use ONLY the context below to answer.

        Context:
        {context}

        Question:
        {question}

        If the answer is not in the context, say "I don't know from the given docu
ments."
        """
    )
```

```
# 6. RAG chain: question → retrieve → format prompt → LLM → string
    rag_chain = (
        {"context": retriever, "question": lambda x: x["question"]}
        | prompt
        | llm
        | StrOutputParser()
    )

    return rag_chain

# Example usage:
texts = [
"LangChain helps build applications using LLMs.",
"RAG stands for Retrieval-Augmented Generation.",
"FAISS is a vector store for efficient similarity search."
]

rag = create_rag_pipeline(texts)
answer = rag.invoke({"question": "What is RAG?"})
print(answer)
```

This is a **fully working mini RAG pipeline** in a single function.

Q34. How do you implement a custom retriever in LangChain?

Sometimes you don't want to use a built-in vector store retriever.

You can implement your own by **subclassing** `BaseRetriever`.

Example: a dumb retriever that just returns all documents containing the query word.

```
from typing import List
from langchain_core.documents import Document
from langchain_core.retrievers import BaseRetriever
```

```
class KeywordRetriever(BaseRetriever):
def __init__(self, docs: List[Document]):
        self.docs = docs

 def _get_relevant_documents(self, query: str) → List[Document]:
        # Simple logic: return docs where query word appears in text
        query_lower = query.lower()
        return [
            doc for doc in self.docs
            if query_lower in doc.page_content.lower()
        ]

# Example usage:
docs = [
 Document(page_content="LangChain is a framework for LLM apps."),
 Document(page_content="Python is a popular programming language."),
 Document(page_content="RAG uses retrieval and generation.")
]

retriever = KeywordRetriever(docs)

for d in retriever.invoke("LangChain"):
    print("-", d.page_content)
```

In practice, you'll write custom retrievers for:

- Custom APIs
- Hybrid search (keyword + vector)
- Database lookups, etc.

Q35. How do you use LangChain to implement semantic search?

Semantic search = search based on meaning, not just keywords.

With LangChain:

1. Use **embeddings** to encode texts
2. Store them in a **vector store**
3. Use `.similarity_search()` or a retriever

We've already done this, but here's a clear semantic search function:

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS

def build_semantic_search_index(texts):
 embeddings = OpenAIEmbeddings()
 vectorstore = FAISS.from_texts(texts, embeddings)
    return vectorstore

def semantic_search(vectorstore, query, k=3):
 results = vectorstore.similarity_search(query, k=k)
    return [doc.page_content for doc in results]

# Example usage
texts = [
 "Apple makes the iPhone.",
 "Bananas are a great source of potassium.",
 "Google is a large technology company.",
 "iPhones are popular smartphones."
]

vs = build_semantic_search_index(texts)
hits = semantic_search(vs, "smartphone company", k=2)

for h in hits:
    print("-", h)
```

Even though "smartphone company" doesn't appear as-is, embeddings help retrieve **Apple/iPhone** text.

Q36. How do you create a custom tool in LangChain?

A **tool** is just a **Pythonfunction** with metadata, which an **agent** can call.

Steps:

1. Define a normal function
2. Decorate it with `@tool`
3. Pass it into an agent

```
from langchain.tools import tool

@tool
def add_numbers(a: int, b: int) → int:
    """Add two integers and return the result."""
    return a + b

# Manual use:
print(add_numbers.invoke({"a": 5, "b": 7})) # 12
```

Use tool inside an agent

```
from langchain_openai import ChatOpenAI
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.prompts import ChatPromptTemplate

tools = [add_numbers]

llm = ChatOpenAI(model="gpt-4o-mini")

prompt = ChatPromptTemplate.from_template(
    """
    You are a helpful assistant that can use tools.
    Use tools when needed to answer user questions.
    """
)
```

```
agent = create_tool_calling_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools)

result = agent_executor.invoke({"input": "What is 123 + 456?"})
pr int(resul t["output"])
```

The agent decides when to call `add_numbers` and how to use its result.

🔄 Q37. What is the role of Input and Output Parsers in LangChain?

Think of them as "adapters" on **both sides** of the LLM:

- **Input side (Prompt/Input parsers)**
  - Turn structured data → prompt text
  - Or combine multiple inputs into a final prompt

    (In practice, prompt templates already cover a lot of this.)
- **Output side (Output parsers)**
  - Turn raw LLM text → Python types
  - e.g., string, JSON, dict, list, custom object

You saw `StrOutputParser`, `JsonOutputParser` earlier.

Why they're important:

- Make your app **less brittle** (you don't manually `.split()` strings)
- Allow **structured outputs** → easier to use downstream
- Help guide the LLM to follow specific formats

Example (tiny refresh):

```
from langchain_core.output_parsers import StrOutputParser

parser = StrOutputParser()
```

```
# Used in a chain:
# chain = prompt | llm | parser
```

Input and output parsers help keep your pipeline **clean and predictable**.

Q38. What is LangChain Expression Language (LCEL)?

LCEL is a way to build LangChain pipelines using Python's `|` (pipe) operator.

It treats components as **Runnables**, so you can write:

```
chain = prompt | llm | parser
```

instead of manually wiring everything.

Benefits:

- **Composable**: you can easily combine/stack steps
- **Declarative**: you describe "what happens" in order
- **Uniform**: everything has `.invoke()`, `.batch()`, `.astream()`

Example LCEL chain:

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

prompt = ChatPromptTemplate.from_template(
  "Explain {topic} in simple terms."
)
llm = ChatOpenAI(model="gpt-4o-mini")
parser = StrOutputParser()

chain = prompt | llm | parser

print(chain.invoke({"topic": "LangChain"}))
```

LCEL is now the **recommended modern style** to build LangChain apps.

Q39. What are the different types of Memory in LangChain, and when should you use each?

Common memory classes:

1. **ConversationBufferMemory**

- Stores **all messages** as-is.
- Good for **short/medium** conversations.
- Easiest to understand.

2. **ConversationSummaryMemory**
   - Uses an LLM to create a **summary** of past messages.
   - Useful for **long chats**, where sending entire history is too big.
3. **ConversationBufferWindowMemory**
   - Stores only the **last N turns**.
   - Good for focusing on **recent context**.
4. **Combined Memory**
   - Mix of summary + recent buffer.

When to use what?

- **Small chatbot / debugging / early prototype** → `ConversationBufferMemory`
- **Long-running assistant** (support bot, tutor, etc.) → `ConversationSummaryMemory` or combined
- **Task-focused conversation** (only last few turns matter) → `ConversationBufferWindowMemory`

Tiny example:

```
from langchain.memory import ConversationBufferMemory, ConversationSummaryMemory
```

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")

buffer_memory = ConversationBufferMemory(return_messages=True)
summary_memory = ConversationSummaryMemory(llm=llm, return_message
s=True)
```

You pick based on how long and "heavy" the conversation will be.

Q40. What are Runnable Interfaces in LangChain and how are they used?

Runnables are the core abstraction behind LCEL.

Anything that can be "run" is a `Runnable`:

- Prompt templates
- LLMs / Chat Models
- Output parsers
- Custom functions

Every Runnable has common methods:

- `.invoke(input)` → single input
- `.batch(list_of_inputs)` → run in parallel on many inputs
- `.astream(input)` → async streaming

This makes chaining easy:

```
chain = prompt | llm | parser
result = chain.invoke({"topic": "RAG"})
```

You can also create your own `RunnableLambda`:

```
from langchain_core.runnables import RunnableLambda

def to_upper(text: str) -> str:
    return text.upper()

upper = RunnableLambda(to_upper)

print(upper.invoke("hello langchain")) # "HELLO LANGCHAIN"
```

Then you can stick it in a pipeline:

```
pipeline = prompt | llm | parser | upper
```

Runnables = **plug-and-play building blocks** for LangChain pipelines.