**Topics:**

* **Components of LangChain**
* **Models (LLMs and ChatModels)**
* **Prompts**
* **Structured Output & Output Parsers**

Here are your first set of questions 👇

**✅ 10 Conceptual Questions (Advanced Level)**

1. **How does LangChain’s modular architecture help in building scalable and maintainable LLM applications?**

**ANS:** langchain modular architecture means langchain is build using separate building blocks(modules) such as Models,Prompts,Chains,Memory, Agents,Tools and Output Parsers each block has its own job you can mix and match them to complete your task.  
So, by the modules your application will be maintainable and scalable because of

**Reuseability(**you can reuse each module in many apps**),**

**Easy to Update(**to change any module you don’t have to touch any other modules**),**

**Debug Friendly(**if any module fails you have to check only that module don’t have to check other modules**),**

**Scalability(**ifyou have to add more modules you that have to update previous code**),**

**Customization(**you can mix different modules based on your requirements**).**

1. **Compare and contrast LLM, ChatModel, and Function Calling in LangChain. When would you prefer one over the other?**

| **Feature** | **LLM** | **ChatModel** | **Function Calling** |
| --- | --- | --- | --- |
| **Type** | Basic language model | Chat-based model | Special feature of ChatModel |
| **Input format** | Plain text | List of chat messages | Function schema |
| **Best for** | Simple tasks (e.g., summaries) | Conversations (chatbots) | Calling tools/functions using AI |
| **Output** | Text | Text + message metadata | Structured function call (like a JSON object) |

**What is Function Calling?**

Function Calling means:

The **ChatModel** can **choose a function to call**, and **fill its arguments** automatically.

It’s like telling the AI:

“Here’s a tool (like a calculator). When needed, call it with the right values.”

1. **Explain the role of Runnable in LangChain and how it supports composability of chains.**
2. **What are the advantages and limitations of using PromptTemplate over hardcoded strings in production?**

Advantages of PromptTemplate

**Reuseable(**you can use one prompt with different inputs**).**

**Default Validation(**you can set validate\_template = True to validate its input**)**

**Clean Code**

**Easy to Update(**change prompt in one place**)**

**Dynamic(**works well with user input or variable content**)**

1. **Why and when would you need a custom output parser instead of using LangChain's StrOutputParser or JsonOutputParser?**

In LangChain, you need a custom output parser when the built-in parsers don't meet your specific needs for structuring language model outputs

**When to use a custom output parser?**

1. **Complex formats**: e.g., extracting bullet points, tables, or key-value pairs.
2. **Data validation**: You want to check or fix output before using it.
3. **Custom objects**: You want output as Python classes or special data types.
4. **Partial outputs**: Handling incomplete or noisy LLM responses.
5. **Post-processing needed**: e.g., extracting numbers, dates, or filtering results.
6. **Describe how prompt chaining works in LangChain and its role in complex workflows.**

Prompt chaining in Langchain involves creating a sequence of prompts where the output of one prompt serves as the input for the subsequent prompt.

This technique allows you to break down large, intricate tasks into smaller, manageable steps, improving the efficiency and accuracy of large language models (LLMs)

1. **In what scenarios can the StructuredOutputParser fail, and how can you make your application more robust?**

**When can StructuredOutputParser fail?**

1. **Bad formatting**: LLM doesn’t return valid JSON.
2. **Missing fields**: Model skips or misnames fields.
3. **Extra text**: LLM adds comments or explanations around JSON.
4. **Incorrect data types**: LLM gives string instead of number, etc.

**How to make it more robust?**

1. **Use clear instructions**: Tell the LLM exactly how to format the output.
2. **Give examples**: Show correct JSON format in the prompt.
3. **Use retry logic**: If parsing fails, try again or fix the output.
4. **Validate data**: Check if all fields exist and types are correct.
5. **Use function calling**: If structure is critical, function calling is safer than parsing.
6. **How would you design a LangChain pipeline that supports different LLMs (e.g., OpenAI and HuggingFace) without changing the core logic?**
7. **What’s the tradeoff between using JSON schema-based parsing vs. function-calling for structured outputs?**
8. **Explain how you can maintain context across multi-step conversations when using LangChain. What are the challenges?**

**🚀 30 Programming Questions / Mini Projects**

These require combining different topics and can be reused as project modules in your portfolio.

**🧠 Prompt Engineering + Model + Parsers**

1. Build a chain that takes a job description and generates a structured JSON with {"required\_skills": [], "experience\_level": "", "summary": ""} using StructuredOutputParser.
2. Create a multi-prompt translation chain that takes a paragraph and translates it into French, then summarizes it in English.
3. Implement a chain that accepts a legal paragraph and extracts names, dates, and legal entities using a custom PydanticOutputParser.
4. Build a prompt pipeline that uses conditional routing: if the input contains a question, summarize it; else, rewrite it in a formal tone.
5. Create a smart prompt chain that generates tweets from article headlines and filters them using a custom scoring function (chain + parser).
6. Develop a chatbot that answers user queries and outputs structured logs of each step (intent, entities, response).
7. Implement a form-filling assistant that asks follow-up questions based on missing fields in a JSON schema.
8. Build a code-review LLM tool that takes code as input and outputs a structured JSON with "bugs":[], "optimizations":[], "complexity\_score": ""}.
9. Design a pipeline that reads a research abstract and produces: (1) field of study, (2) difficulty level, (3) key takeaway, using structured output.
10. Create a parser that detects hallucinated facts in the LLM output by validating it against a known list of facts.

**🔁 Chain Composition & Reusability**

1. Build a modular chain that performs named entity recognition (NER) followed by sentiment classification on the same input.
2. Create a dynamic chain that selects different prompts for LLMs based on the domain (finance, medicine, general).
3. Implement a 3-step pipeline: summarize a blog post → extract FAQs → rewrite FAQs in conversational tone using chained prompts.
4. Build a generic LangChain component that logs every prompt and output to a database for later audit/debug.
5. Develop a retry-enabled parser that attempts to fix output format errors automatically using feedback and retry.

**🛠️ Integration + Prompt/Model Config**

1. Build a function-calling powered tool that answers travel questions and uses structured function output for date, city, and budget.
2. Design a LangChain module that wraps both OpenAI and HuggingFace models with the same interface and switches based on cost/delay thresholds.
3. Create a CLI tool using LangChain that summarizes PDFs and outputs data in a specified JSON format.
4. Build a multilingual FAQ generator using a prompt that supports multiple languages and uses structured output for categories.
5. Implement an LLM chain that compares two user prompts and returns which is more ambiguous with reasons, parsed in structured form.

**⚙️ Advanced Prompting Patterns**

1. Build a chain that uses few-shot examples dynamically based on input length and domain.
2. Create a prompt generator that adjusts itself based on a user’s skill level (beginner, intermediate, expert).
3. Develop a multi-step guided reasoning prompt chain that walks through math problems and parses steps with Pydantic.
4. Design a system that takes a user idea and generates a business plan, splitting it into problem, solution, audience, and revenue.
5. Create a creative writing assistant that generates a character bio and uses function-calling to let the user tweak age, style, and background.

**🧩 Parsing + Validation Logic**

1. Build a tool that takes messy user input (e.g. mixed languages, typos) and parses structured contact info.
2. Implement a structured output chain that validates responses with regex or schema before accepting.
3. Build a debugging assistant that runs LangChain prompts and checks if the output format matches a JSON schema.
4. Create a parser that converts LLM-written SQL queries into JSON format for frontend tools to consume.
5. Develop a LangChain validator that tests different prompts on an LLM and ranks them based on output quality and parse rate.