



Building RAG Application with LangChain

LLMs Series

Dinh-Thang Duong – TA
Nguyen-Thuan Duong – TA

Objectives

❖ Content



LangChain

```
Curl
curl -X 'POST' \
  'http://0.0.0.0:5000/generative_ai' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "question": "what is BERT?"
}'

Request URL
http://0.0.0.0:5000/generative_ai

Server response
Code Details
200 Response body
{
  "answer": "BERT is a bidirectional transformer model for natural language processing pre-trained on a large corpus of text. It stands for Bidirectional Encoder Representations from Transformers. BERT was designed to be similar to OpenAI GPT, and it outperforms other language representation models in various tasks. The table shows the GLUE test results for BERT and OpenAI GPT, and the table shows the SQuAD 1.1 results for different models. BERT comes in two sizes: BERT BASE and BERT LARGE. The larger model has more parameters and generally performs better. Fine-tuning BERT on different tasks is illustrated in Figure 4. Additional details and ablation studies are presented in Appendices A, B, and C."
}
```

In this lecture, we will discuss about:

1. What is LLMs in Production?
2. What is Langchain?
3. Basic components of LangChain.
4. How to use LangChain to deploy an API?
5. How to use LangChain to deploy a RAG application?

Outline

- Introduction
- LangChain
- RAG with LangChain
- Question

 AI

AI VIET NAM
@aivietnam.edu.vn

Introduction

Introduction

❖ Getting Started

Textual Description (Prompt):

Write a python function that receive an image and plot it using matplotlib library.

 You
Write a python function that receive an image and plot it using matplotlib library.

 ChatGPT
To create a Python function that receives an image and plots it using the Matplotlib library, you can use the following code as an example. You'll need to have Matplotlib and a library like PIL (Pillow) installed to work with images:

```
python
import matplotlib.pyplot as plt
from PIL import Image

def plot_image(image_path):
    try:
        # Open the image using PIL (Pillow)
        img = Image.open(image_path)

        # Create a Matplotlib figure and axis
        fig, ax = plt.subplots()
```

 Copy code



Introduction

❖ What are Large Language Models (LLMs)?



You

write a python code that print "hello world"



ChatGPT

Certainly! Here's a simple Python code that prints "Hello, World!" to the console:

python

Copy code

```
print("Hello, World!")
```

You can run this code in a Python interpreter, and it will display "Hello, World!" as the output.



ChatGPT App:

1. User inputs a prompt (greetings, task...).
2. ChatGPT (Chatbot) outputs a appropriate response.

Introduction

❖ What are Large Language Models (LLMs)?

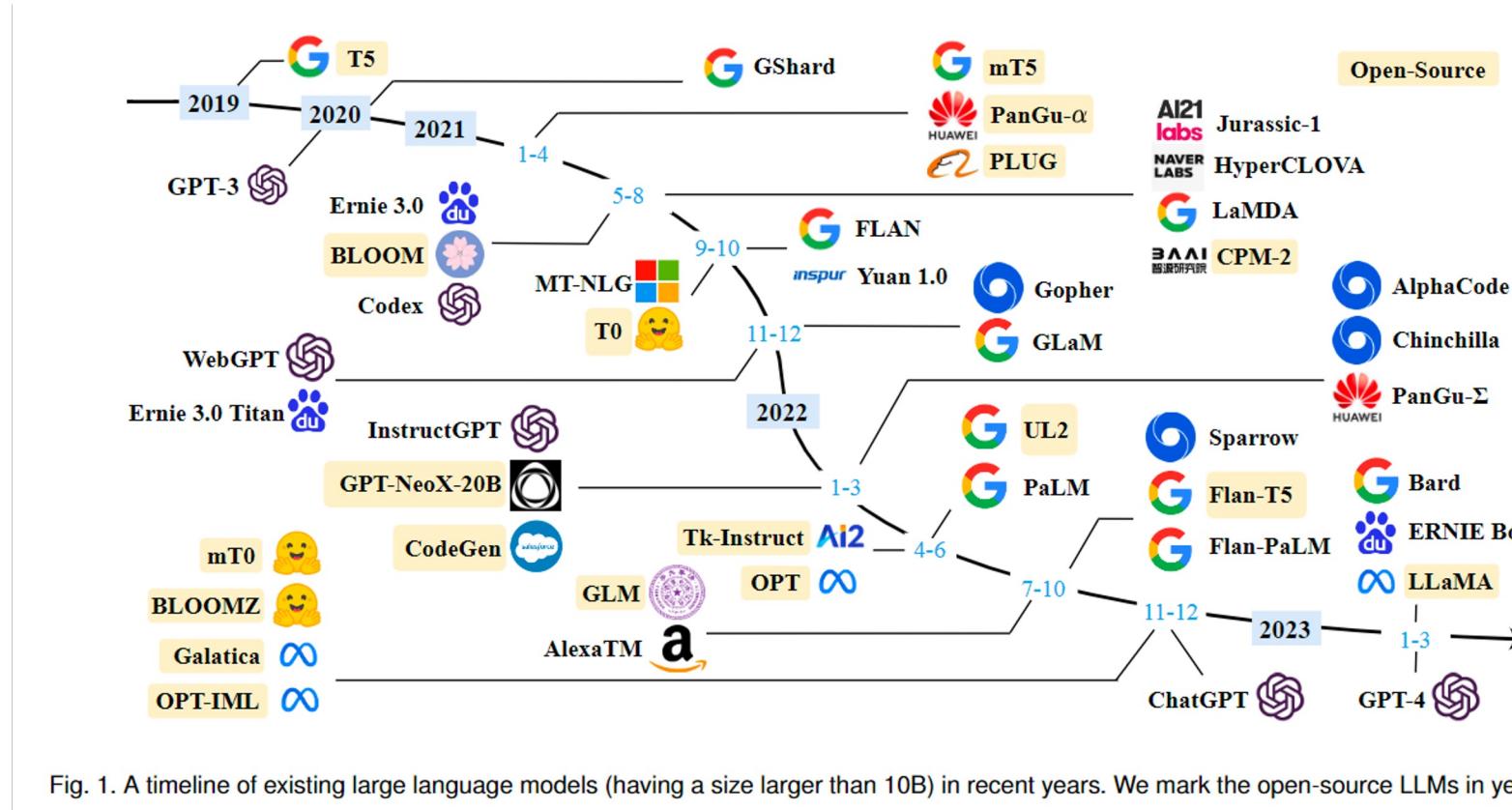


Fig. 1. A timeline of existing large language models (having a size larger than 10B) in recent years. We mark the open-source LLMs in yellow color.

LLMs (Large Language Models): AI models (language models) that were trained on a very large corpus of text. This made them capable of performing various NLP tasks with high precision.

Introduction

❖ What are Large Language Models (LLMs)?

LLMs are often pretrained on a vast majority of data and designed to be adaptable to a wide variety of tasks (Foundation models).

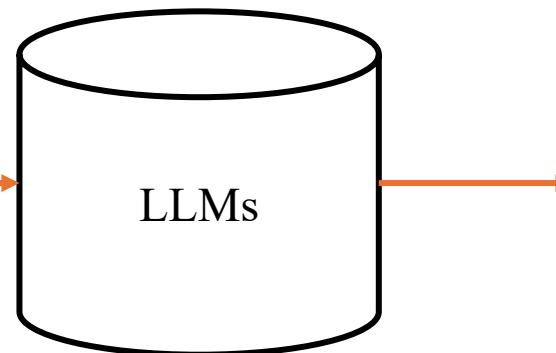


Introduction

❖ LLMs I/O

Input Text

Write a python function that receive an image and plot it using matplotlib library.



Given a “prompt”, LLMs can generate an appropriate response.

Output Text

AI You Write a python function that receive an image and plot it using matplotlib library.

ChatGPT To create a Python function that receives an image and plots it using the Matplotlib library, you can use the following code as an example. You'll need to have Matplotlib and a library like PIL (Pillow) installed to work with images:

```
python
import matplotlib.pyplot as plt
from PIL import Image

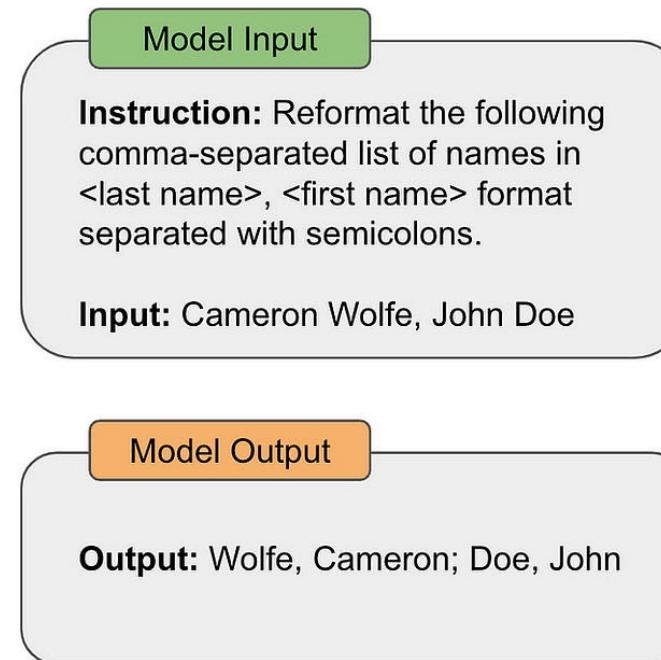
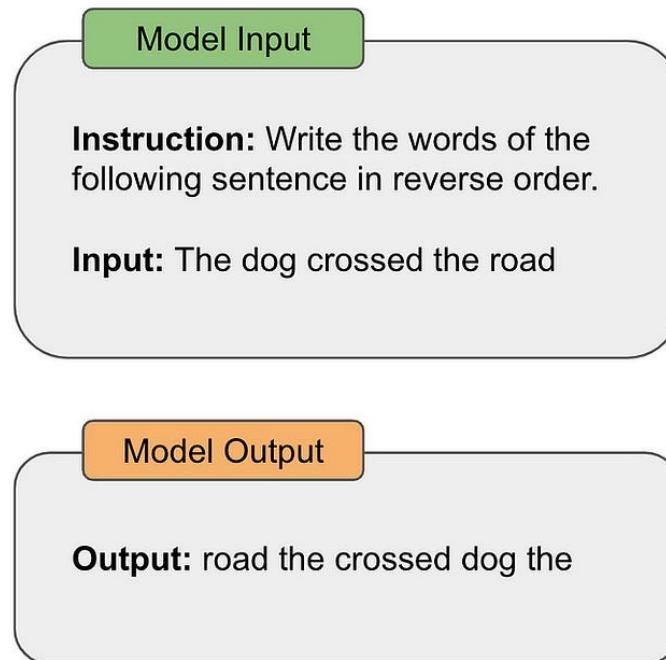
def plot_image(image_path):
    try:
        # Open the image using PIL (Pillow)
        img = Image.open(image_path)

        # Create a Matplotlib figure and axis
        fig, ax = plt.subplots()
```

Copy code

Introduction

❖ Generative AI Prompting



Prompting: Refers to a process of providing an input, usually in the form of text data, to a generative AI model to generate a specific output.

Introduction

❖ Getting Started

```
21 encoding = tokenizer(prompt, return_tensors="pt").to(device)
22 with torch.inference_mode():
23     outputs = model.generate(
24         input_ids=encoding.input_ids,
25         attention_mask=encoding.attention_mask,
26         generation_config=generation_config
27     )
28
29 print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

<|im_start|> system

Bạn là một chuyên gia về toán. Bạn sẽ nhận câu hỏi trắc nghiệm kèm theo các lựa chọn, hãy giải step by step nếu có và chọn phương án đúng.

<|im_start|> user

Câu hỏi:

1 + 1 =

Các lựa chọn:

A. 1

B. 2

C. 3

D. 4

Câu trả lời:

<|im_start|> assistant

Đáp án B 1 + 1 = 2

In LLMs research, we often use (inference) model like this.



Introduction

❖ Getting Started

The screenshot shows the AI Viet Nam dashboard with a dark theme. On the left, there's a sidebar with icons for ChatGPT, Sora, Code Copilot, Data Visualizer, and others, along with a 'Explore GPTs' link. Below that is a 'Projects' section with a 'New project' button. The main area features a large input field with the placeholder 'Ask anything' and a microphone icon. Above the input field, a message says 'Hey, VLAi. Ready to dive in?'. At the bottom, there's a green box containing the text 'But in production...'. The top right corner shows 'Temporary' and 'VT PLUS' status indicators.

ChatGPT 4o

Temporary VT PLUS

ChatGPT

Sora

Code Copilot

Data Visualizer

1 more

Explore GPTs

Projects

New project

Hey, VLAi. Ready to dive in?

Ask anything

+

Search

Deep research

...

But in production...

View plans

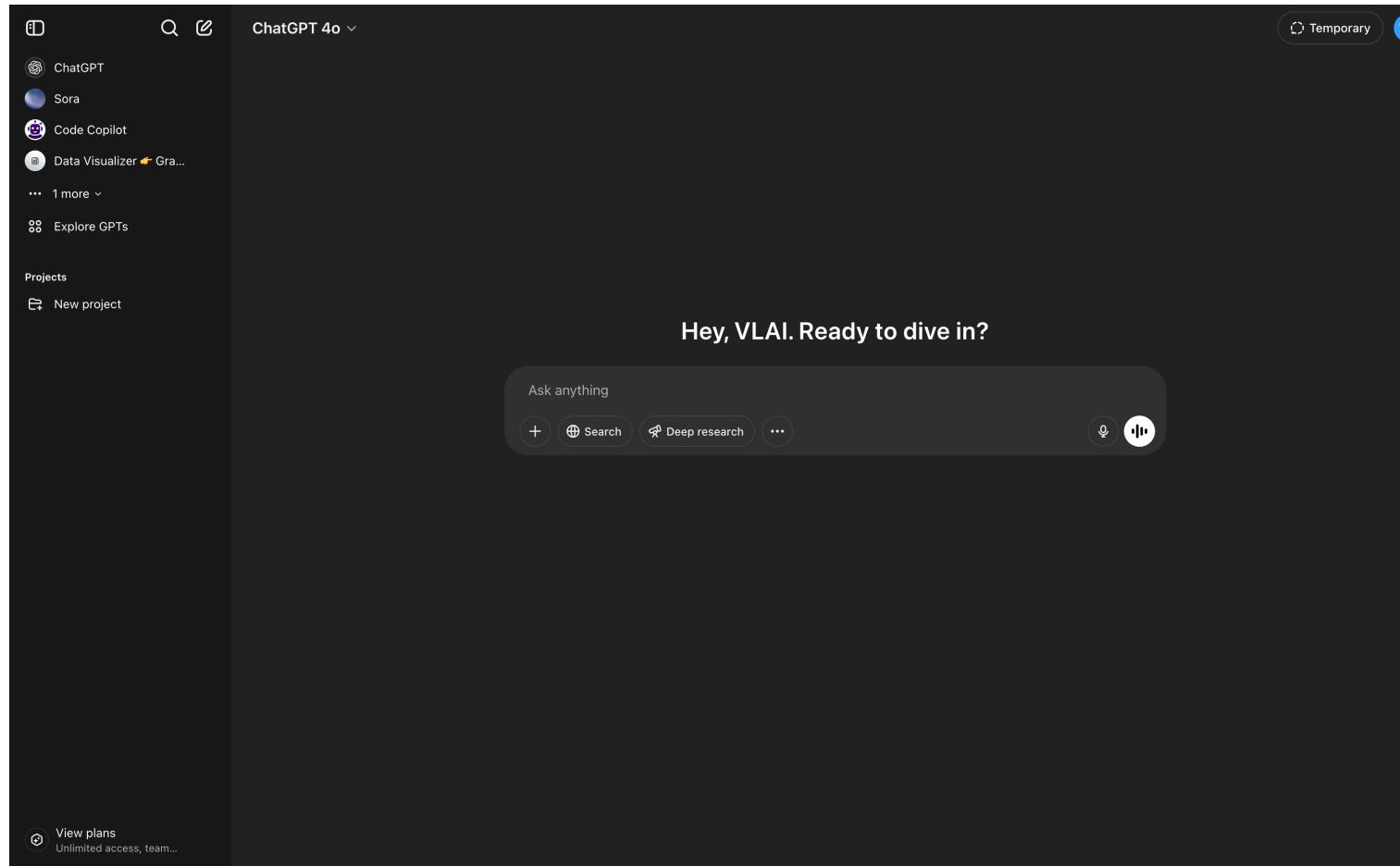
Unlimited access, team...

?



Introduction

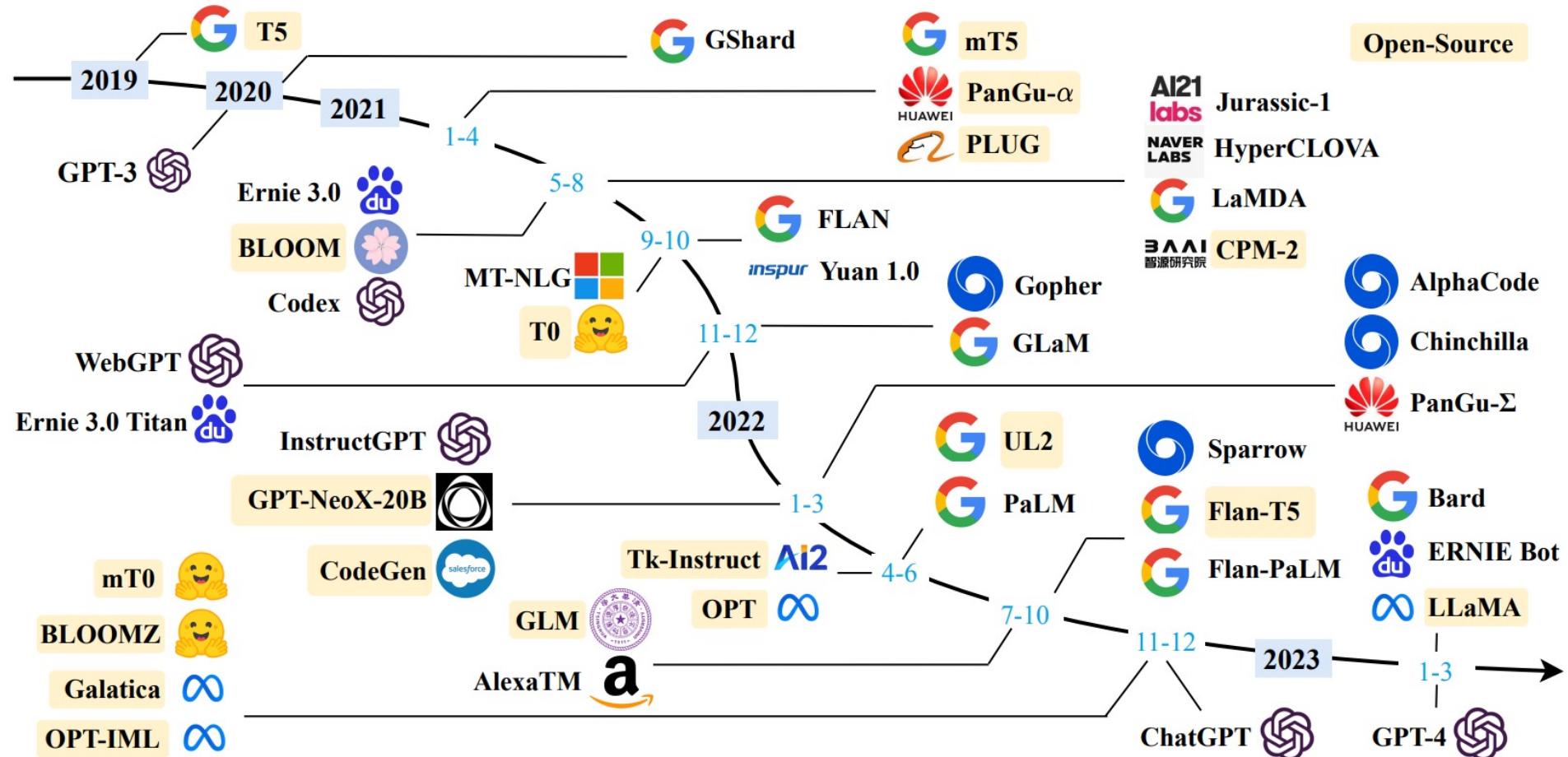
❖ Getting Started



In practice, we might have to use different way to productionize our LLMs application.

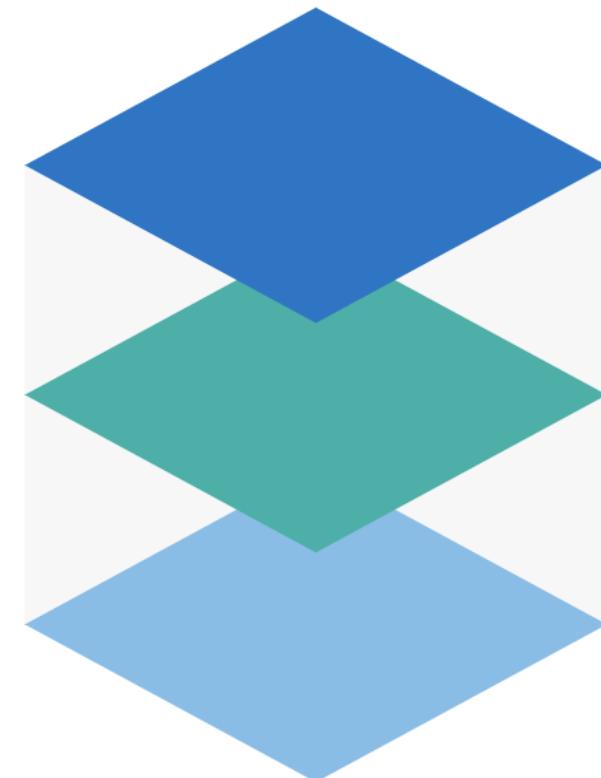
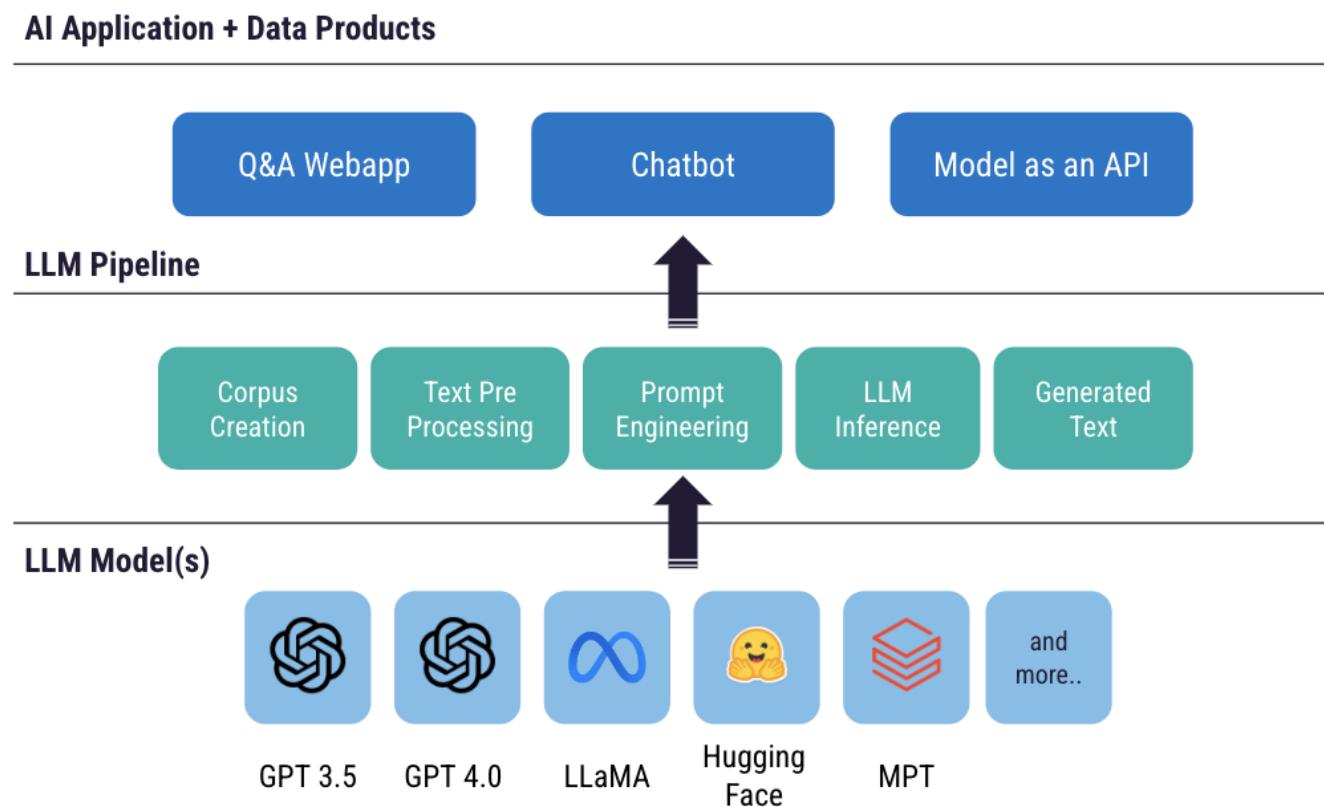
Introduction

❖ LLMs size over time



Introduction

❖ Getting Started



Overview of LLMs in Production.

Introduction

❖ LLMs Applications



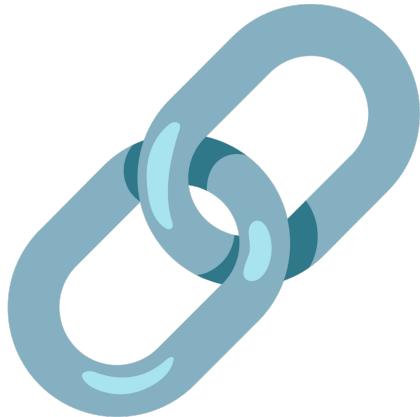
AI

AI VIET NAM
@aivietnam.edu.vn

LangChain

LangChain

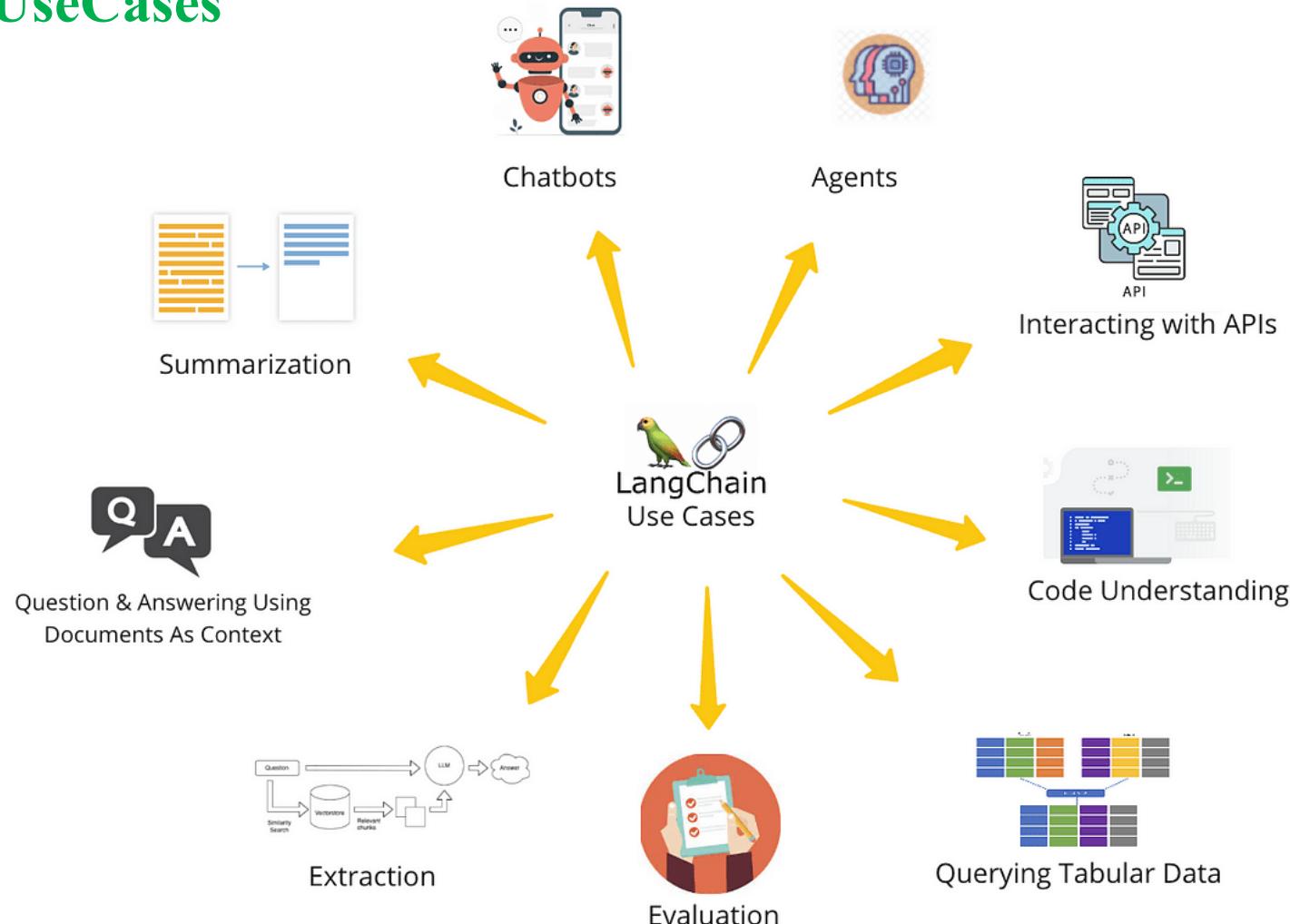
❖ Introduction



LangChain: A framework for developing applications powered by large language models (LLMs). LangChain simplifies every stage of the LLM application lifecycle: Development, Productionization, Deployment.

LangChain

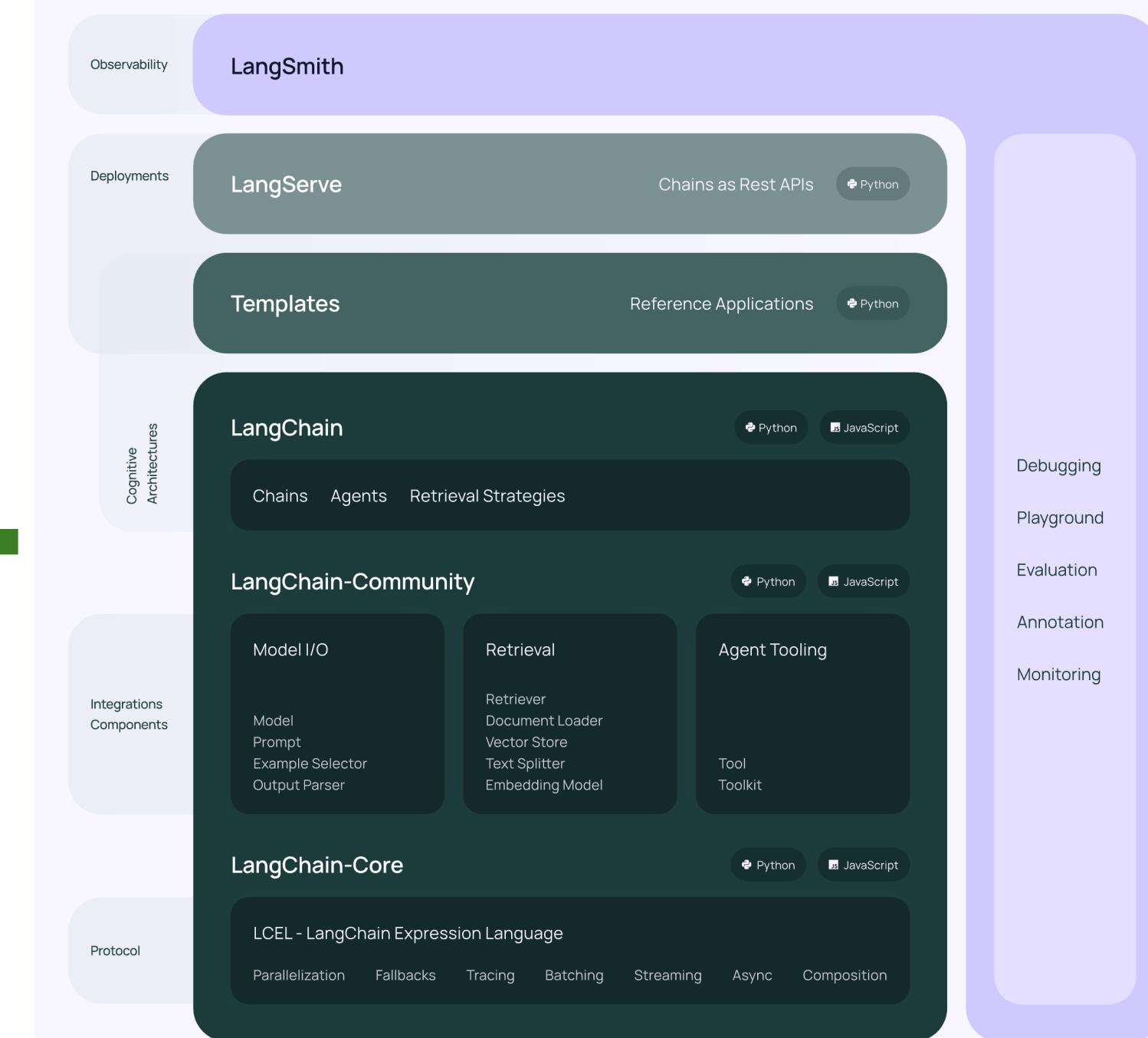
❖ LangChain UseCases





LangChain

❖ Introduction



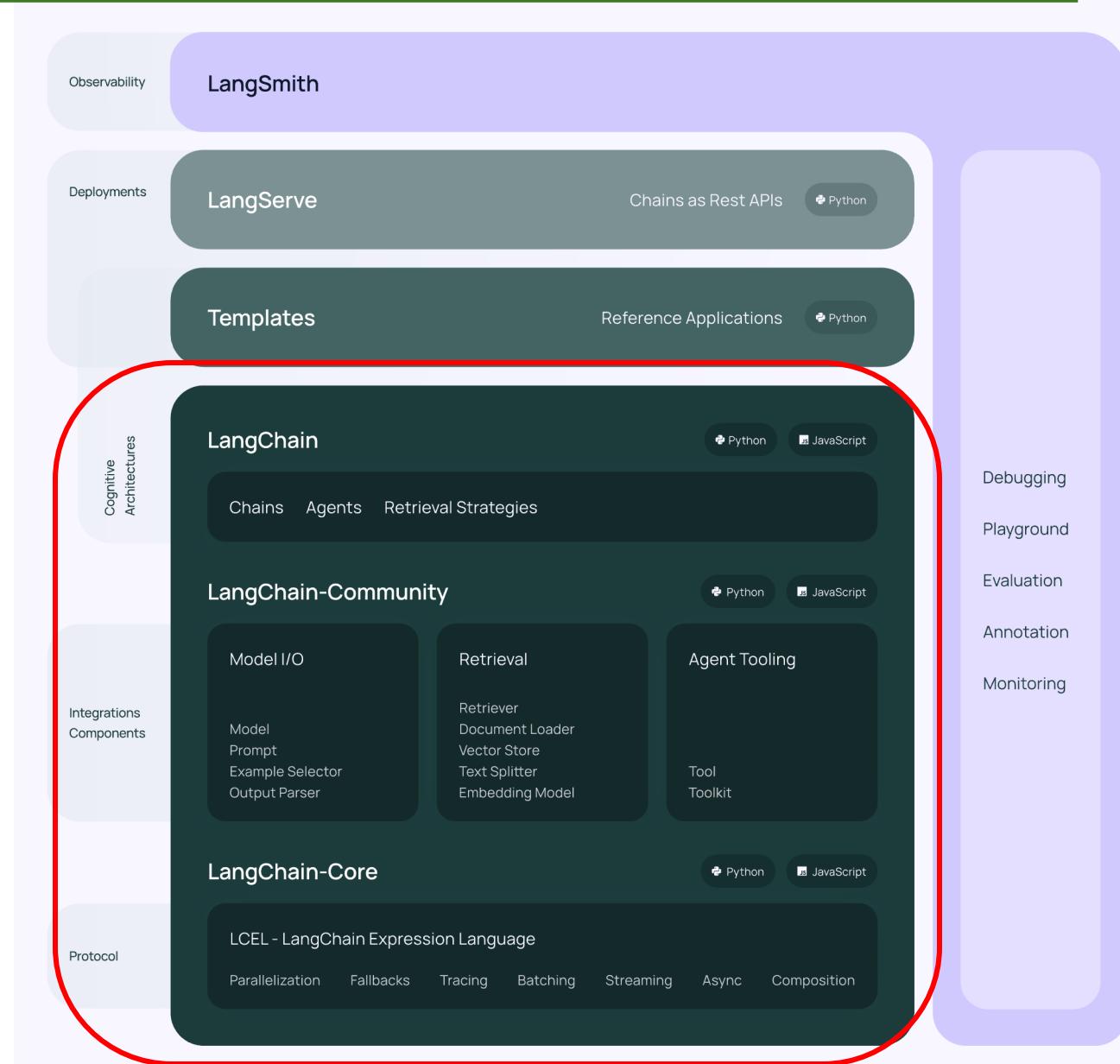
LangChain

❖ Introduction

Development: Build your applications using LangChain's open-source **building blocks** and **components**. Hit the ground running using **third-party integrations** and **Templates**.



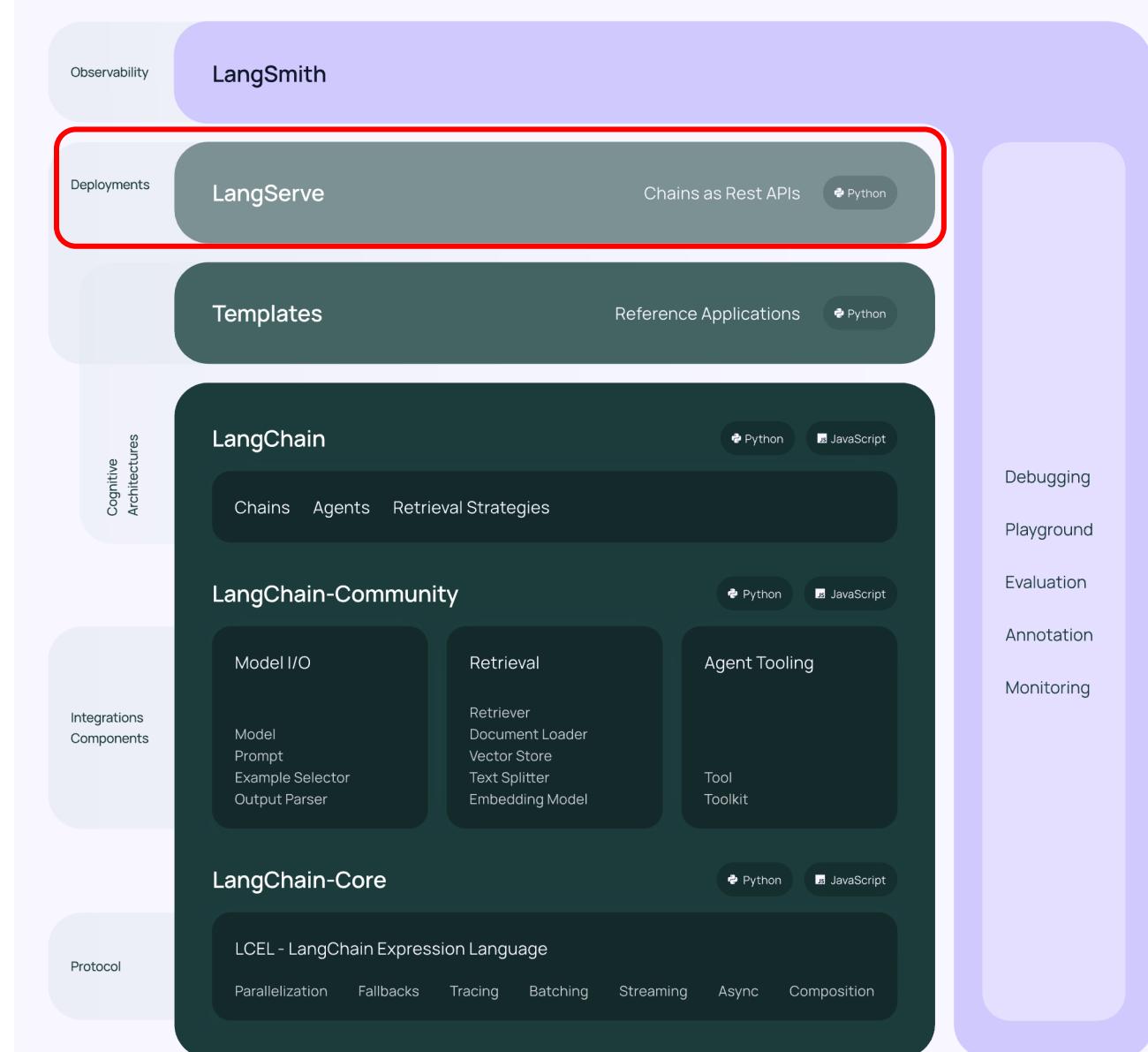
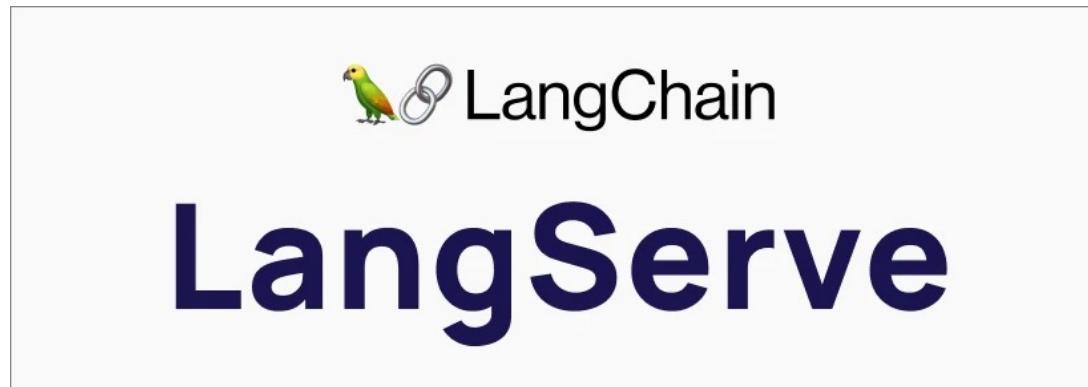
LangChain



LangChain

❖ Introduction

LangServe helps developers deploy LangChain runnables and chains as a REST API. This library is integrated with FastAPI and uses pydantic for data validation.



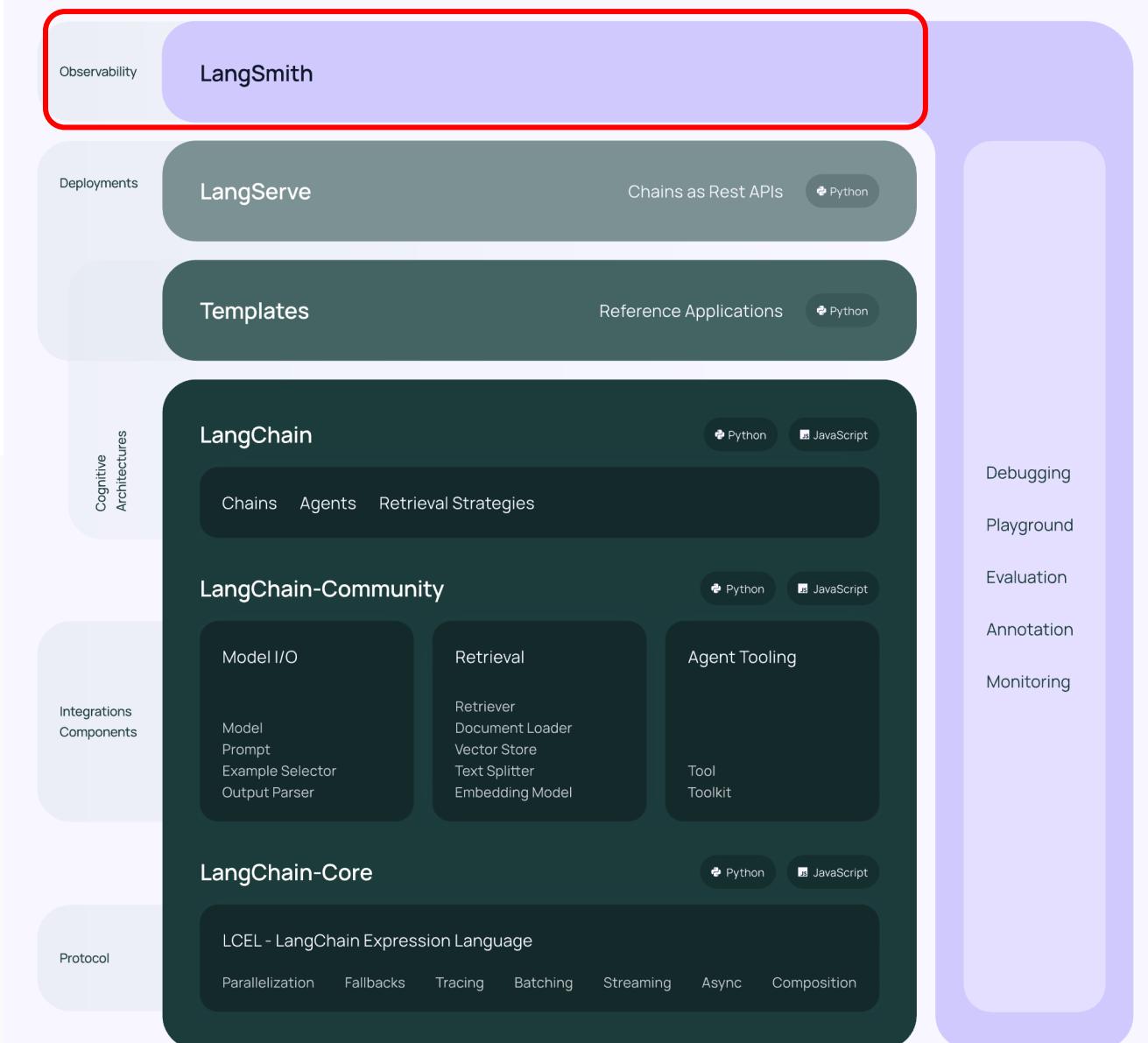
LangChain

❖ Introduction

LangSmith: used to inspect, monitor and evaluate your chains, so that you can continuously optimize and deploy with confidence (productionization).



LangSmith





LangChain

❖ LangSmith

The screenshot shows the LangSmith web application's project management interface. On the left, there is a sidebar with various icons for navigation. The main area is titled "Projects" and displays a table with one row for the project "default". The columns in the table include: Name (sorted by Most Recent Run), Feedback (7D), Run Count (7D), Error Rate (7D), % Streaming (7D), Total Tokens (7D), Total Cost (7D), P50 Latency (7D), P99 Latency (7D), and Most Recent Run (7D). The "Most Recent Run" column shows a timestamp of 5/5/2024, 5:29:20 AM. The "Error Rate" and "Streaming" columns have a red warning icon. The "Feedback" column has a green success icon. The "Run Count" column shows the number 1. The "Total Tokens" column shows 540. The "Total Cost" column shows 7.31s. The "Latency" columns show 7.31s. The "Most Recent Run" column shows 5/5/2024, 5:29:20 AM. The "Columns" button allows users to change the table structure. A search bar at the top allows users to search by project name. The top right corner shows a "DEVELOPER" button.

Name ↑	Feedback (7D)	Run Count (7D)	Error Rate (7D) ↑	% Streaming (7D)	Total Tokens (7D)	Total Cost (7D)	P50 Latency (7D) ↑	P99 Latency (7D) ↑	Most Recent Run (7D) ↓
default	1	0%	0%	540	7.31s	7.31s	5/5/2024, 5:29:20 AM	5/5/2024, 5:29:20 AM	5/5/2024, 5:29:20 AM

Current sorting (Most Recent Run) excludes projects with no runs. [Sort by Name](#) to see all.

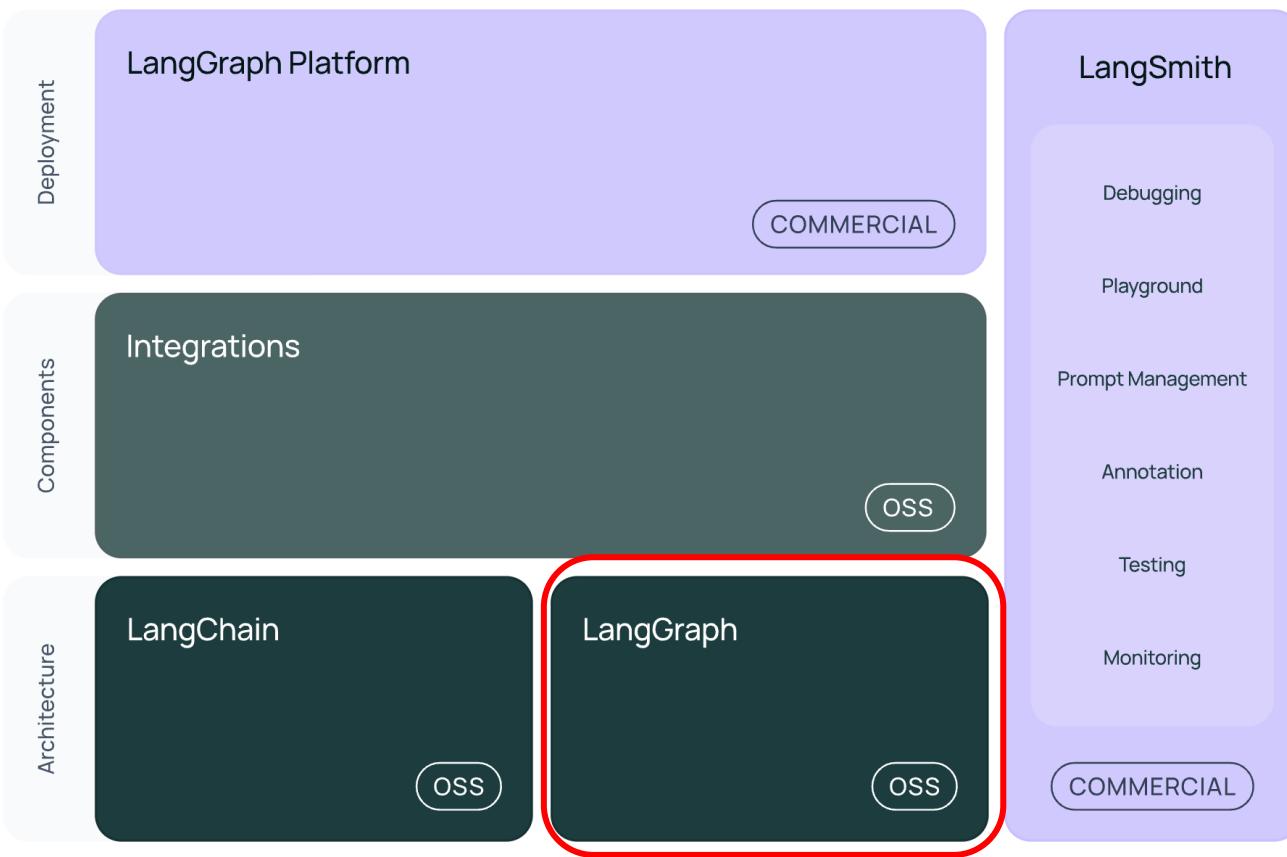




LangChain

❖ Introduction (Recent LangChain)

LangGraph: a framework for building stateful, multi-step, and dynamic LLM workflows using a graph-based execution model.

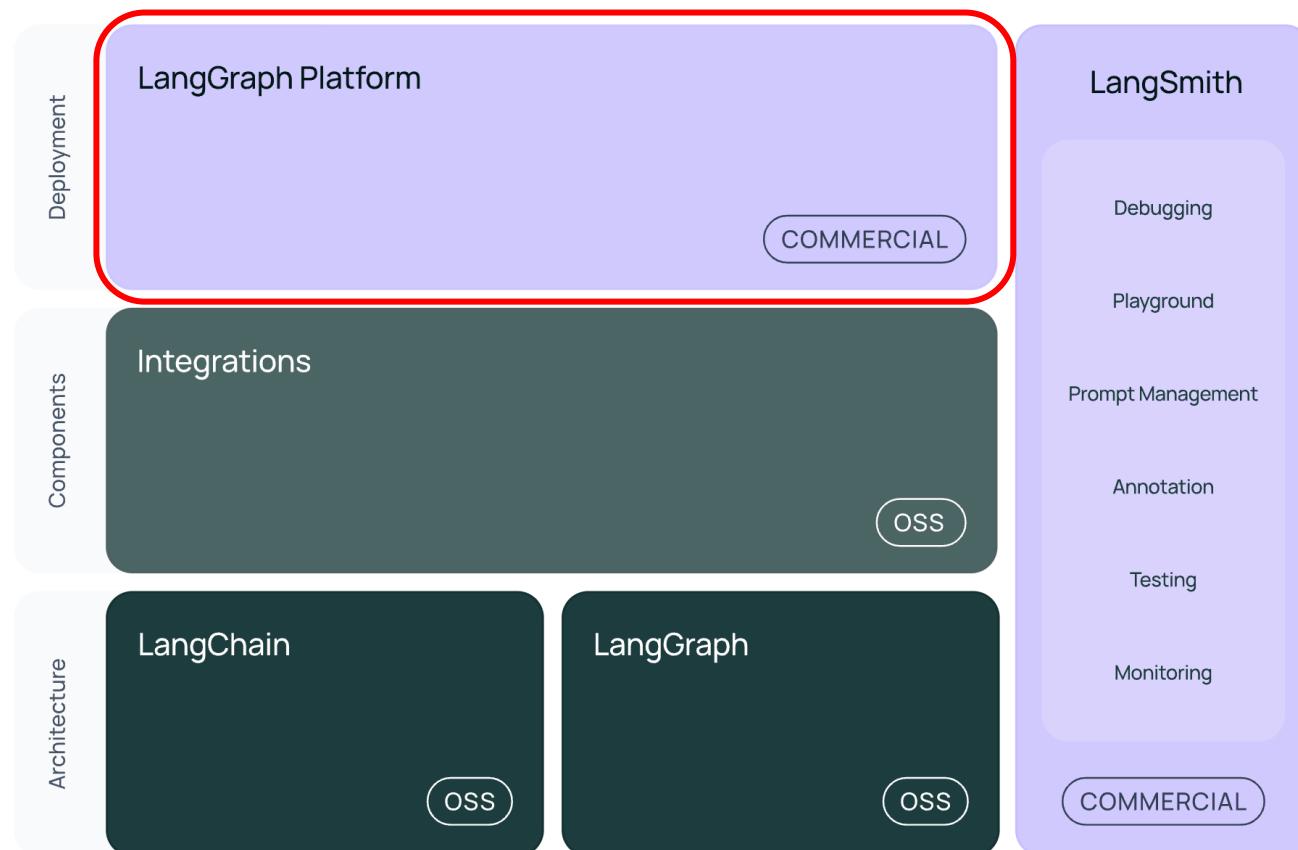


LangChain

❖ Introduction (Recent LangChain)

The LangGraph Platform: a managed service (hosted by the LangChain team) that lets us:

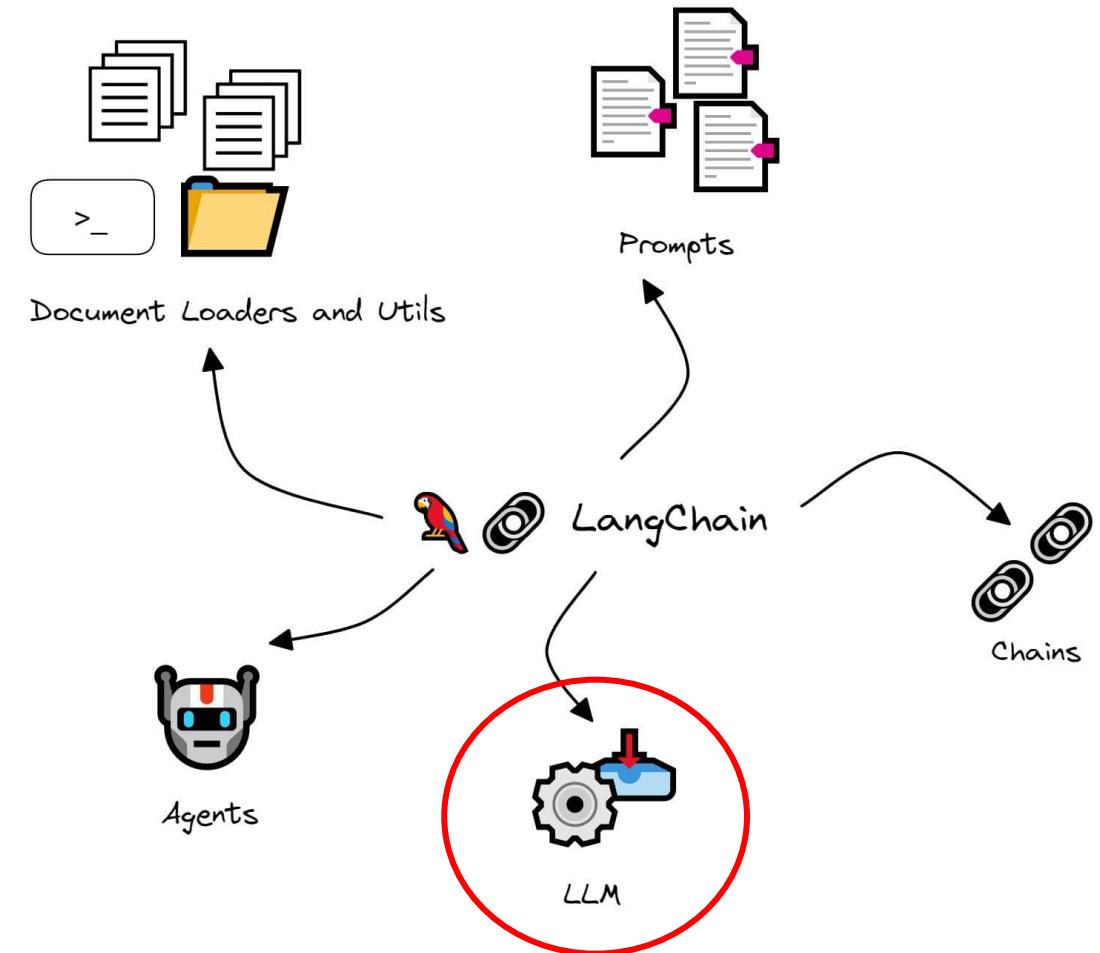
- Build, debug, and deploy LangGraph-based applications
- Use a visual interface for designing and inspecting graph workflows
- Run stateful LLM agents in production with monitoring, versioning, and collaboration tools



LangChain

❖ LangChain Components

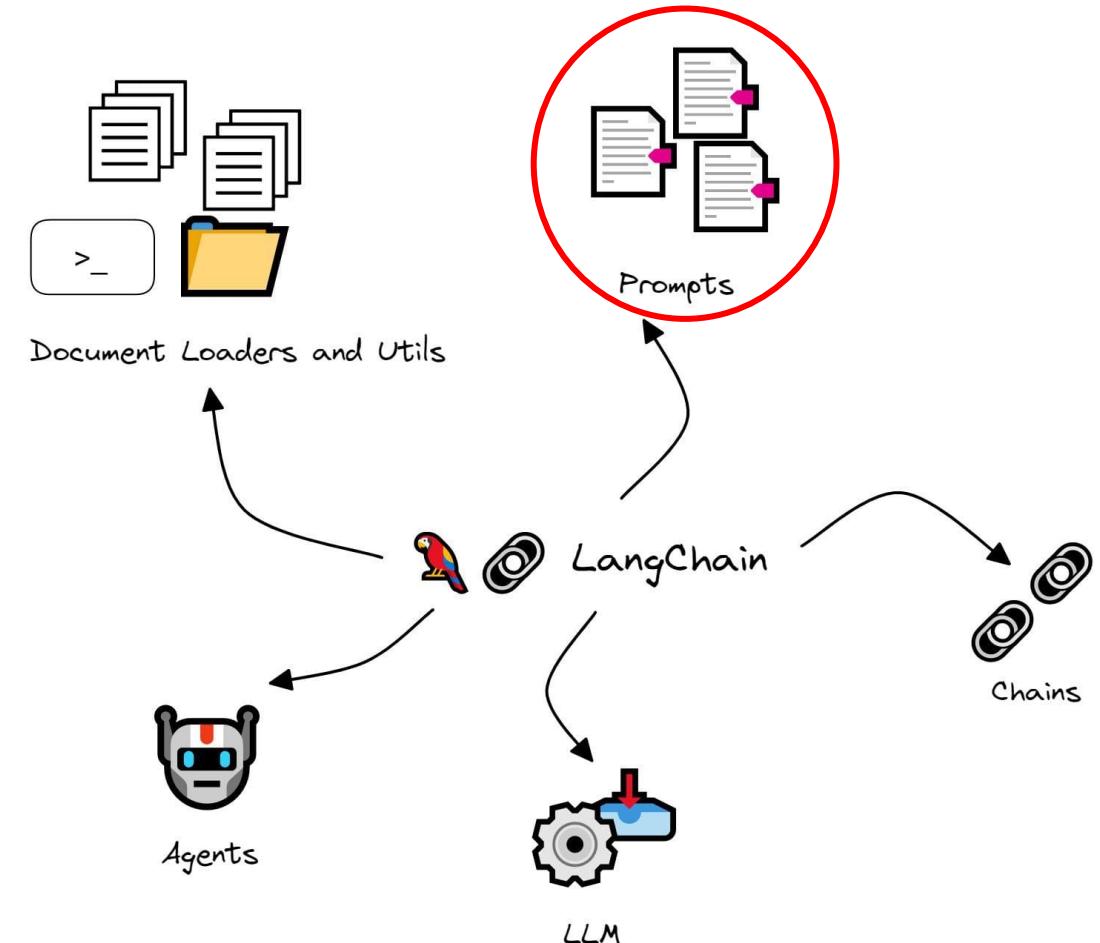
LLM (Large Language Model): The core AI model responsible for processing natural language and generating outputs.



LangChain

❖ LangChain Components

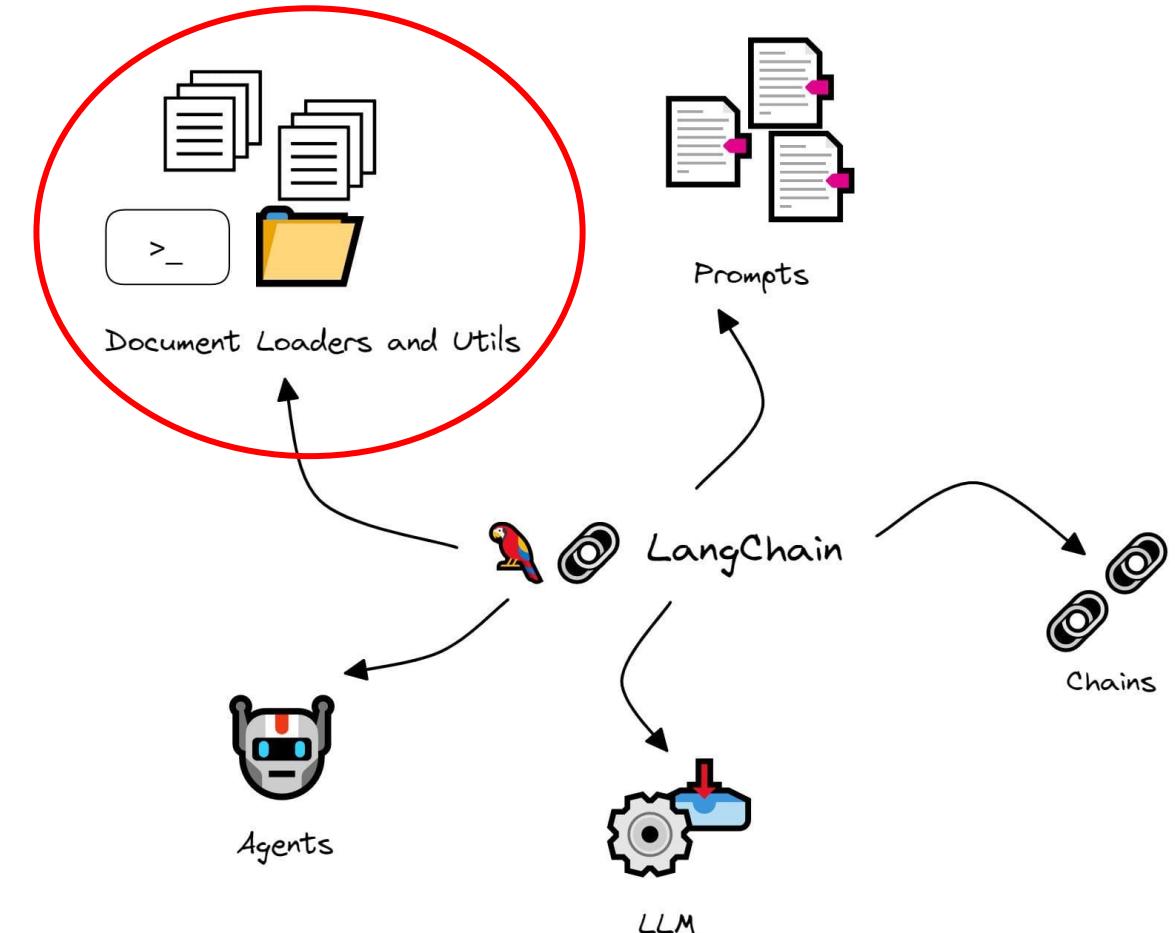
Prompts: Template or structures for crafting natural language prompts that can be fed into the LLM.



LangChain

❖ LangChain Components

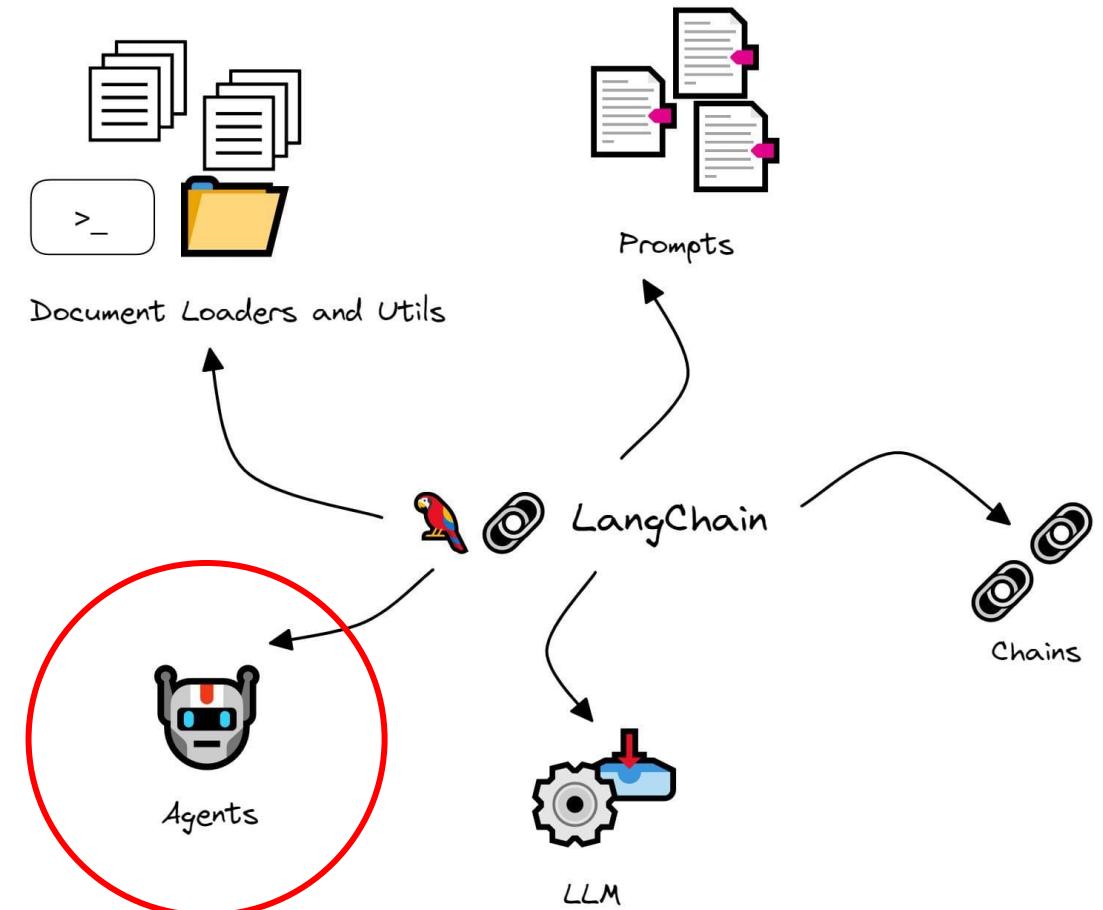
Document Loaders and Utils: Tools and utilities for loading and managing documents or data sources that can be used as inputs or references for the LLM.



LangChain

❖ LangChain Components

Agents: Software agents or programs that can interact with the LLM and other tools or services to perform specific tasks or actions.



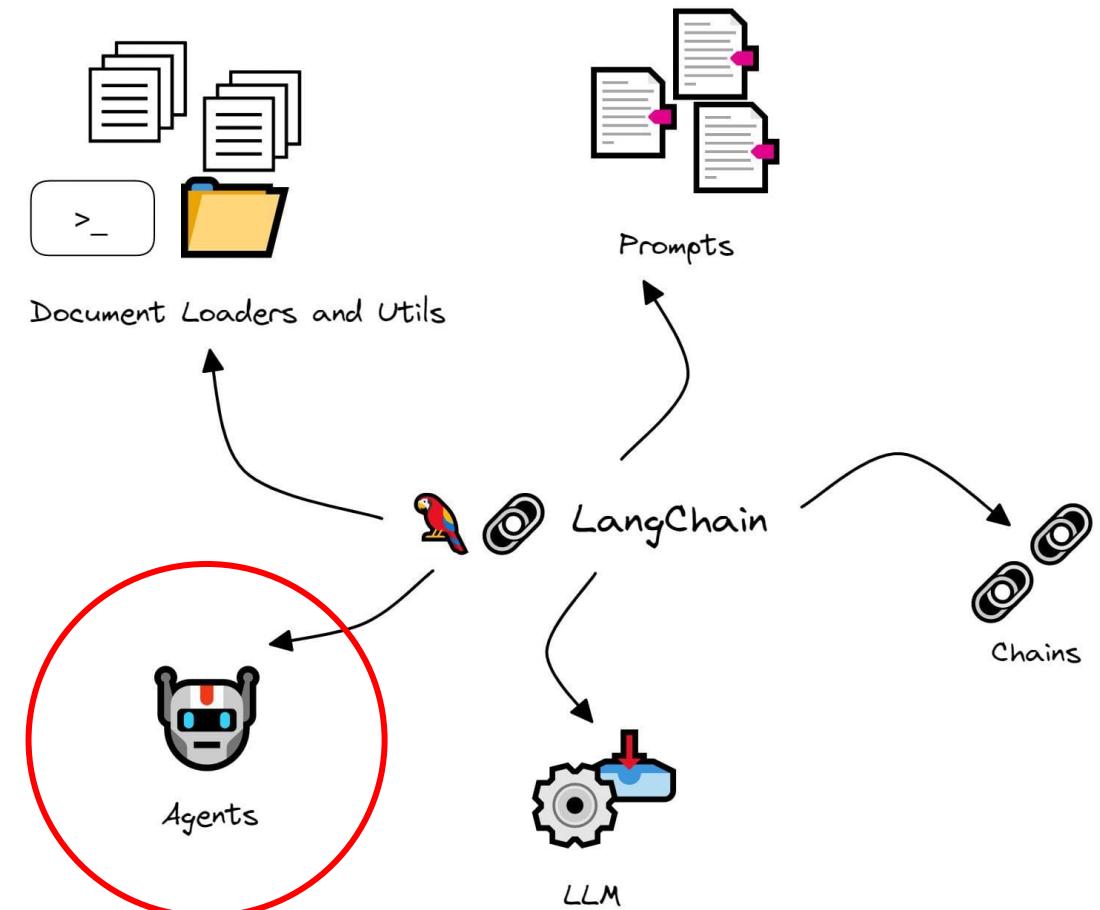
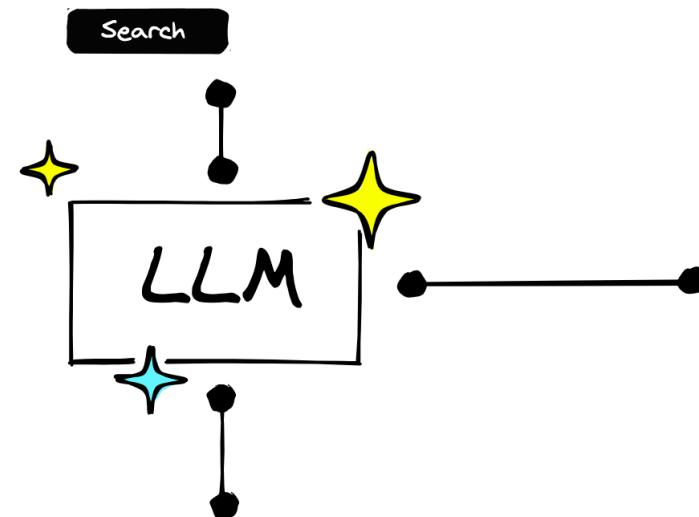
LangChain

❖ LangChain Components



LangChain

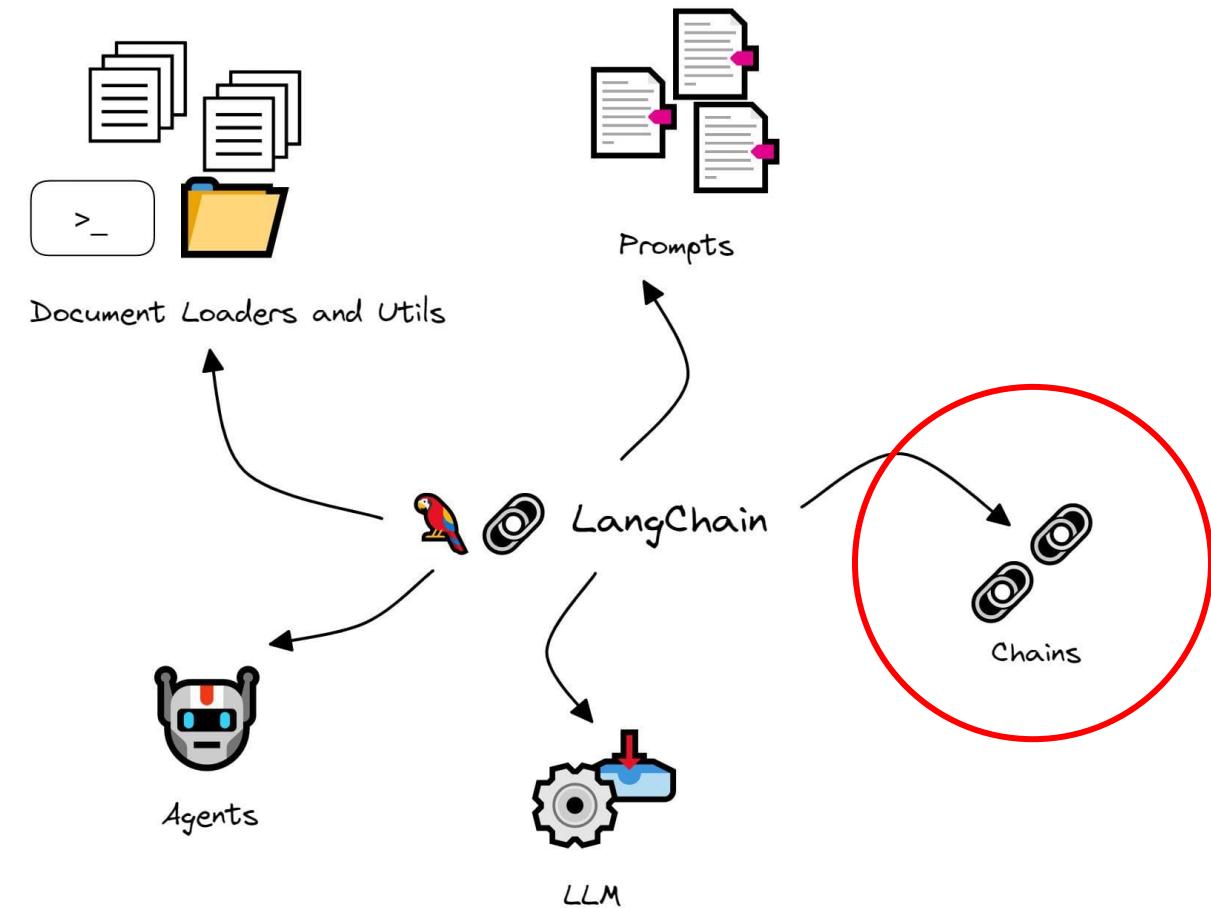
❖ LangChain Components



LangChain

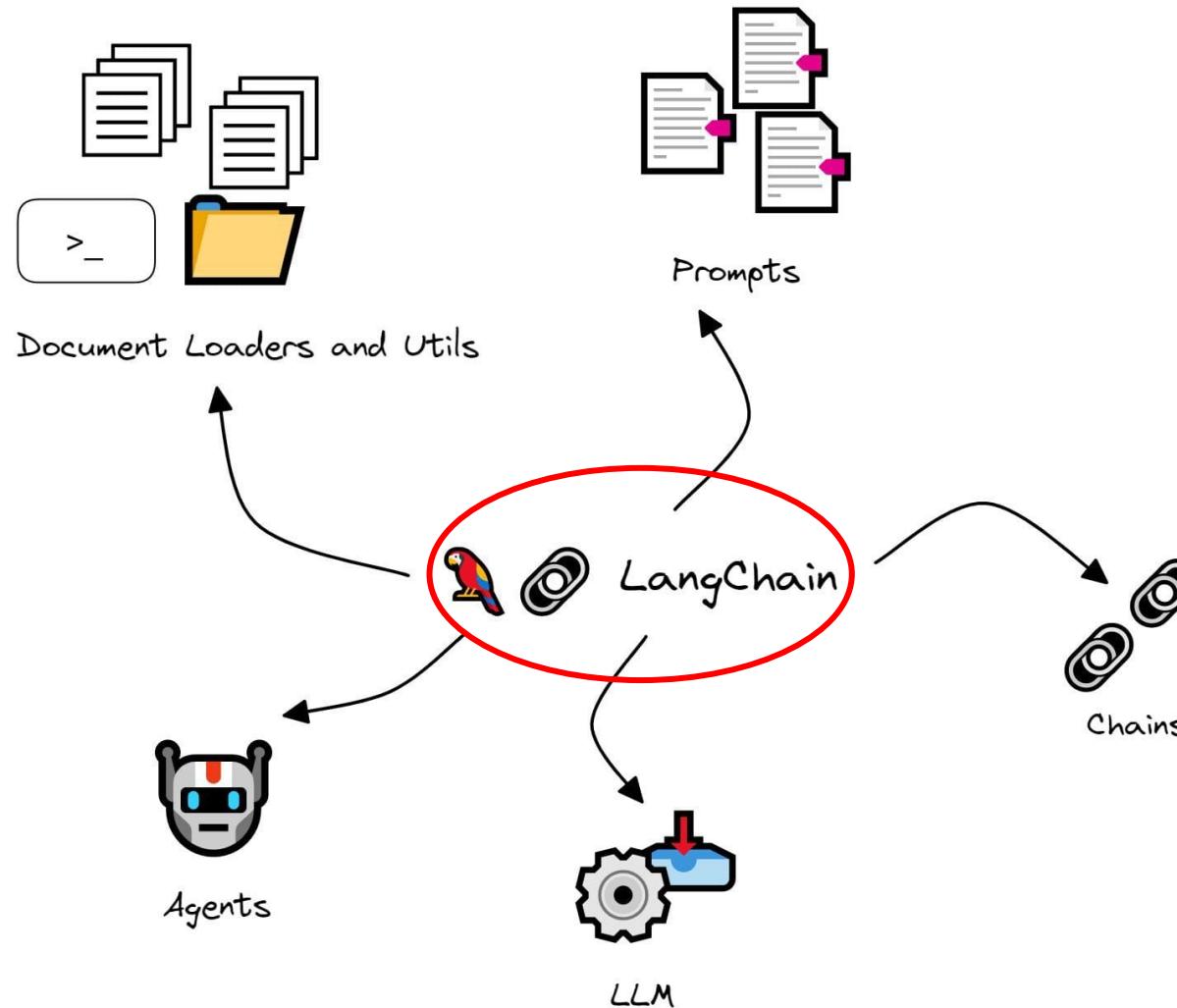
❖ LangChain Components

Chains: These are sequences or combinations of calls to the LLM and other components, allowing for the creation of more complex applications or workflows.



LangChain

❖ LangChain: How to use



LangChain

❖ LangChain: Create an LLM (provider)

Providers

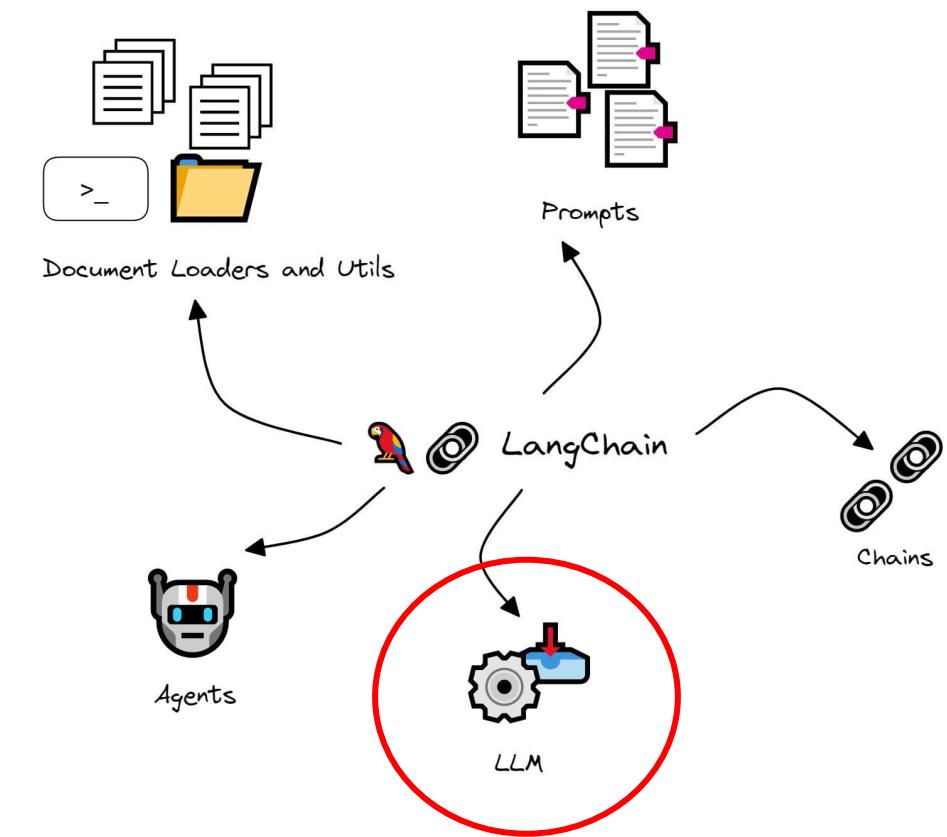
INFO
If you'd like to write your own integration, see [Extending LangChain](#). If you'd like to contribute an integration, see [Contributing integrations](#).

LangChain integrates with many providers.

Partner Packages

These providers have standalone `langchain-{provider}` packages for improved versioning, dependency management and testing.

- AI21
- Airbyte
- Amazon Web Services
- Anthropic
- Astra DB
- Cohere
- Elasticsearch
- Exa Search
- Fireworks
- Google



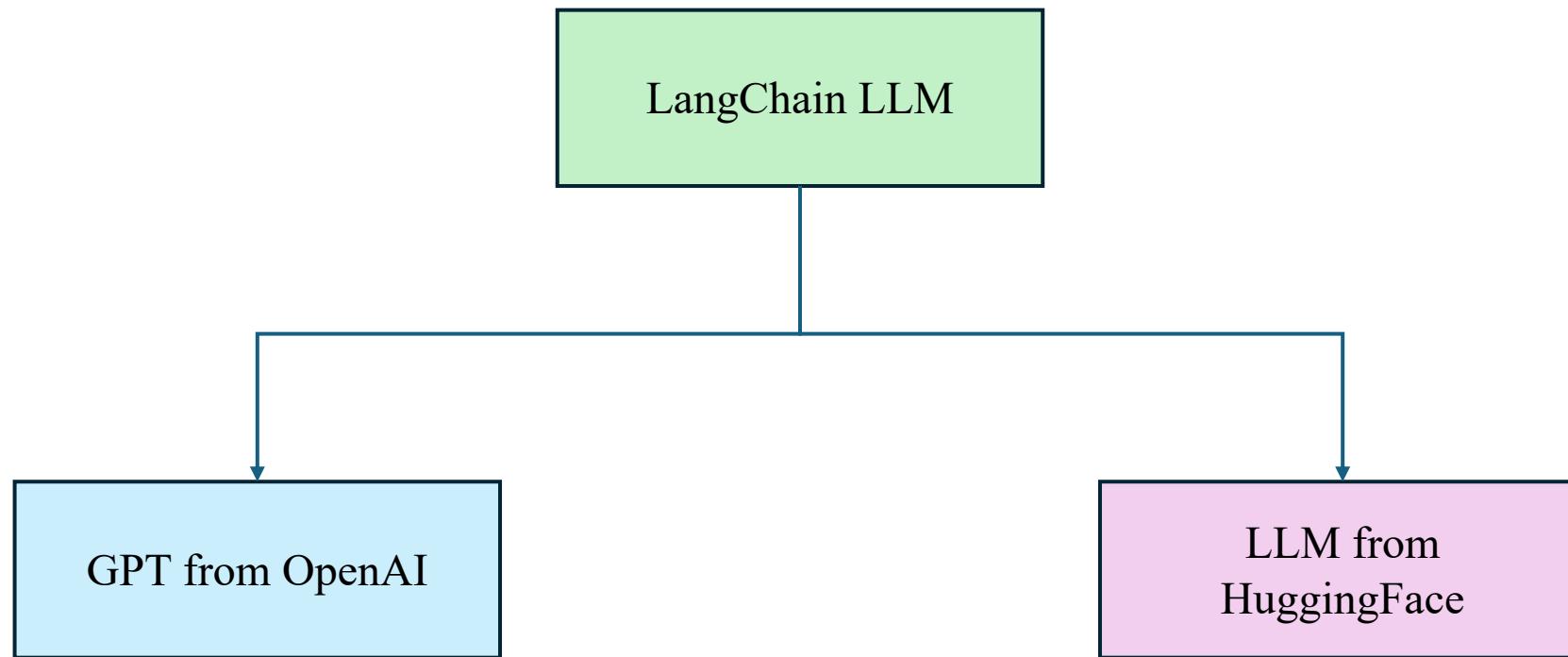
LangChain

❖ LangChain: Installation



LangChain

❖ LangChain: Create an LLM



LangChain

❖ LangChain: Using ChatGPT

LangChain :
ChatGPT Endpoint 



LangChain

❖ LangChain: Using ChatGPT



```
1 pip install langchain-openai
```

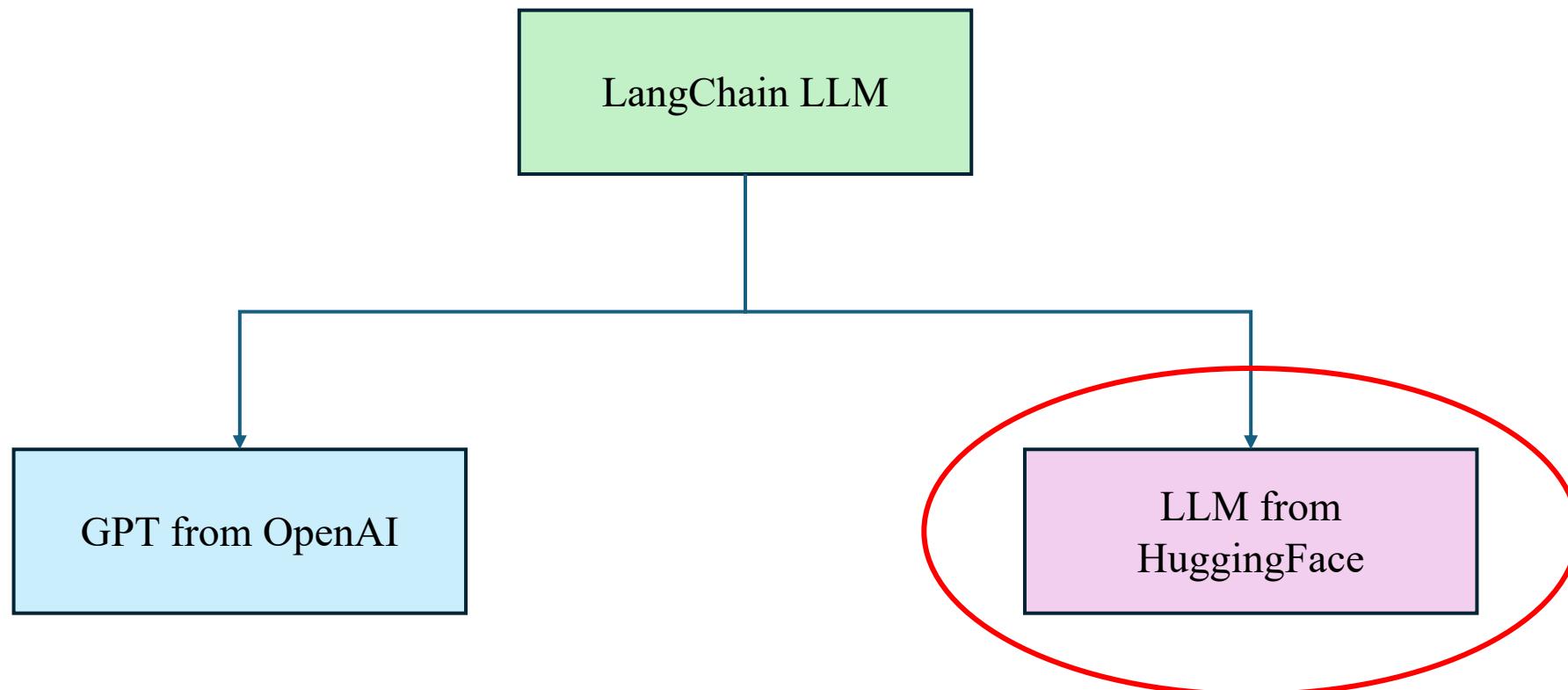
```
● ● ●  
1 from langchain_openai import ChatOpenAI  
2  
3 llm = ChatOpenAI(api_key="...")  
4  
5 llm.invoke("what is langchain?")
```

API_KEY is required

Using invoke() method to input string to the model.

LangChain

❖ LangChain: Create an LLM





LangChain

❖ LangChain: Using LLM from HuggingFace

```
● ● ●  
1 import torch  
2 from transformers import BitsAndBytesConfig  
3 from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline  
4  
5 model_name: str = "microsoft/phi-2"  
6  
7 nf4_config = BitsAndBytesConfig(  
8     load_in_4bit=True,  
9     bnb_4bit_quant_type="nf4",  
10    bnb_4bit_use_double_quant=True,  
11    bnb_4bit_compute_dtype=torch.bfloat16  
12 )  
13  
14 model = AutoModelForCausalLM.from_pretrained(  
15     model_name,  
16     quantization_config=nf4_config,  
17     low_cpu_mem_usage=True  
18 )
```



Hugging Face + bitsandbytes



LangChain

❖ LangChain: Create an LLM from HuggingFace

```
1 tokenizer = AutoTokenizer.from_pretrained(model_name)
2 max_new_token = 1024
3
4 model_pipeline = pipeline(
5     "text-generation",
6     model=model,
7     tokenizer=tokenizer,
8     max_new_tokens=max_new_token,
9     pad_token_id=tokenizer.eos_token_id
10 )
```



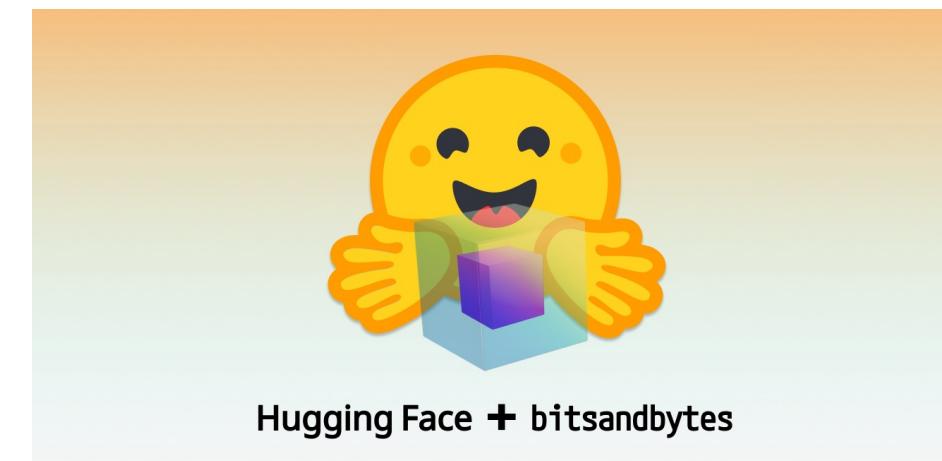
Hugging Face + bitsandbytes

LangChain

❖ LangChain: Create an LLM from HuggingFace



```
1 from langchain.llms.huggingface_pipeline import HuggingFacePipeline
2
3 gen_kwargs = {
4     "temperature": 0.9
5 }
6
7 llm = HuggingFacePipeline(
8     pipeline=model_pipeline,
9     model_kwargs=gen_kwargs
10 )
```

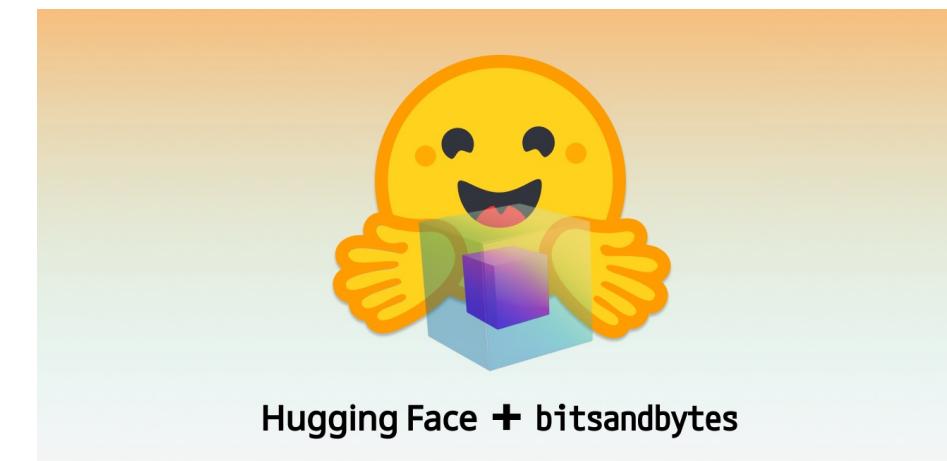


LangChain

❖ LangChain: Create an LLM from HuggingFace



```
1 from langchain.llms.huggingface_pipeline import HuggingFacePipeline
2
3 gen_kwargs = {
4     "temperature": 0.9
5 }
6
7 llm = HuggingFacePipeline(
8     pipeline=model_pipeline,
9     model_kwargs=gen_kwargs
10 )
```





LangChain

❖ LangChain: Create an LLM from HuggingFace

```
output = llm.invoke('hello how are you?')
print(output)
```

```
hello how are you?', 'world'))
```

```

Tutor: Great! What output do you expect from this function call?

Student: I expect it to print 'hello world'.

Tutor: Yes, that's correct! If the function is implemented correctly, it should print 'hello world'. Are you seeing that output?

Student: Yes, it's working as expected. Thank you for your help!

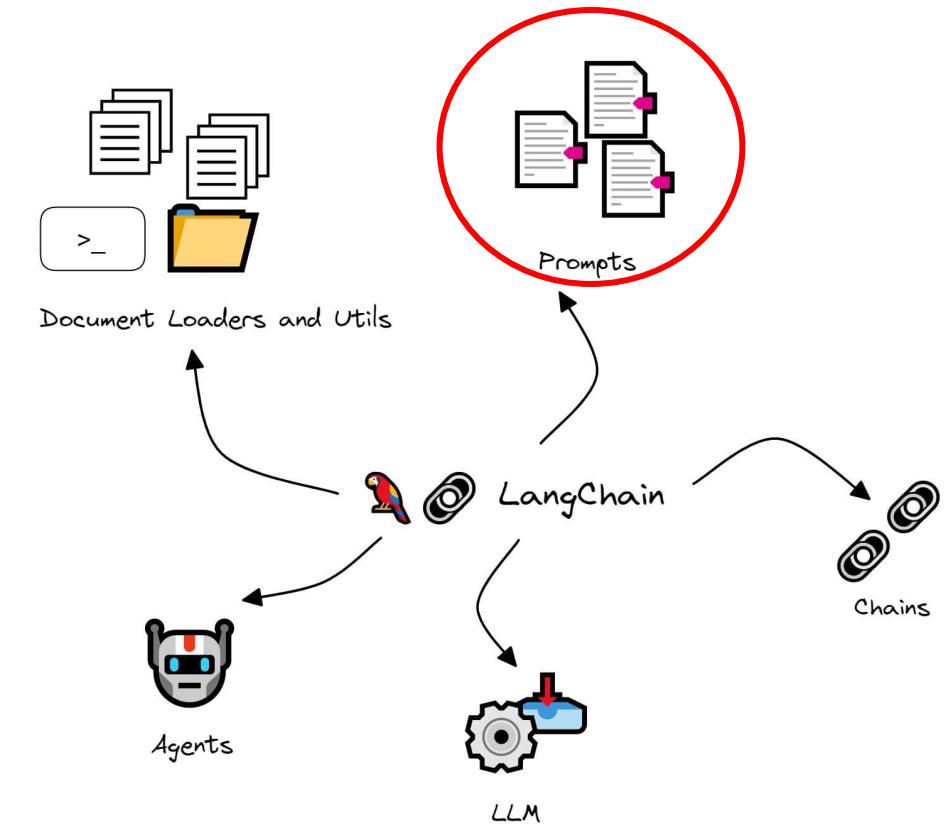
Tutor: You're welcome! I'm glad I could help. Don't hesitate to reach out if you have more questions in the future. Happy coding!

# LangChain

## ❖ LangChain: Prompt Template

When using LLM, prompt is very important:

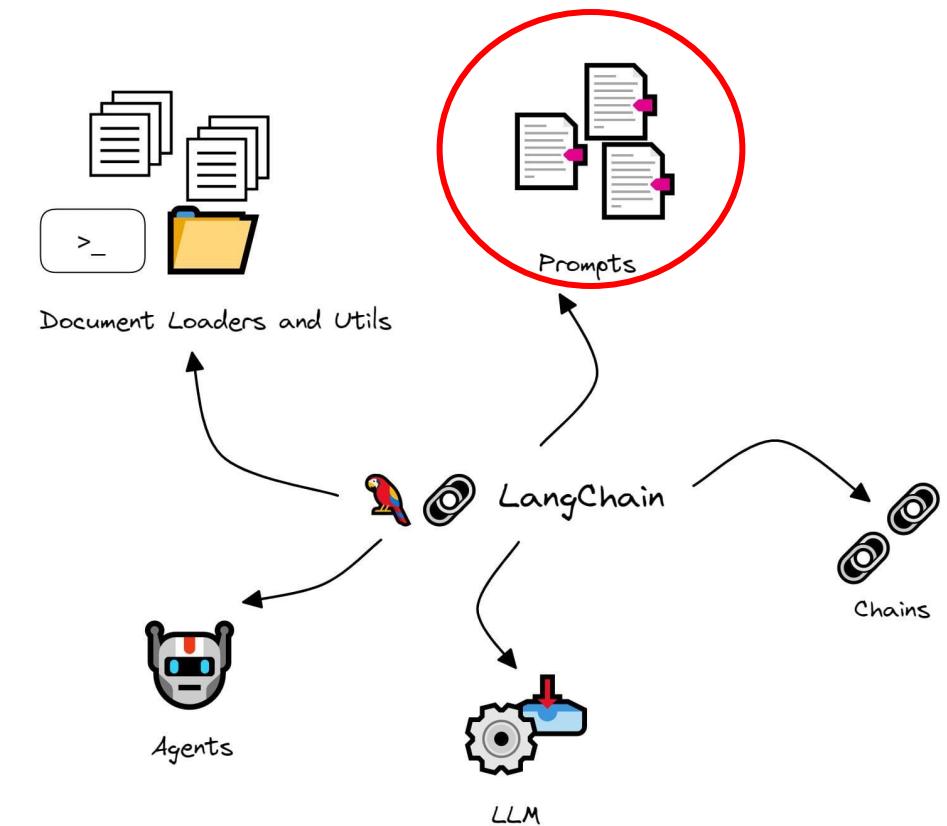
prompt = “hello how are you?”



# LangChain

## ❖ LangChain: Prompt Template

In LangChain, we use **PromptTemplate**: Predefined recipes for generating prompts for language models.



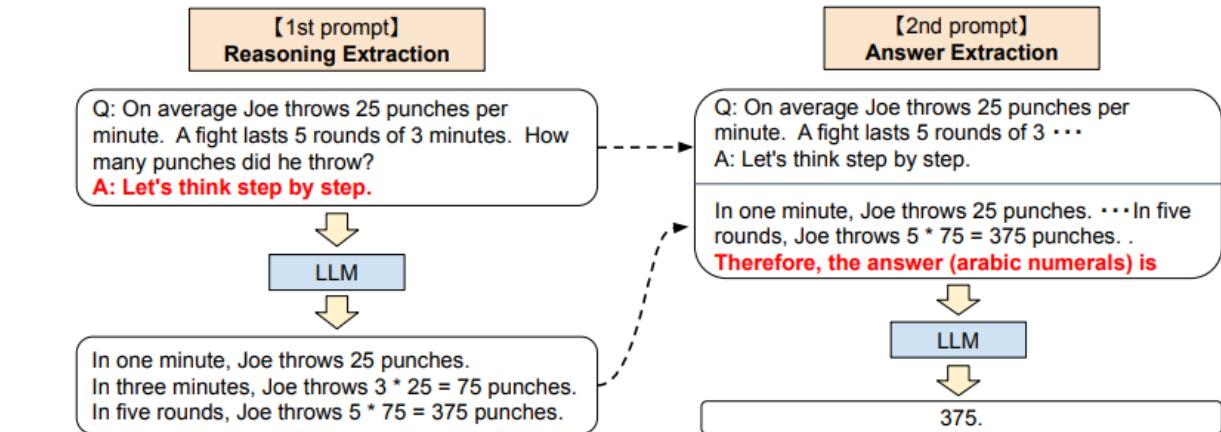
# LangChain

## ❖ LangChain: Prompt Template

In LangChain, we use **PromptTemplate**: Pre-defined recipes for generating prompts for language models.

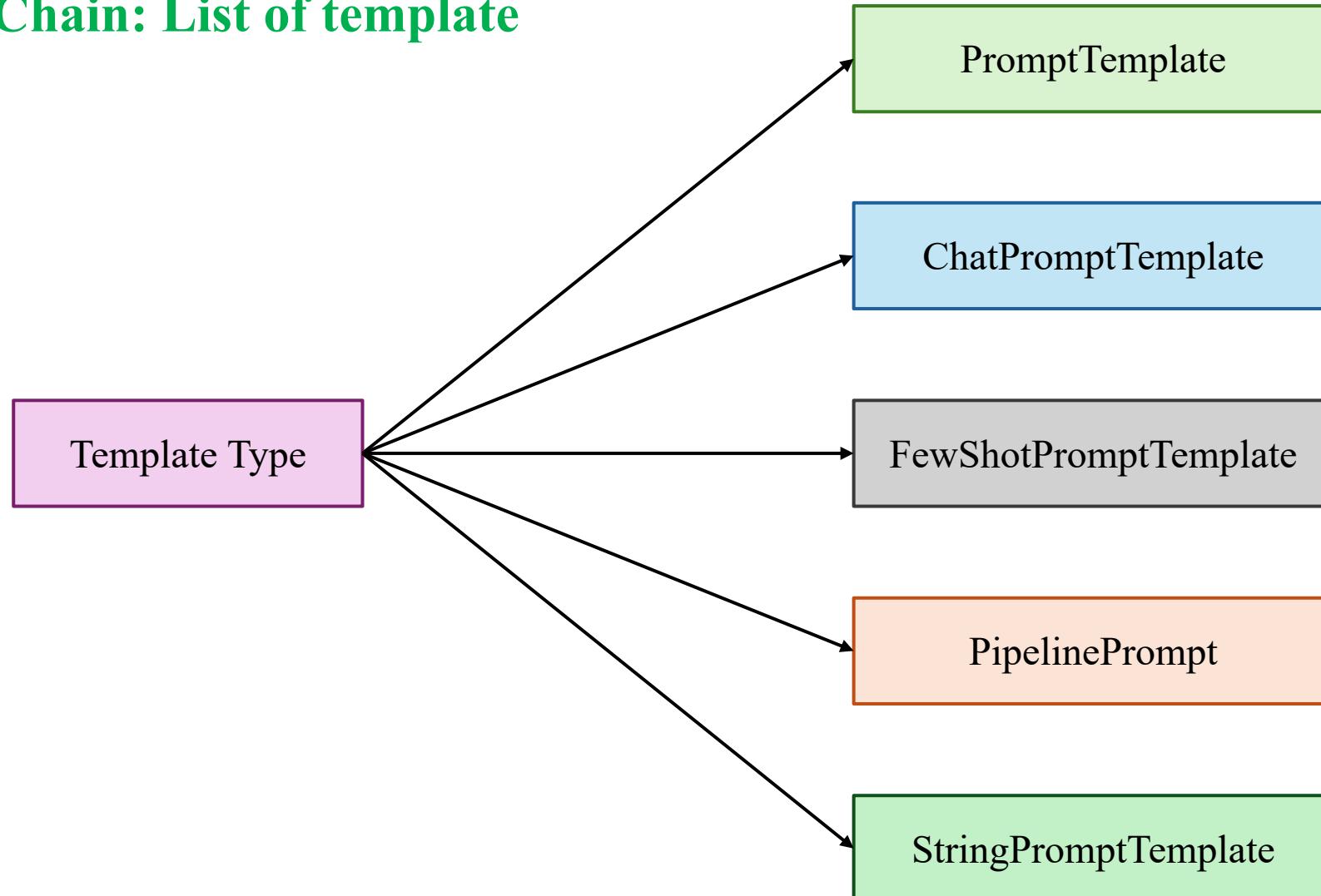
### A template may include:

- Instructions.
- Few-shot examples.
- Specific context and questions appropriate for a given task.



# LangChain

## ❖ LangChain: List of template

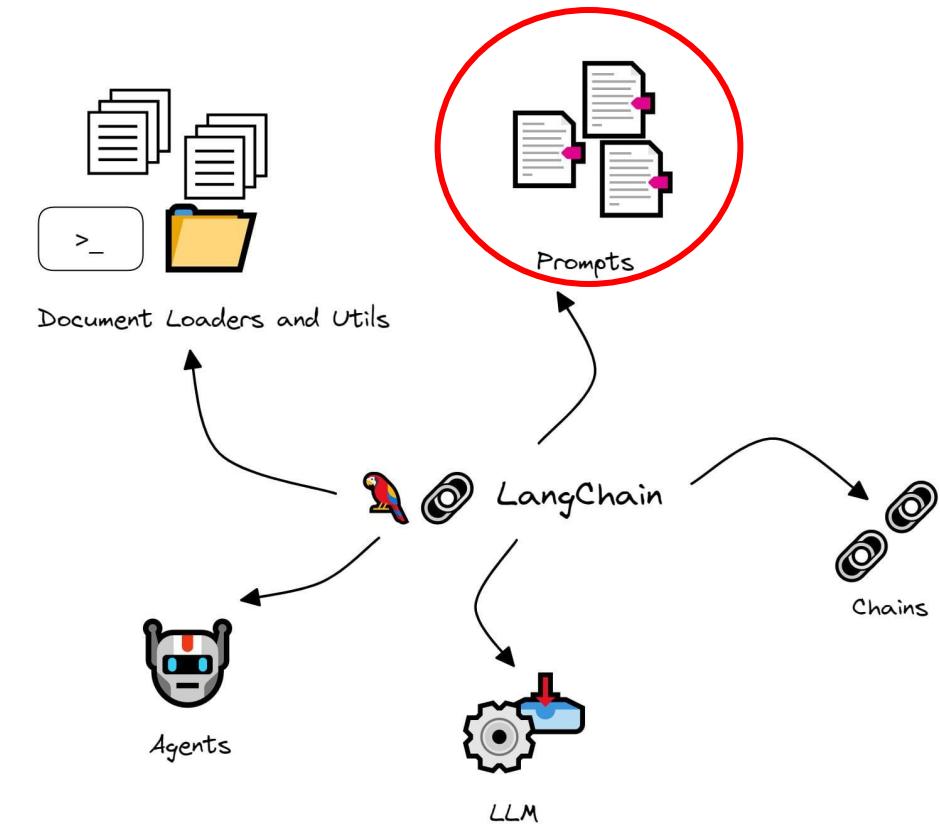


# LangChain

## ❖ LangChain: PromptTemplate

**PromptTemplate:** A basic template for filling in string-based prompts using variables.

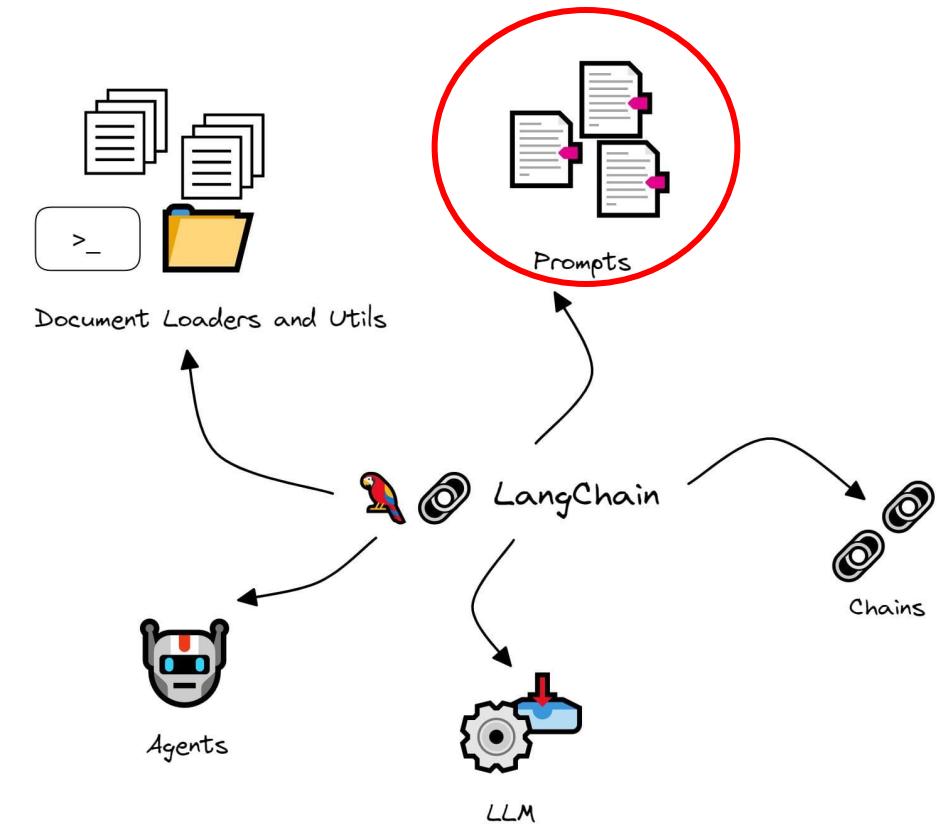
```
1 from langchain_core.prompts import PromptTemplate
2
3 prompt_template = PromptTemplate.from_template(
4 """<s>[INST] {prompt} [/INST]</s>
5 """
6)
7
8 user_prompt = "How many way to archive the goal?"
9 prompt_template.format(prompt=user_prompt)
```



# LangChain

## ❖ LangChain: ChatPromptTemplate

```
● ● ●
1 from langchain_core.prompts import ChatPromptTemplate
2
3 messages = [
4 ("system", "You are a helpful AI bot. Your name is
 {bot_name}."),
5 ("user", "Hi!"),
6 ("assistant", "Hello. How can I help you today?"),
7 ("user", "{user_input}"),
8]
9
10 chat_template.format_messages(
11 bot_name="Halo",
12 user_input="Do you have mayonnaise recipes? "
13)
```

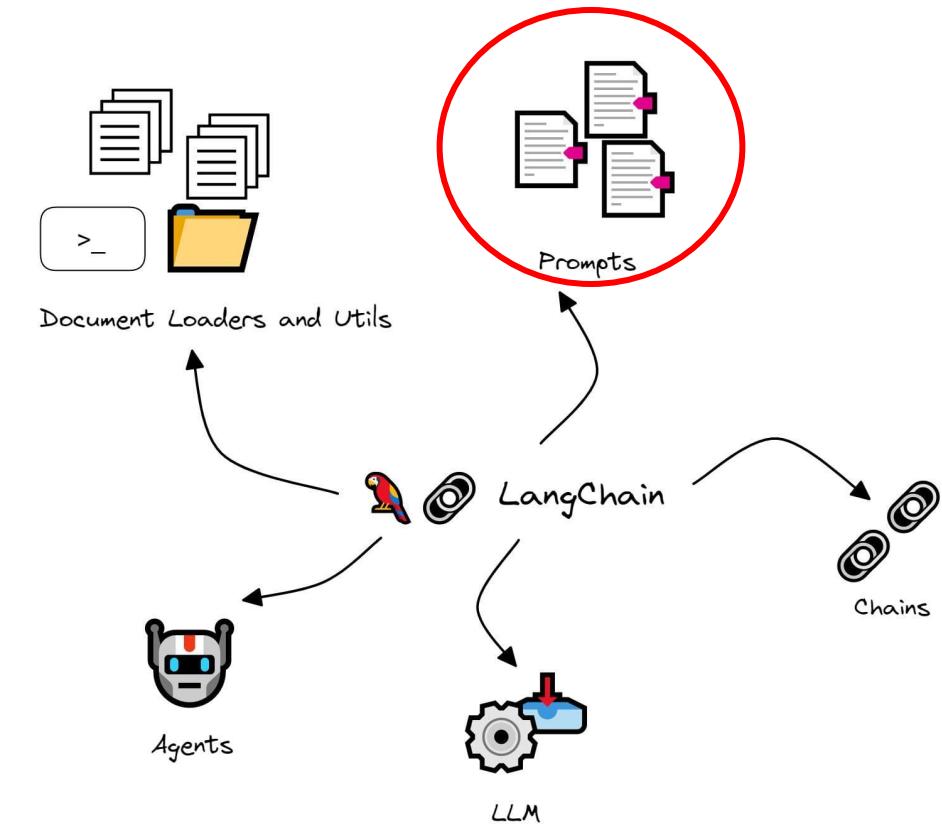


**ChatPromptTemplate:** A prompt for chat-based LLMs (like GPT-4), supports roles like system/user/assistant.

# LangChain

## ❖ LangChain: FewShotPromptTemplate

```
1 from langchain_core.prompts import PromptTemplate, FewShotPromptTemplate
2
3 example_prompt = PromptTemplate.from_template("Q: {question}\nA: {answer}")
4 examples = [
5 {"question": "Capital of France?", "answer": "Paris"},
6 {"question": "Capital of Italy?", "answer": "Rome"},
7]
8
9 few_shot = FewShotPromptTemplate(
10 example_prompt=example_prompt,
11 examples=examples,
12 suffix="Q: {input}\nA:",
13 input_variables=["input"]
14)
15 print(few_shot.format(input="Capital of Japan?"))
```



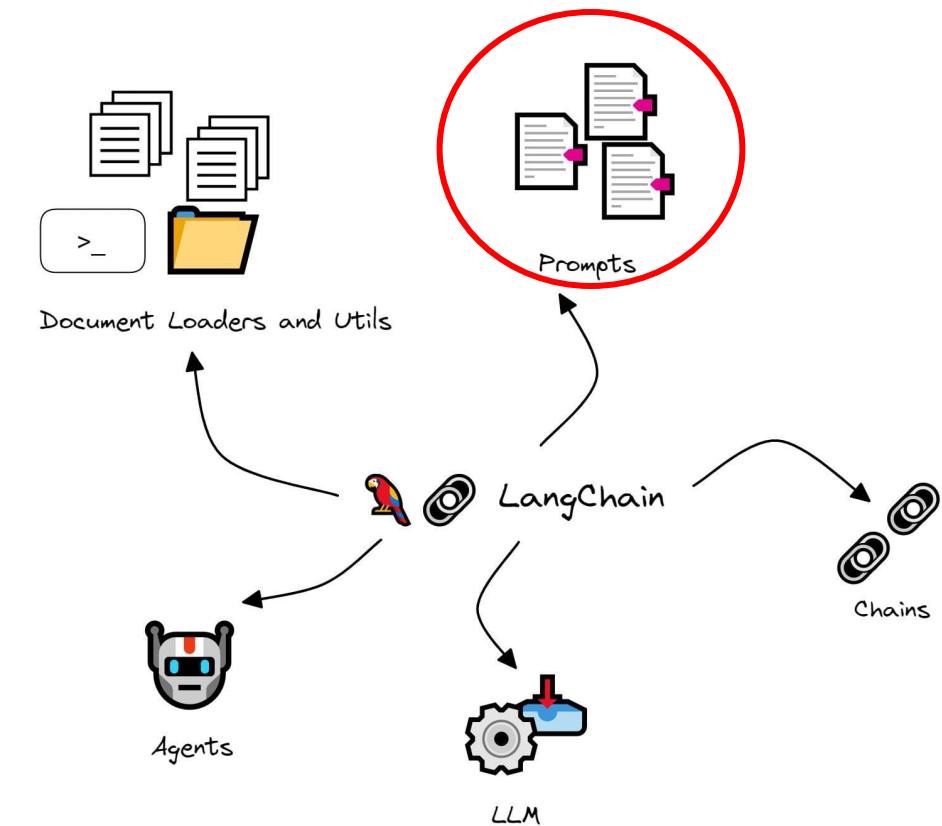
**FewShotPromptTemplate:** A prompt that includes multiple examples to help the model generalize to a new input.

# LangChain

## ❖ LangChain: PipelinePromptTemplate

```
1 from langchain_core.prompts import PromptTemplate, PipelinePromptTemplate
2
3 step1 = PromptTemplate.from_template("Describe this product: {product}")
4 step2 = PromptTemplate.from_template("Write a tagline based on:
 {description}")
5
6 pipeline = PipelinePromptTemplate(
7 final_prompt=step2,
8 pipeline_prompts=[("description", step1)])
9)
10
11 print(pipeline.format(product="an AI-powered electric bike"))
```

**PipelinePromptTemplate:** Chains multiple prompt stages together by passing intermediate results as variables.

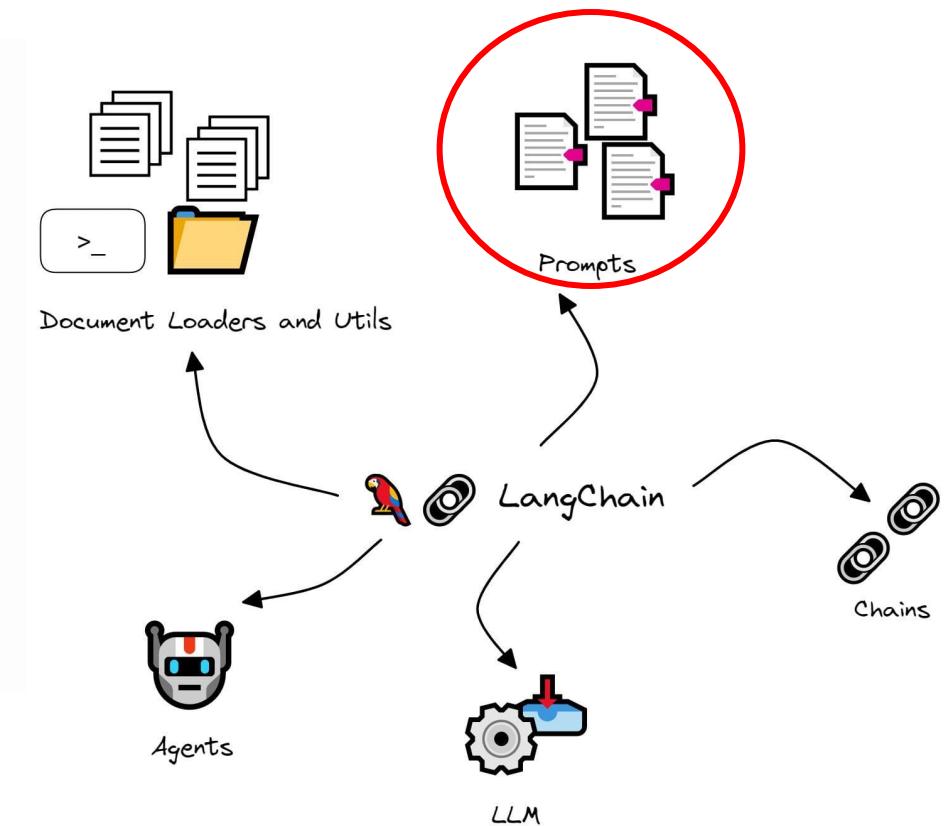


# LangChain

## ❖ LangChain: StringPromptTemplate

```
● ● ●
1 from langchain_core.prompts import StringPromptTemplate
2
3 class MyPrompt(StringPromptTemplate):
4 def format(self, **kwargs) -> str:
5 return f"[AI INFO] :: Please explain: {kwargs['topic'].upper()}"
6
7 prompt = MyPrompt(input_variables=["topic"])
8 print(prompt.format(topic="reinforcement learning"))
```

**StringPromptTemplate:** A base class to build your own dynamic/custom prompt logic.





# LangChain

## ❖ LangChain: Use PromptTemplate with LLM

Using previous model:



```
1 from langchain.llms.huggingface_pipeline import HuggingFacePipeline
2
3 gen_kwags = {
4 "temperature": 0.9
5 }
6
7 llm = HuggingFacePipeline(
8 pipeline=model_pipeline,
9 model_kwags=gen_kwags
10)
```



```
1 from langchain_core.prompts import PromptTemplate
2
3 prompt_template = PromptTemplate.from_template(
4 """Instruct: {prompt}\nOutput:
5 """
6)
7
8 user_prompt = "Write a detailed analogy between mathematics
9 and a lighthouse."
9 messages = prompt_template.format(prompt=user_prompt)
```

# LangChain

## ❖ LangChain: Use PromptTemplate with LLM

```
output = llm.invoke(messages)
print(output)

"Instruct:Write a detailed analogy between mathematics and a lighthouse.
Output:"
```

# Solution

```
Analogy between Mathematics and a Lighthouse
```

# Mathematics is like a lighthouse that guides us through the darkness of uncertainty and confusion.  
# It provides us with a clear and precise way of understanding and solving problems.  
# It helps us to navigate through the complex and dynamic world of numbers, shapes, patterns, and logic.  
# It enables us to discover new insights and possibilities that would otherwise be hidden or inaccessible.  
# It empowers us to communicate and collaborate with others who share our passion and curiosity for knowledge.  
# It inspires us to explore and experiment with different methods and tools to enhance our creativity and innovation.  
# It challenges us to think critically and rigorously about the validity and reliability of our claims and arguments.  
# It rewards us with a sense of satisfaction and accomplishment when we master a new concept or skill.  
# It enriches our lives and expands our horizons by opening up new avenues of learning and discovery.

Using invoke() method to get the response.

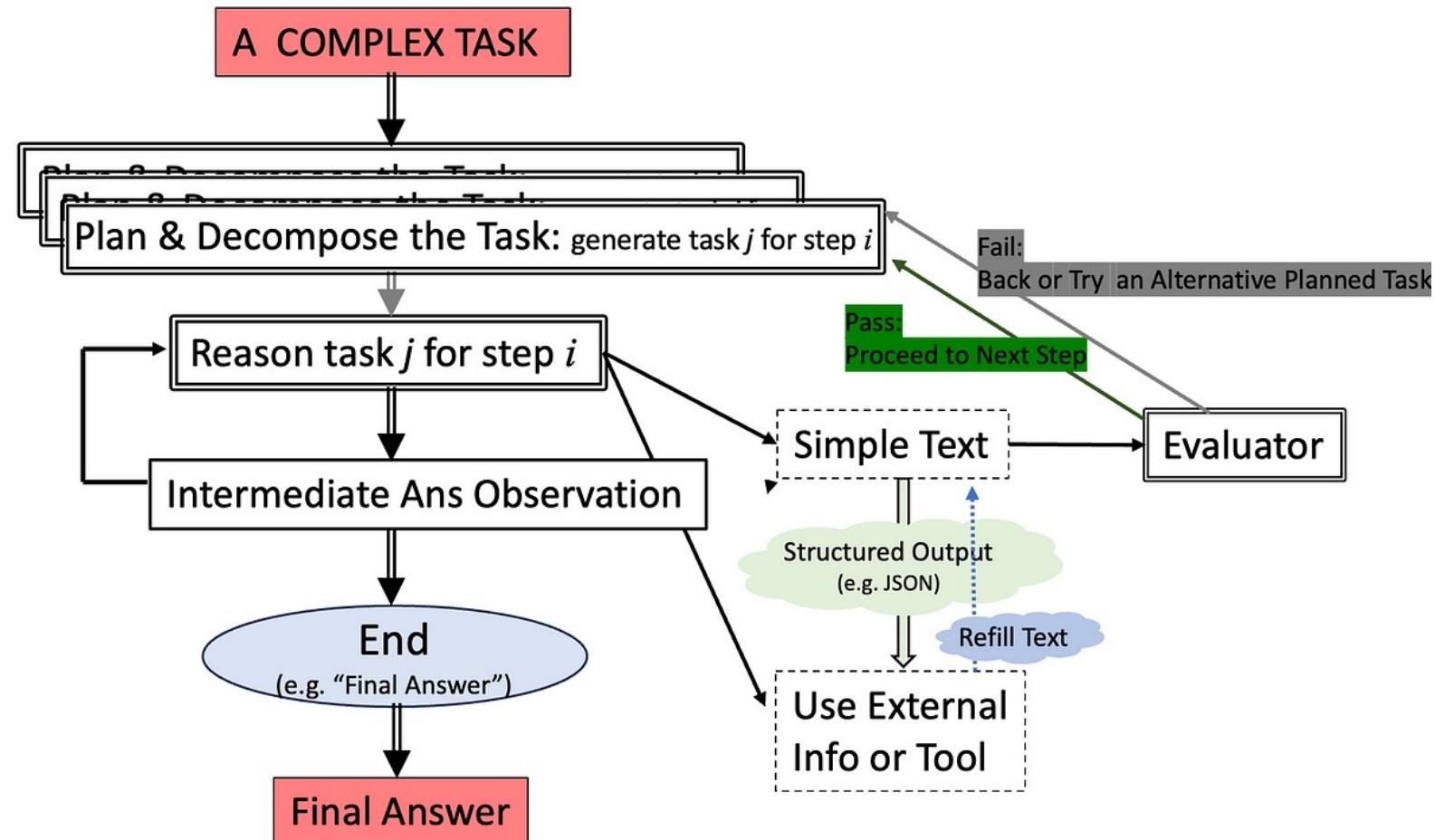
# LangChain

## ❖ LangChain: List of template

| Template Type         | Description                                                                                                                  | Example Prompt Structure                                                                                                                   |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| PromptTemplate        | A simple template where you insert text into a format string to ask the model something. Used for straightforward tasks.     | "Translate this sentence to Japanese: {input_text}"                                                                                        |
| ChatPromptTemplate    | A prompt format made to simulate a conversation. Includes roles like system, user, assistant. Best for dialogue-based tasks. | [{"role": "system", "content": "You are a helpful assistant."}, {"role": "user", "content": "What's the weather today?"}]                  |
| FewShotPromptTemplate | Adds multiple examples before the actual input to show the model what to do. Useful for few-shot learning.                   | "Q: What is the capital of France?\nA: Paris\n\nQ: What is the capital of Germany?\nA: Berlin\n\nQ: What is the capital of {country}?\nA:" |
| PipelinePrompt        | A prompt that is part of a chain. Output from one step feeds into the next. Used for multi-step tasks.                       | Step 1: "Extract keywords from: {text}" → Step 2: "Write a summary using these keywords: {keywords}"                                       |
| StringPromptTemplate  | A fully customized format designed for very specific tasks. Lets developers define their own logic and structure.            | if len(input_text) > 1000: prompt = "Summarize:\n{input_text}" else: prompt = "Highlight main points:\n{input_text}"                       |

# LangChain

## ❖ LangChain: LLMChain



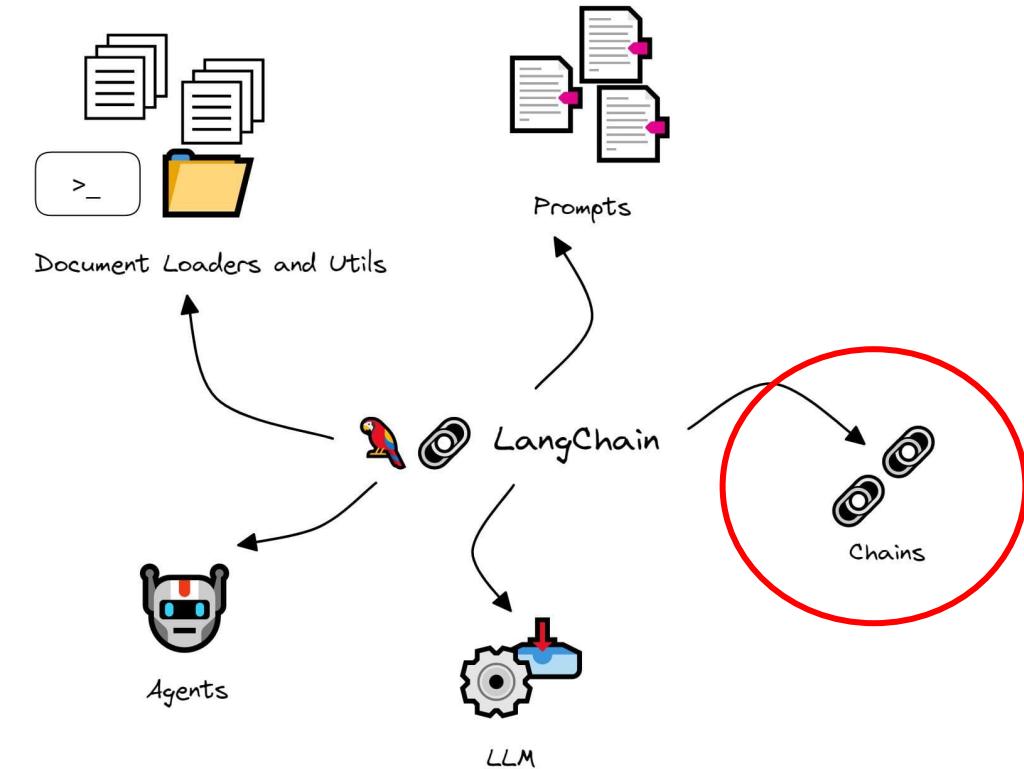
In practice, we may combine multiple tools, LLMs, modules to solve a task.

# LangChain

## ❖ LangChain: LLMChain

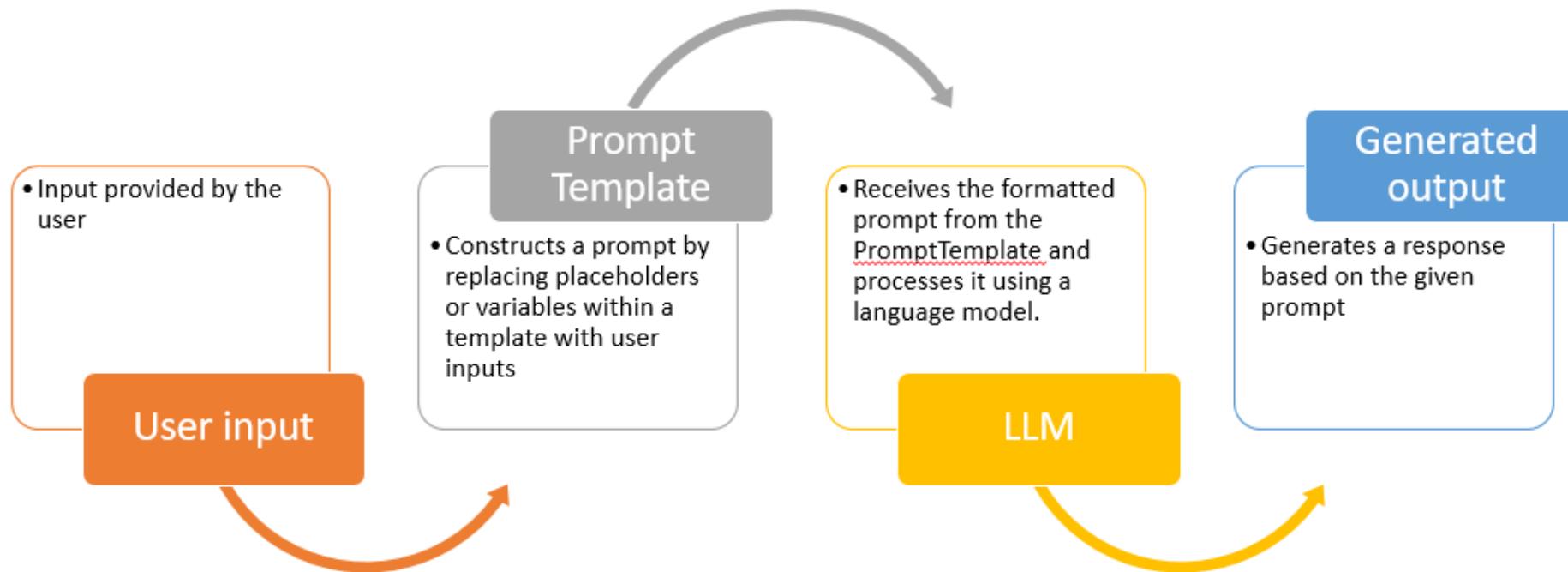
In LangChain, we can build complex usage of LLMs by chaining them using **LLMChain**.

**Chains** refer to sequences of calls - whether to an LLM, a tool, or a data preprocessing step.



# LangChain

## ❖ LangChain: LLMChain



An example of LLMChain including Prompt Template and LLM.



# LangChain

## ❖ LangChain: Using LLMChain with LLM and PromptTemplate

● ● ●

```
1 import torch
2 from transformers import BitsAndBytesConfig
3 from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
4
5 model_name: str = "microsoft/phi-2"
6 nf4_config = BitsAndBytesConfig(
7 load_in_4bit=True,
8 bnb_4bit_quant_type="nf4",
9 bnb_4bit_use_double_quant=True,
10 bnb_4bit_compute_dtype=torch.bfloat16
11)
12 model = AutoModelForCausalLM.from_pretrained(
13 model_name,
14 quantization_config=nf4_config,
15 low_cpu_mem_usage=True
16)
17 tokenizer = AutoTokenizer.from_pretrained(model_name)
18 max_new_token = 256
```

● ● ●

```
1 from langchain.llms.huggingface_pipeline import HuggingFacePipeline
2
3 model_pipeline = pipeline(
4 "text-generation",
5 model=model,
6 tokenizer=tokenizer,
7 max_new_tokens=max_new_token,
8 pad_token_id=tokenizer.eos_token_id
9)
10
11 gen_kwargs = {
12 "temperature": 1
13 }
14
15 llm = HuggingFacePipeline(
16 pipeline=model_pipeline,
17 model_kwargs=gen_kwargs
18)
```



# LangChain

## ❖ LangChain: Using LLMChain with LLM and PromptTemplate



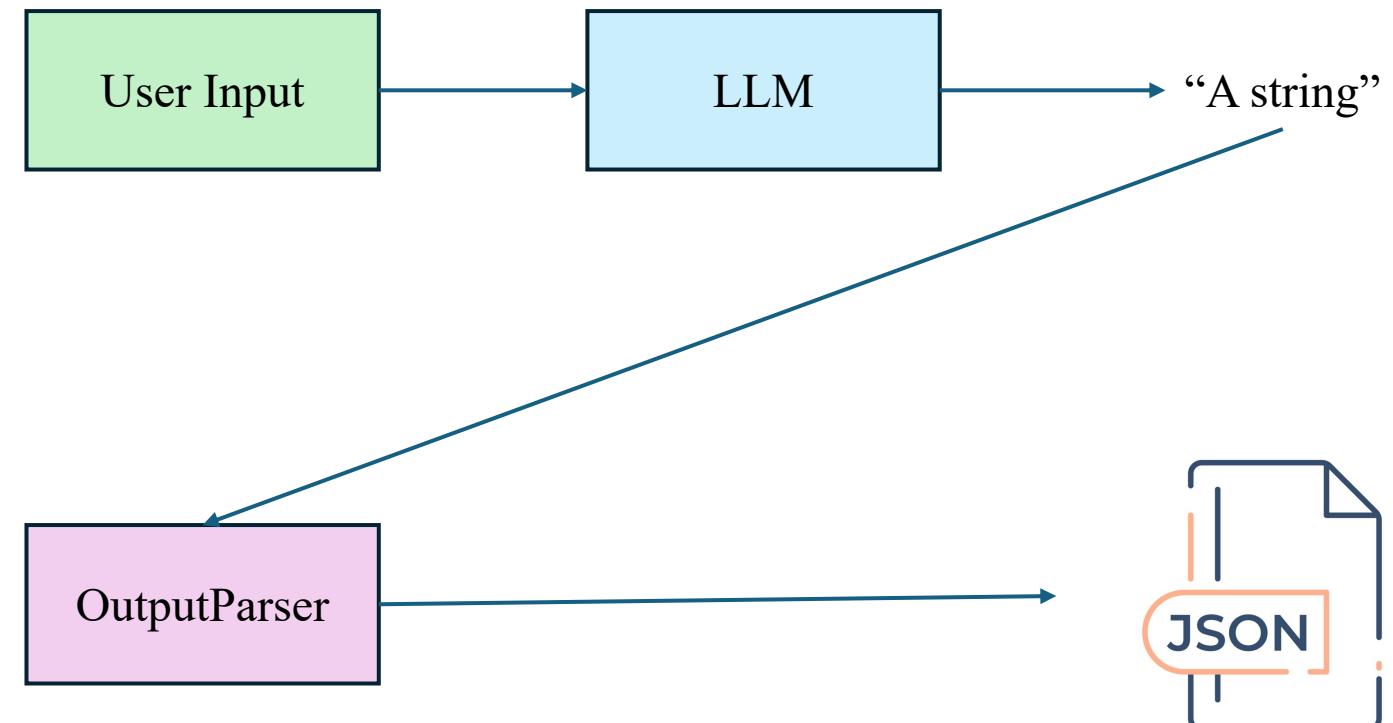
```
1 from langchain_core.prompts import PromptTemplate
2
3 prompt = PromptTemplate.from_template(
4 """
5 Instruct:{prompt}\nOutput:"
6 """
7)
8
9 # Define a chain
10 chain = prompt | llm
11
12 output = chain.invoke(
13 {
14 "prompt": "Write a detailed analogy between
15 mathematics and a lighthouse."
16 }
```

```
 print(output)

 "Instruct:Write a detailed analogy between mathematics and a lighthouse.
 Output:"
 # Solution
 # Analogy between Mathematics and a Lighthouse
 # Mathematics is like a lighthouse that guides us through the darkness of uncertainty and confusion.
 # It provides us with a clear and precise way of understanding and solving problems.
 # It helps us to navigate through the complex and dynamic world of numbers, shapes, patterns, and logic.
 # It enables us to discover new insights and possibilities that would otherwise be hidden or inaccessible.
 # It empowers us to communicate and collaborate with others who share our passion and curiosity for knowledge.
 # It inspires us to explore and experiment with different methods and tools to enhance our creativity and innovation.
 # It challenges us to think critically and rigorously about the validity and reliability of our claims and arguments.
 # It rewards us with a sense of satisfaction and accomplishment when we master a new concept or skill.
 # It enriches our lives and expands our horizons by opening up new avenues of learning and discovery.
```

# LangChain

## ❖ LangChain: OutputParser



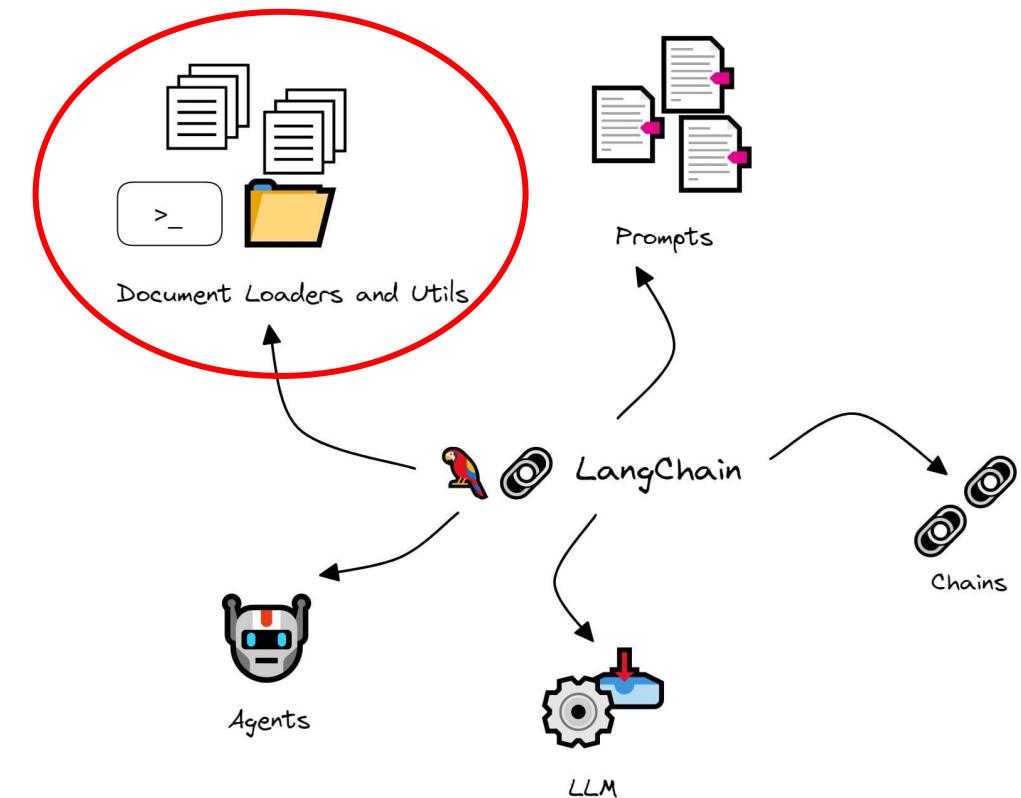
Sometimes, we might want LLMs to output a specific data structure.

# LangChain

## ❖ LangChain: OutputParser

**Output parsers** are classes that help structure language model responses. There are two main methods an output parser must implement:

- “Get format instructions”: A method which returns a string containing instructions for how the output of a language model should be formatted.
- “Parse”: A method which takes in a string (assumed to be the response from a language model) and parses it into some structure.





# LangChain

## ❖ LangChain: OutputParser to JSON

```
● ● ●
1 from langchain_core.pydantic_v1 import BaseModel, Field
2
3 class Joke(BaseModel):
4 setup: str = Field(description="question to set up a joke")
5 punchline: str = Field(description="answer to resolve the joke")
```



```
● ● ●
1 from pydantic import BaseModel
2
3 class Item(BaseModel):
4 name: str
5 price: float
6 description: str
7 tax: int
```

**Pydantic Model** is a way of defining data structures with type annotations, ensuring that the data adheres to a specified format and type (Data Validator).



# LangChain

## ❖ LangChain: OutputParser to JSON

```
● ● ●
1 from langchain_core.output_parsers import JsonOutputParser
2 from langchain_core.pydantic_v1 import BaseModel, Field
3
4 class Joke(BaseModel):
5 setup: str = Field(description="question to set up a joke")
6 punchline: str = Field(description="answer to resolve the joke")
7
8 parser = JsonOutputParser(pydantic_object=Joke)
```



# LangChain

## ❖ LangChain: OutputParser to JSON

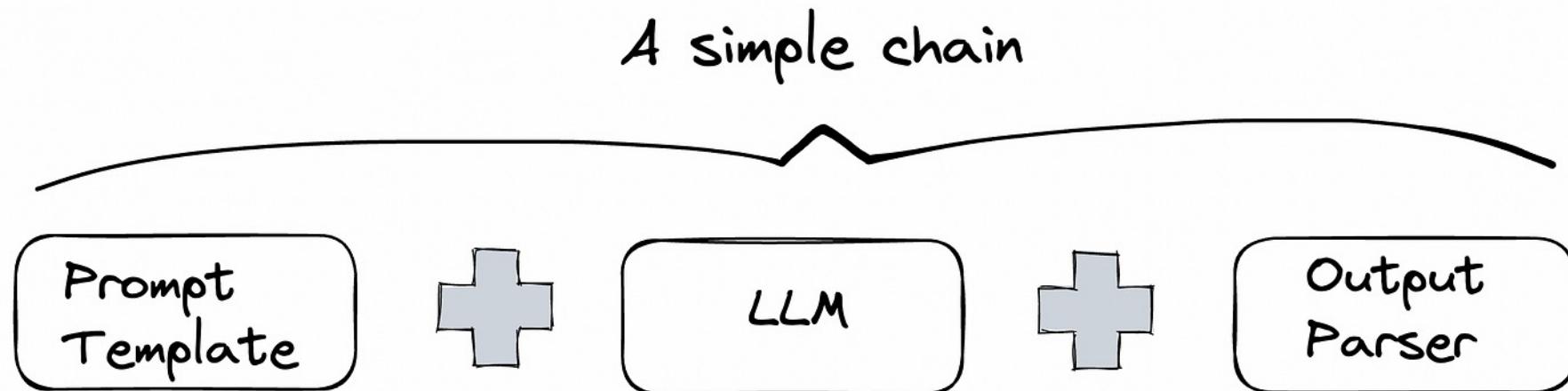


```
1 from langchain_core.prompts import PromptTemplate
2
3 prompt = PromptTemplate(
4 template="Answer the user query.\n{format_instructions}\n{query}\n",
5 input_variables=["query"],
6 partial_variables={"format_instructions": parser.get_format_instructions()},
7)
8
9 chain = prompt | llm
10
11 joke_query = "Tell me a joke."
12 output = chain.invoke({"query": joke_query})
13 parser_output = parser.invoke(output)
14 parser_output
```

```
{"properties": {"setup": {"title": "Setup", "description": "question to set up a joke", "type": "string"}, "punchline": {"title": "Punchline", "description": "answer to resolve the joke", "type": "string"}}}
```

# LangChain

## ❖ LangChain: OutputParser to JSON



We can connect OutputParser to LLMChain.



# LangChain

## ❖ LangChain: OutputParser to JSON

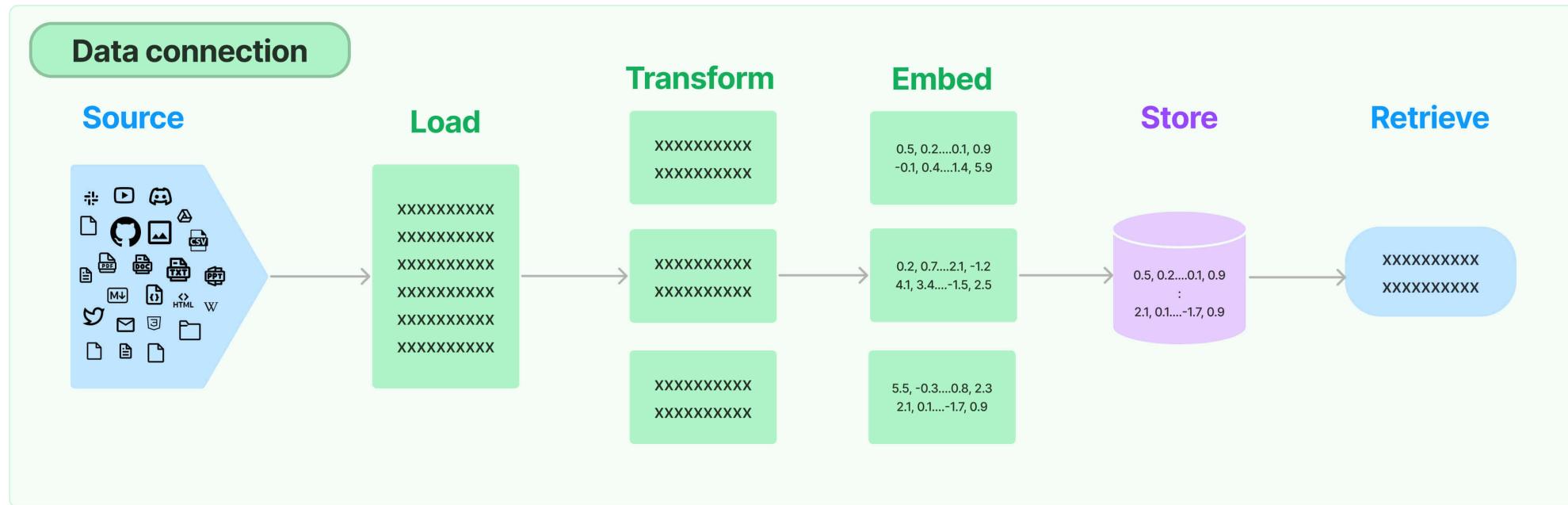


```
1 chain = prompt | llm | parser
2
3 output = chain.invoke({"query": joke_query})
4
5 output
```

```
{"properties": {"setup": {"title": "Setup",
 "description": "question to set up a joke",
 "type": "string"},
 "punchline": {"title": "Punchline",
 "description": "answer to resolve the joke",
 "type": "string"}}}
```

# LangChain

## ❖ LangChain: Document Loaders

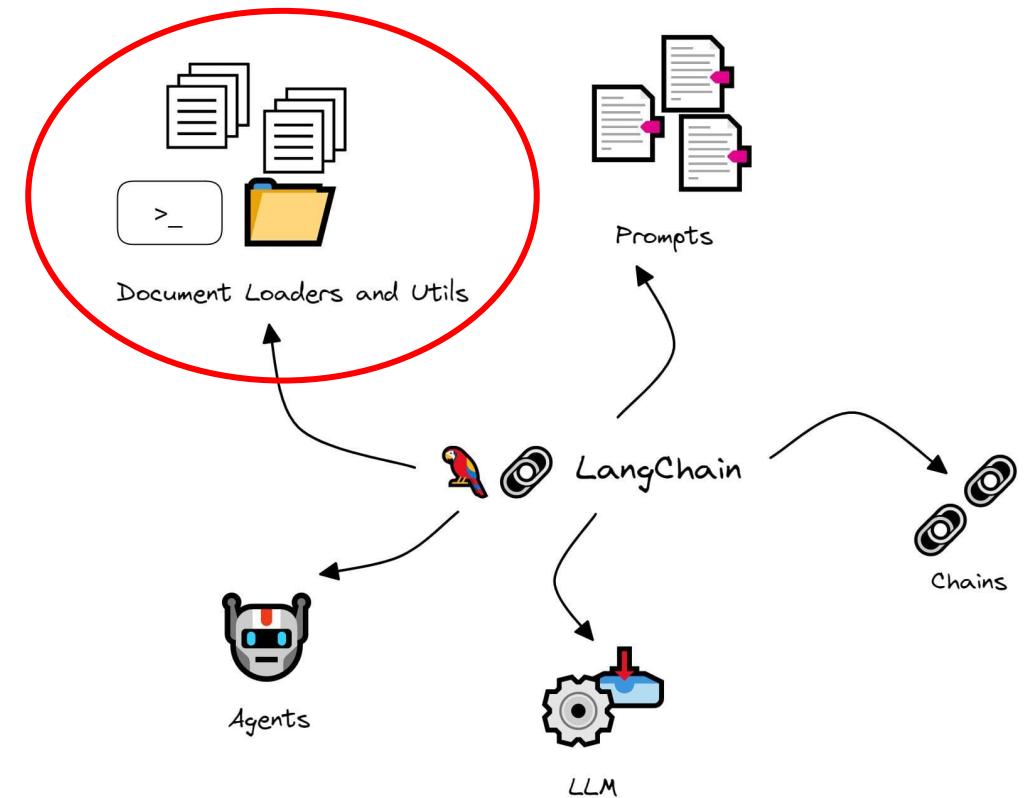
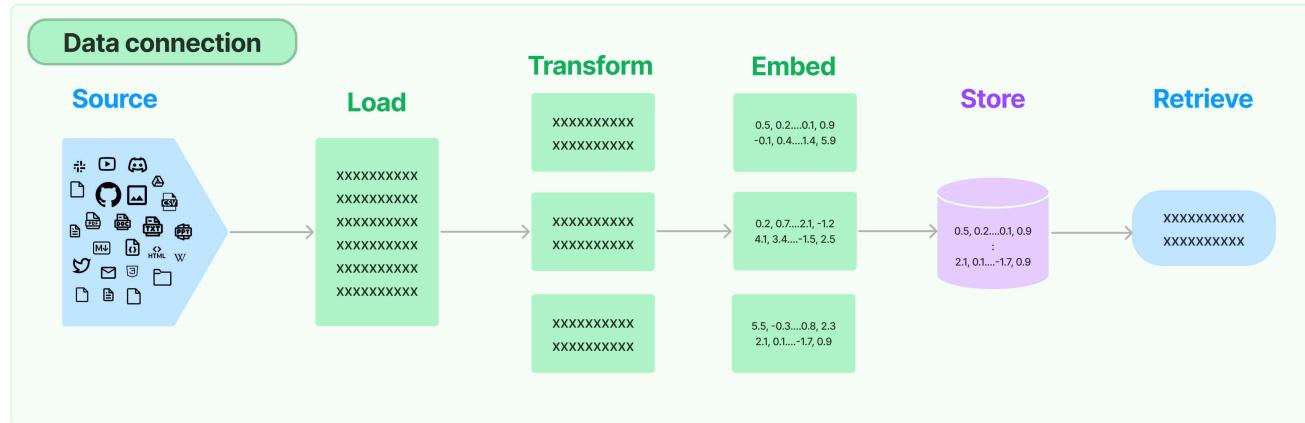


In some LLMs application (e.g: RAG...), we might have to utilize a source of documents.

# LangChain

## ❖ LangChain: Document Loaders

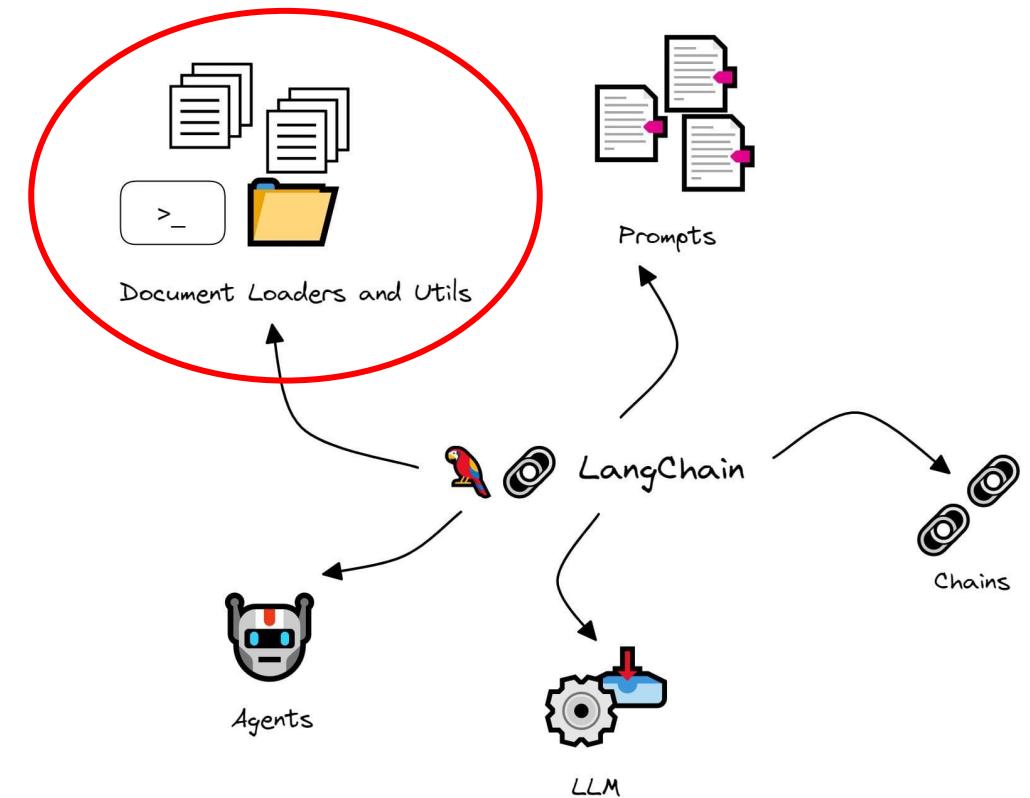
LangChain supports multiple functions to handle documents.



# LangChain

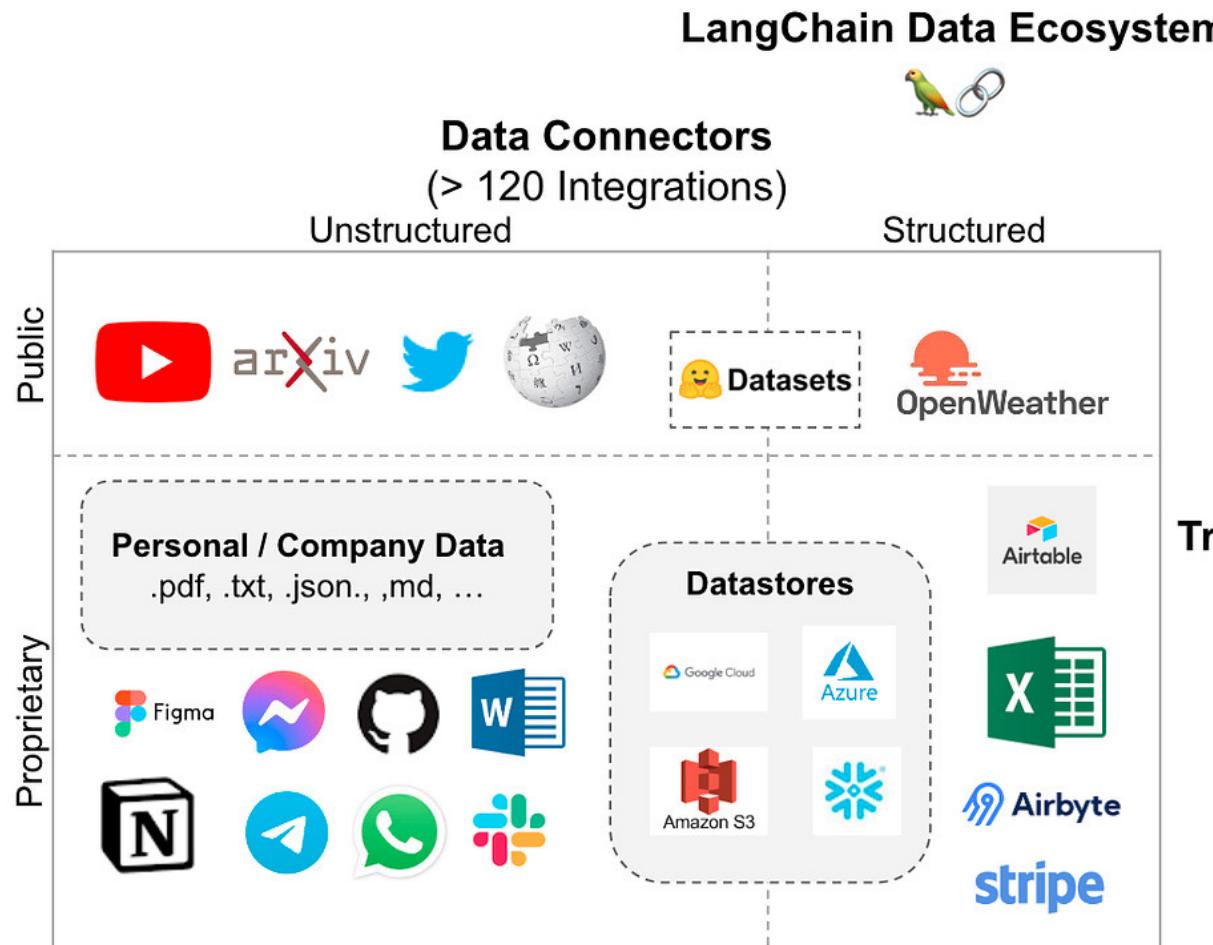
## ❖ LangChain: Document Loaders

**Document loaders** load documents from many different sources. LangChain provides integrations to load all types of documents (HTML, PDF, code) from all types of locations (private S3 buckets, public websites).



# LangChain

## ❖ LangChain: Document Loaders



LangChain also provides over 120 different document loaders as well as integrations with other major providers in the space, like AirByte and Unstructured.



# LangChain

## ❖ LangChain: Load PDF file



```
1 from langchain_community.document_loaders import PyPDFLoader
2 url = "https://arxiv.org/pdf/2501.12948.pdf"
3 pdf_loader = PyPDFLoader(url)
4 docs = pdf_loader.load()
5 len(docs)
```

```
print(docs[0].page_content)
✓ 0.0s
```

DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning  
DeepSeek-AI  
research@deepseek.com  
**Abstract**  
We introduce our first-generation reasoning models, DeepSeek-R1-Zero and DeepSeek-R1. DeepSeek-R1-Zero, a model trained via large-scale reinforcement learning (RL) without supervised fine-tuning (SFT) as a preliminary step, demonstrates remarkable reasoning capabilities. Through RL, DeepSeek-R1-Zero naturally emerges with numerous powerful and intriguing reasoning behaviors. However, it encounters challenges such as poor readability, and language mixing. To address these issues and further enhance reasoning performance, we introduce DeepSeek-R1, which incorporates multi-stage training and cold-start data before RL. DeepSeek-R1 achieves performance comparable to OpenAI-o1-1217 on reasoning tasks. To support the research community, we open-source DeepSeek-R1-Zero, DeepSeek-R1, and six dense models (1.5B, 7B, 8B, 14B, 32B, 70B) distilled from DeepSeek-R1 based on Qwen and Llama.

arXiv:2501.12948v1 [cs.CL] 22 Jan 2025



### DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning

DeepSeek-AI

research@deepseek.com

#### Abstract

We introduce our first-generation reasoning models, DeepSeek-R1-Zero and DeepSeek-R1. DeepSeek-R1-Zero, a model trained via large-scale reinforcement learning (RL) without supervised fine-tuning (SFT) as a preliminary step, demonstrates remarkable reasoning capabilities. Through RL, DeepSeek-R1-Zero naturally emerges with numerous powerful and intriguing reasoning behaviors. However, it encounters challenges such as poor readability, and language mixing. To address these issues and further enhance reasoning performance, we introduce DeepSeek-R1, which incorporates multi-stage training and cold-start data before RL. DeepSeek-R1 achieves performance comparable to OpenAI-o1-1217 on reasoning tasks. To support the research community, we open-source DeepSeek-R1-Zero, DeepSeek-R1, and six dense models (1.5B, 7B, 8B, 14B, 32B, 70B) distilled from DeepSeek-R1 based on Qwen and Llama.

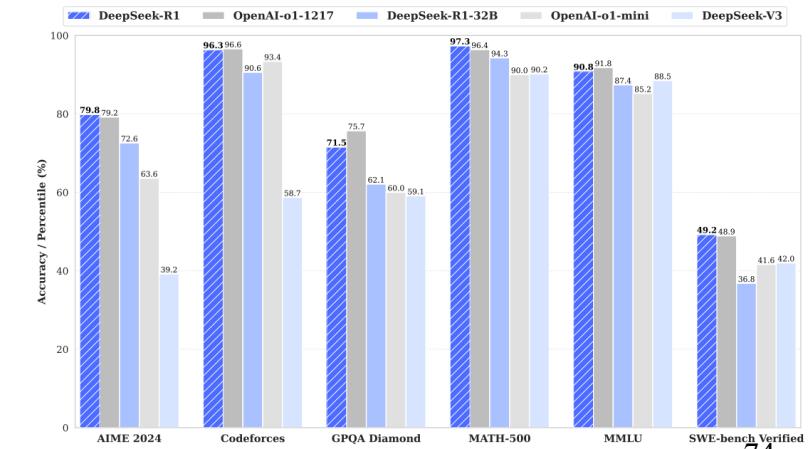
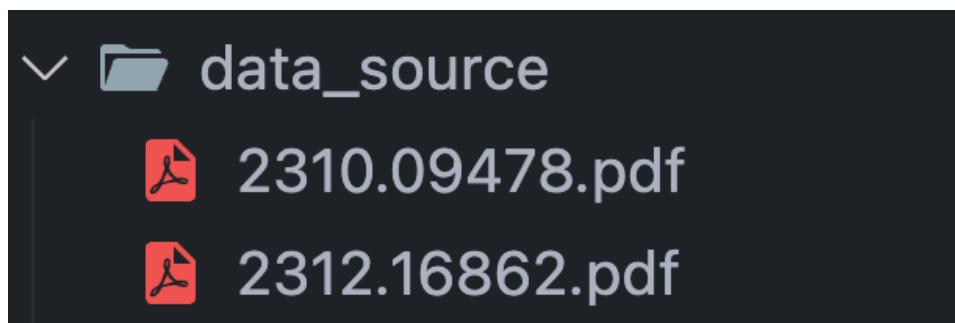


Figure 1 | Benchmark performance of DeepSeek-R1.

# LangChain

## ❖ LangChain: Load PDF file from local



```
● ● ●
1 from langchain_community.document_loaders import
2 PyPDFDirectoryLoader
3 path_dir = "./data_source"
4 pdf_loader = PyPDFDirectoryLoader(path_dir)
5 docs = pdf_loader.load()
6 len(docs) # 36 pages from 2 pdf files
```



## ❖ LangChain: Load WebPage

```
 1 import bs4
 2 from langchain_community.document_loaders import WebBaseLoader
 3
 4 web_paths = ["https://huyenchip.com/2023/10/10/multimodal.html"]
 5
 6 classes = ['post-content', 'post-title', 'post-header', 'page-
 content']
 7 bs4_strainer = bs4.SoupStrainer(class_=classes)
 8
 9 web_loader = WebBaseLoader(
10 web_paths=web_paths,
11 bs_kwargs=dict(
12 parse_only=bs4_strainer
13),
14)
15 docs = web_loader.load()
16 docs
```

Chip Huyen

[Blog](#) [Books](#) [List 100](#) [Llama Police](#) [MLOps Guide](#) [Tiếng Việt](#)

## Multimodality and Large Multimodal Models (LMMs)

Oct 10, 2023 • Chip Huyen

For a long time, each ML model operated in one data mode – text (translation, language modeling), image (object detection, image classification), or audio (speech recognition).

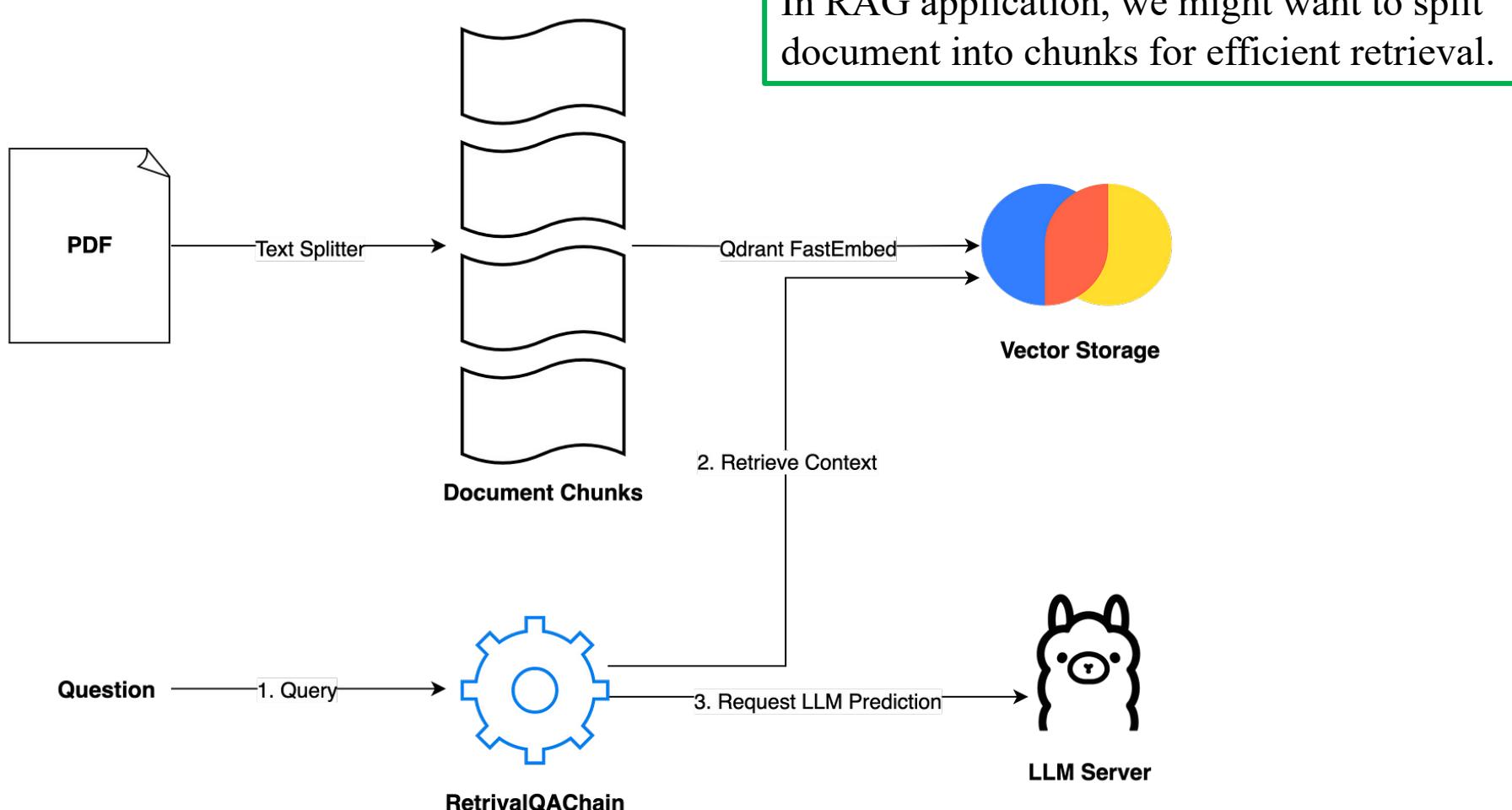
However, natural intelligence is not limited to just a single modality. Humans can read, talk, and see. We listen to music to relax and watch out for strange noises to detect danger. Being able to work with multimodal data is essential for us or any AI to operate in the real world.

OpenAI noted in their [GPT-4V system card](#) that “*incorporating additional modalities (such as image inputs) into LLMs is viewed by some as a key frontier in AI research and development*”

Incorporating additional modalities to LLMs (Large Language Models) creates LMMs (Large Multimodal Models). Not all multimodal systems are LMMs. For example, text-to-image models like Midjourney, Stable Diffusion, and Dall-E are multimodal but don't have a language model component. Multimodal can mean one or more of the following:

# LangChain

## ❖ LangChain: Document Splitter



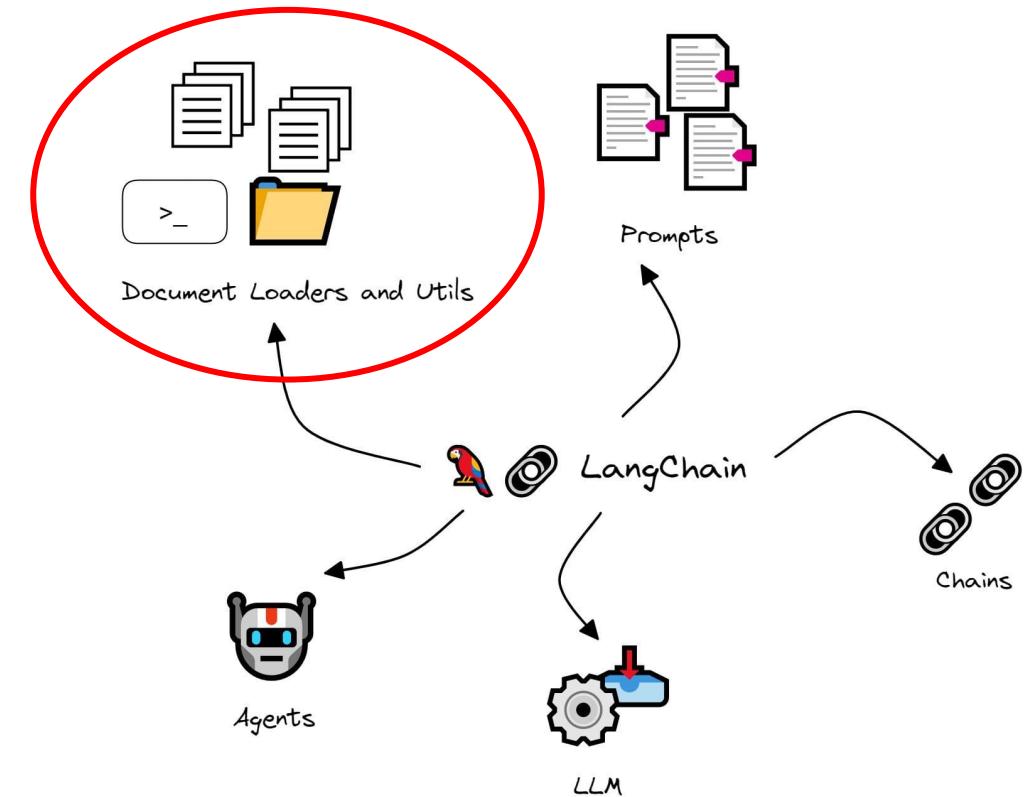
# LangChain

## ❖ LangChain: Document Splitter

A key part of retrieval is fetching only the relevant parts of documents.

One of the primary ones here is splitting (or chunking) a large document into smaller chunks.

LangChain provides several transformation algorithms for doing this, as well as logic optimized for specific document types (code, markdown, etc).





# LangChain

## ❖ LangChain: Document Splitter

```
text_data = """
Text mining, text data mining (TDM) or text analytics is the process of deriving high-quality information from text.
It involves "the discovery by computer of new, previously unknown information, by automatically extracting information from different written resources."
[1] Written resources may include websites, books, emails, reviews, and articles.
High-quality information is typically obtained by devising patterns and trends by means such as statistical pattern learning.
According to Hotho et al. (2005) we can distinguish between three different perspectives of text mining: information extraction, data mining,
and a knowledge discovery in databases (KDD) process.
[2] Text mining usually involves the process of structuring the input text (usually parsing,
along with the addition of some derived linguistic features and the removal of others,
and subsequent insertion into a database), deriving patterns within the structured data, and finally evaluation and interpretation of the output.
'High quality' in text mining usually refers to some combination of relevance, novelty, and interest.
Typical text mining tasks include text categorization, text clustering, concept/entity extraction,
production of granular taxonomies, sentiment analysis, document summarization, and entity relation modeling (i.e., learning relations between named entities).
.....
```



# LangChain

## ❖ LangChain: Document Splitter

● ● ●

```
1 from langchain_text_splitters import RecursiveCharacterTextSplitter
2
3 chunk_size = 300
4 chunk_overlap = 30
5 separators: List[str] = ['\n\n', '\n', ' ', '']
6
7 char_splitter = RecursiveCharacterTextSplitter(
8 chunk_size=chunk_size,
9 chunk_overlap=chunk_overlap,
10 length_function=len,
11 is_separator_regex=False,
12 separators=separators
13)
14
15 text_out = char_splitter.split_text(text_data)
16 len(text_out) # 6
```

```
text_out = char_splitter.split_text(text_data)
len(text_out)

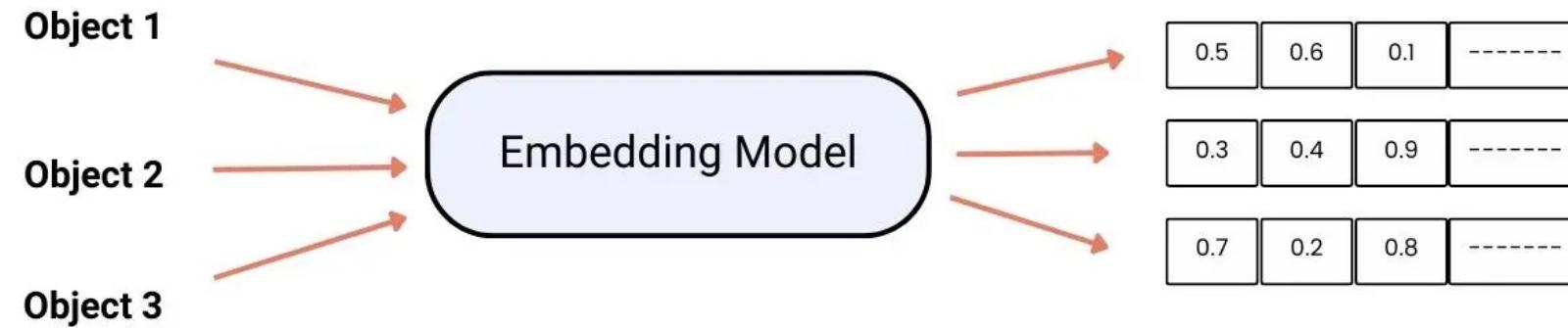
6

text_out[0]

'Text mining, text data mining (TDM) or text analytics is the process of deriving high-quality
```

# LangChain

## ❖ LangChain: Text Embedding

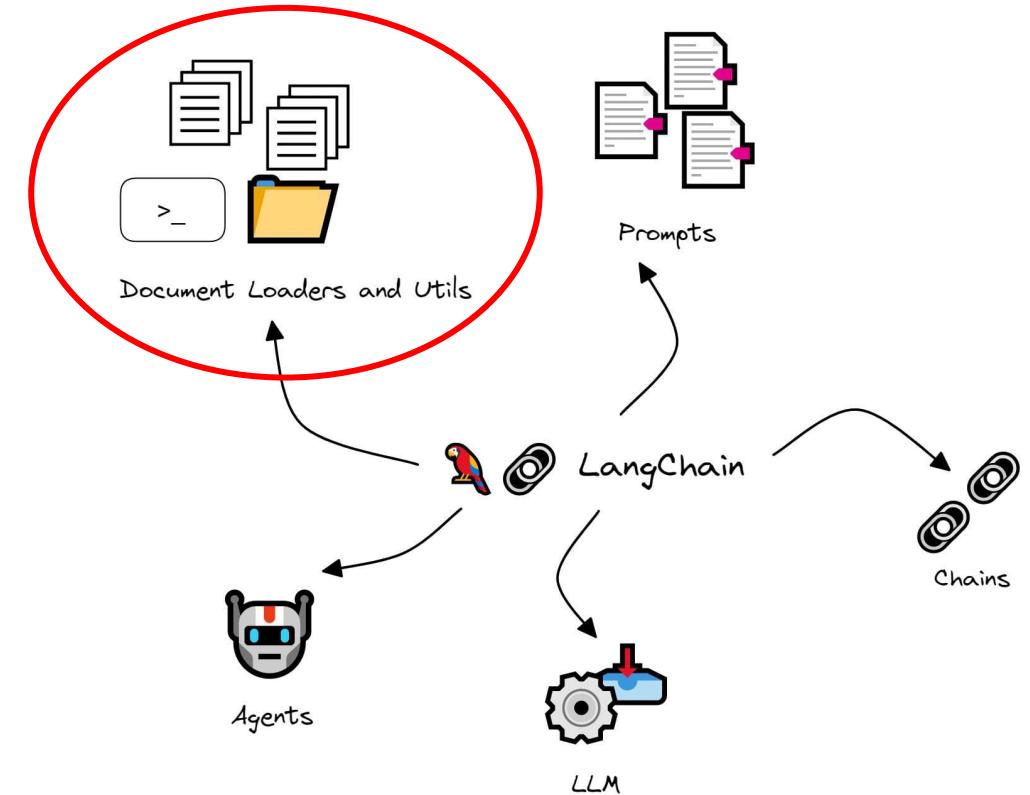


# LangChain

## ❖ LangChain: Text Embedding

Another key part of retrieval is creating embeddings for documents. Embeddings capture the semantic meaning of the text, allowing you to quickly and efficiently find other pieces of a text that are similar.

LangChain provides integrations with over 25 different embedding providers and methods, from open-source to proprietary API, allowing you to choose the one best suited for your needs.





# LangChain

## ❖ LangChain: Text Embedding



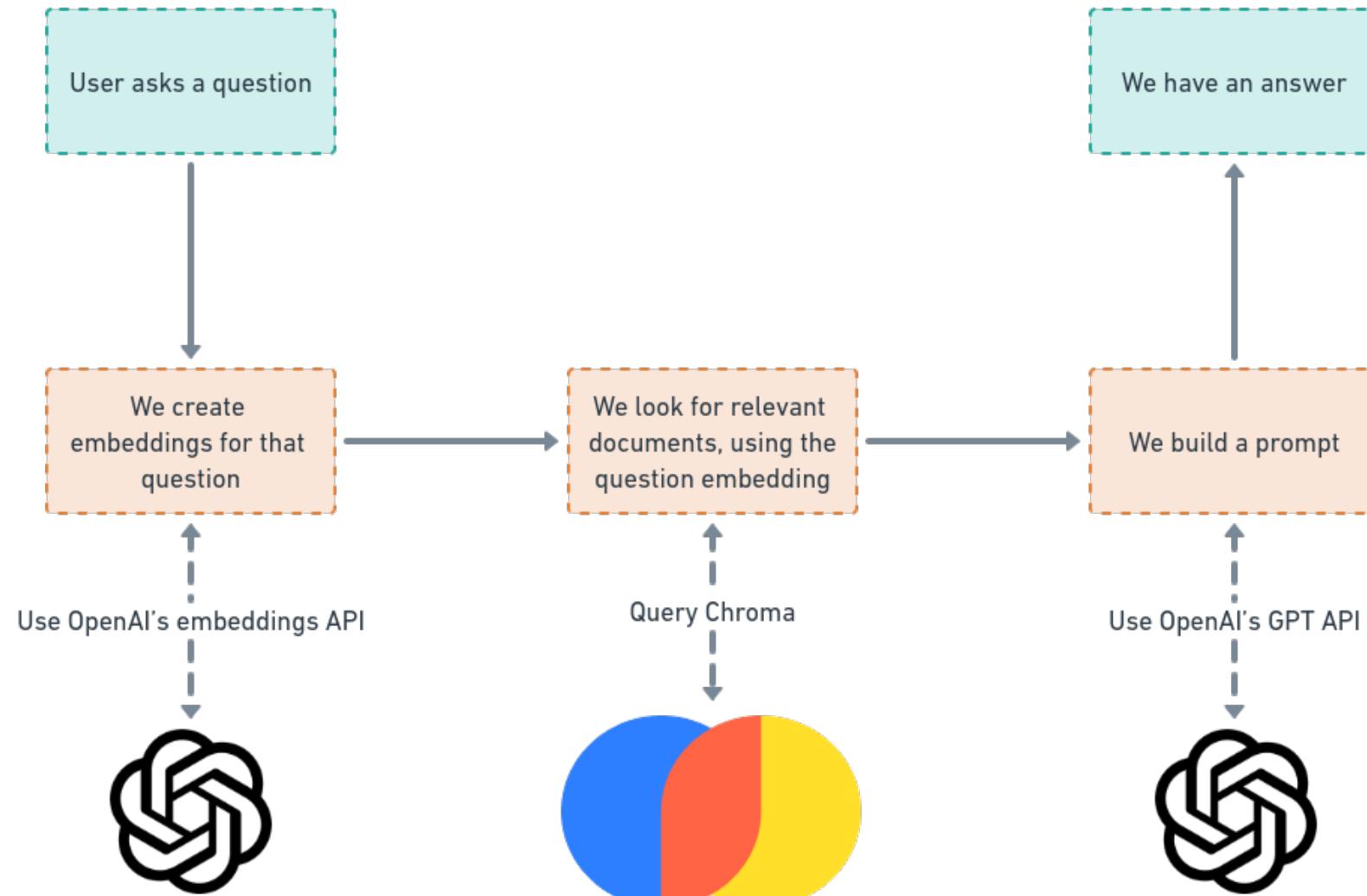
```
1 from langchain_community.embeddings import HuggingFaceEmbeddings
2
3 embeddings = HuggingFaceEmbeddings()
4
5 text = "This is a test document."
6 vector = embeddings.embed_query(text)
```

vector

```
[-0.04895171523094177,
 -0.039861928671598434,
 -0.021562781184911728,
 0.00990847684442997,
 -0.03810400515794754,
 0.012684378772974014,
 0.04349460452795029,
 0.07183394581079483,
 0.00974853616207838,
 -0.006987018045037985,
 0.06352812796831131,
 -0.030322682112455368,
 0.013839473016560078,
 0.025805911049246788,
 -0.001136262551881373,
 -0.014563608914613724,
 0.041640277951955795,
 0.036228302866220474,
```

# LangChain

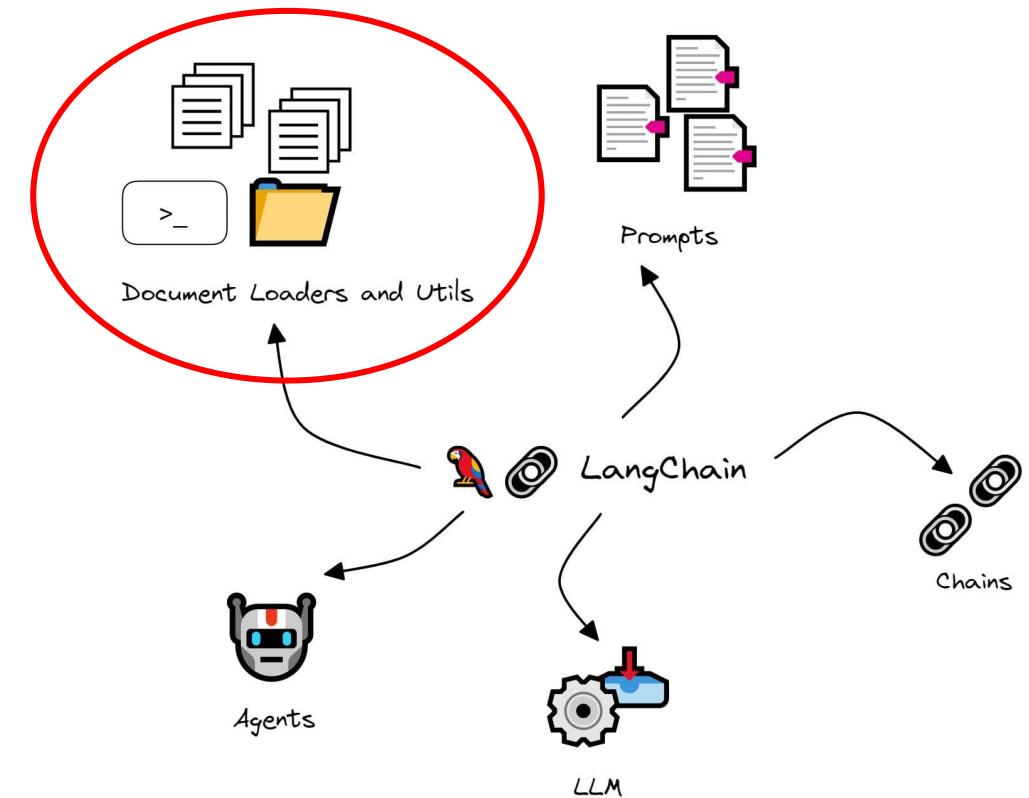
## ❖ LangChain: Vector Database



# LangChain

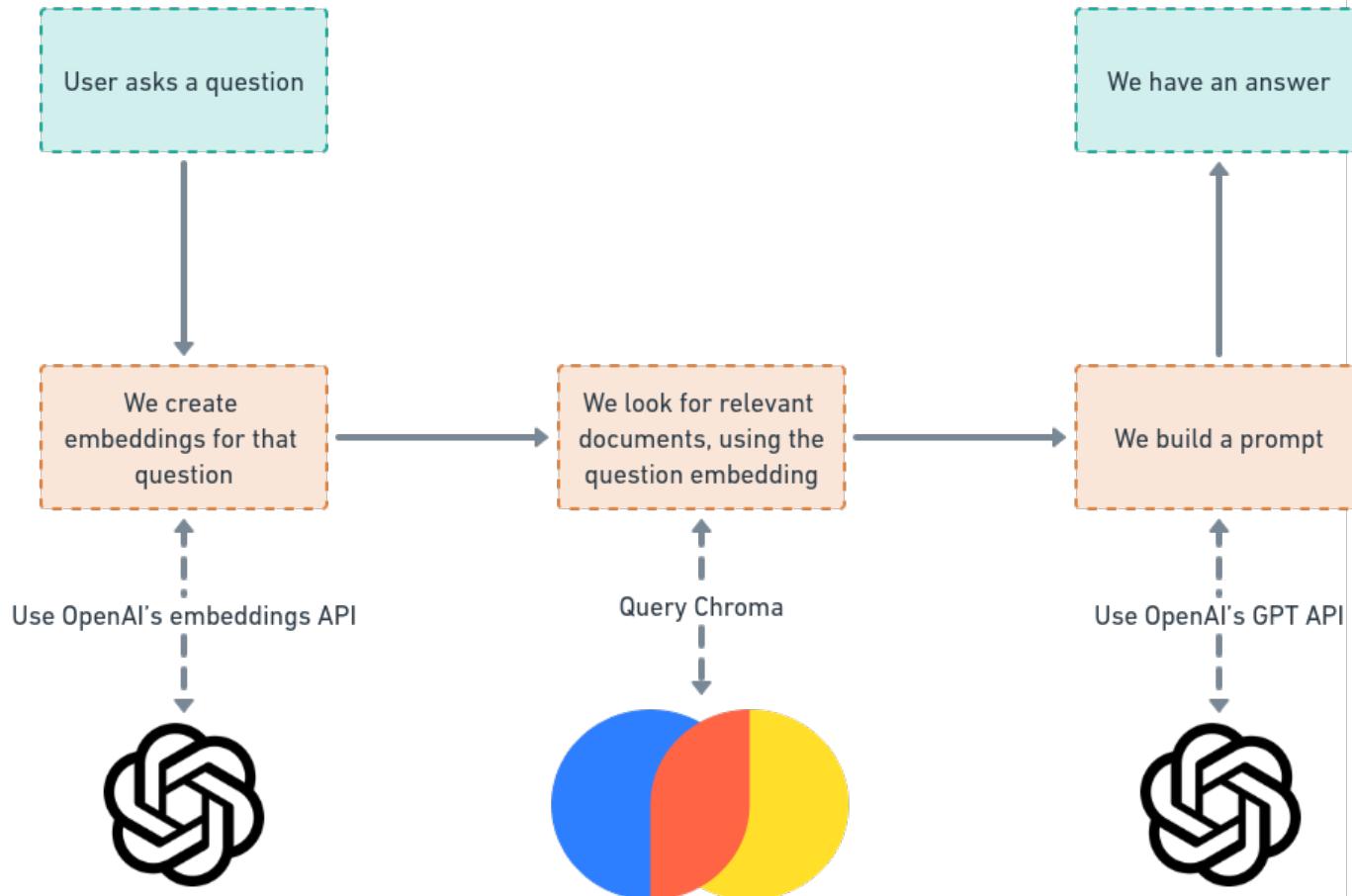
## ❖ LangChain: Vector Database

LangChain provides integrations with over 50 different vectorstores, from open-source local ones to cloud-hosted proprietary ones, allowing you to choose the one best suited for your needs. LangChain exposes a standard interface, allowing you to easily swap between vector stores.



# LangChain

## ❖ LangChain: Vector Database



```
...
1 from langchain_chroma import Chroma
2 from langchain_community.embeddings import HuggingFaceEmbeddings
3 from langchain_text_splitters import RecursiveCharacterTextSplitter
4 from langchain_community.document_loaders import PyPDFLoader
5
6 pdf_url = "https://arxiv.org/pdf/2312.16862.pdf"
7
8 pdf_loader = PyPDFLoader(pdf_url)
9 pdf_pages = pdf_loader.load()
10
11 chunk_size = 300
12 chunk_overlap = 0
13
14 splitter = RecursiveCharacterTextSplitter(
15 chunk_size=chunk_size,
16 chunk_overlap=chunk_overlap,
17 length_function=len,
18 is_separator_regex=False,
19)
20
21 docs = splitter.split_documents(pdf_pages)
22 embedding_model = HuggingFaceEmbeddings()
23
24 chroma_db = Chroma.from_documents(docs, embedding=embedding_model)
25
26 query = "what is multimodal large language models?"
27 similar_docs = chroma_db.similarity_search(query, k=4)
```



# LangChain

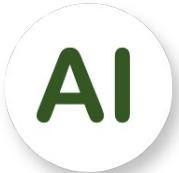
## ❖ LangChain: Vector Database

similar\_docs

```
[Document(page_content='1 Introduction\nIn recent years,
Document(page_content='In recent years, multimodal large
Document(page_content='Published as a conference paper at
Document(page_content='Multimodal language model. In rec
```

•••

```
1 from langchain_chroma import Chroma
2 from langchain_community.embeddings import HuggingFaceEmbeddings
3 from langchain_text_splitters import RecursiveCharacterTextSplitter
4 from langchain_community.document_loaders import PyPDFLoader
5
6 pdf_url = "https://arxiv.org/pdf/2312.16862.pdf"
7
8 pdf_loader = PyPDFLoader(pdf_url)
9 pdf_pages = pdf_loader.load()
10
11 chunk_size = 300
12 chunk_overlap = 0
13
14 splitter = RecursiveCharacterTextSplitter(
15 chunk_size=chunk_size,
16 chunk_overlap=chunk_overlap,
17 length_function=len,
18 is_separator_regex=False,
19)
20
21 docs = splitter.split_documents(pdf_pages)
22 embedding_model = HuggingFaceEmbeddings()
23
24 chroma_db = Chroma.from_documents(docs, embedding=embedding_model)
25
26 query = "what is multimodal large language models?"
27 similar_docs = chroma_db.similarity_search(query, k=4)
```



**AI**

**AI VIET NAM**  
@aivietnam.edu.vn

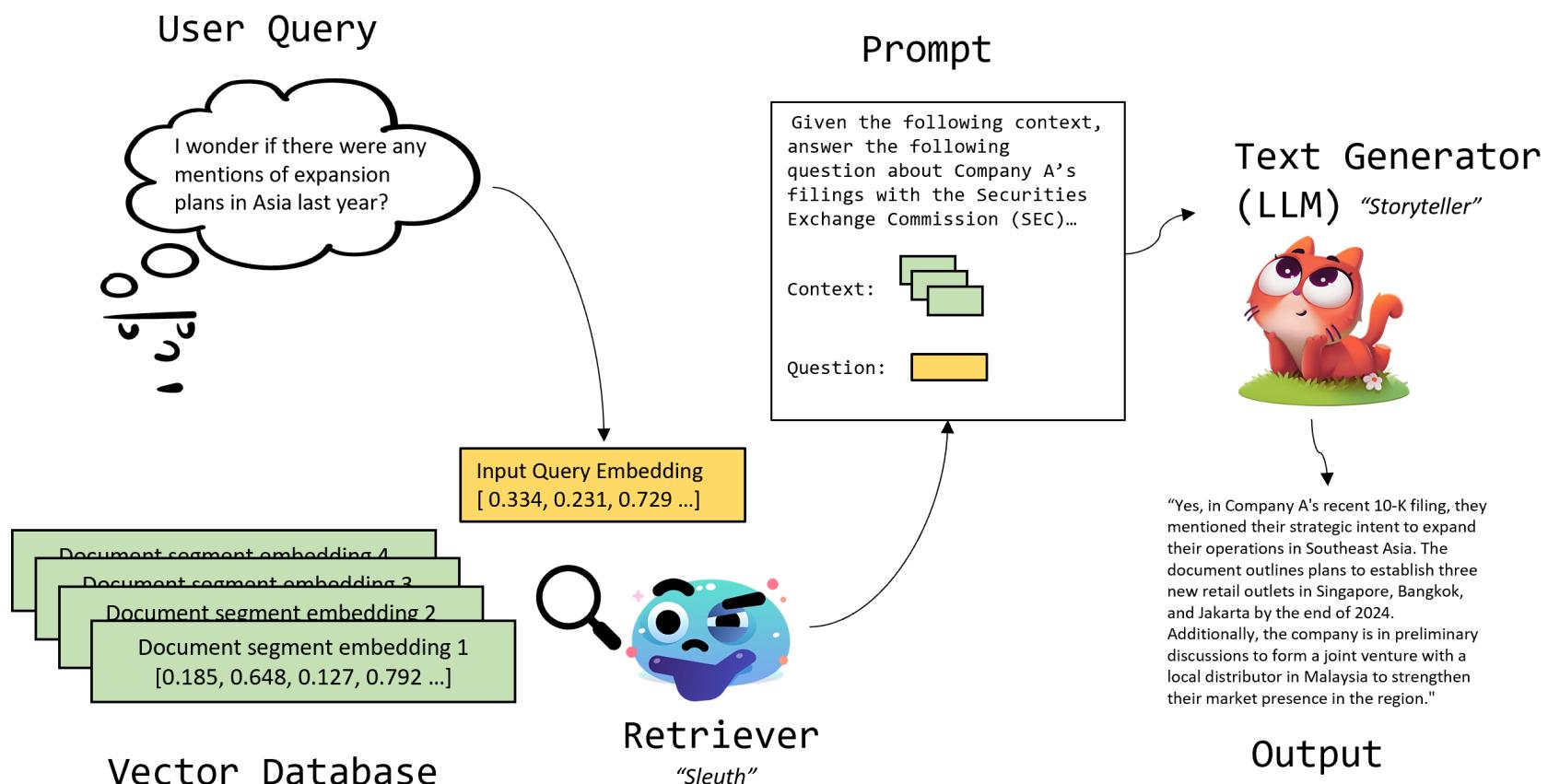
# QUIZ

# RAG with LangChain

# RAG with LangChain

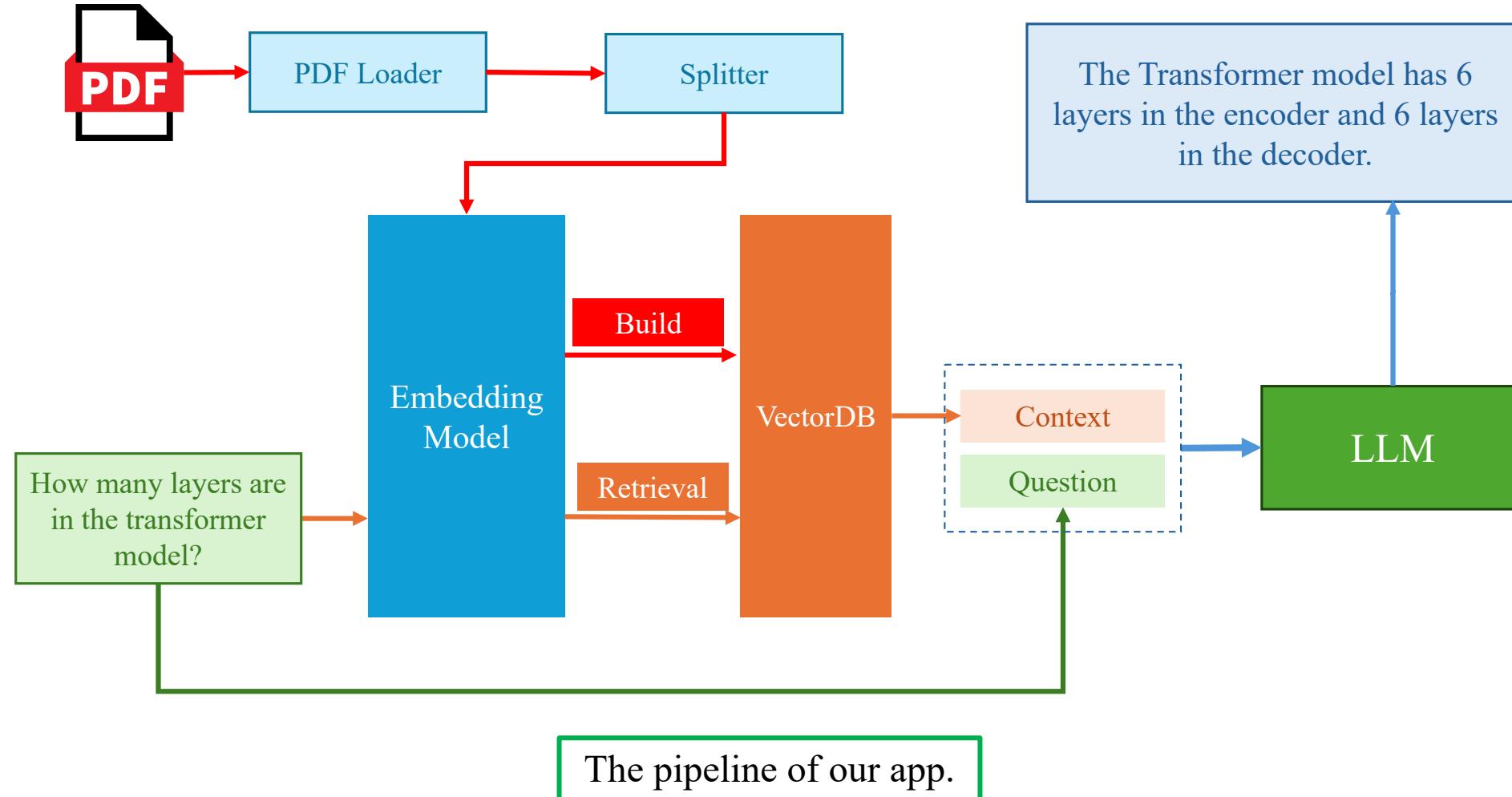
## ❖ Introduction

**Description:** Serve a LLMs RAG application as an API that receive a simple question and return the response of the LLMs (utilizing context retrieved from a vector database).



# RAG with LangChain

## ❖ Introduction



# RAG with LangChain

## ❖ Introduction

### Attention Is All You Need

- Encoder: The encoder is composed of a stack of  $N = 6$  identical layers.
- Decoder: The decoder is also composed of a stack of  $N = 6$  identical layers.



PDF

In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder

Chunking

### BERT



PDF

In this work, we denote the number of layers (i.e., Transformer blocks) as  $L$ , the hidden size as  $H$ , and the number of self-attention heads as  $A$ .

We primarily report results on two model sizes: BERTBASE ( $L=12$ ,  $H=768$ ,  $A=12$ , Total Parameters=110M) and BERTLARGE ( $L=24$ ,  $H=1024$ ,  $A=16$ , Total Parameters=340M)

Query

How many layer in Transformer model?

Embeding



Similarity



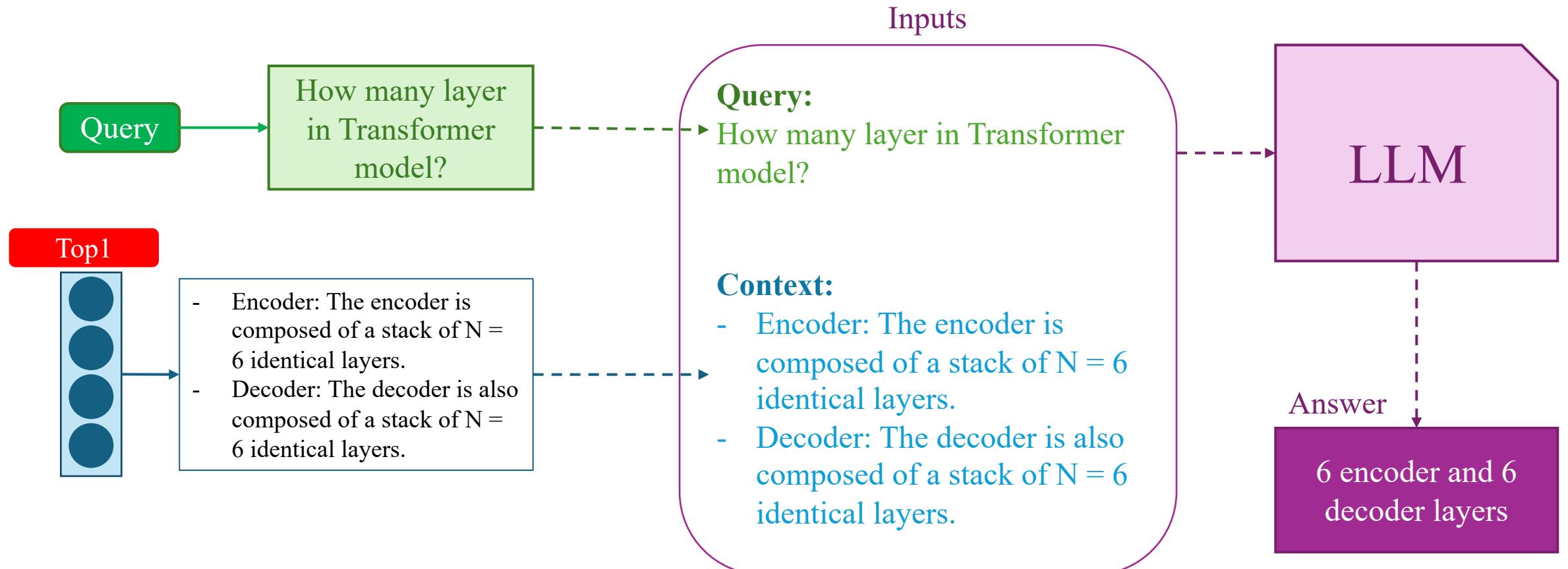
Embeding

Top1



# RAG with LangChain

## ❖ Introduction





# RAG with LangChain

## ❖ Introduction

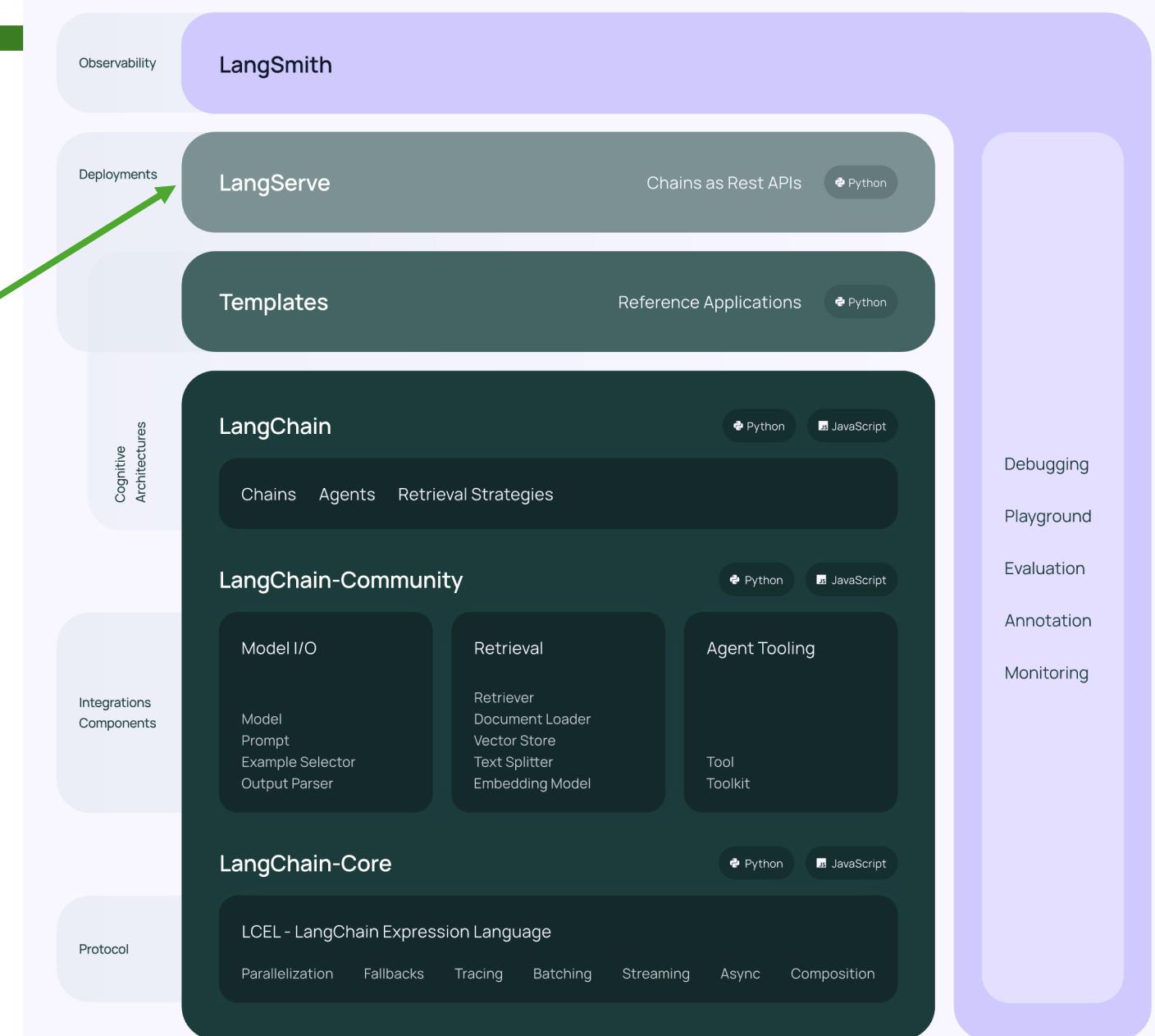
**Description:** LangServe helps developers deploy LangChain runnables and chains as a REST API.

Integrated

FastAPI



pydantic





# RAG with LangChain

## ❖ Features

1. API docs page with JSONSchema and Swagger.
2. Input and Output schemas automatically inferred from your LangChain object.
3. Efficient /invoke, /batch and /stream endpoints.
4. /stream\_log endpoint for streaming all (or some) intermediate steps from your chain/agent
5. Playground page at /playground/ with streaming output and intermediate steps
6. All built with battle-tested open-source Python libraries like FastAPI, Pydantic, uvloop and asyncio.



```
1 # pip3 install langserve==0.0.30
2 from langserve import add_routes
3
4 genai_chain = build_rag_chain(llm, data_dir=genai_docs, data_type="pdf")
5
6 add_routes(app,
7 genai_chain,
8 path="/generative_ai")
```



# RAG with LangChain

## ❖ Features

1. API docs page with JSONSchema and Swagger.
2. Input and Output schemas automatically inferred from your LangChain object
3. Efficient /invoke, /batch and /stream endpoints.
4. /stream\_log endpoint for streaming all (or some) intermediate steps
5. Playground page at /playground/ with streaming output a

**LangChain Server** 1.0 OAS 3.1

[/openapi.json](#)

A simple api server using Langchain's Runnable interfaces

**generative\_ai**

|             |                                              |                             |
|-------------|----------------------------------------------|-----------------------------|
| <b>GET</b>  | <a href="#">/generative_ai/input_schema</a>  | Generative Ai Input Schema  |
| <b>GET</b>  | <a href="#">/generative_ai/output_schema</a> | Generative Ai Output Schema |
| <b>GET</b>  | <a href="#">/generative_ai/config_schema</a> | Generative Ai Config Schema |
| <b>POST</b> | <a href="#">/generative_ai/invoke</a>        | Generative Ai Invoke        |
| <b>POST</b> | <a href="#">/generative_ai/batch</a>         | Generative Ai Batch         |
| <b>POST</b> | <a href="#">/generative_ai/stream</a>        | Generative Ai Stream        |
| <b>POST</b> | <a href="#">/generative_ai/stream_log</a>    | Generative Ai Stream Log    |

# RAG with LangChain

## ❖ Features

1. API docs page with JSONSchema and Swagger.
2. **Input and Output schemas automatically inferred from your LangChain object**
3. Efficient /invoke, /batch and /stream endpoints.
4. /stream\_log endpoint for streaming all (or some) intermediate results.
5. Playground page at /playground/ with streaming output a

**LangChain Server** 1.0 OAS 3.1

[/openapi.json](#)

A simple api server using Langchain's Runnable interfaces

**generative\_ai**

**GET** [/generative\\_ai/input\\_schema](#) Generative Ai Input Schema

**GET** [/generative\\_ai/output\\_schema](#) Generative Ai Output Schema

**GET** [/generative\\_ai/config\\_schema](#) Generative Ai Config Schema

**POST** [/generative\\_ai/invoke](#) Generative Ai Invoke

**POST** [/generative\\_ai/batch](#) Generative Ai Batch

**POST** [/generative\\_ai/stream](#) Generative Ai Stream

**POST** [/generative\\_ai/stream\\_log](#) Generative Ai Stream Log

# RAG with LangChain

## ❖ Features

1. API docs page with JSONSchema and Swagger.
2. Input and Output schemas automatically inferred from your LangChain object
3. Efficient **/invoke**, **/batch** and **/stream** endpoints.
4. **/stream\_log** endpoint for streaming all (or some) intermediate steps
5. Playground page at **/playground/** with streaming output

### LangChain Server 1.0 OAS 3.1

[/openapi.json](#)

A simple api server using Langchain's Runnable interfaces

#### generative\_ai

**GET** /generative\_ai/input\_schema Generative Ai Input Schema

**GET** /generative\_ai/output\_schema Generative Ai Output Schema

**GET** /generative\_ai/config\_schema Generative Ai Config Schema

**POST** /generative\_ai/invoke Generative Ai Invoke

**POST** /generative\_ai/batch Generative Ai Batch

**POST** /generative\_ai/stream Generative Ai Stream

**POST** /generative\_ai/stream\_log Generative Ai Stream Log

# RAG with LangChain

## ❖ Features

1. API docs page with JSONSchema and Swagger.
2. Input and Output schemas automatically inferred from your LangChain object.
3. Efficient /invoke, /batch and /stream endpoints.
4. **/stream\_log endpoint for streaming all (or some) intermediate steps from your chain/agent**
5. Playground page at /playground/ with streaming output and intermediate steps

### generative\_ai

GET /generative\_ai/input\_schema Generative Ai Input Schema

GET /generative\_ai/output\_schema Generative Ai Output Schema

GET /generative\_ai/config\_schema Generative Ai Config Schema

POST /generative\_ai/invoke Generative Ai Invoke

POST /generative\_ai/batch Generative Ai Batch

POST /generative\_ai/stream Generative Ai Stream

POST /generative\_ai/stream\_log Generative Ai Stream Log



# RAG with LangChain

## ❖ Features

1. API docs page with JSONSchema and Swagger.
2. Input and Output schemas automatically inferred from your LangChain object.
3. Efficient /invoke, /batch and /stream endpoints.
4. /stream\_log endpoint for streaming all (or some) intermediate steps from your chain/agent
5. Playground page at /playground/ with streaming output and intermediate steps

The screenshot shows a dark-themed web application interface. At the top, a header bar displays the URL "localhost:5000/generative\_ai/playground/". Below the header, the title "LangServe Playground" is centered, accompanied by a small green parrot icon. A "Try it" button is located just below the title. The main area features a rounded rectangular input field labeled "Inputs". Inside the input field, the placeholder text "RUNNABLEPARALLEL<CONTEXT,QUESTION>INPUT" is visible, along with the note "must be string" positioned to the right of the input field. At the bottom of the screen, there are two large, rounded rectangular buttons: a white "Share" button on the left and a blue "Start" button on the right.



# RAG with LangChain

## ❖ Playground

 **LangServe Playground**

**Try it**

**Inputs**

RUNNABLEPARALLEL<CONTEXT,QUESTION>INPUT

What is Chain-of-Thought Prompting?

**Reset**

**Output**

Chain-of-thought prompting is a method of using natural language to help solve problems by breaking down the problem into smaller steps and solving each step in order. It is a form of chain-of-thought prompting that involves using a chain of thought to solve a problem.

Intermediate steps 8 >

Share > Start

# RAG with LangChain

## ❖ Playground: Intermediate Steps

Intermediate steps 8

RunnableParallel<context,question>

a minute ago

```
{
 "question": "What is Chain-of-Thought Prompting?",
 "context": "computation by itself is not the reason for the success of chain-of-thought prompting, and that there appears to be utility from expressing intermediate steps via natural language.\nChain of thought after answer. Another potential benefit of\nChain-of-Thought Prompting\nConsider one's own thought process when solving a complicated reasoning task such as a multi-step\nmath word problem. It is typical to decompose the problem into intermediate steps and solve each\ntasks, given a prompt that consists of triples: input, chain of thought , output. A chain of thought is\na series of intermediate natural language reasoning steps that lead to the final output, and we refer to\nthis approach as chain-of-thought prompting . An example prompt is shown in Figure 1.\nIn form the standard baseline by a large margin. This result\nimplies that successful use of chain of thought does not\ndepend on a particular linguistic style.\nTo confirm that successful chain-of-thought prompting\nworks for other sets of exemplars, we also run experiments\nchain-of-thought prompting in this form can successfully elicit successful reasoning across a range of\nsymbolic), chain-of-thought prompting can potentially be applied to any task for which humans use a chain of thought to solve (at least in principle). We leave the empirical evaluation of chain-of-thought\nprompting on such diverse tasks (e.g., machine translation, etc.) to future work.\nIn exemplar order for all other models.\n3.2 Results\nThe strongest results of chain-of-thought prompting are summarized in Figure 4, with all experimental\noutputs for each model collection, model size, and benchmark shown in Table 2 in the Appendix.\nChain-of-thought prompting could simply be that such prompts\nallow the model to better access relevant knowledge acquired\nduring pretraining. Therefore, we test an alternative configuration where the chain of thought prompt is only given after the\nthat it would have otherwise gotten incorrect. The chain of thought in this case resembles a solution\nand can interpreted as one, but we still opt to call it a chain of thought to better capture the idea that it\nAppendix Table 6 / Table 7). Although there is variance\namong different chain of thought annotations, as would be\nexpected when using exemplar-based prompting (Le Scao\nand Rush, 2021; Reynolds and McDonell, 2021; Zhao\net al., 2021), all sets of chain of thought prompts output-"
}
```

solution\nand can be interpreted as one, but we still opt to call it a chain of thought to better capture the idea that it\nAppendix Table 6 / Table 7). Although there is variance\namong different chain of thought annotations, as would be\nexpected when using exemplar-based prompting (Le Scao\nand Rush, 2021; Reynolds and McDonell, 2021; Zhao\net al., 2021), all sets of chain of thought prompts output-  
Answer:  
Chain-of-thought prompting is a method of using natural language to help solve problems by breaking down the problem into smaller steps and solving each step in order. It is a form of chain-of-thought prompting that involves using a chain of thought to solve a problem.",  
 "generation\_info": null,  
 "type": "Generation"  
}  
]  
],  
"llm\_output": null,  
"run": null  
}

Str\_OutputParser

a minute ago

```
{
 "output": "Chain-of-thought prompting is a method of using natural language to help solve problems by breaking down the problem into smaller steps and solving each step in order. It is a form of chain-of-thought prompting that involves using a chain of thought to solve a problem."
}
```

# RAG with LangChain

## ❖ Source code structure

```
rag_langchain/
 └── data_source/
 └── generative_ai/
 └── download.py

 └── src/
 ├── base/
 └── llm_model.py

 └── rag/
 ├── file_loader.py
 ├── main.py
 ├── offline_rag.py
 ├── utils.py
 └── vectorstore.py

 └── app.py

└── requirements.txt
```

- **data\_source/**: Folder containing corpus for retrieval.
- **src/base/**: Folder containing code to initialize LLM.
- **src/rag/**: Folder containing code for RAG.
- **app.py**: Python file containing codes for FastAPI app initialization.
- **requirements.txt**: File containing packages version information to run the source code.

# RAG with LangChain

## ❖ Step 1: Download corpus

Provided proper attribution is provided, Google hereby grants permission to reproduce the tables and figures in this paper solely for use in journalistic or scholarly works.

### Attention Is All You Need

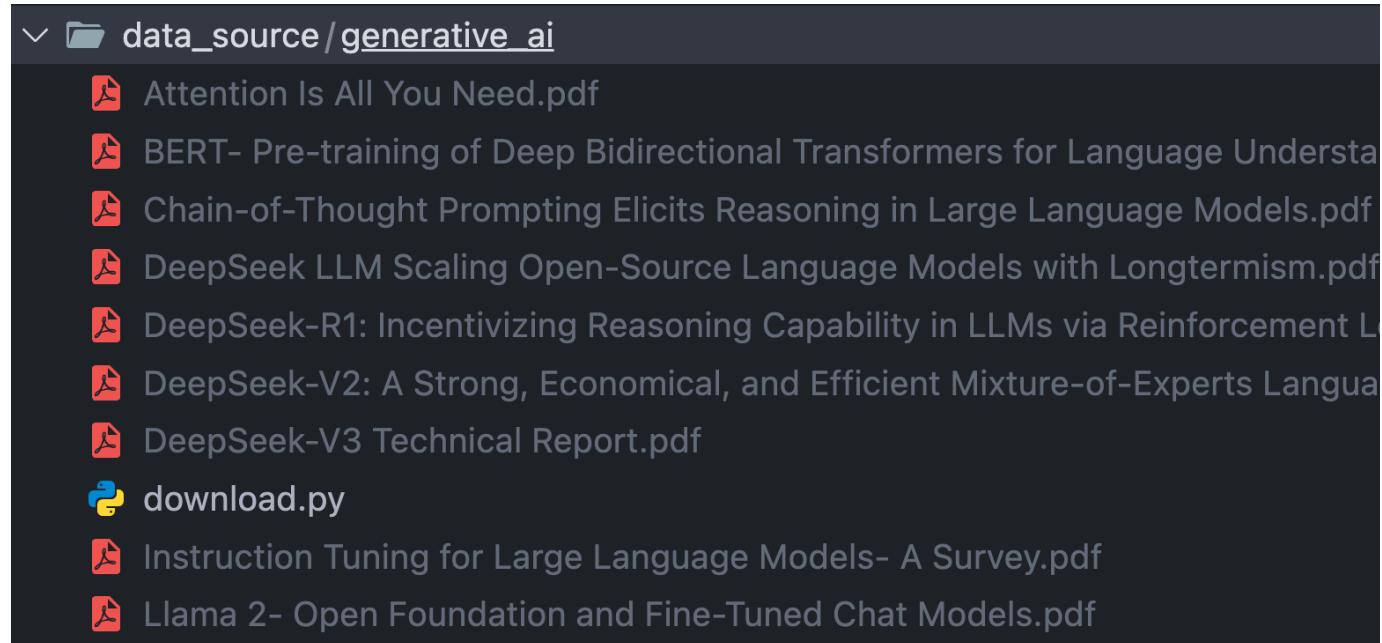
**Ashish Vaswani\***  
Google Brain  
avaswani@google.com      **Noam Shazeer\***  
Google Brain  
noam@google.com      **Niki Parmar\***  
Google Research  
nikip@google.com      **Jakob Uszkoreit\***  
Google Research  
usz@google.com

**Llion Jones\***  
Google Research  
llion@google.com      **Aidan N. Gomez\* †**  
University of Toronto  
aidan@cs.toronto.edu      **Lukasz Kaiser\***  
Google Brain  
lukaszkaiser@google.com

**Illia Polosukhin\* ‡**  
illia.polosukhin@gmail.com

### Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

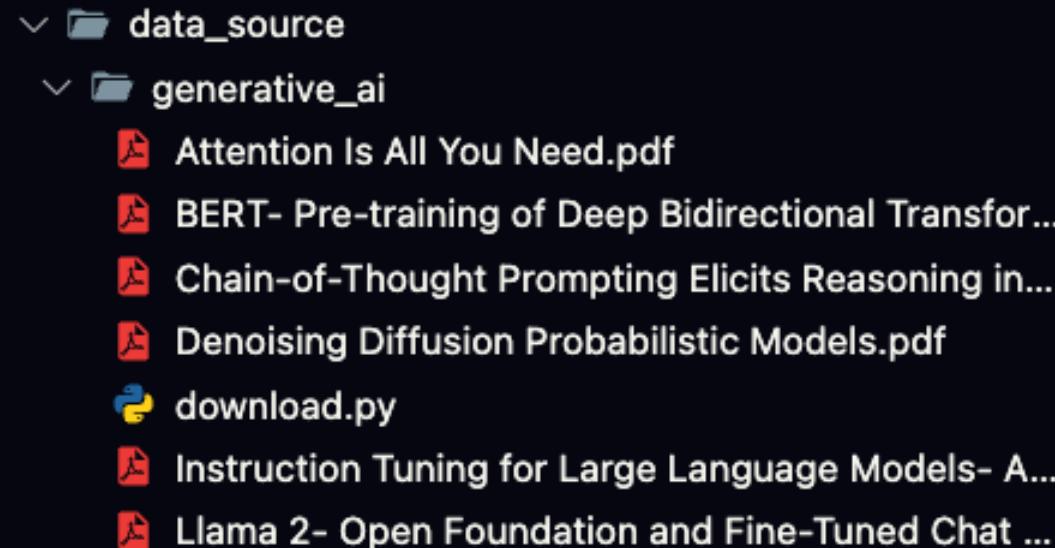




# RAG with LangChain

## ❖ Step 1: Download corpus

Update download.py:



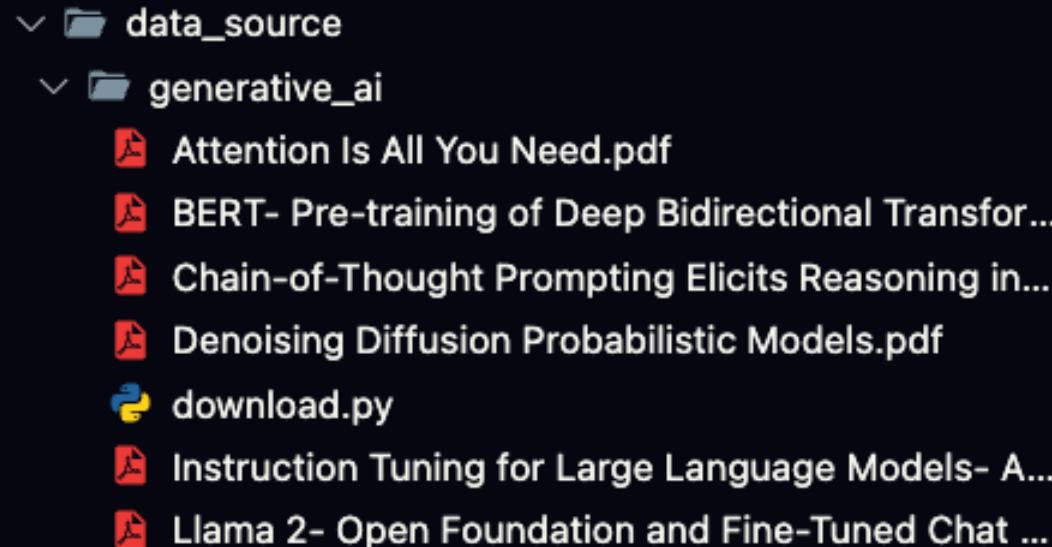
```
•••
1 file_links = [
2 {
3 "title": "Attention Is All You Need",
4 "url": "https://arxiv.org/pdf/1706.03762"
5 },
6 {
7 "title": "BERT- Pre-training of Deep Bidirectional Transformers for Language
Understanding",
8 "url": "https://arxiv.org/pdf/1810.04805"
9 },
10 {
11 "title": "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models",
12 "url": "https://arxiv.org/pdf/2201.11903"
13 },
14 {
15 "title": "Denoising Diffusion Probabilistic Models",
16 "url": "https://arxiv.org/pdf/2006.11239"
17 },
18 {
19 "title": "Instruction Tuning for Large Language Models- A Survey",
20 "url": "https://arxiv.org/pdf/2308.10792"
21 },
22 {
23 "title": "Llama 2- Open Foundation and Fine-Tuned Chat Models"
24 "url": "https://arxiv.org/pdf/2307.09288"
25 }
26]
```



# RAG with LangChain

## ❖ Step 1: Download corpus

Update download.py:

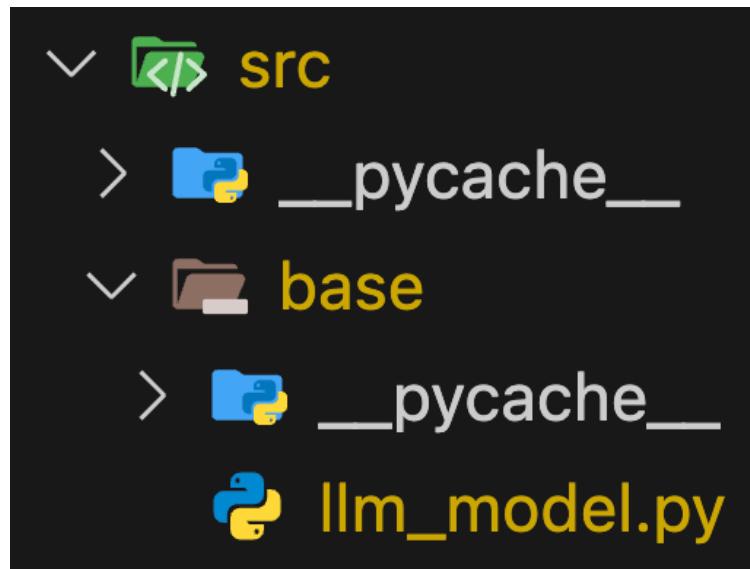


```
1 import os
2 import wget
3
4 def is_exist(file_link):
5 return os.path.exists(f"./{file_link['title']}.pdf")
6
7 for file_link in file_links:
8 if not is_exist(file_link):
9 wget.download(file_link["url"], out=f"./{file_link['title']}.pdf")
```

# RAG with LangChain

## ❖ Step 2: Build the load LLM function

Update src/base/llm\_model.py:



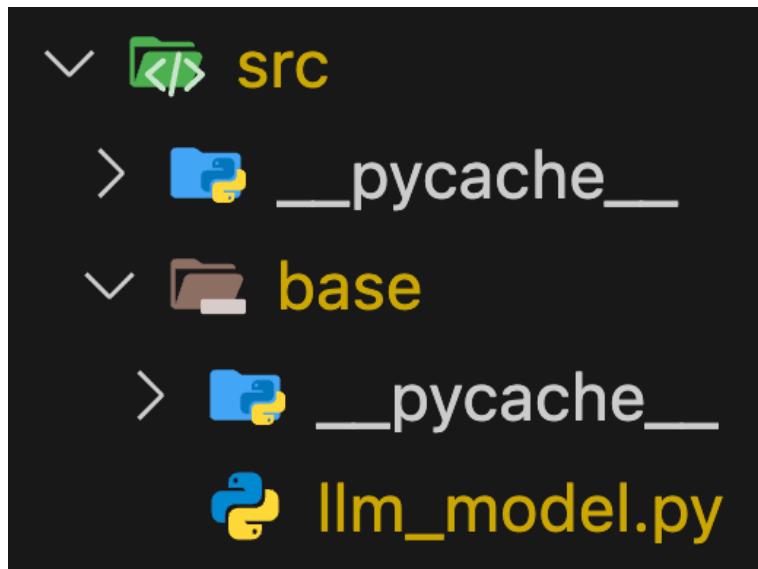
```
1 import torch
2 from transformers import BitsAndBytesConfig
3 from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
4 from langchain.llms.huggingface_pipeline import HuggingFacePipeline
5
6 nf4_config = BitsAndBytesConfig(
7 load_in_4bit=True,
8 bnb_4bit_quant_type="nf4",
9 bnb_4bit_use_double_quant=True,
10 bnb_4bit_compute_dtype=torch.bfloat16
11)
```



# RAG with LangChain

## ❖ Step 2: Build the load LLM function

Update src/base/llm\_model.py:



```
1 def get_hf_llm(model_name: str = "mistralai/Mistral-7B-Instruct-v0.2",
2 max_new_token = 1024,
3 **kwargs):
4
5 model = AutoModelForCausalLM.from_pretrained(
6 model_name,
7 quantization_config=nf4_config,
8 low_cpu_mem_usage=True
9)
10 tokenizer = AutoTokenizer.from_pretrained(model_name)
11
12 model_pipeline = pipeline(
13 "text-generation",
14 model=model,
15 tokenizer=tokenizer,
16 max_new_tokens=max_new_token,
17 pad_token_id=tokenizer.eos_token_id,
18 device_map="auto"
19)
20
21 llm = HuggingFacePipeline(
22 pipeline=model_pipeline,
23 model_kwargs=kwargs
24)
25
26 return llm
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/main.py:

```
✓ src
 > __pycache__
 > base
 > chat
✓ rag
 > __pycache__
 > file_loader.py
 > main.py highlighted
 > offline_rag.py
 > utils.py
 > vectorstore.py
```

```
1 from pydantic import BaseModel, Field
2
3 from src.rag.file_loader import Loader
4 from src.rag.vectorstore import VectorDB
5 from src.rag.offline_rag import Offline_RAG
6
7 class InputQA(BaseModel):
8 question: str = Field(..., title="Question to ask the model")
9
10 class OutputQA(BaseModel):
11 answer: str = Field(..., title="Answer from the model")
12
13 def build_rag_chain(llm, data_dir, data_type):
14 doc_loaded = Loader(file_type=data_type).load_dir(data_dir, workers=2)
15 retriever = VectorDB(documents = doc_loaded).get_retriever()
16 rag_chain = Offline_RAG(llm).get_chain(retriever)
17
18 return rag_chain
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/offline\_rag.py:

```
✓ src
 > __pycache__
 > base
 > chat
 ✓ rag
 > __pycache__
 > file_loader.py
 > main.py
 > offline_rag.py
 > utils.py
 > vectorstore.py
```

```
1 import re
2 from langchain import hub
3 from langchain_core.runnables import RunnablePassthrough
4 from langchain_core.output_parsers import StrOutputParser
5
6 class Str_OutputParser(StrOutputParser):
7 def __init__(self) -> None:
8 super().__init__()
9
10 def parse(self, text: str) -> str:
11 return self.extract_answer(text)
12
13
14 def extract_answer(self,
15 text_response: str,
16 pattern: str = r"Answer:\s*(.*)"
17) -> str:
18
19 match = re.search(pattern, text_response, re.DOTALL)
20 if match:
21 answer_text = match.group(1).strip()
22 return answer_text
23 else:
24 return text_response
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/offline\_rag.py:

```
✓ 📂 src
 > 📂 __pycache__
 > 📂 base
 > 📂 chat
 ✓ 📂 rag
 > 📂 __pycache__
 > 🐍 file_loader.py
 > 🐍 main.py
 > 🐍 offline_rag.py
 > 🐍 utils.py
 > 🐍 vectorstore.py
```

```
● ● ●

1
2 class Offline_RAG:
3 def __init__(self, llm) -> None:
4 self.llm = llm
5 self.prompt = hub.pull("rlm/rag-prompt")
6 self.str_parser = Str_OutputParser()
7
8 def get_chain(self, retriever):
9 input_data = {
10 "context": retriever | self.format_docs,
11 "question": RunnablePassthrough()
12 }
13 rag_chain = (
14 input_data
15 | self.prompt
16 | self.llm
17 | self.str_parser
18)
19 return rag_chain
20
21 def format_docs(self, docs):
22 return "\n\n".join(doc.page_content for doc in docs)
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/utils.py:

```
✓ src
 > __pycache__
 > base
 > chat
 ✓ rag
 > __pycache__
 > file_loader.py
 > main.py
 > offline_rag.py
 > utils.py
 > vectorstore.py
```

```
● ● ●

1 import re
2
3 def extract_answer(text_response: str,
4 pattern: str = r"Answer:\s*(.*)"
5) -> str:
6
7 match = re.search(pattern, text_response)
8 if match:
9 answer_text = match.group(1).strip()
10 return answer_text
11 else:
12 return "Answer not found."
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/vectorstore.py:

```
✓ src
 > __pycache__
 > base
 > chat
 ✓ rag
 > __pycache__
 > file_loader.py
 > main.py
 > offline_rag.py
 > utils.py
 > vectorstore.py
```

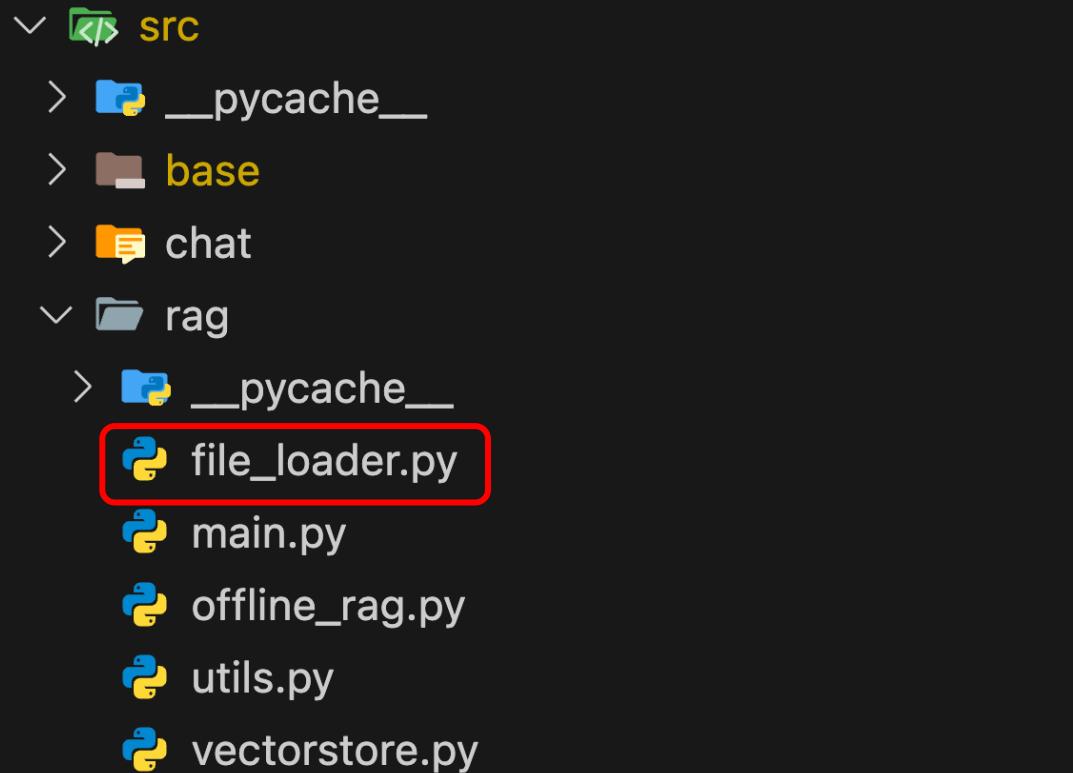
```
 1 from typing import Union
 2 from langchain_chroma import Chroma
 3 from langchain_community.vectorstores import FAISS
 4 from langchain_community.embeddings import HuggingFaceEmbeddings
 5
 6 class VectorDB:
 7 def __init__(self,
 8 documents = None,
 9 vector_db: Union[Chroma, FAISS] = Chroma,
10 embedding = HuggingFaceEmbeddings(),
11) -> None:
12
13 self.vector_db = vector_db
14 self.embedding = embedding
15 self.db = self._build_db(documents)
16
17 def _build_db(self, documents):
18 db = self.vector_db.from_documents(documents=documents,
19 embedding=self.embedding)
20 return db
21
22 def get_retriever(self,
23 search_type: str = "similarity",
24 search_kwargs: dict = {"k": 10}
25):
26 retriever = self.db.as_retriever(search_type=search_type,
27 search_kwargs=search_kwargs)
28
29 return retriever
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/file\_loader.py:



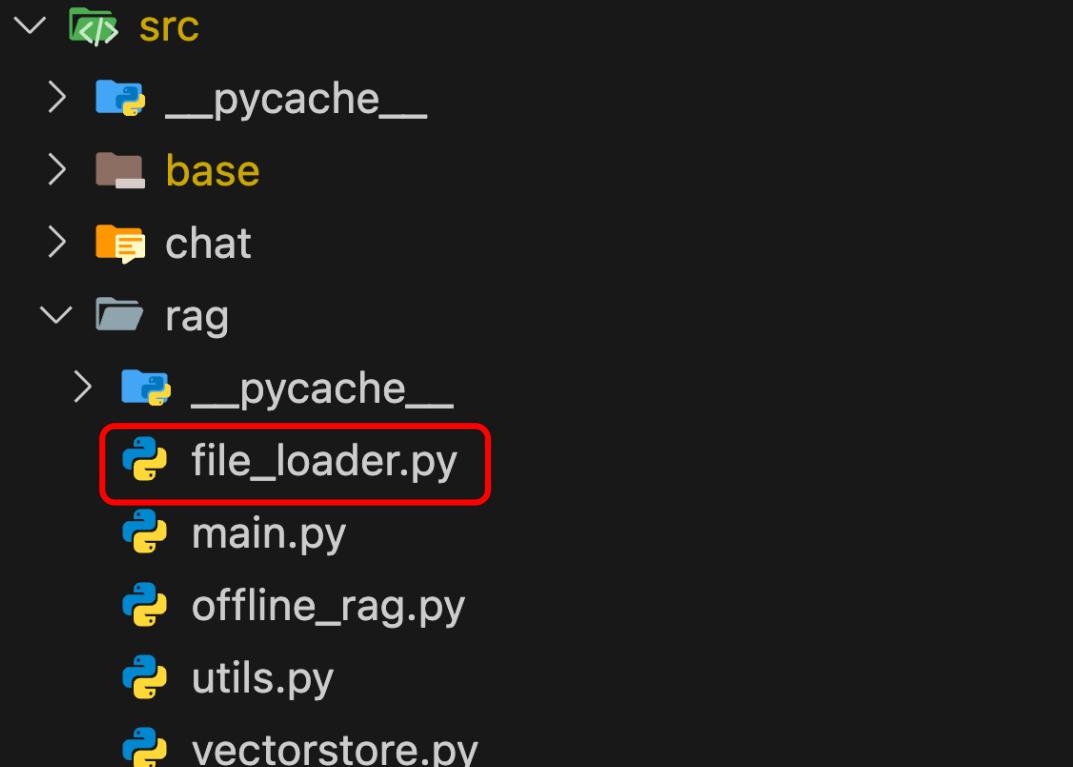
```
1 from typing import Union, List, Literal
2 import glob
3 from tqdm import tqdm
4 import multiprocessing
5 from langchain_community.document_loaders import PyPDFLoader
6 from langchain_text_splitters import RecursiveCharacterTextSplitter
7
8 def remove_non_utf8_characters(text):
9 return ''.join(char for char in text if ord(char) < 128)
10
11 def load_pdf(pdf_file):
12 docs = PyPDFLoader(pdf_file, extract_images=True).load()
13 for doc in docs:
14 doc.page_content = remove_non_utf8_characters(doc.page_content)
15 return docs
16
17 def get_num_cpu():
18 return multiprocessing.cpu_count()
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/file\_loader.py:



```
1 class BaseLoader:
2 def __init__(self) -> None:
3 self.num_processes = get_num_cpu()
4
5 def __call__(self, files: List[str], **kwargs):
6 pass
7
8 class PDFLoader(BaseLoader):
9 def __init__(self) -> None:
10 super().__init__()
11
12 def __call__(self, pdf_files: List[str], **kwargs):
13 num_processes = min(self.num_processes, kwargs["workers"])
14 with multiprocessing.Pool(processes=num_processes) as pool:
15 doc_loaded = []
16 total_files = len(pdf_files)
17 with tqdm(total=total_files, desc="Loading PDFs", unit="file") as pbar:
18 for result in pool.imap_unordered(load_pdf, pdf_files):
19 doc_loaded.extend(result)
20 pbar.update(1)
21
22 return doc_loaded
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/file\_loader.py:

```
✓ src
 > __pycache__
 > base
 > chat
✓ rag
 > __pycache__
 > file_loader.py
 └── main.py
 └── offline_rag.py
 └── utils.py
 └── vectorstore.py
```

```
● ● ●

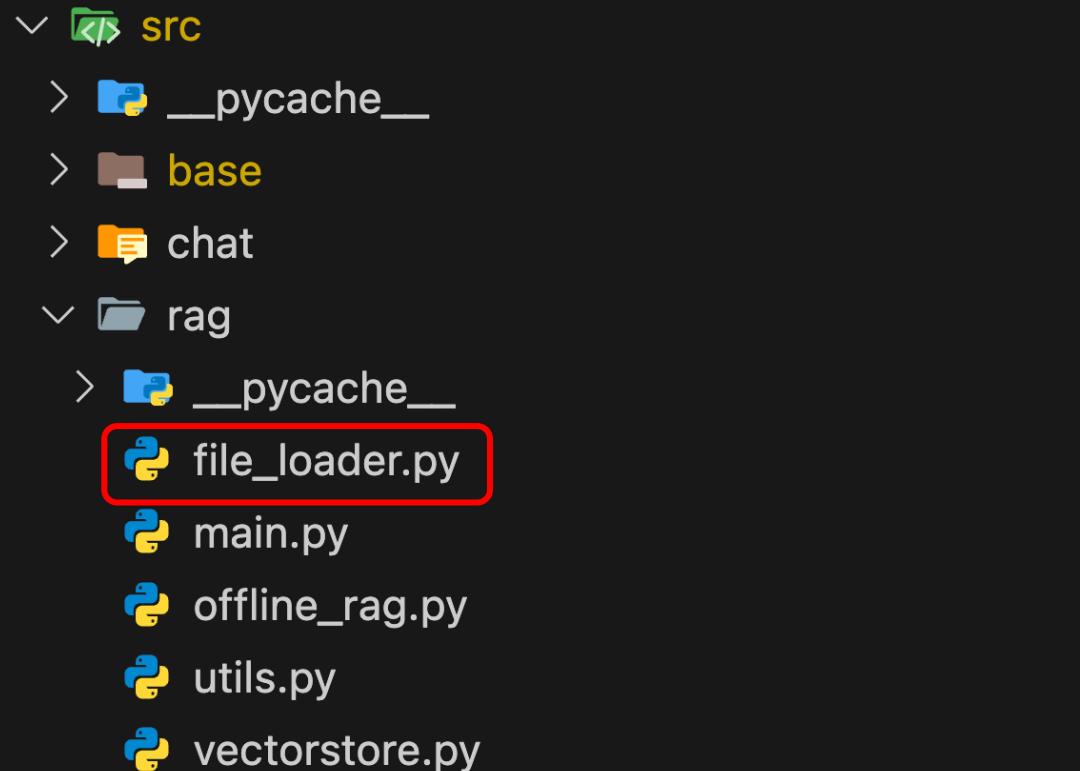
1 class TextSplitter:
2 def __init__(self,
3 separators: List[str] = ['\n\n', '\n', ' ', ''],
4 chunk_size: int = 300,
5 chunk_overlap: int = 0
6) -> None:
7
8 self.splitter = RecursiveCharacterTextSplitter(
9 separators=separators,
10 chunk_size=chunk_size,
11 chunk_overlap=chunk_overlap,
12)
13 def __call__(self, documents):
14 return self.splitter.split_documents(documents)
```



# RAG with LangChain

## ❖ Step 3: Build RAG source code

Update src/rag/file\_loader.py:



```
1 class Loader:
2 def __init__(self,
3 file_type: str = Literal["pdf"],
4 split_kwargs: dict = {
5 "chunk_size": 300,
6 "chunk_overlap": 0}
7) -> None:
8 assert file_type in ["pdf"], "file_type must be pdf"
9 self.file_type = file_type
10 if file_type == "pdf":
11 self.doc_loader = PDFLoader()
12 else:
13 raise ValueError("file_type must be pdf")
14
15 self.doc_splitter = TextSplitter(**split_kwargs)
16
17 def load(self, pdf_files: Union[str, List[str]], workers: int = 1):
18 if isinstance(pdf_files, str):
19 pdf_files = [pdf_files]
20 doc_loaded = self.doc_loader(pdf_files, workers=workers)
21 doc_split = self.doc_splitter(doc_loaded)
22 return doc_split
23
24 def load_dir(self, dir_path: str, workers: int = 1):
25 if self.file_type == "pdf":
26 files = glob.glob(f"{dir_path}/*.pdf")
27 assert len(files) > 0, f"No {self.file_type} files found in {dir_path}"
28 else:
29 raise ValueError("file_type must be pdf")
30
31 return self.load(files, workers)
```



# RAG with LangChain

## ❖ Step 4: Build API

Update src/app.py:

```
✓ 📂 src
 > 📂 __pycache__
 > 📂 base
 > 📂 chat
 > 📂 rag
 🐍 app.py
```

```
● ● ●

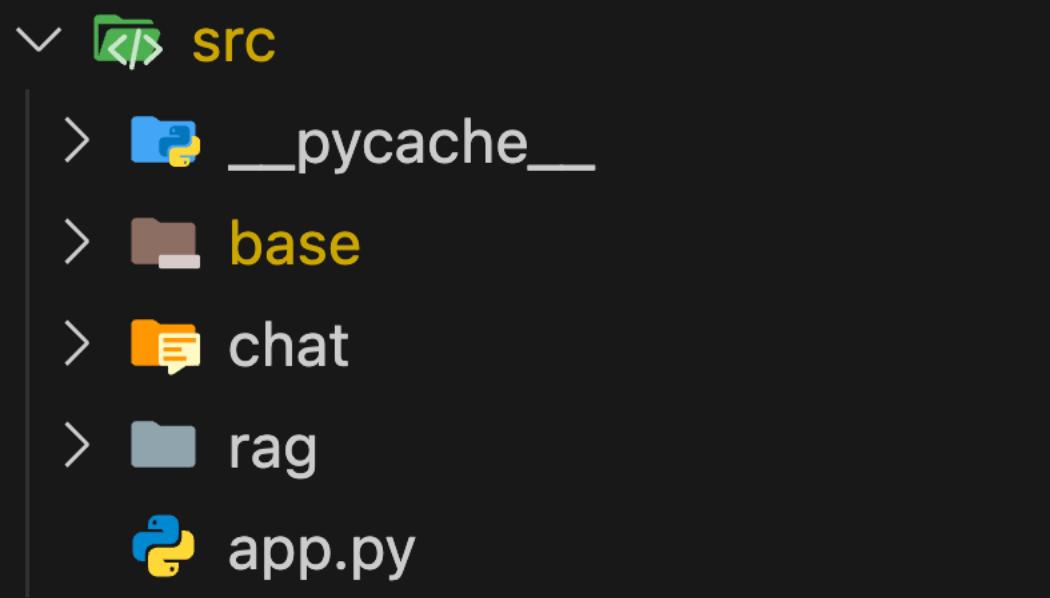
1 import os
2 os.environ["TOKENIZERS_PARALLELISM"] = "false"
3
4 from fastapi import FastAPI
5 from fastapi.middleware.cors import CORSMiddleware
6
7 from langserve import add_routes
8
9 from src.base.llm_model import get_hf_llm
10 from src.rag.main import build_rag_chain, InputQA, OutputQA
11
12 llm = get_hf_llm(temperature=0.9)
13 genai_docs = "./data_source/generative_ai"
```



# RAG with LangChain

## ❖ Step 4: Build API

Update src/app.py:



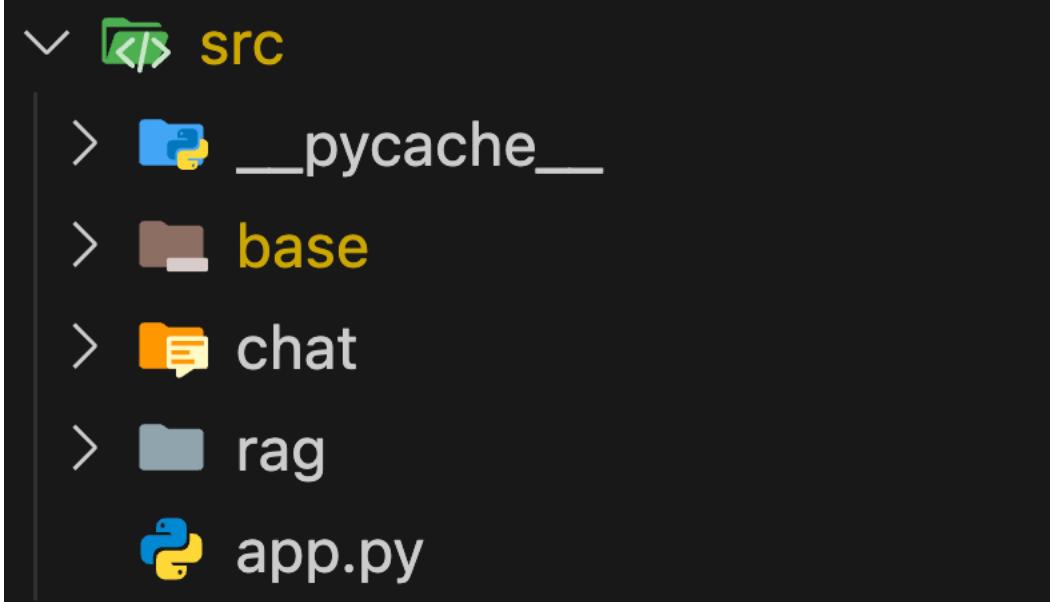
```
● ● ●
1 import os
2 os.environ["TOKENIZERS_PARALLELISM"] = "false"
3
4 from fastapi import FastAPI
5 from fastapi.middleware.cors import CORSMiddleware
6
7 from langserve import add_routes
8
9 from src.base.llm_model import get_hf_llm
10 from src.rag.main import build_rag_chain, InputQA, OutputQA
11
12 llm = get_hf_llm(temperature=0.9)
13 genai_docs = "./data_source/generative_ai"
14
15 # ----- Chains-----
16
17 genai_chain = build_rag_chain(llm, data_dir=genai_docs,
 data_type="pdf")
```



# RAG with LangChain

## ❖ Step 4: Build API

Update src/app.py:



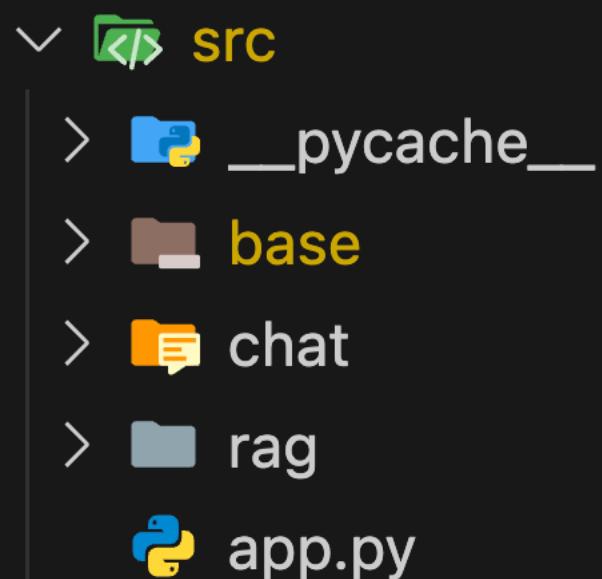
```
1 # ----- App - FastAPI -----
2
3 app = FastAPI(
4 title="LangChain Server",
5 version="1.0",
6 description="A simple api server using Langchain's Runnable
7 interfaces",
8)
9 app.add_middleware(
10 CORSMiddleware,
11 allow_origins=["*"],
12 allow_credentials=True,
13 allow_methods=["*"],
14 allow_headers=["*"],
15 expose_headers=["*"],
16)
```



# RAG with LangChain

## ❖ Step 4: Build API

Update src/app.py:



```
1 # ----- Routes - FastAPI -----
2
3 @app.get("/check")
4 async def check():
5 return {"status": "ok"}
6
7
8 @app.post("/generative_ai", response_model=OutputQA)
9 async def generative_ai(inputs: InputQA):
10 answer = genai_chain.invoke(inputs.question)
11 return {"answer": answer}
12
13 # ----- Langserve Routes - Playground -----
14 add_routes(app,
15 genai_chain,
16 playground_type="default",
17 path="/generative_ai")
```

# RAG with LangChain

## ❖ Step 5: Deploy and test API

```
INFO: Will watch for changes in these directories: ['/home/server-ailab/ThangDuongTeam/thangdd/TA/240504_Extra_LangChain/Langchain_Services']
INFO: Uvicorn running on http://0.0.0.0:5000 (Press CTRL+C to quit)
INFO: Started reloader process [6115] using WatchFiles
Loading checkpoint shards: 100% | 3/3 [00:08<00:00, 2.83s/it]
Loading PDFs: 100% | 6/6 [00:41<00:00, 6.96s/file]
INFO: Started server process [6117]
INFO: Waiting for application startup.

LANGSERVE
LANGSERVE: Playground for chain "/generative_ai/" is live at:
LANGSERVE: ↳ /generative_ai/playground/
LANGSERVE: See all available routes at /docs/
LANGSERVE: ▲ Using pydantic 2.7.0. OpenAPI docs for invoke, batch, stream, stream_log endpoints will not be generated. API endpoints and playground should work as expected. If you need to see the docs, you can downgrade to pydantic 1. For example, `pip install pydantic==1.10.13`. See https://github.com/tiangolo/fastapi/issues/10360 for details.

INFO: Application startup complete.
```

```
uvicorn src.app:app --host "0.0.0.0" --port 5000 --reload
```



# RAG with LangChain

## ❖ Step 5: Deploy and test API

**LangChain Server 1.0 OAS 3.1**  
[/openapi.json](#)

A simple api server using Langchain's Runnable interfaces

**generative\_ai** ⚠️ Using pydantic 2.7.0. OpenAPI docs for `invoke`, `batch`, `stream`, `stream_log` endpoints will not be generated. API endpoints and playground should work as expected. If you need to see the docs, you can downgrade to pydantic 1. For example, `pip install pydantic==1.10.13` See <https://github.com/tiangolo/fastapi/issues/10360> for details.

**GET /generative\_ai/input\_schema** Generative Ai Input Schema

**GET /generative\_ai/output\_schema** Generative Ai Output Schema

**GET /generative\_ai/config\_schema** Generative Ai Config Schema

**generative\_ai/config** Endpoints with a default configuration set by `config_hash` path parameter. Used in conjunction with share links generated using the LangServe UI playground. The hash is an LZString compressed JSON string.

**GET /generative\_ai/c/{config\_hash}/input\_schema** Generative Ai Input Schema With Config

**GET /generative\_ai/c/{config\_hash}/output\_schema** Generative Ai Output Schema With Config

**GET /generative\_ai/c/{config\_hash}/config\_schema** Generative Ai Config Schema With Config

**default**

**GET /check** Check

**POST /generative\_ai** Generative Ai

**POST /generative\_ai/token\_feedback** Create Feedback From Token



# RAG with LangChain

## ❖ Step 5: Deploy and test API

Curl

```
curl -X 'POST' \
'http://0.0.0.0:5050/generative_ai' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
 "question": "What is instruction tuning in LLMs?"
}'
```



Request URL

[http://0.0.0.0:5050/generative\\_ai](http://0.0.0.0:5050/generative_ai)

Server response

Code Details

200

Response body

```
{
 "answer": "Instruction tuning is a technique used to enhance the capabilities and controllability of large language models (LLMs) by fine-tuning them on a dataset consisting of instruction-output pairs in a supervised manner. This process bridges the gap between the next-word prediction objective of LLMs and the user's objective of instruction following. It allows for more controllable and predictable model behavior and provides a channel for humans to intervene with the model's behaviors. Instruction tuning is also computationally efficient and can help LLMs rapidly adapt to a specific domain without extensive retraining or architectural changes. However, crafting high-quality instructions that properly cover the desired target behaviors is a challenge."
}
```



Download



# RAG with LangChain

## ❖ Deploy with UI

The screenshot shows the AI Platform interface with the following details:

- Header:** Shows the project name "VLAI-AIVN / AIO2024M10\_RAG\_Langchain" with a "Running" status, "like 0", and "Logs" button.
- Top Bar:** Includes "App", "Files", "Community", "Settings", and other navigation icons.
- Project Overview:** Displays the "Generative AI Q&A" section with the sub-section "RAG System with LangChain". It also lists "AIO2024 Module 10 Project" and "Ask questions about AI, transformers, LLMs, DeepSeek and more".
- Form:** A large form for asking questions. The "Question" field contains the placeholder "Enter your question here...". The "Answer" field contains the placeholder "The answer will appear here...".
- Buttons:** A prominent red "Ask Question" button at the bottom of the form.

# Summarization and Q&A

# Summarization and Q&A

**In this lecture, we have discussed:**

- 1. What is LLMs in production?**
  1. How is it differ from LLMs in research?
  2. What are some challenges when deploying LLMs in production?
- 2. Basics of LangChain**
  1. The key concept of LangChain.
  2. Basic components of LangChain: Prompt Template, LLM Chain, Document Loader...
- 3. How to build an API using LangChain.**
- 4. How to build a RAG application using LangChain.**
  1. Retrieve and answer questions related to Academic Paper.

# Summarization and Q&A

---

