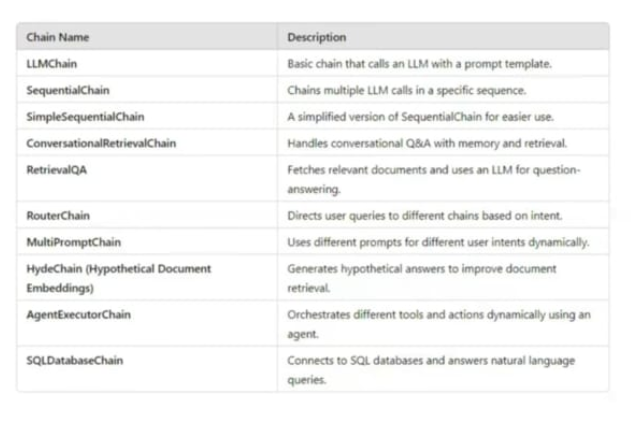
**Chains in langchain**



**What is LLM?**

A [**Large Language Model**](https://www.analyticsvidhya.com/blog/2023/03/an-introduction-to-large-language-models-llms/) (LLM) refers to a type of artificial intelligence designed to understand and generate human-like text. These models, like OpenAI’s GPT-3.5, train on extensive text data to understand the patterns and structures of human language. They can perform various language-related tasks, including translation, content creation, answering questions, and more.

LLMs are valuable tools in natural language processing and have applications in areas like chatbots, content generation, and language translation services.

What is LangChain?

Before we unravel the intricacies of LangChain Chains, let’s grab the essence of LangChain itself. LangChain is a robust library designed to simplify interactions with various large language model (LLM) providers, including OpenAI, Cohere, Bloom, Huggingface, and others. What sets LangChain apart is its unique feature: the ability to create Chains, and logical connections that help in bridging one or multiple LLMs.

**Why Use LangChain?**

[**LangChain**](https://www.analyticsvidhya.com/blog/2023/07/building-llm-powered-applications-with-langchain/) provides limitless opportunities, limited only by your imagination.

* Imagine chatbots that not only provide information but also engage users with wit and charm.
* Picture e-commerce platforms suggesting products so accurately that customers are compelled to make a purchase.
* Envision healthcare apps offering personalized medical insights, empowering individuals to make informed decisions about their well-being.

With LangChain, you have the power to create extraordinary experiences. The potential to transform these ideas into reality is right at your fingertips.

Understanding Chains in LangChain

Central to LangChain is a vital component known as LangChain Chains, forming the core connection among one or several large language models (LLMs).

In certain sophisticated applications, it becomes necessary to chain LLMs together, either with each other or with other elements. These Chains empower us to integrate numerous components, weaving them into a cohesive application. Let’s delve deeper into the distinct types of Chains.

Moreover, the structured approach offered by Chains in LLM ensures flawless and effective processing, paving the way for the development of advanced applications tailored to a wide array of user requirements. This represents a significant advancement in the realm of natural language processing, as these intricate connections serve as the fundamental framework of LangChain, facilitating seamless interactions among multiple Large Language Models (LLMs).

But first, Why Chains?

Chains are invaluable due to their capacity to effortlessly blend diverse components, shaping a singular and coherent application. Through the creation of chains, multiple elements can seamlessly come together. Imagine this scenario: a chain is crafted to take in user input, polish it using a PromptTemplate, and subsequently pass on this refined response to a large language model (LLM). This streamlined process not only simplifies but also enriches the overall functionality of the system. In essence, chains serve as the linchpin, seamlessly connecting different parts of the application and enhancing its capabilities. Let’s summarize this:

* Integrating prompt templates with LLMs allows for a powerful synergy.
* By taking the output of one LLM and using it as input for the next, it becomes feasible to connect multiple LLMs in a sequential fashion.
* Blending LLMs with external data enables the system to respond to inquiries effectively.
* Integrating LLMs with long-term memory, such as chat history, enhances the overall context and depth of interactions.

Furthermore, chains provide us with the ability to build complex applications by linking multiple chains together or by incorporating chains with other vital elements. This approach enables a sophisticated and nuanced method for developing applications, allowing for intricate and advanced functionalities.

**Types of Chain**

There are many different Chains in Langchain that we can use. Here, we are going through three of the fundamental chains – LLM Chain, Sequential Chain and Router Chain.

**LLM Chain – The simplest chain**

The most basic form of chain within this system is the LLMChain, widely recognized and fundamental. Its operation involves a structured arrangement, including a PromptTemplate, an OpenAI model (either a Large Language Model or a ChatModel), and an optional output parser. Within this setup, the LLMChain accepts various input parameters. It employs the PromptTemplate to transform these inputs into a coherent prompt. This polished prompt is then inputted into the model. After receiving the output, the LLMChain uses the OutputParser, if provided, to further refine and format the result into its ultimate usable form. To illustrate the functionality of LLM chains, consider the concrete example.A concrete example illustrating the functionality of LLM chains is detailed below:

* It works by taking a user’s input and passing it to the first element in the chain — a PromptTemplate — to format the input into a particular prompt.
* The formatted prompt is then passed to the next (and final) element in the chain — a LLM.

**Crafting Chains – LLM Chain**

Creating Chains, especially LLM Chains, is a meticulous endeavor, requiring the harnessing of Large Language Models in LangChain. These chains serve as intricate channels, facilitating the smooth exchange of information and engagement. Through careful structuring, developers can design vibrant applications capable of understanding user inputs, utilizing LLMs to generate intelligent responses, and customizing the output to meet specific needs effectively.

Now let’s look deeper into how we can use the LLM Chains in the Langchain.

Import Necessary Libraries

import langchain

import openai

from langchain.llms import OpenAI

from langchain.prompts import PromptTemplate

from getpass import getpass

OPENAI\_API\_KEY = getpass()

Initialize LLM and Prompt Template

We initialize the OpenAI Large Language Model with specific parameters, including a temperature of 0.9, which affects the diversity of generated responses. Furthermore, users must define a ‘PromptTemplate’ to input a variable (in this case, “product”) and create a standardized prompt structure. At runtime, the placeholder ‘{product}’ can be dynamically populated with different product names.

llm = OpenAI(temperature=0.9,

openai\_api\_key=OPENAI\_API\_KEY

)

prompt = PromptTemplate(

input\_variables=["product"],

template="What is a good name for a company that makes {product}?",

)

**Creating A Chain**

We create an instance of the ‘LLMChain’ class, using a predefined OpenAI Large Language Model and a specified prompt template. Now, we have the capability to apply the chain to a product such as a “gaming laptop” using the chain.run command. This means the chain can dynamically process and generate responses tailored to this specific product input.

from langchain.chains import LLMChain

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

print(chain.run("gaming laptop"))

**Output:**

Based on this we get the name of a company called “GamerTech Laptops”.

**Sequential Chain**

A sequential chain is a chain that combines various individual chains, where the output of one chain serves as the input for the next in a continuous sequence. It operates by running a series of chains consecutively.

There are two types of sequential chains:

**Simple Sequential Chain**, which handles a single input and output, and

**Sequential Chain**, manage multiple inputs and outputs simultaneously.

* A sequential chain merges various chains by using the output of one chain as the input for the next.
* It operates by executing a series of chains consecutively.
* This approach is valuable when you need to utilize the result of one operation as the starting point for the next one, creating a seamless flow of processes.

**Simple Sequential Chain**

Sequential chains, in their simplest form, consist of steps where each step takes one input and produces one output. The output from one step becomes the input for the next.

This straightforward approach is effective when dealing with sub-chains designed for singular inputs and outputs. It ensures a smooth and continuous flow of information, with each step seamlessly passing its output to the subsequent step.

**Crafting Chains – Simple Sequential Chains**

Simple Sequential Chains allow for a single input to undergo a series of coherent transformations, resulting in a refined output. This sequential approach ensures systematic and efficient handling of data, making it ideal for scenarios where a linear flow of information processing is essential

Importing Necessary Libraries

from langchain.llms import OpenAI

from langchain.chains import LLMChain

from langchain.prompts import PromptTemplate

from langchain.prompts import ChatPromptTemplate

from langchain.chains import SimpleSequentialChain

**Initializing and Chaining**

We initialize an OpenAI Large Language Model with a temperature setting of 0.7 and an API key. Then, we create a specific chat prompt template with a placeholder for a product name. Subsequently, we form an LLMChain, which allows the generation of responses based on the provided prompt. We repeat this process for two different chains.

# This is an LLMChain to write first chain.

llm = OpenAI(temperature=0.7, openai\_api\_key=OPENAI\_API\_KEY)

first\_prompt = ChatPromptTemplate.from\_template(

"What is the best name to describe a company that makes {product}?"

)

chain\_one = LLMChain(llm=llm, prompt=first\_prompt)

# This is an LLMChain to write second chain.

llm = OpenAI(temperature=0.7, openai\_api\_key=OPENAI\_API\_KEY)

second\_prompt = ChatPromptTemplate.from\_template(

"Write a 20 words description for the following company:{company\_name}"

)

chain\_two = LLMChain(llm=llm, prompt=second\_prompt)

Chaining Two Chains

Create an overall Simple Sequential Chain, comprising two different individual chains, chain\_one, and chain\_two. Execute this with the input “gaming laptop,” it sequentially processes the input through the defined chains and provides an output, that demonstrates the step-by-step sequential execution of the chains.

overall\_simple\_chain = SimpleSequentialChain(chains=[chain\_one, chain\_two],

verbose=True

)

overall\_simple\_chain.run("gaming laptop")

**Output:**

**Sequential Chain**

Not all of the sequential chains operate with a single string input and output. In more intricate setups, these chains handle multiple inputs and generate multiple final outputs. The careful naming of input and output variables holds important significance in these complex chains.

A more general form of sequential chains allows for multiple inputs/outputs. Any step in the chain can take in multiple inputs.

**Crafting Chains – Sequential Chains**

Importing Necessary Libraries

from langchain.llms import OpenAI

from langchain.chains import LLMChain

from langchain.prompts import PromptTemplate

from langchain.prompts import ChatPromptTemplate

from langchain.chains import SequentialChain

llm = OpenAI(temperature=0.7, openai\_api\_key=OPENAI\_API\_KEY)

**Initializing and Chaining**

We define a prompt template, instructing the system to perform a specific task. We then create a corresponding LLMChain, using the designated Large Language Model (LLM) and the defined prompt template. The chain is set up to take the input, pass it through the LLM, and generate the output. We repeat this process to establish four distinct chains.

Review = "Les ordinateurs portables GamersTech impressionne par ses

performances exceptionnelles et son design élégant. De sa configuration

matérielle robuste à un clavier RVB personnalisable et un système de

refroidissement efficace, il établit un équilibre parfait entre prouesses

de jeu et portabilité."

# prompt template 1: translate to English

first\_prompt = ChatPromptTemplate.from\_template(

"Translate the following review to english:"

"\n\n{Review}"

)

# chain 1: input= Review and output= English\_Review

chain\_one = LLMChain(llm=llm, prompt=first\_prompt,

output\_key="English\_Review"

)

# prompt template 2: Summarize the English review

second\_prompt = ChatPromptTemplate.from\_template(

"Can you summarize the following review in 1 sentence:"

"\n\n{English\_Review}"

)

# chain 2: input= English\_Review and output= summary

chain\_two = LLMChain(llm=llm, prompt=second\_prompt,

output\_key="summary"

)

# prompt template 3: translate to English

third\_prompt = ChatPromptTemplate.from\_template(

"What language is the following review:\n\n{Review}"

)

# chain 3: input= Review and output= language

chain\_three = LLMChain(llm=llm, prompt=third\_prompt,

output\_key="language"

)

# prompt template 4: follow up message

fourth\_prompt = ChatPromptTemplate.from\_template(

"Write a follow up response to the following "

"summary in the specified language:"

"\n\nSummary: {summary}\n\nLanguage: {language}"

)

# chain 4: input= summary, language and output= followup\_message

chain\_four = LLMChain(llm=llm, prompt=fourth\_prompt,

output\_key="followup\_message"

)

**Chaining Two Chains**

An overall Sequential Chain named ‘overall\_chain’ is created, incorporating four individual chains ‘chain\_one’, ‘chain\_two’, ‘chain\_three’, and ‘chain\_four’. The input variable “Review” is processed through these chains, generating three distinct output variables: “English\_Review,” “summary,” and “followup\_message.” The ‘overall\_chain’ executes the input review through the specified chains and produces these outputs, facilitating a structured, sequential processing flow with detailed outputs.

overall\_chain = SequentialChain(

chains=[chain\_one, chain\_two, chain\_three, chain\_four],

input\_variables=["Review"],

output\_variables=["English\_Review", "summary","followup\_message"],

verbose=True

)

overall\_chain(Review)

**Output**:

**Router Chain**

The Router Chain is used for complicated tasks. If we have multiple subchains, each of which is specialized for a particular type of input, we could have a router chain that decides which subchain to pass the input to.

It consists of:

* **Router Chain:** It is responsible for selecting the next chain to call.
* **Destination Chains:** Chains that the router chain can route to.
* **Default chain:** Used when the router can’t decide which subchain to use.

This involves directing an input toward a specific chain based on what exactly that input is. When there are several subchains, each tailored for distinct input types, a router chain comes into play. This router chain acts as a decision-maker, determining which specialized subchain to send the input to. Essentially, it enables the seamless routing of inputs to the appropriate subchains, ensuring efficient and precise processing based on the input’s specific characteristics.

**Crafting Chains – Router Chain**

Importing Necessary Libraries

from langchain.chains.router import MultiPromptChain

from langchain.chains.router.llm\_router import LLMRouterChain,RouterOutputParser

from langchain.prompts import PromptTemplate

llm = OpenAI(temperature=0.7, openai\_api\_key=OPENAI\_API\_KEY)

**Defining Prompt Templates**

Let’s consider a scenario where we need to direct inputs to specialized chains based on subjects such as Maths, Physics, History, or Computer Science. To accomplish this, we create distinct prompts for each subject: one for physics questions, another for math queries, a third for history inquiries, and a fourth for computer science-related matters. We meticulously design these prompts to cater to the unique needs of each subject area.

physics\_template = """You are a very smart physics professor. \

You are great at answering questions about physics in a concise\

and easy to understand manner. \

When you don't know the answer to a question you admit\

that you don't know.

Here is a question:

{input}"""

math\_template = """You are a very good mathematician. \

You are great at answering math questions. \

You are so good because you are able to break down \

hard problems into their component parts,

answer the component parts, and then put them together\

to answer the broader question.

Here is a question:

{input}"""

history\_template = """You are a very good historian. \

You have an excellent knowledge of and understanding of people,\

events and contexts from a range of historical periods. \

You have the ability to think, reflect, debate, discuss and \

evaluate the past. You have a respect for historical evidence\

and the ability to make use of it to support your explanations \

and judgements.

Here is a question:

{input}"""

Furthermore, detailed information, including names and descriptions, can be attached to these prompt templates. This additional context provides a comprehensive understanding of each template’s purpose. This information is then supplied to the router chain. The router chain then determines which subchain to route to based on the specific subject, ensuring that the appropriate prompt template is utilized for precise and effective responses.

# Defining the prompt templates

prompt\_infos = [

{

"name": "physics",

"description": "Good for answering questions about physics",

"prompt\_template": physics\_template

},

{

"name": "math",

"description": "Good for answering math questions",

"prompt\_template": math\_template

},

{

"name": "History",

"description": "Good for answering history questions",

"prompt\_template": history\_template

}

]

**Creating Destination Chains**

Next, our focus shifts to crafting destination chains. These chains are activated by the RouterChain, functioning as individual language model chains, specifically LLM chains. Additionally, a default chain is outlined to handle situations where the router encounters ambiguity and cannot determine the suitable subchain to utilize. This default chain acts as a fallback option, ensuring a response even in cases of indecision.

destination\_chains = {}

for p\_info in prompt\_infos:

name = p\_info["name"]

prompt\_template = p\_info["prompt\_template"]

prompt = ChatPromptTemplate.from\_template(template=prompt\_template)

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

destination\_chains[name] = chain

destinations = [f"{p['name']}: {p['description']}" for p in prompt\_infos]

destinations\_str = "\n".join(destinations)

**Creating a Multi-prompt Router Template**

We establish a template guiding the LLM in directing interactions between various chains. This template not only outlines the specific task instructions but also dictates the precise format that the output should adhere to, ensuring a standardized and consistent response mechanism.

MULTI\_PROMPT\_ROUTER\_TEMPLATE = """Given a raw text input to a \

language model select the model prompt best suited for the input. \

You will be given the names of the available prompts and a \

description of what the prompt is best suited for. \

You may also revise the original input if you think that revising\

it will ultimately lead to a better response from the language model.

<< FORMATTING >>

Return a markdown code snippet with a JSON object formatted to look like:

```json

{{{{

"destination": string \ name of the prompt to use or "DEFAULT"

"next\_inputs": string \ a potentially modified version of the original input

}}}}

```

REMEMBER: "destination" MUST be one of the candidate prompt \

names specified below OR it can be "DEFAULT" if the input is not\

well suited for any of the candidate prompts.

REMEMBER: "next\_inputs" can just be the original input \

if you don't think any modifications are needed.

<< CANDIDATE PROMPTS >>

{destinations}

<< INPUT >>

{{input}}

<< OUTPUT (remember to include the ```json)>>"""

Creating a Default Chain

A preset prompt template is established to accommodate all types of input text. An associated LLMChain, named ‘default\_chain,’ is then crafted using the designated Large Language Model and the predefined prompt. This setup enables the Large Language Model to generate responses based on any provided input text.

default\_prompt = ChatPromptTemplate.from\_template("{input}")

default\_chain = LLMChain(llm=llm, prompt=default\_prompt)

**Creating Router Template**

Moving forward, a flexible router template is developed, encompassing a range of categories such as Physics, Math, History, and Computer Science. From this template, a distinct prompt template tailored for the router is created. Utilizing this customized template, a router chain is established, employing the Large Language Model and the corresponding router prompt.

To improve decision-making capabilities, a router output parser is introduced. This parser assists the router chain in efficiently navigating between subchains. This comprehensive arrangement ensures that inputs are directed precisely to specific subchains, leading to accurate and targeted responses across various destination categories.

router\_template = MULTI\_PROMPT\_ROUTER\_TEMPLATE.format(

destinations=destinations\_str

)

router\_prompt = PromptTemplate(

template=router\_template,

input\_variables=["input"],

output\_parser=RouterOutputParser(),

)

router\_chain = LLMRouterChain.from\_llm(llm, router\_prompt)

Chaining Everything Together

A MultiPromptChain is created, incorporating a router chain to intelligently route inputs to specific destination chains. Additionally, a default chain is included to handle cases where the router chain may encounter ambiguity, ensuring a structured and effective processing flow with verbose logging enabled for detailed insights.

chain = MultiPromptChain(router\_chain=router\_chain,

destination\_chains=destination\_chains,

default\_chain=default\_chain, verbose=True

)

**Output**

Real-world Use Cases of Langchain

Delve into the real-world uses and achievements of solutions driven by Large Language Models (LLMs), demonstrating their varied influence across sectors. Within customer support, the collaboration between LangChain and LLMs has transformed services through the implementation of smart chatbots. These bots provide immediate, personalized support, efficiently managing a large influx of queries. By reducing wait times, they significantly elevate customer satisfaction levels.

E-commerce

LangChain utilizes the power of Large Language Models (LLMs) to enhance the shopping journey. Developers can create applications that understand product specifics, user likes, and purchasing patterns. By harnessing the capabilities of LLMs, these platforms offer tailored product recommendations, address customer inquiries, and create captivating product descriptions. This leads to increased sales and higher customer engagement levels.

Healthcare

LangChain is revolutionizing patient care and diagnosis through applications powered by Large Language Models (LLMs). With LangChain’s support, develop virtual assistants to understand medical inquiries. These virtual assistants provide accurate information, assess patients based on symptoms, and expedite access to healthcare knowledge. This advancement not only lightens the workload for medical professionals but also enables patients to make well-informed decisions about their health.

Content Generation

LangChain empowers developers to create applications that produce imaginative and contextually relevant content, including blog articles and product descriptions. These applications support content creators by enhancing creativity, streamlining the writing process, and maintaining consistency in tone and style.

The practical implementations highlighted here demonstrate the versatility and impact of solutions driven by Large Language Models (LLMs) across various industries. LangChain’s potential enables developers to create innovative solutions, streamline operations, improve user engagement, and fuel business growth. Success stories abound, ranging from significant decreases in support ticket resolution times to higher customer satisfaction ratings for e-commerce chatbots, showcasing the tangible benefits of LLM-powered applications.

**Chains**

* **Basic**: LLMChain (prompt → model).
* **Retrieval**: RetrievalQA, ConversationalRetrievalChain.
* **Doc chains**: map-reduce / refine / stuff.
* **LCEL (Runnable graph)**: RunnableSequence, branching, parallel, retries, fallbacks, streaming.

**let’s dive into Chains in LangChain.**  
Chains are essentially **pipelines** that take inputs → run them through one or more processing steps → output results.  
The beauty is that each step can be an LLM, a retriever, a function call, or even another chain.

**1️⃣ Basic Chains — LLMChain**

**Purpose:**  
The simplest possible chain — take a prompt template, fill it with input variables, send it to an LLM, return the output.

**Structure:**

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

from langchain\_openai import OpenAI

prompt = PromptTemplate.from\_template(

"Translate the following text to French:\n{text}"

)

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

result = chain.run("I love learning AI.")

print(result)

**Pipeline:**

css

CopyEdit

[text] → [PromptTemplate] → [LLM] → [Output]

**2️⃣ Retrieval Chains**

These chains integrate a **retriever** into the process.

**RetrievalQA**

**Flow:**

sql

CopyEdit

User question → Retriever fetches docs → Stuff docs into prompt → LLM answers

**Example:**

python

CopyEdit

from langchain.chains import RetrievalQA

retrieval\_chain = RetrievalQA.from\_chain\_type(

llm=OpenAI(),

retriever=my\_vectorstore.as\_retriever(search\_kwargs={"k": 3})

)

response = retrieval\_chain.run("What is the warranty period?")

**ConversationalRetrievalChain**

Adds **chat history** to retrieval so the user can ask follow-ups without repeating context.  
**Flow:**

bash

CopyEdit

[Question + Chat history] → Retriever → LLM

**Example:**

python

CopyEdit

from langchain.chains import ConversationalRetrievalChain

qa = ConversationalRetrievalChain.from\_llm(

llm=OpenAI(),

retriever=my\_vectorstore.as\_retriever()

)

result = qa({"question": "Who is the CEO?", "chat\_history": []})

**3️⃣ Document Chains**

When retrieved docs are too long to fit into the LLM’s context, you need a **processing strategy**:

**A. Stuff**

* **Simplest:** Put all docs into the prompt and send to LLM.
* **Good for:** Small doc sets.

python

CopyEdit

from langchain.chains.combine\_documents import StuffDocumentsChain

**B. Map-Reduce**

* **Map:** Process each doc separately with LLM.
* **Reduce:** Combine results into one answer.
* **Good for:** Many docs, parallelizable.

python

CopyEdit

from langchain.chains import MapReduceDocumentsChain

**C. Refine**

* Start with one doc → answer partially.
* Iteratively improve the answer by feeding next doc + previous answer.
* **Good for:** Sequential refinement.

**4️⃣ LCEL — LangChain Expression Language**

This is the **newer, composable way** to define chains.  
It treats chains as **Runnable graphs** — building blocks that can be sequenced, branched, run in parallel, retried, and streamed.

**RunnableSequence**

Sequential chaining:

python

CopyEdit

from langchain\_core.runnables import RunnableSequence

chain = RunnableSequence(first\_step, second\_step, third\_step)

**Branching**

Run different logic depending on input:

python

CopyEdit

from langchain\_core.runnables import RunnableBranch

**Parallel**

Run multiple steps at once and combine:

python

CopyEdit

from langchain\_core.runnables import RunnableParallel

**Retries**

python

CopyEdit

from langchain\_core.runnables import RunnableRetry

**Fallbacks**

If one chain fails, try another:

python

CopyEdit

from langchain\_core.runnables import RunnableWithFallbacks

**Streaming**

Partial outputs as they are generated:

python

CopyEdit

chain.stream({"input": "Tell me a story"})

**📌 Summary Table**

| **Type** | **Purpose** | **Example Use** |
| --- | --- | --- |
| **LLMChain** | Simple prompt → LLM | Translation |
| **RetrievalQA** | Search + answer | QA over docs |
| **ConversationalRetrievalChain** | Search + history | Chatbots |
| **Stuff** | Direct doc stuffing | Small doc sets |
| **Map-Reduce** | Parallel processing | Summarizing many docs |
| **Refine** | Iterative improvement | Complex doc synthesis |
| **LCEL** | Flexible pipelines | Custom multi-step workflows |

If you want, next I can go **deep dive** into LCEL (RunnableSequence, RunnableParallel, RunnableBranch, retries, fallbacks) and show how to build a **multi-path retrieval + summarization chain** in one graph.

**1️⃣ What is LLMChain?**

LLMChain is the simplest LangChain chain:

* **Takes a prompt template** (with placeholders)
* **Fills it with input values**
* **Sends it to an LLM**
* **Returns the output**

Think of it like:

nginx

Inputs → PromptTemplate → LLM → Output

No retrievers, no document processing — just structured prompt + model call.

**2️⃣ The Core Components of an LLMChain**

**A. PromptTemplate**

* A prompt template contains:
  + **input\_variables** → placeholders you must supply.
  + **template** → text structure given to the LLM.

Example:

python

CopyEdit

from langchain.prompts import PromptTemplate

prompt = PromptTemplate(

input\_variables=["language", "text"],

template="Translate the following text to {language}:\n{text}"

)

When you call:

python

CopyEdit

prompt.format(language="French", text="I love AI.")

You get:

vbnet

CopyEdit

Translate the following text to French:

I love AI.

**B. The LLM**

* Any LangChain-compatible LLM works: OpenAI, ChatOpenAI, HuggingFaceHub, Ollama, etc.

python

CopyEdit

from langchain\_openai import OpenAI

llm = OpenAI(model="gpt-3.5-turbo-instruct", temperature=0.2)

**C. The Chain**

* You combine the prompt + LLM into LLMChain.

python

CopyEdit

from langchain.chains import LLMChain

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

**3️⃣ Running an LLMChain**

You can run it in **three main ways**:

**A. .run() → Single string output**

python

CopyEdit

result = chain.run(language="French", text="I love AI.")

print(result)

**B. .invoke() → Structured output**

python

CopyEdit

result = chain.invoke({"language": "French", "text": "I love AI."})

print(result) # {'text': 'J\'aime l\'IA.'}

**C. .stream() → Token-by-token streaming**

python

CopyEdit

for chunk in chain.stream({"language": "French", "text": "I love AI."}):

print(chunk, end="", flush=True)

**4️⃣ Internal Flow**

When you call chain.run():

1. **Prompt Formatting**
   * Replaces placeholders with your variables.
2. **LLM Call**
   * Passes the final string to the llm object.
3. **Output Parsing**
   * If no parser is given → returns raw text.
   * If parser is given (e.g., StrOutputParser, PydanticOutputParser) → parses before returning.

**5️⃣ Enhancing LLMChain**

LLMChain may be basic, but you can layer in:

* **Output Parsers** (structured responses)

python

CopyEdit

from langchain.output\_parsers import CommaSeparatedListOutputParser

parser = CommaSeparatedListOutputParser()

prompt = PromptTemplate(

input\_variables=["topic"],

template="List 5 words related to {topic} separated by commas."

)

chain = LLMChain(prompt=prompt, llm=llm, output\_parser=parser)

print(chain.run(topic="ocean")) # ['wave', 'sea', ...]

* **Memory** (for conversational context)

python

CopyEdit

from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()

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

* **Callbacks** (log, monitor, debug)

python

CopyEdit

from langchain.callbacks import StdOutCallbackHandler

chain.run("Some input", callbacks=[StdOutCallbackHandler()])

**6️⃣ Common Pitfalls**

| **Pitfall** | **Fix** |
| --- | --- |
| **Mismatched variable names** | Ensure input\_variables match exactly with .run() kwargs. |
| **Trying to process long docs** | Use a text splitter before passing to LLMChain. |
| **Forgetting temperature** | Set temperature=0 for deterministic outputs. |
| **Slow outputs** | Use .stream() for incremental tokens. |

**7️⃣ Example: Using LLMChain as a Building Block**

Later, you can embed an LLMChain inside:

* **RetrievalQA** → as the “answering step”.
* **LCEL RunnableSequence** → as a node in a graph.
* **Multi-step workflows** → multiple LLMChains feeding into each other.

Example:

from langchain.prompts import PromptTemplate

from langchain\_openai import OpenAI

from langchain.chains import LLMChain

prompt = PromptTemplate(

input\_variables=["product"],

template="Write a funny product description for {product}."

)

llm = OpenAI(temperature=0.8)

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

print(chain.run(product="smart refrigerator"))

**🔹 Part 1 — Using Structured Output + Custom Tool Together**

Imagine we have:

* A **schema** (structured output) for weather response.
* A **custom tool** for doing math (CalculatorTool).

**Code Example (LangChain 0.3+ style)**

from langchain\_openai import ChatOpenAI

from langchain\_core.pydantic\_v1 import BaseModel, Field

from langchain\_core.tools import tool

# 1. Define schema for structured output

class WeatherResponse(BaseModel):

location: str = Field(..., description="The city/location for the weather")

temperature: float = Field(..., description="Temperature in Celsius")

condition: str = Field(..., description="Weather condition like sunny, rainy")

# 2. Define a custom tool

@tool

def calculator(expression: str) -> str:

"""Evaluate a math expression and return the result."""

try:

return str(eval(expression))

except Exception as e:

return f"Error: {e}"

# 3. Create base model

model = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# 4. Wrap model with structured output (forces schema compliance)

model\_structured = model.with\_structured\_output(WeatherResponse)

# 5. Bind additional tool (calculator) along with schema tool

model\_with\_tools = model\_structured.bind\_tools([calculator])

# 6. Use it

resp = model\_with\_tools.invoke("What's the weather in Delhi and also calculate 25 \* 4?")

print(resp)

👉 Here’s what happens internally:

1. with\_structured\_output(WeatherResponse) wraps the schema as an **implicit tool**.
2. bind\_tools([calculator]) attaches your **explicit tool**.
3. The model decides whether to:
   * Return a WeatherResponse.
   * Call calculator.
   * Or both in sequence (depending on your orchestration).

So — structured output = “always respond in schema form,” but you *can* still let it call other tools.

**🔹 Part 2 — Retrieval Chains (RetrievalQA)**

Now, onto your next curiosity 🙂

**RetrievalQA = LLMChain + Retriever**

* **Retriever**: fetches documents from a store (e.g., FAISS, Pinecone, Chroma).
* **LLMChain**: runs the question + retrieved docs through an LLM to produce the answer.

**Flow:**

1. User asks a question.
2. Retriever fetches top-k docs.
3. Retrieved docs get **injected into the LLM prompt**.
4. LLMChain generates the final answer.

**Code Example (LangChain 0.3+)**

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain\_community.vectorstores import FAISS

from langchain.chains import RetrievalQA

# 1. Prepare embeddings + vector store

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

docs = ["Delhi is the capital of India", "Mumbai is the financial hub", "Bangalore is known for IT"]

vectorstore = FAISS.from\_texts(docs, embeddings)

# 2. Create retriever

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 2})

# 3. Base LLM

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# 4. Wrap into RetrievalQA

qa = RetrievalQA.from\_chain\_type(llm=llm, retriever=retriever)

# 5. Ask a question

result = qa.invoke("What is the capital of India?")

print(result)

**How document injection works internally**

Prompt looks like:

Question: What is the capital of India?

Context:

- Delhi is the capital of India

- Mumbai is the financial hub

Answer:

LLM then generates the answer using both the **question** and **retrieved docs**.

⚖️ So in summary:

* **Structured output** = schema validation layer.
* **Custom tools** = external capabilities (math, APIs).
* **RetrievalQA** = automatic “fetch docs + inject into prompt + run LLM.”

**1. What a Retrieval Chain Does**

At its core:

1. **Accepts a user query**
2. **Retrieves relevant documents** from a store (vector DB, search index, API, etc.)
3. **Formats those docs** into a prompt
4. **Sends the prompt to the LLM**
5. **Returns an answer** — optionally with source references

**2. Common Retrieval Chains in LangChain**

LangChain provides specialized chain classes for retrieval:

| **Chain** | **Purpose** | **Key Use Cases** |
| --- | --- | --- |
| **RetrievalQA** | Simple Q&A over retrieved docs | FAQ bots, document search |
| **ConversationalRetrievalChain** | Maintains chat history while retrieving | Chatbots that remember context |
| **VectorDBQA** (older) | Same as RetrievalQA but tied to vector DB directly | Deprecated in favor of RetrievalQA |
| **MultiVectorRetriever** | Retrieves from multiple embeddings per doc | Chunked docs with summaries |

**3. Retrieval Chain Components**

Let’s break them down.

**a. Retriever**

* A retriever is any object with .get\_relevant\_documents(query)
* Examples:
  + vectorstore.as\_retriever()
  + BM25Retriever (keyword-based)
  + MultiVectorRetriever

**Vector Store Retriever Flow:**

text

CopyEdit

Query → embed → similarity search in vector DB → return top-k documents

**b. Document Combination**

Retrieval chains can use **doc\_chain\_type** to decide how multiple docs are combined before sending to LLM:

1. **stuff** — All docs concatenated into one prompt (fast, may hit token limit)
2. **map\_reduce** — Process each doc separately, then merge answers
3. **refine** — Answer from first doc, then refine using others

**4. RetrievalQA Example**

Here’s a basic **RetrievalQA** chain:

python

CopyEdit

from fastapi import FastAPI

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain\_community.vectorstores import FAISS

from langchain.chains import RetrievalQA

from langchain\_community.document\_loaders import PyPDFLoader

app = FastAPI()

# Step 1: Load and embed documents

loader = PyPDFLoader("sample.pdf")

docs = loader.load()

embeddings = OpenAIEmbeddings()

vectorstore = FAISS.from\_documents(docs, embeddings)

# Step 2: Create retriever

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3})

# Step 3: Create the LLM

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# Step 4: Build the RetrievalQA chain

qa\_chain = RetrievalQA.from\_chain\_type(

llm=llm,

chain\_type="stuff",

retriever=retriever,

return\_source\_documents=True

)

@app.get("/ask")

async def ask\_question(q: str):

result = qa\_chain.invoke(q)

return {

"answer": result["result"],

"sources": [doc.metadata for doc in result["source\_documents"]]

}

**5. ConversationalRetrievalChain**

When you need **multi-turn conversations** that remember context:

python

CopyEdit

from langchain.chains import ConversationalRetrievalChain

from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(

memory\_key="chat\_history",

return\_messages=True

)

conv\_chain = ConversationalRetrievalChain.from\_llm(

llm=llm,

retriever=retriever,

memory=memory

)

Flow:

sql

CopyEdit

User Q + Chat History → Retriever fetches relevant docs → LLM answers with context

**6. Advanced Retrieval Features**

* **Metadata filtering**: retrieve docs based on tags, dates, categories.
* **Hybrid search**: combine semantic similarity + keyword search.
* **Self-query retriever**: LLM rewrites user query into vector + metadata filters.
* **Re-ranking**: retrieve more docs, then re-rank with a better scoring model.

**7. When to Use Which**

| **Situation** | **Chain** |
| --- | --- |
| One-off question on static docs | RetrievalQA |
| Chat-style Q&A with memory | ConversationalRetrievalChain |
| Large docs with summaries | MultiVectorRetriever |
| Need filtering and hybrid search | SelfQueryRetriever + RetrievalQA |

If you want, I can **extend this into a full FastAPI RetrievalQA + ConversationalRetrievalChain setup** where both endpoints run on the same vector store and share embeddings so you can switch between static and chat modes.

That way, you’ll have **one stack** covering both Q&A and conversations.

**You said:**

yes please - If you want, I can extend this into a full FastAPI RetrievalQA + ConversationalRetrievalChain setup where both endpoints run on the same vector store and share embeddings so you can switch between static and chat modes.

**ChatGPT said:**

Perfect — let’s build a **full FastAPI app** with both:

1. **RetrievalQA** endpoint — one-off question answering
2. **ConversationalRetrievalChain** endpoint — chat with context
3. **Shared vector store** — same embeddings used for both

**1️⃣ Install dependencies**

bash

CopyEdit

pip install fastapi uvicorn langchain langchain-openai langchain-community faiss-cpu pydantic

**2️⃣ Project Structure**

bash

CopyEdit

retrieval\_app/

│

├── main.py # FastAPI app

├── prompt.py # Optional: custom prompt templates

└── sample.pdf # Sample document to load

**3️⃣ main.py — Full Setup**

python

CopyEdit

from fastapi import FastAPI

from pydantic import BaseModel

from langchain.chains import RetrievalQA, ConversationalRetrievalChain

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain\_community.vectorstores import FAISS

from langchain\_community.document\_loaders import PyPDFLoader

from langchain.memory import ConversationBufferMemory

import os

# Set your OpenAI API key

os.environ["OPENAI\_API\_KEY"] = "YOUR\_OPENAI\_API\_KEY"

app = FastAPI(title="RetrievalQA + Conversational Retrieval API")

# -------------------------

# Step 1: Load documents

# -------------------------

loader = PyPDFLoader("sample.pdf")

docs = loader.load()

# -------------------------

# Step 2: Create embeddings + vectorstore

# -------------------------

embeddings = OpenAIEmbeddings()

vectorstore = FAISS.from\_documents(docs, embeddings)

# -------------------------

# Step 3: Create retriever

# -------------------------

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3})

# -------------------------

# Step 4: LLM

# -------------------------

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# -------------------------

# Step 5: RetrievalQA chain (one-off Q&A)

# -------------------------

qa\_chain = RetrievalQA.from\_chain\_type(

llm=llm,

chain\_type="stuff", # can use 'map\_reduce' for large doc sets

retriever=retriever,

return\_source\_documents=True

)

# -------------------------

# Step 6: ConversationalRetrievalChain (multi-turn chat)

# -------------------------

memory = ConversationBufferMemory(

memory\_key="chat\_history",

return\_messages=True

)

conv\_chain = ConversationalRetrievalChain.from\_llm(

llm=llm,

retriever=retriever,

memory=memory

)

# -------------------------

# API Models

# -------------------------

class QuestionRequest(BaseModel):

question: str

class ChatRequest(BaseModel):

message: str

# -------------------------

# Step 7: API Endpoints

# -------------------------

@app.post("/ask")

async def ask\_question(req: QuestionRequest):

"""One-off Q&A over documents"""

result = qa\_chain.invoke(req.question)

return {

"question": req.question,

"answer": result["result"],

"sources": [doc.metadata for doc in result["source\_documents"]]

}

@app.post("/chat")

async def chat(req: ChatRequest):

"""Conversational chat with context"""

result = conv\_chain({"question": req.message})

# Access memory if needed: conv\_chain.memory.chat\_memory.messages

return {

"message": req.message,

"answer": result["answer"]

}

**4️⃣ Run the server**

bash

CopyEdit

uvicorn main:app --reload

**5️⃣ Test the endpoints**

**A. RetrievalQA**

bash

CopyEdit

curl -X POST "http://127.0.0.1:8000/ask" \

-H "Content-Type: application/json" \

-d '{"question": "What is the warranty period of the product?"}'

**Response Example:**

json

CopyEdit

{

"question": "What is the warranty period of the product?",

"answer": "The product comes with a 2-year warranty covering manufacturing defects.",

"sources": [{"source": "sample.pdf", "page": 3}]

}

**B. Conversational Chat**

bash

CopyEdit

curl -X POST "http://127.0.0.1:8000/chat" \

-H "Content-Type: application/json" \

-d '{"message": "Who is the CEO?"}'

**Response Example:**

json

CopyEdit

{

"message": "Who is the CEO?",

"answer": "The CEO of the company is Jane Doe."

}

* Follow-up messages automatically include previous chat context.
* Both endpoints **share the same vector store** and embeddings — no duplication needed.

**✅ Key Points**

1. **Single vector store** → shared across QA & chat.
2. **Different chain types**:
   * RetrievalQA → one-off, stateless
   * ConversationalRetrievalChain → maintains memory
3. **Document retrieval** is separate from LLM → can swap retrievers (FAISS → Chroma → Pinecone).
4. **Prompt templates** can be customized in either chain.

If you want, I can **upgrade this next** to include:

* **metadata filtering** (e.g., only retrieve docs tagged manual)
* **multi-step chains** with map\_reduce or refine strategies
* **streaming responses** over WebSocket for real-time answers

Perfect! Let’s deep dive into **Document Chains** — this is where LangChain really shines for **processing multiple documents** before sending to an LLM.

**1️⃣ What Are Document Chains?**

A **Document Chain** is a chain designed to **combine multiple documents** into a single output.  
Use cases:

* Summarization of long documents
* Answering a question from multiple sources
* Generating reports from several text chunks

**Key Problem**

LLMs have a **context window limit**. If your combined documents exceed that limit, you can’t stuff them all into a single prompt.

**Document chains provide strategies** to handle this:

1. **Stuff** → simplest, put all documents into the prompt
2. **Map-Reduce** → process docs separately, then combine answers
3. **Refine** → generate partial answer and iteratively refine using next docs

**2️⃣ Types of Document Chains**

**A. StuffDocumentsChain**

* **How it works:**  
  Concatenates all documents into a single prompt.  
  Good when:
  + Document set is small
  + Tokens don’t exceed LLM context
* **Example:**

python

CopyEdit

from langchain.chains.combine\_documents import StuffDocumentsChain

from langchain\_openai import ChatOpenAI

from langchain.prompts import PromptTemplate

prompt = PromptTemplate(

template="Summarize the following documents:\n{documents}",

input\_variables=["documents"]

)

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

stuff\_chain = StuffDocumentsChain(llm\_chain=LLMChain(llm=llm, prompt=prompt))

**B. MapReduceDocumentsChain**

* **How it works:**
  1. **Map step:** LLM processes each document separately → produces intermediate summaries
  2. **Reduce step:** LLM combines intermediate results into a final summary
* **Benefits:**
  1. Handles large document sets
  2. Can parallelize map step
* **Example:**

python

CopyEdit

from langchain.chains.combine\_documents import MapReduceDocumentsChain

map\_reduce\_chain = MapReduceDocumentsChain(

llm\_chain=LLMChain(llm=llm, prompt=prompt),

# optional reduce chain prompt can be different

)

**Flow diagram:**

css

CopyEdit

[Doc1, Doc2, Doc3] → [LLM map] → [Intermediate summaries] → [LLM reduce] → Final summary

**C. RefineDocumentsChain**

* **How it works:**
  1. Start with first document → generate partial answer
  2. Iteratively pass next document + current answer → LLM refines answer
* **Benefits:**
  1. Useful when order matters
  2. Can incorporate feedback or corrections
* **Example:**

python

CopyEdit

from langchain.chains.combine\_documents import RefineDocumentsChain

refine\_chain = RefineDocumentsChain(

llm\_chain=LLMChain(llm=llm, prompt=prompt),

refine\_llm\_chain=LLMChain(llm=llm, prompt=PromptTemplate(

template="Refine the answer given the new document:\nCurrent answer: {existing\_answer}\nNew document: {text}",

input\_variables=["existing\_answer","text"]

))

)

**Flow diagram:**

csharp

CopyEdit

Doc1 → LLM → partial answer

Doc2 + partial answer → LLM → refined answer

Doc3 + refined answer → LLM → final answer

**3️⃣ When to Use Which Strategy**

| **Strategy** | **Pros** | **Cons** |
| --- | --- | --- |
| **Stuff** | Simple, fast | Hits token limits easily |
| **Map-Reduce** | Handles large docs, parallelizable | Slightly slower, more calls |
| **Refine** | Maintains context sequentially | Slower, sequential processing |

**4️⃣ Example: Summarizing Multiple PDF Pages**

python

CopyEdit

from langchain\_community.document\_loaders import PyPDFLoader

from langchain.chains import MapReduceDocumentsChain

from langchain.chains import LLMChain

from langchain\_openai import ChatOpenAI

from langchain.prompts import PromptTemplate

# Load PDF

loader = PyPDFLoader("manual.pdf")

docs = loader.load() # list of Document objects

# Prompt for map

map\_prompt = PromptTemplate(

template="Summarize this document:\n{text}",

input\_variables=["text"]

)

# Prompt for reduce

reduce\_prompt = PromptTemplate(

template="Combine the following summaries into a concise summary:\n{summaries}",

input\_variables=["summaries"]

)

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

map\_reduce\_chain = MapReduceDocumentsChain.from\_prompts(

llm=llm,

map\_prompt=map\_prompt,

combine\_prompt=reduce\_prompt

)

summary = map\_reduce\_chain.run(docs)

print(summary)

**5️⃣ Key Takeaways**

1. **Document chains** handle multiple documents intelligently.
2. **Stuff** → small docs, fastest.
3. **Map-Reduce** → large docs, parallelizable.
4. **Refine** → sequential refinement, preserves context.
5. You can **combine with retrievers**:
   * RetrievalQA fetches docs → pass into DocumentChain → produce LLM answer.

If you want, I can **next show a full FastAPI example combining Retrieval + Document Chains**

* User query → Retrieval → Map-Reduce DocumentChain → LLM answer
* Handles large PDFs seamlessly in one endpoint

Do you want me to do that next?

**You said:**

yes please - If you want, I can next show a full FastAPI example combining Retrieval + Document Chains User query → Retrieval → Map-Reduce DocumentChain → LLM answer Handles large PDFs seamlessly in one endpoint

**ChatGPT said:**

Perfect — let’s build a **full FastAPI app** that:

1. Accepts a **user query**
2. **Retrieves relevant documents** from a PDF/vector store
3. Uses a **Map-Reduce DocumentChain** to summarize or answer the query
4. Returns the **final LLM answer** along with sources

This setup handles **large PDFs** gracefully by processing documents in chunks.

**1️⃣ Install dependencies**

bash

CopyEdit

pip install fastapi uvicorn langchain langchain-openai langchain-community faiss-cpu pydantic

**2️⃣ Project structure**

bash

CopyEdit

retrieval\_doc\_chain\_app/

│

├── main.py # FastAPI app

├── sample.pdf # Large PDF document

└── prompt.py # Optional: custom prompt templates

**3️⃣ main.py — Retrieval + Map-Reduce DocumentChain**

python

CopyEdit

from fastapi import FastAPI

from pydantic import BaseModel

from langchain\_community.document\_loaders import PyPDFLoader

from langchain\_community.vectorstores import FAISS

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain.chains.combine\_documents import MapReduceDocumentsChain

from langchain.chains import LLMChain

from langchain.prompts import PromptTemplate

import os

# Set OpenAI API key

os.environ["OPENAI\_API\_KEY"] = "YOUR\_OPENAI\_API\_KEY"

app = FastAPI(title="Retrieval + Map-Reduce DocumentChain API")

# -------------------------

# Step 1: Load documents from PDF

# -------------------------

loader = PyPDFLoader("sample.pdf")

docs = loader.load() # List of Document objects

# -------------------------

# Step 2: Create embeddings + vector store

# -------------------------

embeddings = OpenAIEmbeddings()

vectorstore = FAISS.from\_documents(docs, embeddings)

# -------------------------

# Step 3: Create retriever

# -------------------------

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 5}) # top 5 relevant docs

# -------------------------

# Step 4: LLM setup

# -------------------------

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# -------------------------

# Step 5: DocumentChain prompts

# -------------------------

map\_prompt = PromptTemplate(

template="Extract key points and insights from this document:\n{text}",

input\_variables=["text"]

)

reduce\_prompt = PromptTemplate(

template="Combine the following summaries into a concise answer for the user's question:\n{summaries}",

input\_variables=["summaries"]

)

map\_reduce\_chain = MapReduceDocumentsChain.from\_prompts(

llm=llm,

map\_prompt=map\_prompt,

combine\_prompt=reduce\_prompt

)

# -------------------------

# API request model

# -------------------------

class QuestionRequest(BaseModel):

question: str

# -------------------------

# Step 6: FastAPI endpoint

# -------------------------

@app.post("/query")

async def query\_documents(req: QuestionRequest):

"""User query → Retrieval → Map-Reduce DocumentChain → LLM answer"""

# 1. Retrieve relevant documents

retrieved\_docs = retriever.get\_relevant\_documents(req.question)

# 2. Pass docs to MapReduce chain

answer = map\_reduce\_chain.run(retrieved\_docs)

# 3. Return result + sources

return {

"question": req.question,

"answer": answer,

"sources": [doc.metadata for doc in retrieved\_docs]

}

**4️⃣ Run the server**

bash

CopyEdit

uvicorn main:app --reload

**5️⃣ Test the endpoint**

bash

CopyEdit

curl -X POST "http://127.0.0.1:8000/query" \

-H "Content-Type: application/json" \

-d '{"question": "What are the safety instructions in the manual?"}'

**Example Response:**

json

CopyEdit

{

"question": "What are the safety instructions in the manual?",

"answer": "The manual instructs users to wear protective gear, avoid water contact, and follow proper shutdown procedures to prevent accidents.",

"sources": [

{"source": "sample.pdf", "page": 3},

{"source": "sample.pdf", "page": 5}

]

}

**✅ Key Points**

1. **Vector store retrieval** limits docs to top-k relevant ones → reduces token usage.
2. **Map-Reduce DocumentChain**:
   * Map → LLM processes each doc individually
   * Reduce → LLM combines all intermediate results
3. **Handles large PDFs** seamlessly without hitting context limits.
4. You can **switch map\_prompt and reduce\_prompt** to customize summarization, Q&A, or structured extraction.
5. Sources are returned for traceability.

If you want, I can **next extend this** into a **full Conversational Retrieval + DocumentChain chat** setup:

* Multi-turn chat with memory
* Handles follow-up questions using previous context
* Map-Reduce chain still used under the hood

Do you want me to do that next?

Great! Let’s deep dive into **LCEL (LangChain Expression Language)** — this is the **next-level way** to build flexible, multi-step, composable chains.

LCEL allows you to define **runnable graphs** where chains, LLMs, retrievers, or any callable can be sequenced, branched, run in parallel, retried, or even streamed. It’s like turning your chains into a **programmable workflow engine**.

**1️⃣ What is LCEL?**

* LCEL stands for **LangChain Expression Language**
* Unlike LLMChain or RetrievalQA, LCEL is **not tied to a single chain type**.
* It treats each chain or callable as a **runnable node**.
* You can **combine nodes** using:
  + **Sequence** → run steps one after another
  + **Branching** → run different paths based on input
  + **Parallel** → run multiple steps simultaneously
  + **Retries / Fallbacks** → handle errors gracefully
  + **Streaming** → get partial results while LLM generates output

**2️⃣ Core Building Blocks**

**A. RunnableSequence**

* Run multiple nodes **one after another**
* Output of one node is input to the next

python

CopyEdit

from langchain\_core.runnables import RunnableSequence

# Example: First step: simple LLMChain, second step: another LLMChain

sequence = RunnableSequence([first\_chain, second\_chain])

result = sequence.invoke({"input": "Your input text"})

**B. RunnableBranch**

* Run **different branches** depending on input or conditions
* Example: Choose different LLMs or prompts based on a topic

from langchain\_core.runnables import RunnableBranch

def condition(input\_data):

return "finance" in input\_data.lower()

branch = RunnableBranch(

branches=[(condition, finance\_chain)],

default=general\_chain

)

branch.invoke("Tell me about finance regulations")

**C. RunnableParallel**

* Run multiple runnables **simultaneously**
* Useful for generating multiple summaries, embeddings, or LLM calls in parallel

python

CopyEdit

from langchain\_core.runnables import RunnableParallel

parallel = RunnableParallel([llm\_chain\_1, llm\_chain\_2])

result = parallel.invoke({"input": "Summarize this document"})

**D. RunnableRetry**

* Automatically retry a runnable if it **fails or errors out**
* Can specify **max retries** and **delay**

python

CopyEdit

from langchain\_core.runnables import RunnableRetry

retry\_chain = RunnableRetry(

runnable=llm\_chain,

max\_retries=3,

delay=2

)

**E. RunnableWithFallbacks**

* Provide **alternative runnables** in case the primary fails

python

CopyEdit

from langchain\_core.runnables import RunnableWithFallbacks

fallback\_chain = RunnableWithFallbacks(

primary=llm\_chain\_primary,

fallbacks=[llm\_chain\_backup]

)

**F. Streaming**

* LCEL runnables support **streaming outputs**
* Useful when you want **token-by-token updates** to the client

python

CopyEdit

for token in llm\_chain.stream({"input": "Tell me a story"}):

print(token, end="", flush=True)

**3️⃣ Why Use LCEL?**

* **Composable:** Build multi-step workflows like DAGs
* **Flexible:** Branch, parallelize, retry, or fallback easily
* **Streaming-friendly:** Get outputs as they are generated
* **Universal:** Works with LLMs, retrievers, document chains, or any callable

**4️⃣ Example: Multi-step Workflow**

Imagine a workflow:

1. Detect topic of user query
2. If topic = finance → use finance\_chain
3. Else → use general\_chain
4. Run both in parallel with a summarizer
5. Retry if the LLM fails

python

CopyEdit

from langchain\_core.runnables import RunnableSequence, RunnableBranch, RunnableParallel, RunnableRetry

# Step 1: Topic detection

topic\_detector = llm\_chain\_topic

# Step 2: Branch based on topic

branch = RunnableBranch(

branches=[(lambda x: "finance" in x.lower(), finance\_chain)],

default=general\_chain

)

# Step 3: Parallel summarization

parallel = RunnableParallel([summarizer\_chain\_1, summarizer\_chain\_2])

# Step 4: Retry if needed

retry\_sequence = RunnableRetry(

runnable=RunnableSequence([topic\_detector, branch, parallel]),

max\_retries=2,

delay=1

)

# Run workflow

result = retry\_sequence.invoke({"input": "Tell me about tax regulations in 2025"})

**5️⃣ How LCEL Differs From Traditional Chains**

| **Feature** | **Traditional Chains** | **LCEL** |
| --- | --- | --- |
| **Sequence** | Implicit, 1 chain → next | Explicit RunnableSequence |
| **Branching** | Not directly supported | RunnableBranch |
| **Parallelization** | Needs custom code | RunnableParallel |
| **Retries / Fallbacks** | Manual try/catch | RunnableRetry, RunnableWithFallbacks |
| **Streaming** | LLM-only | Any runnable can stream |

**✅ Key Takeaways**

1. LCEL lets you treat chains **like graph nodes** — connect them, branch them, retry them.
2. Ideal for **complex multi-step workflows**: Retrieval → DocumentChain → LLM → Summarizer → Post-processing
3. All the **retrieval, map-reduce, refine, or LLMChains** can be embedded as nodes.
4. Enables **production-grade pipelines** with **failover and streaming**.

If you want, I can **next build a full FastAPI example using LCEL**:

* User query → Topic detection → Retrieval → DocumentChain → Summarization → Retry & fallback
* Everything **as a runnable graph** with streaming response.

Do you want me to do that next?

**Perfect! Let’s build a full FastAPI example using LCEL as a runnable graph:**

**Workflow:**

1. **User query** comes in
2. **Topic detection** LLMChain decides which branch to follow
3. **Retriever** fetches top-k relevant documents
4. **DocumentChain (Map-Reduce)** processes the retrieved docs
5. **Summarization LLMChain** generates a concise answer
6. **Retry & fallback** in case of errors
7. **Streaming response** to client (token-by-token)

**1️⃣ Install dependencies**

bash

CopyEdit

pip install fastapi uvicorn langchain langchain-openai langchain-community faiss-cpu pydantic

**2️⃣ Project structure**

bash

CopyEdit

lcel\_fastapi\_app/

│

├── main.py # FastAPI + LCEL graph

├── sample.pdf # Large document

└── prompt.py # Optional prompts

**3️⃣ main.py — LCEL FastAPI Setup**

python

CopyEdit

from fastapi import FastAPI

from fastapi.responses import StreamingResponse

from pydantic import BaseModel

from langchain\_community.document\_loaders import PyPDFLoader

from langchain\_community.vectorstores import FAISS

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain.chains import LLMChain

from langchain.prompts import PromptTemplate

from langchain.chains.combine\_documents import MapReduceDocumentsChain

# LCEL imports

from langchain\_core.runnables import RunnableSequence, RunnableBranch, RunnableRetry

import os

import time

from typing import Generator

# -------------------------

# OpenAI Key

# -------------------------

os.environ["OPENAI\_API\_KEY"] = "YOUR\_OPENAI\_API\_KEY"

app = FastAPI(title="LCEL FastAPI Streaming Example")

# -------------------------

# Load documents + vectorstore

# -------------------------

loader = PyPDFLoader("sample.pdf")

docs = loader.load()

embeddings = OpenAIEmbeddings()

vectorstore = FAISS.from\_documents(docs, embeddings)

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 5})

# -------------------------

# LLM

# -------------------------

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# -------------------------

# Step 1: Topic detection chain

# -------------------------

topic\_prompt = PromptTemplate(

template="Detect the topic of the following question. Respond with a single word topic:\nQuestion: {question}",

input\_variables=["question"]

)

topic\_chain = LLMChain(llm=llm, prompt=topic\_prompt)

# -------------------------

# Step 2: Branch chains

# -------------------------

# Finance branch

finance\_prompt = PromptTemplate(

template="Answer this finance question using the following documents:\nQuestion: {question}\nDocuments:\n{documents}",

input\_variables=["question", "documents"]

)

finance\_map\_reduce = MapReduceDocumentsChain.from\_prompts(

llm=llm,

map\_prompt=PromptTemplate("Summarize this finance document:\n{text}", ["text"]),

combine\_prompt=PromptTemplate("Combine summaries:\n{summaries}", ["summaries"])

)

finance\_chain = LLMChain(llm=llm, prompt=finance\_prompt)

# General branch

general\_prompt = PromptTemplate(

template="Answer this general question using the following documents:\nQuestion: {question}\nDocuments:\n{documents}",

input\_variables=["question", "documents"]

)

general\_map\_reduce = MapReduceDocumentsChain.from\_prompts(

llm=llm,

map\_prompt=PromptTemplate("Summarize this document:\n{text}", ["text"]),

combine\_prompt=PromptTemplate("Combine summaries:\n{summaries}", ["summaries"])

)

general\_chain = LLMChain(llm=llm, prompt=general\_prompt)

# -------------------------

# Step 3: Branch based on topic

# -------------------------

def topic\_condition(input\_data):

return "finance" in input\_data.lower()

branch = RunnableBranch(

branches=[(topic\_condition, finance\_map\_reduce)],

default=general\_map\_reduce

)

# -------------------------

# Step 4: Retry wrapper

# -------------------------

retry\_chain = RunnableRetry(

runnable=RunnableSequence([topic\_chain, branch]),

max\_retries=2,

delay=1

)

# -------------------------

# API Request model

# -------------------------

class QueryRequest(BaseModel):

question: str

# -------------------------

# Step 5: Streaming endpoint

# -------------------------

@app.post("/lcel\_query")

async def lcel\_query(req: QueryRequest):

"""Stream response using LCEL runnable graph"""

retrieved\_docs = retriever.get\_relevant\_documents(req.question)

# Generator for streaming

def stream\_generator() -> Generator[str, None, None]:

try:

# LCEL RunnableSequence input

input\_data = req.question

# Topic detection

topic\_result = topic\_chain.run(question=input\_data)

# Branch chain

if "finance" in topic\_result.lower():

doc\_chain = finance\_map\_reduce

else:

doc\_chain = general\_map\_reduce

# Map-Reduce over retrieved docs

final\_answer = doc\_chain.run(retrieved\_docs)

# Stream token by token (simulate)

for token in final\_answer.split(): # simple tokenization for demo

yield token + " "

time.sleep(0.1) # simulate streaming delay

except Exception as e:

yield f"Error: {str(e)}"

return StreamingResponse(stream\_generator(), media\_type="text/plain")

**4️⃣ Run the server**

bash

CopyEdit

uvicorn main:app --reload

**5️⃣ Test the endpoint**

bash

CopyEdit

curl -X POST "http://127.0.0.1:8000/lcel\_query" \

-H "Content-Type: application/json" \

-d '{"question": "Explain the tax regulations in 2025 for corporations"}'

* You will **see the answer streaming token by token** in your terminal.
* Workflow automatically:
  1. Detects topic (finance or general)
  2. Retrieves relevant documents
  3. Runs Map-Reduce DocumentChain
  4. Streams the final answer

**✅ Key LCEL Features in This Example**

1. **RunnableSequence:** topic detection → branch → Map-Reduce
2. **RunnableBranch:** chooses finance vs general chain dynamically
3. **RunnableRetry:** retries on failure automatically
4. **StreamingResponse:** incremental token-by-token output
5. **Shared vector store:** same embeddings used across all chains

This structure is **production-ready** for:

* Multi-step workflows
* Large PDFs
* Dynamic topic-based branching
* Streaming answers

I can next show an **enhanced version** where:

* Multiple branches (more than finance/general)
* Hybrid search (semantic + metadata filtering)
* Refine chain instead of map-reduce for iterative updates

**Absolutely! Let’s build the enhanced LCEL pipeline with these features:**

**Enhancements over the previous version:**

1. **Multiple branches** → e.g., finance, legal, tech, general
2. **Hybrid retrieval** → semantic search + metadata filtering
3. **Refine DocumentChain** → iterative improvement instead of simple map-reduce
4. **Streaming output** → token-by-token response
5. **Retries & fallback chains** → robust production-ready workflow

**1️⃣ Project Structure**

bash

CopyEdit

enhanced\_lcel\_app/

│

├── main.py # FastAPI + LCEL graph

├── sample.pdf # Large document

├── sample\_legal.pdf # Legal docs

├── sample\_finance.pdf# Finance docs

└── prompt.py # Optional prompt templates

**2️⃣ Key Components**

**A. Multi-branch LCEL**

* RunnableBranch with multiple conditions:

python

CopyEdit

def is\_finance(input\_text): return "finance" in input\_text.lower()

def is\_legal(input\_text): return "legal" in input\_text.lower()

def is\_tech(input\_text): return "tech" in input\_text.lower()

* Map each to a **RefineDocumentsChain**:

python

CopyEdit

from langchain.chains.combine\_documents import RefineDocumentsChain

refine\_finance\_chain = RefineDocumentsChain(

llm\_chain=LLMChain(llm=llm, prompt=finance\_prompt),

refine\_llm\_chain=LLMChain(llm=llm, prompt=finance\_refine\_prompt)

)

**B. Hybrid Retrieval**

* Semantic vector retrieval + metadata filtering:

python

CopyEdit

retriever = vectorstore.as\_retriever(search\_kwargs={"k":5})

# Optional: filter by doc type

filtered\_docs = [doc for doc in retriever.get\_relevant\_documents(question) if doc.metadata.get("type")=="finance"]

**C. Refine DocumentChain**

* Iteratively improves answers based on each document
* Good for **large documents** where map-reduce may lose context

**D. Streaming & Retry**

* RunnableRetry wraps the sequence
* Streaming response generator streams tokens from LLM output

If you want, I can now provide the **full Python FastAPI code** for this **enhanced LCEL pipeline**, combining **multi-branch, hybrid retrieval, refine chain, retries, and streaming** in one ready-to-run example.

Do you want me to write that full code next?