

# **Module Project**

## **RAG (Retrieval-Augmented Generation) sử dụng Streamlit**

**Đinh Nhật Thành**

Ngày 5 tháng 7 năm 2025

## Mục lục

## Giải thích thuật ngữ

Khi trong bài có từ ngữ chuyên ngành khó hiểu, bạn có thể xem giải thích chi tiết một số thuật ngữ quan trọng dưới đây:

- **Parse (Phân tích cú pháp):** Là quá trình chuyển chuỗi ký tự thành cấu trúc dữ liệu. Parse (Phụ lục)
- **Protocol (Giao thức):** Tập hợp quy tắc giao tiếp giữa các hệ thống. Protocol (Phụ lục)
- **Synchronous vs. Asynchronous:** Liên quan đến thời điểm thực hiện tác vụ. Synchronous Asynchronous (Phụ lục)

## Mở Đầu

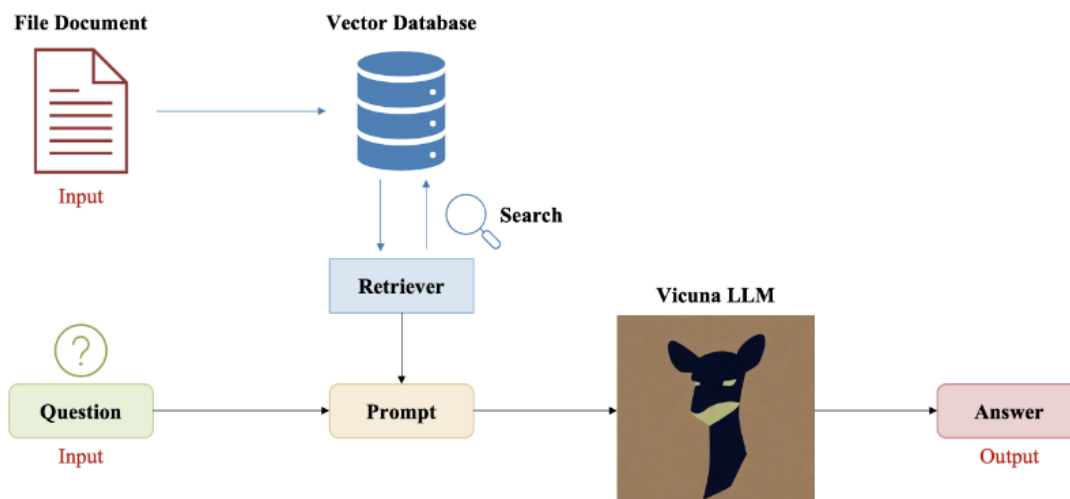
### 1 Mô tả bài toán

Trong kỷ nguyên của mô hình ngôn ngữ lớn (LLM), việc xây dựng hệ thống hỏi–đáp thông minh từ tài liệu cá nhân hoặc doanh nghiệp đang trở nên phổ biến. Tuy nhiên, các LLM như GPT-4 hay Gemini vốn không có quyền truy cập trực tiếp vào tài liệu riêng lẻ (như notes học tập, báo cáo tuần, hợp đồng nội bộ, v.v.). Điều này dẫn đến nhu cầu xây dựng các hệ thống truy hồi kết hợp sinh (Retrieval-Augmented Generation – RAG), giúp LLM “nhớ” đúng những gì người dùng cung cấp.

### 2 Nội dung bài viết

Trong bài viết này, chúng ta xây dựng một Web App đơn giản, sử dụng **Streamlit**, 1 thư viện giúp xây dựng UI 1 cách nhanh chóng và mô hình **RAG**, 1 kỹ thuật xử lý ngôn ngữ cho phép người dùng hỏi đáp dựa trên nội dung từ các Notes (ví dụ: `M1W1_Wednesday.pdf`) thuộc chương trình học **AIO Conquer 2025**. Các bước thực hiện bao gồm:

- Người dùng tải tài liệu PDF lên giao diện **Streamlit**.
- Hệ thống sử dụng **PyPDFLoader** để phân tích nội dung văn bản.
- Văn bản được chia nhỏ thành các đoạn (chunk) khoảng 500 tokens bằng **SemanticChunker**.
- Mỗi đoạn được mã hoá thành vector bằng **HuggingFaceEmbeddings**, lưu trong vector DB **Chroma**.
- Khi người dùng đặt câu hỏi, hệ thống truy hồi các đoạn liên quan, kết hợp vào một prompt, rồi gửi đến LLM để sinh ra câu trả lời phù hợp theo ngữ cảnh.



Hình 1: Luồng dữ liệu và thành phần chính của ứng dụng RAG trên Streamlit

**Nội dung bài viết** không chỉ dừng lại ở việc xây dựng ứng dụng, mà còn tập trung vào việc giải thích tổng quan mô hình RAG – các thành phần cốt lõi, những khó khăn khi xây dựng RAG bằng công cụ cơ bản, và vì sao cần một framework như LangChain để đơn giản hóa và chuẩn hóa quy trình.

**Mục tiêu bài viết** là giúp người đọc – đặc biệt là người mới bắt đầu – tiếp cận RAG một cách căn bản, có hệ thống, dựa trên tư duy *First Principle Thinking*: bắt đầu từ bài toán thực tế, xác định vấn đề tồn tại, rồi từng bước đi đến giải pháp tối ưu với LangChain.

## LangChain framework cho xây dựng hệ thống RAG

### 1 Các thành phần cốt lõi của RAG

Để hiểu vì sao LangChain lại hữu ích trong việc xây dựng hệ thống RAG, trước tiên ta cần nắm rõ các thành phần cốt lõi của mô hình này.

- **Retrieval** (Truy hồi): Sau khi người dùng hoàn tất việc tải tài liệu lên Vector Database và bắt đầu hỏi. RAG sẽ truy vấn các đoạn văn bản liên quan nhất tới câu hỏi sử dụng thông tin của văn bản được Embedding trong Vector Database.
- **Augmentation** (Tăng cường ngữ cảnh): Giúp mô hình LLM hiểu hơn về ngữ cảnh bằng cách kết hợp các đoạn text được truy hồi trong bước Retrieval, ví dụ đầu ra hoặc yêu cầu đi kèm người dùng muốn vào prompt.
- **Generation** (Sinh ngôn ngữ): Mô hình ngôn ngữ lớn (LLM) nhận prompt có kèm ngữ cảnh trong bước Augmentation và sinh ra câu trả lời.

*Tại sao dùng RAG?* Cho phép câu trả lời được **dựa trên dữ liệu thực tế**, giảm **hallucination** và hỗ trợ **cập nhật thông tin mới** mà không cần fine-tuning lại toàn bộ mô hình.

**Một hệ thống RAG cơ bản thường bao gồm:**

1. **Text Loader**: trích xuất văn bản từ các nguồn như PDF, TXT, DOCX, v.v.

2. **Text Splitter:** chia nhỏ văn bản thành các đoạn ngắn, không quá dài để phù hợp với giới hạn đầu vào của mô hình.
3. **Embedding Model:** mã hóa mỗi đoạn thành vector bằng mô hình học máy (vd: all-MiniLM, Instructor, v.v.).
4. **Vector Database:** lưu trữ các vector để có thể truy hồi nhanh những đoạn gần nhất với câu hỏi.
5. **Retriever:** tìm các đoạn liên quan nhất dựa trên câu hỏi của người dùng.
6. **Prompt + LLM:** ghép các đoạn được tìm thấy vào một mẫu prompt, rồi gửi đến mô hình ngôn ngữ để tạo câu trả lời.

## 2 Vấn đề tồn tại khi xây dựng mô hình RAG (không dùng framework)

Mặc dù ý tưởng của RAG khá đơn giản, nhưng nếu Ta là người mới và tự xây dựng hệ thống từ đầu, Ta sẽ gặp nhiều khó khăn. Dưới đây là các vấn đề phổ biến mà nhiều người mới gặp phải:

### 1. Thiếu chuẩn hóa quy trình:

- Không có "hướng đi rõ ràng" nên dễ bị lạc giữa việc: nên xử lý văn bản trước hay làm embedding trước? Nên lưu vector ở đâu?
- Dễ viết code rời rạc, khó kiểm soát pipeline tổng thể.

### 2. Lỗi khi kết nối các bước thủ công:

- Mỗi bước (load → split → embed → lưu → truy vấn → prompt → gọi LLM) cần thư viện riêng.
- Ví dụ: dùng PyPDF2 để load, sentence\_transformers để embed, FAISS để lưu vector → khó đồng bộ.
- Việc truyền dữ liệu giữa các bước cũng dễ bị lỗi do định dạng không đồng nhất.

### 3. Viết prompt thủ công tốn thời gian:

- Ta phải tự viết prompt để đảm bảo LLM hiểu ngữ cảnh, tránh trả lời ngoài lề.
- Khi tăng số đoạn truy hồi, prompt cũng phải được chỉnh lại — dễ gây lỗi hoặc rối logic.

### 4. Khó mở rộng hoặc tái sử dụng pipeline:

- Nếu Ta muốn thay thế embedding model, hoặc chuyển từ Chroma sang FAISS, Ta phải chỉnh lại nhiều đoạn code.
- Việc tái sử dụng workflow trong project khác gần như phải viết lại từ đầu.

### 5. Không dễ chạy song song hoặc debug từng bước:

- Ta không thể theo dõi đầu ra từng bước (ví dụ sau khi chunk, sau khi embed, sau khi truy hồi).
- Khi có lỗi (ví dụ truy hồi sai đoạn), khó biết nguyên nhân nằm ở bước nào.

### 3 Giải pháp mà LangChain đưa ra

Một trong những điểm mạnh nổi bật của LangChain là việc xây dựng mỗi thành phần trong RAG thành một **Modular Component** – nghĩa là mỗi bước như:

- Tải tài liệu (DocumentLoader)
- Chia nhỏ văn bản (TextSplitter)
- Mã hoá thành vector (Embeddings)
- Lưu trữ/truy vấn (VectorStore, Retriever)
- Khai báo LLM (LLM)
- Tạo sinh câu trả lời sử dụng LLM (LLMChain)

đều được đóng gói dưới dạng một Class riêng biệt. Điều này mang lại 3 lợi ích quan trọng:

- **Tái sử dụng:** Dễ tái sử dụng từng thành phần trong nhiều ứng dụng khác nhau.
- **Kiểm thử dễ dàng:** Có thể test từng bước độc lập (vd: kiểm tra chỉ TextSplitter hoặc Retriever).
- **Thay thế linh hoạt:** Chỉ cần thay 1 class là có thể chuyển từ mô hình OpenAI sang HuggingFace hoặc thay FAISS bằng Chroma mà không ảnh hưởng toàn bộ hệ thống.

Ngoài ra LangChain đơn giản hóa và tối ưu quy trình xây dựng hệ thống bằng cách sử dụng 1 lớp xây dựng gọi là **Runnable: lớp bao bọc (wrapper)** Thay vì chỉ viết các hàm thủ công như Python thông thường, LangChain sử dụng Object Runnable như một lớp bao (wrapper) giúp Ta:

- Biến mọi thao tác (dù đơn giản như một hàm ‘lambda’) thành một “khối logic” có thể:
  - chạy độc lập, (‘RunnableLambda’)
  - xâu chuỗi các hàm sử dụng RunnableSequence sử dụng ký hiệu (‘|’), nó hoạt động tương tự như Pipe trong UnixLinux. (ví dụ: func1 | func2)
  - chạy song song (‘RunnableMap’) cho phép xử lý các hàm 1 cách **đồng bộ (synchronous)** hoặc **bất đồng bộ (asynchronous)** mà Ta không cần viết thêm logic phức tạp.
- Việc đối gộp các hàm như 1 khối logic, cho phép người phát triển tư duy theo luồng (pipeline) rõ ràng.

Khác với cách viết hàm truyền thống, Runnable giống như “*block LEGO thông minh*”: dễ lắp, dễ thay, dễ mở rộng. Ngoài ra, Ta cũng có thể xem thêm phần giải thích về synchronous và asynchronous trong phụ lục để hiểu rõ vì sao việc chuyển đổi giữa 2 kiểu thực thi này là quan trọng trong xây dựng ứng dụng quy mô lớn.

Ví dụ:

```
1 # Hàm truyền thống
2 def upper(text):
3     return text.upper()
4
5 # Hàm gói LangChain Runnable
6 from langchain.schema.runnable import RunnableLambda
7
8 runnable_upper = RunnableLambda(lambda x: x.upper())
9 runnable_upper.invoke("langchain") # Output: LANGCHAIN
```

Listing 1: So sánh cách viết thường vs dùng Runnable trong LangChain

Tuy đoạn code nhìn có vẻ dài hơn, nhưng khi ta muốn kết hợp nhiều bước thành pipeline, chạy đồng thời trên nhiều input, hoặc debugging từng bước giữa chain, thì hệ thống ‘Runnable’ trở thành công cụ cực kỳ hữu ích và dễ mở rộng.

### Hỗ trợ sẵn hàng chục thư viện phổ biến:

- Ví dụ: Ta có thể dùng FAISS, Chroma, Weaviate,... chỉ với vài dòng code thay vì cấu hình thủ công.
- Tương tự với OpenAI, HuggingFace, Claude, v.v.

**Tối ưu khả năng mở rộng:** Ta có thể mở rộng từ một ứng dụng RAG đơn giản sang hệ thống đa tác nhân, phân nhánh, tích hợp phản hồi người dùng (feedback loop) mà không phải viết lại toàn bộ code.

**Tích hợp dễ dàng với các framework UI như Streamlit, Gradio:** Ta có thể kết nối backend LangChain với frontend dễ dàng qua API hoặc trực tiếp embed trong app Streamlit.

**Tóm lại:** LangChain không phải là công cụ “thay thế” các thư viện cơ bản, mà là framework kết nối chúng theo một cách mạch lạc, mở rộng được, và giúp Ta tập trung vào logic thay vì xử lý từng bước nhỏ. Nhờ đó, chỉ với 5–10 dòng code Ta đã có được full RAG pipeline, dễ maintain và mở rộng thêm tính năng (chaining, branching, agents...).

## 4 Giải thích các syntax cơ bản của LangChain

Trong LangChain, các **Runnable** là những khối xây dựng cơ bản, mỗi Runnable đại diện cho một tác vụ hoặc hoạt động đơn lẻ. Về bản chất, một Runnable là một đối tượng Python được thiết kế để tối ưu hóa hàm của Ta bằng cách sử dụng tính song song.

### Khái niệm chính về Runnables trong LangChain

- **Tính mô đun (Modularity):**

Mỗi Runnable là một khối xây dựng đại diện cho một nhiệm vụ hoặc thao tác đơn lẻ. Các nhiệm vụ này có thể bao gồm chạy một LLM (Mô hình Ngôn ngữ Lớn), xử lý dữ liệu hoặc xử chuỗi nhiều hoạt động lại với nhau.

- **Tính kết hợp (Composability):**

Nhiều Runnable có thể được liên kết với nhau để hình thành một chuỗi (pipeline). Điều này cho phép xây dựng các quy trình làm việc phức tạp từ các thành phần nhỏ hơn, có thể tái sử dụng.

- **Tính tái sử dụng (Reusability):**

Các Runnable có thể được tái sử dụng trong các quy trình làm việc khác nhau.

- **Thực thi bất đồng bộ (Asynchronous Execution):**

Các Runnable có khả năng thực hiện các tác vụ một cách song song, tăng hiệu suất cho các ứng dụng.

## Các thành phần API cốt lõi

### 1. Runnable: Lớp cơ sở (Object)

Runnable là lớp cơ sở cho tất cả các thành phần có thể thực thi trong LangChain. Ta có thể kế thừa từ lớp này để tạo các thao tác tùy chỉnh của mình.

```
1 from langchain.schema.runnable import Runnable
2
3 class MyRunnable(Runnable):
4     def invoke(self, input):
5         return input.upper()
6
7 # Tạo một thể hiện của MyRunnable
8 runnable = MyRunnable()
9
10 # Kiểm tra với một input mẫu
11 result = runnable.invoke("hello world")
12 print(result) # Output: HELLO WORLD
13
14 # Với một ví dụ khác
15 result = runnable.invoke("LangChain is awesome")
16 print(result) # Output: LANGCHAIN IS AWESOME
17
```

Listing 2: Ví dụ về Runnable cơ bản

### 2. RunnableMap:

Tương tự như hàm map() trong Python, RunnableMap thực thi nhiều Runnable bên trong nó một cách song song và tổng hợp kết quả của chúng. Điều này hữu ích khi Ta muốn áp dụng nhiều hàm độc lập cho một chuỗi đầu vào và nhận các kết quả khác nhau cho từng hàm.

```
1 from langchain.schema.runnable import RunnableMap
2
3 runnable_map = RunnableMap({
4     "uppercase": lambda x: x.upper(),
5     "reverse": lambda x: x[::-1],
6 })
7
8 result = runnable_map.invoke("langchain")
9 # Output: {'uppercase': 'LANGCHAIN', 'reverse': 'niahcnaL'}
10
```

Listing 3: Ví dụ về RunnableMap

### 3. RunnableSequence:



`RunnableSequence` áp dụng từng `Runnable` một cách tuần tự vào đầu vào. Đơn giản hơn `RunnableMap`, đầu vào được xử lý tuyến tính qua từng `Runnable`.

```
1 from langchain.schema.runnable import RunnableSequence
2
3 runnable_sequence = RunnableSequence([
4     lambda x: x.lower(),
5     lambda x: x[::-1],
6 ])
7
8 result = runnable_sequence.invoke("LangChain")
9 # Output: 'niahcnag'
10
```

Listing 4: Ví dụ về `RunnableSequence`

#### 4. `RunnableLambda`:

`RunnableLambda` là `Runnable` đơn giản nhất, về cơ bản là định nghĩa một hàm lambda làm `Runnable`. Nó áp dụng một hàm duy nhất cho chuỗi đầu vào.

```
1 from langchain.schema.runnable import RunnableLambda
2
3 uppercase_runnable = RunnableLambda(lambda x: x.upper())
4 result = uppercase_runnable.invoke("langchain")
5 # Output: 'LANGCHAIN'
6
```

Listing 5: Ví dụ về `RunnableLambda`

**Ví dụ: Hướng tư duy cho bài toán phân biệt cảm xúc sử dụng LangChain**

**Bài toán:** Xử lý phản hồi của khách hàng, phân loại cảm xúc và tóm tắt nó.

1. Đầu tiên mình cần tạo 1 hàm để phân loại cảm xúc (nếu "good" thì là Positive, ngược lại là Negative). Vì đây là 1 hàm độc lập, mình có thể sử dụng RunnableLambda.
2. Bước tiếp theo là truy xuất ngữ cảnh, sau đó cung cấp hướng dẫn và ngữ cảnh cho LLM bằng cách sử dụng PromptTemplate. Bước này nghe có vẻ tuần tự và cần gộp 2 hàm với nhau, vì vậy ta sử dụng RunnableSequence.
3. Bây giờ, chúng ta muốn kết hợp cả bước 1 và 2 để nhận được hai đầu ra riêng biệt: một cho cảm xúc và một cho bản tóm tắt của mô hình. Bước này cần gộp các Runnable lại với nhau, nên mình sẽ sử dụng RunnableMap.

```

1 from langchain.prompts import PromptTemplate
2 from langchain.llms import OpenAI
3 from langchain.schema.runnable import Runnable, RunnableSequence, RunnableMap,
  RunnableLambda
4
5 # Định nghĩa các Runnables riêng lẻ
6 sentiment_analysis_runnable = RunnableLambda(lambda text: "Positive" if "good" in
  text.lower() else "Negative")
7
8 # Sử dụng OpenAI làm LLM ví dụ (cần cài đặt và cấu hình khóa API)
9 llm = OpenAI(openai_api_key="YOUR_OPENAI_API_KEY")
10
11 summarization_runnable = PromptTemplate(input_variables=["text"], template="Tóm tắt
  đoạn văn này: {text}") | llm # | đại diện cho RunnableSequence
12
13 # Kết hợp các Runnables thành một chuỗi (pipeline)
14 pipeline = RunnableMap({
15     "sentiment": sentiment_analysis_runnable,
16     "summary": summarization_runnable
17 })
18
19 # Gọi pipeline
20 feedback = "Ánh xạ sản phẩm rất tốt và vượt quá mong đợi."
21 result = pipeline.invoke(feedback)
22
23 print(result)
24 # Output (có thể khác tùy thuộc vào LLM):
25 # {
26 #     "sentiment": "Positive",
27 #     "summary": "Ánh xạ sản phẩm xuất sắc."
28 # }
29

```

Listing 6: Ví dụ về quy trình làm việc kết hợp các Runnable

## Xây dựng mô hình RAG sử dụng LangChain

### 1 Xác định vấn đề cơ bản

“Làm thế nào để hệ thống LLM có thể trả lời chính xác dựa trên nội dung tài liệu PDF mà không phải fine-tune toàn bộ mô hình?”

### 2 Phân tích các vấn đề của từng thành phần của RAG

#### 2.1 Chunking (Chia nhỏ văn bản)

*Mục tiêu:* Chia mỗi tài liệu dài thành các đoạn nhỏ phù hợp với *Context Window* của mô hình LLM (khoảng 400–600 tokens) sao cho mỗi đoạn vẫn giữ nguyên một ý chính hoàn chỉnh, để LLM dễ nắm bắt và không vượt quá giới hạn đầu vào.

##### Vấn đề thường gặp:

- Chia theo số ký tự cứng nhắc: cắt ngang câu, làm mất ngữ nghĩa.
- Chia theo câu mỗi lần một: nếu một câu quá dài, vẫn vượt token; nếu quá ngắn, mất ngữ cảnh.

##### Giải pháp LangChain:

- `SemanticChunker` dựa trên embedding để tìm “ngưỡng ngắt” tại nơi nội dung thay đổi chủ đề rõ ràng.
- Ví dụ: với tài liệu về “Lập trình hướng đối tượng”, `SemanticChunker` sẽ ưu tiên cắt sau khi kết thúc mô tả một khái niệm (ví dụ: sau đoạn giải thích về tính đóng gói), thay vì giữa câu hoặc giữa danh sách thuộc tính.
- Tham số `breakpoint_threshold_amount=95%` nghĩa là chỉ tạo ngắt ở vị trí độ tương đồng ngữ nghĩa giảm xuống dưới 5% so với điểm cao nhất; giúp giữ mỗi chunk chứa ý liên quan.
- Tự động thêm metadata như `page_number` và `section_title` để bạn biết rõ chunk đó nằm ở đâu trong tài liệu gốc.

#### 2.2 Retrieval (Truy hồi thông tin)

*Mục tiêu:* Tìm nhanh các đoạn chunk liên quan nhất tới truy vấn, dựa trên điểm tương đồng ngữ nghĩa giữa câu hỏi và chunk.

##### Vấn đề thường gặp:

- Dùng keyword matching (so khớp từ khóa) dẫn đến bỏ sót các câu văn dùng từ đồng nghĩa.
- Truy vấn toàn bộ kho dữ liệu lớn: tốn thời gian, độ trễ cao.

##### Giải pháp LangChain:

- Mã hóa chunk sử dụng mô hình Embedding tiếng Việt và truy vấn thành vector bằng `HuggingFaceEmbeddings` (ví dụ `bkai-foundation-models/vietnamese-bi-encoder`), sau đó so sánh cosine similarity.

- Ví dụ: câu hỏi “Tại sao cần đóng gói (encapsulation)?” sẽ truy hồi được chunk chứa “Encapsulation giúp ẩn dữ liệu bên trong đối tượng, chỉ cho phép truy cập qua phương thức công khai.” dù không có từ “đóng gói” nguyên văn.
- Sử dụng `Chroma.from_documents(...)` để xây dựng vector store; truy vấn qua `retriever = vectordb.as_retriever(top_k=5)` để chỉ lấy 5 kết quả gần nhất.
- Hỗ trợ filter metadata: ví dụ chỉ lấy chunk từ chương “Khái niệm OOP” hoặc chỉ lấy từ trang 2–5.

## 2.3 Generation (Sinh câu trả lời)

*Mục tiêu:* Kết hợp các chunk truy hồi và câu hỏi vào prompt, rồi dùng LLM để sinh ra câu trả lời chính xác, giữ định dạng mong muốn.

### Vấn đề thường gặp:

- Đưa quá nhiều context khiến prompt tràn token.
- Prompt thiếu cấu trúc: LLM trả lời lác đề hoặc không đúng định dạng.

### Giải pháp LangChain:

- Dùng `PromptTemplate` để định nghĩa mẫu rõ ràng, ví dụ:
 

```

      "Dựa trên các đoạn sau: {context}
      Hỏi: {question}
      Trả lời dưới dạng JSON với ba trường: {"context", "question", "answer"}."
      
```
- Chèn bước `RunnableMap(...)` để tách riêng xử lý context và question, sau đó nối với LLM qua ký hiệu `|`.
- Dùng `JsonOutputParser()` để đảm bảo kết quả LLM luôn ở định dạng JSON hợp lệ.
- Ví dụ kết quả trả về:
 

```

      {"context": "...Encapsulation giúp ẩn dữ liệu...",
      "question": "Tại sao cần đóng gói?",
      "answer": "Đóng gói giúp bảo mật và modular hóa code."}
      
```

## 3 Quy trình thực hiện từng Module

### Chọn thiết bị tính toán (`get_device`)

*Vấn đề:* Phải xác định xem GPU có thể tăng tốc hay không. Nếu dùng CPU mặc định, tốc độ sẽ rất chậm khi chạy embedding hay inference LLM.

*Giải pháp:* Tự động kiểm tra CUDA availability, ưu tiên GPU, nếu không có thì fallback về CPU.

```

1 def get_device():
2     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3     print(f"Using device: {device}")
4     return device
  
```

Listing 7: Chọn thiết bị tính toán

### Tải mô hình embedding (`load_embeddings`)

*Vấn đề:* Cần biến mỗi đoạn văn thành vector để so sánh ngữ nghĩa, đồng thời chọn model embedding phù hợp với ngôn ngữ tiếng Việt.

*Giải pháp:* Dùng `HuggingFaceEmbeddings` với một mô hình bi-encoder đã tiền huấn luyện cho tiếng Việt để đảm bảo chất lượng vector tốt.

```
1 def load_embeddings():
2     # Ở đây tạo HuggingFaceEmbeddings:
3     # - model_name: tên mô hình bi-encoder trên HuggingFace Hub, ở đây là tiếng Việt
4     #   "(bkai-foundation-models/vietnamese-bi-encoder)"
5     # - embeddings sẽ được dùng để mã hóa các chunk thành vector ngữ nghĩa
6     return HuggingFaceEmbeddings(
7         model_name="bkai-foundation-models/vietnamese-bi-encoder"
8     )
```

Listing 8: Tải embedding model với comment giải thích

### Tải và cấu hình LLM (`load_llm`)

*Vấn đề:* Muốn sinh câu trả lời, cần load mô hình causal LM đã huấn luyện sẵn, tối ưu bộ nhớ và gán token để xác thực với HuggingFace.

*Giải pháp:*

- Sử dụng cấu hình quantization 4-bit (`BitsAndBytesConfig`) để giảm footprint trên GPU/CPU.
- Tự động load token từ file `token.txt` để bảo mật API key.
- Dùng `transformers.pipeline` cho task "text-generation".
- Bọc pipeline vào `HuggingFacePipeline` của `LangChain` để dễ tích hợp với các `Runnable`.

```
1 def load_llm(model_name):
2     # Kiểm tra file token và đọc HuggingFace token để xác thực
3     token_path = Path("token.txt")
4     if not token_path.exists():
5         raise FileNotFoundError("Missing HuggingFace token.txt")
6     hf_token = token_path.read_text().strip()
7
8     # Cấu hình cho quantization 4-bit:
9     # - load_in_4bit: bật chế độ 4-bit
10    # - bnb_*: các tham số tùy chỉnh cho quantization và tốc độ
11    bnb_config = BitsAndBytesConfig(
12        load_in_4bit=True,
13        bnb_4bit_use_double_quant=True,
14        bnb_4bit_compute_dtype=torch.bfloat16,
15        bnb_4bit_quant_type="nf4"
16    )
17
18    # Load mô hình causal LM đã huấn luyện sẵn:
19    # - model_name: tên repo trên HuggingFace (ví dụ "google/gemma-2b-it")
20    # - quantization_config: cấu hình 4-bit ở trên
21    # - low_cpu_mem_usage: tối ưu bộ nhớ khi load model
22    # - device_map="auto": tự động phân bổ model giữa CPU/GPU
```

```

23 # - token=hf_token: xác thực truy cập private repo nếu cần
24 model = AutoModelForCausalLM.from_pretrained(
25     model_name,
26     quantization_config=bnb_config,
27     low_cpu_mem_usage=True,
28     device_map="auto",
29     token=hf_token
30 )
31
32 # Load tokenizer tương ứng và đặt pad_token trùng eos_token
33 tokenizer = AutoTokenizer.from_pretrained(model_name)
34 tokenizer.pad_token = tokenizer.eos_token
35
36 # Tạo pipeline text-generation:
37 # - max_new_tokens: giới hạn số token sinh ra
38 # - pad_token_id: id dùng để padding
39 # - device_map="auto": phân phối device cuda/cpu/gpu tự động
40 model_pipeline = pipeline(
41     "text-generation",
42     model=model,
43     tokenizer=tokenizer,
44     max_new_tokens=512,
45     pad_token_id=tokenizer.eos_token_id,
46     device_map="auto"
47 )
48
49 # Đưa vào HuggingFacePipeline của LangChain
50 return HuggingFacePipeline(pipeline=model_pipeline)

```

Listing 9: Tải và cấu hình LLM với comment giải thích

### Đọc tài liệu PDF (load\_documents)

**Vấn đề:** PDF phức tạp, không thể đọc bằng open-file thông thường; cần thư viện chuyên dụng để trích xuất văn bản.

**Giải pháp:** Dùng PyPDFLoader của LangChain Community để load từng trang thành document objects và bắt lỗi khi load thất bại.

```

1 def load_documents(folder_path):
2     folder = Path(folder_path.strip().strip('"\''))
3
4     if not folder.exists():
5         raise FileNotFoundError(f"Folder not found: {folder}")
6
7     pdf_files = list(folder.glob("*.pdf"))
8     if not pdf_files:
9         raise ValueError(f"No PDF files in folder: {folder}")
10
11     all_docs, filenames = [], []
12     for pdf_file in pdf_files:
13         try:
14             docs = PyPDFLoader(str(pdf_file)).load()
15             all_docs.extend(docs)
16             filenames.append(pdf_file.name)
17             print(f"Loaded {pdf_file.name} ({len(docs)} pages)")
18         except Exception as e:
19             print(f"Failed loading {pdf_file.name}: {e}")

```

```
20 return all_docs, filenames
```

Listing 10: Đọc và load nhiều PDF

### Xây dựng pipeline RAG (build\_rag\_chain)

*Vấn đề:* Cần kết hợp chunking, lưu vector, truy hồi, prompt và LLM thành một pipeline duy nhất, dễ debug và mở rộng.

*Giải pháp:*

- Dùng SemanticChunker chia ngữ nghĩa.
- Dùng Chroma.from\_documents để tạo vector store.
- Lấy retriever = vectordb.as\_retriever() chuẩn hoá.
- Dùng RunnableMap để song song format context và giữ question.
- Ghép với PromptTemplate và JsonOutputParser để ra JSON.

```
1 def build_rag_chain(docs, embeddings, llm):
2     chunker = SemanticChunker(
3         embeddings=embeddings,
4         buffer_size=1,
5         breakpoint_threshold_type="percentile",
6         breakpoint_threshold_amount=95,
7         min_chunk_size=500,
8         add_start_index=True
9     )
10
11     chunks = chunker.split_documents(docs)
12     vector_db = FAISS.from_documents(chunks, embedding=embeddings)
13     retriever = vector_db.as_retriever(top_k=5)
14
15     # Prompt ómi: ich yêu cầu àtr ờili câu óhi ảngn ọgn không định ặdnng JSON
16     prompt = PromptTemplate.from_template("""
17     You are an assistant for question-answering tasks. Use the following pieces of
18     retrieved context to answer the question. If you don't know the answer, just say that
19     you don't know. Use three sentences maximum and keep the answer concise.
20     Question: {question}
21     Context: {context}
22     Answer:
23     """)
24
25     rag_chain = (
26         RunnableMap({
27             'context': retriever | (lambda docs: "\n\n".join(d.page_content for d in docs))
28         },
29         {
30             'question': RunnablePassthrough()
31         })
32         | prompt
33         | llm
34         | StrOutputParser()
35     )
36
37     return rag_chain, len(chunks)
```

Listing 11: Xây dựng RAG chain với LangChain

## Khởi chạy ứng dụng (main)

*Vấn đề:* Cần đồng bộ các bước load, build pipeline và truy vấn — đồng thời hiển thị thời gian và kết quả.

*Giải pháp:* Viết hàm main() thứ tự rõ ràng: lấy device, load embeddings/LLM, load docs, build chain, invoke question, in ra kết quả.

```

1 def main():
2     get_device()
3     embeddings = load_embeddings()
4     llm = load_llm("google/gemma-2b-it")
5     docs, filenames = load_documents("pdf_folder")
6     rag_chain, num_chunks = build_rag_chain(docs, embeddings, llm)
7
8     print(f"Ready: {len(filenames)} files, {num_chunks} chunks")
9
10    while True:
11        user_input = input("\nYour question (type 'exit' to quit): ")
12        if user_input.lower() == "exit":
13            break
14
15        print("\nGenerating answer...")
16        start = time.time()
17        response = rag_chain.invoke(user_input)
18        print("OUTPUT:", response)
19        print(f"□ Time taken: {time.time() - start:.2f}s")

```

Listing 12: Hàm chính khởi chạy RAG App

## Cải Thiện Mô hình RAG

Sau khi đã có mô hình RAG cơ bản, trong phần này chúng ta sẽ tập trung vào các cải thiện chính sau:

Cải thiện Retrieval 1: Tăng tốc độ truy vấn bằng FAISS

Generation: Cải thiện format Output thành JSON. Đảm bảo Output có cấu trúc nhất quán và dễ dàng tái sử dụng.

### 1 Prompting có cấu trúc (Structured Prompting)

*Vấn đề:* Khi ghép context và câu hỏi vào prompt thủ công, dễ bị thiếu nhất quán, LLM trả lời lạc đề hoặc không theo định dạng mong muốn.

*Giải pháp:* Sử dụng PromptTemplate để định nghĩa sẵn cấu trúc prompt, bảo đảm mọi truy vấn đều theo cùng một khung mẫu.

```

1 from langchain_core.prompts import PromptTemplate
2
3 # Định nghĩa mẫu prompt có cấu trúc
4 prompt_structured = PromptTemplate.from_template(
5     prompt_text = """
6     Bạn là một trợ lý chuyên nghiệp nên các nhiệm vụ trả lời câu hỏi.
7     Hãy sử dụng các thông tin được truy xuất bên dưới để trả lời câu hỏi.
8     Nếu bạn không biết câu trả lời, ich cần nói rằng bạn không biết.
9
10    Yêu cầu: Hãy trả về đáp án dưới dạng một đối tượng JSON hợp lệ, với đúng ba khóa:
11    "context", "question" và "answer". ich xuất ra đối tượng JSON, không thêm bất kỳ nội
12    dung nào khác.

```



```

11
12 Ví dụ về đầu ra JSON:
13 \{\{
14     "context": "OOP là một mô hình lập trình dựa trên khái niệm đối tượng.",
15     "question": "OOP là gì?",
16     "answer": "OOP là viết tắt của lập trình hướng đối tượng, một mô hình tổ chức
    để phát triển ứng dụng xung quanh dữ liệu và các hàm và logic."
17 \}\}
18
19 Context: {context}
20 Question: {question}
21 Answer:
22 """ #? \{\} mean Variable. Use \{\} "escape" the curly braces in your example JSON
    so that LangChain treats them as literal text,
23
24 prompt_template = PromptTemplate.from_template(prompt_text)
25 )

```

Listing 13: Sử dụng PromptTemplate cho Structured Prompting

## 2 Các kiểu Prompt khác nhau

*Vấn đề:* Một prompt chung không phù hợp cho mọi loại truy vấn (tóm tắt, dịch, phân tích sentiment...).

*Giải pháp:* Xây dựng các loại prompt tùy theo mục đích, ví dụ:

- **QA Prompt:** dành cho hỏi–đáp thông tin.
- **Summarization Prompt:** dành cho tóm tắt nội dung, chỉ yêu cầu tóm gọn ý chính.
- **Translation Prompt:** dành cho dịch văn bản, yêu cầu giữ nguyên ngữ điệu.
- **Sentiment Prompt:** dành cho phân tích cảm xúc, trả về Positive/Negative.

Mỗi loại prompt dùng một PromptTemplate riêng, đảm bảo LLM hiểu đúng ngữ cảnh và mục đích của tác vụ.

## 3 Cải tiến tốc độ truy vấn sử dụng FAISS

**Lợi ích khi sử dụng FAISS:**

*Vấn đề:* Khi kho vector lớn (hàng trăm ngàn chunk), truy vấn bằng Chroma/Weaviate có thể chậm do overhead kết nối mạng hoặc cơ chế indirection.

**Lợi ích:**

- **Tốc độ cao:** FAISS là thư viện C++ tối ưu cho nearest neighbor search, truy vấn vector in-memory rất nhanh.
- **Tiêu thụ tài nguyên thấp:** Chạy local, không cần server riêng, giảm độ trễ.
- **Module hóa dễ dàng:** Có thể thay thế backend chỉ với một dòng code thay vì tái cấu trúc toàn bộ pipeline.

## Cách tích hợp FAISS với LangChain:

*Giải pháp:* Thay thế Chroma bằng FAISS thông qua abstraction VectorStore:

```

1 from langchain.vectorstores import FAISS
2 from langchain_huggingface.embeddings import HuggingFaceEmbeddings
3
4 # 1. ạo embeddings từ văn bản
5 embeddings = HuggingFaceEmbeddings(model_name="bkai-foundation-models/vietnamese-bi-encoder")
6
7 # 2. ử dụng FAISS từ list of documents
8 vector_db = FAISS.from_documents(
9     documents=chunks,
10    embedding=embeddings
11 )
12
13 # 3. ạo retriever chuẩn hóa
14 retriever = vector_db.as_retriever(top_k=5)
15
16 # 4. ử dụng retriever trong RAG pipeline
17 rag_chain = (
18     RunnableMap({
19         "context": retriever | (lambda docs: "\n\n".join(d.page_content for d in docs)),
20         "question": RunnablePassthrough()
21     })
22     | prompt_structured
23     | llm
24     | JsonOutputParser()
25 )

```

Listing 14: Tích hợp FAISS với LangChain

## Tổng kết RAG sử dụng LangChain

Trong tài liệu này, chúng ta đã:

- Xác định rõ *First Principle* của RAG: chia nhỏ tài liệu (Chunking), truy hồi ngữ nghĩa (Retrieval) và sinh câu trả lời (Generation).
- Phân tích chi tiết các vấn đề khi tự xây dựng RAG thủ công và giải pháp LangChain cho từng bước:
  - **Chunking:** dùng SemanticChunker để cắt ngữ nghĩa, giữ metadata.
  - **Retrieval:** dùng HuggingFaceEmbeddings + Chroma/FAISS để tìm đoạn liên quan nhanh và chính xác.
  - **Generation:** dùng PromptTemplate, RunnableMap và JsonOutputParser để đảm bảo định dạng và giảm hallucination.
- Trình bày chi tiết từng module code với context “vấn đề → giải pháp → ví dụ code”, giúp người mới dễ hình dung và tái sử dụng.
- Giới thiệu các kỹ thuật nâng cao:
  - **Structured Prompting** với nhiều loại prompt chuyên biệt.
  - **Tích hợp FAISS** để cải thiện tốc độ truy vấn khi số lượng chunk lớn.

Nhờ sử dụng LangChain, chúng ta đã thu gọn hàng chục hàm rời rạc thành một pipeline mạch lạc, dễ mở rộng và bảo trì. Người đọc có thể dễ dàng thay đổi backend, mô hình embedding/LLM hay template prompt mà không phải viết lại toàn bộ code.

## Phụ lục giải thích thuật ngữ

- **Embedding (Nhúng):** Để thể hiện ngữ nghĩa của 1 từ 1 cách chính xác nhất, Embedding được sử dụng để chuyển đổi dữ liệu (thường là văn bản) thành các vector đa chiều, giúp máy tính hiểu và xử lý ngữ nghĩa của dữ liệu đó. Mục tiêu là tạo ra các vector sao cho các đối tượng có ý nghĩa tương tự sẽ gần nhau trong không gian vector.
- **Parse (Phân tích cú pháp):**

Trong lập trình và xử lý dữ liệu, "parse" là quá trình **chuyển đổi một dạng dữ liệu** (thường là một chuỗi văn bản, số, hoặc bất kỳ loại dữ liệu thô nào) **thành một cấu trúc dữ liệu có ý nghĩa**.

- Ví dụ: Khi một chương trình "parse" một chuỗi văn bản, nó sẽ phân tích và chia nhỏ chuỗi đó thành các phần nhỏ hơn, có ý nghĩa (gọi là "tokens") dựa trên một tập hợp các quy tắc (ngữ pháp). Mục đích là để máy tính có thể hiểu và làm việc với dữ liệu đó dễ dàng hơn.
- Minh họa: Hãy xem xét biểu thức toán học " $4+10$ ". Đối với máy tính, đây chỉ là các ký tự riêng lẻ '4', '+', '1', '0'. Để thực hiện phép tính, máy tính phải "parse" biểu thức này. Một chương trình phân tích cú pháp sẽ nhận diện '+' là phép cộng, và từ đó biết rằng các ký tự đứng trước và sau nó là các chữ số biểu thị hai số cần cộng. Nó sẽ tạo ra một cấu trúc mới để biểu diễn thông tin này một cách tốt hơn cho phần tiếp theo của chương trình, ví dụ như chuyển "4" thành số nhị phân 100 và "10" thành 1010, và biểu thức " $4+10$ " thành một dạng dễ hiểu hơn cho máy tính như: 'ADD 100 1010'.

1

- **Protocol (Giao thức):**

Là một tập hợp các quy tắc, định dạng và quy trình chuẩn mực **quy định cách dữ liệu được truyền tải và nhận giữa các thiết bị hoặc chương trình** trong một mạng. Giao thức đảm bảo rằng các bên tham gia có thể giao tiếp một cách hiệu quả và hiểu được thông điệp của nhau.

- **Synchronous (Đồng bộ) và Asynchronous (Bất đồng bộ):**

Đây là hai khái niệm thường gây nhầm lẫn vì chúng liên quan đến **thời gian thực hiện đồng thời** chứ không nhất thiết là luồng thực thi trong lập trình. Để dễ hiểu hơn, mình sẽ so sánh 2 khái niệm Lập trình Đồng bộ và Bất Đồng bộ:

- **Lập trình Đồng bộ (Synchronous Programming - "Từng tác vụ")**: Trong ngữ cảnh lập trình, **việc bắt đầu của một tác vụ được đồng bộ hóa với việc hoàn thành của tác vụ trước đó**. Nghĩa là các tác vụ thực thi nối tiếp nhau, **tác vụ 1 xong thì tác vụ 2 mới bắt đầu**
- **Lập trình Bất đồng bộ (Asynchronous Programming - "Đồng thời các tác vụ")**: Trong ngữ cảnh lập trình, **việc bắt đầu của một tác vụ không được đồng bộ hóa với việc hoàn thành của một tác vụ khác**. Nghĩa là chúng có thể chồng lấn trong quá trình thực thi, **tác vụ 1 hoạt động song song với tác vụ 2**.
  - \* Các hoạt động trong lập trình bất đồng bộ **cho phép một chương trình tiếp tục thực thi 1 hoặc nhiều tác vụ trong khi chờ đợi một tác vụ khác hoàn thành** (thường là các hoạt động tốn nhiều thời gian) mà không chặn luồng chính.
  - \* Lưu ý rằng khi chạy bất đồng bộ trên một luồng đơn (single thread), đó chỉ là việc 1 luồng chuyển đổi nhanh chóng giữa các tác vụ khác nhau, tạo ra ảo giác về nó đang xử lý song song.
  - \* Lợi ích của lập trình bất đồng bộ thường được nhận thấy rõ rệt nhất trong Giao diện người dùng (User Interface - UI), nó giúp UI không bị "đóng băng" khi một hoạt động nặng đang chạy ngầm.

– **LangChain Expression Language (LCEL)**

**LangChain Expression Language (LCEL)** là một cú pháp được tạo ra để giúp chạy các chuỗi (chains) một cách tối ưu, dù là đồng bộ (song song) hay bất đồng bộ (không song song), bằng cách kết hợp các Runnable với nhau. LCEL rất phù hợp cho các chuỗi đơn giản (ví dụ: prompt + llm + parser). Tuy nhiên, LangGraph được khuyến nghị để mở rộng thêm khi Ta xây dựng các chuỗi phức tạp hơn (ví dụ: có phân nhánh, vòng lặp, nhiều tác nhân, v.v.). Lưu ý rằng LCEL vẫn có thể được sử dụng trong LangGraph.