Tuần 4 - Phối hợp giữa Streamlit và RAG Chatbot

Time-Series Team

Ngày 5 tháng 7 năm 2025

Tuần 4 bao gồm bốn nội dung chính:

- ***Phần I: RAG Chatbot và Langchain***
- ***Phần II: Streamlit***

Mục lục

Phần I: RAG Chatbot và Langchain

1 Giải thích thuật ngữ

Khi trong bài có từ ngữ chuyên ngành khó hiểu, ta 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)

2 Mở đầu

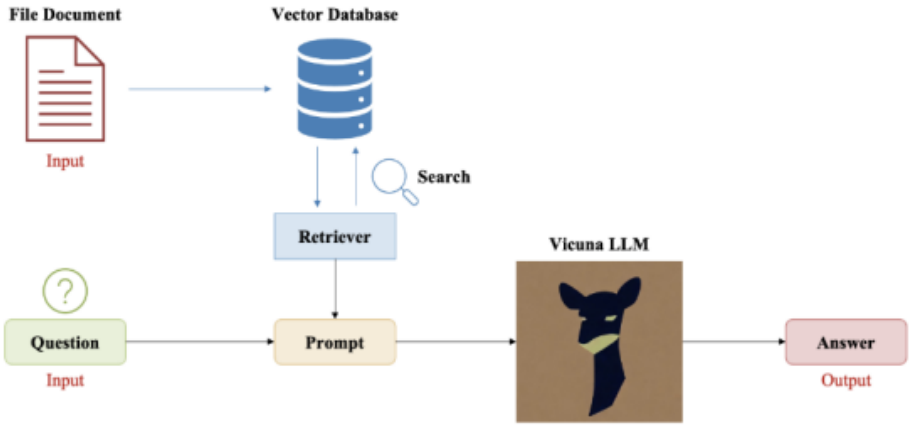
2.1 Mô tả bài toán

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.

2.2 Nội dung bài viết

Trong bài viết này, chúng ta xây dựng một Web App đơn giản, sử dụng **Streamlit**, một thư viện giúp xây dựng UI một cách nhanh chóng, và mô hình **RAG**, một 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.

3 LangChain framework cho xây dựng hệ thống RAG

3.1 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.

3.2 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, 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, 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.

3.3 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`)
- 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 (ví dụ: 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 một 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` với ký hiệu "`|`", hoạt động tương tự pipe trong Unix/Linux (ví dụ: `func1 | func2`)
  - chạy song song (`RunnableMap`) cho phép xử lý các hàm một cách **đồng bộ (synchronous)** hoặc **bất đồng bộ (asynchronous)** mà không cần viết thêm logic phức tạp
- Việc đối gộp các hàm như một 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:

```
# Traditional function
def upper(text):
    return text.upper()

# LangChain Runnable Function
from langchain.schema.runnable import RunnableLambda

runnable_upper = RunnableLambda(lambda x: x.upper())
runnable_upper.invoke("langchain")  # Output: 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 debug 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…).

3.4 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 xâu 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.

```python
from langchain.schema.runnable import Runnable

class MyRunnable(Runnable):
    def invoke(self, input):
        return input.upper()

runnable = MyRunnable()

result = runnable.invoke("hello world")
print(result)  # Output: HELLO WORLD

result = runnable.invoke("LangChain is awesome")
print(result)  # Output: LANGCHAIN IS AWESOME
```

Code Listing 1: 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.

```python
from langchain.schema.runnable import RunnableMap

runnable_map = RunnableMap({
    "uppercase": lambda x: x.upper(),
    "reverse": lambda x: x[::-1],
})

result = runnable_map.invoke("langchain")
# Output: {'uppercase': 'LANGCHAIN', 'reverse': 'niahcnagL'}

```

Code Listing 2: Ví dụ về RunnableMap

3. **RunnableSequence:**

`RunnableSequence` áp dụng từng `Runnable` một cách tuần tự vào đầu vào.

```python
from langchain.schema.runnable import RunnableSequence

runnable_sequence = RunnableSequence([
    lambda x: x.lower(),
    lambda x: x[::-1],
])

result = runnable_sequence.invoke("LangChain")
# Output: 'niahcnag'

```

Code Listing 3: Ví dụ về RunnableSequence

4. **RunnableLambda:**

`RunnableLambda` là `Runnable` đơn giản nhất, dùng để gói gọn một hàm lambda thành một đối tượng `Runnable`.

```
from langchain.schema.runnable import RunnableLambda

uppercase_runnable = RunnableLambda(lambda x: x.upper())
result = uppercase_runnable.invoke("langchain")
# Output: 'LANGCHAIN'

```

Code Listing 4: 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 cần tạo một hàm phân loại cảm xúc (nếu chứa từ "good" thì là Positive, ngược lại là Negative), sử dụng `RunnableLambda`.
2. Sau đó, xây dựng prompt và sử dụng LLM để tạo bản tóm tắt bằng `PromptTemplate` kết hợp với `RunnableSequence`.
3. Cuối cùng, kết hợp cả hai bước trên bằng `RunnableMap` để có hai đầu ra: phân loại cảm xúc và tóm tắt.

```python
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
from langchain.schema.runnable import Runnable, RunnableSequence, RunnableMap,
    RunnableLambda

# Define individual Runnables
sentiment_analysis_runnable = RunnableLambda(
    lambda text: "Positive" if "good" in text.lower() else "Negative"
)

# Use OpenAI as the example LLM (requires API key configuration)
llm = OpenAI(openai_api_key="YOUR_OPENAI_API_KEY")

# Create a summarization prompt template and connect it to the LLM
summarization_runnable = PromptTemplate(
    input_variables=["text"],
    template="Summarize this paragraph: {text}"
) | llm  # "|" is the pipe operator in LCEL (LangChain Expression Language)

# Combine both steps into a pipeline
pipeline = RunnableMap({
    "sentiment": sentiment_analysis_runnable,
    "summary": summarization_runnable
})

# Sample input to test the pipeline
feedback = "The product quality is excellent and exceeds expectations."
result = pipeline.invoke(feedback)

print(result)
# Expected output:
# {
#    "sentiment": "Positive",
#    "summary": "The product quality is outstanding."
# }

```

Code Listing 5: Ví dụ về quy trình làm việc kết hợp các Runnable

4 Xây dựng mô hình RAG sử dụng LangChain

4.1 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?"

4.2 Phân tích các vấn đề của từng thành phần của RAG

4.2.1 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.

4.2.2 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.

4.2.3 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 lạc 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."}
```

4.3 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.

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

Code Listing 6: 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.

```python
def load_embeddings():
    # Initialize HuggingFaceEmbeddings:
    # - model_name: the name of the bi-encoder model on HuggingFace Hub, which supports Vietnamese
    #   ("bkai-foundation-models/vietnamese-bi-encoder")
    # - embeddings will be used to encode chunks into semantic vectors
    return HuggingFaceEmbeddings(
        model_name="bkai-foundation-models/vietnamese-bi-encoder"
    )
```

Code Listing 7: 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`.

```python
def load_llm(model_name):
    # Check for the token file and read the HuggingFace token for authentication
    token_path = Path("token.txt")
    if not token_path.exists():
        raise FileNotFoundError("Missing HuggingFace token.txt")
    hf_token = token_path.read_text().strip()

    # Configuration for 4-bit quantization:
    # - load_in_4bit: enable 4-bit mode
    # - bnb_*: custom parameters for quality and speed
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_compute_dtype=torch.bfloat16,
        bnb_4bit_quant_type="nf4"
    )

    # Load the pre-trained causal LM model:
    # - model_name: repo name on HuggingFace (e.g., "google/gemma-2b-it")
    # - quantization_config: the 4-bit configuration from above
    # - low_cpu_mem_usage: optimize memory usage when loading the model
    # - device_map="auto": automatically distribute the model between CPU/GPU
```

```
    # - token=hf_token: authenticate access to private repos if needed
    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        quantization_config=bnb_config,
        low_cpu_mem_usage=True,
        device_map="auto",
        token=hf_token
    )

    # Load the corresponding tokenizer and set pad_token to be the same as eos_token
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    tokenizer.pad_token = tokenizer.eos_token

    # Create a text-generation pipeline:
    # - max_new_tokens: limit the number of generated tokens
    # - pad_token_id: the id used for padding
    # - device_map="auto": automatically distribute across cuda/cpu/gpu devices
    model_pipeline = pipeline(
        "text-generation",
        model=model,
        tokenizer=tokenizer,
        max_new_tokens=512,
        pad_token_id=tokenizer.eos_token_id,
        device_map="auto"
    )

    # Wrap it in LangChain's HuggingFacePipeline
    return HuggingFacePipeline(pipeline=model_pipeline)
```

Code Listing 8: 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.

```
def load_documents(folder_path):
    folder = Path(folder_path.strip().strip('"\''))

    if not folder.exists():
        raise FileNotFoundError(f"Folder not found: {folder}")

    pdf_files = list(folder.glob("*.pdf"))
    if not pdf_files:
        raise ValueError(f"No PDF files in folder: {folder}")

    all_docs, filenames = [], []
    for pdf_file in pdf_files:
        try:
            docs = PyPDFLoader(str(pdf_file)).load()
            all_docs.extend(docs)
            filenames.append(pdf_file.name)
            print(f"  Loaded {pdf_file.name} ({len(docs)} pages)")
        except Exception as e:
            print(f"  Failed loading {pdf_file.name}: {e}")
```

```
    return all_docs, filenames
```

Code Listing 9: 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.

```
def build_rag_chain(docs, embeddings, llm):
    chunker = SemanticChunker(
        embeddings=embeddings,
        buffer_size=1,
        breakpoint_threshold_type="percentile",
        breakpoint_threshold_amount=95,
        min_chunk_size=500,
        add_start_index=True
    )

    chunks = chunker.split_documents(docs)
    vector_db = FAISS.from_documents(chunks, embedding=embeddings)
    retriever = vector_db.as_retriever(top_k=5)

    prompt = PromptTemplate.from_template("""
    You are an assistant for question-answering tasks. Use the following pieces of
    retrieved context to answer the question. If you don't know the answer, just say that
    you don't know. Use three sentences maximum and keep the answer concise.
    Always answer in Vietnamese

    Question: {question}
    Context: {context}
    Answer:
    """)

    rag_chain = (
        RunnableMap({
            'context': retriever | (lambda docs: "\n\n".join(d.page_content for d in docs))
    ,
            'question': RunnablePassthrough()
        })
        | prompt
        | llm
        | StrOutputParser()
    )

    return rag_chain, len(chunks)
```

Code Listing 10: 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.

```python
def main():
    get_device()
    embeddings = load_embeddings()
    llm = load_llm("google/gemma-2b-it")
    docs, filenames = load_documents("pdf_folder")
    rag_chain, num_chunks = build_rag_chain(docs, embeddings, llm)

    print(f"Ready: {len(filenames)} files, {num_chunks} chunks")

    while True:
        user_input = input("\nYour question (type 'exit' to quit): ")
        if user_input.lower() == "exit":
            break

        print("\nGenerating answer...")
        start = time.time()
        response = rag_chain.invoke(user_input)
        print("OUTPUT:", response)
        print(f"  Time taken: {time.time() - start:.2f}s")
```

Code Listing 11: Hàm chính khởi chạy RAG App

5 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.

5.1 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.

```python
from langchain_core.prompts import PromptTemplate

# Define a structured prompt template
prompt_structured = PromptTemplate.from_template(
    prompt_text = """
        You are an assistant specializing in question-answering tasks.
        Use the retrieved context sections below to answer the question.
        If you don't know the answer, just say that you don't know. Always answer in
    Vietnamese

```

```
10
11          Requirement: Return the response as a valid JSON object, with exactly three keys: "
    context", "question", and "answer". Only output the JSON object, do not add any other
    content.
12
13
14          Example JSON output:
15          {{
16              "context": "OOP is a programming paradigm based on the concept of objects.",
17              "question": "What is OOP?",
18              "answer": "OOP stands for Object-Oriented Programming, a paradigm that
    organizes software design around data, or objects, rather than functions and logic."
19          }}
20
21          Context: {context}
22          Question: {question}
23          Answer:
24      """ #? {} means Variable. Use {{ }} to "escape" the curly braces in your example JSON
    so that LangChain treats them as literal text,
25
26      prompt_template = PromptTemplate.from_template(prompt_text)
27  )
```

Code Listing 12: Sử dụng PromptTemplate cho Structured Prompting

5.2 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.

5.3 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`:

```python
from langchain.vectorstores import FAISS
from langchain_huggingface.embeddings import HuggingFaceEmbeddings

# 1. Create embeddings as before
embeddings = HuggingFaceEmbeddings(model_name="bkai-foundation-models/vietnamese-bi-encoder")

# 2. Initialize FAISS from a list of documents
vector_db = FAISS.from_documents(
    documents=chunks,
    embedding=embeddings
)

# 3. Get the standard retriever
retriever = vector_db.as_retriever(top_k=5)

# 4. Use the retriever in the RAG pipeline
rag_chain = (
    RunnableMap({
        "context": retriever | (lambda docs: "\n\n".join(d.page_content for d in docs)),
        "question": RunnablePassthrough()
    })
    | prompt_structured
    | llm
    | JsonOutputParser()
)
```

Code Listing 13: Tích hợp FAISS với LangChain

6 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.

7 Điểm khác biệt khi tài file PDF từ GitHub repository lên Chatbot

Trong các ứng dụng RAG hoặc các hệ thống hỏi đáp dựa trên tài liệu, việc truy cập và xử lý tài liệu lưu trữ trên GitHub có những đặc thù riêng so với việc đọc file từ đường dẫn tệp tin cục bộ hoặc các nguồn dữ liệu truyền thống.

- **Khác biệt giữa truy cập file trên GitHub và file cục bộ:**
  - **File cục bộ** là tài liệu lưu trữ trực tiếp trên máy tính hoặc máy chủ, có thể truy cập dễ dàng qua hệ thống tập tin (file system).
  - **File trên GitHub** được lưu trữ dưới dạng repository trên nền tảng web, cần thông qua API hoặc giao diện web để truy xuất nội dung.
  - Truy cập trực tiếp URL GitHub thường trả về trang HTML, không phải nội dung file thô, do đó không thể tải trực tiếp file bằng các phương pháp đọc file thông thường.
- **Sử dụng GitHub REST API để truy xuất nội dung:**
  - GitHub cung cấp API chuẩn (REST API) để truy vấn thông tin về các repository, bao gồm danh sách file, nội dung file, và metadata.
  - Khi biết chính xác nhánh (branch) và đường dẫn thư mục trong repository, ta có thể gọi API để lấy danh sách file dạng JSON.
  - Qua API, ta dễ dàng lọc ra các file cần thiết như PDF mà không phải tải toàn bộ repository về máy.
  - Đây là cách tiếp cận tối ưu cho các ứng dụng web hoặc cloud, giúp tiết kiệm băng thông và xử lý linh hoạt.
- **Tại sao cần phân tích URL và trích xuất branch, thư mục?**
  - Một URL GitHub tới thư mục có dạng: `https://github.com/user/repo/tree/branch/path`.
  - Để gọi API đúng, ta cần tách ra từng thành phần `user/repo`, `branch` và `path` thư mục.
  - Việc này giúp xác định chính xác điểm truy cập dữ liệu trong repository.
- **Lọc file PDF dựa trên metadata:**
  - Thông tin trả về từ API bao gồm tên file, loại (file hay thư mục), URL tải xuống.
  - Lọc theo đuôi file ".pdf" và loại "file" giúp tránh nhầm lẫn với thư mục hoặc file không phải PDF.
- **Ý nghĩa trong xây dựng hệ thống RAG:**
  - Việc lấy chính xác các file PDF từ GitHub giúp hệ thống RAG có thể truy xuất tài liệu gốc để xử lý.
  - Khả năng tự động tải tài liệu từ GitHub tăng tính tự động hóa, không cần người dùng phải tải file thủ công.
  - Giúp duy trì dữ liệu luôn mới nhất theo repository mà không cần đồng bộ thủ công.

8 Phụ lục giải thích thuật ngữ

- **Embedding (Nhúng):** Để thể hiện ngữ nghĩa của một từ một cách chính xác, **Embedding** được sử dụng để chuyển đổi dữ liệu (thường là văn bản) thành các vector đa chiều. Điều này giúp máy tính hiểu và xử lý được ý nghĩa của dữ liệu. Mục tiêu là tạo ra các vector sao cho các đối tượng có ý nghĩa tương tự thì 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 chuỗi văn bản, số, hoặc dữ liệu thô) **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 các quy tắc ngữ pháp.
  - Minh họa: Với biểu thức "4+10", máy tính ban đầu chỉ nhìn thấy các ký tự '4', '+', '1', '0'. Để thực hiện phép tính, máy cần "parse" biểu thức này để hiểu rằng '+' là phép cộng, '4' và '10' là các toán hạng. Biểu thức sẽ được chuyển thành một cấu trúc dễ hiểu hơn cho máy như: `ADD 100 1010`.

- **Protocol (Giao thức):**

  Là 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**. Giao thức đảm bảo các bên có thể giao tiếp 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 điểm thực hiện** chứ không nhất thiết là luồng. Phân biệt như sau:

  - **Lập trình Đồng bộ (Synchronous Programming): Mỗi tác vụ được thực hiện lần lượt**, tác vụ sau chỉ bắt đầu khi tác vụ trước hoàn thành.
  - **Lập trình Bất đồng bộ (Asynchronous Programming): Các tác vụ có thể bắt đầu và chạy đồng thời** mà không cần chờ nhau, thường được dùng để xử lý các công việc mất thời gian (ví dụ: tải dữ liệu, chờ phản hồi mạng).
    - Lập trình bất đồng bộ giúp **giữ cho chương trình tiếp tục hoạt động trong khi chờ đợi**, tránh bị "treo".
    - Trên môi trường đơn luồng, điều này được thực hiện bằng cách chuyển đổi nhanh giữa các tác vụ, tạo ảo giác xử lý song song.
    - Đặc biệt hữu ích trong giao diện người dùng (UI) để tránh bị đứng khi có xử lý nền.

- **LangChain Expression Language (LCEL)**

LangChain Expression Language (LCEL) là một cú pháp giúp xây dựng các chuỗi xử lý (chains) một cách tối ưu, hỗ trợ cả thực thi tuần tự (synchronous) và song song (asynchronous) bằng cách kết hợp các thành phần `Runnable`.
LCEL phù hợp cho các pipeline đơn giản như: `prompt → llm → parser`. Với các chuỗi phức tạp hơn (có phân nhánh, vòng lặp, đa tác nhân), nên dùng **LangGraph**. Tuy nhiên, LCEL vẫn có thể được tích hợp như một phần trong LangGraph.

Phần II: Streamlit

1 Giới thiệu về Streamlit

Streamlit là một thư viện mã nguồn mở được phát triển bằng Python, giúp các lập trình viên nhanh chóng xây dựng các ứng dụng web tương tác mà không cần phải lo lắng nhiều về phần giao diện phức tạp. Đặc biệt, Streamlit rất được ưa chuộng trong lĩnh vực Khoa học Dữ liệu (Data Science) và Trí tuệ Nhân tạo (Artificial Intelligence) vì nó giúp minh họa dữ liệu, tạo dashboard, và thử nghiệm mô hình một cách dễ dàng và trực quan.

Một điểm mạnh lớn của Streamlit là tính đơn giản và khả năng phản hồi nhanh (real-time), giúp người dùng cuối tương tác với ứng dụng như nhập dữ liệu, chọn lựa tùy chọn, tải tệp, mà không cần kỹ năng lập trình web chuyên sâu.

2 Thành phần cơ bản trong Streamlit

Một số hàm, widget phổ biến giúp bạn xây dựng giao diện và tương tác với người dùng trong Streamlit bao gồm:

- `st.title()` – Hiển thị tiêu đề chính lớn của trang, thường dùng để giới thiệu hoặc đặt tên ứng dụng.
- `st.header()` – Hiển thị tiêu đề phụ, nhỏ hơn `st.title()` để phân chia các phần trong ứng dụng.
- `st.write()` – Hàm linh hoạt để hiển thị nhiều kiểu nội dung khác nhau như văn bản thuần, bảng dữ liệu, biểu đồ, hình ảnh, v.v.
- `st.text_input()` – Tạo ô nhập liệu văn bản để người dùng nhập dữ liệu dạng chuỗi.
- `st.button()` – Tạo một nút bấm để người dùng có thể kích hoạt hành động nào đó trong ứng dụng.
- `st.file_uploader()` – Cho phép người dùng tải lên tệp tin từ máy tính của họ, rất hữu ích cho các ứng dụng xử lý file.
- `st.selectbox()` – Tạo một dropdown (hộp lựa chọn) để người dùng chọn một trong các phương án có sẵn.

Nhờ các thành phần cơ bản này, bạn có thể xây dựng các ứng dụng web tương tác với người dùng rất nhanh chóng mà không cần thiết kế front-end phức tạp.

3 Quản lý trạng thái với `session_state` trong Streamlit

`st.session_state` là một đối tượng đặc biệt trong Streamlit dùng để lưu giữ trạng thái (state) của ứng dụng trong suốt phiên làm việc của người dùng. Điều này rất quan trọng vì Streamlit sẽ tự động chạy lại toàn bộ mã nguồn mỗi khi có tương tác, nên nếu không lưu trạng thái, các biến sẽ bị khởi tạo lại và mất dữ liệu.

```
if "count" not in st.session_state:
    st.session_state.count = 0
```

Dòng mã trên kiểm tra xem biến `"count"` đã tồn tại trong `session_state` chưa. Nếu chưa, tức là lần chạy đầu tiên, nó sẽ khởi tạo `count` với giá trị 0. Nhờ vậy, biến này được giữ nguyên giá trị qua các lần chạy lại ứng dụng.

Ví dụ ứng dụng:

```
import streamlit as st
if 'chat_history' not in st.session_state:
    st.session_state.chat_history = []
if 'rag_chain' not in st.session_state:
    st.session_state.rag_chain = None
if 'models_loaded' not in st.session_state:
    st.session_state.models_loaded = False
if 'embeddings' not in st.session_state:
    st.session_state.embeddings = None
if 'documents_loaded' not in st.session_state:
    st.session_state.documents_loaded = False
if 'pdf_source' not in st.session_state:
    st.session_state.pdf_source = "github"
if 'github_repo_url' not in st.session_state:
    st.session_state.github_repo_url = "https://github.com/Jennifer1907/Time-Series-Team-Hub/tree/main/assets/pdf"
if 'local_folder_path' not in st.session_state:
    st.session_state.local_folder_path = "./knowledge_base"
if 'processing_query' not in st.session_state:
    st.session_state.processing_query = False
if 'query_input' not in st.session_state:
    st.session_state.query_input = ""
```

Mỗi lần người dùng nhấn nút, giá trị `count` sẽ tăng lên và không bị mất đi do được lưu trong `session_state`.

4 Tùy biến giao diện trong Streamlit với HTML/CSS

Đoạn mã dưới đây sử dụng hàm `st.markdown()` của Streamlit để chèn trực tiếp đoạn mã CSS tùy chỉnh nhằm tạo kiểu cho giao diện ứng dụng web. Việc này giúp bạn có thể kiểm soát chi tiết về bố cục, màu sắc, hiệu ứng,... mà Streamlit mặc định không hỗ trợ hoặc hỗ trợ hạn chế.

```
st.markdown("""
<style>
  /* CSS code */
</style>
""", unsafe_allow_html=True)
```

4.1 Cách vận hành và cú pháp

- `st.markdown()` là hàm để hiển thị nội dung Markdown hoặc HTML. Tham số `unsafe_allow_html=True` cho phép Streamlit xử lý và render mã HTML/CSS nội tuyến, nếu không thì đoạn mã sẽ được hiển thị nguyên văn.
- Toàn bộ đoạn mã CSS được đặt trong thẻ `<style> ... </style>`. Đây là cách để khai báo các quy tắc định dạng cho các phần tử HTML trong ứng dụng.
- Mỗi lớp CSS (ví dụ: `.main-header`, `.chat-container`) định nghĩa kiểu dáng riêng biệt: màu nền, canh lề, bo góc, hiệu ứng animation, v.v.
- Các selector như `.stTextInput > div > div > input` được dùng để can thiệp sâu vào cấu trúc HTML do Streamlit tạo ra, từ đó tùy chỉnh giao diện các widget mặc định.

4.2 Giải thích chi tiết các phần CSS quan trọng

- **.main-header**: Tạo vùng tiêu đề chính căn giữa, có nền gradient màu đỏ-vàng, bo tròn, chữ màu trắng.
- **.chat-container**: Khung chat chính với chiều rộng tối đa 800px, có viền và nền màu xám sáng, hỗ trợ cuộn dọc nếu nội dung dài.
- **.user-message** và **.assistant-message**: Các tin nhắn của người dùng và trợ lý có màu nền, màu chữ khác nhau, bo tròn và có margin để phân biệt hai bên trò chuyện.
- **.chat-input-container**: Vùng nhập tin nhắn được cố định ở cuối trang với nền trắng và viền trên.
- **.stTextInput > div > div > input**: Tùy chỉnh trực tiếp ô nhập văn bản mặc định của Streamlit, ví dụ bo tròn và tăng padding cho ô input.
- **.status-indicator** cùng các lớp trạng thái (ready, loading, error): Hiển thị các dấu hiệu trạng thái bằng các chấm tròn màu sắc khác nhau.
- **.thinking-indicator**: Khung tin nhắn hiển thị trạng thái "đang suy nghĩ" với hiệu ứng nhấp nháy nhẹ (animation pulse).
- **.upload-section** và **.file-counter**: Vùng tải file với viền đứt và số lượng file được upload hiện thị đẹp mắt.
- **.vietnam-flag**: Mô phỏng quốc kỳ Việt Nam với nền đỏ và sao vàng bằng CSS.

4.3 Lưu ý khi sử dụng

- Việc nhúng HTML/CSS với `unsafe_allow_html=True` có thể gây rủi ro bảo mật nếu nội dung không kiểm soát kỹ, đặc biệt với dữ liệu đầu vào từ người dùng.
- Không phải toàn bộ CSS đều được Streamlit hỗ trợ do cấu trúc DOM phức tạp và có thể thay đổi giữa các phiên bản, nên bạn cần kiểm tra kỹ khi nâng cấp Streamlit.
- Cấu trúc HTML do Streamlit tạo ra không được tài liệu chính thức chi tiết nên việc tùy chỉnh sâu (như selector phức tạp) có thể không bền vững hoặc gây lỗi.

4.4 Ứng dụng thực tế

Đoạn CSS trên thường được dùng trong các ứng dụng chatbot, dashboard hoặc các ứng dụng có giao diện tương tác phức tạp, ví dụ:

- Tạo khung chat với các tin nhắn người dùng và trợ lý phân biệt rõ ràng bằng màu sắc và bố cục.
- Tạo vùng nhập liệu cố định dưới cùng cho trải nghiệm người dùng tốt hơn.
- Hiển thị trạng thái xử lý (đang tải, lỗi, sẵn sàng) bằng các chỉ báo màu sắc dễ nhận biết.
- Trang trí giao diện bằng các hiệu ứng gradient, bo tròn, shadow,... giúp ứng dụng chuyên nghiệp, bắt mắt hơn.
- Tùy biến widget mặc định của Streamlit để phù hợp với phong cách riêng.

4.5 Ví dụ tích hợp trong RAG Chatbot Streamlit

```
import streamlit as st
st.markdown("""
<style>
  .main-header{
    text-align: center;
    padding: 1rem 0;
    margin-bottom: 2rem;
    background: linear-gradient(90deg, #ff0000, #ffff00);
    border-radius: 10px;
    color: white;
  }
  .chat-container{
    max-width: 800px;
    margin: 0 auto;
    padding: 1rem;
    max-height: 500px;
    overflow-y: auto;
    border: 1px solid #e0e0e0;
    border-radius: 10px;
    margin-bottom: 20px;
    background-color: #fafafa;
  }
  .user-message{
    background-color: #000000;
    color: #ffffff;
    border-radius: 18px;
    padding: 12px 16px;
    margin: 8px 0;
    margin-left: 20%;
    text-align: left;
    border: 1px solid #333333;
  }
  .assistant-message{
    background-color: #006400;
    color: #ffffff;
    border-radius: 18px;
    padding: 12px 16px;
    margin: 8px 0;
    margin-right: 20%;
    text-align: left;
    border: 1px solid #228b22;
  }
  .chat-input-container {
    position: sticky;
    bottom: 0;
    background-color: white;
    padding: 1rem;
    border-top: 2px solid #e0e0e0;
    border-radius: 10px;
    margin-top: 20px;
  }
  .vietnam-flag::after {
    content: "";
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    color: #ffcd00;
```

```
    font-size: 16px;
  }
</style>
""", unsafe_allow_html=True)

```

Đoạn code trên thể hiện cách sử dụng CSS tùy chỉnh kết hợp với HTML để tạo giao diện chatbot đơn giản trong Streamlit.

5 Streamlit nâng cao

Trong phần này, chúng ta sẽ tìm hiểu một số tính năng nâng cao của Streamlit giúp tối ưu hiệu suất, mở rộng khả năng phát triển và cải thiện trải nghiệm người dùng. Các ví dụ minh họa được rút ra từ đoạn mã ứng dụng truy xuất tài liệu và trả lời câu hỏi sử dụng mô hình embedding tiếng Việt.

5.1 Caching tài nguyên với `@st.cache_resource`

Caching là cơ chế lưu lại kết quả của một hàm để tránh việc tính toán hoặc tải lại nhiều lần, đặc biệt quan trọng khi xử lý tài nguyên nặng như mô hình học máy.

```
@st.cache_resource
def load_embeddings():
    """Tải mô hình embedding tiếng Việt"""
    return HuggingFaceEmbeddings(model_name="bkai-foundation-models/vietnamese-bi-encoder")
```

Giải thích:

- `@st.cache_resource` là decorator của Streamlit dùng để cache một tài nguyên như mô hình ML, database connection, hoặc vector store.
- Khi ứng dụng rerun (ví dụ: do người dùng tương tác), hàm `load_embeddings()` sẽ không được gọi lại nếu không có thay đổi trong nội dung hàm, giúp tiết kiệm thời gian tải.
- Trong ví dụ này, mô hình bi-encoder từ HuggingFace được tải một lần và có thể được sử dụng nhiều lần sau đó trong ứng dụng mà không cần reload lại.

Lưu ý:

- Nếu thay đổi tham số đầu vào hoặc cập nhật mô hình, cần xóa cache thủ công hoặc sử dụng phiên bản cache khác.
- Không nên dùng cache cho tài nguyên có trạng thái thay đổi thường xuyên (ví dụ như kết nối thời gian thực).

5.2 Multi-page App trong Streamlit

Trong các ứng dụng phức tạp, việc gói toàn bộ logic vào một file duy nhất như `app.py` là không tối ưu. Streamlit cung cấp tính năng **multi-page app** giúp chia nhỏ giao diện thành nhiều trang (pages), mỗi trang phụ trách một chức năng riêng biệt. Điều này không chỉ giúp mã nguồn rõ ràng, dễ bảo trì mà còn tạo trải nghiệm người dùng tốt hơn.

Cấu trúc thư mục cơ bản

Khi muốn tạo ứng dụng nhiều trang, ta cần tổ chức dự án như sau:

```
project_folder/

   app.py
   pages/
       Trang_1.py
       Trang_2.py
```

- `app.py`: Là trang chính, được thực thi đầu tiên khi chạy lệnh `streamlit run app.py`.
- `pages/`: Là thư mục chứa các file đại diện cho từng trang riêng biệt. Tên file sẽ được Streamlit sử dụng làm nhãn trong Sidebar.
- `Trang_1.py`, `Trang_2.py`: Mỗi file tương ứng với một trang riêng của ứng dụng. Ví dụ: Trang upload file, trang hỏi đáp, trang thống kê,...

Cách hoạt động

Khi người dùng chạy `streamlit run app.py`, Streamlit sẽ tự động:

1. Dò tìm thư mục `pages/` cùng cấp với `app.py`.
2. Liệt kê tất cả file `.py` trong thư mục này.
3. Hiển thị tên các file như một menu trong Sidebar.
4. Khi người dùng chọn một trang trong Sidebar, Streamlit sẽ thực thi file tương ứng.

Ví dụ minh họa

app.py:

```
import streamlit as st

st.set_page_config(
    page_title="Trợ Lý AI Tiếng Việt",
    layout="wide",
    initial_sidebar_state="expanded"
)
```

pages/Trang_1.py:

```
import streamlit as st

st.title("Trang 1: Tải tài liệu")
uploaded_file = st.file_uploader("Tải tệp tài liệu lên", type=["pdf", "docx", "xlsx"])
if uploaded_file:
    st.success(f"Đã tải lên: {uploaded_file.name}")
```

pages/Trang_2.py:

```
import streamlit as st

st.title("Trang 2: Đặt câu hỏi")
question = st.text_input("Nhập câu hỏi của bạn")
if question:
    st.write(f"Bạn đã hỏi: {question}")
```

Lợi ích của Multi-page App

- **Tổ chức code rõ ràng**: Mỗi chức năng độc lập nằm trong file riêng biệt.
- **Dễ mở rộng**: Có thể thêm trang mới chỉ bằng cách thêm file mới vào thư mục `pages/`.
- **Trải nghiệm người dùng tốt**: Giao diện chia thành nhiều bước, dễ điều hướng.
- **Hỗ trợ session state giữa các trang**: Có thể chia sẻ dữ liệu giữa các trang bằng `st.session_state`.

Lưu ý khi sử dụng Multi-page App

- Tên file trong `pages/` sẽ xuất hiện trong Sidebar nên cần đặt tên dễ hiểu, tránh tên file như `test1.py`.
- Tránh việc đặt lại `set_page_config` trong các file trong `pages/` (chỉ nên set ở `app.py`).
- Sử dụng `st.session_state` để chia sẻ dữ liệu giữa các trang (nếu cần lưu kết quả từ trang trước).

Giải thích:

- `app.py` là trang chính của ứng dụng.
- Thư mục `pages/` chứa các file Python đại diện cho từng trang riêng biệt.
- Khi chạy ứng dụng, Streamlit sẽ tự động nhận diện và hiển thị các trang này trong sidebar để người dùng chuyển đổi qua lại.

Ứng dụng: Tổ chức các chức năng như "Tải tài liệu", "Truy vấn câu hỏi", "Xem lịch sử tương tác" thành các trang riêng biệt để tăng tính modular.

5.3 Tự động làm mới ứng dụng với `st.experimental_rerun()`

Trong một số trường hợp, ta cần làm mới toàn bộ ứng dụng khi có thay đổi dữ liệu hoặc trạng thái, ví dụ sau khi tải xong file hoặc khi thay đổi tuỳ chọn.

```
if st.button("Tải lại tài liệu"):
    st.experimental_rerun()
```

Giải thích:

- `st.experimental_rerun()` giúp reload lại toàn bộ ứng dụng Streamlit ngay lập tức.
- Có thể dùng kết hợp với điều kiện hoặc trigger bởi sự kiện từ phía người dùng.
- Thích hợp để reset giao diện, cập nhật dữ liệu hoặc đưa ứng dụng về trạng thái ban đầu.

Ứng dụng thực tế trong ví dụ: Sau khi người dùng upload file ZIP, ứng dụng có thể tự động làm mới để hiển thị kết quả mới:

```
if uploaded_file.name.endswith(".zip"):
    documents, files_loaded = process_zip_file(uploaded_file)
    st.success(f"Đã load {len(files_loaded)} file từ ZIP.")
    st.experimental_rerun()
```

Lưu ý:

- Vì là hàm `experimental`, có thể thay đổi hoặc nâng cấp trong tương lai.
- Nên tránh gọi `st.experimental_rerun()` trong vòng lặp vô hạn hoặc ngoài điều kiện kiểm soát.