Gemini Chat

xrick/mypdfchat
GitHub

Analyze this project and generate a plan of refactoring the architecture which must satisfy the following requirements
The architecture include the for the following Layers
1. LLM Provider Layer:
- Including Ollama, vLLM, llama.cpp
- Unified OpenAI Compatible APIs
2. Services Layer
- CoreLogicService
- InputDataHandleService
- PromptService
- RetrievalService
- StateTransitionService
3. Presentation Layer
- Optimistic Progressive Markdown Parsing
- Data Export

程式夥伴

您好！我是程式夥伴，我很樂意協助您進行專案的架構重構。您提出的三層架構（LLM Provider、Services、Presentation）是一個非常專業且可擴展的設計模式，特別適合像 RAG 這樣的複雜應用。

根據您提供的專案分析報告（`system_overview_20251021.md`）和現有程式碼（`app.py`, `modular_app.py`），目前的程式碼存在「業務邏輯與介面耦合過緊」和「程式碼碎片化」的問題。重構計畫的重點將是**解耦**，並將現有功能精確地對應到您要求的新服務層中。

專案架構重構計畫：三層解耦 RAG 系統

我們將採用嚴格的分層架構（Layered Architecture）來達成您的目標，確保各層之間僅透過定義好的介面（Interfaces）互動。

第 1 步：定義新的目錄結構 (Project Structure)

首先，建議將所有核心業務邏輯從 `app.py` 和 `modular_app.py` 中分離出來，建立專門的服務目錄：

```
.
├── src/
│   ├── services/           # 核心服務層 (Services Layer)
│   │   ├── __init__.py
│   │   ├── llm_provider/   # LLM Provider Layer (實作類別)
│   │   ├── data_handler/   # InputDataHandleService
│   │   ├── retrieval/      # RetrievalService
│   │   ├── core/           # CoreLogicService
│   │   └── state/          # StateTransitionService
│   ├── models/             # 資料模型 (Pydantic/TypedDict)
│   └── prompts/            # PromptService
├── web/
```

```
│   ├── streamlit_app.py    # Presentation Layer (Streamlit 介面)
│   └── api_routes.py       # Presentation Layer (FastAPI 介面)
├── requirements.txt
└── .env
```

第 2 步：LLM Provider Layer (統一 API 介面)

目標： 建立一個抽象介面，讓上層的 **CoreLogicService** 無需知道它正在與 Ollama、vLLM 或 llama.cpp 互動，所有模型都透過 **OpenAI-Compatible API** 進行通訊。

類別 / 檔案	描述	導入方式 (Implementation Details)
`src/services/llm_provider/base.py`	抽象基礎類別 (`BaseLLMClient`)	定義 `async def generate(self, messages, stream=False)` 介面。
`src/services/llm_provider/openai_compat.py`	實作類別 (`OpenAICompatClient`)	使用 `openai` 或 `langchain_openai` 函式庫，將 `base_url` 設定為 Ollama (現有)、vLLM 或 llama.cpp 的 OpenAI 相容端點。
整合	`app.py` 中的 `init_ollama_model` 將被此層取代。	處理連線、重試邏輯和模型名稱 (例如： `gpt-oss:20b`) 的設定。

第 3 步：Services Layer (核心業務邏輯)

這是重構最關鍵的一層，它將把目前分散在 `app.py` 中的所有 RAG 邏輯模組化。

2.1. InputDataHandleService (`src/services/data_handler/`)

職責： 處理原始輸入資料（PDF 檔案）的解析、清理和分塊（Chunking）。

- **現有功能映射：**
  - `app.py` 中的 `PdfReader` 和 `RecursiveCharacterTextSplitter`。
- **重構重點：**
  - 實現 `extract_text_streaming` 以優化大型 PDF 的文本提取。
  - 實現 **Parent-Child Chunking** 邏輯，這已在 `modular_app.py` 的標題中提及。
  - 加入 `system_overview_20251021.md` 建議的**輸入驗證**（例如：檔案大小限制、檔名淨化）。

2.2. RetrievalService (`src/services/retrieval/`)

職責： 管理向量儲存、嵌入模型（Embeddings）和文件檢索。

- **現有功能映射：**
  - `app.py` 中的 `get_embedder`、`HuggingFaceEmbeddings` 和 `FAISS` 操作。
- **重構重點：**

- **持久化：** 棄用不安全的 Pickle 序列化 (`filename + ".pkl"`)，改用 FAISS 的原生 `write_index` / `read_index` 方法。
- **效能：** 實作 `system_overview_20251021.md` 中建議的 `ThreadSafeLRUCache` 或 Redis，用於快取向量儲存，解決記憶體無限制增長的問題。
- **異步化：** 將同步的嵌入模型初始化改為異步 (`asyncio.to_thread`)，解決啟動時阻塞事件循環的效能問題。

2.3. PromptService (`src/prompts/`)

職責： 建立和管理 LLM 的提示模板（Prompt Templates），以及執行查詢轉換（Query Transformation）。

- **現有功能映射：**
  - `app.py` 中 `conversational_chat` 內部隱含的 RAG 提示組裝邏輯。
- **重構重點：**
  - 將系統提示、RAG 提示和聊天歷史格式化邏輯集中管理。
  - 導入進階 RAG 技術（參考您專案中的 Notebooks 檔案），例如 **HyDE** 或 **Query Transformations** 的預檢索步驟，以優化檢索結果。

2.4. StateTransitionService (`src/services/state/`)

職責： 管理使用者會話狀態、聊天歷史記錄和文件狀態，使其與前端框架（Streamlit/FastAPI）**解耦**。

- **現有功能映射：**
  - `app.py` 中直接使用的 `st.session_state['history']` 和 `st.session_state['ready']`。
- **重構重點：**
  - 定義一個標準的 `ChatState` 類別來保存歷史記錄。
  - 在 Streamlit 應用中，將 `st.session_state` 僅用於儲存 `StateTransitionService` 的實例，並讓所有狀態操作透過該實例進行。
  - 在 FastAPI 應用中，此服務將使用外部的 Redis 或資料庫來追蹤每個會話的狀態，從而實現水平擴展的可能性。

2.5. CoreLogicService (`src/services/core/`)

職責： 作為總協調者，定義 RAG 流程的執行順序。

- **現有功能映射：**
  - `app.py` 中 `conversational_chat` 函式 的主要執行步驟。
- **重構重點：**
  1. 從 **StateTransitionService** 獲取歷史記錄。
  2. 呼叫 **PromptService** 執行查詢轉換。
  3. 呼叫 **RetrievalService** 根據轉換後的查詢獲取相關文件上下文。
  4. 呼叫 **PromptService** 組合最終的生成提示。
  5. 呼叫 **LLM Provider Layer** 進行 LLM 呼叫。
  6. 將結果更新回 **StateTransitionService**。

第 4 步：Presentation Layer (使用者介面與資料呈現)

目標： 徹底將 UI 程式碼和業務邏輯分離。所有 UI 元素僅呼叫 **CoreLogicService** 的方法。

介面 / 功能	描述	導入方式 (Implementation Details)
Streamlit UI (`web/streamlit_app.py`)	聊天介面	將 `modular_app.py` 作為基礎，但所有「Process PDF」和「Thinking...」邏輯 都將導向 **CoreLogicService**。
Optimistic Progressive Markdown Parsing	串流回應	核心服務應以 **異步產生器**（Async Generator）的形式回傳 LLM 的回應。Streamlit UI 應迭代此產生器，並利用 **st.empty()** 或其他技術實現 **即時渲染** (Progressive Markdown Parsing)，提升使用者體驗。
Data Export	匯出聊天記錄	在 Streamlit 介面中新增一個按鈕（例如在側邊欄），呼叫 **StateTransitionService** 獲取完整的聊天記錄，並將其格式化為 JSON 或 Markdown 檔案，供使用者下載。

步驟總結與教學

這個重構計畫遵循 **SOLID** 原則中的**單一職責原則** (Single Responsibility Principle)，將每個核心邏輯封裝在專門的服務類別中。

1. **從底層開始 (LLM Provider)：** 建立 LLM 抽象層，這讓您可以在不修改任何上層程式碼的情況下，輕鬆切換 Ollama、vLLM 或未來的 LLM 服務。
2. **建立服務層 (Services)：** 逐一建立 `InputDataHandleService`、`RetrievalService` 等類別，將現有 `app.py` 中的函式重寫為類別的方法，並加入 `system_overview_20251021.md` 建議的效能和安全優化。
3. **頂層協調 (CoreLogicService)：** 撰寫 `CoreLogicService` 類別，該類別只負責呼叫其他服務的公開方法，確保流程正確。
4. **最後連接 (Presentation)：** 重寫 Streamlit/FastAPI 程式碼，使其僅依賴於 `CoreLogicService` 來處理所有使用者請求。

這將極大地提高程式碼的可維護性、可測試性，並解決您專案中現有的架構限制。