

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）是現代化設計，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「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，所有模型都透過通訊。

類別 / 檔案	描述	導入方式 (Implementation Details)
<code>src/services/llm_provider/base.py</code>	抽象基礎類別 (<code>BaseLLMClient</code>)	定義 <code>async def generate(selected_model: str, prompt: str, stream=False)</code> 介面。
<code>src/services/llm_provider/openai_compat.py</code>	實作類別 (<code>OpenAICompatClient</code>)	使用 <code>openai</code> 或 <code>langchain_openai</code> 設定為 Ollama (現有)、vLLM 或 llama.cpp。
整合	<code>app.py</code> 中的 <code>init_ollama_model</code> 將被此層取代。	處理連線、重試邏輯和模型名稱 (例如 <code>gpt-4o</code>)。

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，解決高併發下的效能問題。
 - **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，以改善檢索品質。

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
 - 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
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 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，這讓您可以不修改任何上層程式碼的情況下，輕鬆切換 Ollama、vLLM
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 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。

System Role: You are an experienced and profound architect task:

1. We are building a LLM RAG Web application which let users can chat, query and ask with uploaded documentation.
2. please analyze the uploaded ai_chat.txt which is the architecture of this web application and OPMP_Phase_Flow approach of progressive streaming output.
3. What your primary mission is to improve the architecture describe in ai_chat_v0.2.txt.
4. You need step by step, carefully and completely to accomplish your primary mission.
5. You need to implement the skeleton Python code for this framework, especially the opmp five steps codes.
6. you ****should implement**** the codes with FastAPI, LangChain, LangGraph, and other third-party libraries which implementation.

<> 程式夥伴

身為一位經驗豐富且深刻的架構師，我將針對您提供的 `ai_chat_v0.2.txt` 架構進行分析，並將 `OPMP_Phase_Flow_Analysis` (OPMP) 流程完美整合到您的 **FastAPI + LangChain/LangGraph** 框架中。

核心的架構改進是將原本單一的 RAG 流程，轉化為一個由 **LangGraph** 驅動、**狀態轉換服務**管理的**多階段工作流**，並透過 **Fast** 出。

1. 架構改進與整合計畫 (Architectural Improvement Plan)

您的 `ai_chat_v0.2.txt` 已經建立了清晰的三層架構 (LLM Provider, Services, Presentation)。主要的改進是將 **OPMP 五階段** 的核心執行單元，並透過 **LangGraph** 進行編排。

Layer 層級	元件 (Component)	整合 OPMP 角色 (Integration Role)
LLM Provider	<code>OpenAICompatClient</code>	統一介面，支援 Phase 1 (Query Analysis) 和 Phase 3 (Synthesis) 的串流呼叫。
Services Layer	<code>CoreLogicService</code>	核心編排器 。使用 LangGraph 實現 OPMP 流程，並串接所有其他服務。
	<code>RetrievalService</code>	執行 Phase 2 。處理多源異步檢索 (e.g., Milvus, DuckDB) 和 Reranking。
	<code>StateTransitionService</code>	管理 <code>ChatState</code> 。記錄並更新每階段的輸出 (Analysis, Evidence, Answer)，確保狀態一致性。
	<code>PromptService</code>	處理所有階段的提示模板，特別是 Phase 1 的 Query Transformation 和 Phase 3 的 Synthesis Prompting。
Presentation Layer	<code>http_stream.py</code>	負責 FastAPI 的 SSE 輸出 。將 <code>CoreLogicService</code> 產生的 OPMP 訊息流轉發給客戶端。

2. 骨架程式碼實作與 OPMP 整合步驟 (Skeleton Code Implementation)

我們將按照邏輯依賴性，從資料模型開始，逐步建立 **LangGraph** 工作流和 **FastAPI** 介面。

步驟一：定義領域模型與 OPMP 資料傳輸物件 (DTO)

我們需要定義 RAG 系統的狀態以及 OPMP 流程中傳輸的訊息格式。

`domain/models.py` 骨架程式碼

```
from pydantic import BaseModel, Field
from typing import List, Dict, Any, Literal, Tuple
```

```

# --- OPMP DTOs ---
class ProgressMessage(BaseModel):
    """用於 FastAPI SSE 串流輸出的訊息格式 (OPMP)"""
    type: Literal["progress", "token", "final", "error"] = "progress"
    phase: int = 0
    message: str = ""
    content: str = "" # 串流 token 或最終內容

class Citation(BaseModel):
    source_id: str
    content_snippet: str
    page: int

class RetrievalResult(BaseModel):
    chunks: List[str]
    citations: List[Citation]

# --- RAG 核心狀態模型 ---
class ChatState(BaseModel):
    """LangGraph 內部用於傳遞和更新的狀態 (等同於 StateTransitionService 的狀態)"""
    query: str
    chat_history: List[Tuple[str, str]] = Field(default_factory=list)

    # OPMP Phase 1 Output
    analyzed_query: str = "" # 經改寫或實體提取後的查詢

    # OPMP Phase 2 Output
    retrieval_result: RetrievalResult = Field(default_factory=RetrievalResult)

    # OPMP Phase 3-5 Output
    final_answer: str = ""
    usage_metadata: Dict[str, Any] = Field(default_factory=dict)

# 將 LangGraph 的 State 映射到 Pydantic Model
class RAGGraphState(ChatState):
    # LangGraph 節點間的額外狀態，例如路由決策

Python

```

步驟二：OPMP LangGraph 工作流編排 (CoreLogicService 核心)

將 OPMP 的五個階段映射到 LangGraph 的節點 (Nodes) 和邊 (Edges) 中。

`services/core_logic.py` **骨架程式碼 (LangGraph)**

```

from langgraph.graph import StateGraph, END, START
from langgraph.prebuilt import ToolExecutor
from typing import TypedDict, Annotated
import operator
import asyncio

# 引入自定義的服務
from services.retrieval_service import RetrievalService
from services.prompt_service import PromptService
from services.llm_gateway import LLMGateway
from domain.models import RAGGraphState, ProgressMessage, RetrievalResult

# 初始化服務 (LangGraph Node 需依賴這些服務)
# 這裡使用 None 佔位，實際應透過依賴注入 (DI) 初始化
RETRIEVAL_SVC = RetrievalService()
PROMPT_SVC = PromptService()
LLM_GATEWAY = LLMGateway()

# 讓 LangGraph 知道狀態是如何傳遞的

```

```

class RAGState(TypedDict):
    state: RAGGraphState
    messages: Annotated[List[ProgressMessage], operator.add] # 收集輸出的串流訊息

class RAGGraphExecutor:
    """使用 LangGraph 實現 OPMP 五階段流程的執行器"""

    def __init__(self, retrieval_svc: RetrievalService, prompt_svc: PromptService, llm_gateway: LLMGateway):
        self.retrieval_svc = retrieval_svc
        self.prompt_svc = prompt_svc
        self.llm_gateway = llm_gateway
        self.graph = self._build_graph()

    def _build_graph(self):
        workflow = StateGraph(RAGState)

        # ----- OPMP Phase 1: Query Understanding & Entity Extraction -----
        def query_analysis_node(state: RAGState) -> RAGState:
            """Phase 1: 執行查詢改寫 (Query Rewriting) 或實體提取"""
            # 1. 發送進度訊息
            progress_msg = ProgressMessage(phase=1, message="Analyzing query and extracting intent...")
            state["messages"].append(progress_msg)

            # 2. 執行 LLM 呼叫 (非串流)
            analyzed_query = self.prompt_svc.rewrite_query(state["state"].query)

            # 3. 更新狀態
            state["state"].analyzed_query = analyzed_query
            return state

        # ----- OPMP Phase 2: Parallel Multi-source Data Retrieval -----
        async def retrieval_node(state: RAGState) -> RAGState:
            """Phase 2: 並行執行多源檢索和 Reranking"""
            progress_msg = ProgressMessage(phase=2, message="Executing parallel multi-source retrieval..")
            state["messages"].append(progress_msg)

            # 異步並行呼叫 RetrievalService (模擬，實際應使用 asyncio.gather)
            retrieval_result: RetrievalResult = await self.retrieval_svc.parallel_retrieve(state["state"].analyzed_query)

            # 更新狀態
            state["state"].retrieval_result = retrieval_result
            return state

        # ----- OPMP Phase 3: RAG Synthesis & Answer Generation -----
        async def synthesis_node(state: RAGState) -> RAGState:
            """Phase 3: LLM 串流生成答案"""
            progress_msg = ProgressMessage(phase=3, message="Synthesizing final answer from context (Streaming)")
            state["messages"].append(progress_msg)

            # 1. 準備提示 (Prompt Bundle)
            prompt_bundle = self.prompt_svc.build_rag_prompt(
                query=state["state"].query,
                history=state["state"].chat_history,
                context=state["state"].retrieval_result.chunks
            )

            # 2. 執行 LLM 串流呼叫
            full_answer = ""
            async for chunk in self.llm_gateway.stream_completion(prompt_bundle):
                # 3. 輸出串流 token
                token_msg = ProgressMessage(type="token", phase=3, content=chunk.text)
                state["messages"].append(token_msg)
                full_answer += chunk.text

            # 更新狀態
            state["state"].final_answer = full_answer
            return state

        # ----- OPMP Phase 4 & 5: Post-processing & Validation -----
        def post_processing_node(state: RAGState) -> RAGState:

```

```

"""Phase 4 & 5: 內容後處理、引用映射和 Markdown 驗證"""

# Phase 4: Citation Mapping
progress_msg_4 = ProgressMessage(phase=4, message="Mapping citations and post-processing complete. Response ready.")
state["messages"].append(progress_msg_4)
# 模擬 citation mapping 邏輯 (將 插入到答案中)
final_answer_with_citations = self.retrieval_svc.map_citations(
    answer=state["state"].final_answer,
    citations=state["state"].retrieval_result.citations
)

# Phase 5: Final Validation
progress_msg_5 = ProgressMessage(phase=5, message="Final Markdown validation complete. Response ready.")
state["messages"].append(progress_msg_5)

# 發送最終內容
final_msg = ProgressMessage(type="final", phase=5, content=final_answer_with_citations)
state["messages"].append(final_msg)

# 更新狀態
state["state"].final_answer = final_answer_with_citations
return state

# ----- 建立 Graph 節點和邊 -----
workflow.add_node("query_analysis", query_analysis_node) # Phase 1
workflow.add_node("retrieval", retrieval_node) # Phase 2
workflow.add_node("synthesis", synthesis_node) # Phase 3
workflow.add_node("post_processing", post_processing_node) # Phase 4 & 5

# 定義邊 (順序執行)
workflow.add_edge(START, "query_analysis")
workflow.add_edge("query_analysis", "retrieval")
workflow.add_edge("retrieval", "synthesis")
workflow.add_edge("synthesis", "post_processing")
workflow.add_edge("post_processing", END)

return workflow.compile()

async def execute_rag(self, initial_state: ChatState):
    """執行 LangGraph，並以 ProgressMessage generator 形式輸出"""
    # 建立初始 LangGraph 狀態
    initial_rag_state: RAGState = {"state": initial_state, "messages": []}

    # 執行 Graph
    async for step in self.graph.astream(initial_rag_state):
        # 每次循環，從 step 中提取新的 ProgressMessage
        if "messages" in step and step["messages"]:
            new_messages = step["messages"]

            # 僅產出最新的訊息
            for msg in new_messages:
                yield msg.model_dump_json() # 輸出為 JSON 字串

            # 清空已發送的訊息，避免重複
            step["messages"] = []

        # 更新 LangGraph 內部狀態
        # ... (這裡需要處理 LangGraph state 的更新邏輯，以確保狀態在步驟間正確傳遞)

    # 最終更新 StateTransitionService (如果需要持久化)
    # self.state_svc.update_chat_history(initial_state.query, initial_state.final_answer)

# 為了讓骨架程式碼能運行，我們需要定義佔位服務
class InputDataHandleService:
    async def process_pdf(self, file_content: bytes, filename: str) -> str:
        # 這裡應該是 PDF 解析、分塊和嵌入邏輯
        return f"File {filename} processed."

class RetrievalService:
    async def parallel_retrieve(self, query: str) -> RetrievalResult:

```

```

        # 模擬異步檢索 (Phase 2)
        await asyncio.sleep(0.5)
        return RetrievalResult(
            chunks=["Chunk 1: ...", "Chunk 2: ..."],
            citations=[Citation(source_id="doc1", content_snippet="...", page=1)],
        )

    def map_citations(self, answer: str, citations: List[Citation]) -> str:
        # 模擬 Phase 4
        return f"{answer}"

class PromptService:
    def rewrite_query(self, query: str) -> str:
        # 模擬 Phase 1
        return f"Rewritten Query: {query}"

    def build_rag_prompt(self, query, history, context) -> str:
        return "RAG Prompt..."

class LLMGateway:
    async def stream_completion(self, prompt: str):
        # 模擬 Phase 3 LLM 串流
        mock_response = "The answer is synthesized from the retrieved context."
        for word in mock_response.split():
            await asyncio.sleep(0.05)
            # 模擬 LangChain BaseMessageChunk
            yield type('Chunk', (object,), {'text': word + " "})()
        await asyncio.sleep(0.05)
        yield type('Chunk', (object,), {'text': ""})() # 結束訊號

class StateTransitionService:
    def get_initial_state(self, session_id: str) -> ChatState:
        return ChatState(queriv="Initial Test". chat historv=[])

```

Python

步驟三：Presentation Layer - FastAPI SSE 實作

使用 FastAPI 的 `StreamingResponse` 和 `EventSourceResponse` 來實現 Server-Sent Events (SSE) 串流。

presentation/http_stream.py 骨架程式碼

```

from fastapi import APIRouter, Depends, HTTPException, Body
from fastapi.responses import StreamingResponse
import json
from domain.models import ChatState, ProgressMessage
from services.core_logic import RAGGraphExecutor, RetrievalService, PromptService, LLMGateway, StateTransitionService
import asyncio

router = APIRouter()

# 模擬依賴注入 (Dependency Injection)
def get_rag_executor():
    # 實際部署中，這些服務應該是 Singleton
    retrieval_svc = RetrievalService()
    prompt_svc = PromptService()
    llm_gateway = LLMGateway()
    return RAGGraphExecutor(retrieval_svc, prompt_svc, llm_gateway)

def get_state_svc():
    return StateTransitionService()

# 範例輸入模型
class ChatRequest(BaseModel):
    session_id: str
    user_query: str

```



```

@router.post("/chat/stream", tags=["Chat"])
async def chat_stream(
    request: ChatRequest = Body(...),
    executor: RAGGraphExecutor = Depends(get_rag_executor),
    state_svc: StateTransitionService = Depends(get_state_svc)
):
    """
    執行 RAG 流程並以 OPMP 五階段的 SSE 串流輸出。
    """

    # 1. 取得當前會話狀態 (StateTransitionService)
    # 實際應用中，會從 StateTransitionService 獲取 Session ID 對應的 ChatState
    initial_state = state_svc.get_initial_state(request.session_id)
    initial_state.query = request.user_query

    # 2. 啟動 LangGraph 執行器
    generator = executor.execute_rag(initial_state)

    async def event_generator():
        try:
            async for message_json in generator:
                # 每個 message_json 已經是 ProgressMessage 的 JSON 字串

                # 3. 實作 SSE 格式 (data: <JSON string>\n\n)
                yield f"data: {message_json}\n\n"

                # 檢查是否為最終訊息，如果是則結束串流
                message_dict = json.loads(message_json)
                if message_dict.get("type") in ["final", "error"]:
                    break
        except Exception as e:
            # 4. 錯誤處理 (Error Handling)
            error_msg = ProgressMessage(type="error", phase=5, message=f"Critical Error: {str(e)}", content="")
            yield f"data: {error_msg}\n\n"

    # 使用 StreamingResponse 確保異步 generator 被正確處理為串流
    # 設置 media_type 為 text/event-stream (SSE 標準)

```

Python

步驟四：主程式整合 (main.py)

最後將所有服務和路由組合成一個可運行的 FastAPI 應用程式。

main.py 骨架程式碼

```

from fastapi import FastAPI
from presentation.http_stream import router as stream_router
from starlette.middleware.cors import CORSMiddleware
import logging

logging.basicConfig(level=logging.INFO)

def build_app() -> FastAPI:
    """應用程式建構函數"""
    app = FastAPI(
        title="RAG OPMP Streaming System",
        version="0.3.0",
        description="Modular RAG system with LangGraph and Optimistic Progressive Markdown Parsing (OPMP)"
    )

    # 註冊 CORS 中間件 (請根據 S4 建議修改 allow_origins)
    app.add_middleware(
        CORSMiddleware,
        allow_origins=["*"]
    )

```

Python

```
        allow_methods=["*"],
        allow_headers=["*"],
    )

    # 註冊 Presentation Layer 路由
    app.include_router(stream_router, prefix="/api/v1")

    @app.get("/")
    async def root():
        return {"message": "RAG System Operational"}

    return app

app = build_app()

if __name__ == "__main__":
    import uvicorn
    # uvicorn main:app --reload
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

程式碼架構與步驟說明總結

- OPMP DTOs (Domain Models):** 定義了 `ProgressMessage` 作為所有跨層級通訊和前端串流的標準載體，確保每個階段的消息都能被一致地處理。
- LangGraph as Core Logic:** 使用 `RAGGraphExecutor` 將 OPMP 五階段流程（Query Analysis, Retrieval, Synthesis, Post-processing, Evaluation）集成到 LangGraph 的循序工作流中。這提供了**可視化、可測試和可修改**的 RAG 流程編排。
- Asynchronous Streaming:** `RAGGraphExecutor.execute_rag` 方法使用 `async for` 迭代 LangGraph 的 `astream` 輸出，將 Phase 1, 2, 4, 5 的進度，以及 Phase 3 的 LLM Token) 轉化為 `ProgressMessage` 產出。
- FastAPI SSE:** `http_stream.py` 中的 `/chat/stream` 端點使用 `StreamingResponse`，將 LangGraph 產生的 JSON 訊息流格式（`data: <JSON string>\n\n`），實現漸進式串流輸出。
- 解耦與擴展性:** 每個 OPMP 階段都是一個獨立的 LangGraph 節點，它們只依賴於其他服務（`RetrievalService`, `LLMGateway`）。我們可以輕鬆地在 `RetrievalService` 中引入並行檢索、在 `LLMGateway` 中切換 Ollama/vLLM，或在 `post_processing` 中增加 Markdown 驗證，而不會影響核心的 RAG 流程結構。

請

<> 程式夥伴

這是一個結構清晰、遵循您三層架構要求（LLM Provider, Services, Presentation）並整合了 **OPMP 漸進式串流** 的 FastAPI 專案。

OPMP RAG 系統檔案結構 (FastAPI + LangGraph)

```
.
├── main.py                # 應用程式入口 (FastAPI ASGI), 負責服務初始化與組裝
├── requirements.txt       # 專案所需依賴清單 (FastAPI, LangChain, LangGraph, Pydantic)
├── .env                  # 環境變數 (OLLAMA_BASE_URL, API_KEY, MODEL_NAME 等)
├── Dockerfile            # (可選) 部署用的容器化文件
├── src/
│   ├── __init__.py       # 模組化標記
│   ├──
│   │   # 1. Domain Layer (領域模型層)
│   │   ├── domain/
│   │   │   ├── __init__.py
│   │   │   └── models.py          # Pydantic 模型定義: ChatState, ProgressMessage (OPMP DTO), Citation,
│   │   └──
│   │       # 2. LLM Provider Layer (LLM 供應商層)
│   │       ├── llm_providers/
│   │       │   ├── __init__.py
│   │       │   └── openai_compat.py      # 統一的 OpenAI-Compatible API 實作 (用於 Ollama, vLLM, llama.cpp 等)
│   │       └──
│   │           # 3. Services Layer (核心業務邏輯層)
│   │           └── services/
```

```
├── __init__.py
├── core_logic.py           # CoreLogicService 核心: 包含 RAGGraphExecutor, 實現 OPMP 五階段編排
├── llm_gateway.py         # LLMGateway: LLM 呼叫抽象層, 負責串接 llm_providers
├── prompt_service.py      # PromptService: 處理所有提示模板 (Query Rewriting, RAG Context)
├── retrieval_service.py   # RetrievalService: 處理 Phase 2 檢索 (並行多源檢索、Reranking、Citation)
├── input_data_service.py  # InputDataHandleService: 處理 PDF 上傳、分塊 (Parent-Child, Semantic)
└── state_transition_service.py # StateTransitionService: 處理會話狀態 (ChatState) 的存取與持久化 (e.

# 4. Presentation Layer (呈現層)
└── presentation/
    ├── __init__.py
    └── http_stream.py      # FastAPI 路由實作 (處理 /chat/stream) 和 SSE 訊息格式化
```