

Gemini Chat

I design a general three-layer architecture:

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

now I need you to integrate the three-layer architecture to a python+FastAPI web application, which can let users to upload files and chat with the data in these files. It means the system need to extract, analyze and chunk data from these files. The most important is system's answers only and always from there uploaded files. Architecting my three-layer architectures into the general python+FastAPI web application architecture as you are a talent and proficient architect

- ◆ 這是一個非常好的三層架構設計，完全符合現代 RAG (Retrieval-Augmented Generation) 系統的設計理念。將您的架構整合到一個健壯的 Python + FastAPI 應用程式中，關鍵在於實現「關注點分離」(Separation of Concerns) 和「依賴注入」(Dependency Injection)。

作為您的架構師，我將把您的三層模型映射到一個可維護、可擴展的 FastAPI 專案結構中。

總體架構藍圖：RAG 流程整合

您的三層架構將被整合成兩個核心工作流程：**1. 數據提取 (Ingestion)** 和 **2. 聊天檢索 (Retrieval & Chat)**。

1. 數據提取流程 (File Upload)

[使用者] -> [Presentation Layer (Client)] -> FastAPI Endpoint (/upload) -> [Services Layer (InputDataHandleService)] -> [Services Layer (RetrievalService)] -> [Vector DB]

1. **使用者** 上傳一個或多個文件。
2. **FastAPI Endpoint** 接收 `UploadFile` 對象。
3. **InputDataHandleService** 負責：
 - 從文件中提取原始文本（例如使用 `pypdf` 處理 PDF）。
 - 將文本進行「分塊」(Chunking)。
 - 調用 `RetrievalService`。
4. **RetrievalService** 負責：
 - 為每個文本塊 (chunk) 產生嵌入 (Embeddings)。(這通常是透過一個嵌入模型完成，它可以是您 LLM Provider 的一部分，或是一個獨立的模型如 `sentence-transformers`)。
 - 將文本塊及其嵌入向量存儲到 **Vector Database**（例如 ChromaDB, FAISS, Milvus）。

2. 聊天檢索流程 (Chat)


```
├─ requirements.txt
├─ uvicorn_runner.py                # 用於啟動 uvicorn
```

整合您的三層架構

以下是將您的每一層具體實現到上述結構中的方法。

1. LLM Provider Layer (實現)

這一層在 FastAPI 專案中表現為一個**客戶端 (Client)**。我們在 `app/llm_provider/client.py` 中定義它。

- **統一 OpenAI 兼容 API**：這是關鍵。無論您後端是 Ollama、vLLM 還是 llama.cpp，它們都提供了 OpenAI 兼容的 `/v1/chat/completions` 端點。
- **實現**：我們將創建一個可重用的服務，它從 `app/core/config.py` 讀取 `LLM_PROVIDER_BASE_URL`（例如 `http://localhost:11434/v1` 或 `http://localhost:8000/v1`）。

`app/llm_provider/client.py` (範例):

Python



```
import httpx
from app.core.config import settings # 從 Pydantic SettingsManagement 讀取

class LLMProviderClient:
    def __init__(self):
        # 從 .env 讀取 LLM 服務的基礎 URL
        self.base_url = settings.LLM_PROVIDER_BASE_URL
        self.api_key = settings.LLM_PROVIDER_API_KEY or "ollama" # Ollama/vLLM 通常不需要 key

    async def get_chat_completion_stream(self, messages: list[dict], model: str = "default-model"):
        """
        請求 LLM 進行流式聊天補全
        """
        async with httpx.AsyncClient(timeout=60.0) as client:
            request_body = {
                "model": model,
                "messages": messages,
                "stream": True,
                # 這裡可以添加其他參數，如 temperature=0.0
            }

            headers = {"Authorization": f"Bearer {self.api_key}"}

            async with client.stream(
                "POST",
                f"{self.base_url}/chat/completions",
                json=request_body,
                headers=headers
            ) as response:
                response.raise_for_status() # 確保請求成功
                async for chunk in response.aiter_bytes():
                    yield chunk # 直接將原始 SSE 數據塊傳遞出去
```

2. Services Layer (實現)

這是您應用的**大腦**。我們使用 FastAPI 的**依賴注入 (Dependency Injection)** 系統來管理這些服務。

`app/services/retrieval_service.py` (範例):

Python



```
from some_vector_db import VectorStore # 假設使用 ChromaDB 或 FAISS
from some_embedding_model import EmbeddingModel # 假設使用 sentence-transformers
```

```

class RetrievalService:
    def __init__(self):
        # 理想情況下，這些應該是單例（Singleton）模式
        self.vector_store = VectorStore(path="./db/chroma_db")
        self.embedding_model = EmbeddingModel('all-MiniLM-L6-v2')

    async def add_chunks(self, file_id: str, chunks: list[str]):
        embeddings = self.embedding_model.embed(chunks)
        metadatas = [{ "file_id": file_id, "chunk_index": i } for i, _ in enumerate(chunks)]
        self.vector_store.add(ids=[f"{file_id}_{i}" for i in range(len(chunks))],
                               documents=chunks,
                               embeddings=embeddings,
                               metadatas=metadatas)

    async def query(self, query_text: str, file_ids: list[str], top_k: int = 5) -> list[str]:
        query_embedding = self.embedding_model.embed([query_text])[0]

        # 關鍵：只在使用者上傳的文件中搜索
        results = self.vector_store.query(
            query_embeddings=[query_embedding],
            n_results=top_k,
            where={ "file_id": { "$in": file_ids } } # 假設的過濾語法
        )
        return results['documents'][0] # 返回相關的文本塊

```

app/services/core_logic_service.py (範例):

Python



```

from fastapi import Depends
from app.llm_provider.client import LLMPProviderClient
from app.services.retrieval_service import RetrievalService
from app.services.prompt_service import PromptService
from app.services.state_transition_service import StateTransitionService

class CoreLogicService:
    def __init__(
        self,
        retrieval_service: RetrievalService = Depends(),
        prompt_service: PromptService = Depends(),
        llm_client: LLMPProviderClient = Depends(),
        state_service: StateTransitionService = Depends()
    ):
        # 透過 Depends(), FastAPI 會自動注入這些服務的實例
        self.retrieval_service = retrieval_service
        self.prompt_service = prompt_service
        self.llm_client = llm_client
        self.state_service = state_service

    async def generate_response_stream(self, user_id: str, chat_id: str, query: str):
        # 1. 獲取此聊天的文件範圍（假設已儲存）
        allowed_file_ids = await self.state_service.get_files_for_chat(chat_id)

        # 2. Retrieval: 檢索上下文
        context_chunks = await self.retrieval_service.query(query, allowed_file_ids)

        # 3. Prompt: 構建提示詞（這是實現「僅從文件回答」的關鍵）
        messages = self.prompt_service.build_rag_prompt(query, context_chunks)

        # 4. State Transition: (非同步) 保存用戶訊息
        await self.state_service.save_message(chat_id, "user", query)

        # 5. LLM Provider: 獲取流式回應
        # 我們將 LLM 的流直接傳遞回客戶端
        response_stream = self.llm_client.get_chat_completion_stream(messages)

        # 6. (可選) 在流結束後保存 AI 回應
        # full_response = await self.state_service.process_and_save_stream(chat_id, response_stream)

```

```
# return full_response

return response_stream
```

3. Presentation Layer (實現支持)

您的後端 (FastAPI) **不包含** Presentation Layer 本身（它是一個獨立的 Web 前端，例如 React/Vue）。您的後端職責是**提供 API 來支持這一層**。

- **Optimistic Progressive Markdown Parsing**：為了支持這一點，您的 `/chat` 端點**必須**使用 `StreamingResponse`。

`app/api/v1/endpoints/chat.py` (範例):

Python



```
from fastapi import APIRouter, Depends, UploadFile, File
from fastapi.responses import StreamingResponse
from app.services.core_logic_service import CoreLogicService
from app.services.input_data_handle_service import InputDataHandleService
from app.models.schemas import ChatRequest

router = APIRouter()

@router.post("/chat")
async def chat_endpoint(
    request: ChatRequest,
    core_service: CoreLogicService = Depends()
):
    """
    接收用戶查詢並以流式回應
    """
    stream = await core_service.generate_response_stream(
        user_id=request.user_id,
        chat_id=request.chat_id,
        query=request.query
    )

    # 關鍵：將 LLM 服務的流直接轉發給客戶端
    # 這使得 Markdown 能夠漸進式解析
    return StreamingResponse(stream, media_type="text/event-stream")

@router.post("/upload")
async def upload_files(
    files: list[UploadFile] = File(...),
    input_service: InputDataHandleService = Depends()
):
    """
    處理文件上傳、提取和分塊
    """
    file_ids = []
    for file in files:
        file_id = await input_service.process_and_store_file(file)
        file_ids.append(file_id)

    return {"message": "Files processed successfully", "file_ids": file_ids}
```

- **Data Export**：這只是 `StateTransitionService` 的一個簡單 API 端點，用於查詢數據庫並將聊天記錄格式化為 JSON 或 CSV。

關鍵約束：「僅從上傳的文件中回答」

這是整個架構中**最重要**的業務規則。它**不是**透過 LLM Provider 層（例如 vLLM）的配置來實現的，而是透過**服務層 (Services Layer)** 中的 `PromptService` 來**嚴格執行**的。

您的 `PromptService` 必須產生一個類似這樣的提示詞：

`app/services/prompt_service.py` (範例模板):

Python

```
class PromptService:
    def __init__(self):
        self.SYSTEM_PROMPT_TEMPLATE = """
        您是一個嚴謹的問答機器人。
        您必須「僅」使用下面提供的「上下文」來回答使用者的問題。
        絕對禁止使用您的任何內部知識或來自上下文之外的資訊。

        如果「上下文」中沒有足夠的資訊來回答問題，您必須明確回答：
        「根據您提供的文件，我找不到相關資訊。」

        請勿杜撰、猜測或推斷「上下文」中未明確說明的內容。

        ---
        [上下文]
        {context}
        ---
        """

    def build_rag_prompt(self, query: str, context_chunks: list[str]) -> list[dict]:
        # 將所有上下文塊合併為一個字符串
        context_str = "\n\n".join(context_chunks)

        system_prompt = self.SYSTEM_PROMPT_TEMPLATE.format(context=context_str)

        # 遵循 OpenAI 的消息格式
        return [
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": query}
        ]
```

透過這種方式，您不是在**請求** LLM 合作，而是在**指示**它在一個嚴格受限的沙盒（即您提供的上下文）內運作。這就是您實現「答案**僅**來自上傳文件」的核心機制。

Presentation Layer must implement using optimistic progress markdown parsing(OPMP) design



您說的沒錯。**Optimistic Progressive Markdown Parsing (OPMP)** 是現代聊天機器人 UI 中最關鍵的使用者體驗 (UX) 設計之一，它提供了「打字機」般的即時反饋感。

在我們的三層架構中，這個需求**完全**由 **Presentation Layer (前端)** 實現，而您的 **FastAPI 後端 (Services Layer + LLM Layer 接口)** 則扮演**啟用者 (Enabler)** 的角色。

這兩層的職責劃分非常明確：

1. 後端 (FastAPI) 的架構支持：啟用 OPMP

您的後端架構**不需要**解析 Markdown。它的**唯一且關鍵**的職責是：以**最低的延遲**，將 LLM Provider 層的令牌流 (token stream) **原封不動地、即時地傳遞給前端**。

您在 `chat.py` 端點中使用的 `StreamingResponse` 正是實現此目的的完美工具。

Python

```
# app/api/v1/endpoints/chat.py (回顧)

@router.post("/chat")
async def chat_endpoint(
    request: ChatRequest,
    core_service: CoreLogicService = Depends()
):
    # 1. core_service 調用 LLM Provider (Ollama/vLLM)
    # 並獲取一個異步生成器 (async generator)
    stream = await core_service.generate_response_stream(...)

    # 2. 關鍵：
    # FastAPI 不緩衝 (buffer) 回應。
    # LLM 產生一個 token，FastAPI 就將該 token
    # 封裝成一個 SSE (Server-Sent Events) 封包並立即發送。
    return StreamingResponse(stream, media_type="text/event-stream")
```

架構上的含義：

- **低延遲**：這是 OPMP 的前提。您的後端是一個「透明代理」，幾乎不增加流式傳輸的開銷。
- **協議**：`text/event-stream` (Server-Sent Events) 是前端實現 OPMP 的標準協議。前端將使用 `EventSource` API 或 `fetch` API 的 `ReadableStream` 來消費 (consume) 這個流。

2. 前端 (Presentation Layer) 的實現關鍵：實現 OPMP

您的前端 (例如 React, Vue, Svelte) 將接收這個 SSE 流。OPMP 的所有複雜邏輯都在這裡發生。

「Optimistic Progressive Markdown Parsing」可以分解為三個動作：

A. Progressive (漸進式)

前端**不會**等待流結束。它會訂閱 SSE 流，每當一個新的數據塊 (chunk) 到達時 (例如 `data: {"content": "..."}`)，它就會立即處理。

JavaScript



```
// 範例：前端 JavaScript (使用 fetch)
const response = await fetch('/api/v1/chat', { ... });
const reader = response.body.getReader();
const decoder = new TextDecoder();
let fullResponse = "";

while (true) {
    const { done, value } = await reader.read();
    if (done) break;

    const chunk = decoder.decode(value);
    // (這裡還需要解析 SSE 格式，簡單起見先假設 chunk 是純文本)

    // 1. 漸進式地累積文本
    fullResponse += chunk;

    // 2. 樂觀地 (Optimistic) 渲染
    renderMarkdown(fullResponse); // <-- 關鍵步驟
}
```

B. Optimistic (樂觀)

`renderMarkdown(fullResponse)` 這一步就是「樂觀」的體現。我們**假設** `fullResponse` 變數中不完整的 Markdown 字符串是可以被渲染的。

- **樂觀更新**：我們不等待一個完整的 Markdown 區塊 (例如整個代碼塊或列表) 完成，而是**立即**將當前的、不完整的 `fullResponse` 扔給 Markdown 解析器。

- **UI 狀態更新**：在 React 中，這意味著 `setState(prev => prev + newChunk)`。

C. Markdown Parsing (Markdown 解析)

這是最棘手的部分。一個常規的 Markdown 解析器 (Parser) 在遇到不完整的語法時（例如 ````python` 但沒有結尾的 `````）可能會崩潰或渲染錯誤。

您的 OPMP 解析器**必須具備容錯性 (Fault Tolerance)**：

1. **渲染已知內容**：它會渲染所有已完成的 Markdown 語法（例如一個完整的段落）。
2. **處理不完整語法**：
 - **範例 1 (代碼塊)**：當解析器遇到 ````python\nprint("hi` 時，它應該**樂觀地**開始渲染一個代碼塊。它不會因為沒有結尾的 ````` 而拒絕渲染。
 - **範例 2 (列表)**：當解析器遇到 `* 項目一\n*` 項 時，它應該渲染「項目一」列表，並將「* 項」暫時視為純文本，直到 `目二` 到達，然後**重新解析**並將其更正為第二個列表項。

總結：

您的三層架構設計是**完美契合** OPMP 需求的。

- **後端 (FastAPI)**：作為一個高效的、非緩衝的**流式數據提供者 (Stream Provider)**。
- **前端 (Presentation Layer)**：作為一個複雜的**流式數據消費者 (Stream Consumer)**，它內部包含一個能處理不完整輸入的、容錯的 Markdown 渲染引擎。

```
/DocAI
├── app/
│   ├── api/
│   │   ├── __init__.py
│   │   └── v1/
│   │       ├── __init__.py
│   │       ├── endpoints/
│   │       │   ├── __init__.py
│   │       │   ├── chat.py      # /chat (Streaming) 和 /upload 端點
│   │       │   └── export.py    # /export (Data Export) 端點
│   │       └── router.py       # API v1 總路由器
│   └──
├── core/
│   ├── __init__.py
│   └── config.py              # 應用程式配置 (例如 LLM API URL, DB 路徑)
├──
├── Provider
│   ├── llm_provider/         <--您的【Layer 1】客戶端
│   │   ├── __init__.py
│   │   └── client.py         # 統一的 OpenAI API 客戶端 (使用 httpx)
│   └──
├── services/                 <-- 您的【Layer 2】核心
│   ├── __init__.py
│   ├── core_logic_service.py # 協調器
│   ├── input_data_handle_service.py # 提取、分塊
│   ├── prompt_service.py    # 提示詞模板管理
│   ├── retrieval_service.py # 嵌入和向量檢索
│   └── state_transition_service.py # 聊天歷史
├──
└── models/
```



```

| | └─ schemas.py          # Pydantic 模型 (API 請求/回應)
| |
| └─ main.py              # FastAPI 應用實例和中間件
|
└─ .env                  # 環境變數 (API Keys, LLM_PROVIDER_URL)
└─ requirements.txt
└─ uvicorn_runner.py

```

I make a small change that create a Providers, and put llm_provider into it. The reason is that we may have some fundamental providers

◆ 這是一個**非常出色**的架構重構。

您提出了一個關鍵的觀點：`llm_provider` 本質上是一種**基礎設施接口 (Infrastructure Interface)**，而不是一個業務邏輯服務 (Business Logic Service)。

將其移至 `app/Providers/` 目錄是一個非常專業的舉動。這完全符合**依賴反轉原則 (Dependency Inversion Principle)** —— 您的「服務層」(Layer 2) 不應該依賴於具體的實現，而應該依賴於抽象的「提供者」(Providers)。

架構演進：Providers 層的真正潛力

您的這個改動，讓我們的架構變得更加清晰和可擴展。您說的「we may have some fundamental providers」完全正確。在我們的 RAG 系統中，至少還有兩個核心組件也屬於 `Providers` 層：

1. Embedding Provider (嵌入模型提供者):

- `RetrievalService` 需要將文本塊 (chunks) 和用戶查詢轉換為向量。
- 這個嵌入模型可以是一個本地的 `sentence-transformers`，也可以是一個 API (如 OpenAI Embedding API)。
- 將它抽象為 `EmbeddingProvider`，`RetrievalService` 就不需要關心它具體是哪裡來的。

2. Vector Store Provider (向量儲存提供者):

- `RetrievalService` 需要查詢向量數據庫 (ChromaDB, Milvus, Qdrant 等)。
- 連接、認證和查詢數據庫的邏輯應該被封裝在 `VectorStoreProvider` 中。

更新後的架構藍圖 (整合您的變更)

這是整合了您 `Providers` 目錄概念的、更健壯的專案結構：

Bash



```

/DocAI
├─ app/
│   ├── api/
│   │   └─ v1/
│   │       ├── endpoints/
│   │       └─ router.py
│   ├── core/
│   │   └─ config.py
│   ├── models/
│   │   └─ schemas.py
│   └─ Providers/                                <-- 您的【Layer 1】基礎設施接口
│       ├── __init__.py
│       ├── llm_provider/                        # (您的)
│       │   ├── __init__.py
│       │   └─ client.py                        # 統一的 OpenAI API 客戶端
│       ├── embedding_provider/                 # (新增)
│       │   └─ __init__.py

```

```

├── client.py          # 嵌入模型（例如 SentenceTransformer）
├── vector_store_provider/ # （新增）
│   ├── __init__.py
│   └── client.py      # 向量數據庫（例如 ChromaDB）
├── services/          <-- 您的【Layer 2】核心業務邏輯
│   ├── __init__.py
│   ├── core_logic_service.py # 協調器
│   ├── input_data_handle_service.py # 提取、分塊
│   ├── prompt_service.py
│   ├── retrieval_service.py  # <== 將使用 Providers
│   └── state_transition_service.py
├── main.py            # FastAPI 應用實例
├── .env
├── requirements.txt
└── uvicorn_runner.py

```

變更的連鎖反應：Services 層如何演進

這個變更最大的好處體現在 `Services` 層。以 `RetrievalService` 為例，它現在**不再需要知道**具體的數據庫或模型是什麼，它只需要**聲明它的依賴**。

`app/services/retrieval_service.py` (演進後的範例):

Python



```

from fastapi import Depends
from app.Providers.embedding_provider.client import EmbeddingProvider
from app.Providers.vector_store_provider.client import VectorStoreProvider

class RetrievalService:
    def __init__(
        self,
        # 依賴注入：從 Providers 層獲取客戶端實例
        embed_provider: EmbeddingProvider = Depends(EmbeddingProvider),
        vector_store: VectorStoreProvider = Depends(VectorStoreProvider)
    ):
        self.embed_provider = embed_provider
        self.vector_store = vector_store

    async def add_chunks(self, file_id: str, chunks: list[str]):
        # 1. 使用 EmbeddingProvider
        embeddings = self.embed_provider.embed_documents(chunks)

        metadatas = [{"file_id": file_id, "chunk_index": i} for i, _ in enumerate(chunks)]

        # 2. 使用 VectorStoreProvider
        await self.vector_store.add(
            ids=[f"{file_id}_{i}" for i in range(len(chunks))],
            documents=chunks,
            embeddings=embeddings,
            metadatas=metadatas
        )

    async def query(self, query_text: str, file_ids: list[str], top_k: int = 5) -> list[str]:
        # 1. 使用 EmbeddingProvider
        query_embedding = self.embed_provider.embed_query(query_text)

        filter_criteria = {"file_id": {"$in": file_ids}}

        # 2. 使用 VectorStoreProvider
        results = await self.vector_store.query(
            query_embeddings=[query_embedding],
            n_results=top_k,

```

```
        where=filter_criteria
    )
    return results['documents'][0]
```

結論

您的這項變更非常棒。

1. **清晰的邊界**： `Services` (Layer 2) 包含**業務邏輯**（如何協調檢索、提示和生成）。 `Providers` (Layer 1) 包含**基礎設施邏輯**（如何與 LLM、數據庫、模型 API 通信）。
2. **可測試性 (Testability)**：在單元測試中，我們可以輕易地模擬 (Mock) `Providers` 層的任何一個組件，來獨立測試 `Services` 層的邏輯。
3. **可插拔性 (Pluggability)**：明天如果您想從 `ChromaDB` 切換到 `Milvus`，您只需要修改 `VectorStoreProvider` 的實現， `RetrievalService` **完全不需要**改動任何一行代碼。

這是一個真正可維護和可擴展的架構。