EGOV - API - PIPELINE

План действий по интеграции FastAPI и Open Web UI (Pipeline)

Ниже представлен официальный план, пошаговая инструкция, а также пример кода пайплайна для Open Web UI. Предполагается, что у вас **уже есть** рабочее FastAPI-приложение (например, из файла `mgov-apiv4-rest.py`), которое обрабатывает запросы, генерирует эмбеддинги (через Zilliz или OpenAI), выполняет поиск по векторной базе (Zilliz/Milvus) и вызывает модель OpenAI или локальную LLM.

1. Общая концепция

1. **FastAPI**:
   - Содержит готовые эндпоинты (например, `/threads/...`) для API-запросов.
   - Хранит историю в Redis, подключается к Zilliz и OpenAI и обрабатывает запросы.
   - Обеспечивает «массовый» REST-доступ извне (другие сервисы, клиенты).
2. **Open Web UI**:
   - Даёт «чат-интерфейс» для тех, кто хочет использовать тот же функционал без написания REST-запросов.
   - Состоит из набора «пайплайнов» (Pipelines), которые можно собрать из «узлов» (Nodes), описанных в YAML или Python.
   - Осуществляет ту же логику (эмбеддинги, поиск, генерация ответа), но «визуализирует» её в браузере.
3. **Задача**:
   - Дать пользователям **два способа** работать с вашей логикой: через REST API и через интерфейс Open Web UI.
   - При этом **не дублировать** код, использовать **одни и те же** методы для эмбеддинга и поиска (Zilliz), и **один и тот же** LLM (OpenAI).

---

2. Шаги по интеграции

Шаг 1. Убедиться, что вся «ядровая» логика вынесена/готова

- В вашем FastAPI-приложении (например, `mgov-apiv4-rest.py`) уже есть код, который:
  1. Загружает (или подключается) к эмбеддинговой модели (либо OpenAI Embedding, либо локальной в transformers).
  2. Выполняет поиск в Zilliz (через REST или pymilvus).
  3. Вызывает OpenAI (ChatCompletion) или другую LLM.

Если **логика разбросана** по коду эндпоинтов, вы можете вынести основное в **отдельный модуль** – условно, `mgov_core.py`.
Например:

```
# mgov_core.py (пример)

import httpx, torch
# import pymilvus / requests для Zilliz
# ...

async def embed_text(text: str) -> list:
    """
    Генерирует эмбеддинг либо через OpenAI, либо через локальную модель transformers.
    """
    # ваш код
    pass

async def search_zilliz(embedding: list) -> list:
    """
    Поиск top-k документов в Zilliz
    """
    # ваш код
    pass

async def generate_answer(context: str, user_question: str) -> str:
    """
    Вызывает OpenAI ChatCompletion (или любую LLM).
    """
    # ваш код
    pass
```

А в `mgov-apiv4-rest.py` вы уже просто импортируете `embed_text`, `search_zilliz`, `generate_answer` и вызываете их.

> **Если** у вас уже всё работает прямо в `mgov-apiv4-rest.py`, и вы не хотите выделять модуль, можете оставить как есть. Но обычно удобнее иметь общий модуль, если планируете переиспользовать в разных местах.

Шаг 2. Подготовить «узлы» (Nodes) для Open Web UI

Создайте Python-файл, который будет виден движку Open Web UI. Например, `my_pipeline_nodes.py`.

Там определите функции – «узлы» пайплайна (Nodes). Каждая функция делает ровно одну задачу:

4. **Node: Генерация эмбеддинга**
5. **Node: Поиск в Zilliz**
6. **Node: Генерация ответа** (GPT/OpenAI)

Пример **упрощённого** кода (заготовка):

```
# my_pipeline_nodes.py

from typing import List
from mgov_core import embed_text, search_zilliz, generate_answer

async def node_embedding(user_text: str):
    """
    Превращает входную строку (user_text) в вектор (embedding).
    Возвращает словарь с ключом "embedding".
    """
    embedding = await embed_text(user_text)  # вызов вашей общей функции
    return {"embedding": embedding}

async def node_search(embedding: List[float]):
    """
    Выполняет поиск в Zilliz, возвращает список документов (search_results).
    """
    results = await search_zilliz(embedding)
    return {"search_results": results}

async def node_llm_answer(user_text: str, search_results: list):
    """
    Сформировать итоговый контекст, вызвать LLM и вернуть ответ.
    """
    # Превращаем search_results в строку (или JSON),
    # далее формируем prompt, вызываем generate_answer(...).
    context_text = ""
    for doc in search_results:
        context_text += f"{doc['name']}\n{doc['description']}\nLink:
```

```
{doc['link']}\n\n"
    final_answer = await generate_answer(context_text, user_text)
    return {"final_answer": final_answer}
```

Если у вас нет `mgov_core.py`, и вся логика в `mgov-apiv4-rest.py`, можно дублировать вызовы здесь напрямую. Но лучше придерживаться DRY (don't repeat yourself).

Шаг 3. Создать YAML-пайплайн для Open Web UI

Создайте файл, например, `zilliz_openai_pipeline.yaml` (или `.json`, смотря по формату). В нём опишите **узлы** (steps). Пример (упрощённо):

```yaml
version: 1
pipelines:
  - name: "zilliz_openai_pipeline"
    steps:
      - name: "UserInput"
        type: "input"

      - name: "Embedding"
        type: "python"
        script_path: "my_pipeline_nodes.py"
        function: "node_embedding"
        inputs:
          - from: "UserInput"
            param: "user_text"
        outputs:
          - "embedding"

      - name: "Search"
        type: "python"
        script_path: "my_pipeline_nodes.py"
        function: "node_search"
        inputs:
          - from: "Embedding"
            param: "embedding"
        outputs:
          - "search_results"

      - name: "LLM"
        type: "python"
        script_path: "my_pipeline_nodes.py"
        function: "node_llm_answer"
```

```
        inputs:
          - from: "UserInput"
            param: "user_text"
          - from: "Search"
            param: "search_results"
        outputs:
          - "final_answer"

      - name: "Output"
        type: "output"
        inputs:
          - from: "LLM"
            param: "final_answer"
```

Пояснения:

- `UserInput` – специальный узел (Node), который берёт ввод пользователя в UI.
- `Embedding` – наш узел, который зовёт `node_embedding(user_text)`. Возвращает `{"embedding": ...}`.
- `Search` – зовёт `node_search(embedding)`. Возвращает `{"search_results": ...}`.
- `LLM` – зовёт `node_llm_answer(user_text, search_results)`. Возвращает `{"final_answer": "..."}`
- `Output` – выводит результат в UI.

В итоге при запуске Open Web UI пользователь вводит вопрос → пайплайн последовательно вызывает эти узлы → показывает ответ.

Шаг 4. Подготовка окружения (env, зависимости)

7. **Env-переменные**:
   - `OPENAI_API_KEY`, `ZILLIZ_URI`, `ZILLIZ_TOKEN`, `REDIS_URL` и т.д.
   - Убедитесь, что Open Web UI и ваше FastAPI-приложение видят одинаковые `.env` или переменные окружения.
8. **Зависимости** (requirements):
   - Установить `fastapi`, `requests`, `httpx`, `transformers`, `pydantic`, `uvicorn`, `redis`, `asyncpg`, `pymilvus` (или REST).
   - Убедиться, что **Open Web UI** тоже имеет доступ к тем же библиотекам.

Шаг 5. Запуск (Deployment)

9. **FastAPI**:
    - Запускаете ваш `mgov-apiv4-rest.py` (через `uvicorn`, Docker или на сервере).
    - Проверяете, что эндпоинты `/threads/...` работают, Redis подключён, Zilliz загружен, OpenAI отвечает.
10. **Open Web UI**:
    - Клонируете/устанавливаете Open Web UI (см. документы Open Web UI).
    - Кладёте файлы `my_pipeline_nodes.py` и `zilliz_openai_pipeline.yaml` в папку, где Open Web UI ищет пайплайны (часто `pipelines/` или `plugins/`).
    - Запускаете Open Web UI (это может быть отдельный процесс).
    - На экране Open Web UI (в браузере) видите ваш пайплайн «zilliz_openai_pipeline».
    - Подаёте вопрос – пайплайн должен корректно вывести ответ, используя Zilliz + OpenAI.

Шаг 6. Разграничение хранения истории

- Если хотите, чтобы история в Open Web UI **не конфликтовала** с историей FastAPI, можно:
    - Использовать **отдельный префикс** ключей в Redis: `chat_history:ui:{user_id}` vs. `chat_history:api:{user_id}`.
    - В пайплайне (через Python-узел) сохранять/читать историю из Redis по своим ключам.
    - Либо не хранить историю в Redis вообще (Open Web UI может хранить короткую «память» в себе самом, зависит от настроек).

---

3. Пример итогового кода пайплайна (Python-узлы)

Ниже — сводный пример файла `my_pipeline_nodes.py`. (Предполагаем, что ваш общий функционал вынесен в `mgov_core.py`.)

```python
# my_pipeline_nodes.py
import asyncio
from mgov_core import embed_text, search_zilliz, generate_answer

async def node_embedding(user_text: str):
    """
    1) Получаем эмбеддинг (через OpenAI Embedding или локальную модель).
    """
    embedding = await embed_text(user_text)
    return {"embedding": embedding}
```

```
async def node_search(embedding):
    """
    1) Ищем top-k документов в Zilliz/Milvus (или fallback в Postgres),
возвращаем список результатов.
    """
    results = await search_zilliz(embedding)
    return {"search_results": results}

async def node_llm_answer(user_text: str, search_results: list):
    """
    1) Формируем окончательный ответ, вызывая модель (OpenAI ChatCompletion).
    Подготавливаем контекст из search_results.
    """
    if not search_results:
        context_text = "No relevant services found."
    else:
        # форматируем контекст из списка результатов
        context_lines = []
        for doc in search_results:
            name = doc.get("name", "")
            desc = doc.get("description", "")
            link = doc.get("link", "")
            context_lines.append(f"Service: {name}\nDescription: {desc}\nLink:
{link}\n")
        context_text = "\n".join(context_lines)

    final_answer = await generate_answer(context_text, user_text)
    return {"final_answer": final_answer}
```

Примечание:

- `embed_text`, `search_zilliz`, `generate_answer` – функции, которые вы уже используете в FastAPI.
- Все «await»-вызовы зависят от того, асинхронные ли у вас функции. Если у вас синхронная логика, уберите `async`.

---

4. Файл-конфиг пайплайна (YAML)

```
version: 1
pipelines:
  - name: "zilliz_openai_pipeline"
```

```yaml
  steps:
    - name: "UserInput"
      type: "input"

    - name: "Embedding"
      type: "python"
      script_path: "my_pipeline_nodes.py"
      function: "node_embedding"
      inputs:
        - from: "UserInput"
          param: "user_text"
      outputs:
        - "embedding"

    - name: "Search"
      type: "python"
      script_path: "my_pipeline_nodes.py"
      function: "node_search"
      inputs:
        - from: "Embedding"
          param: "embedding"
      outputs:
        - "search_results"

    - name: "LLM"
      type: "python"
      script_path: "my_pipeline_nodes.py"
      function: "node_llm_answer"
      inputs:
        - from: "UserInput"
          param: "user_text"
        - from: "Search"
          param: "search_results"
      outputs:
        - "final_answer"

    - name: "Output"
      type: "output"
      inputs:
        - from: "LLM"
          param: "final_answer"
```

> Поместите этот YAML в каталог, который сканирует Open Web UI (обычно `pipelines/`).

5. Кратко о деплое (облако)

11. **Dockerfile** (примерный), в котором устанавливаем Python, FastAPI, Open Web UI.
12. **docker-compose** или **Kubernetes** для запуска:
    - **Сервис 1**: FastAPI (при необходимости в несколько реплик),
    - **Сервис 2**: Open Web UI + пайплайны,
    - **Redis** (может быть Managed),
    - **Zilliz** (Managed Cloud или локальный).
13. Указываем одинаковые переменные окружения (OpenAI key, ZILLIZ URI/token, REDIS url).
14. Проверяем, что и FastAPI, и Open Web UI корректно видят сервисы друг друга.

---

6. Заключение

Таким образом, **завершить интеграцию** и **запустить** ваш API и Pipeline нужно:

15. **Убедиться**, что всё, что касается эмбеддингов, поиска и вызова LLM, доступно в одном месте (общий модуль или «склеить» логику).
16. **Создать** Python-«узлы» (Nodes) для Open Web UI, либо же просто «звонить» из Open Web UI к вашему REST (оба пути легитимны).
17. **Настроить** YAML-пайплайн, который связывает узлы: Input → Embedding → Search → LLM → Output.
18. **Запустить** (deploy) Open Web UI в облаке, рядом с вашим FastAPI, с нужными env-переменными, зависимостями и доступом к Redis/Zilliz.
19. При необходимости, **использовать** общий Redis, чтобы хранить/разграничивать историю разговоров.

Результат:

- Вы имеете «старый» (или «основной») REST API в FastAPI (для массовых запросов)
- И **дополнительный** визуальный интерфейс Open Web UI, который даёт чат-режим к той же самой логике.

Оба компонента могут работать параллельно и масштабироваться независимо, предоставляя единый функционал для разных типов клиентов.