SK네트웍스 **Family AI** 과정 **15**기

모델배포 개발된 LLM 연동 웹 애플리케이션

산출물 단계	모델배포
평가 산출물	LLM 연동 웹 애플리케이션
제출 일자	2026/02/04
깃허브 경로	https://github.com/Scentence/Scentence
작성 팀원	신지섭

1 프로젝트 개요

- 프로젝트명: Scentence
- 작성일: 2026/02/03
- 작성자: 신지섭
- 버전: v1.2
- 승인관리자: 신지섭

2. 시스템 개요

- 목표: 이 애플리케이션은 관계형 데이터베이스, 벡터 데이터베이스와 LLM을 연동하여
- 기능:
    - 관계형 데이터베이스와 LLM 연동
    - 벡터 데이터베이스와 LLM 연동
    - 효율적인 프롬프트 최적화
    - 안전한 API 키 관리 및 예외 처리

3. 시스템 아키텍처

- 백엔드:
    - 언어: Python

- 프레임워크: Fast API
- LLM 연동: GPT-4o-mini / GPT-4.1 / GPT 5.2
- 벡터 데이터베이스: pgvector
- 데이터베이스: PostgreSQL
- 기술 스택:
  - API 관리: Fast API
  - 보안
    - 인증(Authentication): NextAuth + Kakao OAuth(2.0)
    - 비밀번호 관리: 비밀번호 생성시에 PBKDF2 해시로 검증
    - 인가(Authorization): DB의 role_type(USER/ADMIN) 기반으로 권한을 구분합니다.
    - API 보호: Fast API CORS

- 프론트엔드:
  - 프레임워크: Next.js 16.1.1 (React 19), TypeScript
  - 인증: NextAuth 기반 로그인(카카오 Provider 연동), 세션 기반 사용자 상태 관리
  - 상호작용/통신: fetch 및 axios를 통한 비동기 API 통신
    - Next.js rewrites로 /api/* 요청을 백엔드(FastAPI) 및 각 서비스로 프록시
  - UI/스타일링: TailwindCSS(v4) 기반 스타일링 + 전역 CSS(globals.css)
    - 아이콘: lucide-react
    - 모션/인터랙션: framer-motion
    - 3D/비주얼: @react-three/fiber, @react-three/drei, postprocessing
    - 시각화: vis-network, vis-data
    - 상태 관리: React Hooks + NextAuth SessionProvider (Providers)

4. LLM 연동 및 벡터 데이터베이스 구현

4.1 관계형 데이터베이스와 LLM 연동

- 연동 목적: LLM이 PostgreSQL에 저장된 향수/리뷰/메타데이터를 기반으로 그라운딩된 추천/설명 응답을 생성하게 하기 위해 연동합니다.
- 프롬프트: LLM에 제공되는 프롬프트는 사용자의 요청을 분류하고 해당하는 향수에 대한 데이터를 이용하여 정확한 답변을 하도록 프롬프팅 되어있습니다. DB에서 검색된 결과가 없을 때에는 검색 결과가 없음을 알리고 다른 방향의 요청을 유도하도록 설계되었습니다.

예시 코드**: LLM**과 벡터 데이터베이스 연동

 # DB연결 관련 함수

```
def get_db_connection()
def release_db_connection(conn)
def get_recom_db_connection()
def release_recom_db_connection(conn)
def get_member_db_connection()
def release_member_db_connection(conn)
```

추천 로직에서 향수 검색 과정

```
plan = await plan_llm.ainvoke([...], config={"tags": ["internal_helper"]})  # LLM이 hard/strategy filters 생성
candidates, _ = await smart_search_with_retry_async(
    plan.hard_filters.model_dump(exclude_none=True),
    plan.strategy_filters.model_dump(exclude_none=True),
    exclude_ids=exclude_ids,
    query_text=plan.reason,
    rank_mode=rank_mode,
)
```

```
# smart_search_with_retry_async 내부 (/backend/agent/graph.py 211~)
results = await advanced_perfume_search_tool.ainvoke({
  "hard_filters": sanitized_hard,
  "strategy_filters": sanitized_strategy,
  "exclude_ids": exclude_ids,
  "query_text": plan.reason,
  "rank_mode": rank_mode,
})
```

advanced_perfume_search_tool 내부 (/backend/agent/tools.py 54~)

raw = await asyncio.to_thread(search_perfumes, ..., limit=20) # RDB SELECT

top = await rerank_perfumes_async(raw, plan.reason, top_k=5)

return top

정보 조회에서의 **RDB** 조회 과정

LLM이 브랜드/이름 JSON으로 정리

norm = NORMALIZER_LLM.invoke(normalization_prompt).content

그 다음 **RDB**에서 상세조회

cur.execute(SQL_SELECT, params) # RDB SELECT

return json.dumps(dict(result), ensure_ascii=False) # 이 JSON을 상위 그래프/LLM이 받아 설명

- 프롬프트 최적화:
    - 기본적으로 검색결과, error 발생 여부에 따라서 분기 처리되어서 할루시네이션을 방지하지만 예상치 못한 상황에서 할루시네이션을 발생시키기 않도록 프롬프트에 할루시네이션 관련 룰을 추가했습니다.
    - 출력 형식, 설명 과정, 예시 표현 등의 세부적인 표현을 프롬프트에 추가하여 의도한 출력이 나오도록 프롬프팅했습니다.
    - LLM이 요구 사항에 맞는 최적의 결과를 도출할 수 있도록 주기적으로 테스트하고 최적화합니다.

4.2 벡터 데이터베이스와 LLM 연동

- 연동 목적: LLM이 벡터 DB(현재 레포에서는 PostgreSQL + pgvector) 를 통해 리뷰/노트 임베딩 테이블에서 의미적으로 가까운 데이터를 찾아(유사도 검색), 그 결과를 근거로 그라운딩된 추천/설명을 생성하기 위해서 연동합니다.

- 프롬프트: 벡터 검색은 “정답 생성”이 아니라 근거 확보 단계로 쓰입니다. 추천 리랭킹에서는 사용자의 의도/전략 텍스트를 검색 친화적인 형태로 변환하고 리뷰데이터와 비교하여 검색 결과를 리랭킹 하도록 설계 되어있고 노트/향수 정보 조회에서는 사용자 입력을 DB에 맞는 정규화된 키워드(JSON) 로 변환하여 RDB에서 조회가능하도록 하는 역할을 하도록 설계되어있습니다.

예시 코드**: LLM**과 벡터 데이터베이스 연동

```
# 임베딩 생성
async def get_embedding_async(text: str) -> List[float]:
    response = await async_client.embeddings.create(
        input=text.replace("\n", " "),
        model="text-embedding-3-small",
    )
    return response.data[0].embedding


# 벡터 DB 검색 함수
def lookup_note_by_vector(keyword: str) -> List[str]:
    query_vector = get_embedding(keyword)
    sql = "SELECT note FROM TB_NOTE_EMBEDDING_M ORDER BY embedding <=>
%s::vector LIMIT 10"
    cur.execute(sql, (query_vector,))
    return [r[0] for r in cur.fetchall()]


#결과 리랭킹 로직
translation = await async_client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "system", "content": system_prompt},
              {"role": "user", "content": query_text}],
    temperature=0,
)
stylized_query = translation.choices[0].message.content.strip()
query_vector = await get_embedding_async(stylized_query)
sql = f"""
SELECT m.perfume_id,
       MAX(1 - (e.embedding <=> %s::vector)) as similarity_score,
```

```
    (ARRAY_AGG(m.content ORDER BY (e.embedding <=> %s::vector) ASC))[1] as
best_review

FROM TB_PERFUME_REVIEW_M m

JOIN TB_REVIEW_EMBEDDING_M e ON m.review_id = e.review_id

WHERE m.perfume_id IN ({placeholders})

GROUP BY m.perfume_id

ORDER BY similarity_score DESC

"""

cur.execute(sql, [query_vector, query_vector] + candidate_ids)
```

5. 예외 처리 및 보안

5.1 예외 처리

- 예상치 못한 상황에 대한 예외 처리:
    - API 요청이 실패하거나 기타 오류에 대해서 별도의 로직을 적용하여 에러 처리 노드로 라우팅합니다.
    - 검색 결과가 없으면 에러와 유사하게 별도의 노드로 라우팅 처리하여 할루시네이션을 방지 합니다.

```
try:
   decision = SMART_LLM.with_structured_output(RoutingDecision).invoke(messages)
   next_step = decision.next_step
   return {"next_step": next_step}
except Exception as e:
   print(f"... {e} -> 기본값 Writer로 이동")
   return {"next_step": "writer"}
```

- 보안:
    - LLM API 키는 환경 변수를 사용하여 코드에서 직접 노출되지 않도록 합니다.
    - DB연결에 관한 키들도 환경변수로 사용하여 코드에서 직접 노출되지 않도록 합니다.

```
# .env 예시
```

```
OPENAI_API_KEY="your-openai-api-key"
DB_HOST="your host address"
DB_PORT="your host port"
DB_USER="username"
DB_PASSWORD="password"

# 코드 사용 예시
import os
openai.api_key = os.getenv("OPENAI_API_KEY")
DB_CONFIG = {
    "dbname": os.getenv("DB_NAME", "perfume_db"),
    "user": os.getenv("DB_USER", "scentence"),
    "password": os.getenv("DB_PASSWORD", "scentence"),
    "host": os.getenv("DB_HOST", "host.docker.internal"),
    "port": os.getenv("DB_PORT", "5432"),
}
```

5.2 보안 관리

- 보안에 민감한 정보 (예: API 키, 데이터베이스 연결 정보 등)는 모두 환경 변수나 비밀 관리 시스템을 사용하여 보호됩니다.
- 환경 변수를 사용하여 코드에 민감한 정보를 노출시키지 않도록 관리합니다.

6. 코드 모듈화 및 주석

6.1 모듈화

- 코드는 각 기능을 별도의 모듈로 분리하여 유지보수를 용이하게 합니다.
    - 예시: graph.py, tools.py, schemas.py 등 역할 별로 분리

6.2 주석

- 주석 예시:

```
# ================================================================
# Helper: 어코드 설명 주입기 (Enricher)
```

```
# ================================================================
def enrich_accord_description(text: str) -> str:
    """
    텍스트 내에 'Woody', 'Citrus' 같은 어코드 키워드가 있으면
    CSV 사전의 묘사를 괄호 안에 넣어 풍성하게 만듭니다.
    """
    if not text:
        return ""

    enriched_text = text

    # 일반적인 어코드 키워드 목록 (CSV에 있는 것들)
    common_accords = [
        "Animal", "Aquatic", "Chypre", "Citrus", "Creamy", "Earthy",
        "Floral", "Fougère", "Fresh", "Fruity", "Gourmand", "Green",
        "Leathery", "Oriental", "Powdery", "Resinous", "Smoky", "Spicy",
        "Sweet", "Synthetic", "Woody"
    ]
    # ================================================================
# Helper: 어코드 설명 주입기 (Enricher)
# ================================================================
def enrich_accord_description(text: str) -> str:
    """
    텍스트 내에 'Woody', 'Citrus' 같은 어코드 키워드가 있으면
    CSV 사전의 묘사를 괄호 안에 넣어 풍성하게 만듭니다.
    """
    if not text:
```

```
    return ""
enriched_text = text

# 일반적인 어코드 키워드 목록 (CSV에 있는 것들)
common_accords = [
    "Animal", "Aquatic", "Chypre", "Citrus", "Creamy", "Earthy",
    "Floral", "Fougère", "Fresh", "Fruity", "Gourmand", "Green",
    "Leathery", "Oriental", "Powdery", "Resinous", "Smoky", "Spicy",
    "Sweet", "Synthetic", "Woody"
]
```