법률 문서 기반 질의응답 RAG 시스템

기본 라이브러리

```
# 정규 표현식 모듈
# LLM이 내뱉는 답변에서 특정 패턴을 찾아내거나 텍스트 정제
import re
import os, json
from glob import glob
# 멀티라인 문자열의 공통도니 앞 공백 제거
# 파이썬 코드 간 프롬프트 작성 시, 코드 가독성을 위해 넣은 들여쓰기가 프롬프트에 포함되지 않게 정리
from textwrap import dedent
# 복잡한 딕셔너리나 리스트 구조를 보기 좋게 줄바꿈해 출력
# state 객체가 복잡해졌을 때, 현재 데이터가 어떻게 흘러가는지 터미널에서 파악하기 위해 사용
from pprint import pprint
# 범용 고유 식별자 생성
# 각 사용자가 세션을 구분하거나, DB에 저장할 각 노드의 실행 기록에 고유 ID 부여
import uuid

import warnings
warnings.filterwarnings("ignore")
```

법률 문서를 로드해 벡터 저장소에 저장

```
def parse_law(law_text):
    # 서문 분리
    # '^'로 시작하여 '제1장' 또는 '제1조' 직전까지의 모든 텍스트를 탐색
    preamble_pattern = r'^(.*?)(?=제1장|제1조)'
    preamble = re.search(preamble_pattern, law_text, re.DOT
```

```
ALL)
    if preamble:
        preamble = preamble.group(1).strip()

    # 장 분리
    # '제X장' 형식의 제목과 그 뒤에 오는 모든 조항을 하나의 그룹화
    chapter_pattern = r'(제\d+장\s+.+?)\n((?:제\d+조(?:의\d
+)?(?:\(\w+\))?.*?)(?=제\d+장|부칙|$))'
    chapters = re.findall(chapter_pattern, law_text, re.DOT
ALL)

    # 부칙 분리
    # '부칙'으로 시작하는 모든 텍스트를 탐색
    appendix_pattern = r'(부칙.*)'
    appendix = re.search(appendix_pattern, law_text, re.DOT
ALL)
    if appendix:
        appendix = appendix.group(1)

    # 파싱 결과를 저장할 딕셔너리 초기화
    parsed_law = {'서문': preamble, '장': {}, '부칙': append
ix}

    # 각 장 내에서 조 분리
    for chapter_title, chapter_content in chapters:
        # 조 분리 패턴
        # 1. '제X조'로 시작 ('제X조의Y' 형식도 가능)
        # 2. 조 번호 뒤에 반드시 '(항목명)' 형식의 제목이 와야 함
        # 3. 다음 조가 시작되기 전까지 또는 문서의 끝까지의 모든 내
용을 포함
        article_pattern = r'(제\d+조(?:의\d+)?\s*\([^)]+\).
*?)(?=제\d+조(?:의\d+)?\s*\([^)]+\)|$)'

        # 정규표현식을 이용해 모든 조항을 탐색
        articles = re.findall(article_pattern, chapter_cont
ent, re.DOTALL)

        # 각 조항의 앞뒤 공백을 제거하고 결과 딕셔너리에 저장
```

```
        parsed_law['장'][chapter_title.strip()] = [article.
strip() for article in articles]

    return parsed_law


# 각 페이지의 텍스트를 결합하여 재분리
text_for_delete = r"법제처\s+\d+\s+국가법령정보센터\n개인정보 보
호법"

law_text = "\n".join([re.sub(text_for_delete, "", p.page_co
ntent).strip() for p in pages])

parsed_law = parse_law(law_text)

# 분할된 아이템 갯수 확인
print(len(parsed_law["장"]))
```

- 랭체인 Document 객체에 메타 데이터와 함께 정리

```
# LangChain 프레임워크의 기본 데이터 단위인 Document 클래스 불
러옴
# 텍스트만 있는 문자열을 [내용(page_content) + 정보(metadat
a)]가 결합된 객체로 구조화
from langchain_core.documents import Document

final_docs = []
for law in parsed_law['장'].keys():
    for article in parsed_law['장'][law]:

        # metadata 내용을 정리
        metadata = {
                "source": pdf_file,
                "chapter": law,
                "name" : "개인정보 보호법"
                }
```

```
        # metadata 내용을 본문에 추가
        content = f"[법률정보]\n다음 조항은 {metadata['nam
e']} {metadata['chapter']}에서 발췌한 내용입니다.\n\n[법률조
항]\n{article}"

        final_docs.append(Document(page_content=content,
metadata=metadata))

len(final_docs)
```

벡터 저장소에 인덱싱

```
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings
# 텍스트를 고정된 크기의 벡터로 변환하는 번역기 역할
embeddings_model = OllamaEmbeddings(model="bge-m3")

# Chroma 인덱스 생성
personal_db = Chroma.from_documents(
    # final_docs 내의 모든 page_content를 bge-m3 모델에 통과시
켜 벡터값으로 변환
    documents=final_docs,
    embedding=embeddings_model,
    # 논리적인 저장 공간을 만듦
    collection_name="personal_law",
    # 데이터를 물리적으로 저장해 나중에 코드 다시 실행시 불러올 수
있도록 함
    persist_directory="./chroma_db",
)
```

법률 정보 검색 도구, 웹 검색 도구 정의

```
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings
```

```
from langchain_core.documents import Document

from langchain.retrievers import ContextualCompressionRetri
ever
from langchain.retrievers.document_compressors import Cross
EncoderReranker
from langchain_community.cross_encoders import HuggingFaceC
rossEncoder
from langchain_community.retrievers import TavilySearchAPIR
etriever
from langchain_core.tools import tool
from typing import List

# 문서 임베딩 모델
embeddings_model = OllamaEmbeddings(model="bge-m3")

# Re-rank 모델
# 벡터 DB가 1차로 찾아온 k개의 문서 중 re-rank 모델이 문서의 관계
를 계산해 가장 관련성 높은 2개 남김
rerank_model = HuggingFaceCrossEncoder(model_name="BAAI/bge
-reranker-v2-m3")
cross_reranker = CrossEncoderReranker(model=rerank_model, t
op_n=2)

# 개인정보보호법 검색
personal_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="personal_law",
    persist_directory="./chroma_db",
)

# base_retriever와 base_compressor를 하나로 묶음
# DB에서 문서를 꺼내고 Re-rank 모델로 정제하는 복잡한 과정을 하나의
리트리버 객체로 캡슐화
personal_db_retriever = ContextualCompressionRetriever(
    base_compressor=cross_reranker,
    base_retriever=personal_db.as_retriever(search_kwargs=
{"k":5}),
```

```
)

@tool
def personal_law_search(query: str) -> List[Document]:
    """개인정보보호법 법률 조항을 검색합니다."""
    docs = personal_db_retriever.invoke(query)

    if len(docs) > 0:
        return docs

    return [Documen t(page_content="관련 정보를 찾을 수 없습니
다.")]


# 근로기준법 검색
labor_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="labor_law",
    persist_directory="./chroma_db",
)

#
labor_db_retriever = ContextualCompressionRetriever(
    base_compressor=cross_reranker,
    base_retriever=labor_db.as_retriever(search_kwargs=
{"k":5}),
)


@tool
def labor_law_search(query: str) -> List[Document]:
    """근로기준법 법률 조항을 검색합니다."""
    docs = labor_db_retriever.invoke(query)

    if len(docs) > 0:
        return docs

    return [Document(page_content="관련 정보를 찾을 수 없습니
```

```
다.")]


# 주택임대차보호법 검색
housing_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="housing_law",
    persist_directory="./chroma_db",
)

housing_db_retriever = ContextualCompressionRetriever(
    base_compressor=cross_reranker,
    base_retriever=housing_db.as_retriever(search_kwargs=
{"k":5}),
)



@tool
def housing_law_search(query: str) -> List[Document]:
    """주택임대차보호법 법률 조항을 검색합니다."""
    docs = housing_db_retriever.invoke(query)

    if len(docs) > 0:
        return docs

    return [Document(page_content="관련 정보를 찾을 수 없습니
다.")]



# 웹 검색
# 로컬DB (개인정보법, 근로기준법 등)에 없는 최신 정보나 외부 사례를
찾기 위해 Tavily API연동
# 웹에서 가져온 정보는 <Document href="..."/> 태그로 감싸서, LLM
이 나중에 출처를 답변에 표기하기 쉽게함

web_retriever = ContextualCompressionRetriever(
```

```
    base_compressor=cross_reranker,
    base_retriever=TavilySearchAPIRetriever(k=10),
)

@tool
def web_search(query: str) -> List[str]:
    """데이터베이스에 없는 정보 또는 최신 정보를 웹에서 검색합니다."""

    docs = web_retriever.invoke(query)

    formatted_docs = []
    for doc in docs:
        formatted_docs.append(
            Document(
                page_content= f'<Document href="{doc.metadata["source"]}"/>\n{doc.page_content}\n</Document>',
                metadata={"source": "web search", "url": doc.metadata["source"]}
            )
        )

    if len(formatted_docs) > 0:
        return formatted_docs

    return [Document(page_content="관련 정보를 찾을 수 없습니다.")]
```

- 시스템 구조 요약
  1. 도구 선택
     - LLM이 질문에 맞춰 적절한 검색 함수 @tool 선택
       - 개인정보 vs 근로 vs 웹
  2. 1차 검색
     - Chroma DB 또는 Tavily에서 관련 문서 k개 추출
     - Vector 유사도 기반

3. Re-ranking
    - bge-reranker가 질문과의 연관성을 정밀 재계산
    - 상위 2개만 선별
4. 결과 반환
    - 최종 정제된 문서를 에이전트에게 전달

Agent RAG 구현

- 각 법률에 특화된 RAG 에이전트를 구현
- 질문 라우팅을 통해 각 에이전트를 도구 형태로 사용
- 생성된 답변에 대한 피드백을 제공하는 에이전트 사용
- 필요한 경우 사람의 피드백 요청 (답변이 애매한 경우 - 재검색 여부 판단)

기본 RAG 에이전트 구조

```
from pydantic import BaseModel, Field
from typing import List, TypedDict, Annotated, Optional
from operator import add
from langchain_core.documents import Document

# LangGraph에서 노드 간 어떤 데이터를 주고받을지 결정하는 역할

class CorrectiveRagState(TypedDict):    # RAG 시스템의 공통
상태
    question: str                  # 사용자의 질문
    generation: str                # LLM 생성 답변
    documents: List[Document]      # 컨텍스트 문서 (검색된 문서)
    num_generations: int           # 질문 or 답변 생성 횟수 (무
한 루프 방지에 활용)

class InformationStrip(BaseModel):    # 정보의 최소 단위
    """추출된 정보에 대한 내용과 출처, 관련성 점수"""
    content: str = Field(..., description="추출된 정보 내용")
```

```
    source: str = Field(..., description="정보의 출처(법률 조항 또는 URL 등). 예시: 환경법 제22조 3항 or 블로그 환경법 개정 (https://blog.com/page/123)")
    relevance_score: float = Field(..., ge=0, le=1, description="관련성 점수 (0에서 1 사이)")
    faithfulness_score: float = Field(..., ge=0, le=1, description="충실성 점수 (0에서 1 사이)")

class ExtractedInformation(BaseModel):
    strips: List[InformationStrip] = Field(..., description="추출된 정보 조각들")
    query_relevance: float = Field(..., ge=0, le=1, description="질의에 대한 전반전인 답변 가능성 점수 (0에서 1 사이)")

class RefinedQuestion(BaseModel):
    """개선된 질문과 이유"""
    question_refined : str = Field(..., description="개선된 질문")
    reason : str = Field(..., description="이유")

# 개인정보보호법
class PersonalRagState(CorrectiveRagState):    # CorrectiveRagState를 상속받은 특화 상태
    rewritten_query: str   # 재작성한 질문
    extracted_info: Optional[ExtractedInformation]   # 추출된 정보 조각
    node_answer: Optional[str]
```

- 구조화된 데이터 추출 (Pydantic 모델)
  - 검증 가능한 객체로 LLM이 응답하도록 강제하는 스키마
- 이 설계의 강점 CRAG (Corrective RAG 전략)
  1. 신뢰도 평가 기반 : relevance_score 등을 통해 검색 결과가 부적절하다고 판단되면, 그냥 답변하는 것이 아니라 질문을 재작성(RefineQuestion)하거나 웹 검색으로 전환하는 로직 짤 수 있음
  2. 데이터 추적성 : InformationStrip에 출처(source)를 명시하게 해 법률 정보 서비스에서 가장 중요한 근거 제시를 명확히 함

3. 유연한 상속 : CorrectiveRagState를 기본으로 두고 PersonalRagState 처럼 각 법률 도메인별로 확장할 수 있어 유지보수 용이

> 💡 **코드 정리 요약**
> CorrectiveRagState [TypedDict] : 그래프 전체를 흐르는 공통 데이터 구조
> PersonalRagState [TypedDict] : 개인정보보호법 노드 전용 확장 데이터
> InformationStrip [Pydantic] : 개별 정보 조각의 내용, 출처, 점수(신뢰도)
> ExtractedInformation [Pydantic] : 여러 조항에서 추출된 정보의 집합 및 통합 점수
> RefinedQuestion [Pydantic] : 질문 재작성 및 개선 사유 기록

```
from langchain_core.prompts import ChatPromptTemplate
from typing import Literal

# 정보 수집 노드
# 벡터 DB에서 질문과 관련된 법률 문서를 찾음
# 처음에 question을 사용하지만, 루프를 돌아 질문이 재작성되었다면 r
ewritten_query를 우선 사용
def retrieve_documents(state: PersonalRagState) -> Personal
RagState:
    print("---문서 검색---")
    query = state.get("rewritten_query", state["question"])
    docs = personal_law_search.invoke(query)
    return {"documents": docs}

# 정보 검증 및 필터링 노드
# 검색된 문서가 정말 쓸모 있는지 LLM에게 수치적 평가를 맡김
def extract_and_evaluate_information(state: PersonalRagStat
e) -> PersonalRagState:
    print("---정보 추출 및 평가---")
    extracted_strips = []

    for doc in state["documents"]:
        extract_prompt = ChatPromptTemplate.from_messages([
            ("system", """당신은 개인정보보호법 전문가입니다. 주
```

```
어진 문서에서 질문과 관련된 주요 사실과 정보를 3~5개 정도 추출하세
요.
            각 추출된 정보에 대해 다음 두 가지 측면을 0에서 1 사이
의 점수로 평가하세요:
            1. 질문과의 관련성
            2. 답변의 충실성 (질문에 대한 완전하고 정확한 답변을
제공할 수 있는 정도)

            추출 형식:
            1. [추출된 정보]
            - 관련성 점수: [0-1 사이의 점수]
            - 충실성 점수: [0-1 사이의 점수]
            2. [추출된 정보]
            - 관련성 점수: [0-1 사이의 점수]
            - 충실성 점수: [0-1 사이의 점수]
            ...

            마지막으로, 추출된 정보를 종합하여 질문에 대한 전반적인
답변 가능성을 0에서 1 사이의 점수로 평가하세요."""),
            ("human", "[질문]\n{question}\n\n[문서 내용]\n{do
cument_content}")
        ])

        extract_llm = llm.with_structured_output(ExtractedI
nformation)

        extracted_data = extract_llm.invoke(extract_prompt.
format(
            question=state["question"],
            document_content=doc.page_content
        ))
                # 문서 전체가 질문과 무관하면 과감히 버림
        if extracted_data.query_relevance < 0.8:
            continue
                # 만약 앞 단계에서 추출된 정보가 부족하다면,
                # LLM이 기존 질문을 분석해 더 검색이 잘 될 만한
전문 용어로 다시 만듦
                # 검색 실패 시, 사용자가 아닌 AI가 스스로 해결책
```

```
을 찾는 에이전트의 자율성
        for strip in extracted_data.strips:
            if strip.relevance_score > 0.7 and strip.faithf
ulness_score > 0.7:
                extracted_strips.append(strip)

    return {
        "extracted_info": extracted_strips,
        "num_generations": state.get("num_generations", 0)
+ 1
    }

def rewrite_query(state: PersonalRagState) -> PersonalRagSt
ate:
    print("---쿼리 재작성---")

    rewrite_prompt = ChatPromptTemplate.from_messages([
        ("system", """당신은 개인정보보호법 전문가입니다. 주어진
원래 질문과 추출된 정보를 바탕으로, 더 관련성 있고 충실한 정보를 찾기
위해 검색 쿼리를 개선해주세요.

        다음 사항을 고려하여 검색 쿼리를 개선하세요:
        1. 원래 질문의 핵심 요소
        2. 추출된 정보의 관련성 점수
        3. 추출된 정보의 충실성 점수
        4. 부족한 정보나 더 자세히 알아야 할 부분

        개선된 검색 쿼리 작성 단계:
        1. 2-3개의 검색 쿼리를 제안하세요.
        2. 각 쿼리는 구체적이고 간결해야 합니다(5-10 단어 사이).
        3. 개인정보보호법과 관련된 전문 용어를 적절히 활용하세요.
        4. 각 쿼리 뒤에는 해당 쿼리를 제안한 이유를 간단히 설명하세
요.

        출력 형식:
        1. [개선된 검색 쿼리 1]
        - 이유: [이 쿼리를 제안한 이유 설명]
        2. [개선된 검색 쿼리 2]
```

```
        - 이유: [이 쿼리를 제안한 이유 설명]
        3. [개선된 검색 쿼리 3]
        - 이유: [이 쿼리를 제안한 이유 설명]

        마지막으로, 제안된 쿼리 중 가장 효과적일 것 같은 쿼리를 선택하고 그 이유를 설명하세요."""),
        ("human", "원래 질문: {question}\n\n추출된 정보:\n{extracted_info}\n\n위 지침에 따라 개선된 검색 쿼리를 작성해주세요.")
    ])

    extracted_info_str = "\n".join([strip.content for strip in state["extracted_info"]])

    rewrite_llm = llm.with_structured_output(RefinedQuestion)

    response = rewrite_llm.invoke(rewrite_prompt.format(
        question=state["question"],
        extracted_info=extracted_info_str
    ))

    return {"rewritten_query": response.question_refined}
# 검증을 통과한 알짜 정보(extracted_info)만을 모아 마크다운 형식의 전문적인 답변 작성
# strip.source를 사용해 법적 근거(출처)를 명시하도록 강제해 답변의 신뢰도 확보
def generate_node_answer(state: PersonalRagState) -> PersonalRagState:
    print("---답변 생성---")

    answer_prompt = ChatPromptTemplate.from_messages([
        ("system", """당신은 개인정보보호법 전문가입니다. 주어진 질문과 추출된 정보를 바탕으로 답변을 생성해주세요.
        답변은 마크다운 형식으로 작성하며, 각 정보의 출처를 명확히 표시해야 합니다.
        답변 구조:
        1. 질문에 대한 직접적인 답변
        2. 관련 법률 조항 및 해석
```

```
        3. 추가 설명 또는 예시 (필요한 경우)
        4. 결론 및 요약
        각 섹션에서 사용된 정보의 출처를 괄호 안에 명시하세요. 예: (출처: 개인정보 보호법 제15조)"""),
        ("human", "질문: {question}\n\n추출된 정보:\n{extracted_info}\n\n위 지침에 따라 최종 답변을 작성해주세요.")
    ])

    extracted_info_str = "\n".join([f"내용: {strip.content}\n출처: {strip.source}\n관련성: {strip.relevance_score}\n충실성: {strip.faithfulness_score}" for strip in state["extracted_info"]])

    node_answer = llm.invoke(answer_prompt.format(
        question=state["question"],
        extracted_info=extracted_info_str
    ))

    return {"node_answer": node_answer.content}
# 흐름 제어 로직
# 충분한 정보인지 (len(extracted_info) >= 1) 이면 즉시 종료 후 답변 생성
# 너무 많이 반복했는지 (num_generations >= 2) 이면 무한 루프를 막기 위해 강제 종료
# 그 외 : 정보가 부족하면 "계속"을 반환해 rewrite_query 노드로 보냄
def should_continue(state: PersonalRagState) -> Literal["계속", "종료"]:
    if state["num_generations"] >= 2:
        return "종료"
    if len(state["extracted_info"]) >= 1:
        return "종료"
    return "계속"
```

질문 라우팅

- 사용자 질문을 분석해 적절한 에이전트를 선택 (Adaptive RAG 적용)

```
from typing import Annotated
from operator import add

# 메인 그래프 상태 정의
class ResearchAgentState(TypedDict):
        # 사용자의 최초 입력 유지
    question: str
    # LangGraph의 add Reducer를 사용
    # 덮어쓰지 않고 차곡차곡 쌓이게 저장
    answers: Annotated[List[str], add]
    # 수집된 여러 조각 답변들을 LLM이 최종적으로 정리해 사용자에게
전달할 완성된 답변
    final_answer: str
    # 검색에 사용된 데이터 소스 목록을 기록해 투명성을 높임
    datasources: List[str]
    # 생성된 답변의 정확성이나 충실도를 평가한 결과 저장
    evaluation_report: Optional[dict]
    # 사용자에게 추가 질문을 하거나, 만족 여부를 확인받아 다음 단계
를 결정할 때 사용
    user_decision: Optional[str]
```

- 질문 라우팅의 워크플로우
  1. 분석 : 질문에 개인정보가 포함되었는가? → PersonalLawAgent로 전송
  2. 분류 : 질문에 해고/수당이 포함되었는가? → LaborLawAgent로 전송
  3. 병합 : 각 노드에서 보낸 답변들이 answers 리스트에 합쳐짐
  4. 최종 생성 : 수집된 정보를 바탕으로 final_answer 작성

```
from typing import Literal
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field

# 라우팅 결정을 위한 데이터 모델
# 단일 도구 선택을 위한 단위. Literal을 사용해 LLM이 정의된 4가지
도구 외에 엉뚱한 답변을 하지 못하도록 강제
# 질문에 여러 개의 도구가 필요한 경우 이를 한꺼번에 반환
```

```
class ToolSelector(BaseModel):
    """Routes the user question to the most appropriate too
l."""
    tool: Literal["search_personal", "search_labor", "searc
h_housing", "search_web"] = Field(
        description="Select one of the tools, based on the
user's question.",
    )

class ToolSelectors(BaseModel):
    """Select the appropriate tools that are suitable for t
he user question."""
    tools: List[ToolSelector] = Field(
        description="Select one or more tools, based on the
user's question.",
    )

# 구조화된 출력을 위한 LLM 설정
structured_llm_tool_selector = llm.with_structured_output(T
oolSelectors)

# 라우팅을 위한 프롬프트 템플릿
system = dedent("""You are an AI assistant specializing in
routing user questions to the appropriate tools.
Use the following guidelines:
- For questions specifically about legal provisions or arti
cles of the privacy protection law (개인정보 보호법), use the
search_personal tool.
- For questions specifically about legal provisions or arti
cles of the labor law (근로기준법), use the search_labor too
l.
- For questions specifically about legal provisions or arti
cles of the housing law (주택임대차보호법), use the search_ho
using tool.
- For any other information, including questions related to
these laws but not directly about specific legal provision
s, or for the most up-to-date data, use the search_web too
l.
```

```
Always choose all of the appropriate tools based on the use
r's question.
If a question is about a law but doesn't seem to be asking
about specific legal provisions, include both the relevant
law search tool and the search_web tool.""")

route_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system),
        ("human", "{question}"),
    ]
)

# 질문 라우터 정의
question_tool_router = route_prompt | structured_llm_tool_s
elector

# 테스트 실행
print(question_tool_router.invoke({"question": "근로계약 체결
할 때 개인정보 취급 상의 유의사항은 무엇인가요?"}))
print(question_tool_router.invoke({"question": "법에서 정한
연차휴가 기준을 알려주세요."}))
print(question_tool_router.invoke({"question": "개인정보보호
법에서 정한 가명정보의 정의는 무엇인가요?"}))
```

- 지능형 라우팅의 핵심
    1. 구조화된 출력(with_structured_output) : 단순 텍스트가 아닌 리스트 형태의 객체 반환. 이후 LangGraph에서 for 문을 사용해 각 노드 병렬로 실행시키기 매우 적합한 구조
    2. 데이터 소스의 분리 : 각 법령 DB를 따로 운영. 서로 다른 법령 간 데이터 간섭(노이즈)를 줄이고 검색 정확도 높임
    3. 병렬 실행 가능성 : ToolSelectors가 리스트를 반환하면, LangGraph의 send API 등을 활용해 여러 검색 노드를 동시에 띄울 수 있는 기반 마련

```
# 질문 라우팅 노드
# 사용자의 질문을 분석해 어떤 데이터 소스(도구)를 사용할지 결정 및
```

```
목록 기록
# question_tool_router를 호출해 질문에 적합한 도구 리스트를 객체 형태로 받음
def analyze_question_tool_search(state: ResearchAgentState):
    question = state["question"]
    result = question_tool_router.invoke({"question": question})
    # 결과 객체 result에서 tool 이름들만 추출해 리스트로 만듦
    datasources = [tool.tool for tool in result.tools]
    # 이 리스트를 ResearchAgentState의 datasources 필드에 저장
    return {"datasources": datasources}

# 조건부 엣지
# datasources 에 담긴 목록을 보고, 다음에 실행할 노드들의 이름을 리스트로 반환
# Set 활용 : set(state['datasources'])를 통해 혹시 모를 중복된 도구 선택 제거
# 안전장치 Validation : valid_source를 정의해 LLM이 실수로 정의되지 않은 도구 이름을 반환해도
# 시스템이 에러로 멈추지 않고 전체 소스를 검색하도록 유도하는 방어적 설계
def route_datasources_tool_search(state: ResearchAgentState) -> List[str]:
    datasources = set(state['datasources'])
    valid_sources = {"search_personal", "search_labor", "search_housing", "search_web"}

    if datasources.issubset(valid_sources):
        return list(datasources)

    return list(valid_sources)
```

- 데이터 흐름
    1. 분석 : “회사에서 개인정보를 유출했을 때 처벌은?” 질문 유입.
    2. 결정 : datasources = [”serach_personal”, “search_web”] 저장
    3. 분기 : route_datasources_tool_search 가 해당 리스트 반환

4. 병렬 실행 : 개인정보보호법 전용 RAG 노드와 웹 검색 노드가 동시에 가동

```
# 최종 답변 생성 노드
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

# RAG 프롬프트 정의
rag_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are an assistant answering questions
based on provided documents. Follow these guidelines:

1. Use only information from the given documents.
2. If the document lacks relevant info, say "제공된 정보로는
충분한 답변을 할 수 없습니다."
3. Cite the source of information for each sentence in your
answer. Use the following format:
    - For legal articles: "법률명 제X조 Y항"
    - For web sources: "출처 제목 (URL)"
4. Don't speculate or add information not in the documents.
5. Keep answers concise and clear.
6. Omit irrelevant information.
7. If multiple sources provide the same information, cite a
ll relevant sources.
8. If information comes from multiple sources, combine them
coherently while citing each source.

Example of citation usage:
"부동산 거래 시 계약서에 거래 금액을 명시해야 합니다 (부동산 거래신고
등에 관한 법률 제3조 1항). 또한, 계약 체결일로부터 30일 이내에 신고
해야 합니다 (부동산 거래 신고 안내 블로그, https://example.com/re
alestate)."
"""
    ),
    ("human", "Answer the following question using these do
cuments:\n\n[Documents]\n{documents}\n\n[Question]\n{questi
on}"),
])
```

```
# 각 도구 노드에서 수집되어 answers 리스트에 쌓인 모든 정보 (documents)를 바탕으로 최종 답변 생성

def answer_final(state: ResearchAgentState) -> ResearchAgentState:
    """
    Generate answer using the retrieved_documents
    """
    print("---최종 답변---")
    question = state["question"]
    # 여러 노드에서 add 연산자로 합쳐진 답변들을 가져와 텍스트로 결합
    documents = state.get("answers", [])
    if not isinstance(documents, list):
        documents = [documents]

    # 문서 내용을 문자열로 결합
    documents_text = "\n\n".join(documents)

    # RAG generation
    rag_chain = rag_prompt | llm | StrOutputParser()
    generation = rag_chain.invoke({"documents": documents_text, "question": question})
    return {"final_answer": generation, "question":question}


# LLM Fallback 프롬프트 정의
fallback_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are an AI assistant helping with various topics. Follow these guidelines:

1. Provide accurate and helpful information to the best of your ability.
2. Express uncertainty when unsure; avoid speculation.
3. Keep answers concise yet informative.
4. Respond ethically and constructively.
5. Mention reliable general sources when applicable."""),
```

```
    ("human", "{question}"),
])
# 검색 결과가 없거나 내부 DB로 해결이 불가능한 경우
# LLM의 자체 지식을 바탕으로 답변
def llm_fallback(state: ResearchAgentState) -> ResearchAgen
tState:
    """
    Generate answer using the LLM without context
    """
    print("---Fallback 답변---")
    question = state["question"]

    # LLM chain
    llm_chain = fallback_prompt | llm | StrOutputParser()

    generation = llm_chain.invoke({"question": question})
    return {"final_answer": generation, "question":questio
n}
```

Human In The Loop 추가

```
# 답변 평가하는 노드를 추가
# 생성 답변이 질문의 의도에 맞는지, 법률적으로 타당해 보이는지 LLM이
다시 한번 검토자 입장에서 채점
def evaluate_answer_node(state:ResearchAgentState):
    question = state["question"]
    final_answer = state["final_answer"]

    messages = [HumanMessage(content=f"""[질문]\n\{questio
n}n\n[답변]\n{final_answer}""")]
    # 질문과 답변 answer_reviewer 모델에 전달
    response = answer_reviewer.invoke({"messages": message
s})
    # 결과를 JSON 형태로 파싱해 evaluation_report에 저장
    response_dict = json.loads(response['messages'][-1].con
tent)
```

```
    return {"evaluation_report": response_dict, "question":
question, "final_answer": final_answer}

# HITL 조건부 엣지 정의
def human_review(state: ResearchAgentState):
    print("\n현재 답변:")
    print(state['final_answer'])
    print("\n평가 결과:")
    print(f"총점: {state['evaluation_report']['total_scor
e']}/60")
    print(state['evaluation_report']['brief_evaluation'])
    # 인간의 개입으로 인해 의사결정
    user_input = input("\n이 답변을 승인하시겠습니까? (y/n):
").lower()

    if user_input == 'y':
        return "approved"
    else:
        return "rejected"
```