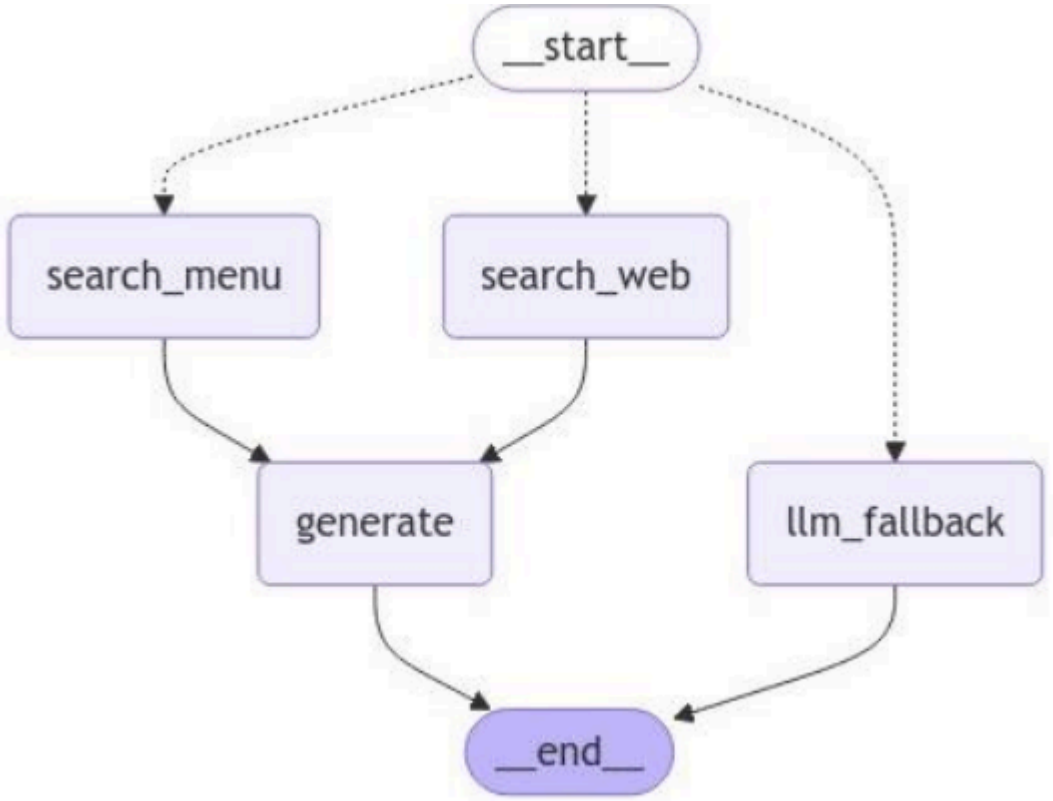
다양한 에이전트 기반 RAG 구현하기

Adaptive RAG 개요

- 질문 복잡성에 따라 가장 적합한 검색 및 생성 전략을 동적으로 선택하는 방법
- 작동 방식
  1. 사용자 질문의 복잡성 수준을 분석
  2. 분석 결과에 따라 가장 적합한 처리 전략 선택
     - 단순 질문 : 기본 LLM 또는 단순 검색
     - 중간 복잡성 : 단일 단계 검색 증강 LLM 사용
     - 복잡한 질문 : 여러 단계의 검색과 추론을 수행
  3. 선택된 전략에 따라 질문을 처리하고 응답 생성

Adaptive RAG 구현 실습

- 질문 유형에 따라 가장 적합한 검색 전략 사용 (질문 라우팅)
- 질문 라우팅을 통한 검색 도구 선택
  1. 레스토랑 메뉴 관련 질문 : search_menu 도구로 라우팅
  2. 메뉴 외의 정보나 일반적 질문 : search_web 도구로 라우팅




```
# 라우팅 결정을 위한 데이터 모델
class ToolSelector(BaseModel):
    """사용자 질문을 가장 적절한 도구로 라우팅합니다."""
    tool: Literal["search_menu", "search_web"] = Field(
        description = "질문에 따라 search_menu 또는 search_we
b 도구 중 하나를 선택",
    )
# 구조화된 출력을 위한 LLM 설정
llm = ChatGoogleGenerativeAI(model = "gemini-1.5-flash", te
mperature = 0)
structured_llm_router = llm.with_structured_output(ToolSele
ctor)

# 라우팅을 위한 프롬프트 템플릿
system = """당신은 사용자 질문을 적절한 도구로 라우팅하는 전문가입
니다.
레스토랑 메뉴에 관한 질문은 serach_menu 도구를 사용하세요.
레스토랑 메뉴에 없는 정보 또는 일반적인 질문은 search_web 도구를 사
용하세요."""

route_prompt = ChatPromptTemplate.from_messages(
```

```
    [
        ("system", system),
        ("human", "{question}"),
    ]
)

# 질문 라우터 정의
question_router = route_prompt | structured_llm_router
```

- 상태 정의

```
from typing import TypedDict, List
from langchain_core.documents import Document

# 상태 Schema 정의
class AdaptiveRagState(TypedDict):
    question: str
    documents: List[Document]
    generation: str
```

- 질문 분석 → 라우팅

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

# 라우팅 결정을 위한 데이터 모델
class ToolSelector(BaseModel):
    """Routes the user question to the most appropriate
tool."""
    tool: Literal["search_menu", "search_web", "search_w
ine"] = Field(
        description="Select one of the tools: search_men
u, search_wine or search_web based on the user's questio
n.",
    )

# 구조화된 출력을 위한 LLM 설정
```

```
structured_llm = llm.with_structured_output(ToolSelecto
r)

# 라우팅을 위한 프롬프트 템플릿
system = dedent("""You are an AI assistant specializing
in routing user questions to the appropriate tool.
Use the following guidelines:
- For questions about the restaurant's menu, use the sea
rch_menu tool.
- For wine recommendations or pairing information, use t
he search_wine tool.
- For any other information or the most up-to-date data,
use the search_web tool.
Always choose the most appropriate tool based on the use
r's question.""")

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

# 질문 라우터 정의
question_router = route_prompt | structured_llm

# 테스트 실행
print(question_router.invoke({"question": "채식주의자를 위
한 메뉴가 있나요?"}))
print(question_router.invoke({"question": "스테이크 메뉴와
어울리는 와인을 추천해주세요."}))
print(question_router.invoke({"question": "2022년 월드컵
우승 국가는 어디인가요?"}))
```

```
# 질문 라우팅 노드
def route_question_adaptive(state: AdaptiveRagState) -> Lit
eral["search_menu", "search_wine", "search_web", "llm_fallb
ack"]:
```

```
    question = state["question"]
    try:
        result = question_router.invoke({"question": questi
on})
        datasource = result.tool

        if datasource == "search_menu":
            return "search_menu"
        elif datasource == "search_wine":
            return "search_wine"
        elif datasource == "search_web":
            return "search_web"
        else:
            return "llm_fallback"

    except Exception as e:
        print(f"Error in routing: {str(e)}")
        return "llm_fallback"
```

- 검색 노드

```
def search_menu_adaptive(state: AdaptiveRagState):
    """
    Node for searching information in the restaurant men
u
    """
    question = state["question"]
    docs = search_menu.invoke(question)
    if len(docs) > 0:
        return {"documents": docs}
    else:
        return {"documents": [Document(page_content="관
련 메뉴 정보를 찾을 수 없습니다.")]}


def search_wine_adaptive(state: AdaptiveRagState):
    """
    Node for searching information in the restaurant's w
```

```
ine list
    """
    question = state["question"]
    docs = search_wine.invoke(question)
    if len(docs) > 0:
        return {"documents": docs}
    else:
        return {"documents": [Document(page_content="관
련 와인 정보를 찾을 수 없습니다.")]}


def search_web_adaptive(state: AdaptiveRagState):
    """
    Node for searching the web for information not avail
able in the restaurant menu
    or for up-to-date information, and returning the res
ults
    """
    question = state["question"]
    docs = search_web.invoke(question)
    if len(docs) > 0:
        return {"documents": docs}
    else:
        return {"documents": [Document(page_content="관
련 정보를 찾을 수 없습니다.")]}
```

- 생성 노드

```
from langchain_core.output_parsers import StrOutputParse
r
from langchain_core.prompts import ChatPromptTemplate

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

1. Use only information from the given documents.
```

```
2. If the document lacks relevant info, say "The provide
d documents don't contain information to answer this que
stion."
3. Cite relevant parts of the document in your answers.
4. Don't speculate or add information not in the documen
ts.
5. Keep answers concise and clear.
6. Omit irrelevant information."""
),
    ("human", "Answer the following question using these
documents:\n\n[Documents]\n{documents}\n\n[Question]\n{q
uestion}"),
])

def generate_adaptive(state: AdaptiveRagState):
    """
    Generate answer using the retrieved_documents
    """
    question = state.get("question", None)
    documents = state.get("documents", [])
    if not isinstance(documents, list):
        documents = [documents]

    # 문서 내용을 문자열로 변환
    documents_text = "\n\n".join([f"---\n본문: {doc.page
_content}\n메타데이터:{str(doc.metadata)}\n---" for doc i
n documents])

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

```
# LLM Fallback 프롬프트 정의
fallback_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are an AI assistant helping with v
arious 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. Inform users they can ask for clarification if neede
d.
5. Respond ethically and constructively.
6. Mention reliable general sources when applicabl
e."""),
    ("human", "{question}"),
])

def llm_fallback_adaptive(state: AdaptiveRagState):
    """
    Generate answer using the LLM without context
    """
    question = state.get("question", "")

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

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

- 그래프 연결

```
from langgraph.graph import StateGraph, START, END
from IPython.display import Image, display

# 그래프 구성
builder = StateGraph(AdaptiveRagState)

# 노드 추가
builder.add_node("search_menu", search_menu_adaptive)
```

```
builder.add_node("search_wine", search_wine_adaptive)
builder.add_node("search_web", search_web_adaptive)
builder.add_node("generate", generate_adaptive)
builder.add_node("llm_fallback", llm_fallback_adaptive)

# 엣지 추가
builder.add_conditional_edges(
    START,
    route_question_adaptive
)

builder.add_edge("search_menu", "generate")
builder.add_edge("search_wine", "generate")
builder.add_edge("search_web", "generate")
builder.add_edge("generate", END)
builder.add_edge("llm_fallback", END)

# 그래프 컴파일
adaptive_rag = builder.compile()

# 그래프 시각화
display(Image(adaptive_rag.get_graph().draw_mermaid_png
()))
```

```
# 그래프 실행
inputs = {"question": "스테이크 메뉴의 가격은 얼마인가요?"}
for output in adaptive_rag.stream(inputs):
    for key, value in output.items():
        print(f"Node '{key}':")
        print(f"State '{value.keys()}':")
        print(f"Value '{value}':")
    print("\n---\n")

# 최종 답변
print(value["generation"])
```

Human-in-the-Loop (HITL)

- AI 시스템의 자동화된 처리와 인간의 전문 지식 결합
- 시스템의 결정이나 출력에 대해 인간이 검토하고 개입할 수 있는 지점

  → 이를 통해 AI의 효율성과 인간의 판단력을 모두 활용
- LangGraph의 Breakpoints 활용
  - Breakpoints : 그래프 실행을 특정 지점에서 일시 중지하는 매커니즘
  - LangGraph의 체크 포인트 시스템을 기반으로 구현
  - 이를 통해 인간 전문가가 중간 결과를 검토하고 필요한 경우 개입

```
# 체크 포인트 설정
from langgraph.checkpoint.memory import MemorySaver
memory = MemorySaver()

# 컴파일 : 'retrieve', 'generate' 노드 전에 중단점 추가
graph = builder.compile(
    chechpointer = memory,
    interrupt_before = ['retrieve', 'generate']
    )
```

- Breakpoint 실행
  - graph.stream()을 사용해 그래프를 단계별로 실행 (스레드 ID 지정)
  - 그래프를 실행하면, 설정된 Breakpoints마다 멈춤

```
# 도구 사용 전 중단점에서 실행을 멈춤
from langchain_core.messages import HumanMessage

thread = {"configurable", : {"thread_id": "breakpoint_te
st"}}
inputs = [HumanMessage(content = "대표 메뉴는 무엇인가
요?")]
for event in graph.stream({"messages" : inputs}, thread,
stream_mode = "values"):
    event["messages"][-1].pretty_print()
```

  - 체크포인트 설정은 그래프 컴파일 단계에서 메모리 설정

    - stream() 메소드를 사용해야함
        - invoke 메소드를 사용하면 start부터 end까지 전체를 실행하고 최종상태만 출력됨
        - stream을 사용하면 단계별로 가능
        - thread id를 지정해야 별도의 thread로 관리 가능

- Breakpoint 상태 관리
    - graph.get_state(thread)를 호출해 현재 그래프의 상태 확인

```
# 상태  확인
current_state = graph.get_state(thread)
print("---그래프 상태---")
print(corrent_state)
print()
for m in current_state.values.get("messages"):
    m.pretty_print()
```

    - 다음에 실행될 노드 확인
        - 중단점 이후에 실행될 다음 노드 확인 가능

```
current_state.next
```

    - Breakpoint 이후 단계 계속해서 실행
        - 입력값을 None으로 지정하면 중단점부터 실행

```
for event graph.stream(None, thread, stream_mode = "v
alues"):
    event["messages"][-1].pretty_print()
```

- 그래프 상태 업데이트
    - graph.update_state() 메서드를 사용해 그래프의 상태 업데이트
    - thread는 이전에 정의한 스레드 설정
    - {”num_generations”: 2}는 업데이트할 상태 정보

- 여기서는 num_generations 필드의 값을 2로 설정

```
# num_generations 필드 확인
current_state.values.get("num_generations", None)

# 상태 업데이트 - num_generations 필드값을 업데이트
graph.update_state(thread, {"num_generations": 2})
new_state = graph.get_state(thread)

for m in new_state.values.get("messages"):
    m.pretty_print()
print()
print("-"*50)
print(new_state.values.get("num_generations"))
```

- 체크 포인트 설정

```
from langgraph.checkpoint.memory import MemorySaver
memory = MemorySaver()
```

- Breakpoint 추가

```
# 컴파일 - 'generate' 노드 전에 중단점 추가
adaptive_rag_hitl = builder.compile(checkpointer=memory,
interrupt_before=["generate"])

# 그래프 출력
display(Image(adaptive_rag_hitl.get_graph().draw_mermaid
_png()))
```

- Breakpoint 실행 확인

```
# 도구 사용 전 중단점에서 실행을 멈춤

thread = {"configurable": {"thread_id": "breakpoint_tes
t"}}
inputs = {"question": "스테이크 메뉴의 가격은 얼마인가요?"}
for event in adaptive_rag_hitl.stream(inputs, config=thr
```

```
ead):
    for k, v in event.items():
        # '__end__' 이벤트는 미출력
        if k != "__end__":
            print(f"{k}: {v}")  # 이벤트의 키와 값을 함께 출
력
```

- Breakpoint 상태 관리

```
# 상태 확인
current_state = adaptive_rag_hitl.get_state(thread)
print("---그래프 상태---")
print(current_state)
print("-"*50)
print(current_state.values.get("generation"))
```

- 다음 실행 노드 확인

```
# 다음에 실행될 노드를 확인
current_state.next
```

- Breakpoint 이후 단계를 계속 실행

```
# 입력값을 None으로 지정하면 중단점부터 실행하는 의미
for event in adaptive_rag_hitl.stream(None, config=threa
d):
    for k, v in event.items():
        # '__end__' 이벤트는 미출력
        if k != "__end__":
            print(f"{k}: {v}")  # 이벤트의 키와 값을 함께 출
력
```

```
# 다음에 실행될 노드를 확인
current_state = adaptive_rag_hitl.get_state(thread)
current_state.next
```

```
# 최종 답변
current_state = adaptive_rag_hitl.get_state(thread)
print(current_state.values.get("generation"))
```

- 상태 업데이트

```
# 새로운 thread를 생성하고, 새로운 질문을 수행
thread = {"configurable": {"thread_id": "breakpoint_upda
te"}}
inputs = {"question": "매운 음식이 있나요?"}
for event in adaptive_rag_hitl.stream(inputs, config=thr
ead):
    for k, v in event.items():
        if k != "__end__":
            print(f"{k}: {v}")
```

```
# 다음에 실행될 노드를 확인
current_state = adaptive_rag_hitl.get_state(thread)
current_state.next
```

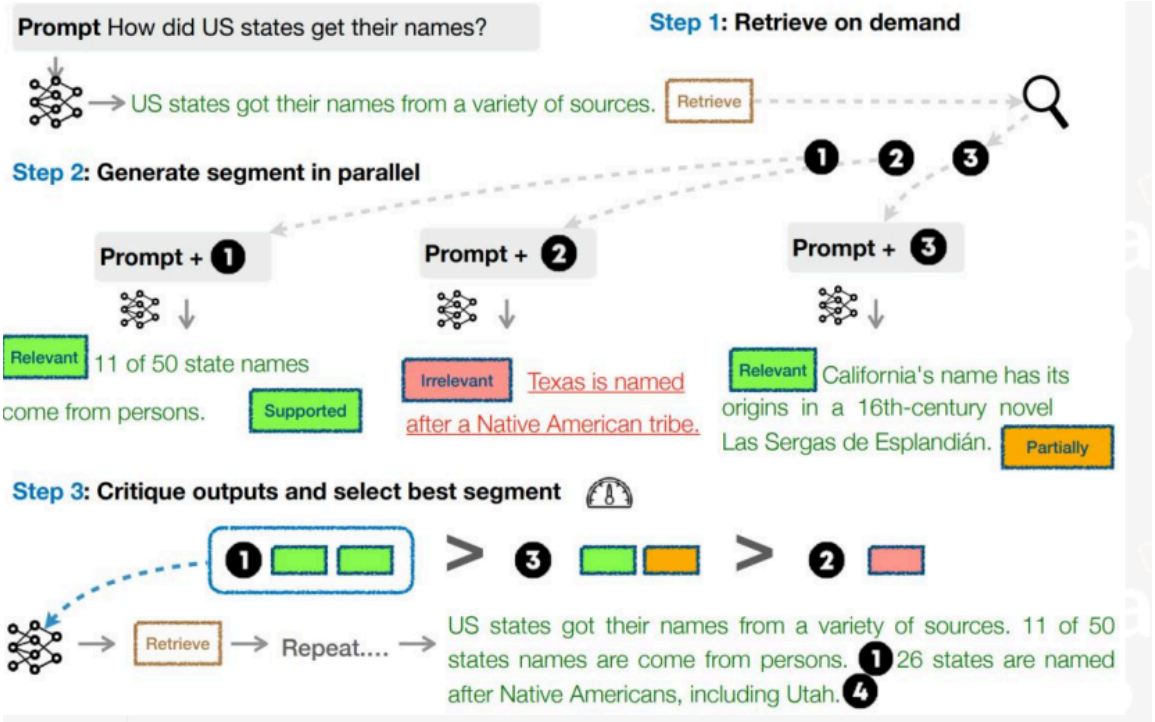
```
# question, generation 필드 확인
current_state = adaptive_rag_hitl.get_state(thread)
print(current_state.values.get("question"))
print("-"*50)
print(current_state.values.get("generation"))
```

```
# 입력값을 None으로 지정하면 중단점부터 실행하고 최종 답변을 생성
for event in adaptive_rag_hitl.stream(None, config=threa
d):
    for k, v in event.items():
        # '__end__' 이벤트는 미출력
        if k != "__end__":
            print(f"{k}: {v}")  # 이벤트의 키와 값을 함께 출
력
```

```
# 최종 답변 확인
print(event["generate"]["generation"])
```

Self-RAG

- 기존 RAG 모델에 자기 반영(self-reflection)능력을 추가한 확장 모델
- 정보 검색, 생성, 그리고 자체 평가를 통합해 더 정확하고 관련성 높은 응답을 생성하는 것을 목표
- 작동 방식
  1. 초기 쿼리 처리 : 사용자의 질문을 받아 관련 정보 검색
  2. 초기 응답 생성 : 검색된 정보를 바탕으로 첫 번째 응답 생성
  3. 자기 평가 : 생성된 응답의 품질, 관련성, 정확성을 평가
  4. 개선 결정 : 추가 정보 검색 또는 응답 재생성을 결정
  5. 반복 : 만족스러운 결과를 얻을 때까지 이전 단계 반복



Self-RAG 구현

- Self-RAG (Retreival-Augmented Generation with Self-Reflection)
- 주요 단계
  1. 검색 설정 (Retrieval Decision)
     - 입력 : 질문 x 또는 질문 x와 생성된 답변 y
     - 목적 : 검색기 R을 사용해 D개의 청크를 검색할지 결정
     - 출력 : “yes”, “no”, “continue” 중 하나
     - 의미 : 시스템이 추가 정보가 필요한지 판단
  2. 검색된 문서 관련성 평가
     - 입력 : 질문 x와 각 검색된 청크 d
     - 목적 : 각 청크가 질문에 유용한 정보를 제공하는지 평가
     - 출력 : “relevant” 또는 “irrelevant”
     - 의미 : 관련 없는 정보를 필터링해 품질 향상
  3. 생성된 답변 환각 평가
     - 입력 : 질문 x, 청크 d, 생성된 텍스트 y
     - 목적 : 생성된 텍스트가 청크의 정보에 의해 지지되는지 (근거가 있는지) 평가
     - 출력 : “fully supported”, “partially supported”, “no support”
     - 의미 : 환각 (hallucination)을 감지하고 정보의 신뢰성 확인
  4. 생성된 답변의 유용성 평가
     - 입력 : 질문 x와 생성된 텍스트 y
     - 목적 : 생성된 텍스트가 질문에 유용한 응답인지 평가
     - 출력 : 5점 척도 (5: 매우 유용, 1: 전혀 유용하지 않음)
     - 의미 : 응답의 품질과 관련성 수치화
- LLM 체인
  1. Retrieval Grader

```
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI
```

```
# 검색된 문서의 관련성 평가 결과를 위한 데이터 모델 정의
class BinaryGradeDocuments(BaseModel):
    """Binary score for relevance check on retrieved
documents."""

    binary_score: str = Field(
        description="Documents are relevant to the qu
estion, 'yes' or 'no'"
    )

# LLM 모델 초기화 및 구조화된 출력 설정
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
structured_llm_grader = llm.with_structured_output(Bi
naryGradeDocuments)

# 문서 관련성 평가를 위한 시스템 프롬프트 정의
system_prompt = """You are an expert in evaluating th
e relevance of search results to user queries.

[Evaluation criteria]
1. 키워드 관련성: 문서가 질문의 주요 단어나 유사어를 포함하는
지 확인
2. 의미적 관련성: 문서의 전반적인 주제가 질문의 의도와 일치하
는지 평가
3. 부분 관련성: 질문의 일부를 다루거나 맥락 정보를 제공하는
문서도 고려
4. 답변 가능성: 직접적인 답이 아니더라도 답변 형성에 도움될
정보 포함 여부 평가

[Scoring]
- Rate 'yes' if relevant, 'no' if not
- Default to 'no' when uncertain

[Key points]
- Consider the full context of the query, not just wo
rd matching
- Rate as relevant if useful information is present,
```

```
even if not a complete answer

Your evaluation is crucial for improving information
retrieval systems. Provide balanced assessments."""

# 채점 프롬프트 템플릿 생성
grade_prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    ("human", "[Retrieved document]\n{document}\n\n[U
ser question]\n{question}"),
])

# Retrieval Grader 파이프라인 구성
retrieval_grader_binary = grade_prompt | structured_l
lm_grader

# 관련성 평가 실행
question = "대표 메뉴는 무엇인가요?"
retrieved_docs = menu_db.similarity_search(question,
k=2)
print(f"검색된 문서 수: {len(retrieved_docs)}")
print("==========================================
================================")
print()


relevant_docs = []

for doc in retrieved_docs:
    print("문서:", doc.page_content)
    print("----------------------------------------
--------------------------------")

    relevance = retrieval_grader_binary.invoke({"ques
tion": question, "document": doc})
    print(f"문서 관련성: {relevance}")

    if relevance.binary_score == 'yes':
```

```
        relevant_docs.append(doc)

    print("==========================================
==================================")
```

2. Answer Generator

```
# 기본 RAG 체인
from langchain_core.output_parsers import StrOutputPa
rser

def generator_rag_answer(question, docs):

    template = """
    Answer the question based solely on the given con
text. Do not use any external information or knowledg
e.

    [Instructions]
        1. 질문과 관련된 정보를 문맥에서 신중하게 확인합니
다.
        2. 답변에 질문과 직접 관련된 정보만 사용합니다.
        3. 문맥에 명시되지 않은 내용에 대해 추측하지 않습니
다.
        4. 불필요한 정보를 피하고, 답변을 간결하고 명확하게
작성합니다.
        5. 문맥에서 답을 찾을 수 없으면 "주어진 정보만으로는
답할 수 없습니다."라고 답변합니다.
        6. 적절한 경우 문맥에서 직접 인용하며, 따옴표를 사용
합니다.

    [Context]
    {context}

    [Question]
    {question}

    [Answer]
```

```
    """

    prompt = ChatPromptTemplate.from_template(templat
e)
    llm = ChatOpenAI(model='gpt-4o-mini', temperature
=0)

    def format_docs(docs):
        return "\n\n".join([d.page_content for d in d
ocs])

    rag_chain = prompt | llm | StrOutputParser()

    generation = rag_chain.invoke({"context": format_
docs(docs), "question": question})

    return generation


# 관령성 평가를 통과한 문서를 기반으로 질문에 대한 답변 생성
generation = generator_rag_answer(question, docs=rele
vant_docs)
print(generation)
```

3. Hallucination Grader

```
# 환각(Hallucination) 평가 결과를 위한 데이터 모델 정의
class GradeHallucinations(BaseModel):
    """Binary score for hallucination present in gene
ration answer."""

    binary_score: str = Field(
        description="Answer is grounded in the facts,
'yes' or 'no'"
    )

# LLM 모델 초기화 및 구조화된 출력 설정
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

```
structured_llm_grader = llm.with_structured_output(Gr
adeHallucinations)

# 환각 평가를 위한 시스템 프롬프트 정의
system_prompt = """
You are an expert evaluator assessing whether an LLM-
generated answer is grounded in and supported by a gi
ven set of facts.

[Your task]
    - Review the LLM-generated answer.
    - Determine if the answer is fully supported by t
he given facts.

[Evaluation criteria]
    - 답변에 주어진 사실이나 명확히 추론할 수 있는 정보 외의
내용이 없어야 합니다.
    - 답변의 모든 핵심 내용이 주어진 사실에서 비롯되어야 합니
다.
    - 사실적 정확성에 집중하고, 글쓰기 스타일이나 완전성은 평
가하지 않습니다.

[Scoring]
    - 'yes': The answer is factually grounded and ful
ly supported.
    - 'no': The answer includes information or claims
not based on the given facts.

Your evaluation is crucial in ensuring the reliabilit
y and factual accuracy of AI-generated responses. Be
thorough and critical in your assessment.
"""

# 환각 평가 프롬프트 템플릿 생성
hallucination_prompt = ChatPromptTemplate.from_messag
es(
    [
        ("system", system_prompt),
```

```
        ("human", "[Set of facts]\n{documents}\n\n[LL
M generation]\n{generation}"),
    ]
)

# Hallucination Grader 파이프라인 구성
hallucination_grader = hallucination_prompt | structu
red_llm_grader

# 환각 평가 실행
hallucination = hallucination_grader.invoke({"documen
ts": relevant_docs, "generation": generation})
print(f"환각 평가: {hallucination}")
```

4. Answer Grader

```
# 답변 평가 결과를 위한 데이터 모델 정의
class BinaryGradeAnswer(BaseModel):
    """Binary score to assess answer addresses questi
on."""

    binary_score: str = Field(
        description="Answer addresses the question,
'yes' or 'no'"
    )

# LLM 모델 초기화 및 구조화된 출력 설정
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
structured_llm_grader = llm.with_structured_output(Bi
naryGradeAnswer)

# 답변 평가를 위한 시스템 프롬프트 정의
system_prompt = """
You are an expert evaluator tasked with assessing whe
ther an LLM-generated answer effectively addresses an
d resolves a user's question.

[Your task]
```

```
    - Carefully analyze the user's question to unders
tand its core intent and requirements.
    - Determine if the LLM-generated answer sufficien
tly resolves the question.

[Evaluation criteria]
    - 관련성: 답변이 질문과 직접적으로 관련되어야 합니다.
    - 완전성: 질문의 모든 측면이 다뤄져야 합니다.
    - 정확성: 제공된 정보가 정확하고 최신이어야 합니다.
    - 명확성: 답변이 명확하고 이해하기 쉬워야 합니다.
    - 구체성: 질문의 요구 사항에 맞는 상세한 답변이어야 합니
다.

[Scoring]
    - 'yes': The answer effectively resolves the ques
tion.
    - 'no': The answer fails to sufficiently resolve
the question or lacks crucial elements.

Your evaluation plays a critical role in ensuring the
quality and effectiveness of AI-generated responses.
Strive for balanced and thoughtful assessments.
"""

# 답변 평가 프롬프트 템플릿 생성
answer_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt),
        ("human", "[User question]\n{question}\n\n[LL
M generation]\n{generation}"),
    ]
)

# Answer Grader 파이프라인 구성
answer_grader_binary = answer_prompt | structured_llm
_grader

# 답변 평가 실행
```

```
print("Question:", question)
print("Generation:", generation)

answer_score = answer_grader_binary.invoke({"questio
n": question, "generation": generation})
print(f"답변 평가: {answer_score}")
```

5. Question Re-writer

```
def rewrite_question(question: str) -> str:
    """
    주어진 질문을 벡터 저장소 검색에 최적화된 형태로 다시 작
성합니다.

    :param question: 원본 질문 문자열
    :return: 다시 작성된 질문 문자열
    """
    # LLM 모델 초기화
    llm = ChatOpenAI(model="gpt-4o-mini", temperature
=0)

    # 시스템 프롬프트 정의
    system_prompt = """
    You are an expert question re-writer. Your task i
s to convert input questions into optimized versions
    for vectorstore retrieval. Analyze the input care
fully and focus on capturing the underlying semantic
    intent and meaning. Your goal is to create a ques
tion that will lead to more effective and relevant
    document retrieval.

    [Guidelines]
        1. 질문에서 핵심 개념과 주요 대상을 식별하고 강조합
니다.
        2. 약어나 모호한 용어를 풀어서 사용합니다.
        3. 관련 문서에 등장할 수 있는 동의어나 연관된 용어를
포함합니다.
        4. 질문의 원래 의도와 범위를 유지합니다.
```

```
        5. 복잡한 질문은 간단하고 집중된 하위 질문으로 나눕
니다.

    Remember, the goal is to improve retrieval effect
iveness, not to change the fundamental meaning of the
question.
    """

    # 질문 다시 쓰기 프롬프트 템플릿 생성
    re_write_prompt = ChatPromptTemplate.from_message
s(
        [
            ("system", system_prompt),
            (
                "human",
                "[Initial question]\n{question}\n\n[I
mproved question]\n",
            ),
        ]
    )

    # 질문 다시 쓰기 체인 구성
    question_rewriter = re_write_prompt | llm | StrOu
tputParser()

    # 질문 다시 쓰기 실행
    rewritten_question = question_rewriter.invoke({"q
uestion": question})

    return rewritten_question

# 질문 다시 쓰기 테스트
rewritten_question = rewrite_question(question)
print(f"원본 질문: {question}")
print(f"다시 쓴 질문: {rewritten_question}")
```

```
# 다시 쓴 질문을 사용하여 벡터 저장소에서 문서 검색
query = rewritten_question
```

```
retrieved_docs = menu_db.similarity_search(query, k=
2)
print(len(retrieved_docs))
print("==========================================
============================")

for doc in retrieved_docs:
    print("문서:", doc.page_content)
    print("----------------------------------------
--------------------------------")
```

LangGraph로 그래프 구현

- 기본 State 사용

1. 그래프 state 생성

```
from typing import List, TypedDict
from langchain_core.documents import Document

class SelfRagState(TypedDict):
    question: str                # 사용자의 질문
    generation: str              # LLM 생성 답변
    documents: List[Document]    # 컨텍스트 문서 (검색된
문서)
    num_generations: int         # 질문 or 답변 생성 횟수
(무한 루프 방지에 활용)
```

2. Node 구성

```
def retrieve_menu_self(state: SelfRagState):
    """문서를 검색하는 함수"""
    print("--- 문서 검색 ---")
    question = state["question"]

    # 문서 검색 로직
    documents = menu_db.similarity_search(question, k=2)
    return {"documents": documents}      # 가장 마지막에
```

```
검색한 문서 객체들로 상태를 업데이트

def generate_self(state: SelfRagState):
    """답변을 생성하는 함수"""
    print("--- 답변 생성 ---")
    question = state["question"]
    documents = state["documents"]

    # RAG를 이용한 답변 생성
    generation = generator_rag_answer(question, docs=doc
uments)

    # 생성 횟수 업데이트
    num_generations = state.get("num_generations", 0)
    num_generations += 1
    return {"generation": generation, "num_generations":
num_generations}      # 답변, 생성횟수 업데이트


def grade_documents_self(state: SelfRagState):
    """검색된 문서의 관련성을 평가하는 함수"""
    print("--- 문서 관련성 평가 ---")
    question = state["question"]
    documents = state["documents"]

    # 각 문서 평가
    filtered_docs = []
    for d in documents:
        score = retrieval_grader_binary.invoke({"questio
n": question, "document": d})
        grade = score.binary_score
        if grade == "yes":
            print("---문서 관련성: 있음---")
            filtered_docs.append(d)
        else:
            print("---문서 관련성: 없음---")

    return {"documents": filtered_docs}       # 관련성 평
```

```
가에 합격한 문서들만 저장 (override)


def transform_query_self(state: SelfRagState):
    """질문을 개선하는 함수"""
    print("--- 질문 개선 ---")
    question = state["question"]

    # 질문 재작성
    rewritten_question = rewrite_question(question)

    # 생성 횟수 업데이트
    num_generations = state.get("num_generations", 0)
    num_generations += 1
    return {"question": rewritten_question, "num_generat
ions": num_generations}      # 재작성한 질문을 저장, 생성횟
수 업데이트
```

3. Edge 구성

```
def decide_to_generate_self(state: SelfRagState):
    """답변 생성 여부를 결정하는 함수"""

    num_generations = state.get("num_generations", 0)
    if num_generations > 2:
        print("--- 결정: 생성 횟수 초과, 답변 생성 (-> gener
ate)---")
        return "generate"

    print("--- 평가된 문서 분석 ---")
    filtered_documents = state.get("documents", None)

    if not filtered_documents:
        print("--- 결정: 모든 문서가 질문과 관련이 없음, 질문
개선 필요 (-> transform_query)---")
        return "transform_query"
    else:
        print("--- 결정: 답변 생성 (-> generate)---")
```

```
        return "generate"


def grade_generation_self(state: SelfRagState):
    """생성된 답변의 품질을 평가하는 함수"""

    num_generations = state.get("num_generations", 0)
    if num_generations > 2:
        print("--- 결정: 생성 횟수 초과, 종료 (-> END)---")
        return "end"

    # 1단계: 환각 여부 확인
    print("--- 환각 여부 확인 ---")
    question, documents, generation = state["question"],
state["documents"], state["generation"]

    hallucination_grade = hallucination_grader.invoke(
        {"documents": documents, "generation": generatio
n}
    )

    if hallucination_grade.binary_score == "yes":
        print("--- 결정: 환각이 없음 (답변이 컨텍스트에 근거
함) ---")

        # 1단계 통과할 경우 -> 2단계: 질문-답변 관련성 평가
        print("---질문-답변 관련성 확인---")
        relevance_grade = answer_grader_binary.invoke
({"question": question, "generation": generation})
        if relevance_grade.binary_score == "yes":
            print("--- 결정: 생성된 답변이 질문을 잘 다룸 (->
END) ---")
            return "useful"
        else:
            print("--- 결정: 생성된 답변이 질문을 제대로 다루
지 않음 (-> transform_query) ---")
            return "not useful"
    else:
```

```
        print("--- 결정: 생성된 답변이 문서에 근거하지 않음,
재시도 필요 (-> generate) ---")
        return "not supported"
```

4. 그래프 연결

```
from langgraph.graph import StateGraph, START, END
from IPython.display import Image, display

# 워크플로우 그래프 초기화
builder = StateGraph(SelfRagState)

# 노드 정의
builder.add_node("search_menu", retrieve_menu_self)
# 문서 검색
builder.add_node("grade_documents", grade_documents_sel
f)  # 문서 평가
builder.add_node("generate", generate_self)
# 답변 생성
builder.add_node("transform_query", transform_query_sel
f)  # 질문 개선

# 그래프 구축
builder.add_edge(START, "search_menu")
builder.add_edge("search_menu", "grade_documents")

# 조건부 엣지 추가: 문서 평가 후 결정
builder.add_conditional_edges(
    "grade_documents",
    decide_to_generate_self,
    {
        "transform_query": "transform_query",
        "generate": "generate",
    },
)
builder.add_edge("transform_query", "search_menu")

# 조건부 엣지 추가: 답변 생성 후 평가
```

```
builder.add_conditional_edges(
    "generate",
    grade_generation_self,
    {
        "not supported": "generate",          # 환각이 발생한 경우 -> 답변을 다시 생성
        "not useful": "transform_query",      # 질문과 답변의 관련성이 부족한 경우 -> 쿼리 개선해서 다시 검색
        "useful": END,
        "end": END,
    },
)

# 그래프 컴파일
self_rag = builder.compile()

# 그래프 시각화
display(Image(self_rag.get_graph().draw_mermaid_png()))
```

5. 그래프 실행

```
inputs = {"question": "이 식당의 대표 메뉴는 무엇인가요? 주재료는 무엇인가요?"}
final_output = self_rag.invoke(inputs)
```

```
# 최종 답변
final_output["generation"]
```

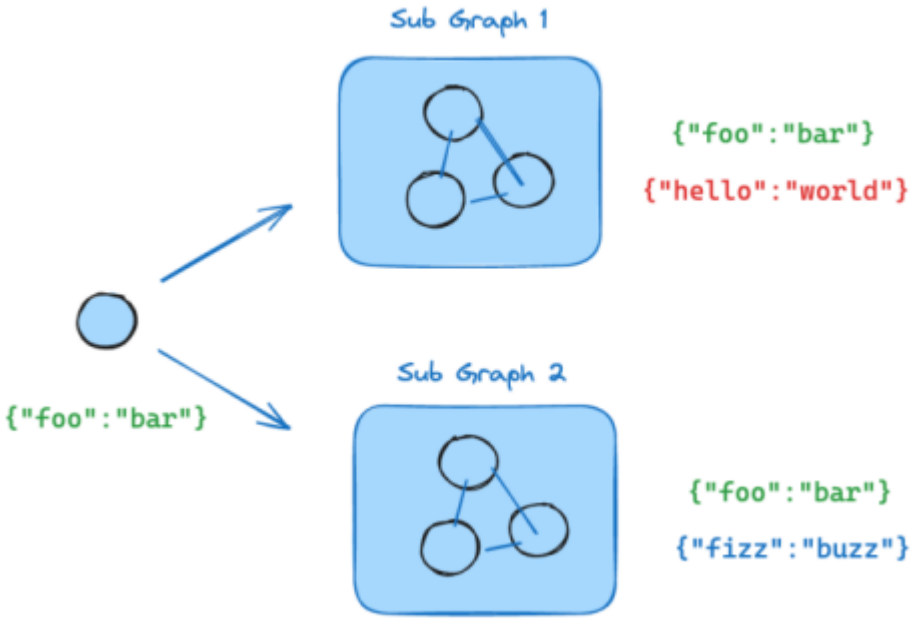
```
inputs = {"question": "김치를 재료로 하는 메뉴가 있나요?"}
for output in self_rag.stream(inputs):
    for key, value in output.items():
        # 노드 출력
        pprint(f"Node '{key}':")
        pprint(f"Value: {value}", indent=2, width=80, depth=None)
```

```
    print("\n----------------------------------------------
---------------\n")
```

```
# 최종 답변
print(value["generation"])
```

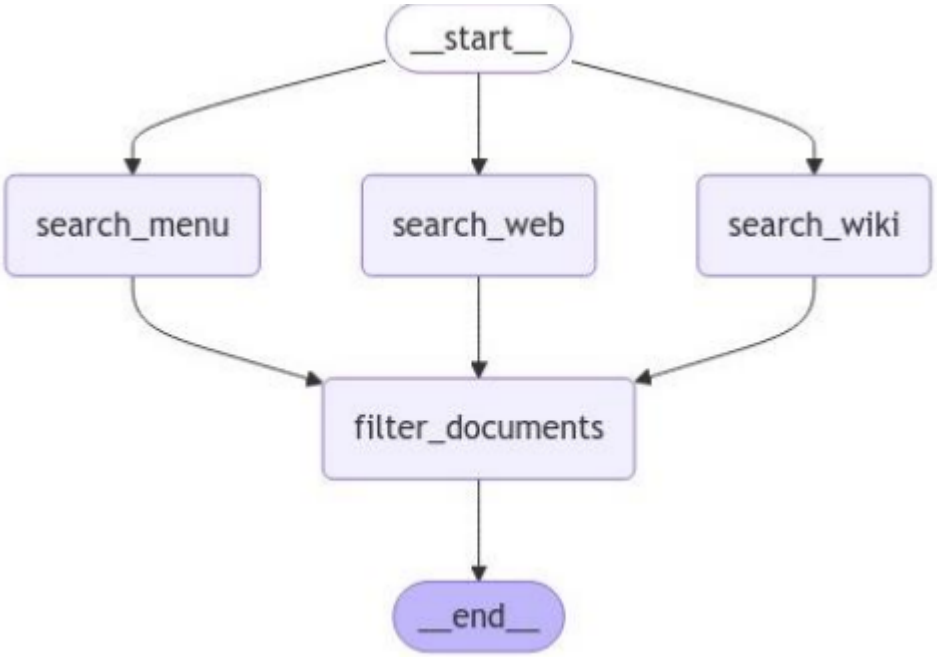
서브그래프 (Subgraphs)

- 주요 특징
  - 각 서브 그래프는 독립적인 상태 관리 가능
  - 메인 그래프와 서브 그래프 간의 정보 교환 지원
  - 모듈화된 설계로 복잡한 워크플로우 구현 용이
- 장점
  - 모듈성 향상 : 복잡한 시스템을 관리 가능한 단위로 분할
  - 재사용성 : 서브그래프를 다른 프로젝트나 컨텍스트에서 재활용 가능
  - 유연성 : 시스템의 일부분만 수정하거나 확장하기 쉬움
  - 디버깅 용이성 : 각 서브그래프를 독립적으로 테스트하고 디버그 가능
- 구현 방법
  - 서브 그래프 상태 정의
  - 서브 그래프 노드 및 엣지 구성
  - 메인 그래프에 서브 그래프 통합



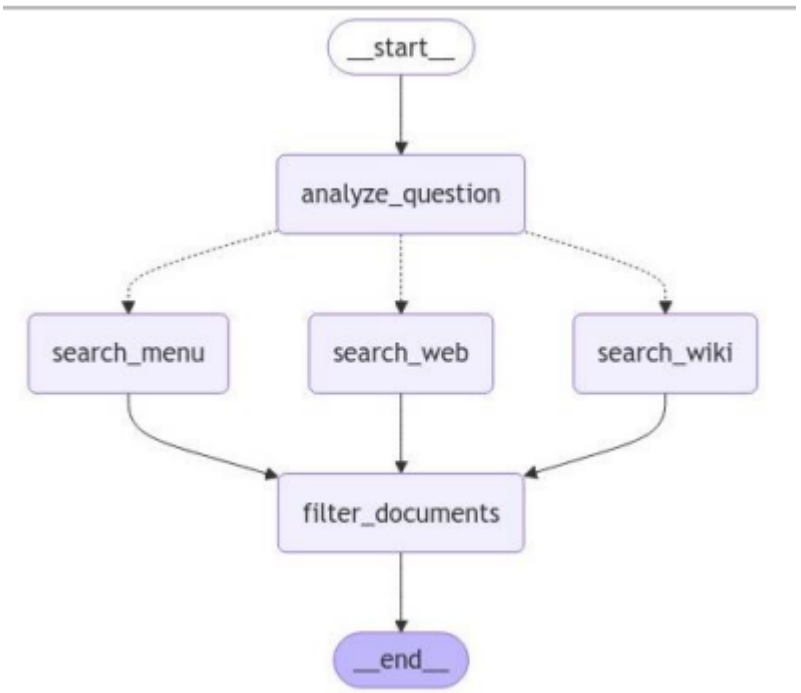
- 병렬 노드 실행
  - 전체 그래프 작업속도를 높이는 데 필수
  - LangGraph는 노드의 병렬 실행을 기본적으로 지원
  - 팬아웃(fan-out)과 팬인(fan-in) 메커니즘을 통해 구현

```
builder.add_edge(START, "search_menu")
builder.add_edge(START, "search_web")
builder.add_edge(START, "search_wiki")
builder.add_edge("search_menu", "filter_documents")
builder.add_edge("serach_web", "filter_documents")
builder.add_edge("serach_wiki", "filter_documents")
builder.add_edge("filter_documents", END)
```

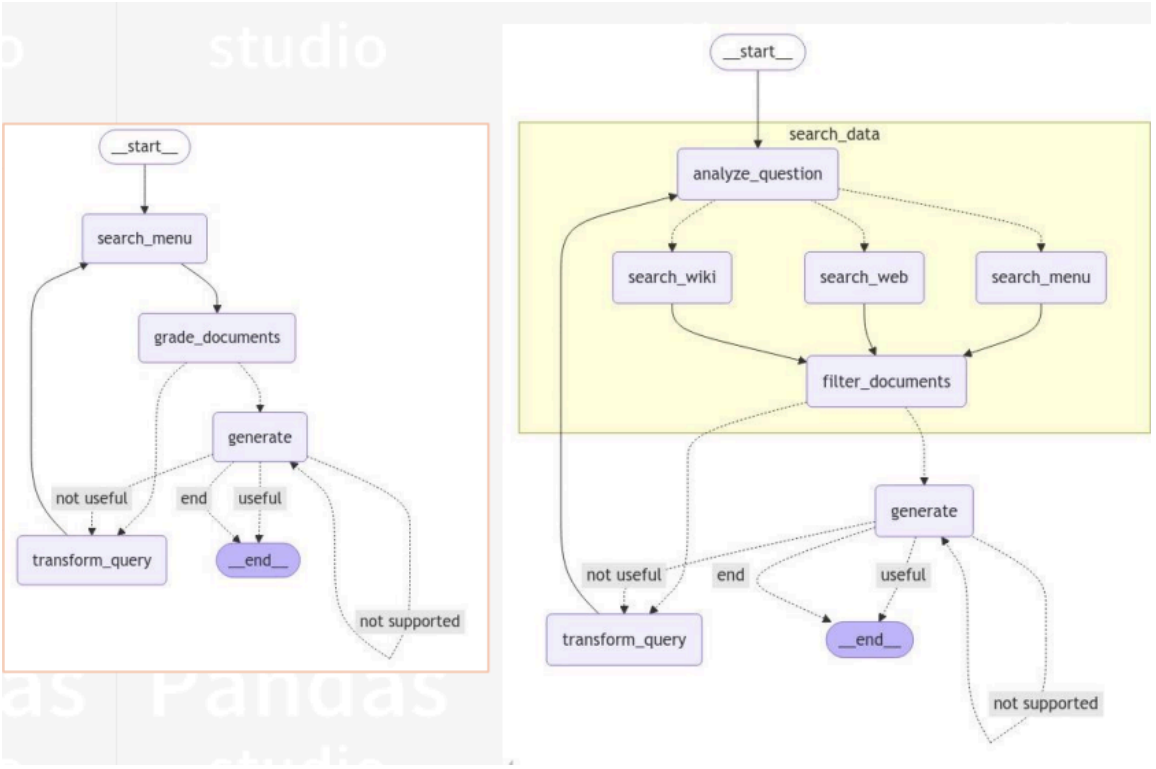


- 조건부 엣지로 병렬 노드 실행
  - 실행 시간에 동적으로 경로를 결정
  - 특정 조건에 따라 다른 노드 세트를 병렬로 실행 가능

```
search_builder.add_edge(START, "analyze_question")
search_builder.add_conditional_edges(
    "analyze_question",
    route_datasources,
    ["search_menu", "search_web", "search_wiki"]
)
search_builder.add_edge("search_menu", "filter_document
s")
search_builder.add_edge("search_web", "filter_document
s")
search_builder.add_edge("search_wiki", "filter_document
s")
serach_builder.add_edge("filter_documents", END)
```



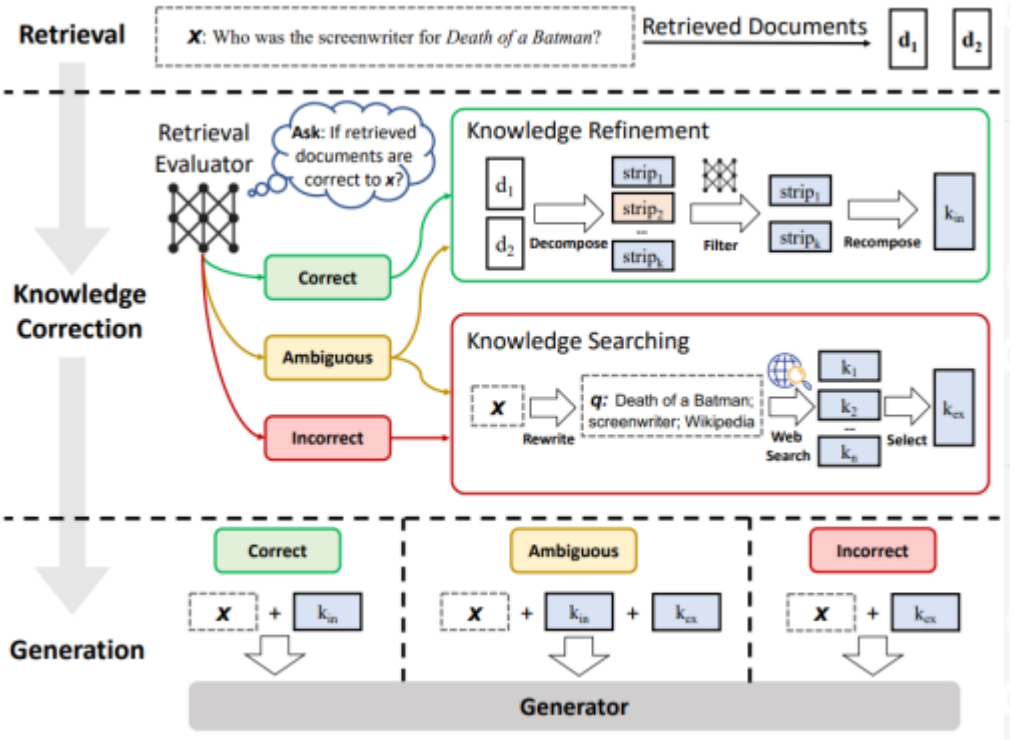
- 기존 Self-RAG 그래프와 결합

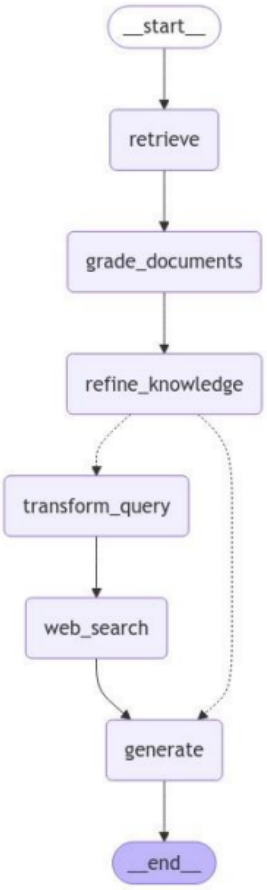


Corrective RAG

- 기존 RAG 시스템을 개선해 검색된 정보의 품질과 관련성을 향상시키는 접근 방식

- 문서 관련성 평가, 지식 정제, 필요 시 외부 지식 탐색, 정제된 지식을 바탕으로 한 답변 생성
- CRAG 작동 방식
  1. 문서 관련성 평가 : Retrieval 문서에 대한 평가
  2. 지식 정제 : Retrieval 문서 중에서 중요한 정보만 추출
  3. 지식 검색 : Retrieval 문서가 부족할 경우, 외부 지식(웹)을 탐색
  4. 답변 생성 : 정제 지식을 활용해 답변 생성





- 주요 과정

  검색 → 평가 → 지식 정제 또는 웹 검색 → 답변 생성

  1. 문서 관련성 평가 ('grade_documents'):
     - 각 문서의 관련성 평가
     - 기준을 통과하는 문서만을 유지
  2. 지식 정제 ('refine_knowledge'):
     - 문서를 지식 조각으로 분할하고 각각 관련성 평가
     - 관련성 높은(0.5 초과) 지식 조각만 유지
  3. 웹 검색 ('web_search'):
     - 문서에 충분한 정보가 없는 경우 외부 지식 활용
     - 웹 검색 결과를 기존 문서에 추가
  4. 답변 생성 ('generate_answer'):
     - 정제된 지식 조각을 사용해 답변 생성
     - 관련 정보가 없을 경우 적절한 메시지를 반환

```
# 노드 정의
builder.add_node("retrieve", retrieve)        # 문서 검색
builder.add_node("grade_documents", grade_documents)    #
문서 평가
builder.add_node("refine_knowledge", refine+knowledge)    #
지식 정제
builder.add_node("web_search", web_search)        # 웹 검색
builder.add_node("generate", generate)        # 답변 생성
builder.add_node("transform_query", transform_query)    #
질문 개선

# 경로 정의
builder.add_edge(START, "retrieve")
builder.add_edge("retrieve", "grade_documents")
builder.add_edge("grade_documents", "refine_knowledge")

# 조건부 엣지 추가 : 문서 평가 후 결정
```

```
builder.add_conditional_edges(
    "refine_knowledge",
    decide_to_generate,
    {
        "transform_query": "transform_query",
        "generate": "generate",
    },
)

# 추가 경로
builder.add_edge("transform_query", "web_search")
builder.add_edge("web_search", "generate")
builder.add_edge("generate", E
```