랭그래프(LangGraph) 활용 에이전트 RAG 구현

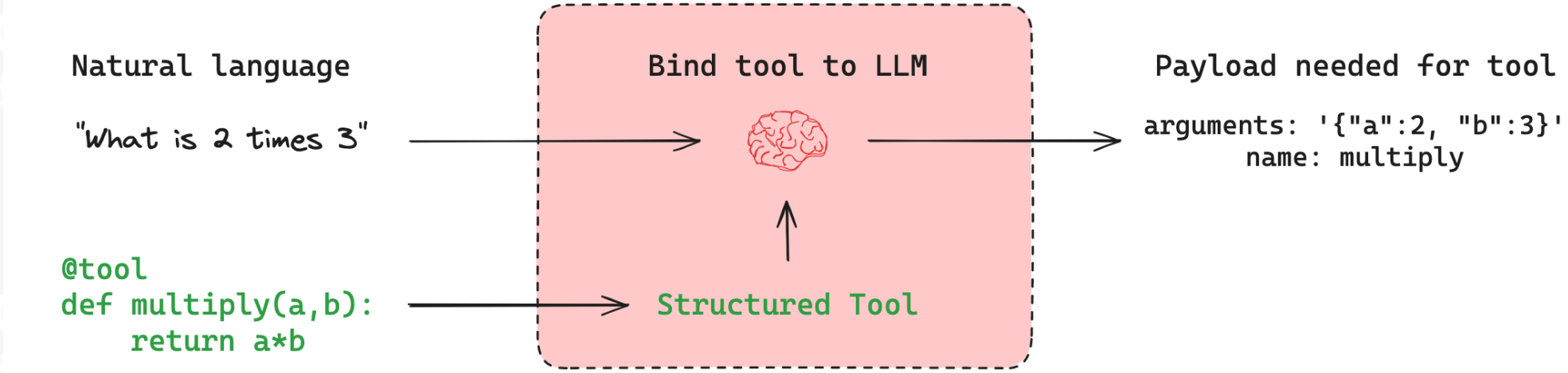
LangGraph 기본 사용법부터 Modular RAG 까지

주요 내용

- LangChain ToolCalling
- LangChain Agent
- State Graph
- Message Graph (Reducer)
- ReAct (Memory)
- Adaptive RAG (Human-in-the-Loop)
- Self-RAG (sub-graph)
- Corrective RAG

LangChain ToolCalling 개요

- Tool Calling 개념: LLM이 **외부 기능이나 데이터에 접근**할 수 있게 해주는 메커니즘
- 중요성: LLM의 한계(최신 정보 부족, 특정 작업 수행 불가 등)를 극복하는 방법 (**RAG에서도 중요**)
- LLM과 외부 도구 연동의 필요성: 실시간 데이터 접근, 특수 기능 수행, 정확성 향상 등



<이미지 출처> https://langchain-ai.github.io/langgraph/concepts/agentic_concepts/#tool-calling-agent

LangChain 에서 제공하는 내장 도구(Tool)

- 랭체인은 검색, 코드 인터프리터, 생산성 도구 등 다양한 Tool(도구)을 직접/제휴 형태로 제공

  (검색 : DuckDuckGo, TavilySearch 등)

Search

The following table shows tools that execute online searches in some shape or form:

Tool/Toolkit	Free/Paid	Return Data
Bing Search	Paid	URL, Snippet, Title
Brave Search	Free	URL, Snippet, Title
DuckDuckgoSearch	Free	URL, Snippet, Title

https://python.langchain.com/docs/integrations/tools/

- 예시: **Tavily 웹 검색 도구** 사용
  - AI 기반의 웹 검색 API를 제공하는 서비스
  - 인증키: 환경변수 TAVILY_API_KEY 설정

```
name:
tavily_search_results_json
--------------------------------------------------------------------
description:
A search engine optimized for comprehensive, accurate, and trusted
results. Useful for when you need to answer questions about current
events. Input should be a search query.
--------------------------------------------------------------------
args:
{'query': {'description': 'search query to look up', 'title': 'Query',
'type': 'string'}}
--------------------------------------------------------------------
```

Tool의 구성 요소:

- name: 도구의 이름
- description: 도구가 수행하는 작업에 대한 설명
- JSON schema: 도구의 입력을 정의하는 스키마
- function: 실행할 함수 (선택적으로 비동기 함수도 가능)

LangChain 에서 제공하는 내장 도구(Tool)

- 랭체인은 검색, 코드 인터프리터, 생산성 도구 등 다양한 Tool(도구)을 직접/제휴 형태로 제공

  (검색 : DuckDuckGo, TavilySearch 등)

Search

The following table shows tools that execute online searches in some shape or form:

Tool/Toolkit	Free/Paid	Return Data
Bing Search	Paid	URL, Snippet, Title
Brave Search	Free	URL, Snippet, Title
DuckDuckgoSearch	Free	URL, Snippet, Title

https://python.langchain.com/docs/integrations/tools/

- 예시: **Tavily 웹 검색 도구** 사용
  - AI 기반의 웹 검색 API를 제공하는 서비스
  - 인증키: 환경변수 TAVILY_API_KEY 설정

```
name:
tavily_search_results_json
--------------------------------------------------------------------
description:
A search engine optimized for comprehensive, accurate, and trusted
results. Useful for when you need to answer questions about current
events. Input should be a search query.
--------------------------------------------------------------------
args:
{'query': {'description': 'search query to look up', 'title': 'Query',
'type': 'string'}}
--------------------------------------------------------------------
```

Tool의 구성 요소:

- name: 도구의 이름
- description: 도구가 수행하는 작업에 대한 설명
- JSON schema: 도구의 입력을 정의하는 스키마
- function: 실행할 함수 (선택적으로 비동기 함수도 가능)

Tavily 웹 검색 도구 사용

LangChain ToolCalling

Pandas studio

도구 직접 실행

```python
from langchain_community.tools import TavilySearchResults

# 검색할 쿼리 설정
query = "스테이크와 어울리는 와인을 추천해주세요."

# Tavily 검색 도구 초기화 (최대 2개의 결과 반환)
web_search = TavilySearchResults(max_results=2)

# 웹 검색 실행
search_results = web_search.invoke(query)
```

그대로 검색

```
{'url': 'https://blog.naver.com/PostView.nhn?blogId=cyahnnn&logNo=222766631086', 'content': '스테이크와 어울리는 와인 : 네이버 블로그 변경 전 공유된 블로그/글/클립 링크는 연결이 끊길 수 있습니다. 블로그 블로그 블로그 블로그 카베르네 소비뇽(Cabernet Sauvignon) 및 말벡(Malbec) 켓(Fiona Becket)이 2007년 디캔터에서 스테이크와 함께 스테이크와 궁합이라고 생각하지 않지만, 고기를 레어(rare)로 요리했을 때 지금까지 최고의 궁합은 클래식하게 실크처럼 부드럽고 매혹적인 다니엘 리옹의 본 로마네(Daniel Rion, Vosne-Romanée 2001)이다'라고 썼다.'} ---------- {'url': 'https://secrettsteaks.com/blog/steak-wine-pairing.php', 'content': '스테이크와 가장 잘 어울리는 와인을 추천해드리며, 어떤 와인이 여러분의 식사 경험을 한층 더 업그레이드할 수 있을지 알아보겠습니다. 첫 번째로 추천하는 와인은 카베르네 소비뇽(Cabernet Sauvignon)입니다.'} ----
```

Tool Calling

```python
from langchain_openai import ChatOpenAI

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model="gpt-4o-mini")

# 웹 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=[web_search])

# 쿼리를 LLM에 전달하여 결과 얻기
ai_msg = llm_with_tools.invoke(query)
```

LLM이 검색에 맞는 적절한 검색어를 생성

```
content='' additional_kwargs={'tool_calls': [{'id': 'call_5lPOU0gWyOJXqVR5PZsY6uPa', 'function': {'arguments': '{"query": "스테이크에 어울리는 와인 추천"}', 'name': 'tavily_search_results_json'}, 'type': 'function'}, {'id': 'call_iC7y2CjvSR0vLxUoNLAIYFel', 'function': {'arguments': '{"query": "스테이크와 와인 페어링 팁"}', 'name': 'tavily_search_results_json'}, 'type': 'function'}], 'refusal': None}
 … …
 tool_calls=[{'name': 'tavily_search_results_json', 'args': {'query': '스테이크에 어울리는 와인 추천'}, 'id': 'call_5lPOU0gWyOJXqVR5PZsY6uPa', 'type': 'tool_call'}, {'name': 'tavily_search_results_json', 'args': {'query': '스테이크와 와인 페어링 팁'}, 'id': 'call_iC7y2CjvSR0vLxUoNLAIYFel', 'type': 'tool_call'}] usage_metadata={'input_tokens': 91, 'output_tokens': 71, 'total_tokens': 162}
```

Tool Calling 결과를 가지고 검색 작업을 실행

LangChain ToolCalling

Pandas studio

args 스키마 사용

```
tool_call = ai_msg.tool_calls[0]
tool_output = web_search.invoke(tool_call["args"])
```


{'url': 'https://blog.naver.com/PostView.nhn?blogId=cyahnnn&logNo=222766631086', 'content': '스테이크와 어울리는 와인 : 네이버 블로그 변경 전 공유된 블로그/글/클립 링크는 연결이 끊길 수 있습니다. 블로그 블로그 블로그 블로그 카베르네 소비뇽(Cabernet Sauvignon) 및 말벡(Malbec) 켓(Fiona Becket)이 2007년 디캔터에서 스테이크와 함께 스테이크와 궁합이라고 생각하지 않지만, 고기를 레어(rare)로 요리했을 때 지금까지 최고의 궁합은 클래식하게 실크처럼 부드럽고 매혹적인 다니엘 리옹의 본 로마네(Daniel Rion, Vosne-Romanée 2001)이다'라고 썼다.'} ---------- -- {'url': 'https://secrettsteaks.com/blog/steak-wine-pairing.php', 'content': '스테이크와 가장 잘 어울리는 와인을 추천해드리며, 어떤 와인이 여러분의 식사 경험을 한층 더 업그레이드할 수 있을지 알아보겠습니다. 첫 번째로 추천하는 와인은 카베르네 소비뇽(Cabernet Sauvignon)입니다.'} ----

tool_call 사용

```
tool_message = web_search.invoke(tool_call)
```


ToolMessage(content='[{"url": "https://secrettsteaks.com/blog/steak-wine-pairing.php", "content": "풍미를 보기"}, {"url": "https://blog.naver.com/PostView.nhn?blogId=cyahnnn&logNo=222766631086", "content": "스테이크와 어울리는 와인 : 블로그 썼다."}]', name='tavily_search_results_json', tool_call_id='call_5lPOU0gWyOJXqVR5PZsY6uPa', artifact={'query': '스테이크에 어울리는 와인 추천', 'follow_up_questions': None, 'answer': None, 'images': [], 'results': [{'title': '스테이크와 어울리는 와인 추천 - secrettsteaks.com', 'url': 'https://secrettsteaks.com/blog/steak-wine-pairing.php', 'content': '스테이크와 ', 'score': 0.9998074, 'raw_content': None}, {'title': '스테이크와 어울리는 와인 : 네이버 ... - 네이버 블로그', 'url': 'https://blog.naver.com/PostView.nhn?blogId=cyahnnn&logNo=222766631086', 'content': '스테이크와 어울리는 와인 : 스테이크와 궁합이라고 썼다.', 'score': 0.99961853, 'raw_content': None}], 'response_time': 1.84})

ToolMessage 정의

```
tool_message = ToolMessage(
    content=tool_output,
    tool_call_id=tool_call["id"],
    name=tool_call["name"]
)
```


ToolMessage(content=[{'url': 'https://secrettsteaks.com/blog/steak-wine-pairing.php', 'content': '스테이크와 가장 잘 어울리는 와인의 부드러운 맛은 스테이크의 풍미를 덮지 않고 자연스럽게 어우러져, 부담 없이 즐길 수 있는 조합을 만들어줍니다. 개인정보 처리 방침 개인정보 처리 방침 보기'}, {'url': 'https://blog.naver.com/PostView.nhn?blogId=cyahnnn&logNo=222766631086', 'content': '스테이크와 어울리는 와인 : 네이버 블로그 변경 전 공유된 블로그/글/클립 링크는 연결이 끊길 수 있습니다. 블로그 블로그 블로그 블로그 카베르네 소비뇽지금까지 최고의 궁합은 클래식하게 실크처럼 부드럽고 매혹적인 다니엘 리옹의 본 로마네(Daniel Rion, Vosne-Romanée 2001)이다'라고 썼다.'}], name='tavily_search_results_json', tool_call_id='call_5lPOU0gWyOJXqVR5PZsY6uPa')

ToolMessage를 LLM에 전달하여 AI 답변을 생성하기

LangChain ToolCalling

Pandas studio

LLM 체인을 정의

```
# 오늘 날짜 설정
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
prompt = ChatPromptTemplate([
    ("system", f"You are a helpful AI assistant"),
    ("system", f"Today's date is {today}."),
    ("human", "{user_input}"),
    ("placeholder", "{messages}"),
])

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model="gpt-4o-mini")

# LLM에 도구를 바인딩
llm_with_tools = llm.bind_tools(tools=[web_search_tool])

# LLM 체인 생성
llm_chain = prompt | llm_with_tools
```

LLM 체인에 ToolMessage 전달

```
@chain
def web_search_chain(user_input: str, config:
RunnableConfig):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config=config)
    tool_msgs = web_search_tool.batch(
        ai_msg.tool_calls, config=config
        )
    return llm_chain.invoke(
        {**input_, "messages": [ai_msg, *tool_msgs]},
        config=config
        )

# 체인 실행
web_search_chain.invoke("오늘 모엣샹동의 가격은 얼마인가요?")
```

사용자 질문

검색 결과

검색 요청

```
현재 모엣 샹동(Moët & Chandon)의 가격은 대부분의 병에 대해 평균적으로 약 51
달러에서 65달러 사이입니다. 구체적인 가격은 구매하는 매장이나 지역에 따라 다
를 수 있습니다.
```

사용자 정의 도구(Tool)

- 랭체인은 사용자가 직접 도구를 정의하여 사용하는 방법을 제공
- 가장 대표적인 방법: **@tool데코레이터** 사용
  - 함수를 LangChain 도구로 변환하는 방법
  - 도구 함수 작성 가이드라인: 명확한 입출력 정의, 단일 책임 원칙 준수
  - 도구 설명(description) 작성: LLM이 도구의 기능을 정확히 이해하고 사용하도록 작성

name:
blog_search
---
description:
네이버 블로그 API에 검색 요청을 보냅니다.
---
args:
{'query': {'title': 'Query', 'type': 'string'}}

```python
from langchain_core.tools import tool

@tool
def blog_search(query: str) -> List[Dict]:
    """네이버 블로그 API에 검색 요청을 보냅니다."""
    url = "https://openapi.naver.com/v1/search/blog.json"
    headers = {
        "X-Naver-Client-Id": NAVER_CLIENT_ID,
        "X-Naver-Client-Secret": NAVER_CLIENT_SECRET
    }
    params = {"query": query, "display": 10, "start": 1}
    response = requests.get(url, headers=headers,
                            params=params)
    if response.status_code == 200:
        return response.json()['items']
    else:
        return []

query = "스테이크와 어울리는 와인을 추천해주세요."
search_results = blog_search.invoke(query)
```

```
[{'title': '<b>스테이크와</b> 잘 <b>어울리는</b> 추천</b>', 'link':
'https://blog.naver.com/dolmory9/223556006280', 'description': '레드
<b>와인</b> - Body : Medium-Full ~ Full - Sweetness : Dry - <b>와인
</b><b>추천</b>... ', 'bloggername': '와인과 커피 엔지니어',
'bloggerlink': 'blog.naver.com/dolmory9', 'postdate': '20240821'}, …]
```

Few-shot 프롬프팅을 활용한 ToolCalling 성능 개선

LangChain ToolCalling

Pandas studio

- **Few-shot 프롬프팅**
  - 모델에게 몇 가지 예시를 제공하여 원하는 출력 형식이나 작업 수행 방식을 보여주는 기법
  - 모델에게 도구를 어떻게 사용해야 하는지 예시를 통해 보여주는 목적으로 사용

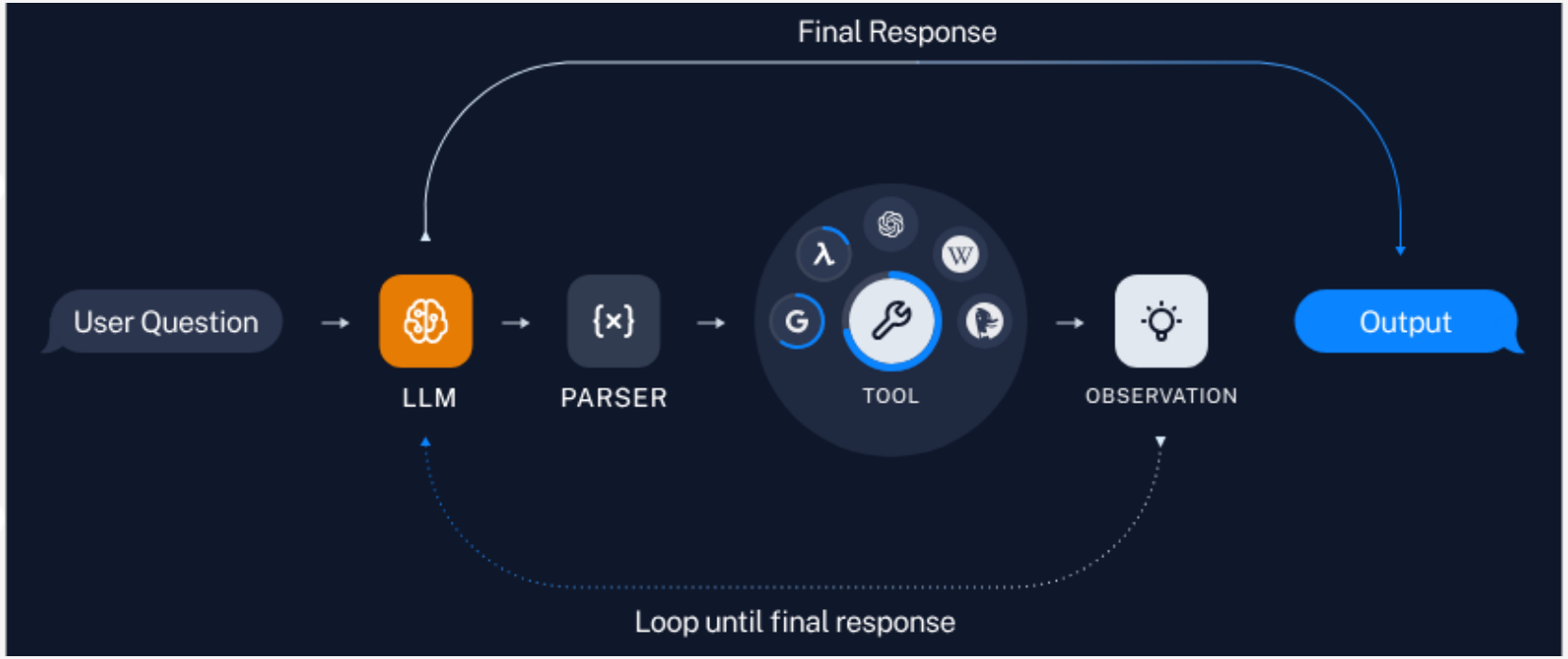
- **ToolCalling 적용 과정**

1. 예시 생성
2. 프롬프트 템플릿 생성
3. 프롬프트 생성 및 모델 호출
4. 응답 파싱

```python
from langchain_core.messages import AIMessage, HumanMessage, ToolMessage
from langchain_core.prompts import ChatPromptTemplate

examples = [
    HumanMessage("트러플 리조또의 가격과 특징, 그리고 사람들의 평가에 대해 알려주세요.", name="example_u
    AIMessage("메뉴 정보를 검색하고, 웹에서 추가 정보를 찾은 후, 블로그에서 사람들의 평가를 찾아보겠습니다
    AIMessage("", name="example_assistant", tool_calls=[{"name": "search_menu", "args": {"query":
    ToolMessage("트러플 리조또: 가격 ₩28,000, 이탈리아 카나롤리 쌀 사용, 블랙 트러플 향과 파르메산 치즈를
    AIMessage("트러플 리조또의 가격은 ₩28,000이며, 이탈리아 카나롤리 쌀을 사용하고 블랙 트러플 향과 파르
    AIMessage("", name="example_assistant", tool_calls=[{"name": "web_search", "args": {"query": "
    ToolMessage("트러플 리조또는 이탈리아 북부 지방의 대표적인 요리로, 크리미한 질감과 독특한 트러플 향이
    AIMessage("트러플 리조또의 특징에 대해 알아보았습니다. 이제 사람들의 평가를 블로그에서 찾아보겠습니다.
    AIMessage("", name="example_assistant", tool_calls=[{"name": "blog_search", "args": {"query":
    ToolMessage("블로그 검색 결과: 트러플 리조또 후기 - 1. '향긋한 트러플 향과 크리미한 식감이 일품이에요
    AIMessage("트러플 리조또(₩28,000)는 이탈리아 북부 지방의 대표적인 요리로, 이탈리아 카나롤리 쌀을 사용
]

system = """당신은 레스토랑 메뉴 정보와 일반적인 음식 관련 지식을 제공하는 AI 어시스턴트입니다.
메뉴 정보는 search_menu 도구를 사용하여 검색하고, 일반적인 정보는 web_search 도구를 사용하세요.
사람들의 평가나 개인적인 경험에 대해서는 blog_search 도구를 사용하여 검색하세요.
"""

few_shot_prompt = ChatPromptTemplate.from_messages([
    ("system", system),
    *examples,
    ("human", "{query}"),
])

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model="gpt-4o")

# 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=[web_search_tool, blog_search, search_menu])

# Few-shot 프롬프트를 사용한 체인 구성
fewshot_search_chain = few_shot_prompt | llm_with_tools
```

LangChain Agent 개요

- **LangChain Agent란**:
  - 정의: LLM을 추론 엔진으로 사용하여 어떤 행동을 할지, 그 행동의 입력은 무엇일지 결정하는 시스템
  - 목적: 언어 모델이 단순히 텍스트를 출력하는 것을 넘어 실제 행동을 취하게 함
  - 작동 방식: 행동의 결과를 다시 Agent에 피드백하여 추가 행동할 지 또는 작업을 완료할 지를 결정



<이미지 출처> https://python.langchain.com/v0.1/docs/use_cases/tool_use/agents/

LangChain Agent 개요

create_tool_calling_agent

- 프롬프트: "agent_scratchpad", "input" 변수 포함

```python
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

agent_prompt = ChatPromptTemplate.from_messages([
    ("system", """당신은 레스토랑 메뉴 정보를 제공하고 음식에 대한 추천을 하는 AI 어시스턴트입니다.

    주요 지침:
    1. 메뉴 정보 요청 시 반드시 search_menu 도구를 사용하세요. 이 도구로 메뉴의 가격, 재료, 조리법
    2. 메뉴와 관련된 추가 정보(예: 와인 추천, 요리 팁 등)가 필요한 경우 search_web 도구를 사용하세요
    3. 검색 결과를 바탕으로 명확하고 간결한 응답을 제공하세요.
    4. 질문이 모호하거
    5. 항상 도움이 되
    6. 메뉴 정보를 제
    7. 추천을 할 때는

    기억하세요, 메뉴
    MessagesPlaceholder(variable_name="chat_history", option
    ("human", "{input}"),
    MessagesPlaceholder(variable_name="agent_scratchpad"),
])
```

- A사용자의 초기 쿼리나 지시사항
- Agent가 작업을 시작하는 출발점

- Agent의 사고 과정과 중간 단계를 기록
- 이전 단계의 결과와 다음 단계를 계획하는 데 사용

- **create_tool_calling_agent** 함수 사용

```python
from langchain.agents import create_tool_calling_agent

tools = [web_search_tool, blog_search, search_menu]
agent = create_tool_calling_agent(llm, tools, agent_prompt)
```

AgentExecutor

```python
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

query = "시그니처 스테이크의 가격과 특징은 무엇인가요? 그리고 스테이크와 어울리는 와인 추천도 해주세요."

agent_response = agent_executor.invoke({"input": query})
```

```
> Entering new AgentExecutor chain...

Invoking: `search_menu` with `{'query': '시그니처 스테이크'}`


[Document(metadata={'menu_name': '시그니처 스테이크', 'menu_number': 1, 'source': './data/restaurant_menu.txt'}, page_content
Invoking: `web_search` with `{'query': '스테이크와 어울리는 와인 추천'}`


[{'url': 'https://secrettsteaks.com/blog/steak-wine-pairing.php', 'content': '스테이크와 가장 잘 어울리는 와인을 추천해드리며,
- **가격:** ₩35,000
- **주요 재료:** 최상급 한우 등심, 로즈메리 감자, 그릴드 아스파라거스
- **특징:** 셰프의 특제 시그니처 메뉴로, 21일간 건조 숙성한 최상급 한우 등심을 사용합니다. 미디엄 레어로 조리하여 육즙을 최대한 보존하

### 스테이크와 어울리는 와인 추천
1. **카베르네 소비뇽 (Cabernet Sauvignon)**
   - **이유:** 고소하고 진한 풍미가 스테이크의 육즙과 잘 어울립니다.

2. **시라 또는 쉬라즈 (Syrah/Shiraz)**
   - **이유:** 스파이시한 향과 함께 베리류의 풍부한 맛이 스테이크와 조화를 이룹니다.
```

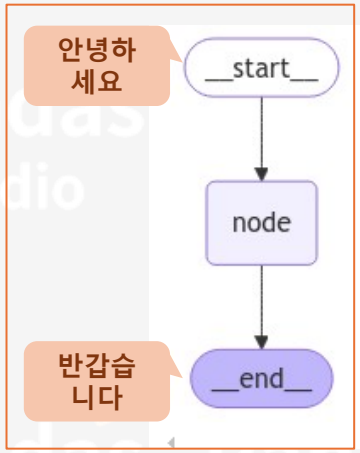
StateGraph 개요

- 정의: 상태(state)를 기반으로 작동하는 그래프 구조
- 목적: 복잡한 작업 흐름을 상태와 전이로 모델링하여 **유연하고 제어 가능한 시스템**을 구축
- 특징: 각 노드(node)가 특정 상태를 나타내며, 엣지(edge)가 상태 간 전이 조건을 정의

1. **State (상태):**
   - 정의: 애플리케이션의 현재 스냅샷을 나타내는 공유 데이터 구조
   - 특징:
     - 주로 TypedDict나 Pydantic BaseModel을 사용
     - 시스템의 전체 컨텍스트를 포함

2. **Node (노드):**
   - 정의: Agent의 로직을 인코딩하는 Python 함수
   - 특징:
     - 현재 State를 입력으로 받고,
     - 계산이나 부작용(side-effect)을 수행한 다음,
     - 업데이트된 State를 반환.

3. **Edge (엣지):**
   - 정의: 현재 State를 기반으로 다음에 실행할 Node를 결정하는 함수
   - 특징:
     - 조건부 분기 or 고정 전이
     - 시스템의 흐름을 제어

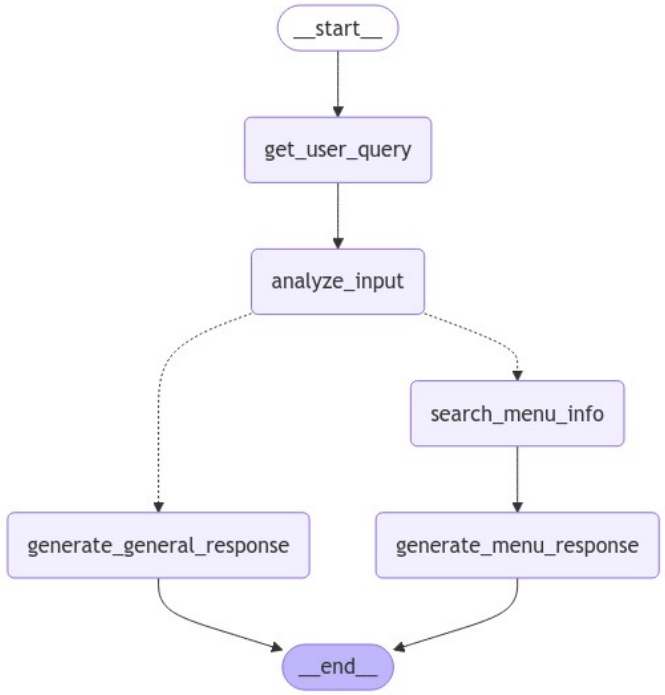



```python
# 그래프의 전체 상태 (이것은 노드 간 공유되는 공용 상태)
class OverallState(BaseModel):
    text: str

# 노드 함수
def node(state: OverallState):
    return {"text": "반갑습니다"}  # 상태를 변경해서 출력

# 그래프 구축 (엣지로 연결)
builder = StateGraph(OverallState)
builder.add_node(node)                    # 첫 번째 노드
builder.add_edge(START, "node")           # 그래프는 node로 시작
builder.add_edge("node", END)             # node 실행 후 그래프를 종료
graph = builder.compile()

# 유효한 입력으로 그래프를 테스트
graph.invoke({"text": "안녕하세요"})
```

조건부 엣지(Edge)를 활용 분기(Branching)

StateGraph

- 정의: 현재 상태나 입력에 따라 다음 노드를 동적으로 결정하는 연결
- 목적: 시스템의 흐름을 더 유연하고 상황에 맞게 제어
- 특징: 조건문을 사용하여 다음 실행할 노드를 선택 (분기, Branching)

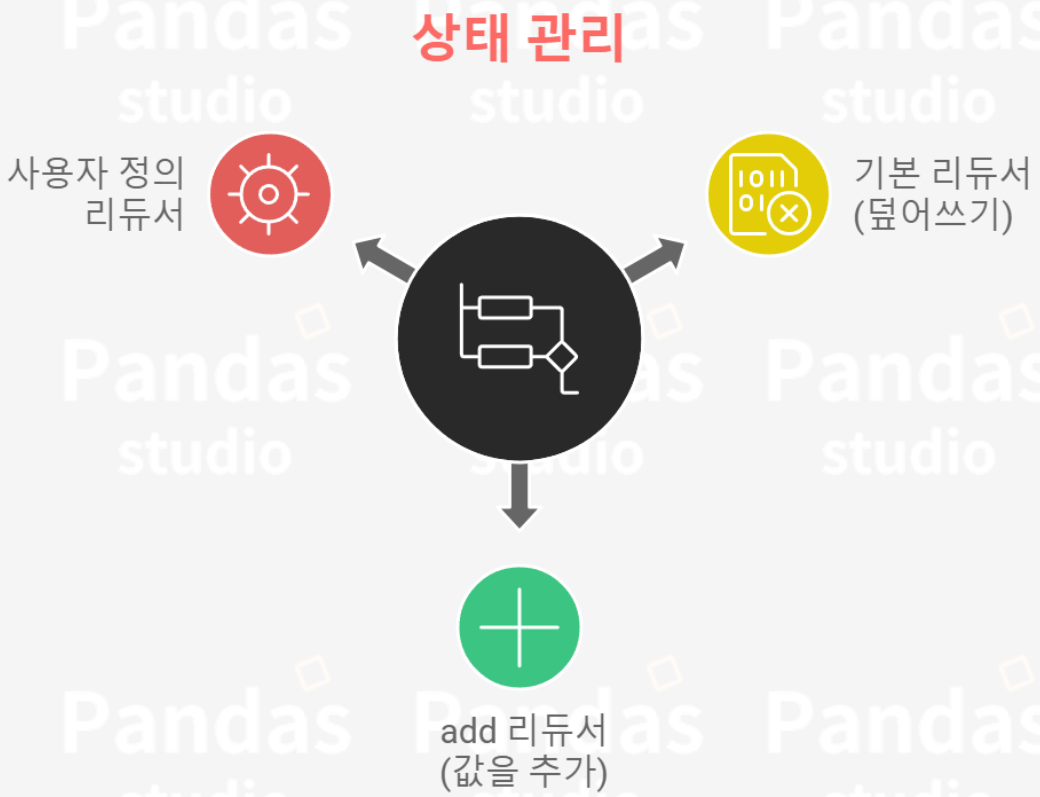



```python
def decide_next_step(state: MenuState):
    if state['is_menu_related']:
        return "search_menu_info"
    else:
        return "generate_general_response"
```

```python
# 조건부 엣지 추가
builder.add_conditional_edges(
    "analyze_input",
    decide_next_step,
    {
        "search_menu_info": "search_menu_info",
        "generate_general_response": "generate_general_response"
    }
)
```

State Reducer

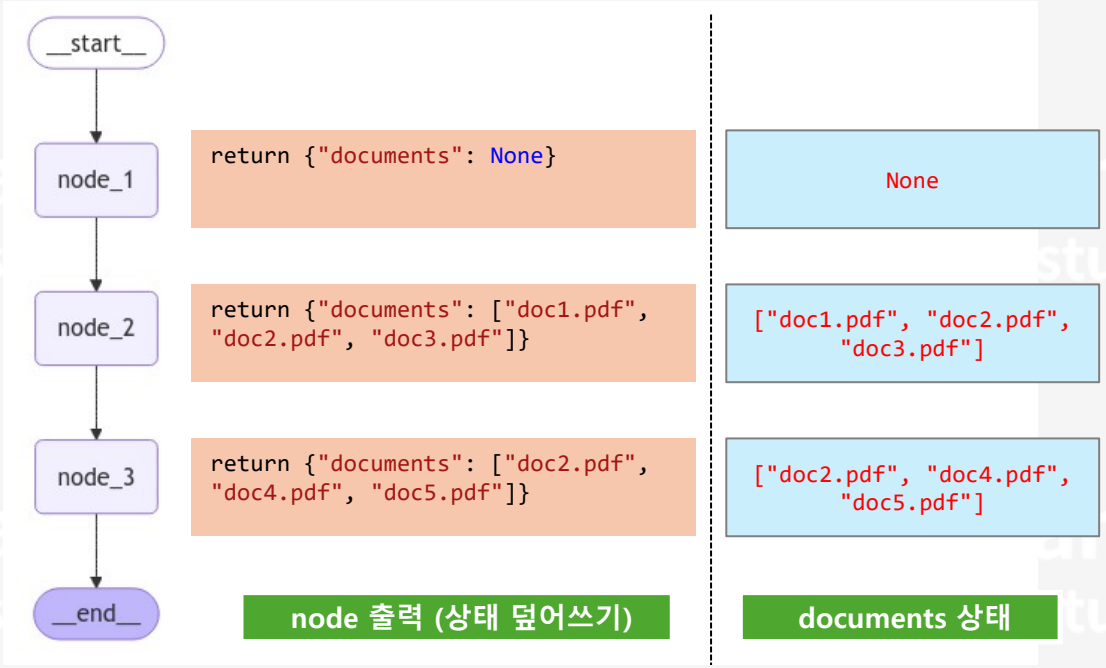
- 정의: **상태 업데이트**를 관리하는 함수
- 목적: 그래프의 각 노드의 출력을 그래프의 전체 상태에 통합하는 방법을 정의
- 동작 방식:
  - 각 노드의 반환값은 해당 상태 키의 이전 값을 **덮어쓰기** 저장 (**기본** 리듀서)
  - 메시지 리스트 등에서 이전 상태에 새로운 값을 추가할 때 사용 (**add** 리듀서)
  - 중복 제거, 정렬 등 특수한 상태 관리를 하는 사용자 정의 리듀서 지원 (**custom** 리듀서)



State Reducer

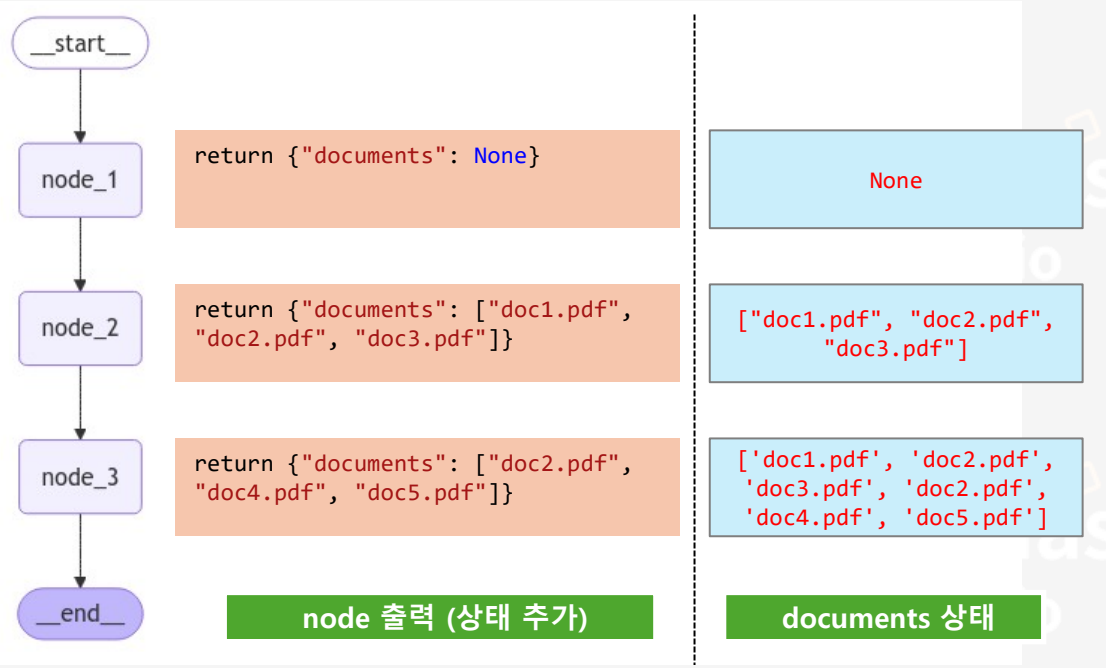
Reducer를 별도로 지정하지 않은 경우

```
class DocumentState(TypedDict):
    query: str
    documents: List[str]
```



Reducer를 별도로 지정하는 경우

```
class ReducerState(TypedDict):
    query: str
    documents: Annotated[List[str], add]
```



MessageGraph 개요

- 정의: LangChain의 ChatModel을 위한 특수한 형태의 StateGraph
- 특징: Message 객체 목록(HumanMessage, AIMessage 등)을 입력으로 처리
- 목적: 대화 기록을 효과적으로 관리하고 활용할 수 있음 (자연스러운 대화 흐름, 컨텍스트 활용 답변)

Messages State 정의

1. 대화 기록을 그래프 상태에 메시지 목록으로 저장
2. Message 객체 목록을 저장하는 **messages** 키를 추가
3. 이 키에 리듀서 함수를 추가하여 메시지 업데이트를 관리
   - operator.add: 새 메시지를 기존 목록에 단순히 추가
   - add_messages 함수:기존 메시지 업데이트 처리(메시지 ID를 추적).

```python
from typing import Annotated
from langchain_core.messages import AnyMessage
from langgraph.graph.message import add_messages

# 기본 State 초기화 방법을 사용
class GraphState(TypedDict):
    messages: Annotated[list[AnyMessage], add_messages]
```

```python
# LangGraph MessagesState라는 미리 만들어진 상태를 사용

from langgraph.graph import MessagesState
from typing import List
from langchain_core.documents import Document

class GraphState(MessagesState):

    # messages 키는 기본 제공
    # 다른 키를 추가하고 싶을 경우 아래 주석과 같이 적용 가능
    documents: List[Document]
    grade: float
    num_generation: int
```

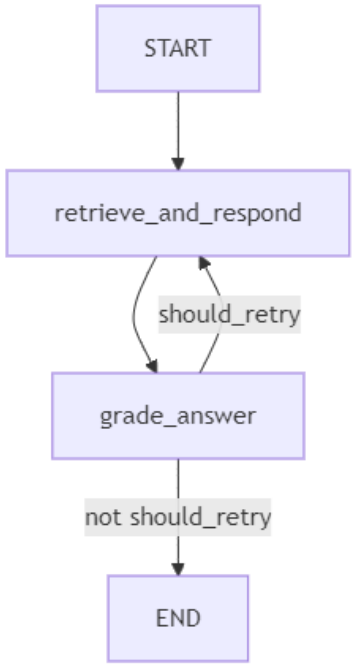
주의사항:

- 메모리 관리: 대화 기록이 길어질 경우 오래된 메시지를 제거하는 로직을 추가
- 프라이버시: 민감한 정보가 포함된 메시지는 적절히 처리
- 성능 최적화: 메시지 처리 로직이 복잡해질 경우 성능 최적화 필요

조건부 엣지(Edge)를 활용 반복(Iteration/ Feedback Loop)

MessageGraph

- 조건부 엣지를 활용하여 반복적인 작업을 수행하는 그래프
- 평가 결과를 바탕으로 다시 응답을 생성하는 과정이 반복
- grade_answer에서 retrieve_and_respond로 돌아가는 구조는 피드백 루프를 형성




```python
def should_retry(state: GraphState)
    if state["grade"] < 0.7:
        return "retrieve_and_respond"
    else:
        return "generate"
```

```python
# 조건부 엣지 추가
builder.add_conditional_edges(
    "grade_answer",
    should_retry,
    {
        "retrieve_and_respond": "retrieve_and_respond",
        "generate": END
    }
)
```

ReAct(Reasoning + Acting) 개념

- 정의: ReAct는 '추론(Reasoning)'과 '행동(Acting)'을 결합한 접근 방식
- 목적: LLM이 단순히 텍스트를 생성하는 것을 넘어, 환경과 상호작용하며 복잡한 작업을 수행

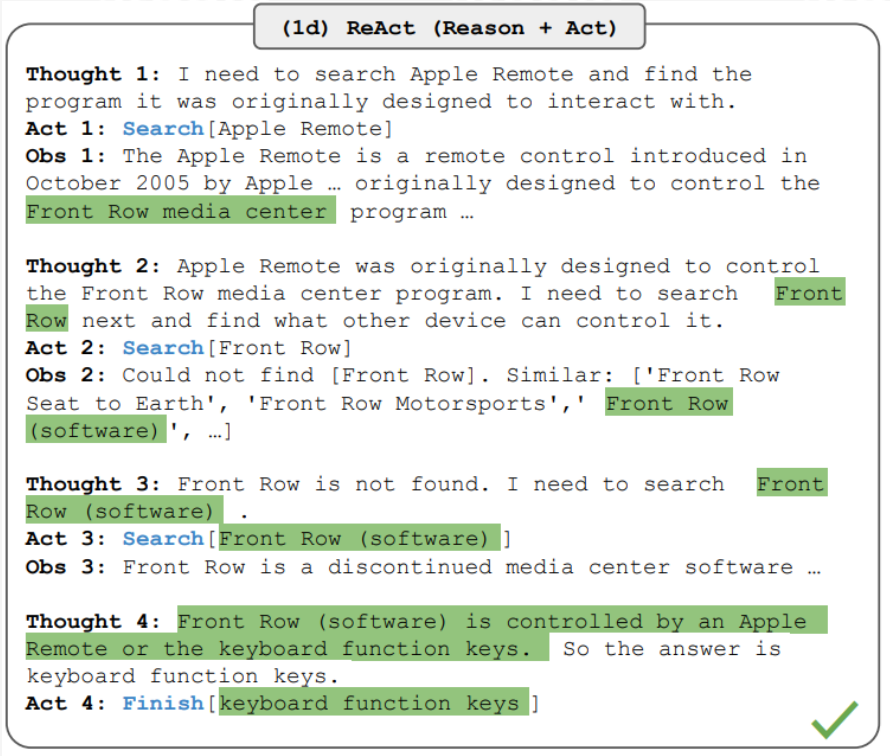
ReAct(Reasoning + Acting) 동작 방식

1. 행동 (Act):
- 모델이 주어진 상황을 분석하고 적절한 도구를 선택
- 선택한 도구를 호출하고 필요한 입력을 제공

2.관찰 (Observe):
- 호출된 도구의 실행 결과나 출력을 모델에 다시 전달
- 에이전트가 자신의 행동 결과를 이해/학습

3.추론 (Reason):
- 이전 단계의 관찰 결과를 분석하여 다음 행동을 결정
- 다른 도구를 호출하거나, 또는 직접 응답을 생성
- 현재 상황을 평가하고 최선의 다음 단계를 선택

→ 추론과 행동의 반복적인 사이클을 통해 복잡한 작업을 단계적으로 해결



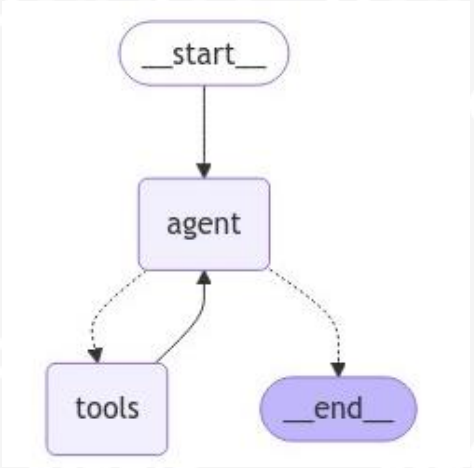
출처: ReAct: Synergizing Reasoning and Acting in Language Models (https://arxiv.org/abs/2210.03629)

랭그래프(LangGraph)에 내장된 ReAct 에이전트 사용

create_react_agent 함수

- 가장 쉽게 ReAct 에이전트를 생성하는 방법

- 주요 단계:
  1. 필요한 라이브러리 임포트
  2. 언어 모델 (LLM) 설정
  3. 도구 정의
  4. ReAct 에이전트 생성
  5. 에이전트 실행



코드 예시

```python
from langgraph.prebuilt import create_react_agent

# 그래프 생성
graph = create_react_agent(
    llm,
    tools=tools,
    state_modifier=system_prompt)

# 그래프 실행
inputs = {"messages": [HumanMessage(content="스테이크 메뉴의 가격은 얼마인가요?")]}
messages = graph.invoke(inputs)
for m in messages['messages']:
    m.pretty_print()
```

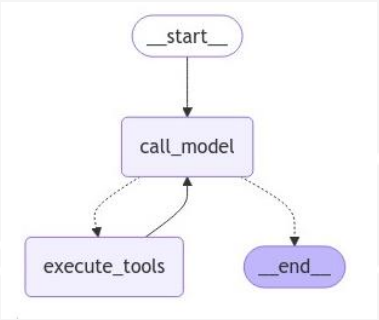
```
================================ Human Message =================================
스테이크 메뉴의 가격은 얼마인가요?
================================== Ai Message ==================================
Tool Calls:
  search_menu (call_jUlg9K9UGANFb3NE4bGwNYRe)
Call ID: call_jUlg9K9UGANFb3NE4bGwNYRe
  Args:
    query: 스테이크
================================= Tool Message =================================
Name: search_menu
<Document source="./data/restaurant_menu.txt"/> 1. 시그니처 스테이크 • 가격: ₩35,000 • 주요 식재료: 최상급 한우 등심, 로즈메리 감자, 그릴드 아스파라거스 • 설명: 셰프의 특제 시그니처 메뉴로, 21일간 건조 숙성한 최상급 한우 등심을 사용합니다. 미디엄 레어로 조리하여 육즙을 최대한 보존하며, 로즈메리 향의 감자와 아삭한 그릴드 아스파라거스가 곁들여집니다. 레드와인 소스와 함께 제공되어 풍부한 맛을 더합니다. </Document>
================================== Ai Message ==================================
스테이크 메뉴의 가격은 다음과 같습니다: 시그니처 스테이크 - 가격: ₩35,000 …
```

StateGraph 구조를 사용하여 ReAct 에이전트 만들기

조건부 엣지 함수를 사용자 정의

- 조건부 엣지 함수를 정의하면 ReAct 에이전트의 동작을 더 세밀하게 제어 가능
- 도구 호출 여부에 따라 실행 지속 여부를 결정

```python
def should_continue(state: GraphState):
    last_message = state["messages"][-1]
    # 도구 호출이 있으면 도구 노드로 이동
    if last_message.tool_calls:
        return "execute_tools"
    # 도구 호출이 없으면 답변 생성하고 종료
    return END

# 그래프 구성
builder = StateGraph(GraphState)
builder.add_node("call_model", call_model)
builder.add_node("execute_tools", ToolNode(tools))

builder.add_edge(START, "call_model")
builder.add_conditional_edges(
    "call_model",
    should_continue,
)
builder.add_edge("execute_tools", "call_model")
graph = builder.compile()
```



도구 노드 ToolNode(tools)

- 목적: 모델이 요청한 도구 호출을 실제로 실행
- 작동 방식:

1. 최신 AIMessage의 tool_calls 필드에서 도구 호출 정보 추출
2. 추출된 도구 호출 요청들을 동시에(병렬로) 실행
3. 각 도구 호출의 결과에 대해 ToolMessage를 생성 (결과 포함)
4. ToolMessage는 다시 AI 모델에게 전달 -> 답변 생성

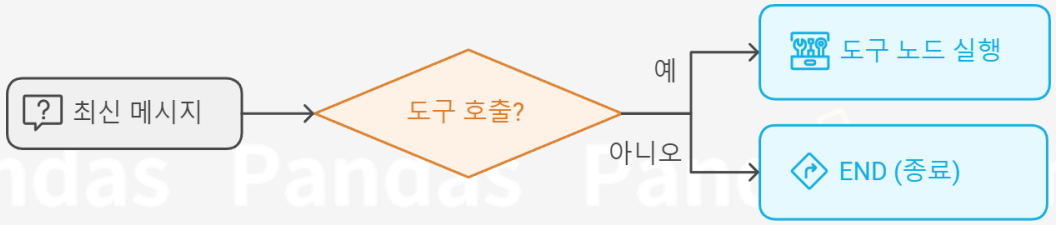
```python
from langgraph.prebuilt import ToolNode


# 도구 목록
tools = [search_menu, search_web]

# 도구 노드 정의
tool_node = ToolNode(tools)
```

StateGraph 구조를 사용하여 ReAct 에이전트 만들기

랭그래프 tools_condition 함수 활용

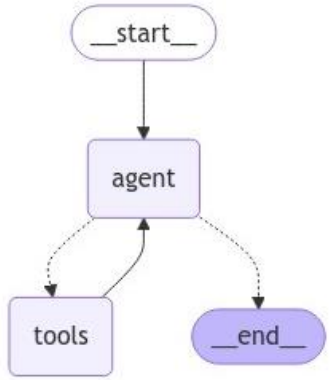
- tools_condition은 LangGraph에서 제공하는 매우 유용한 조건부 엣지 함수
- 작동 방식:

1. 최신 메시지(또는 결과)를 검사
2. 도구 호출이 포함: tools 노드에서 도구 실행
3. 도구 호출이 없음: END노드에서 종료



코드 예시

```python
from langgraph.prebuilt import tools_condition

# 노드 함수 정의
def call_model(state: GraphState):
    system_prompt = SystemMessage(content=system_prompt_template)
    messages = [system_prompt] + state['messages']
    response = llm_with_tools.invoke(messages)
    return {"messages": [response]}

# 그래프 구성
builder = StateGraph(GraphState)

builder.add_node("agent", call_model)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "agent")

# tools_condition을 사용한 조건부 엣지 추가
builder.add_conditional_edges(
    "agent",
    tools_condition,
)

builder.add_edge("tools", "agent")
graph = builder.compile()
```



MemorySaver

- 정의: 그래프의 각 단계 실행 후 자동으로 상태를 저장 (체크포인터 역할)
- 목적: 상태의 일시성(stateless) 문제 해결 (즉, 그래프는 **각 실행마다 새로운 상태로 초기화**되는 문제)
- 필요성: 대화의 연속성 (멀티 턴), 대화 중단 후 복원 가능, 독립적인 대화 스레드 관리

MemorySaver 기능

1. 체크포인터: 그래프의 각 단계 실행 후 상태를 저장
2. 인메모리 키-값 저장소: 상태 검색 기능
3. 지속성 제공: 저장된 체크포인트로부터 실행 재개


```python
from langgraph.checkpoint.memory import MemorySaver

# 메모리 초기화
memory = MemorySaver()

# 체크포인터 지정하여 그래프 컴파일
graph_memory = builder.compile(checkpointer=memory)
```

체크포인터 사용 방법

1. 메모리 사용 시 thread_id를 지정
2. 체크포인터는 그래프의 각 단계에서 상태를 기록 (모든 상태를 저장)
3. 나중에 thread_id를 사용하여 이 스레드에 접근 가능

```python
config = {"configurable": {"thread_id": "1"}}
messages = [HumanMessage(content="스테이크 메뉴의 가격은 얼마인가요?")]
messages = graph_memory.invoke({"messages": messages}, config)
for m in messages['messages']:
    m.pretty_print()
```

```python
config = {"configurable": {"thread_id": "1"}}
messages = [HumanMessage(content="둘 중에 더 저렴한 메뉴는 무엇인가요?")]
messages = graph_memory.invoke({"messages": messages}, config)
for m in messages['messages']:
    m.pretty_print()
```

Adaptive RAG 개요

- 질문의 복잡성에 따라 가장 적합한 검색 및 생성 전략을 동적으로 선택하는 방법

Adaptive RAG 작동 방식

1. 사용자 **질문의 복잡성** 수준을 분석
2. 분석 결과에 따라 가장 적합한 처리 전략을 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 응답을 생성

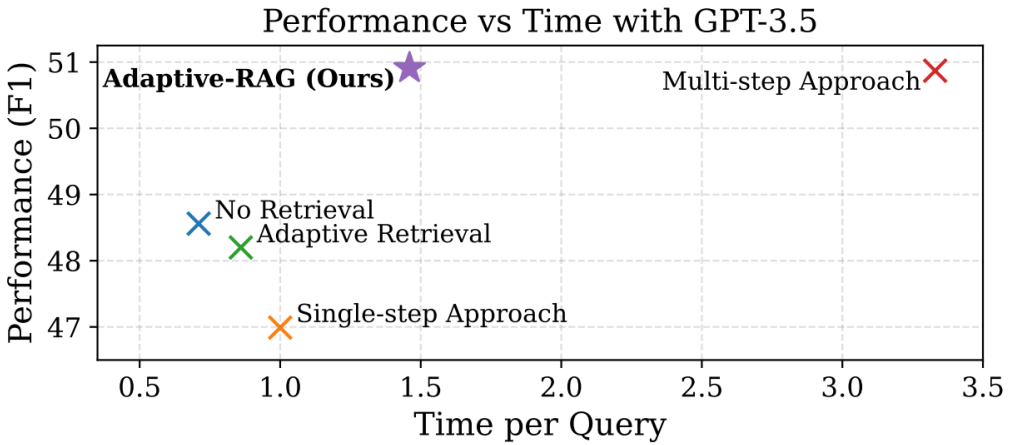
처리 전략 선택

단순 질문

기본 LLM 또는 검색

복잡한 질문

반복적 검색 증강 LLM



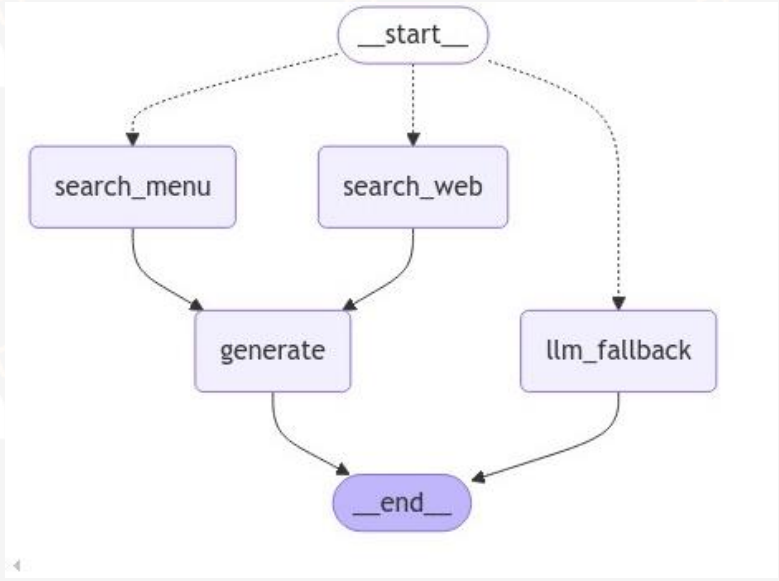
논문(이미지 출처): https://arxiv.org/abs/2403.14403

Adaptive RAG 구현 실습

- 질문 유형에 따라 가장 적합한 검색 전략을 사용 (질문 라우팅)

질문 라우팅을 통한 검색 도구 선택

1. 레스토랑 메뉴 관련 질문: search_menu 도구로 라우팅
2. 메뉴 외의 정보나 일반적인 질문: search_web 도구로 라우팅




```python
# 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_web 도구 중 하나를 선택",
    )

# 구조화된 출력을 위한 LLM 설정
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0)
structured_llm_router = llm.with_structured_output(ToolSelector)

# 라우팅을 위한 프롬프트 템플릿
system = """당신은 사용자 질문을 적절한 도구로 라우팅하는 전문가입니다.
레스토랑 메뉴에 관한 질문은 search_menu 도구를 사용하세요.
레스토랑 메뉴에 없는 정보 또는 일반적인 질문은 search_web 도구를 사용하세요."""

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

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

Human-in-the-Loop (HITL)

- AI 시스템의 자동화된 처리와 인간의 전문 지식을 결합
- 시스템의 결정이나 출력에 대해 인간이 검토하고 개입할 수 있는 지점을 제공
  → **이를 통해 AI의 효율성과 인간의 판단력을 모두 활용**

LangGraph의 Breakpoints 활용

- Breakpoints: 그래프 실행을 특정 지점에서 일시 중지하는 메커니즘
- LangGraph의 체크포인트 시스템을 기반으로 구현
- 이를 통해 인간 전문가가 중간 결과를 검토하고 필요한 경우 개입

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

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

Breakpoint 실행

- graph.stream()을 사용하여 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_test"}}
inputs = [HumanMessage(content="대표 메뉴는 무엇인가요?")]
for event in graph.stream({"messages": inputs}, thread,
stream_mode="values"):
    event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================
대표 메뉴는 무엇인가요?
================================== Ai Message ==================================
Tool Calls:
  search_menu (call_ixDipQ0n8Vl0kXOX7pnvvWJS)
Call ID: call_ixDipQ0n8Vl0kXOX7pnvvWJS
  Args:
    query: 대표 메뉴
```

Human-in-the-Loop (HITL)

Breakpoint 상태 관리

- graph.get_state(thread)를 호출하여 현재 그래프의 상태 확인

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

```
---그래프 상태---
StateSnapshot(values={'messages': [HumanMessage(content='대표 메뉴는 무엇인가요?',
additional_kwargs={}, response_metadata={}, id='d401df75-32ce-4985-a568-db19d4948fee'),
AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id':
'call_ixDipQ0n8Vl0kXOX7pnvvWJS', _... , 'checkpoint_id': '1ef81557-1d57-6e54-8000-
77392c1f1630'}}, tasks=(PregelTask(id='b7ffc42e-08dc-1ff6-ad18-dd9bd2a010a9',
name='retrieve', path=('__pregel_pull', 'retrieve'), error=None, interrupts=(),
state=None),))
================================ Human Message =================================
대표 메뉴는 무엇인가요?
================================== Ai Message ==================================
Tool Calls:
  search_menu (call_ixDipQ0n8Vl0kXOX7pnvvWJS)
Call ID: call_ixDipQ0n8Vl0kXOX7pnvvWJS
  Args:
    query: 대표 메뉴
```

다음에 실행될 노드를 확인

- 중단점 이후에 실행될 다음 노드를 확인할 수 있음

```
current_state.next
```

```
('retrieve',)
```

Breakpoint 이후 단계를 계속해서 실행

- 입력값을 None으로 지정하면 중단점부터 실행

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

```
================================== Ai Message ==================================
Tool Calls:
  search_menu (call_ixDipQ0n8Vl0kXOX7pnvvWJS)
Call ID: call_ixDipQ0n8Vl0kXOX7pnvvWJS
  Args:
    query: 대표 메뉴
---결정: 문서 관련성 있음---
================================= Tool Message =================================
Name: search_menu [Document(metadata={'menu_name': '시그니처 스테이크', 'menu_number': 1,
'source': './data/restaurant_menu.txt'}, page_content='1. 시그니처 스테이크\n • 가격:
₩35,000\n • 주요 식재료… 제공되어 풍부한 맛을 더합니다.'), Document(metadata={'menu_name': '
안심 스테이크 샐러드', 'menu_number': 8, 'source': './data/restaurant_menu.txt'},
page_content='8. 안심 스테이크 샐러드\n • 가격: ₩26,000\n • 주요 식재료: 소고기 안심, 루꼴라,
체리 토마토, 발사믹 글레이즈\n • 설명: … 고기의 풍미를 한층 끌어올렸습니다.')]
```

Human-in-the-Loop (HITL)

그래프 상태 업데이트

- graph.update_state() 메서드를 사용하여 그래프의 상태를 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"))
```

Self-RAG 개요

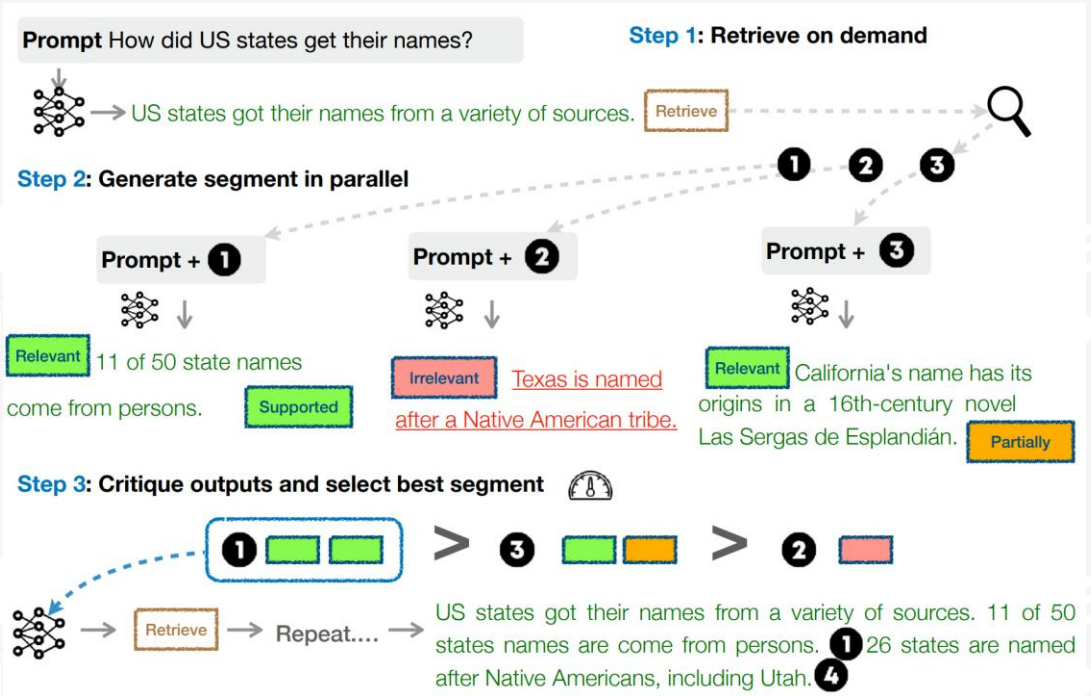
- Self-RAG는 기존의 RAG 모델에 자기 반영(self-reflection) 능력을 추가한 확장 모델
- 정보 검색, 생성, 그리고 자체 평가를 통합하여 더 정확하고 관련성 높은 응답을 생성하는 것을 목표

Self-RAG 작동 방식

1. 초기 쿼리 처리: 사용자의 질문을 받아 관련 정보를 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 단계를 반복

Algorithm 1 SELF-RAG Inference

Require: Generator LM $\mathcal{M}$, Retriever $\mathcal{R}$, Large-scale passage collections $\{d_1, \ldots, d_N\}$
Input: input prompt x and preceding generation $y_{<t}$, **Output:** next output segment y_t
$\mathcal{M}$ predicts Retrieve given $(x, y_{<t})$
if Retrieve == Yes **then**
  Retrieve relevant text passages $\mathbf{D}$ using $\mathcal{R}$ given (x, y_{t-1}) ▷ Retrieve
  $\mathcal{M}$ predicts IsRel given x, d and y_t given $x, d, y_{<t}$ for each $d \in \mathbf{D}$ ▷ Generate
  $\mathcal{M}$ predicts IsSup and IsUse given x, y_t, d for each $d \in \mathbf{D}$ ▷ Critique
  Rank y_t based on IsRel, IsSup, IsUse ▷ Detailed in Section 3.3
else if Retrieve == No **then**
  $\mathcal{M}_{gen}$ predicts y_t given x ▷ Generate
  $\mathcal{M}_{gen}$ predicts IsUse given x, y_t ▷ Critique



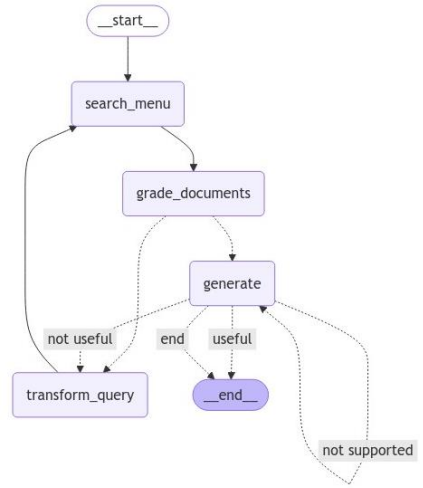
논문: https://arxiv.org/abs/2310.11511

Self-RAG 구현 실습

- 검색, 생성, 평가, 개선의 과정을 반복적으로 수행하는 지능적인 시스템을 구현

Self-RAG 구현 아이디어

1. **반복적 개선**: 문서 평가와 답변 생성 과정에 조건부 분기를 통해 지속적으로 개선
2. **자기 평가**: grade_documents와 grade_generation 노드를 통해 검색 결과와 생성된 답변의 품질을 평가
3. **동적 쿼리 개선**: transform_query 노드를 통해 검색 쿼리를 개선



- **동적 쿼리 개선 (검색 쿼리를 재작성)**

```
def transform_query(state: GraphState) -> GraphState:
    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_generations": num_generations}
```

- **관련성 평가에 합격한 문서들만 저장**

```
def grade_documents(state: GraphState) -> GraphState:
    question = state["question"]
    documents = state["documents"]
    filtered_docs = []
    for d in documents:
        score = retrieval_grader.invoke({"question": question, "document": d})
        grade = score.binary_score
        if grade == "yes":
            filtered_docs.append(d)
    return {"documents": filtered_docs}
```

- **생성된 답변의 품질을 평가**

```
def grade_generation(state: GraphState) -> str:
    question, documents, generation = state["question"], state["documents"], state["generation"]
    hallucination_grade = hallucination_grader.invoke(
        {"documents": documents, "generation": generation})
    if hallucination_grade.binary_score == "yes":
        # 1단계 환각 여부 통과할 경우 -> 2단계: 질문-답변 관련성 평가
        relevance_grade = answer_grader.invoke({"question": question, "generation": generation})
        if relevance_grade.binary_score == "yes":
            return "useful"
        else:
            return "not useful"
    else:
        return "not supported"
```

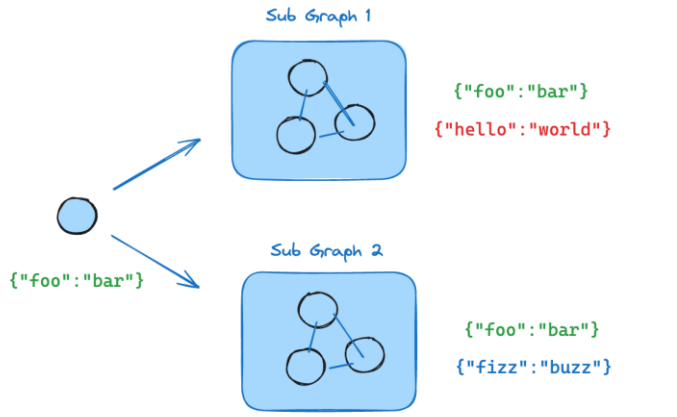
서브그래프 (Subgraphs)

Sub-graph

서브그래프 개요

- 주요 특징:
  - 각 서브그래프는 독립적인 상태 관리 가능
  - 메인 그래프와 서브그래프 간의 정보 교환 지원
  - 모듈화된 설계로 복잡한 워크플로우 구현 용이

- 구현 방법:
  - 서브그래프 상태 정의
  - 서브그래프 노드 및 엣지 구성
  - 메인 그래프에 서브그래프 통합

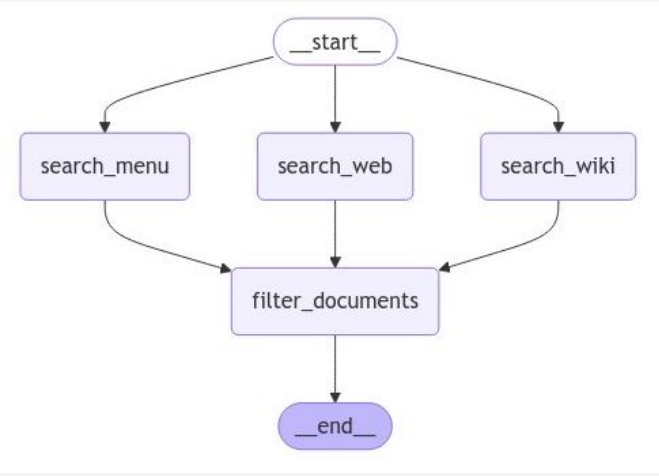


https://langchain-ai.github.io/langgraph/how-tos/subgraph/

병렬 노드 실행

- 병렬 노드 실행은 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("search_web", "filter_documents")
builder.add_edge("search_wiki", "filter_documents")
builder.add_edge("filter_documents", END)
```



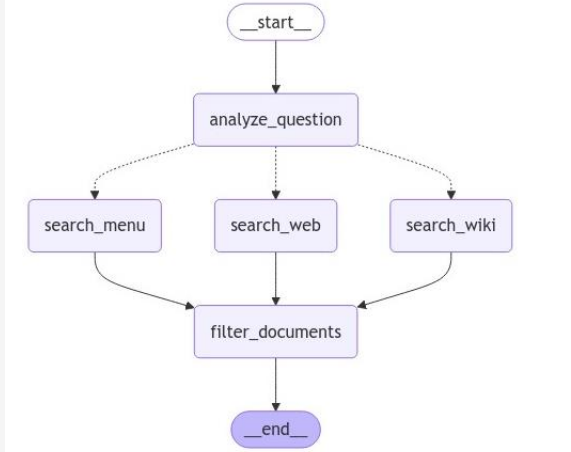
서브그래프 (Subgraphs)

Sub-graph

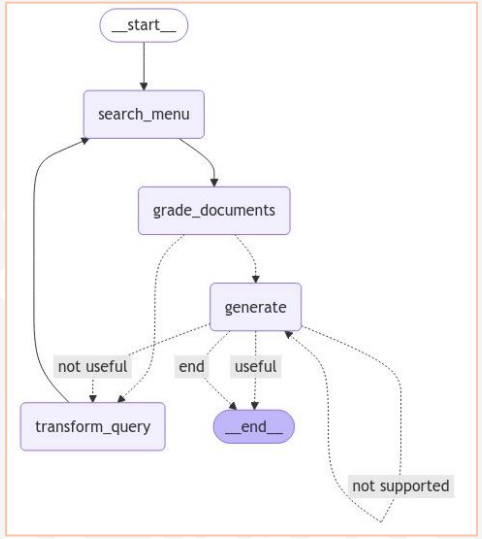
조건부 엣지로 병렬 노드 실행

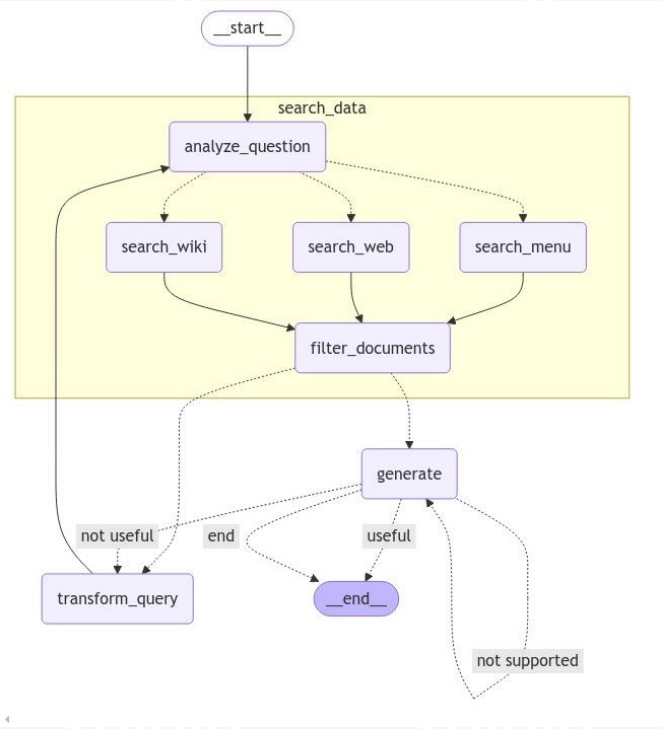
- 조건부 엣지는 실행 시간에 동적으로 경로를 결정
- 특정 조건에 따라 다른 노드 세트를 병렬로 실행 가능

```
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_documents")
search_builder.add_edge("search_web", "filter_documents")
search_builder.add_edge("search_wiki", "filter_documents")
search_builder.add_edge("filter_documents", END)
```



기존 Self-RAG 그래프와 결합



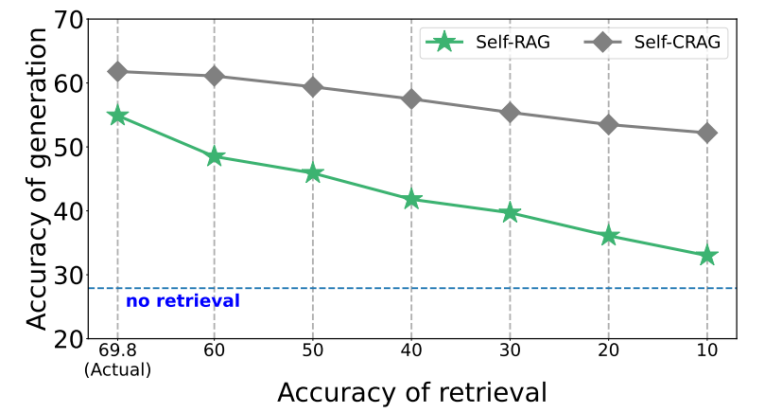


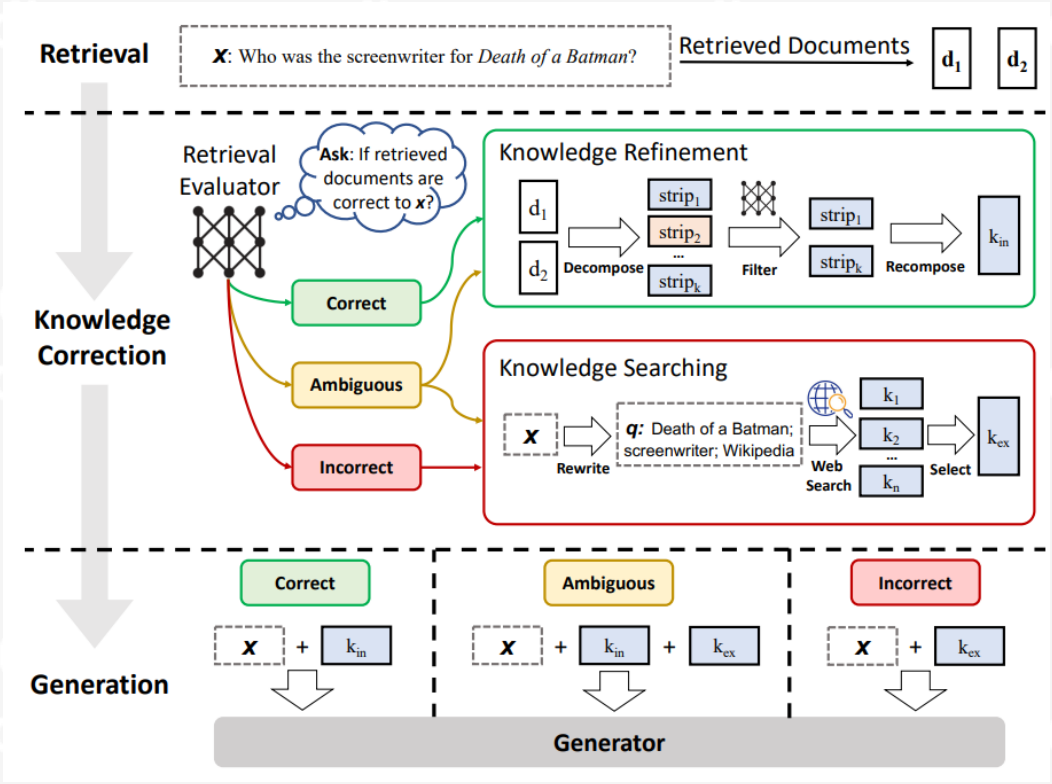
Corrective RAG (CRAG) 개요

- CRAG는 기존 RAG 시스템을 개선하여 검색된 정보의 품질과 관련성을 향상시키는 접근 방식
- 문서 관련성 평가, 지식 정제, 필요 시 외부 지식 탐색, 그리고 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 활용하여 답변 생성





논문: https://arxiv.org/pdf/2401.15884

Corrective RAG (CRAG) 구현 실습

- 주요 과정: 검색 -> 평가 -> 지식 정제 또는 웹 검색 -> 답변 생성

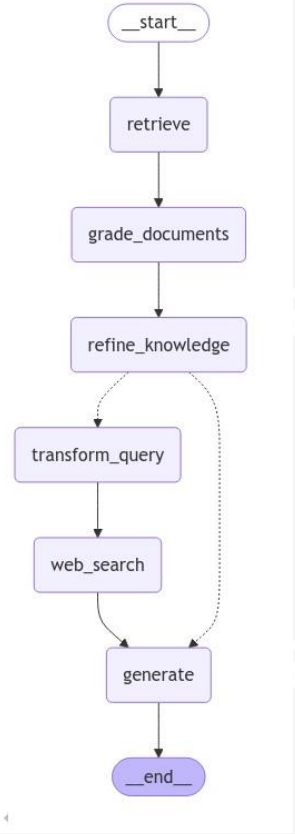
CRAG 구현 과정

1. 문서 관련성 평가 (`grade_documents`):
   - 각 문서의 관련성을 평가
   - 기준을 통과하는 문서만을 유지

2. 지식 정제 (`refine_knowledge`):
   - 문서를 "지식 조각"으로 분할하고 각각 관련성 평가
   - 관련성 높은(0.5 초과) 지식 조각만 유지

3. 웹 검색 (`web_search`):
   - 문서에 충분한 정보가 없는 경우 외부지식 활용
   - 웹 검색 결과를 기존 문서에 추가

4. 답변 생성 (`generate_answer`):
   - 정제된 지식 조각을 사용하여 답변을 생성
   - 관련 정보가 없을 경우 적절한 메시지를 반환




```python
# 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", END)
```

감사합니다!!