LangGragh 의 탄생 배경

RAG 라는 기술을 사용하다보면 한 번쯤 다음의 문제를 고민하게 됩니다.

1. LLM이 생성한 답변이 Hallucination이 아닐까?
2. RAG를 적용하여 받은 답변이 문서에 없는 "사전지식" 으로 답변한 것은 아닐까?
3. 문서 검색에서 원하는 내용이 없을 경우, 인터넷 혹은 논문에서 부족한 정보를 검색하여 지식을 보강할 수 있지 않을까?

문제

- 원하는 정보가 제대로 나올때까지 무한 반복 검색
    - 토큰량 증가, 사용금액 증가
- Hallucination을 방지하는 LLM을 추가해야하나?

결국,

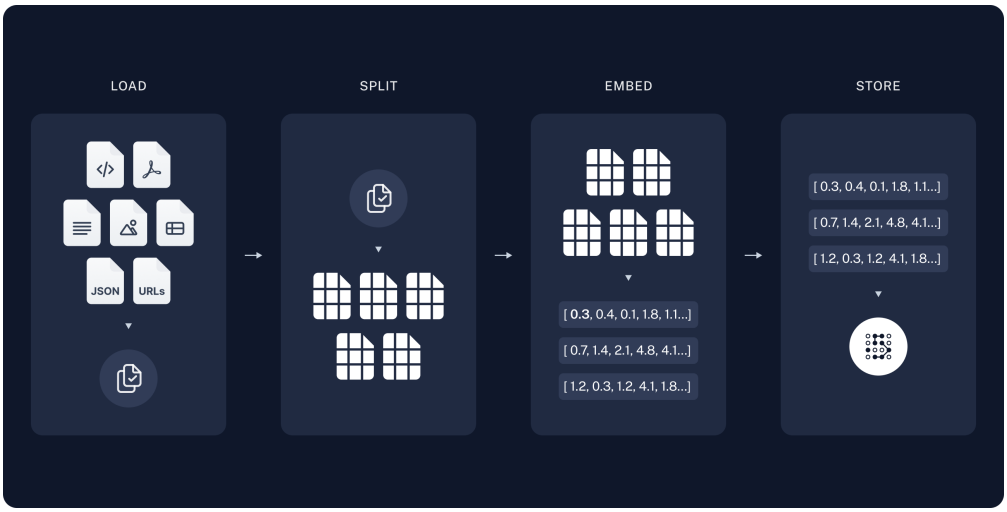
- 코드가 점점 길어지고 복잡해짐
- LLM의 일관되지 않은 답변

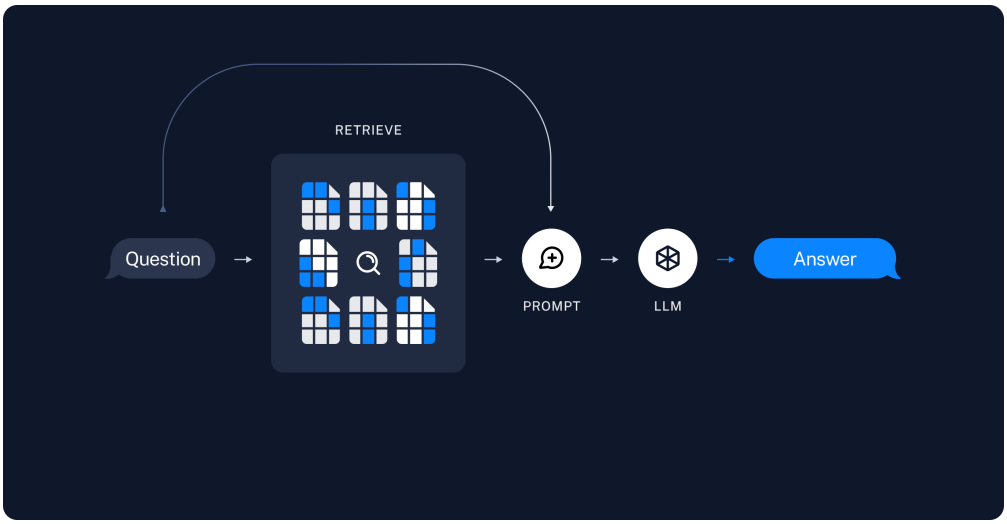
Conventional RAG 문제점

- **사전에 정의된** 데이터 소싱(PDF, DB, Table 등) 자원
- **사전에 정의된** Fixed Size Chunk
- **사전에 정의된** Query 입력
- **사전에 정의된** 검색 방법
- **신뢰하기 어려운** LLM or Agent
- **고정된** 프롬프트 형식
- LLM의 답변 결과에 대한 **문서와의 관련성/신뢰성**

RAG의 8단계

세부과정 `Data Load -> Text Split -> Embed -> Store -> Question -> Prompt -> LLM -> Answer





Conventional RAG or Traditional RAG 는 단방향 파이프라인으로 이런 것들이 한 번에 잘 되어야 좋은 답변이 나올까 말까 합니다.

LangGragh 의 제안

- 각 세부과정을 **Node** 라고 정의합니다.
- 이전 노드에서 다음 노드를 연결할때 **Edge** 연결을 합니다.
- **조건부 Edge** 를 통해 분기 처리가 가능합니다.
    - 조건부 Edge는 여러 갈래로 분기 처리 가능합니다.

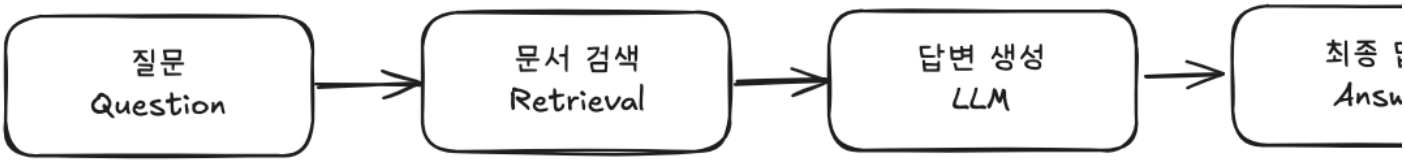
RAG 파이프라인을 유연하게 설계할 수 있습니다.

FLow Engineering

Conventional RAG 에서의 파이프라인은 단방향이기 때문에 흐름을 제어할 수는 없었습니다. 하지만 LangGragh 에서는 흐름을 제어할 수 있습니다. 흐름을 제어하여 설계하는 작업을 **Flow Engineering** 이라고 합니다.

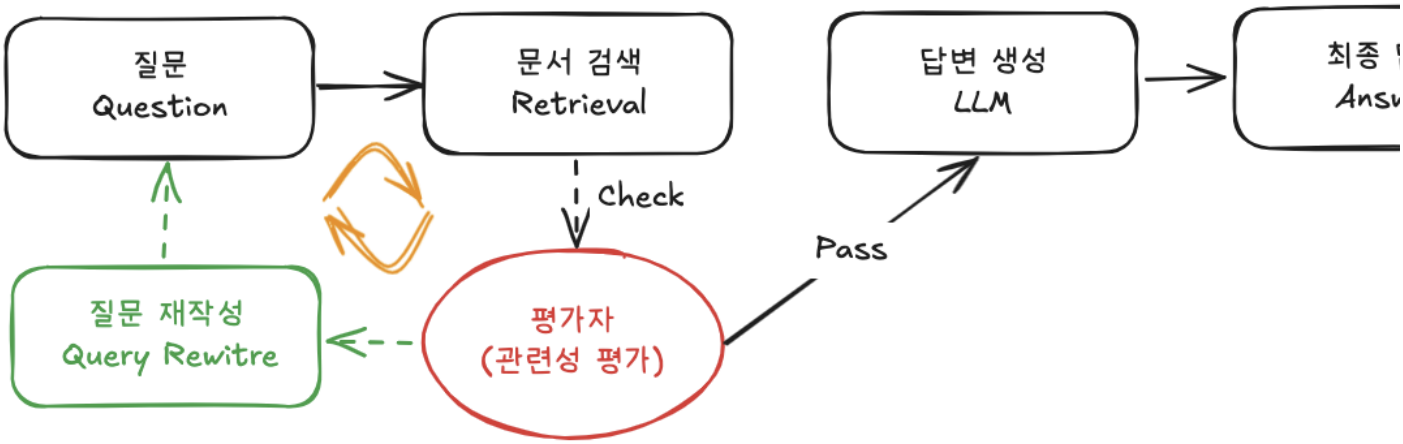
LangGragh Example

단방향 파이프라인

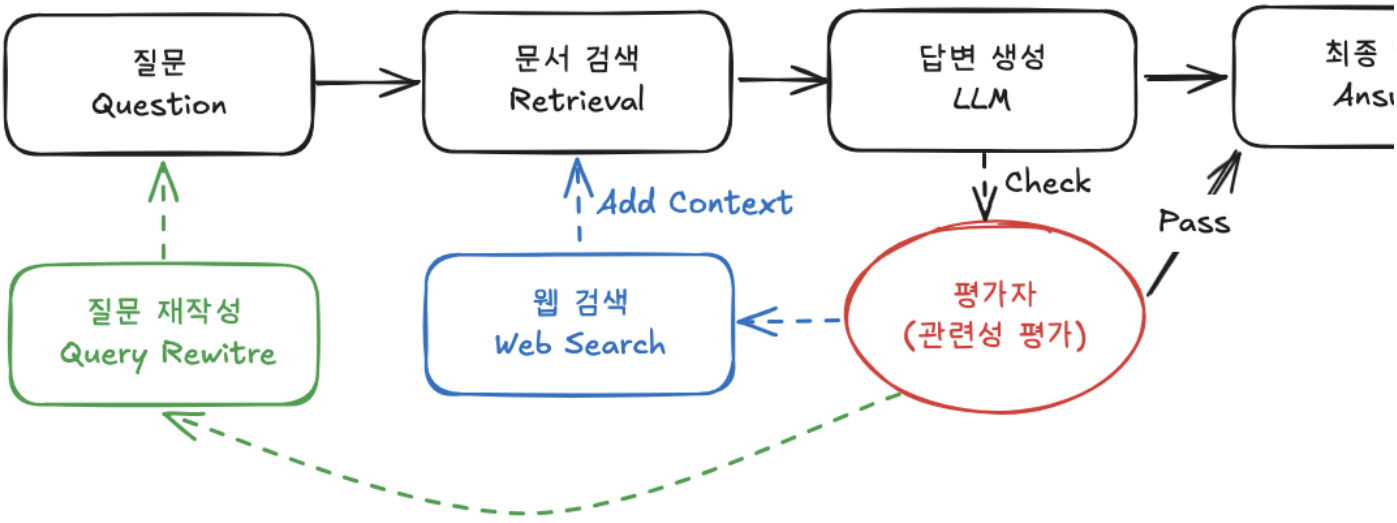


Flow Engineering

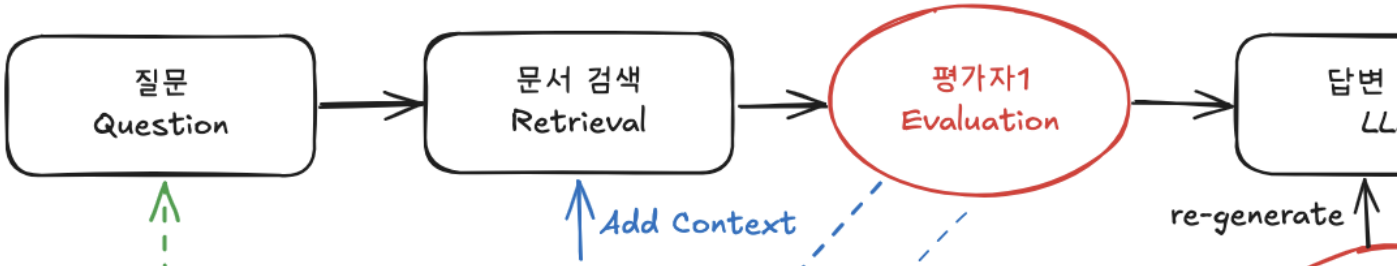
평가자 & Query Transform 추가



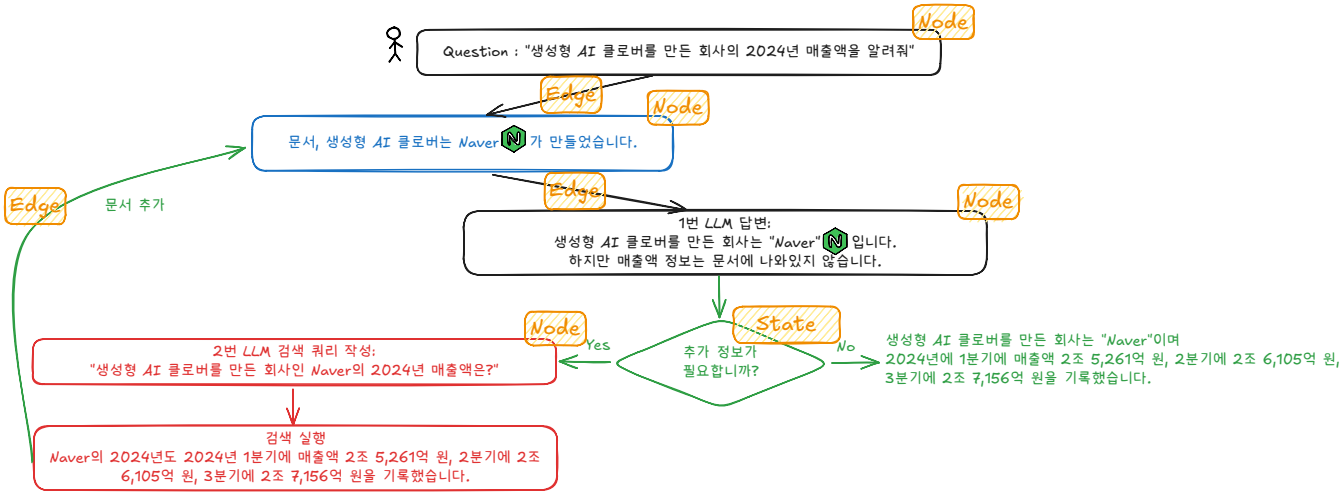
추가 검색기를 통해 문백(Context) 보강



문서-답변 간 관련성 여부를 판단하는 평가자2를 추가하여 검증



LangGragh 로 구현한 예시



용어

Node

- 사용자가 정의한 함수로 구현되고 **데이터를 입력받아 처리한 결과를 출력**합니다.

Edge

- 한 노드의 출력이 다른 노드의 입력으로 **전달**되도록 설정합니다.

State

- 작업의 진행 상황을 추적하고, **노드 간 데이터를 공유**하며 조건에 따라 작업의 흐름을 제어합니다.

Conditional Edge

- 작업 흐름에서 조건부 로직을 구현하여 데이터나 실행 경로를 분기합니다.

Human-in-the-loop

- 자동화된 프로세스 중간에 사람의 판단이나 결정을 필요로 할때 사용합니다.

Checkpointer

- 과거의 실행과정에 일어났던 내용들을 저장하는 역합니다. 과거의 대화 내용을 기억하고, 멀티턴 구현이 굉장히 쉽게 구현가능하고, 수정 & 리플레이 기능을 제공합니다.

LangGragh 세부기능

State

노드와 노드 간의 정보를 전달할 때 상태(State) 객체에 담아 전달합니다.

- TypeDict 를 상속받아 구현합니다. TypeDict 는 파이썬문법이며, dict 타입에 힌트를 추가한 개념입니다.
- 모든 값을 채우지 않아도 됩니다.
- 새로운 노드에서 값을 덮어쓰기(Overwrite) 방식으로 채웁니다.
- Reducer(add_messages 혹은 operator.add): 자동으로 list 에 메시지를 **추가**해주는 LangGragh 의 기능입니다.
  - `left(messages)` : 기본 메시지 리스트
  - `right(messages)` : 병합할 메시지 리스트 또는 단일 메세지

코드 예시)

```
from typing import Annotation, TypedDict
from langgragh.graph.message import add_messages

# GraghState 상태를 저장하는 용도
class GraghState(TypedDict):
        question: Annotated[list, add_messages] # 질문(Query rewrite 누적)
        context: Annotated[str, "Context"] # 문서의 검색 결과
        answer: Annotated[str, "Answer"] # 답변
        messages: Annotated[str, add_messages] # 메시지
        relevance: Annotated[str, "Relevance"] # 관련성
```

노드 별 상태 값의 변화

- 각 노드에서 새롭게 업데이트 하는 값은 **기존 Key 값을 덮어쓰는 방식**입니다.
- 노드에서 **필요한 상태 값을 조회하여 동작에 활용**할 수 있습니다.
- 노드4에서 노드1에서 반영된 llm 값이 그대로 **상태전달**되어 조회가 가능합니다.

Key	Question (Node1)	Retriever (Node2)	Answer (Node3)	Evaluate (Node4)
`context`	없음	"비트코인은 블록체인 기술을 기반으로	"비트코인은 블록체인 기술을 기반으로	"비트코인은 블록체인 기술을 기반으로 한 디

Key	Question (Node1)	Retriever (Node2)	Answer (Node3)	Evaluate (Node4)
		한 디지털 통화입니다."	한 디지털 통화입니다."	지털 통화입니다."
`question`	"비트코인의 작동 원리를 설명해주세요."	"비트코인의 작동 원리를 설명해주세요."	"비트코인의 작동 원리를 설명해주세요."	"비트코인의 작동 원리를 설명해주세요."
`answer`	없음	없음	"비트코인은 거래 내역을 블록체인에 기록하며, 채굴자들이 이를 검증합니다."	"비트코인은 거래 내역을 블록체인에 기록하며, 채굴자들이 이를 검증합니다."
`score`	없음	없음	없음	BAD

- socre 가 "BAD" 인 경우, 선택할 수 있는 3가지가 있습니다.
  - 노드1: 질문을 재작성요청할 수 있습니다.
  - 노드2: 문서를 다시 검색 / 검색을 통한 정보 보완을 할 수 있습니다.
  - 노드3: 답변을 재작성 요청할 수 있습니다.

Node

좋은 노드를 만드는 것은 굉장히 중요하나 일입니다. 상태를 입력받고 상태를 출력합니다.

Edge

노드와 노드간의 연결 하는 역할을 합니다. 노드명을 가지고 add_edge를 통해서 from, to 로 세팅하여 사용합니다.

Conditional Edge

노드에 조건부 엣지를 추가하여 분기를 수행할 수 있습니다. add_conditional_edges(노드이름, 조건부 판단 함수, dict로 다음 단계 결정) 예를 들어, 답변에 대한 평가를 수행하고 평가의 점수가 기준에 못미치면 passmap 을 통해서 다음단계를 결정할 수 있게 합니다.

체크포인터(memory)

각 노드간 실행결과를 추적하기 위한 메모리입니다. 체크포인터를 활용하여 특정 시점으로 되돌리는 기능입니다. 멀티턴 대화에서도 유용하게 사용할 수 있습니다.

그래픽 시각화

get_graph(xray=Ture).draw_mermaid_png() 를 사용하여 시각화를 할 수 있습니다.