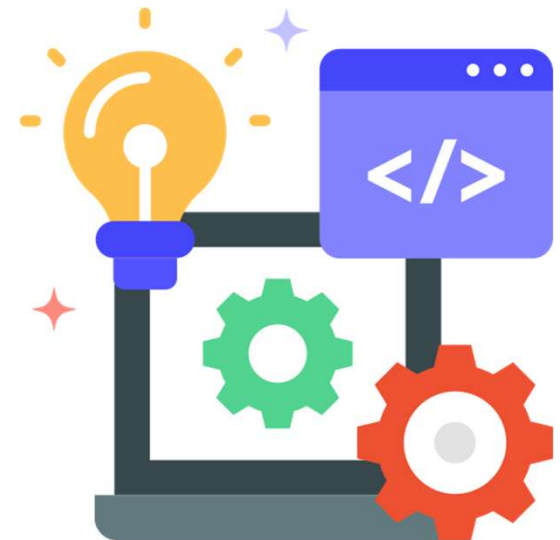


LLM이해와 활용 (RAG)

목차

- Large Language Model 이해
- OpenAI API 주요 기능 활용
- RAG(Retrieval-Augmented Generation)



검색 증강 생성 RAG



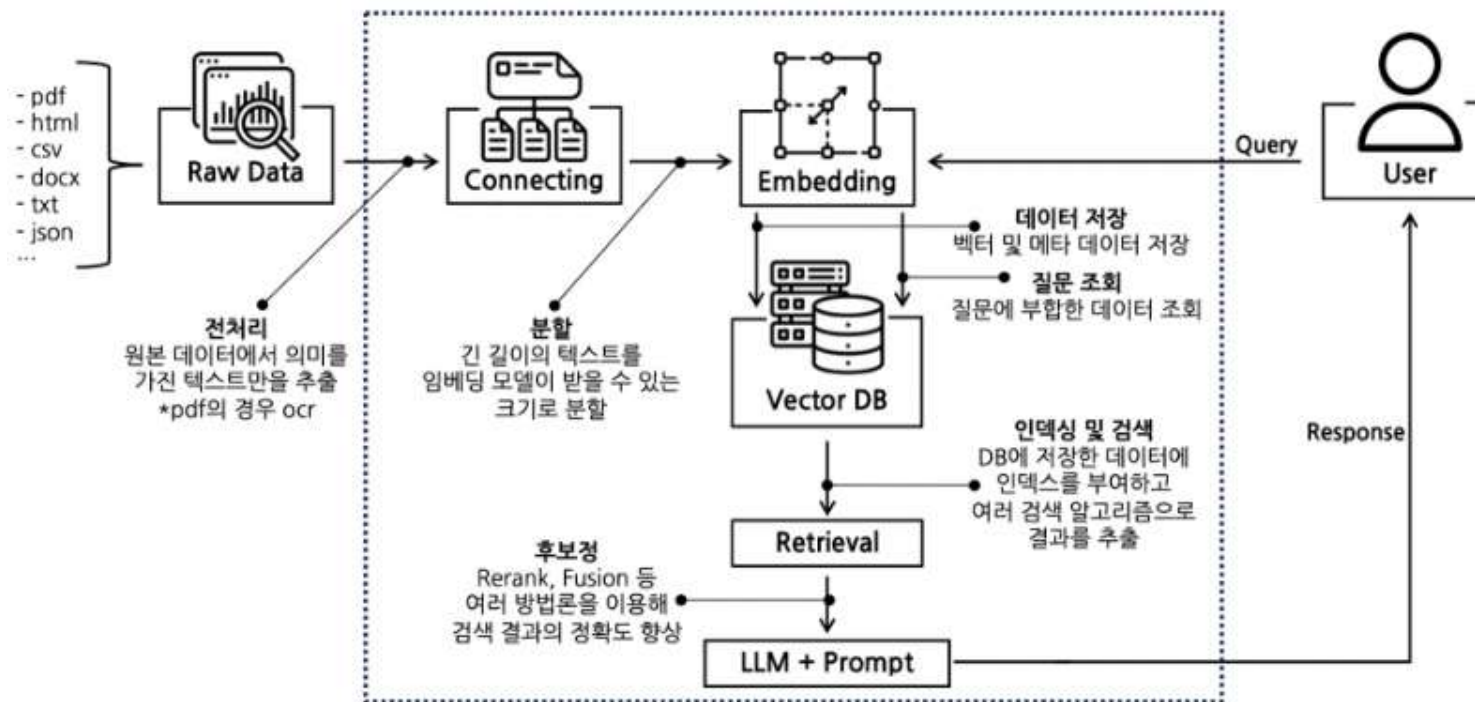
소목차

- RAG
- Llama Index
- LangChain

RAG(Retrieval-Augmented Generation)

● RAG(Retrieval-Augmented Generation)

- 생성(Generation)형 AI 모델에 실시간 정보 검색(Retrieval) 능력을 결합한 접근 방식
- 대형 언어 모델(LLM)이 외부 지식 소스와 연계하여 모델의 범용성과 적응력을 유지하면서도 정확하고 신뢰할 수 있는 답변을 생성
- 질문 응답 시스템(QA), 지식 검색, 고객 지원 등 다양한 NLP 응용 분야에 활용



RAG(Retrieval-Augmented Generation)

○ 챗GPT의 한계와 RAG의 해결책

→ 챗GPT의 한계

- ▶ 고정된 지식 : ChatGPT(~GPT-3.5)의 지식은 학습 데이터의 기준 시점에 고정되어 있어, 최신 정보를 반영하지 못한다.
- ▶ 한정된 도메인 지식 : 특정 분야의 깊이 있는 전문 지식을 제공하는 데 한계가 있다.
- ▶ 맥락 이해의 한계 : 사용자의 특정 상황이나 맥락을 완벽히 이해하고 반영하기 어렵다.
- ▶ 환각(Hallucination) : 잘못된 정보를 사실인 것처럼 제시하는 경우가 있다.

→ RAG의 해결책

- ▶ **최신 정보 접근** : 실시간으로 외부 데이터베이스를 검색하여 항상 최신 정보를 반영할 수 있다.
- ▶ **전문 지식 강화** : 특정 도메인의 전문 데이터베이스를 연결하여 깊이 있는 전문 지식을 제공할 수 있다.
- ▶ **맥락 인식 향상** : 사용자의 질문과 관련된 구체적인 정보를 검색하여 더 정확한 맥락 이해가 가능하다.
- ▶ **정보의 신뢰성 향상** : 검색된 실제 데이터를 바탕으로 답변을 생성하므로 환각 현상을 크게 줄일 수 있다.

RAG(Retrieval-Augmented Generation)

● RAG 파이프라인

→ **로딩>Loading** : 다양한 소스(텍스트 파일, PDF, 웹사이트, 데이터베이스, API 등)에서 데이터를 가져와 파이프라인에 입력합니다.

LlamaHub에서 제공하는 다양한 커넥터를 활용할 수 있습니다.

→ **인덱싱/Indexing** : 데이터를 쿼리 가능한 구조로 변환합니다.

주로 벡터 임베딩을 생성하여 데이터의 의미를 수치화하고, 관련 메타데이터를 함께 저장합니다.

→ **저장/Storing** : 생성된 인덱스와 메타데이터를 저장하여 재사용할 수 있게 합니다.

→ **쿼리/Querying** : LLM과 LlamaIndex 데이터 구조를 활용하여 다양한 방식(서브쿼리, 다단계 쿼리, 하이브리드 전략 등)으로 데이터를 검색합니다.

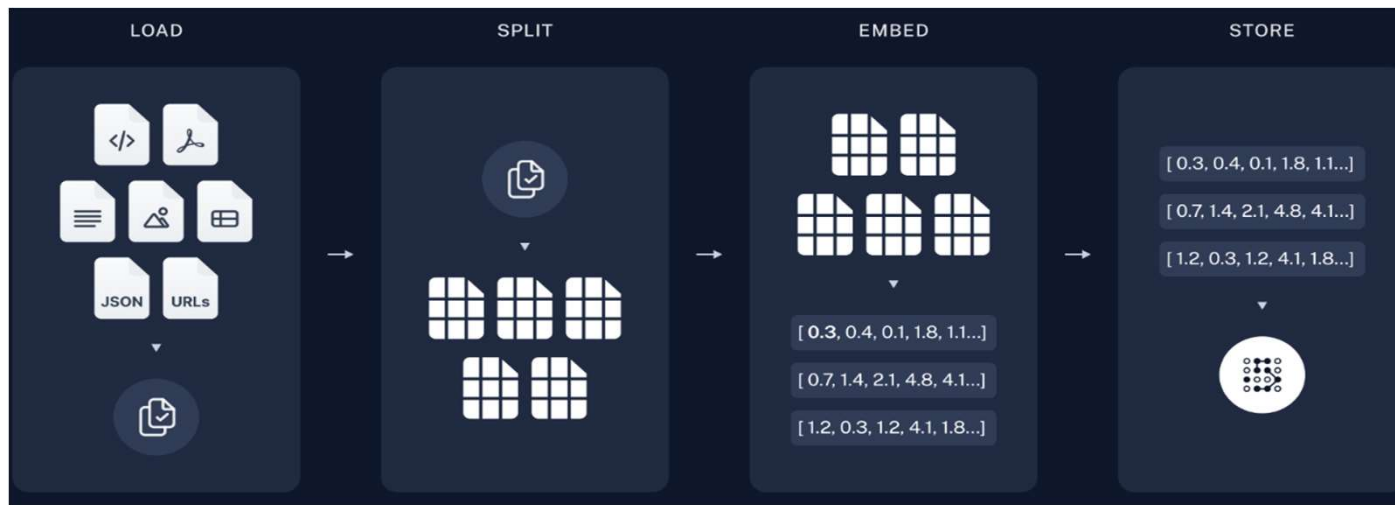
→ **평가/Evaluation** : 파이프라인의 효과성을 객관적으로 측정합니다. 응답의 정확성, 충실도, 속도 등을 평가합니다.

RAG(Retrieval-Augmented Generation)

● RAG 작동 원리

→ 색인 작업 (Indexing)

- ▶ 다양한 외부 데이터 소스(예: 코드 파일, PDF, 텍스트 문서, 이미지, 스프레드시트, JSON, URLs 등)에서 정보를 추출
- ▶ 로드 (Load) → 분할 (Split) → 임베딩 (Embed) : 분할된 데이터를 벡터 형태로 변환 → 벡터 스토어에 저장 (Store)



RAG(Retrieval-Augmented Generation)

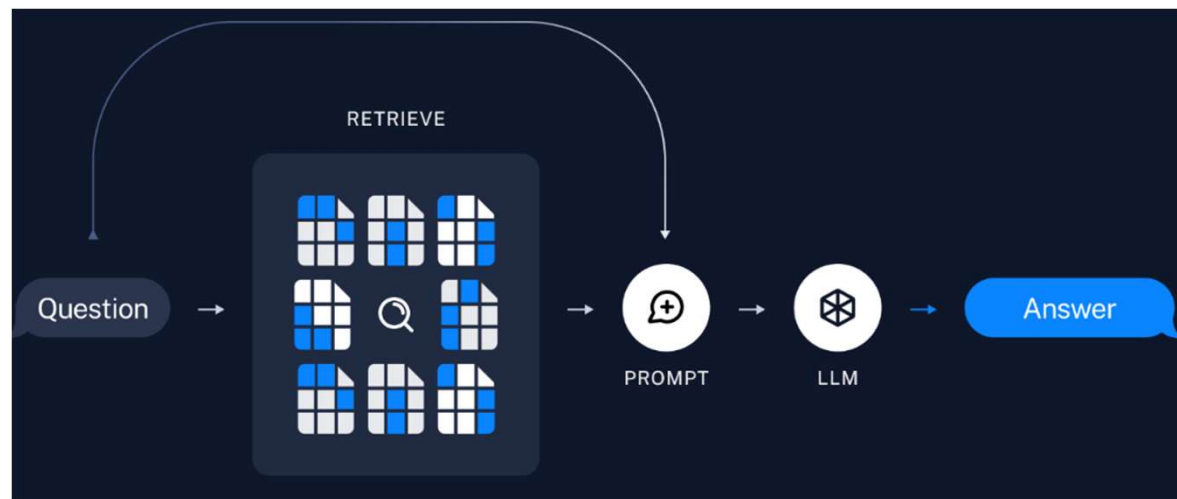
○ RAG 작동 원리

→ Retrieval (검색 단계)

- ▶ 질문 분석 (Question Analysis) : 사용자의 질문을 벡터 형태로 변환
- ▶ 벡터 검색(Vector Search) 기법을 사용하여 입력 문장과 관련 있는 문서를 찾음
- ▶ 예: 유사도 검색, TF-IDF, BM25, Dense Vector Search

→ Augmentation (정보 보강 단계)

- ▶ 검색된 정보를 언어 모델의 입력으로 제공
- ▶ LLM이 기본적으로 이해하는 언어적 지식에 검색된 데이터를 보강



RAG(Retrieval-Augmented Generation)

● RAG 작동 원리

→ Generation (생성 단계)

- ▶ LLM이 보강된 데이터를 바탕으로 자연스러운 텍스트를 생성
- ▶ 응답은 단순히 검색된 내용을 복사하는 것이 아니라, 입력과 관련된 문맥을 바탕으로 생성

RAG의 실제 적용 사례	설명
기업 내부 지식관리 시스템	회사의 내부 문서, 정책, 프로세스 등을 RAG 시스템과 연동하여 직원들이 항상 최신의 정확한 정보에 접근할 수 있게 합니다. 예: Slack에 통합된 RAG 기반 챗봇으로 회사 정책에 대한 실시간 질의 응답 제공
의료 분야 의사결정 지원 시스템	최신 의학 연구 결과와 환자의 의료 기록을 결합하여 의사의 진단과 치료 결정을 지원합니다. 예: 의사가 환자의 증상을 입력하면 관련된 최신 연구 결과와 유사 사례를 제시하는 RAG 기반 시스템
금융 자문 서비스	실시간 시장 데이터와 개인의 재무 상황을 결합하여 맞춤형 투자 조언을 제공합니다. 예: 사용자의 포트폴리오와 최신 시장 동향을 분석하여 개인화된 투자 전략을 제시하는 AI 금융 어드바이저
고객 서비스 개선	기업의 제품 정보, FAQ, 고객 이력 등을 실시간으로 참조하여 더 정확하고 맞춤형 고객 응대를 제공

RAG(Retrieval-Augmented Generation)

○ RAG(Retrieval-Augmented Generation)

→ RAG 장점

- ▶ Fine tuning에 비해 시간과 비용이 적게 소요됩니다.
 - 외부 데이터베이스를 활용하기 때문에 별도의 학습 데이터를 준비할 필요가 없습니다.
- ▶ 모델의 일반성을 유지할 수 있습니다.
 - 특정 도메인에 국한되지 않고 다양한 분야에 대한 질문에 답변할 수 있습니다.
- ▶ 답변의 근거를 제시할 수 있습니다.
 - 답변과 함께 정보 출처를 제공하여 답변의 신뢰도를 높일 수 있습니다.
- ▶ 할루시네이션 가능성을 줄일 수 있습니다.
 - 외부 데이터를 기반으로 답변을 생성하기 때문에 모델 자체의 편향이나 오류를 줄일 수 있습니다.

Fine tuning은 언어 모델(A)이 사용자의 질문에 정확히 답하기 위해 특정 도메인 지식을 공부하고 학습하여 암기한 상태로 성장시키는 것이라면, RAG는 언어 모델(A)과 도서관 사서가 협업하는 것과 같습니다. 사용자가 질문을 하면, 사서가 도서관의 책 중에서 그 질문에 대한 정보를 담고 있는 책을 찾아낸 후, 언어 모델(A)이 그 책의 내용을 참고하여 질문에 답변하는 것

RAG(Retrieval-Augmented Generation)

● Data Load 클래스

→ 다양한 형태의 데이터 소스에서 정보를 읽어 들여 처리하는 기능을 제공

클래스	설명
WebBaseLoader	웹 기반의 데이터 소스로부터 데이터를 로드 웹 페이지나 API에서 데이터를 수집할 때 사용
TextLoader	일반 텍스트 데이터를 읽어서 LangChain에서 사용할 수 있는 형태 로 변환
DirectoryLoader	프로젝트 폴더나 지정된 경로 내의 다수의 파일을 일괄적으로 처리 할 때 사용
CSVLoader	테이블 형태의 데이터를 처리하거나 데이터 분석 작업에 주로 사용
PyPDFLoader	PyPDF2 라이브러리를 사용하여 PDF 파일에서 텍스트를 추출
UnstructuredPDFLoader	구조화되지 않은 PDF 파일로부터 데이터를 추출 복잡하거나 일정하지 않은 레이아웃을 가진 PDF에서 텍스트 추출
PyMuPDFLoader	PyMuPDF (Fitz) 라이브러리를 사용하여 PDF 파일로부터 텍스트를 추출
OnlinePDFLoader	웹 URL을 통해 접근 가능한 PDF 문서를 처리할 때 사용
PyPDFDirectoryLoader	디렉토리 내의 모든 PDF 파일을 로드하여 PyPDF2를 이용해 데이터 를 추출

RAG(Retrieval-Augmented Generation)

● RAG : Load Data

→ langchain_community.document_loaders 모듈에서 WebBaseLoader 클래스를 사용하여 특정 웹사이트의 텍스트 데이터를 추출하여 Document 객체의 리스트로 변환

```
from langchain_community.document_loaders import WebBaseLoader
url =
'https://ko.wikipedia.org/wiki/%EC%9C%84%ED%82%A4%EB%B0%B1%EA%B3%BC:%EC%A0%95%EC%B1%8
5%EA%B3%BC_%EC%A7%80%EC%B9%A8' # 위키피디아 정책과 지침
loader = WebBaseLoader(url)
docs = loader.load()
print(len(docs))
print(len(docs[0].page_content))
print(docs[0].page_content[5000:6000])
```

RAG(Retrieval-Augmented Generation)

● RAG : 텍스트 분할(Text Split)

→ 텍스트 분할(Text Split) 도구는 검색 효율성을 높이기 위해 로드한 데이터를 문단, 문장 또는 구 단위의 작은 크기의 단위(chunk)로 분할을 수행합니다.

클래스	설명
CharacterTextSplitter	텍스트를 특정 문자 수에 따라 분할합니다
RecursiveCharacterTextSplitter	재귀적 방법을 사용하여 텍스트를 분할(더 나은 문맥 보존을 시도)
SentenceTextSplitter	텍스트를 문장 단위로 분할합니다.
ParagraphTextSplitter	텍스트를 단락 단위로 분할
SlidingWindowTextSplitter	슬라이딩 윈도우 방식을 사용하여 텍스트를 분할 정해진 길이의 창을 텍스트에 슬라이딩하면서 데이터를 추출하고, 겹치는 영역을 통해 정보의 연속성을 유지
FixedSizeTextSplitter	고정된 크기의 블록으로 텍스트를 나누는 방법 각 분할이 동일한 길이를 가지도록 보장하며, 일관된 처리를 요구하는 시나리오에서 효과적

RAG(Retrieval-Augmented Generation)

○ RAG : 텍스트 분할(Text Split)

→ LLM 모델이나 API의 입력 크기에 대한 제한이 있기 때문에, 제한에 걸리지 않도록 적절한 크기로 텍스트의 길이를 줄일 필요가 있습니다. (프롬프트가 지나치게 길어질 경우 중요한 정보가 상대적으로 희석되는 문제가 있을 수도 있습니다)

```
# 긴 문장을 최대 1000글자 단위로 분할하며, 200글자는 각 분할마다 겹치게 하여 문맥이 잘려나가지 않고 유지되게 합니다.
```

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
splits = text_splitter.split_documents(docs)
```

```
print(len(splits))
```

```
print(splits[10])
```

```
print(splits[10].page_content) #page_content 속성에는 분할된 텍스트 조각이 저장
```

```
print(splits[10].metadata) #원본 문서의 정보를 포함하는 메타데이터
```

RAG(Retrieval-Augmented Generation)

● RAG : 인덱싱(Indexing)

- 인덱싱은 검색 시간을 단축시키고, 검색의 정확도를 높이는 데 중요한 역할을 합니다.
- LangChain 라이브러리를 사용하여 텍스트를 임베딩으로 변환하고 벡터 저장소에 저장 후 유사성 검색을 수행

클래스	설명
OpenAIEmbeddings	OpenAI의 API를 사용하여 텍스트의 임베딩을 생성 GPT-3 등의 모델을 활용하여 주어진 텍스트에 대한 벡터 표현을 생성
HuggingFaceEmbeddings	Hugging Face의 transformers 라이브러리를 통해 다양한 사전 훈련된 모델(BERT, RoBERTa, DistilBERT 등)로부터 임베딩을 생성합니다.
SpacyEmbeddings	spaCy 라이브러리를 활용하여 텍스트에서 임베딩을 생성 다양한 언어를 지원하며, 텍스트의 문법적, 의미적 특성을 반영한 임베딩을 제공

RAG(Retrieval-Augmented Generation)

● RAG : 인덱싱(Indexing)

- 외부 데이터를 LLM에 효과적으로 통합하는 프로세스
- 인덱싱은 검색 시간을 단축시키고, 검색의 정확도를 높이는 데 중요한 역할을 합니다.
- LangChain 라이브러리를 사용하여 텍스트를 임베딩으로 변환하고 벡터 저장소에 저장 후 유사성 검색을 수행

```
# # Indexing (Texts -> Embedding -> Store)
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

vectorstore = Chroma.from_documents(documents=splits,
                                     embedding=OpenAIEmbeddings())

docs = vectorstore.similarity_search("격하 과정에 대해서 설명해주세요.")
print(len(docs))
print(docs[0].page_content)
```


RAG(Retrieval-Augmented Generation)

● RAG : 인덱싱(Indexing)

→ RAG Indexing 구성 요소

- ▶ 데이터 로드: 소스로부터 데이터를 수집
- ▶ 청크 분할: 데이터를 적절한 크기로 분할
- ▶ 임베딩 생성: 텍스트를 벡터로 변환
- ▶ 저장: 벡터 데이터베이스에 저장

→ 데이터 전처리 및 청크 분할 최적화

- ▶ 청크 크기 최적화: 2000-3000자 정도가 적절
- ▶ 오버랩 설정: 문맥 유지를 위해 500자 정도 중첩
- ▶ 메타데이터 추가: 출처, 시간 등 관련 정보 포함
- ▶ 품질 관리: 노이즈 제거 및 포맷 정규화

RAG(Retrieval-Augmented Generation)

● RAG : 인덱싱(Indexing)

→ 벡터 임베딩 전략

- ▶ 벡터 임베딩은 RAG의 검색 성능을 좌우하는 핵심 요소입니다
- ▶ 모델 선택: text-embedding-ada-002 등 고성능 모델 활용
- ▶ 차원 최적화: 일반적으로 1536 차원이 표준
- ▶ 정규화: 임베딩 벡터의 품질 유지
- ▶ 배치 처리: 대규모 데이터의 효율적 처리

RAG(Retrieval-Augmented Generation)

● RAG : Retrieval & Generation

- 사용자의 질문이나 주어진 컨텍스트에 가장 관련된 정보는 사용자의 입력을 바탕으로 쿼리를 생성하고, 인덱싱된 데이터에서 가장 관련성 높은 정보를 LangChain의 retriever()로 검색
- LLM 모델에 검색 결과와 함께 사용자의 입력을 전달
- 모델은 사전 학습된 지식과 검색 결과를 결합하여 주어진 질문에 가장 적절한 답변을 생성합니다.

```
template = "Answer the question based only on the following context:
{context}

Question: {question}"
prompt = ChatPromptTemplate.from_template(template)
model = ChatOpenAI(model='gpt-4o-mini', temperature=0)
retriever = vectorstore.as_retriever()
....
rag_chain = (
    {'context': retriever | format_docs, 'question': RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
rag_chain.invoke("격하 과정에 대해서 설명해주세요.")
```

RAG(Retrieval-Augmented Generation)

○ RAG Retrievers 최적화

→ Retrieval-Augmented Generation(RAG) 시스템에서 검색기(Retriever)는 사용자의 질문에 대해 관련성 높은 정보를 찾아주는 역할을 합니다.

→ BM25 Retriever

- ▶ 전통적인 키워드 기반 검색 방식으로, 정확한 단어 매칭에 탁월한 성능을 보입니다.
- ▶ 대규모 문서 컬렉션에서도 효율적으로 작동
- ▶ 간단한 알고리즘으로 쉽게 구현하고 튜닝할 수 있다
- ▶ 문맥을 고려하지 않아 관련 있지만 정확한 키워드를 포함하지 않은 문서를 놓칠 수 있습니다.
- ▶ 특정 단어가 자주 등장하는 긴 문서에 편향될 수 있습니다.

→ Vector Store Retriever

- ▶ 텍스트를 고차원 벡터 공간에 매핑하여 의미론적 유사성을 기반으로 검색을 수행
- ▶ 키워드가 정확히 일치하지 않아도 의미적으로 관련된 문서를 검색
- ▶ 최신 자연어 처리 기술의 이점을 활용할 수 있다
- ▶ 임베딩 생성과 저장에 많은 리소스가 필요함
- ▶ 사용된 언어 모델의 품질에 따라 성능이 크게 좌우됨

RAG(Retrieval-Augmented Generation)

○ RAG Retrievers 최적화

→ MultiQuery Retriever

- ▶ 다양한 형태의 질의를 생성하여 더 많은 관련 문서를 찾아낼 수 있습니다.
- ▶ 사용자 질의의 특정 표현에 덜 민감합니다.
- ▶ 여러 질의를 생성하고 각각에 대해 검색을 수행하므로 계산 비용이 높습니다
- ▶ 관련성이 낮은 문서도 함께 검색될 수 있습니다.

→ Ensemble Retriever

- ▶ 여러 검색 방법의 장점을 결합하여 더 포괄적이고 정확한 검색 결과를 제공
- ▶ 키워드 기반 검색과 의미 기반 검색의 장점을 모두 활용
- ▶ 단일 검색 방법보다 일반적으로 더 나은 성능을 보입니다.
- ▶ 여러 검색기를 관리하고 튜닝하는 것이 더 복잡합니다.
- ▶ 여러 검색기를 실행하고 결과를 결합하는 데 더 많은 컴퓨팅 자원이 필요합니다.

RAG(Retrieval-Augmented Generation)

○ RAG Retrievers 최적화

→ Maximal Marginal Relevance (MMR)

- ▶ 검색 결과의 다양성과 관련성 사이의 균형을 맞추는 기술
- ▶ 유사한 문서들이 상위 결과를 독점하는 것을 방지
- ▶ 질의와 관련된 문서를 우선적으로 선택
- ▶ 다양성과 관련성 사이의 적절한 균형을 위해 세심한 튜닝이 필요할 수 있습니다.
- ▶ 문서 간 유사도를 계산하는 추가 단계가 필요합니다.

RAG(Retrieval-Augmented Generation)

○ Rerank

→ 검색 결과의 순위를 재조정하는 과정

- ▶ RAG 시스템에서 Rerank는 초기 검색 결과에서 가져온 문서들의 순위를 다시 매기는 역할을 합니다.
- ▶ 사용자의 질문과 가장 관련성 높은 문서들을 상위에 배치하여 LLM이 더 정확한 답변을 생성할 수 있도록 돕습니다.

→ 한국어 특화 Reranker

- ▶ 한국어 RAG 시스템의 성능을 높이기 위해서는 한국어에 특화된 Reranker를 사용
- ▶ BAAI/bge-reranker-large 모델을 한국어 데이터로 파인튜닝한 'Dongjin-kr/ko-reranker' 모델을 활용

→ 다양한 언어를 지원하는 Reranker

- ▶ 글로벌 서비스를 제공하는 경우, 다양한 언어를 지원하는 Reranker를 사용하는 것이 효과적입니다. 'BAAI/bge-reranker-v2-m3' 모델은 여러 언어를 지원하면서도 빠른 연산 속도를 자랑합니다.

RAG(Retrieval-Augmented Generation)

● RAG기반 지능형 Q&A 웹서비스

→ LangChain으로 RAG기반 지능형 Q&A 웹서비스 개발

```
pip install langchain langchain_openai chromadb streamlit Wikipedia langchain_community
```

- ▶ WikipediaLoader : Wikipedia에서 문서를 로드합니다.
- ▶ RecursiveCharacterTextSplitter : 긴 문서를 작은 청크로 분할합니다.
- ▶ OpenAIEmbeddings : 텍스트를 벡터로 변환합니다.
- ▶ Chroma : 벡터 데이터베이스로 사용됩니다.
- ▶ ChatOpenAI : OpenAI의 GPT 모델을 사용합니다.
- ▶ RetrievalQA : 검색-질문 응답 체인을 생성합니다.
- ▶ PromptTemplate : 사용자 정의 프롬프트를 생성합니다.

RAG(Retrieval-Augmented Generation)

◉ RAG기반 지능형 Q&A 웹서비스

→ Streamlit : 데이터 앱을 만들 수 있도록 설계된 오픈 소스 라이브러리

- ▶ 복잡한 대시보드를 코딩 없이 몇 줄의 파이썬 코드만으로 생성
- ▶ 데이터 과학자와 엔지니어가 분석 결과를 시각화하고, 데모 앱을 신속하게 프로토타이핑하며, 고객이나 이해관계자와 효과적으로 소통할 수 있는 도구를 제공하는 것이 목적입니다.
- ▶ 복잡한 웹 앱 구조를 몰라도 데이터 앱을 쉽게 구축할 수 있다
- ▶ 코드를 변경하고 저장하는 즉시 앱이 자동으로 업데이트
- ▶ 데이터 표현을 위한 다양한 위젯과 차트를 내장하고 있으며, 라이브러리와의 통합을 통해 더 많은 시각화 옵션을 제공
- ▶ Streamlit Sharing을 통해 GitHub에서 직접 앱을 호스팅하고 공유할 수 있습니다

RAG(Retrieval-Augmented Generation)

● RAG기반 지능형 Q&A 웹서비스

```
import streamlit as st
import pandas as pd
import numpy as np

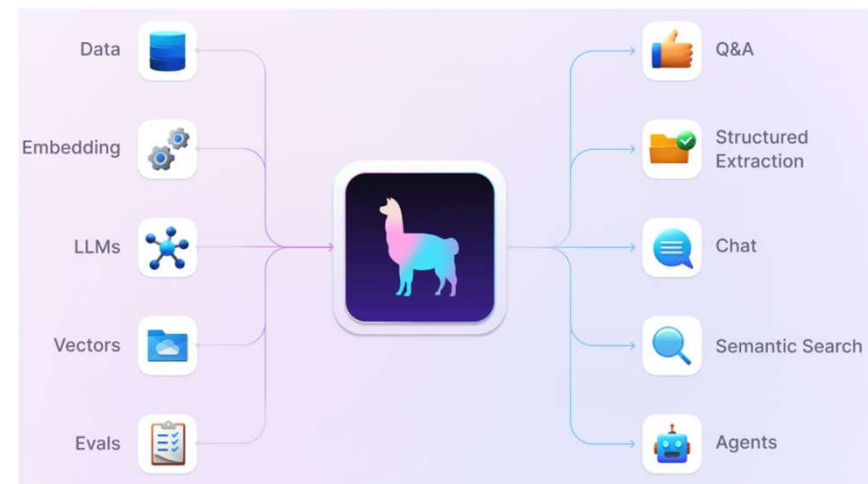
# 제목 추가
st.title('Streamlit Example')
# 데이터 생성
data = pd.DataFrame(
    np.random.randn(50, 3),
    columns=['a', 'b', 'c']
)
# 데이터 표시
st.write("Here's our random data:")
st.write(data)
# 차트 그리기
chart = st.line_chart(data)
```

```
streamlit run your_script.py
```

Llama Index

○ LlamaIndex란?

- LLM에서 학습되지 않은 데이터를 참조해서 질의응답 채팅 AI를 쉽게 만들 수 있는 오픈 소스 라이브러리
- RAG(Retrieval-Augmented Generation) 작업 흐름을 간단한 Python 코드로 구현할 수 있게 해주는 강력한 오픈소스 라이브러리
- 외부 데이터를 LLM에 주입해 더 정확하고 최신의 응답을 생성할 수 있게 해주는 기술
- 내부적으로 LangChain 라이브러리를 사용
- https://github.com/run-llama/llama_index
- https://docs.llamaindex.ai/en/stable/examples/data_connectors/simple_directory_reader//



Llama Index

○ LlamaIndex 란?

- RAG 파이프라인을 직접 구축하려면 데이터 로딩부터 인덱싱, 검색, 프롬프트 생성 등 복잡한 과정이 필요합니다.
- LlamaIndex는 RAG(Retrieval-Augmented Generation) 작업흐름을 간단한 Python 코드로 구현할 수 있게 해주는 강력한 오픈소스 라이브러리입니다. RAG는 외부 데이터를 LLM에 주입해 더 정확하고 최신의 응답을 생성할 수 있게 해주는 기술로, LlamaIndex는 이러한 RAG 파이프라인을 쉽고 효과적으로 구현할 수 있게 해줍니다
- LlamaIndex는 개인 데이터를 가져 오고 구조화하는 도구를 제공하며 데이터에 대한 고급 검색 / 쿼리 인터페이스를 생성하고 외부 응용 프레임 워크와 쉽게 통합 할 수 있도록 지원하여 ChatGPT와 함께 작동합니다.
- 응용 분야 :
 - ▶ 고객 지원을 위한 고급 챗봇 생성, 연구 및 학술계를위한 도메인 특정 응답 제공, 보건 전문가를위한 상세한 의료 정보 제공 등
- LlamaIndex를 ChatGPT와 함께 구현하려면 데이터 수집, 적재, 구조화, 쿼리 및 통합과 같은 여러 단계가 필요합니다.
- 다른 모델 및 프레임 워크와 함께 사용할 수 있도록 유연하게 설계된 솔루션

Llama Index

● LlamaIndex 주요 기능

→ 다양한 데이터 소스 지원

- ▶ 텍스트, PDF, ePub, 워드, 파워포인트, 오디오를 비롯한 다양한 파일 형식과 트위터, 슬랙, 위키 피디아 등 웹 서비스를 자체 데이터로 지정 가능

→ 로드한 데이터를 벡터 임베딩으로 변환하고 효율적인 검색을 위해 인덱싱

→ 사용자의 쿼리에 대해 가장 관련성이 높은 문서나 데이터 조각을 검색하는 다양한 검색 알고리즘을 지원

→ 검색된 관련 문서를 바탕으로 LLM을 활용하여 사용자 쿼리에 대한 정확하고 상세한 응답을 생성

→ LlamaIndex는 모듈화된 구조로 설계

- ▶ 각 컴포넌트를 필요에 따라 커스터마이징하거나 교체할 수 있습니다.

→ 다양한 벡터 데이터베이스, 임베딩 모델, LLM 등을 지원

→ 쿼리 최적화

- ▶ 복잡한 쿼리를 자동으로 분해하고 최적화하여 더 정확한 응답을 생성

→ 멀티모달 데이터 처리

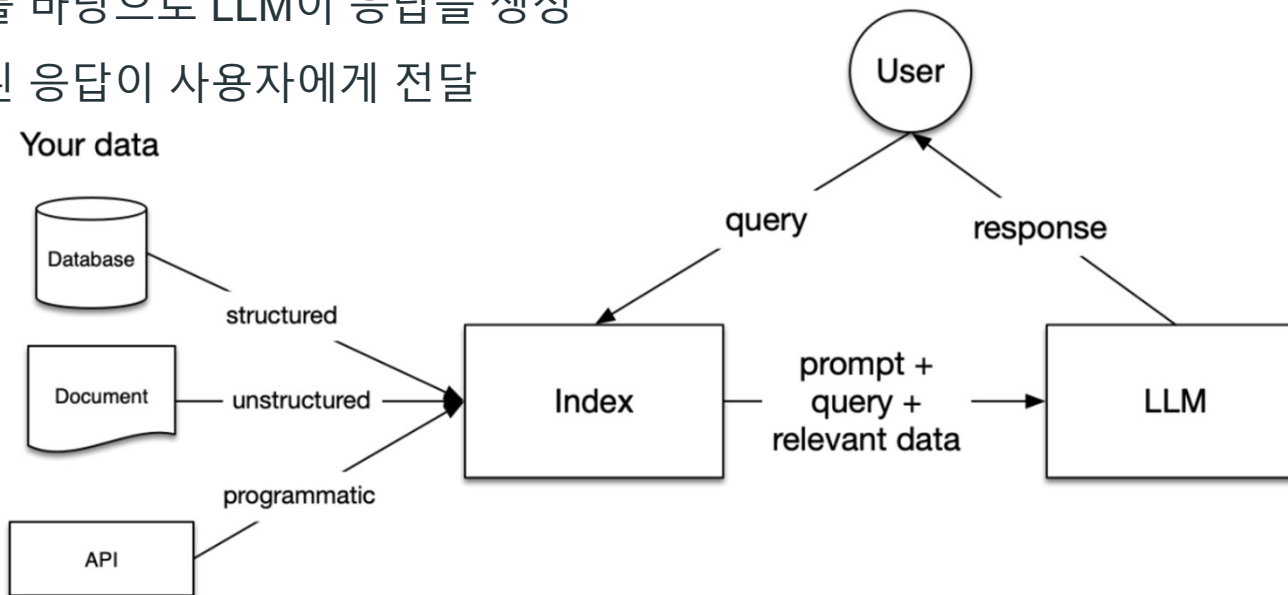
- ▶ 텍스트뿐만 아니라 이미지, 오디오 등 다양한 형태의 데이터를 처리할 수 있는 기능을 제공

Llama Index

● LlamaIndex 아키텍처

→ 다양한 데이터 소스 지원

1. 데이터는 "인덱스"라는 형태로 변환되어 쿼리에 사용될 수 있도록 준비
2. 사용자가 질문을 입력하면, 쿼리는 인덱스에 전달
3. 인덱스는 사용자의 쿼리와 가장 관련성 높은 데이터를 필터링
4. 필터링된 관련 데이터, 원래 쿼리, 적절한 프롬프트가 LLM에 전달
5. 정보를 바탕으로 LLM이 응답을 생성
6. 생성된 응답이 사용자에게 전달



Llama Index

● LlamaIndex

→ Loading

- ▶ 텍스트 파일, PDF 파일, 웹 사이트, 데이터베이스, API 등의 데이터를 파이프라인에 넣는 것이다.
- ▶ LlamaHub에 다양한 connector가 있다.

→ Document

- ▶ 데이터 소스에 대한 컨테이너이다.

→ Node

- ▶ 라마인덱스 데이터의 한 단위
- ▶ Document의 chunk이다.
- ▶ Node에는 메타데이터가 포함된다.

→ Connectors

- ▶ Reader라고도 불린다.
- ▶ 데이터를 여러 소스로부터 받아서 Document와 Node를 만든다.

Llama Index

○ LlamaIndex

→ Indexing

- ▶ 데이터를 쿼리할 수 있는 자료 구조를 생성하는 것이다.
- ▶ LLM에서는 주로 벡터 임베딩을 의미한다.

→ Embeddings

→ Storing

→ Querying

Llama Index

● LlamaIndex 데이터 로드 및 인덱스 생성

```
pip install llama-index
```

```
# 로컬 파일 시스템에서 텍스트 데이터 로드, 질문에 대한 답변을 제공하는 시스템을 구현
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
```

```
documents = SimpleDirectoryReader("data").load_data()
index = VectorStoreIndex.from_documents(documents)
#쿼리 실행
query_engine = index.as_query_engine()
response = query_engine.query("원하는 질문을 입력하세요")
print(response)
```

→ 문서 로드

→ 인덱스 생성

→ 쿼리 엔진 생성

→ 질의 응답

→ 인덱스 저장 및 로드

https://docs.llamaindex.ai/en/stable/module_guides/indexing/vector_store_index/

Llama Index

○ LlamaIndex

→ LlamaIndex를 활용한 Chatbot 개발

- ▶ 사용할 LLM api를 통해서 Custom Model 클래스를 생성
- ▶ 사용할 embedding api를 통해서 Custom Embed 클래스를 생성
- ▶ 임베딩 값을 저장할 vector db를 세팅
- ▶ 임베딩할 문서가 임베딩 가능한 사이즈보다 크다면 parser를 사용하여 나눠준다.
- ▶ 알맞은 사이즈로 분할된 문서를 임베딩한다.
- ▶ 필요하다면 노드에 metadata를 추가한다. 표 데이터의 경우는 각 column이 어떤 의미를 갖는지를 metadata에 추가할 수 있다. 이 table_summary라는 metadata는 임베딩은 되지 않지만 LLM에게만 전달되도록 설정할 수 있다.
- ▶ 사용자 쿼리가 들어오면 임베딩을 한다.
- ▶ Vector DB에서 similarity score가 높은 노드들을 뽑는다.
- ▶ 필요에 따라서 post processor를 통해서 retrieved 노드들을 가공한다.
- ▶ Prompt template을 통해서 LLM에게 요청을 보낸다.
- ▶ Retrieved nodes 수가 많아서 LLM이 한 번에 처리할 수 있는 데이터의 크기보다 크다면 refine 과정을 통해서 답을 만들어간다.

Llama Index

● Ollama

- ➔ 대규모 언어 모델(LLMs)을 로컬 환경에서 설정하고 사용할 수 있도록 지원하는 도구
 - ▶ 사용자가 대규모 언어 모델을 손쉽게 로컬 컴퓨터에 설치하고 설정할 수 있도록 도와줍니다.
 - ▶ Ollama는 모델 가중치, 구성 및 데이터를 Modelfile로 정의된 단일 패키지로 번들링합니다.
 - ▶ GPU 사용을 포함하여 설정 및 구성 세부 정보를 최적화합니다.
 - ▶ 현재 OSX와 Linux에서 사용 가능하며, Windows 사용자는 WSL 2를 통해 간접적으로 이용할 수 있습니다
 - ▶ 로컬 환경에서 LLM을 운영하면서 필요한 리소스 관리, 성능 최적화, 업데이트 관리 등을 수행할 수 있게 지원할 수 있습니다
 - ▶ 로컬에서 모델을 직접 관리함으로써 개발자와 연구자들은 실험 및 개발 과정에서 더 빠르고 유연하게 작업을 진행할 수 있습니다.

https://docs.llamaindex.ai/en/stable/getting_started/starter_example_local/

<https://ollama.com/library>

Llama Index

● llamahub

- ➔ 라마인덱스에서 제공하는 데이터 커넥터를 이용하면 다양한 데이터(PDF, ePub, 워드, 파워포인트, 오디오 등)와 웹 서비스(트위터, 슬랙, 위키백과 등)를 문서의 데이터 소스로 활용할 수 있습니다
- ▶ 데이터 로더와 에이전트 도구를 혼합하고 조합하여 맞춤형 RAG 앱을 만들거나

<https://llamahub.ai/>

Vector Store

● Vector Store

- 데이터를 임베딩(벡터 표현)으로 저장해서 벡터 공간 내에서 빠른 검색을 구현하기 위한 DB
 - ▶ 임베딩 벡터는 텍스트, 이미지, 소리 등 다양한 형태의 데이터를 벡터 공간에 매핑한 것으로, 데이터의 의미적, 시각적, 오디오적 특성을 수치적으로 표현
 - ▶ 코사인 유사도, 유클리드 거리, 맨해튼 거리 등 다양한 유사도 측정 방법을 제공
- 패키지로 제공
 - ▶ Faiss (<https://github.com/facebookresearch/faiss>)
 - ▶ Qdrant (<https://qdrant.tech>)
 - ▶ Chroma (<https://docs.trychroma.com>)
 - ▶ Milvus (<https://milvus.io>)
- 클라우드 서비스로 제공
 - ▶ Pinecone(<https://www.pinecone.io>)
 - ▶ Weaviate (<https://weaviate.io>)

Vector Store

● Vector Store

- 비정형 데이터 처리 : 텍스트, 이미지 등 다양한 형태의 비정형 데이터를 효율적으로 저장하고 검색할 수 있습니다.
- 고차원 데이터 처리 : 수백에서 수천 차원의 고차원 벡터 데이터를 빠르게 처리할 수 있습니다.
- 유사도 기반 검색 : 벡터 간 유사도를 계산하여 가장 유사한 데이터를 빠르게 찾아낼 수 있습니다.
- LLM 성능 향상 : 방대한 양의 정보를 효율적으로 검색하여 LLM에 제공함으로써 성능을 크게 향상시킬 수 있습니다.

VectorDB	설명
Faiss	Facebook AI Research에서 개발한 오픈소스 라이브러리로, 대규모 데이터셋에서 효율적인 유사도 검색을 지원
Annoy	Spotify에서 개발한 라이브러리로, 근사 최근접 이웃(Approximate Nearest Neighbor) 검색에 특화
Chroma	오픈소스 벡터 데이터베이스로, 로컬 환경에서 쉽게 사용할 수 있습니다
Pinecone	클라우드 기반의 벡터 데이터베이스 서비스로, 대규모 데이터 처리에 적합
Qdrant	비동기 작업을 지원하는 벡터 데이터베이스로, 실시간 애플리케이션에 적합

Vector Store

● Chroma Vector Store

- Chroma.from_texts 메소드를 사용하여 분할된 텍스트들을 임베딩하고, Chroma 벡터 저장소에 임베딩을 저장합니다.
- 저장소는 collection_name으로 구분
- collection_metadata에서 'hnsw:space': 'cosine'을 설정하여 유사도 계산에 코사인 유사도를 사용

```
.....
embeddings_model = OpenAIEmbeddings()
db = Chroma.from_texts(
    texts,
    embeddings_model,
    collection_name = 'history',
    persist_directory = './db/chromadb',
    collection_metadata = {'hnsw:space': 'cosine'}, # l2 is the default
)
query = '누가 한글을 창제했나요?'
docs = db.similarity_search(query)
print(docs[0].page_content)
```

Vector Store

● MMR (Maximum marginal relevance search)

- 최대 한계 관련성(Maximum Marginal Relevance, MMR) 검색 방식은 유사성과 다양성의 균형을 맞추어 검색 결과의 품질을 향상시키는 알고리즘
- 검색 쿼리에 대한 문서들의 관련성을 최대화하는 동시에, 검색된 문서들 사이의 중복성을 최소화하여, 사용자에게 다양하고 풍부한 정보를 제공
 - ▶ query: 사용자로부터 입력 받은 검색 쿼리입니다.
 - ▶ k: 최종적으로 선택할 문서의 수 (반환할 문서의 총 개수를 결정)
 - ▶ fetch_k : MMR 알고리즘을 수행할 때 고려할 상위 문서의 수 (초기 후보 문서 집합의 크기를 의미하며, 이 중에서 MMR에 의해 최종 문서가 k개 만큼 선택됩니다.)
 - ▶ lambda_mult : 쿼리와의 유사성과 선택된 문서 간의 다양성 사이의 균형을 조절합니다. ($\lambda=1$) 은 유사성만 고려하며, ($\lambda=0$)은 다양성만을 최대화합니다.

Vector Store

● FAISS (Facebook AI Similarity Search)

- Facebook AI Research에서 개발한 라이브러리
- 고차원 벡터 간의 유사성 검색을 빠르고 효율적으로 수행할 수 있도록 설계
- CPU와 GPU 모두에서 사용 가능
- 최적화된 인덱싱 구조를 사용하여 수백만 또는 수십억 개의 벡터를 처리할 수 있으며, 벡터 간의 유사성 검색을 매우 빠르게 수행
- 다양한 인덱싱 메커니즘을 제공
 - ▶ 정확한 검색을 위한 Flat 인덱스
 - ▶ 공간 효율성을 높이는 IVF (Inverted File Indexing) 인덱스
 - ▶ 양자화를 사용하여 저장 공간을 절약하는 PQ (Product Quantization) 인덱스
- 단일 머신에서와 클러스터 환경에서도 확장 가능하도록 설계

* 추천 시스템 : 사용자나 아이템의 특성을 벡터로 표현한 후, 가장 유사한 아이템을 빠르게 찾아내 추천합니다.

이미지 검색 : 이미지를 대표하는 특성 벡터를 인덱싱하여, 비슷한 이미지를 신속하게 검색

클러스터링 : 대량의 데이터를 효과적으로 클러스터링하기 위해 사용됩니다.

자연어 처리 : 문서나 단어의 임베딩 벡터를 통해 텍스트의 유사도를 판별하거나 관련 문서를 검색합니다.

Vector Store

● FAISS (Facebook AI Similarity Search)

→ <https://github.com/facebookresearch/faiss>

→ Index검색 알고리즘

알고리즘	설명
IndexFlatL2	L2 거리(유클리디안 거리)를 사용하여 쿼리 벡터와 데이터베이스 내 모든 벡터 간의 거리를 계산하는 브루트-포스 방식의 검색 인덱스 모든 벡터를 단순히 저장하고, 쿼리가 주어질 때 모든 벡터와의 거리를 계산하여 가장 가까운 이웃을 찾습니다. 정확도는 매우 높지만, 벡터의 수가 많을 때는 계산 비용이 크게 증가합니다.
IndexFlatIP	내적(dot product)을 기반으로 검색을 수행하는 인덱스 내적은 코사인 유사도와 관련이 깊으며, 벡터 간의 각도 또는 방향성을 비교하는 데 유용 추천 시스템이나 텍스트 임베딩과 같이 벡터 간의 유사도를 평가할 필요가 있을 때 사용
IndexIVFFlat	데이터베이스를 여러 개의 클러스터로 나누고 각 클러스터에 대한 인덱스를 구성합니다. 쿼리 벡터는 가장 가까운 클러스터 몇 개를 빠르게 찾고, 그 안에서만 보다 정밀한 검색을 수행합니다. 이 방식은 대규모 데이터셋에 대해 높은 검색 속도를 제공하면서도 적당한 정확도를 유지할 수 있습니다.

Vector Store

● FAISS (Facebook AI Similarity Search)

→ Index검색 알고리즘

알고리즘	설명
IndexIVFPQ	IndexIVFFlat의 개념을 확장, 데이터를 더욱 효율적으로 저장하기 위해 Product Quantization을 적용 메모리 사용량을 크게 줄이면서도 빠른 검색 속도를 제공하고, 대규모 데이터 셋에 적합 (Inverted File with Product Quantization)
IndexLSH	고차원 데이터를 저차원 공간으로 해싱하여, 유사한 데이터 포인트가 같은 해시 버킷에 떨어지도록 합니다 매우 빠른 검색 속도를 제공하지만, 정확도는 다른 방법보다 낮을 수 있습니다. (Locality-Sensitive Hashing)
IndexHNSW	그래프 기반 검색 알고리즘 (Hierarchical Navigable Small World) 다수의 계층에서 효율적인 경로를 통해 빠르게 근접 이웃을 찾습니다. 높은 차원의 데이터에서도 우수한 검색 성능과 정확도를 제공
IndexPQ	벡터를 여러 부분으로 나누고 각 부분을 양자화하여 저장합니다. (Product Quantization)
IndexSQ	벡터의 각 차원을 독립적으로 양자화합니다. (Scalar Quantizer) 차원별로 양자화를 수행하여 더욱 세밀한 제어가 가능합니다.

Vector Store

● Faiss Vector DB 초기화

```
pip install faiss-cpu # CPU 버전  
pip install faiss-gpu # GPU 버전
```

```
import faiss  
import numpy as np  
  
dimension = 64 # 벡터의 차원  
num_vectors = 1000 # 벡터의 수  
  
# 벡터 데이터 생성  
db_vectors = np.random.random((num_vectors, dimension)).astype('float32')  
# 인덱스 생성  
index = faiss.IndexFlatL2(dimension)  
index.add(db_vectors)  
# 쿼리 벡터 생성  
query_vectors = np.random.random((5, dimension)).astype('float32')  
# 가장 가까운 이웃 검색  
D, I = index.search(query_vectors, k=5)  
print('Distances:', D)  
print('Indices:', I)
```

Vector Store

● Faiss Vector DB 기반 llama_index 실습

```
pip install llama-index-vector-stores-faiss
```

```
import faiss
# faiss의 인덱스 생성
faiss_index = faiss.IndexFlatL2(1536)

from llama_index.vector_stores.faiss import FaissVectorStore

#인덱스 생
vector_store = FaissVectorStore(faiss_index=faiss_index)
storage_context = StorageContext.from_defaults(vector_store=vector_store)
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context
)
# 쿼리 엔진 생성
query_engine = index.as_query_engine()
# 질의응답
print(query_engine.query("미코의 소꿉친구 이름은?"))
```

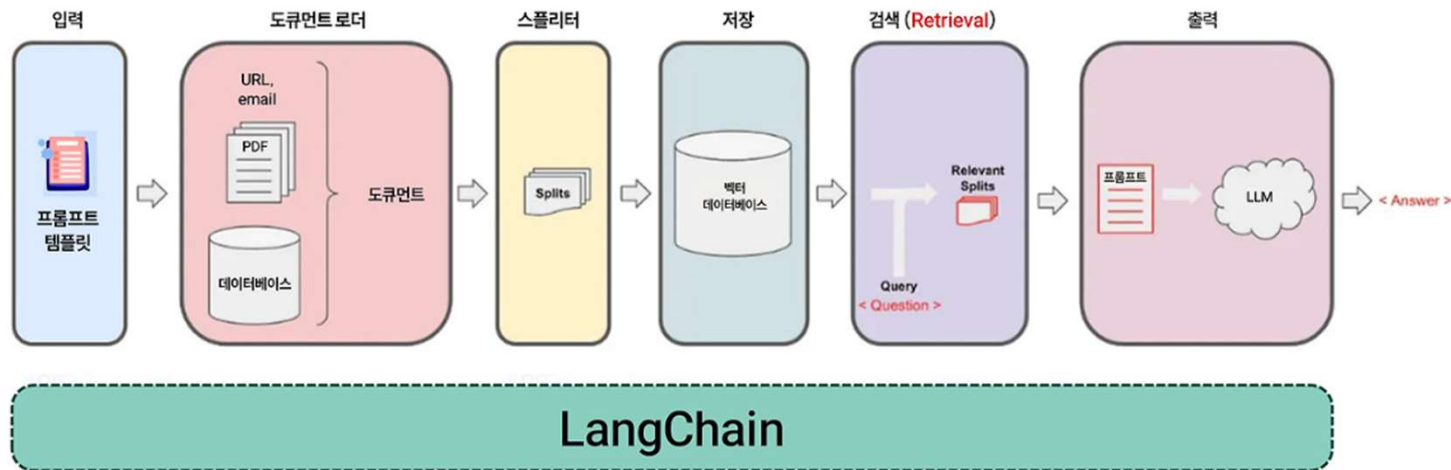
https://docs.llamaindex.ai/en/stable/examples/vector_stores/FaissIndexDemo/

LangChain

● LLM 프레임워크

→ LangChain (Language Model + Chain)

- ▶ LLM을 쉽게 구현하고 통합할 수 있도록 도와주는 오픈 소스 프레임워크
- ▶ Python과 JavaScript 라이브러리를 제공
- ▶ 2022년 10월 해리슨 체이스가 출시
- ▶ 표준 인터페이스를 제공하도록 설계
- ▶ <https://github.com/hwchase17/langchain>
- ▶ <https://python.langchain.com/docs/introduction/>



랭체인으로 구성된 RAG 워크 프로세스

LangChain

● LLM 프레임워크

→ 'langchain-core'

- ▶ 주요 추상화, 인터페이스, 그리고 핵심 기능이 포함되어 있습니다

→ 'langchain-community' 와 독립적인 파트너 패키지(langchain-openai 등)를 구분하여 제공

→ LangChain 구성 요소

▶ 랭체인 라이브러리(LangChain Libraries)

- 파이썬과 자바스크립트 라이브러리를 포함하며, 다양한 컴포넌트의 인터페이스와 통합, 이 컴포넌트들을 체인과 에이전트로 결합할 수 있는 기본 런타임

▶ 랭체인 템플릿(LangChain Templates)

- 다양한 작업을 위한 쉽게 배포할 수 있는 참조 아키텍처 모음

▶ 랭서브(LangServe)

- 랭체인 체인을 REST API로 배포할 수 있게 하는 라이브러리

▶ 랭스미스(LangSmith)

- LLM 프레임워크에서 구축된 체인을 디버깅, 테스트, 평가, 모니터링할 수 있는 개발자 플랫폼

LangChain

LangChain 모듈

→ LangChain : 대화형 AI, 정보 검색 및 자동화된 응답 시스템의 구축을 용이하게 하는 오픈 소스 프레임워크

▶ **LLM 추상화 (Large Language Model Abstraction)** : LLM 캡슐화하여 일관된 인터페이스를 제공

모델을 쉽게 교체하거나 업그레이드할 수 있다.

모델의 입력 및 출력을 관리하고, 성능 최적화를 위한 여러 전략을 적용할 수 있다

▶ **Prompt Template** : LLM에 전달되는 입력 텍스트의 구조 컨텍스트와 쿼리를 구조화

▶ **Chain** : 여러 LLM 작업을 조합하여 복잡한 작업 흐름을 구성, 입출력을 연결

▶ **Index** : 정보 검색을 위한 구조로 학습 데이터 세트에 포함되지 않은 특정 외부 데이터 소스 (내부 문서, 이메일 등)에 액세스

Document Loaders , Vector Database , Text Splitters,

▶ **Memory** : 과거 대화나 상호작용에서 얻은 정보를 저장하는 시스템

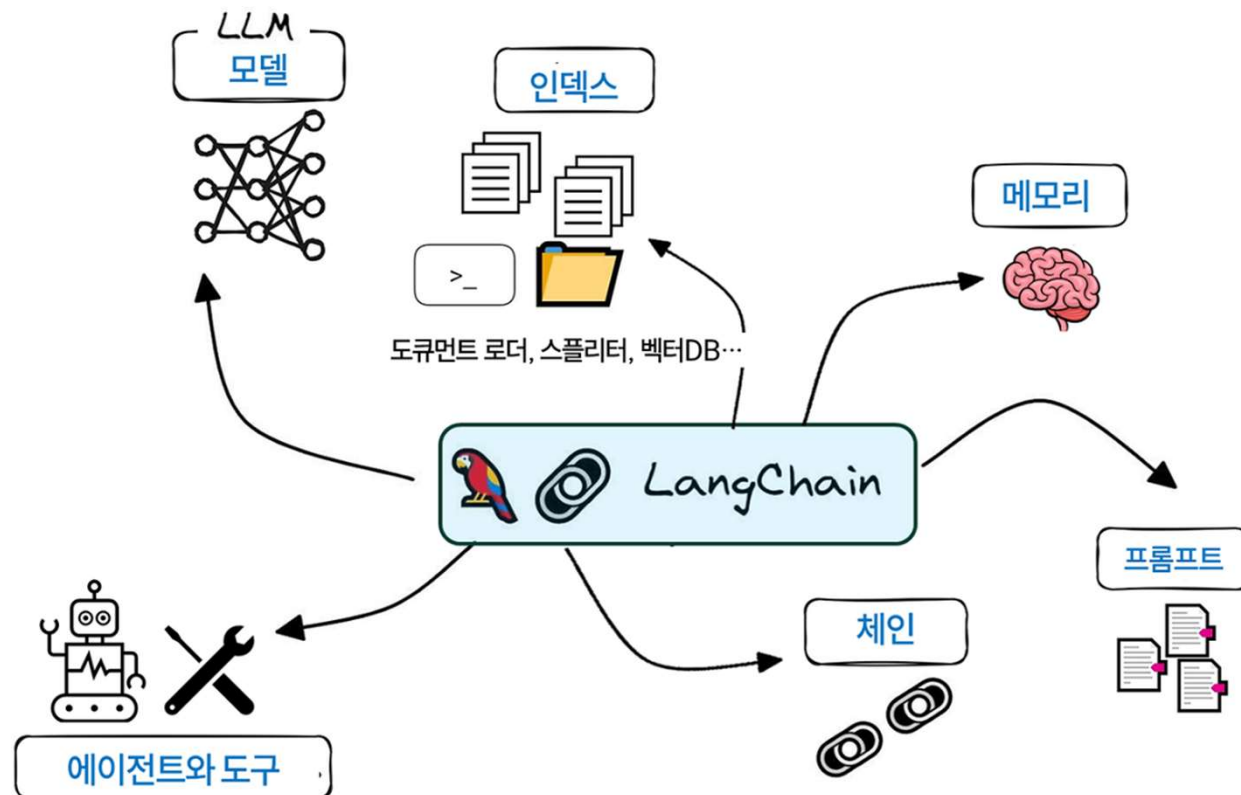
대화 전체를 기억하는 옵션과 지금까지의 대화 요약만 기억하는 요약 옵션

▶ **Agent** : 사용자의 요청에 따라 어떤 기능을 어떤 순서로 실행할 것인지 결정

서비스 챗봇, LLM과 다른 데이터 소스 , 도구 두 가지 이상을 조합하여 사용이 가능

LangChain

LangChain 모듈



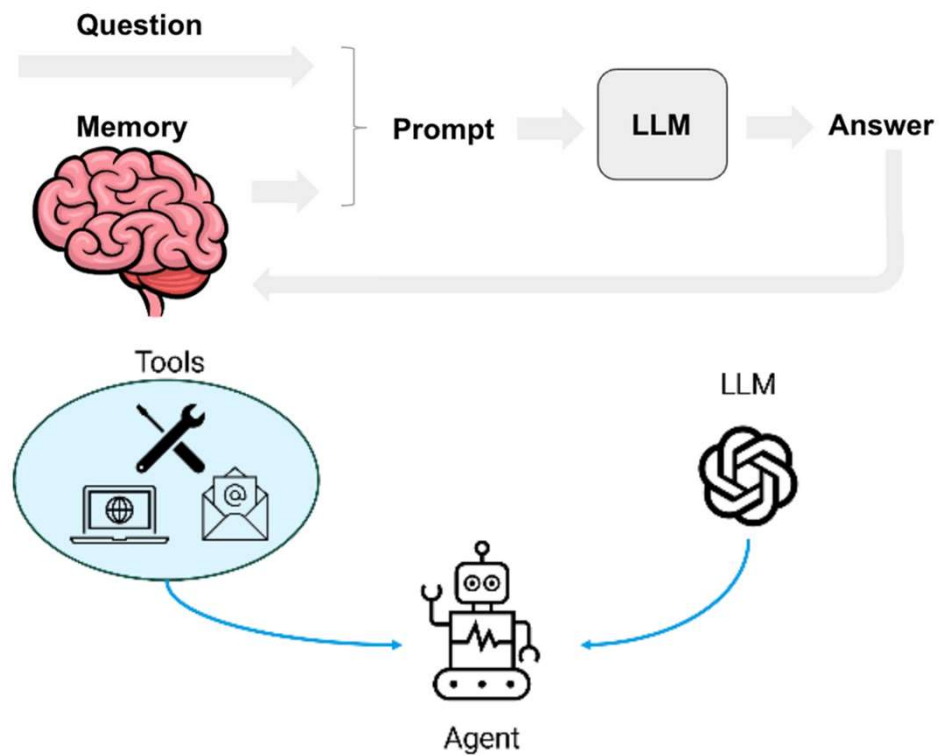
랭체인 구성도

LangChain

● 데이터 다양성 유지

→ LangChain (Language Model + Chain)

▶ LLM을 사용하는 애플리케이션 개발을 위한 오픈 소스 프레임워크



LangChain

● PromptTemplate+LLM+Chain 실습

```
# ChatPromptTemplate + ChatOpenAI + Chain
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_template("You are an expert in astronomy. Answer the question.
<Question>: {input}")

print(prompt)

from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o-mini")

# chain 연결 (LCEL)
chain = prompt | llm

# chain 호출
chain.invoke({"input": "지구의 자전 주기는?"})
```

LangChain

● PromptTemplate+LLM+Chain 실습

- AIMessage : 인공지능 모델의 메시지를 나타내는 객체(모델의 답변)
- StrOutputParser : 모델의 출력을 문자열 형태로 파싱하여 최종 결과를 반환
- ChatPromptTemplate.from_template() : 문자열 형태의 템플릿을 인자로 받아, 해당 형식에 맞는 프롬프트 객체를 생성

```
#프롬프트, LLM, 문자열 출력 파서(StrOutputParser)를 연결하여 체인을 생성
from langchain_openai import ChatOpenAI

from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# prompt + model + output parser

prompt = ChatPromptTemplate.from_template("You are an expert in astronomy. Answer the question.
<Question>: {input}")

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

output_parser = StrOutputParser()

chain = prompt | llm | output_parser          # LCEL chaining

chain.invoke({"input": "지구의 자전 주기는?"})      # chain 호출
```

LangChain

● Multi Chain 실습

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

prompt1 = ChatPromptTemplate.from_template("translates {korean_word} to English.")
prompt2 = ChatPromptTemplate.from_template(
    "explain {english_word} using oxford dictionary to me in Korean."
)

llm = ChatOpenAI(model="gpt-4o-mini")
chain1 = prompt1 | llm | StrOutputParser()

chain1.invoke({"korean_word": "미래"})

chain2 = ( {"english_word": chain1} | prompt2 | llm | StrOutputParser())

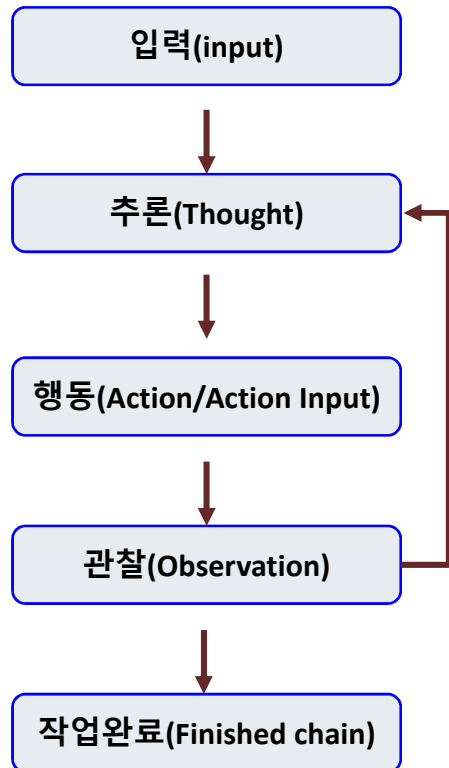
chain2.invoke({"korean_word": "미래"})
```

LangChain

● Agent

→ 사용자의 요청에 따라 어떤 기능을 어떤 순서로 실행할지 결정하는 모듈

- ▶ Chain은 미리 정해진 기능을 수행하지만 에이전트는 사용자의 요청에 따라 수행되는 기능이 달라집니다.



입력(input) : 사용자가 에이전트에게 작업을 부여하는 방식

추론(Thought) : 에이전트는 무엇을 해야 할지 생각합니다

행동(Action/Action Input) : 에이전트는 사용할 '도구'와 '도구에 대한 입력'을 결정합니다.

관찰(Observation) : 도구의 출력 결과를 관찰합니다.

'행동'은 외부 환경에 영향을 주어 새로운 정보를 '관찰'로 수집합니다.

'추론'은 외부 환경에 영향을 미치지 않지만, 상황과 맥락을 추론해서 미래의 추론과 행동에 유용한 정보를 업데이트해서 내부 상태에 영향을 미칩니다.

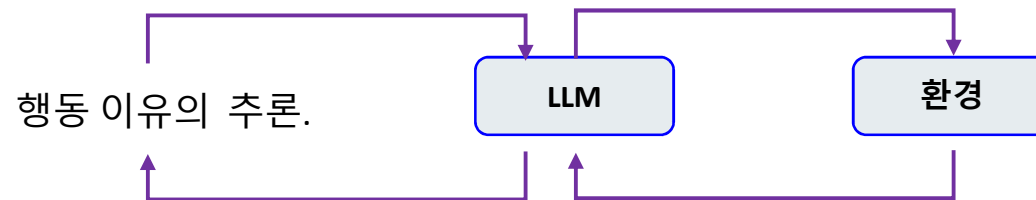
LangChain

● Agent 종류

→ ReAct(Reason + Act)

▶ 추론(Thought) 과 행동(Action)을 번갈아가며 반복하는 방식

▶ <https://research.google/blog/react-synergizing-reasoning-and-acting-in-language-models/>



LangChain

● Agent 종류

→ 특정 작업이나 상황에 맞추어 최적화

Agent	설명
conversational-react-description	대화형 에이전트 (Conversational Agents)
zero-shot-react-description	ReAct를 이용해 도구에 대한 설명만으로 사용할 도구를 결정하는 Agent
react-docstore	ReAct를 사용해 문서를 보관하는 데이터베이스인 문서 저장소와 상호작용하는 Agent
Self-ask-with-search	Intermediate Answer라는 도구를 사용하는 에이전트

LangChain

● Agent+tool 실습

```
#pip install google-search-results

import os

from langchain.agents import load_tools
from langchain.llms import OpenAI
from langchain.agents import initialize_agent

os.environ["SERPAPI_API_KEY"] = SERPAPI_API_KEY
os.environ["OPENAI_API_KEY"] = OPENAI_API_KEY

tools = load_tools( tool_names=["serpapi", "llm-math"],
                    llm=OpenAI( model="gpt-3.5-turbo-instruct", temperature=0 ) )

agent = initialize_agent( agent="zero-shot-react-description",
                          llm=OpenAI( model="gpt-3.5-turbo-instruct", temperature=0 ),
                          tools=tools, verbose=True )

agent.run("123*4를 계산기로 계산하세요")
response = agent.run("오늘 한국 서울의 날씨를 웹 검색으로 확인해주세요")
print(response)
```

LangChain

● Memory

- Agent의 처리 흐름 중 추론에서 사용되는 모듈
- 추론과 행동에 유용한 정보를 Agent 내부에 저장합니다.
- Agent와 사용자 간의 대화 내용을 기억해서 다음에 무엇을 말해야 할지 결정하는 정보로 사용
- 에이전트가 정보를 저장하고 장기간에 걸쳐 사용할 수 있게 돕는 메모리 시스템을 Agent 초기화에 설정합니다

Memory 유형	설명
Simple Memory	간단한 키-값 저장소를 사용하여 정보를 저장합니다. 복잡한 관계나 문맥을 효과적으로 처리하는 데는 한계가 있습니다
Persistent Memory	데이터베이스 또는 파일 시스템과 같은 영구 저장소를 사용하여 정보를 저장합니다 장기간에 걸친 학습과 상호작용에 유용
Contextual Memory	현재 대화의 문맥을 저장하고 관리합니다. 실시간 대화에 유용하며, 동적 변화에 빠르게 적응할 수 있습니다.
Distributed Memory	클라우드 기반 또는 분산 데이터베이스를 활용하여 대규모 데이터를 관리합니다.

LangChain

● Memory

→ LangChain에서 제공하는 Memory

Memory	설명
ConversationBufferMemroy	모든 대화 기록을 사용하는 메모리
ConversationBufferWindowMemroy	최근 K번의 대화 기록을 사용하는 메모리
ConversationTokenBufferMemroy	최신 k개 토큰의 대화 기록을 사용하는 메모리
ConversationSummaryMemroy	대화 기록의 요약을 사용하는 메모리
ConversationSummeryBufferMemroy	최신 K개 토큰의 대화 내역 요약을 사용하는 메모리
ConversationEntityrMemroy	대화 내 엔티티 정보를 저장하고 필요에 따라 사용하는 메모리
ConversationKGMemroy	지식 그래프의 정보를 사용하는 메모리
VectorStoreRetrieveMemory	대화 기록을 벡터 데이터베이스에 저장해서 상위 K개의 유사한 정보를 사용하는 메모리

→ https://python.langchain.com/docs/versions/migrating_memory/

LangChain

● Agent+tool+Memory 실습 1

```
from langchain.chains.conversation.memory import ConversationBufferMemory
memory = ConversationBufferMemory ( memory_key="chat_history",
                                    return_messages = True )

# 에이전트 생성
agent = initialize_agent(
    agent="conversational-react-description",
    llm=OpenAI(
        model="gpt-3.5-turbo-instruct",
        temperature=0 ),
    tools=tools,
    memory=memory,
    verbose=True )
agent.run("좋은 아침입니다")
agent.run("우리집 반려견 이름은 뭡실입니다.")
agent.run("우리집 반려견 이름을 불러주세요")
```

LangChain

● Agent+tool+Memory 실습 2

```
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory
from langchain_core.messages import SystemMessage
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.prompts.chat import (
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
    MessagesPlaceholder,
)
from langchain_openai import ChatOpenAI

prompt = ChatPromptTemplate( [
    MessagesPlaceholder(variable_name="chat_history"),
    HumanMessagePromptTemplate.from_template("{text}"),
])
```

LangChain

● Agent+tool+Memory 실습 2

```
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)

legacy_chain = LLMChain(
    llm=ChatOpenAI(),
    prompt=prompt,
    memory=memory,
)

legacy_result = legacy_chain.invoke({"text": "우리집 반려견 이름은 몽실이 입니다."})
print(legacy_result)
legacy_result = legacy_chain.invoke({"text": "우리집 반려견 이름은 무엇인가요?"})
```

LangChain

LangSmith

- LLM 애플리케이션 개발, 모니터링 및 테스트 를 위한 플랫폼
- Trace는 LLM 애플리케이션의 동작을 이해하기 위한 강력한 도구로서 예상치 못한 최종 결과, 에이전트가 루핑되는 이유, 체인이 예상보다 느린 이유,에이전트가 각 단계에서 사용한 토큰 수 문제를 추적하는 데 도움이 될 수 있습니다.
- 프로젝트 단위로 실행 카운트, Error 발생률, 토큰 사용량, 과금 정보등을 확인할 수 있습니다.

Name ↑↓	Feedback (7D)	Run Count (7D)	Error Rate (7D) ↑↓	% Streaming (7D)	Total Tokens (7D)	Total Cost (7D)	P50 Latency (7D) ↑↓
CH04-Models		22	0%	0%	6,501	\$0.0091835	0.00s
CH01-Basic		48	0%	9%	189,876	\$0.9631875	1.02s
llama3-agent		33	12%	94%	94,709		18.39s
CH03-OutputParser		3	0%	67%	3,332	\$0.002268	3.38s
CH02-Prompt		29	3%	24%	17,389	\$0.26302	3.30s

LangChain

LangSmith

→ 1개의 실행을 한 뒤 retrieve 된 문서의 검색 결과 뿐만 아니라, GPT의 입출력 내용에 대해서 자세하게 기록합니다. 따라서, 문서의 검색된 내용을 확인 후 검색 알고리즘을 변경해야 할지 혹은 프롬프트를 변경해야 할지 판단하는데 도움이 됩니다.

The screenshot displays the LangSmith interface. At the top, there are tabs for 'Runs', 'Threads', 'Monitor', and 'Setup'. Below these, a filter bar shows '1 filter', 'Last 7 days', and tabs for 'Root Runs', 'LLM Calls', and 'All Runs'. A table lists several runs, each with a status icon, name, input, start time, latency, and dataset. A red arrow points to the 'RunnableSequence' run in the table. To the right of the table, red text says '개별 실행(Run) 클릭하여 세부내용을 확인할 수 있습니다.' (Click on individual runs to view details).

Below the table, the 'TRACE' section shows a detailed view of the 'RunnableSequence' run. It includes a 'Collapse' button, 'Stats', and 'Most relevant' dropdown. The trace shows a sequence of steps: 'map:key:context' (1.08s), 'Retriever' (1.08s), 'Retriever' (0.03s), 'Retriever' (1.05s), 'reorder_documents' (0.00s), and 'ChatOpenAI gpt-4-turbo-preview' (28.89s). Red arrows point to each of these steps. To the right of the trace, the 'RunnableSequence' details are shown, including the 'Run' tab, 'Feedback', and 'Metadata'. The 'Input' section shows a list of inputs, including a prompt and a code snippet for importing and using a retriever.

```
1 input: |-
2   !ask
3
4   from langchain_community.retrievers import BM25Retriever
5   에 대한 공식문서 내용을 알려줘
```

YAML ↕

LangChain

● Runnable 프로토콜

- 프로토콜은 사용자가 사용자 정의 체인을 쉽게 생성하고 관리할 수 있도록 설계된 핵심적인 개념
- 일관된 인터페이스를 사용하여 다양한 타입의 컴포넌트를 조합하고, 복잡한 데이터 처리 파이프라인을 구성

알고리즘	설명
invoke	주어진 입력에 대해 체인의 모든 처리 단계를 한 번에 실행하고, 결과를 반환 단일 입력에 대해 동기적으로 작동
batch	입력 리스트에 대해 체인을 호출하고, 각 입력에 대한 결과를 리스트로 반환 여러 입력에 대해 동기적으로 작동, 배치 처리
stream	입력 데이터를 스트림으로 처리하며, 반복적으로 데이터를 체인을 통해 보내고 결과를 받습니다. invoke보다 더 유연하고 동적인 데이터 처리를 가능하게 합니다. 연속적인 데이터 흐름이나, 대용량 데이터 처리나 실시간 데이터 처리에 유용
비동기 버전	ainvoke, abatch, astream 등의 메소드는 각각의 동기 버전에 대한 비동기 실행을 지원

LangChain

● Runnable 프로토콜

→ Runnable 객체들은 데이터 처리, 조건 판단, 외부 API 호출 등의 기능을 수행하며, 데이터를 전달

Runnable 클래스	설명
RunnablePassthrough	조건 분기 없이 입력을 받아 다음 단계로 전달
RunnableMap	입력 데이터에 함수를 매핑하여 새로운 데이터를 생성 입력 텍스트에 대해 언어 모델을 사용하여 처리한 결과를 매핑
RunnableFilter	주어진 조건에 따라 입력 데이터를 필터링합니다 데이터 스트림에서 특정 조건을 만족하는 데이터만을 추출하고자 할 때 사용
RunnableReduce	여러 데이터 조각을 하나로 통합하는 리듀스(reduce) 작업을 수행 여러 입력값을 받아 하나의 결과로 합치는 과정에서 사용
RunnableInvoke	외부 함수나 메소드를 호출하고 결과를 반환 외부 시스템과의 통합을 구현할 수 있으며, API 호출 등에 사용
RunnableSequence	여러 Runnable 객체를 순차적으로 실행하는 시퀀스를 구성 복잡한 처리 로직을 단계별로 구성하고자 할 때 사용
RunnableParallel	여러 Runnable 객체를 병렬로 실행 동시에 여러 작업을 수행하고 결과를 동시에 수집

LangChain

● Runnable 프로토콜 사용 실습

```
from langchain.prompts import ChatPromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.schema.output_parser import StrOutputParser

# 1. 컴포넌트 정의
prompt = ChatPromptTemplate.from_template("지구과학에서 {topic}에 대해 간단히 설명해주세요.")
model = ChatOpenAI(model="gpt-4o-mini")
output_parser = StrOutputParser()

# 2. 체인 생성
chain = prompt | model | output_parser

from langchain.prompts import ChatPromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.schema.output_parser import StrOutputParser
```

LangChain

● Runnable 프로토콜 사용 실습

```
# batch 메소드 사용
topics = ["지구 공전", "화산 활동", "대륙 이동"]
results = chain.batch([{"topic": t} for t in topics])
for topic, result in zip(topics, results):
    print(f"{topic} 설명: {result[:50]}...")
```

```
# stream 메소드 사용
stream = chain.stream({"topic": "지진"})
print("stream 결과:")
for chunk in stream:
    print(chunk, end="", flush=True)
print()
```

LangChain

● Runnable 프로토콜 사용 실습

```
import nest_asyncio
import asyncio

# nest_asyncio 적용 (구글 코랩 등 주피터 노트북에서 실행 필요)
nest_asyncio.apply()

# 비동기 메소드 사용 (async/await 구문 필요)
async def run_async():
    result = await chain.ainvoke({"topic": "해류"})
    print("ainvoke 결과:", result[:50], "...")

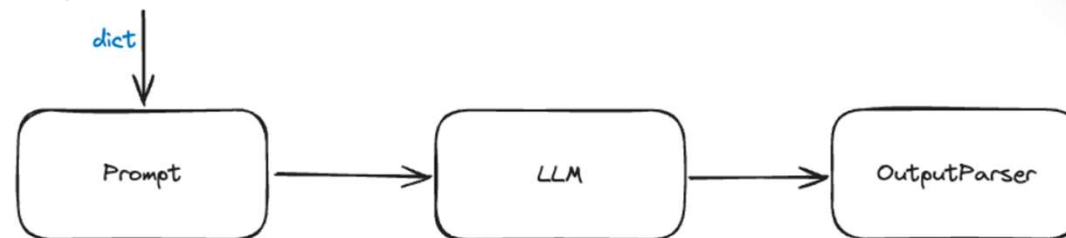
asyncio.run(run_async())
```

LangChain

● LCEL (LangChain Expression Language)

- 언어 모델과 상호작용하는 프로세스를 간소화하고 구조화하기 위해 설계된 스크립팅 언어
- 다양한 구성 요소를 단일 체인으로 결합, 복잡한 데이터 파이프라인을 구축할 때 특히 유용합니다
- 데이터 흐름을 명확하게 관리하고, 다양한 처리 단계에서 데이터를 쉽게 변환하거나 합성할 수 있도록 지원합니다.
- | 기호는 unix 파이프 연산자와 유사하며, 서로 다른 구성 요소를 연결하고 한 구성 요소의 출력을 다음 구성 요소의 입력으로 전달합니다.
- 다른 시스템이나 라이브러리와 쉽게 통합될 수 있도록 설계
- LCEL은 조건문과 반복문을 지원하여, 동적인 결정과 데이터 처리가 필요한 경우에 유연하게 대응

`{"question": "{topic}에 대해 쉽게 설명해주세요."}`



```
chain = prompt | model | output_parser
```

LangChain

● LCEL 인터페이스

- 사용자 정의 체인을 가능한 쉽게 만들 수 있도록, Runnable 프로토콜을 구현
- 표준 인터페이스로, 사용자 정의 체인을 정의하고 표준 방식으로 호출하는 것을 쉽게 만듭니다.

메서드	설명
stream	주어진 토픽에 대한 데이터 스트림을 생성하고, 이 스트림을 반복하여 각 데이터의 내용(content)을 즉시 출력합니다.
invoke	입력에 대해 체인을 호출합니다.
batch	여러 개의 딕셔너리를 포함하는 리스트를 인자로 받아, 각 딕셔너리에 있는 topic 키의 값을 사용하여 일괄 처리를 수행 config 딕셔너리는 max_concurrency 키를 통해 동시에 처리할 수 있는 최대 작업 수를 설정
astream	비동기 스트림을 생성하며, 주어진 토픽에 대한 메시지를 비동기적으로 처리합니다.
ainvoke	비동기적으로 입력에 대해 체인을 호출합니다.
abatch	비동기적으로 입력 목록에 대해 체인을 호출합니다.
astream_log	최종 응답뿐만 아니라 발생하는 중간 단계를 스트리밍합니다

LangChain

● Prompt

→ Prompt : 사용자와 언어 모델 간의 대화에서 질문이나 요청의 형태로 제시되는 입력문
모델이 어떤 유형의 응답을 제공할지 결정하는 데 중요한 역할을 합니다.

질문은 명확하고 구체적이어야 합니다.

예시: "다음 주 주식 시장에 영향을 줄 수 있는 예정된 이벤트들은 무엇일까요?"는 "주식 시장에 대해 알려주세요."보다 더 구체적이고 명확한 질문입니다.

모델이 문맥을 이해할 수 있도록 필요한 배경 정보를 제공하는 것이 좋습니다.

예시: "2020년 미국 대선 결과를 바탕으로 현재 정치 상황에 대한 분석을 해주세요."

핵심 정보에 초점을 맞추고, 불필요한 정보는 배제합니다.

예시: "2021년에 발표된 삼성전자의 ESG 보고서를 요약해주세요."

열린 질문을 통해 모델이 자세하고 풍부한 답변을 제공하도록 유도합니다.

예시: "신재생에너지에 대한 최신 연구 동향은 무엇인가요?"

얻고자 하는 정보나 결과의 유형을 정확하게 정의합니다.

예시: "AI 윤리에 대한 문제점과 해결 방안을 요약하여 설명해주세요."

대화의 맥락에 적합한 언어와 문체를 선택합니다.

예시: 공식적인 보고서를 요청하는 경우, "XX 보고서에 대한 전문적인 요약을 부탁드립니다."와 같이 정중한 문체를 사용합니다.

LangChain

● Prompt

→ LLM 모델에 입력할 프롬프트 구성 요소

- ▶ **지시** : 언어 모델에게 어떤 작업을 수행하도록 요청하는 구체적인 지시
- ▶ **예시** : 요청된 작업을 수행하는 방법에 대한 하나 이상의 예시
- ▶ **맥락** : 특정 작업을 수행하기 위한 추가적인 맥락
- ▶ **질문** : 어떤 답변을 요구하는 구체적인 질문

예시: 제품 리뷰 요약

지시: "아래 제공된 제품 리뷰를 요약해주세요."

예시: "예를 들어, '이 제품은 매우 사용하기 편리하며 배터리 수명이 길다'라는 리뷰는 '사용 편리성과 긴 배터리 수명이 특징'으로 요약할 수 있습니다."

맥락: "리뷰는 스마트워치에 대한 것이며, 사용자 경험에 초점을 맞추고 있습니다."

질문: "이 리뷰를 바탕으로 스마트워치의 주요 장점을 두세 문장으로 요약해주세요."

LangChain

● Prompt Template

→ langchain_core.prompts 모듈의 PromptTemplate.from_template ()를 사용하여 문자열 템플릿으로부터 PromptTemplate 인스턴스를 생성

```
from langchain_core.prompts import PromptTemplate

template = "{country}의 수도는 어디인가요?"

prompt = PromptTemplate( template=template,
                        input_variables=["country"],
)

print(prompt)

prompt.format(country="대한민국")
```

→ PromptTemplate 클래스는 문자열을 기반으로 프롬프트 템플릿을 생성하고, + 연산자를 사용하여 직접 결합하는 동작을 지원합니다.

- ▶ 문자열 + 문자열
- ▶ PromptTemplate + PromptTemplate
- ▶ PromptTemplate + 문자열

LangChain

● Prompt Template

→ PromptTemplate은 프롬프트 템플릿 기본 클래스

```
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
# 문자열 템플릿 결합 (PromptTemplate + PromptTemplate + 문자열)
combined_prompt = (
    prompt_template
    + PromptTemplate.from_template("\n\n아버지를 아버지라 부를 수 없습니다.")
    + "\n\n{language}로 번역해주세요."
)
print(combined_prompt)
# combined_prompt.format(name="홍길동", age=30, language="영어")
llm = ChatOpenAI(model="gpt-4o-mini")
chain = combined_prompt | llm | StrOutputParser()
chain.invoke({"age":30, "language":"영어", "name":"홍길동"})
```

LangChain

● ChatPrompt Template

- ➔ 대화형 상황에서 여러 메시지 입력을 기반으로 단일 메시지 응답을 생성하는 데 사용됩니다.
- ▶ `ChatPromptTemplate.from_messages()`를 사용하여 메시지 리스트로부터 `ChatPromptTemplate`인스턴스를 생성하는 방식
- ▶ 메시지는 튜플(tuple) 형식으로 구성하며, (role, message) 로 구성하여 리스트로 생성할 수 있습니다.
- ▶ `format_messages()`는 사용자의 입력을 프롬프트에 동적으로 삽입하여, 최종적으로 대화형 상황을 반영한 메시지 리스트를 생성합니다.
 - `SystemMessage`: 시스템의 기능을 설명합니다.
 - `HumanMessage`: 사용자의 질문을 나타냅니다.
 - `AIMessage`: AI 모델의 응답을 제공합니다.
 - `FunctionMessage`: 특정 함수 호출의 결과를 나타냅니다.
 - `ToolMessage`: 도구 호출의 결과를 나타냅니다.

LangChain

● ChatPrompt Template

→ chat_prompt, llm, StrOutputParser()를 순차적인 파이프라인으로 연결하여 chain 사용 실습

```
#튜플 형태의 메시지 목록으로 프롬프트 생성 (type, content)
from langchain_core.prompts import ChatPromptTemplate
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "이 시스템은 천문학 질문에 답변할 수 있습니다."),
    ("user", "{user_input}"),
])
messages = chat_prompt.format_messages(user_input="태양계에서 가장 큰 행성은 무엇인가요?")
print(messages)

from langchain_core.output_parsers import StrOutputParser

chain = chat_prompt | llm | StrOutputParser()
chain.invoke({"user_input": "태양계에서 가장 큰 행성은 무엇인가요?"})
```

LangChain

● ChatPrompt Template

→ SystemMessagePromptTemplate와 HumanMessagePromptTemplate를 메시지 리스트로 구성, 대화형 프롬프트 생성

```
from langchain_core.prompts import SystemMessagePromptTemplate,
HumanMessagePromptTemplate

chat_prompt = ChatPromptTemplate.from_messages(
    [ SystemMessagePromptTemplate.from_template("이 시스템은 천문학 질문에 답변할 수 있습니다."),
      HumanMessagePromptTemplate.from_template("{user_input}"),  ]
)

messages = chat_prompt.format_messages(user_input="태양계에서 가장 큰 행성은 무엇인가요?")
#print(messages)

chain = chat_prompt | llm | StrOutputParser()
chain.invoke({"user_input": "태양계에서 가장 큰 행성은 무엇인가요?"})
```

LangChain

● MessagePlaceholder

- LangChain은 포맷하는 동안 렌더링할 메시지를 완전히 제어할 수 있는 MessagePlaceholder를 제공
- 메시지 프롬프트 템플릿에 어떤 역할을 사용해야 할지 확실하지 않거나 서식 지정 중에 메시지 목록을 삽입하려는 경우 유용

```
chat_prompt = ChatPromptTemplate.from_messages( [
    ( "system",
      "당신은 요약 전문 AI 어시스턴트입니다. 당신의 임무는 주요 키워드로 대화를 요약하는 것입니다.", ),
    MessagesPlaceholder(variable_name="conversation"),
    ("human", "지금까지의 대화를 {word_count} 단어로 요약합니다."),
  ] )
print(chat_prompt)
formatted_chat_prompt = chat_prompt.format(word_count=5,
    conversation=[ ("human", "안녕하세요! 저는 오늘 새로 입사한 테디 입니다. 만나서 반갑습니다."),
    ("ai", "반가워요! 앞으로 잘 부탁 드립니다."), ],
)
print(formatted_chat_prompt)
```

LangChain

● Few-shotPromptTemplate

- Few-shot 학습은 언어 모델에 몇 가지 예시를 제공하여 특정 작업을 수행하도록 유도하는 기법으로 더 정확하고 일관된 응답을 생성할 수 있습니다.
- 특정 도메인이나 형식의 질문에 대해 모델의 성능을 향상시키는 데 효과적입니다.
- SemanticSimilarityExampleSelector는 입력 질문과 의미적 유사성을 기반으로 가장 관련성 높은 예제를 선택합니다

```
from langchain_core.prompts import PromptTemplate

example_prompt = PromptTemplate.from_template("질문: {question}\n{answer}")

examples = [ { "question": "지구의 대기 중 가장 많은 비율을 차지하는 기체는 무엇인가요?",
               "answer": "지구 대기의 약 78%를 차지하는 질소입니다." },
              { "question": "광합성에 필요한 주요 요소들은 무엇인가요?",
               "answer": "광합성에 필요한 주요 요소는 빛, 이산화탄소, 물입니다."
            },
            .... ]
```


LangChain

● Few-shotPromptTemplate 주요 매개변수

매개변수	설명
examples	답변 예시
example_prompt	프롬프트 템플릿
prefix	접두사
suffix	접미사
input_variables	입력 변수
example_separator	구분 기호

LangChain

● Few-shotPromptTemplate

```
from langchain_core.prompts import FewShotPromptTemplate

prompt = FewShotPromptTemplate(
    examples=examples,          # 사용할 예제들
    example_prompt=example_prompt, # 예제 포맷팅에 사용할 템플릿
    suffix="질문: {input}",      # 예제 뒤에 추가될 접미사
    input_variables=["input"],   # 입력 변수 지정
)

# 새로운 질문에 대한 프롬프트를 생성하고 출력합니다.
print(prompt.invoke({"input": "화성의 표면이 붉은 이유는 무엇인가요?").to_string())

from langchain_chroma import Chroma

from langchain_core.example_selectors import SemanticSimilarityExampleSelector

from langchain_openai import OpenAIEmbeddings
```

LangChain

● Few-shotPromptTemplate

```
example_selector = SemanticSimilarityExampleSelector.from_examples(  
    examples,          # 사용할 예제들  
    OpenAIEmbeddings(), # 임베딩 모델  
    Chroma,           # 벡터 저장소  
    k=1,              # 선택할 예제 수  
)  
# 새로운 질문에 대해 가장 유사한 예제를 선택합니다.  
question = "화성의 표면이 붉은 이유는 무엇인가요?"  
selected_examples = example_selector.select_examples({"question": question})  
print(f"입력과 가장 유사한 예제: {question}")  
for example in selected_examples:  
    print("\n")  
    for k, v in example.items():  
        print(f"{k}: {v}")
```

LangChain

● FewShotChatMessagePromptTemplate

→ 고정 예제를 사용한 Few-shot 프롬프팅 기법 실습

```
from langchain_core.prompts import ChatPromptTemplate, FewShotChatMessagePromptTemplate

examples = [
    {"input": "지구의 대기 중 가장 많은 비율을 차지하는 기체는 무엇인가요?", "output": "질소입니다."},
    {"input": "광합성에 필요한 주요 요소들은 무엇인가요?", "output": "빛, 이산화탄소, 물입니다."},
]

example_prompt = ChatPromptTemplate.from_messages(
    [
        ("human", "{input}"),
        ("ai", "{output}"),
    ]
)

few_shot_prompt = FewShotChatMessagePromptTemplate(
    example_prompt=example_prompt,
    examples=examples,
)

.....
```

LangChain

● FewShotChatMessagePromptTemplate

→ 동적 Few-shot 프롬프팅 실습 : ExampleSelector를 사용해서 입력에 따라 전체 예제 세트에서 가장 관련성 높은 예제만 선택

```
final_prompt = ChatPromptTemplate.from_messages([
    ("system", "당신은 과학과 수학에 대해 잘 아는 교육자입니다."),
    few_shot_prompt,
    ("human", "{input}"),
])
from langchain_openai import ChatOpenAI
model = ChatOpenAI(model="gpt-4o-mini", temperature=0.0)
chain = final_prompt | model
result = chain.invoke({"input": "지구의 자전 주기는 얼마인가요?"})
print(result.content)
```

LangChain

◉ ExampleSelector

→ 여러 개의 답변 예시가 포함된 프롬프트 템플릿을 만들려면 ExampleSelector 클래스를 상속받아 사용합니다.

Selector	설명
LengthBasedExampleSelector	문자열 길이를 기준으로 사용할 답변 예시를 선택 examples : 답변 예시 example_prompt : 프롬프트 템플릿 max_length : 문자열의 최대 길이
SemanticSimilarityExampleSelector	입력과 가장 유사한 답변 예제 선택 입력과 코사인 유사도가 가장높은 임베딩을 사용 examples : 답변 예시 embeddings : 임베디드 생성 클래스 vectorstore_cls : 임베디스 유사 검색 클래스 k : 답변 예시 개수
MaxMarginalRelevanceExampleSelector	다양성을 최적화하면서 입력과 가장 유사한 답변 예시 조합을 기반으로 답변 예시를 선택 examples : 답변 예시 embeddings : 임베디드 생성 클래스 vectorstore_cls : 임베디스 유사 검색 클래스 k : 답변 예시 개수

LangChain

● LengthBasedExampleSelector 실습

→ FewShotPromptTemplate은 주어진 예시를 토대로 한 템플릿을 사용하여 새로운 입력에 대한 반의어를 예측

```
examples = [ {"input": "밝은", "output": "어두운"},
              {"input": "재미있는", "output": "지루한"},
              {"input": "활기찬", "output": "무기력한"},
              {"input": "높은", "output": "낮은"},
              {"input": "빠른", "output": "느린"},]
example_prompt = PromptTemplate(
    input_variables=["input","output"],
    template="입력: {input}\n출력: {output}",
)
example_selector = LengthBasedExampleSelector(
    examples=examples, # 답변 예시
    example_prompt=example_prompt, # 프롬프트 템플릿
    max_length=10, # 문자열의 최대 길이 )
```

LangChain

◉ LLM 클래스

- LM 클래스는 텍스트 문자열을 입력으로 받아 처리한 후, 텍스트 문자열을 반환합니다.
- 광범위한 언어 이해 및 텍스트 생성 작업에 사용됩니다.
- 주로 단일 요청에 대한 복잡한 출력을 생성하는 데 적합
- 문서 요약, 콘텐츠 생성, 질문에 대한 답변 생성 등 복잡한 자연어 처리 작업을 수행할 수 있습니다.
- 랭체인은 OpenAI의 GPT 시리즈, Cohere의 LLM, Hugging Face의 Transformer 모델 등 다양한 LLM 제공 업체와의 통합을 지원합니다.

```
from langchain_openai import OpenAI  
  
llm = OpenAI()  
  
llm.invoke("한국의 대표적인 관광지 3군데를 추천해주세요.")
```


LangChain

● Chat Model 클래스

- Chat Model 클래스는 메시지의 리스트를 입력으로 받고, 하나의 메시지를 반환합니다.
- 사용자와의 상호작용을 통한 연속적인 대화 관리에 더 적합합니다.
- 대화의 맥락을 유지하면서 적절한 응답을 생성하는 데 중점을 둡니다.

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

chat = ChatOpenAI()
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "이 시스템은 여행 전문가입니다."),
    ("user", "{user_input}"),
])

chain = chat_prompt | chat
chain.invoke({"user_input": "안녕하세요? 한국의 대표적인 관광지 3군데를 추천해주세요."})
```

LangChain

● Chat Model 클래스

→ Anthropic의 Claude 모델

```
pip install -U langchain-anthropic
```

```
from langchain_anthropic import ChatAnthropic
llm = ChatAnthropic( model="claude-3-haiku-20240307",
    temperature=0,
    max_tokens=200,
    api_key="인증키 입력" )
messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "LLM은 어떤 원리로 작동하나요? 100자 이내로 설명해주세요."},
]
ai_msg = llm.invoke(messages)
print(ai_msg)
```

LangChain

● Chat Model 클래스

→ Google의 Gemini 모델

```
pip install -U langchain-google-genai
```

```
from langchain_google_genai import ChatGoogleGenerativeAI
llm = ChatGoogleGenerativeAI( model="gemini-1.5-flash",
    temperature=0,
    max_output_tokens=200,
    google_api_key="인증키 입력" )
messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "LLM은 어떤 원리로 작동하나요? 100자 이내로 설명해주세요."},
]
ai_msg = llm.invoke(messages)
print(ai_msg.content)
```

LangChain

● 모델 클래스에 파라미터 전달

→ LLM 모델에 직접 파라미터를 전달 :

- ▶ 모델 인스턴스 생성 시 또는 모델을 호출하는 시점에 파라미터를 인수로 제공
- ▶ 파라미터를 사용자가 직접 세밀하게 조정할 수 있다
- ▶ 다양한 설정을 실험하거나 특정 요청에 대해 최적화된 응답을 생성하는 데 유용

```
from langchain_openai import ChatOpenAI

params = { "temperature": 0.7,      # 생성된 텍스트의 다양성 조정
           "max_tokens": 100,      # 생성할 최대 토큰 수 }

kwargs = { "frequency_penalty": 0.5, # 이미 등장한 단어의 재등장 확률
           "presence_penalty": 0.5,  # 새로운 단어의 도입을 장려
           "stop": ["\n"]           # 정지 시퀀스 설정}

model = ChatOpenAI(model="gpt-4o-mini", **params, model_kwargs = kwargs)
question = "태양계에서 가장 큰 행성은 무엇인가요?"
response = model.invoke(input=question)
print(response)
```

LangChain

● 모델 클래스에 파라미터 전달

→ invoke 메소드를 사용하여 새로운 호출을 할 때 모델의 기본 파라미터(params)를 설정

```
params = { "temperature": 0.7,      # 생성된 텍스트의 다양성 조정
           "max_tokens": 10,        # 생성할 최대 토큰 수 }
response = model.invoke(input=question, **params)
print(response.content)
```

→ bind 메소드를 사용하여 모델 인스턴스에 파라미터를 추가로 제공

▶ 특정 모델 설정을 기본값으로 사용하고, 특수한 상황에서 일부 파라미터를 다르게 적용하고 싶을 때 사용

```
.....
model = ChatOpenAI(model="gpt-4o-mini", max_tokens=100)
messages = prompt.format_messages(user_input="태양계에서 가장 큰 행성은 무엇인가요?")
before_answer = model.invoke(messages)

.....
chain = prompt | model.bind(max_tokens=10)
after_answer = chain.invoke({"user_input": "태양계에서 가장 큰 행성은 무엇인가요?"})
```

LangChain

● Output Parser

- 모델의 출력을 사용자가 원하는 형식으로 변환합니다.
- 복잡한 텍스트 데이터에서 구조화된 정보를 얻을 수 있습니다.
- 모델 출력에서 불필요한 정보를 제거하거나, 응답을 더 명확하게 만드는 등의 후처리 작업을 수행합니다.
- 출력 데이터를 기반으로 특정 조건에 따라 다른 처리를 수행합니다.
- 원시 텍스트 출력에서 필요한 정보(예: 날짜, 이름, 위치 등)를 추출합니다

출력 파서 (Output Parser)의 사용 사례

자연어 처리(NLP) 애플리케이션 : 질문 답변 시스템에서 정확한 답변만을 추출하여 사용자에게 제공

데이터 분석 : 대량의 텍스트 데이터에서 특정 패턴이나 통계 정보를 추출하여 분석 보고서를 생성

챗봇 개발 : 대화형 모델의 출력을 분석하여 사용자의 의도를 파악하고, 적절한 대화 흐름을 유지합니다.

콘텐츠 생성 : 생성된 콘텐츠에서 중요한 정보를 요약하거나, 특정 형식(예: 블로그 포스트, 뉴스 기사)에 맞게 콘텐츠를 재구성합니다.

LangChain

● Output Parser

Output Parser	설명
PydanticOutputParser	언어 모델의 출력을 더 구조화된 정보로 변환 단순 텍스트 형태의 응답 대신, 사용자가 필요로 하는 정보를 명확하고 체계적인 형태로 제공
CommaSeparatedListOutputParser	쉼표로 구분된 항목 목록을 반환
StructuredOutputParser	LLM에 대한 답변을 dict 형식으로 구성하고 key/value 쌍으로 갖는 여러 필드를 반환
JsonOutputParser	사용자가 원하는 JSON 스키마를 지정
PandasDataFrameOutputParser	구조화된 데이터를 다루기 위한 포괄적인 도구 세트를 제공하여, 데이터 정제, 변환 및 분석과 같은 작업에 다 양하게 활용될 수 있습니다.
DatetimeOutputParser	LLM의 출력을 datetime 형식으로 파싱하는 데 사용
EnumOutputParser	
OutputFixingParser	출력 파싱 과정에서 발생할 수 있는 오류를 자동으로 수 정하는 기능을 제공

LangChain

● CommaSeparatedListOutputParser

→ 모델이 생성한 텍스트에서 쉼표(,)로 구분된 항목을 추출하여 리스트 형태의 구조화된 데이터 형태로 변환

```
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import CommaSeparatedListOutputParser

output_parser = CommaSeparatedListOutputParser()
format_instructions = output_parser.get_format_instructions()
#print(format_instructions)

prompt = PromptTemplate( template="List five {subject}.\n{format_instructions}",
                        input_variables=["subject"],
                        partial_variables={"format_instructions": format_instructions},
                        )

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

chain = prompt | llm | output_parser

chain.invoke({"subject": "popular Korean cuisine"})
```


LangChain

● JSONOutputParser

- JsonOutputParser는 모델의 출력을 JSON으로 해석
- 자료구조를 의미하는 CusineRecipe 클래스를 Pydantic BaseModel을 사용하여 정의

```
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.pydantic_v1 import BaseModel, Field

class CusineRecipe(BaseModel):                                # 자료구조 정의 (pydantic)
    name: str = Field(description="name of a cuisine")
    recipe: str = Field(description="recipe to cook the cuisine")

output_parser = JsonOutputParser(pydantic_object=CusineRecipe) # 출력 파서 정의
format_instructions = output_parser.get_format_instructions()
print(format_instructions)

prompt = PromptTemplate( template="Answer the user query.\n{format_instructions}\n{query}\n",
                        input_variables=["query"],
                        partial_variables={"format_instructions": format_instructions}, )

chain = prompt | model | output_parser
chain.invoke({"query": "Let me know how to cook Bibimbap"})
```

LangChain

● Chain

→ 여러 개의 LLM이나 프롬프트의 입출력을 연결할 수 있는 모듈

▶ 프롬프트 템플릿, 모델, 임의의 함수, 다른 체인 등으로 구성

제네릭 체인	설명
LLMChain	사용자 입력을 기반으로 프롬프트 템플릿으로 프롬프트를 생성해서 LLM을 호출합니다.
SimpleSequentialChain	입출력이 하나씩 있는 여러 개의 체인을 연결합니다.
SequentialChain	여러 개의 입출력을 가진 체인을 연결합니다.

인덱스 체인	설명
RetrievalQA	질의응답을 수행합니다.
RetrievalQAWaithSourceChain	소스가 있는 질의응답을 수행합니다.
SummaryizeChain	요약

LangChain

● Chain

→ 여러 개의 LLM이나 프롬프트의 입출력을 연결할 수 있는 모듈

▶ 프롬프트 템플릿, 모델, 임의의 함수, 다른 체인 등으로 구성

유틸리티 체인	설명
PALChain	질문을 입력으로 받아 파이썬 코드로 변환하고, 파이썬 REPL을 통해 실행합니다.
SQLDatabaseChain	데이터베이스에 대한 질문을 입력으로 받아 SQL 쿼리로 변환하고 쿼리를 실행합니다.
LLMMathChain	수학 문제를 입력등로 받아 파이썬 코드로 변환해 파이썬 REPL로 실행합니다.
LLMBashChain	질문을 입력으로 받아 bash 명령어로 변환해서 터미널에서 실행합니다.
LLMCheckerChain	질문을 받고, LLMChain으로 그 질문에 답하고 다른 LLMChain에서 그 답변을 자체적으로 확인합니다.
LLMRequestsChain	URL과 파라미터 입력을 받아 이를 기반으로 웹 요청을 생성해서 실행합니다.
OpenAIModerationChain	OpenAI의 콘텐츠 모더레이션 API를 사용합니다

LangChain

● tool

- Agent의 처리 흐름 중 행동에서 수행하는 특정 기능
- 외부 환경에 영향을 주어 새로운 정보를 관찰로 획득

```
from langchain.tools import load_tools
```

```
tools = load_tools(tool_names=["faiss", "openai_embeddings", "sentiment_analysis"])
```

LangChain

● tool

tool	설명
python_repl	파이썬 명령을 실행하는 도구 입력은 유효한 파이썬 명령, 출력이 필요한 경우 print() 사용
serpapi	웹 검색을 위한 도구
wolfram-alpha	Wolfram Alpha를 검색하는 도구 수학, 과학, 기술, 문화, 사회, 일상 생활에 관한 질문에 대한 답을 찾아야 할 때 유용
requests	파이썬의 requests를 실행하는 도구
terminal	터미널에서 명령을 실행하는 도구
pal-math	복잡한 수학 문제를 푸는 데 도움이 되는 도구 (LLM 필수)
pal-colored-objects	물체의 위치 및 색상 속성에 대한 추론 문제를 해결하는데 도움이 되는 도구 (LLM 필수)
llm-math	수학 관련 문제를 해결하는데 도움이 되는 도구 (LLM 필수)
open-meteo-api	OpenMeteo API에서 기상 정보를 얻을 때 유용한 도구 (LLM 필수)
News-api	현재 뉴스 기사의 주요 헤드라인에 대한 정보를 얻는 데 도움이 되는 도구(LLM 필수)
tmdb-api	The Movie Database에서 정보를 검색할 때 유용한 도구(LLM 필수)

LangChain

● tool

tool	설명
google-search	구글 맞춤 검색을 사용하는 도구
searx-search	SearxNG의 메타 검색을 사용하는 도구
google-serper	저비용 구글 검색 API를 사용하는 도구
wikipedia	위키피디아의 정보를 검색할 수 있는 도구
podcast-api	Listen Notes Podcast API를 사용해 모든 팟캐스트 또는 에피소스를 검색하는 도구 (LLM 필수)
openweathermap-api	OpenWeatherMap API를 사용하는 도구

● Prompt engineering

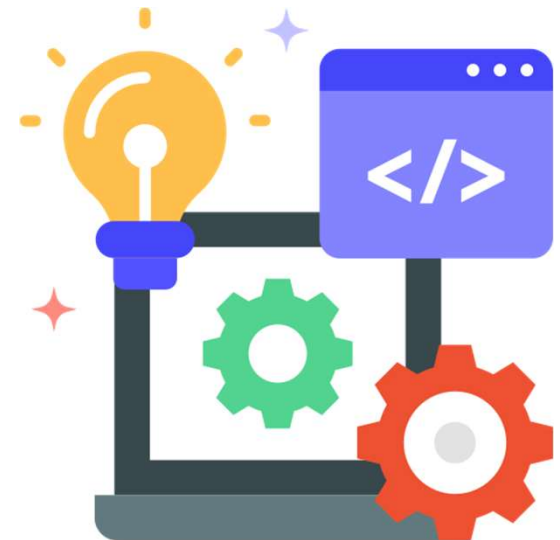
→ <https://platform.openai.com/docs/guides/prompt-engineering>

● ChromaDB

- AI에 최적화된 오픈 소스 임베딩 데이터베이스
- 다양한 데이터 포인트를 임베딩 형태로 저장하고, 빠르고 효율적인 유사성 검색을 수행
- 다중 언어 지원
- 간편한 통합 및 설정

Appendix

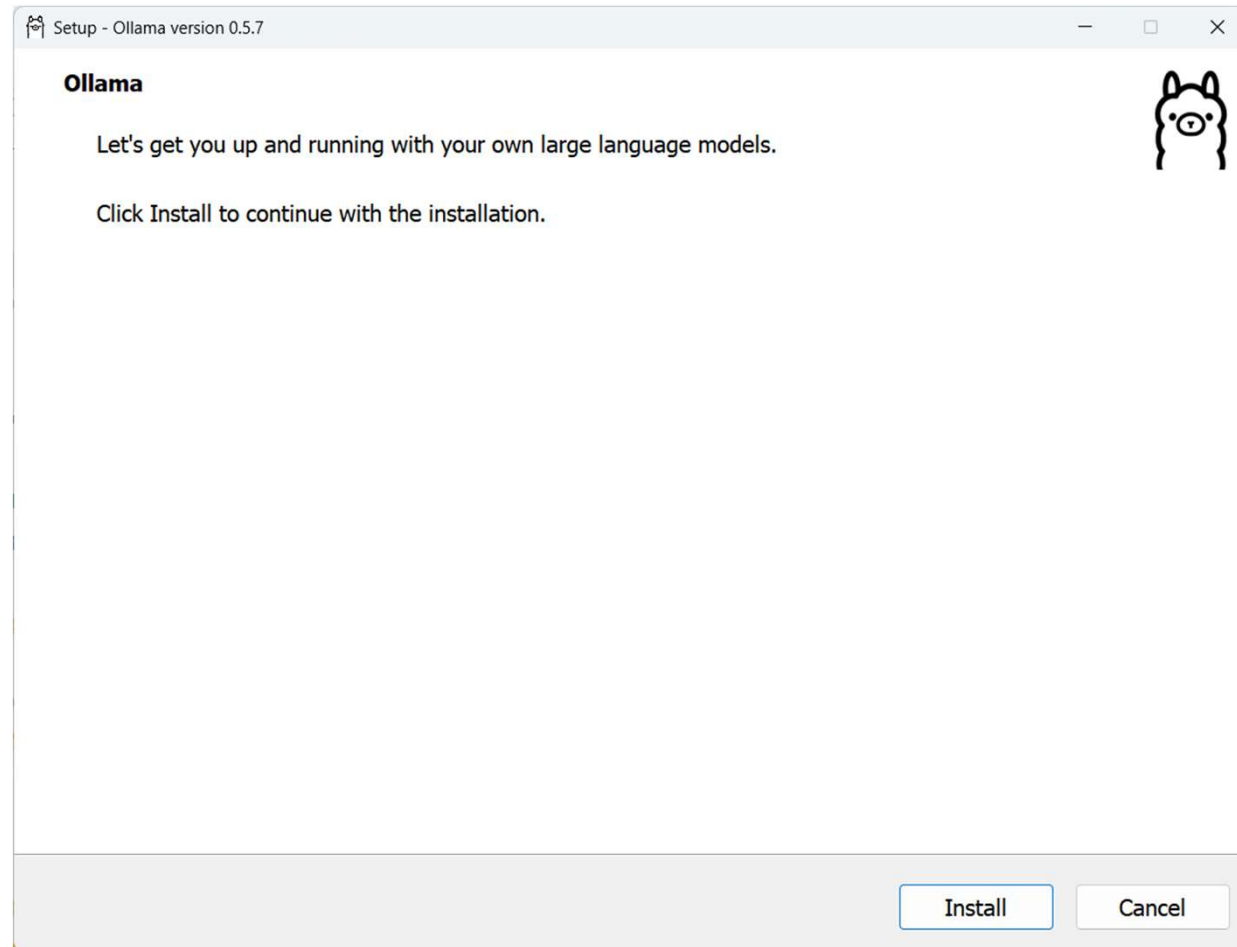
- 개발환경
-
-



로컬 LLM 환경 구성

ollama

→ <https://ollama.com/download>



로컬 LLM 환경 구성

ollama

→ ollama

Usage:

```
ollama [flags]
ollama [command]
```

Available Commands:

serve	Start ollama
create	Create a model from a Modelfile
show	Show information for a model
run	Run a model
pull	Pull a model from a registry
push	Push a model to a registry
list	List models
cp	Copy a model
rm	Remove a model
help	Help about any command

Flags:

-h, --help	help for ollama
-v, --version	Show version information

Use "ollama [command] --help" **for** more information about a command.

로컬 LLM 환경 구성

ollama

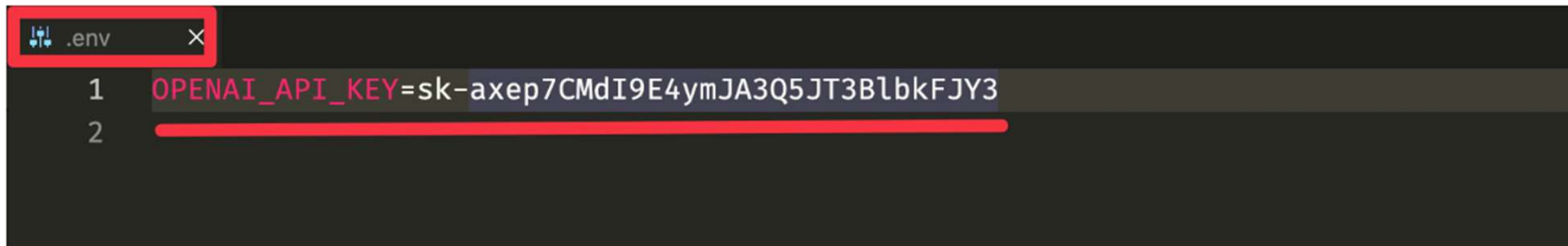
→ <https://ollama.com/library>

Model	Parameters	Size	Download Command
Llama 3	8B	4.7GB	<code>ollama run llama3</code>
Llama 3	70B	40GB	<code>ollama run llama3:70b</code>
Phi-3	3.8B	2.3GB	<code>ollama run phi3</code>
Mistral	7B	4.1GB	<code>ollama run mistral</code>
Neural Chat	7B	4.1GB	<code>ollama run neural-chat</code>
Starling	7B	4.1GB	<code>ollama run starling-lm</code>
Code Llama	7B	3.8GB	<code>ollama run codellama</code>
Llama 2 Uncensored	7B	3.8GB	<code>ollama run llama2-uncensored</code>
LLaVA	7B	4.5GB	<code>ollama run llava</code>
Gemma	2B	1.4GB	<code>ollama run gemma:2b</code>
Gemma	7B	4.8GB	<code>ollama run gemma:7b</code>
Solar	10.7B	6.1GB	<code>ollama run solar</code>

openai key 생성

○ .env 파일 설정

- 프로젝트 루트 디렉토리에 .env 파일을 생성합니다.
- .env 파일에 OPENAI_API_KEY=방금복사한 키를 입력 한 뒤 Ctrl + S 를 눌러 저장하고 파일을 닫습니다.



```
.env
1 OPENAI_API_KEY=sk-axep7CMdI9E4ymJA3Q5JT3B1bkFJY3
2
```

LangChain 업데이트

```
!pip install -r https://raw.githubusercontent.com/teddylee777/langchain-kr/main/requirements.txt
```

API KEY를 환경변수로 관리하기 위한 설정 파일

설치: pip install python-dotenv

```
from dotenv import load_dotenv
```

API KEY 정보로드

```
load_dotenv()
```

openai key 생성

● .env 파일 설정

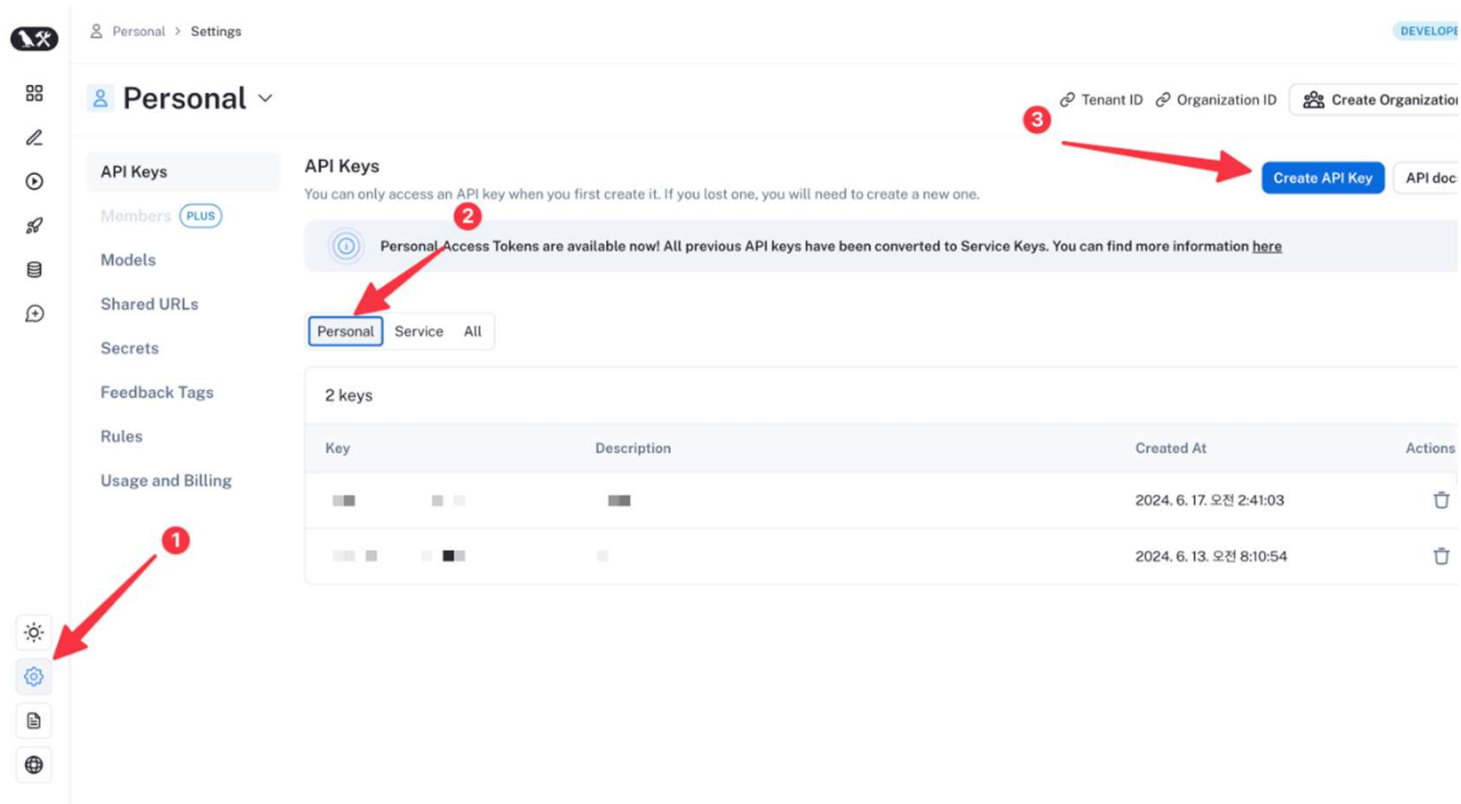
→ API Key 가 잘 설정되었는지 확인합니다.

```
import os  
  
print(f"[API KEY]\n{os.environ['OPENAI_API_KEY']}")
```

LangSmith API Key

LangSmith API Key 발급

- <https://smith.langchain.com/> 으로 접속하여 회원가입을 진행합니다.
- 가입후 이메일 인증하는 절차를 진행해야 합니다.
- 왼쪽 톱니바퀴(Settings) - 가운데 "Personal" - "Create API Key" 를 눌러 API 키를 발급 받습니다.

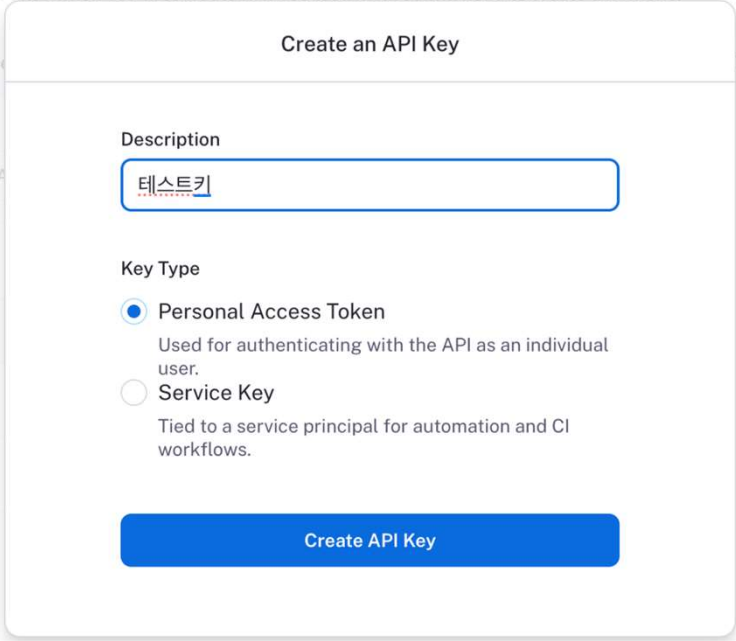


LangSmith API Key

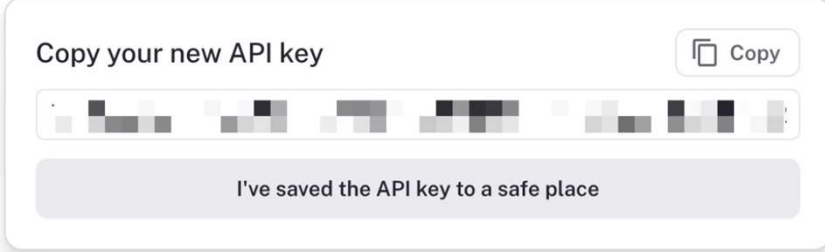
LangSmith API Key 발급

→ 생성한 키를 유출하지 않도록 안전한 곳에 복사해 두세요.

→



The screenshot shows a web form titled "Create an API Key". It has a "Description" field with the text "테스트키" (Test Key) entered. Below this is the "Key Type" section with two radio button options: "Personal Access Token" (which is selected) and "Service Key". The "Personal Access Token" option has a subtext: "Used for authenticating with the API as an individual user." The "Service Key" option has a subtext: "Tied to a service principal for automation and CI workflows." At the bottom of the form is a blue button labeled "Create API Key".



The screenshot shows a confirmation screen titled "Copy your new API key". It features a long, pixelated representation of the API key. To the right of the key is a "Copy" button with a clipboard icon. Below the key is a light purple button with the text "I've saved the API key to a safe place".

LangSmith API Key

LangSmith API Key 발급

→ <https://smith.langchain.com/onboarding?organizationId=befcfb5f-a055-492c-a8c5-0cfa2ad8a855&step=1>

1. **Generate API Key**

2. Install dependencies

Python

TypeScript

```
1 pip install -U langchain langchain-openai
```

Copy

3. Configure environment to connect to LangSmith.

Project Name

pr-damp-yak-93

```
1 v LANGSMITH_TRACING=true
2 LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
3 LANGSMITH_API_KEY="<your-api-key>"
4 LANGSMITH_PROJECT="pr-damp-yak-93"
5 OPENAI_API_KEY="<your-openai-api-key>"
```

Copy

4. Run any LLM, Chat model, or Chain. Its trace will be sent to this project.

```
1 from langchain_openai import ChatOpenAI
2
3 llm = ChatOpenAI()
4 llm.invoke("Hello, world!")
```

Copy

LangSmith API Key

○ .env 파일에 LangSmith 키 설정

- LANGCHAIN_TRACING_V2 : "true" 로 설정하면 추적을 시작합니다.
- LANGCHAIN_ENDPOINT : <https://api.smith.langchain.com> 변경하지 않습니다.
- LANGCHAIN_API_KEY : 이전 단계에서 발급받은 키 를 입력합니다.
- LANGCHAIN_PROJECT : 프로젝트 명 을 기입하면 해당 프로젝트 그룹으로 모든 실행 (Run) 이 추적됩니다.

```
OPENAI_API_KEY=sk-iiYAv7V7YXfhy7QDgkz3T3  
LANGCHAIN_TRACING_V2=false  
LANGCHAIN_ENDPOINT=https://api.smith.langchain.com  
LANGCHAIN_API_KEY=ls__69f264b1b0774d55  
LANGCHAIN_PROJECT=랭스미스에_표기할_프로젝트명
```

Jupyter Notebook 혹은 코드에서 추적을 활성화

● .env 파일에 LangSmith 키 설정

→ 설정한 추적이 활성화 되어 있고, API KEY 와 프로젝트 명이 제대로 설정되어 있다면,

```
from dotenv import load_dotenv  
  
load_dotenv()
```

→ 프로젝트 명을 변경하거나, 추적을 변경하고 싶을 때는 아래의 코드로 변경할 수 있습니다.

```
import os  
  
os.environ["LANGCHAIN_TRACING_V2"] = "true"  
os.environ["LANGCHAIN_ENDPOINT"] = "https://api.smith.langchain.com"  
os.environ["LANGCHAIN_PROJECT"] = "LangChain 프로젝트명"  
os.environ["LANGCHAIN_API_KEY"] = "LangChain API KEY 입력"
```

langchain-teddynote

🔴 langchain-teddynote

→ langchain 관련 기능을 보다 더 편리하게 사용하기 위한 목적의 패키지

```
pip install langchain-teddynote
```

→ .env 파일에 LangSmith API 키가 설정 되어 있어야 합니다.(LANGCHAIN_API_KEY)

```
from langchain_teddynote import logging  
  
# 프로젝트 이름을 입력합니다.  
logging.langsmith("원하는 프로젝트명")
```

→ 추적을 원하지 않을 때는 다음과 같이 추적을 끌 수 있습니다.

```
from langchain_teddynote import logging  
  
# set_enable=False 로 지정하면 추적을 하지 않습니다.  
logging.langsmith("랭체인 튜토리얼 프로젝트", set_enable=False)
```

→ <https://serpapi.com/manage-api-key>

The screenshot shows the SerpApi website's API Key management interface. On the left is a dark sidebar with the SerpApi logo and a list of navigation items: API DASHBOARD, Your Account, Api Key (highlighted in orange), Billing Information, Change Plan, Edit Profile, Extra Credits, Invoices, Manage Plan, Team, and Your Searches. The main content area has a light blue header with a search bar, a search count of '0 / 100 searches', and a user profile icon. Below the header, the page title 'Api Key' is displayed with a key icon. The main section, titled 'Your Private API Key', shows a long alphanumeric key: '45ae94787d3be6441259dd586435f22b4019989f87897d44130b0ed2a70493b7'. To the right of the key is a clipboard icon. A blue button labeled 'Regenerate API Key' is positioned below the key.

SerpApi

API DASHBOARD

- Your Account
- Api Key**
- Billing Information
- Change Plan
- Edit Profile
- Extra Credits
- Invoices
- Manage Plan
- Team
- Your Searches

Search / 0 / 100 searches

Api Key

Your Private API Key

45ae94787d3be6441259dd586435f22b4019989f87897d44130b0ed2a70493b7

Regenerate API Key

Huggingface Key 발급

○ Huggingface Key 발급

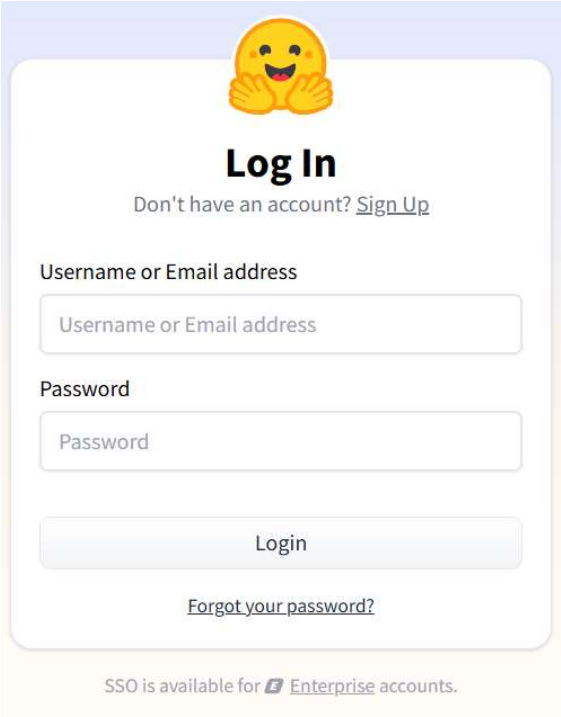
→ Hugging Face Hub

- ▶ 120k 이상의 모델, 20k의 데이터셋, 50k의 데모 앱(Spaces)을 포함하는 플랫폼
- ▶ 모든 것은 오픈 소스이며 공개적으로 이용할 수 있다

→ Huggingface

- ▶ 회원가입 (<https://huggingface.co>)
- ▶ 토큰 발급 (<https://huggingface.co/docs/hub/security-tokens>)
- ▶ LLM 의 READ 키를 복사

LLM 키: <https://huggingface.co/settings/tokens>



The image shows the Huggingface login interface. At the top is a yellow smiley face emoji with its hands clasped. Below it is the text 'Log In' in bold. Underneath is a link: 'Don't have an account? [Sign Up](#)'. There are two input fields: 'Username or Email address' and 'Password'. Below these is a 'Login' button. At the bottom of the form is a link: '[Forgot your password?](#)'. At the very bottom of the page, it says 'SSO is available for [Enterprise](#) accounts.'

● Anthropic API 𐄂

→ <https://console.anthropic.com/login?returnTo=%2F%3F>

● Google AI Studio API 키

→ <https://aistudio.google.com/app/apikey>

ChatGPT 모델 이해와 활용

● ChatGPT 크롬 확장 프로그램

→ 확장 프로그램 : ChatGPT 기능 확장

- ▶ [프롬프트 지니](#) : ChatGPT 상에서 자동 한영/영한 번역
- ▶ [ShareGPT](#) : 검색자료 공유하기 예: <https://chatgpt.com/share/670f23c0-8f30-8012-bfbf-6d94879d4e9b>
- ▶ [WebChatGPT](#) : 최신 정보로 결과 받기
- ▶ [AIPRM for ChatGPT](#) : 전문가가 만든 프롬프트를 제공
- ▶ [Sider](#) : 웹 문서 내용을 바로 요약 (블로그)

→ 확장 프로그램: GPT 기반 생산성 툴

- ▶ [ChatPDF](#) : PDF를 학습해서 PDF 문서 기반으로 요약, 질문, 콘텐츠 생성
- ▶ [Eightify](#) : 유튜브 비디오 내용을 바로 요약 (유튜브)

→ 네이버 영화 리뷰, 텍스트 파일로 제공

▶ <https://github.com/e9t/nsmc>

→ NNST - 한국어 네이버 뉴스 데이터셋

▶ <https://github.com/jason9693/NNST-Naver-News-for-Standard-and-Technology-Database>

▶ https://github.com/cranberryai/todak_todak_python/commit/6c1c31a8f9bc625c2abde593a3b6880719c0f72c

자연어 처리 기초

● 자연어처리(NLP) 기술의 발전

→ 자연어 처리(NLP)의 응용 분야와 성공 사례

응용 분야	설명
문서 처리 및 정보 검색	검색 엔진, 문서를 자동으로 요약, 문서의 주제, 카테고리를 분류 Google Search SummarizeBot : 뉴스, 연구 논문 등을 자동 요약하여 정보 과부하 문제 해결
고객 서비스	실시간 자동 응답 챗봇, 감정 분석(고객 피드백 분석을 통해 긍정/부정 감정 이해) ChatGPT : 대화형 AI 모델로, 고객 지원, 학습 보조, 코드 디버깅 등 다양한 작업 지원 Zendesk : NLP를 활용하여 고객 서비스 요청을 자동으로 분류하고 라우팅
번역 및 언어 처리	한 언어에서 다른 언어로 텍스트 변환(기계 번역) 음성을 텍스트로 변환하고 다국어 지원(언어 인식 및 변환) Google Translate : NLP와 딥러닝 기술을 사용한 다국어 번역 서비스 DeepL : 문맥을 더 잘 이해하는 고품질 번역
음성 인식 및 합성	음성 텍스트 변환 및 텍스트 음성 변환(TTS) Amazon Alexa, Apple Siri : 음성 기반 가상 비서 Otter.ai : 회의 녹음 내용을 실시간 텍스트로 변환하여 저장

자연어 처리 기초

● 자연어처리(NLP) 기술의 발전

→ 자연어 처리(NLP)의 응용 분야와 성공 사례

응용 분야	설명
의료 및 생명 과학	의료 기록 및 보고서에서 중요한 정보 추출(임상 문서 분석) 증상 기술을 분석하여 적절한 진단 제안(질병 진단 지원) IBM Watson Health : NLP를 통해 방대한 의료 데이터를 분석하여 의사에게 인사이트 제공 Mayo Clinic : 의료 기록 분석을 통해 환자 맞춤형 치료법 추천
금융 및 비즈니스	금융 뉴스 분석으로 시장 트렌드 예측 계약서 및 문서 자동 처리 Bloomberg Terminal : NLP로 금융 뉴스를 실시간 분석하여 투자 의사결정 지원 JP Morgan's COiN : 계약서 검토를 자동화하여 업무 효율성 향상
교육 및 학습	텍스트 작성 중 문법 및 스타일 오류 감지하고 교정 개인화된 학습 도우미 Grammarly : 텍스트의 문법, 스타일, 어조를 분석하고 제안 Duolingo : 다국어 학습에 NLP를 적용하여 개인 맞춤형 학습 제공

Gemma

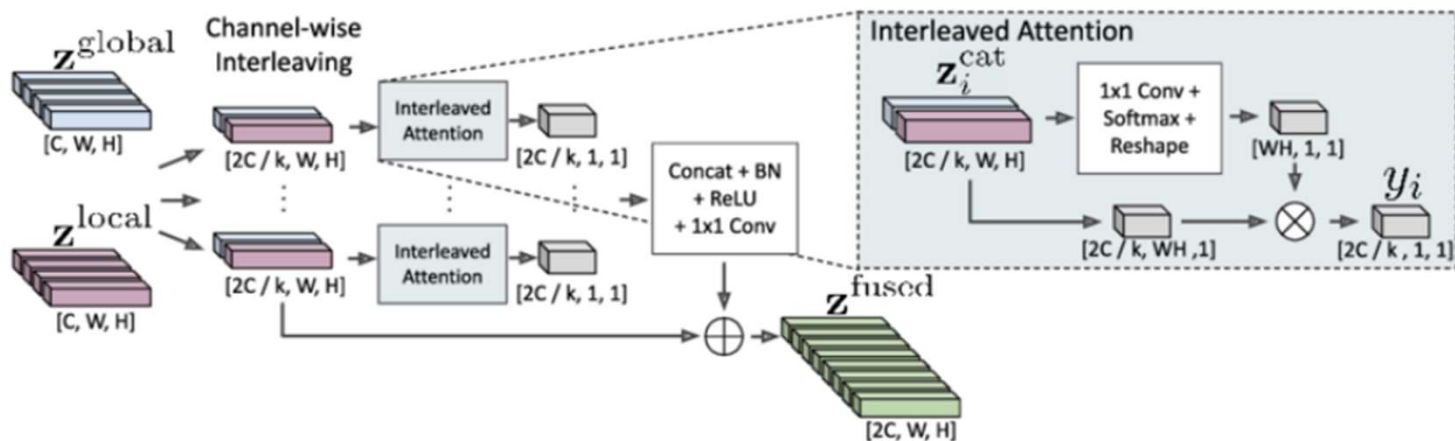
○ Gemma Model

- 구글 딥마인드
- 젤마 2 2b : 20억 개의 파라미터를 갖춘 모델
- 로컬-글로벌 어텐션 교차와 그룹 쿼리 어텐션 도입
- 벤치마크에서 GPT-3.5와 Mixtral 8x7B를 모두 능가 (LMSYS 챗봇 아레나에서 1130점을 획득)
- 2조 개의 토큰으로 구성된 방대한 데이터 세트를 학습
- 90억(9B) 및 270억(27B) 매개변수 변형과 같은 Gemma 제품군의 다른 모델과 비교했을 때 Gemma 2 2b는 크기와 효율성 간의 균형이 돋보입니다.
- 크기가 작아 성능 저하 없이 다양한 소비자급 디바이스에 배포하기에 적합하며, 스마트폰과 기타 휴대용 기기에 새로운 가능성을 열어줍니다

Gemma

● Gemma Model

- 로컬-글로벌 어텐션 교차 (Interleaving Local-Global Attentions)는 텍스트, 음성, 비디오 등 긴 시퀀스의 데이터를 처리할 때, 근거리와 원거리 종속성을 모두 포착하기 위해 로컬 어텐션과 글로벌 어텐션을 번갈아가며 적용하는 방식
 - ▶ 효율성을 높이기 위한 전략으로 사용됨
- Local Attention : 각 토큰이 자신의 주변 일정 범위 내의 토큰들에만 주목합니다
 - ▶ 계산 효율성이 높고 근접 문맥을 잘 포착합니다
- Global Attentions : 각 토큰이 전체 시퀀스의 모든 토큰에 주목합니다.
 - ▶ 장거리 의존성을 포착하는데 유용하지만 계산 비용이 높습니다.



Gemma

● Gemma Model

→ 로컬-글로벌 어텐션 교차 (Interleaving Local-Global Attentions) 동작 방식

1. 입력 특성 맵 분리 : 입력 데이터가 두 부분으로 나뉩니다.

z_{global} 은 글로벌 컨텍스트 정보를, z_{local} 은 로컬 컨텍스트 정보를 나타냅니다.

Gemma

● Groq

- Groq API는 대형 언어 모델 전용 추론 가속 장치인 Groq LPU를 기반으로 언어 모델을 사용하여 자연어 처리 및 다양한 언어 관련 작업을 수행하는 데 사용되는 API
- Meta AI의 Llama-2 70B 모델을 이용해 사용자당 초당 300 토큰을 처리
- Language Processing Unit (LPU)
- <https://console.groq.com/keys>

Q & A