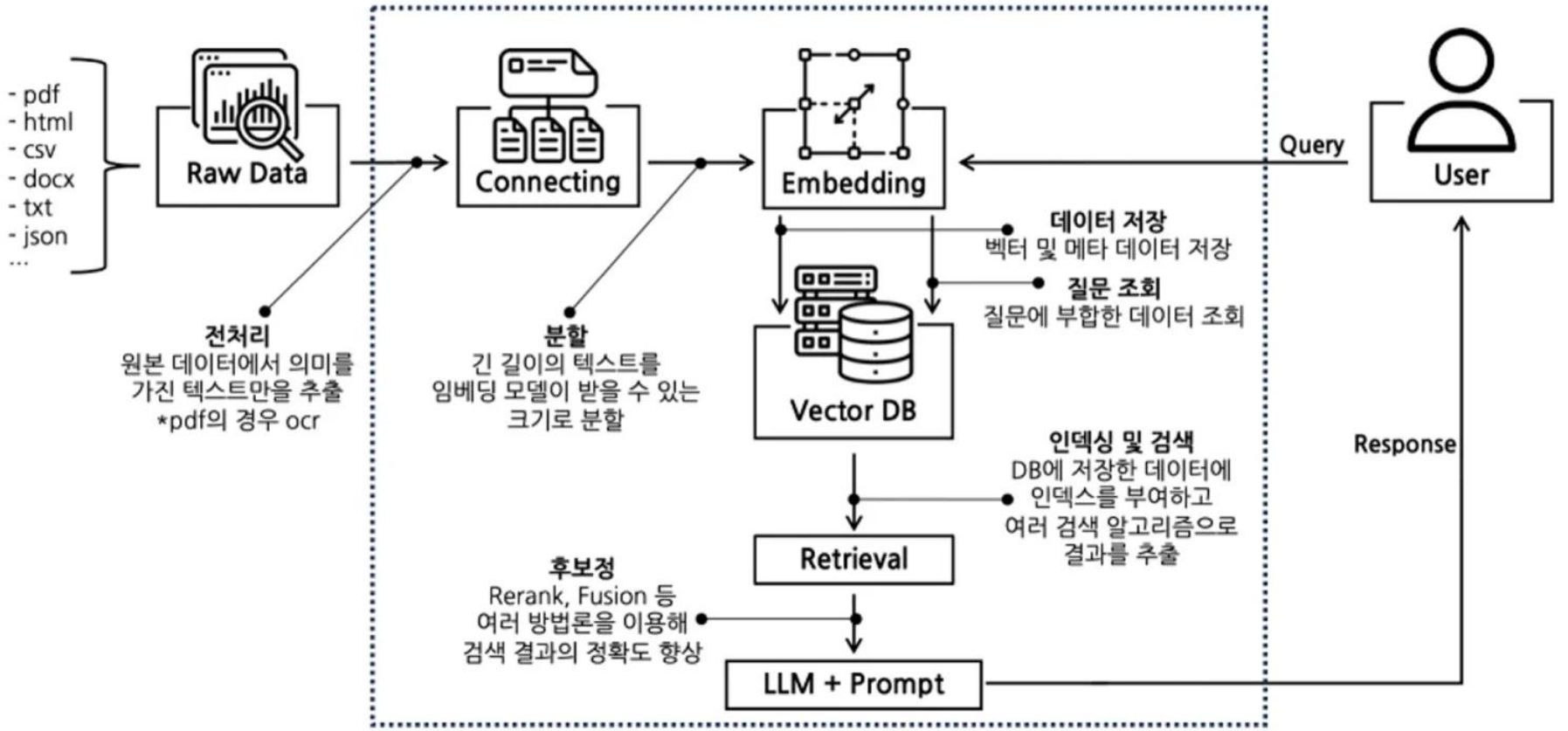
계획3

목차

- 진행 상황
- What shall we aim to?
- How to do?
- approachs

RAG 프로세스


- pdf
- html
- csv
- docx
- txt
- json
...
Raw Data
Connecting
Embedding
Query
User
데이터 저장
벡터 및 메타 데이터 저장
질문 조회
질문에 부합한 데이터 조회
Vector DB
전처리
원본 데이터에서 의미를
가진 텍스트만을 추출
*pdf의 경우 ocr
분할
긴 길이의 텍스트를
임베딩 모델이 받을 수 있는
크기로 분할
인덱싱 및 검색
DB에 저장한 데이터에
인덱스를 부여하고
여러 검색 알고리즘으로
결과를 추출
Response
Retrieval
후보정
Rerank, Fusion 등
여러 방법론을 이용해
검색 결과의 정확도 향상
LLM + Prompt

RAG 프로세스

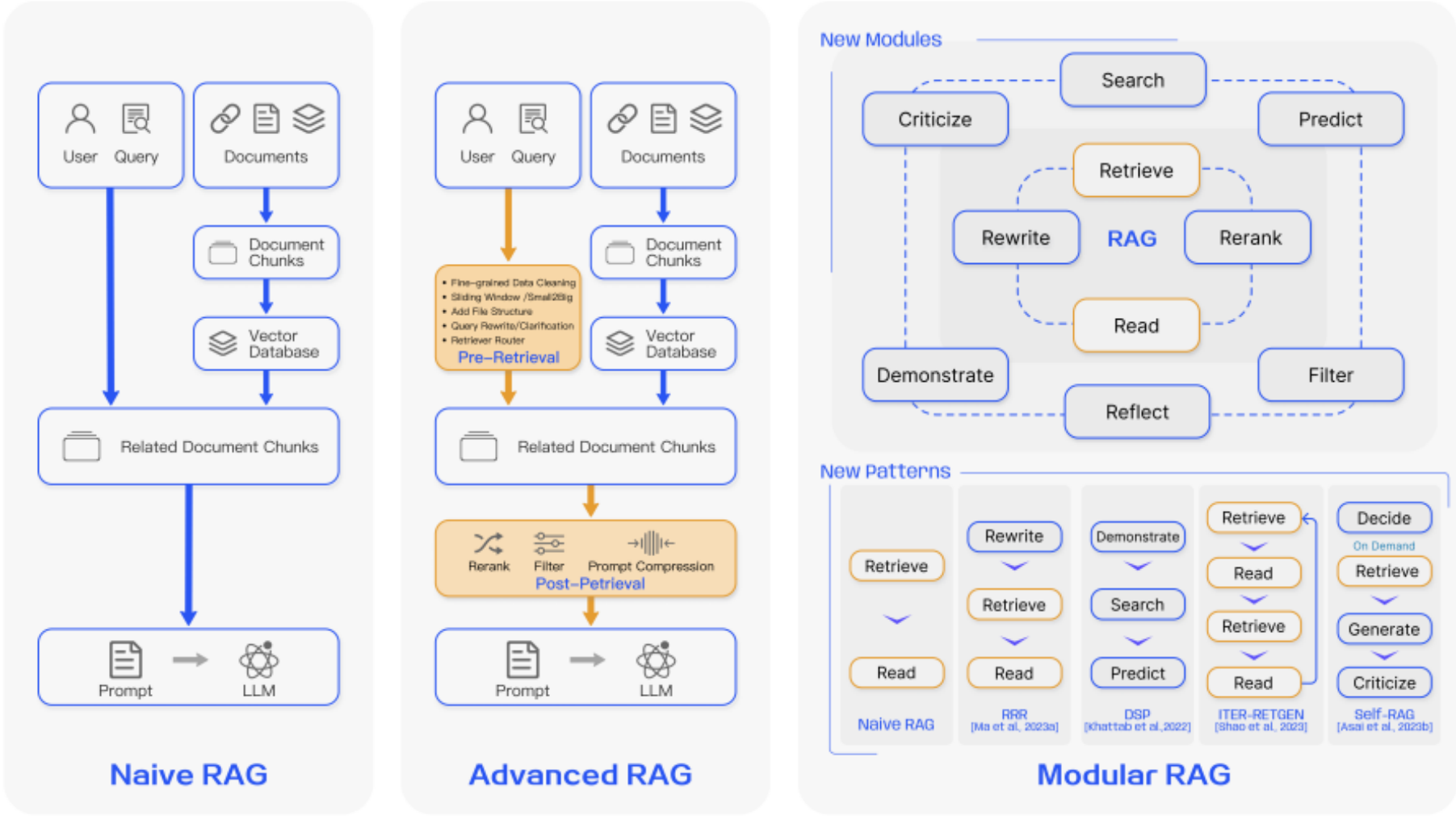
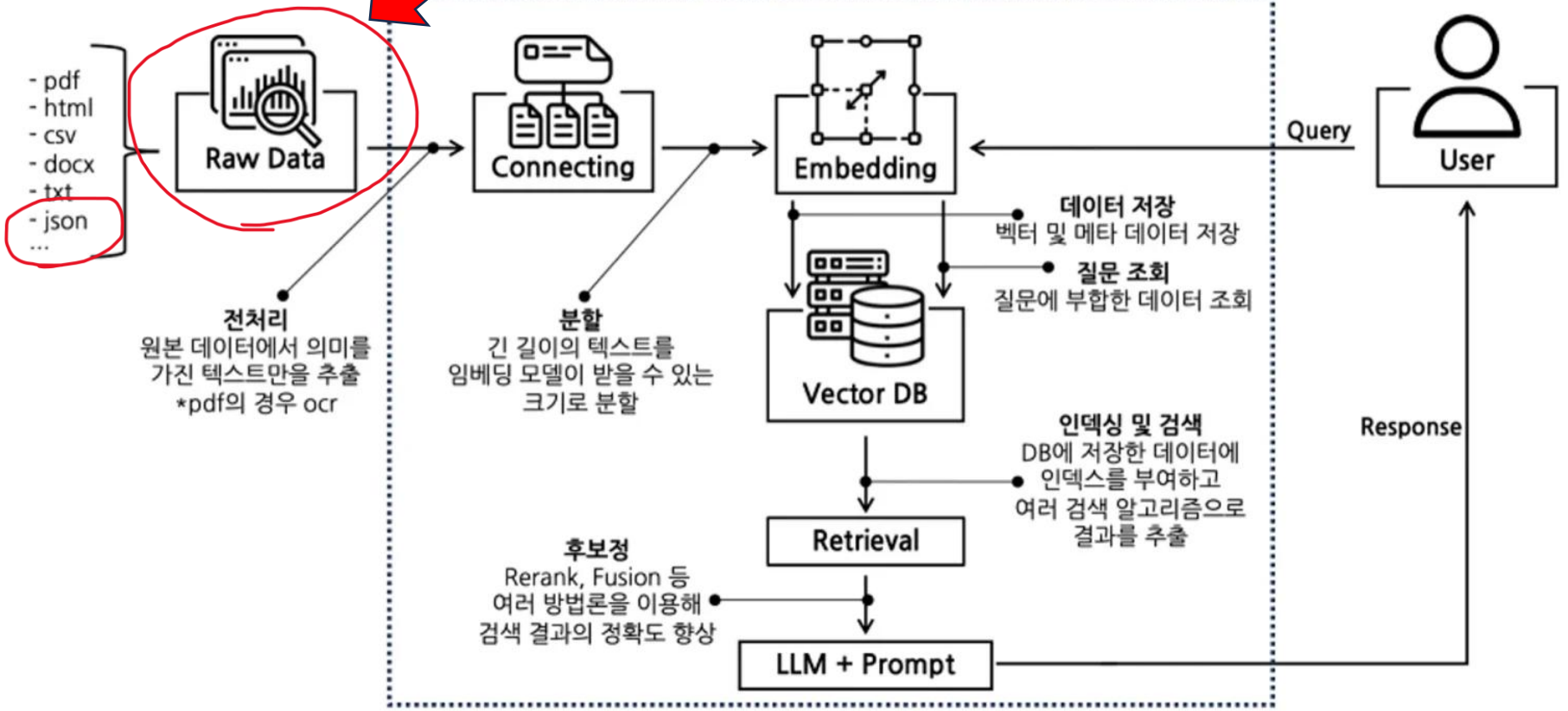


Figure 3: Comparison between the three paradigms of RAG

현재 진행 상황

현재 진행 상황



What shall we aim to?

- **1. 개요**
- **목표**: 사용자의 반려동물 관련 질문에 대해 협력 기업의 제품 데이터를 기반으로 정확하고 관련성 높은 솔루션을 제공.
- **기술**: 검색 증강 생성 (Retrieval-Augmented Generation (RAG))를 사용하여 LLM(대형 언어 모델)의 응답을 최적화.
- **핵심 질문**: 어떻게 RAG를 활용해 협력 기업 데이터를 효과적으로 통합하고, 사용자 경험(UX)을 향상시킬 것인가?

How to do?(1/3)

- 사용자 쿼리 입력: 예: “강아지 피부 가려움증에 좋은 샴푸 추천 해줘.”
- 쿼리 임베딩: 쿼리를 임베딩 벡터화
- 데이터 검색: 벡터 데이터베이스에서 쿼리와 유사한 협력 기업의 제품 정보 검색(유사도 계산, 리랭커링)
- 프롬프트 강화: 검색된 제품 정보(샴푸 성분, 리뷰 등)를 쿼리 추가
- 응답 생성: LLM이 강화된 프롬프트를 바탕으로 사용자 맞춤 솔루션 제공.
- 예: “협력사 A의 저자극 샴푸는 오트밀 성분으로 가려움증 완화에 효과적이며, 95%의 사용자가 만족도를 표시했습니다.”

How to do?(2/3)

- 고객 지원: 웹 사이트 챗봇이 협력사 제품 추천
- 마케팅: 사용자 질문 기반 맞춤형 제품 제안
- 내부 직원 지원: 제품 정보에 대해 빠른 검색 및 제공

How to do?(3/3)

- 데이터 품질:
  - 협력 기업의 데이터 정확성 및 최신성 유지
  - 민감 정보(고객 데이터 등) 보호를 위한 접근 제어
- 성능 최적화:
  - 검색 속도 향상을 위한 효율적인 벡터 인덱싱
  - LLM 응답 시간 단축을 위한 캐싱 메커니즘
- 확장성
  - 새로운 협력사 데이터 통합 용이성.
  - 다국어 지원 가능성(글로벌 협력사 확장 시)

Approachs

JSON 데이터를 자연어처리에 적합한 형태로 변환하는 프로세스 필요 → JSON LOADER 사용

```python
from langchain_community.document_loaders import JSONLoader

# JSONLoader 설정
loader = JSONLoader(
    file_path="products.json",
    jq_schema=".products[] | {name: .name, description: .description}",  # jq
    text_content=True
)

# 데이터 로드
documents = loader.load()

# 결과: Document 객체 리스트 (예: name과 description이 자연어로 변환됨)
for doc in documents:
    print(doc.page_content)  # "name: Product A, description: This is a pet s
```

```python
def metadata_extractor(record: dict, metadata: dict) -> dict:
    """
    JSON 레코드에서 '원문_요약_GPT_생성' 필드를 제외한 모든 필드를 Document의 메타데이터로 추출.

    """

    for key, value in record.items():
        if key != "원문_요약_GPT_생성":
            metadata[key] = value
    return metadata


loader = JSONLoader(
    file_path=file_path,
    jq_schema=".",  # JSON 파일의 루트(최상위 객체)를 로드
    content_key="원문_요약_GPT_생성",  # 이 필드의 내용이 Document의 Page_content가 됨.
    metadata_func=metadata_extractor,  # 나머지 필드를 메타데이터로
)


data = loader.load()

print(type(data))
```

Approchs

- 처리한 텍스트 데이터들에 대해 임베딩 벡터를 계산하기에 앞서 텍스트의 분할과 같은 청킹이 필요함.
- CharacterTextSplitter → 주어진 텍스트를 문자or문자열 단위로 분할
- RecursiveCharacterTextSplitter → 긴 텍스트를 작은 청크(chunk)로 나누되, 문맥과 의미가 파괴되지 않도록 관련 있는 내용을 함께 유지, 단순히 고정된 길이로 자르는 대신, 문장이나 문단의 자연스러운 경계를 존중해 의미 단위를 보존.

Approachs

- TokenTextSplitter → 토큰 단위로 분할, LLM처리에 적합. 토큰 기반 검색 최적화
- SemanticChunker → 텍스트를 의미적 유사성을 기준으로 분할
- RecuresiveJSONSplitter → JSON 데이터를 재귀적으로 탐색하여 계층적 구조를 유지하며 텍스트로 분할, JSON의 key-value 쌍 , 배열, 중첩 객체를 자연어로 변환 , JSON 특화, 의미 보존, 유연성의 부분에서 유리함

Approachs

- RecursiveJSONSplittter → JSON특화, 의미 보존, 유연성이 좋음

```
from langchain_text_splitters import RecursiveJsonSplitter
import json

# JSON 데이터
json_data = {
    "products": [
        {"id": 1, "name": "Pet Shampoo", "description": "Hypoallergenic shampoo for dogs."},
        {"id": 2, "name": "Dog Food", "description": "Organic grain-free food."}
    ]
}

# RecursiveJSONSplitter 초기화
splitter = RecursiveJsonSplitter(max_chunk_size=100)

# JSON 분할
chunks = splitter.split_json(json_data=json_data)
print(chunks)
# 출력 예: [{"id": 1, "name": "Pet Shampoo", "description": "Hypoallergenic shampoo for dogs."}, ...]
```

Approachs

- SemanticChunker → JSON 데이터를 먼저 자연어로 변환(JSON Loader로 “Pet Shampoo is hypoallergenic” 한 후, 의미적 문맥을 더 정교하게 분할하고 싶을 때
- 제품 설명이나 리뷰처럼 텍스트가 길고 의미적 연관성이 중요한 경우

```
from langchain_experimental.text_splitter import SemanticChunker
from langchain_huggingface import HuggingFaceEmbeddings

# 임베딩 모델
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")

# SemanticChunker 초기화
splitter = SemanticChunker(embeddings, breakpoint_threshold_type="percentile")

# 자연어로 변환된 텍스트
text = "Pet Shampoo is hypoallergenic. It soothes skin. Dog Food is organic."
chunks = splitter.split_text(text)
print(chunks)
```

Text Splitter Comparison Table

Splitter	분할 기준	장점	단점	주요 사용 사례	JSON 데이터 적합성
TokenTextSplitter	토큰 수 (토크나이저 기반)	LLM 토큰 제한 준수, 균일한 청크 크기	의미적 문맥 손실 가능, 토크나이저 의존성	LLM 입력 관리, 벡터 저장소 검색 최적화	중간 (토큰 기반 처리 가능, 의미 보존 제한적)
SemanticChunker	의미적 유사성 (임베딩 기반)	의미적 문맥 유지, 문맥 손실 최소화	계산 비용 높음, 청크 크기 예측 어려움	RAG, 문맥 중요한 텍스트 (제품 설명, 리뷰)	높음 (의미적 분할로 제품 데이터 문맥 유지)
CodeSplitter	코드 구문 구조 (언어별)	코드 논리적 단위 유지, 문맥 손실 방지	코드 외 텍스트 부적합, 지원 언어 제한적	코드 문서, 주석 처리	낮음 (JSON은 코드 아님)
MarkdownSplitter	Markdown 구조 (헤더, 리스트 등)	Markdown 구조적 분할, 문서 탐색 용이	Markdown 외 형식 부적합, 비정형 텍스트 처리 어려움	기술 문서, README, 블로그	낮음 (JSON은 Markdown 아님)
HTMLHeaderTextSplitter	HTML 헤더 및 구조	웹 콘텐츠 구조적 분할, 헤더 기반 탐색 용이	HTML 외 형식 부적합, 복잡한 HTML 처리 어려움	웹 페이지, 온라인 문서	낮음 (JSON은 HTML 아님)
RecursiveJSONSplitter	JSON 계층 구조	JSON 데이터에 특화, 계층적 구조 유지	비 JSON 데이터 부적합, 복잡한 JSON의 청크 크기 조정 필요	JSON 기반 데이터 (API 응답, 제품 데이터)	매우 높음 (JSON 구조 직접 처리)

Approachs

- 벡터화 단계

◆Huggingface embedding(로컬 모델 임베딩)→ 무료, 오픈소스

◆OpenAIEmbeddings(오픈 AI 임베딩)→ 무난한 모델, 다국어 지원에 뛰어남

◆CacheBackedEmbeddings(캐시 임베딩)

◆OllamaEmbeddings(올라마 임베딩)

◆UpstageEmbeddings(업스테이지 임베딩) → 한국어 처리 성능이 뛰어남

Approachs

- ◆PineCone → 클라우드 기반의 관리형 벡터 데이터 베이스, 랭체인 통합
- ◆Chroma → 오픈소스, 로컬/ 임베디드 벡터 저장소, 랭체인 통합
- ◆FAISS → Facebook의 오픈소스 벡터 검색 라이브러리 → 랭체인 통합, 다양한 인덱싱 알고리즘

Vector Store Comparison Table

Vector Store	배포 방식	장점	단점	주요 사용 사례	JSON 데이터 적합성
Pinecone	클라우드 (서버리스)	빠른 검색, 높은 확장성, 쉬운 관리	비용 발생, 인터넷 의존	대규모 RAG, 실시간 검색	매우 높음 (확장성, 성능 우선)
Chroma	로컬/임베디드	무료, 설정 간단, 오프라인 가능	대규모 데이터 성능 저하, 확장성 제한	프로토타입, 소규모 데이터	높음 (비용 절감, 초기 테스트)
Weaviate	온프레미스/클라우드	벡터+텍스트 검색, 고급 필터링	설정 복잡, 관리 부담	복잡한 검색, 메타데이터 활용	중간 (메타데이터 활용 시 유리)
FAISS	로컬 (메모리 기반)	무료, 빠른 검색, 유연한 인덱스	영구 저장소 없음, 확장 어려움	연구, 프로토타입, 오프라인	중간 (소규모, 오프라인 적합)
Elasticsearch	온프레미스/클라우드	하이브리드 검색, 엔터프라이즈 기능	설정 복잡, 벡터 검색 성능 제한	기존 Elasticsearch 환경	낮음 (벡터 검색 특화 부족)