

Day 2 – Advanced RAG Systems

Naive → Advanced → Modular

고급 RAG(검색-보강 생성) 시스템 구축을 위한 실무 중심 가이드입니다. 기초적인 Naive RAG부터 시작하여 고급 기술을 거쳐 모듈러 아키텍처까지, 실제 프로덕션 환경에서 사용 가능한 RAG 시스템을 단계별로 구축하는 방법을 학습합니다.

Day 2 학습 목표

핵심 목표

이 과정을 통해 다음을 달성할 수 있습니다:

- Naive RAG의 한계를 이해하고 실패 패턴 분석
- 쿼리 개선, 메타데이터 필터링, 하이브리드 검색 등 고급 기법 적용
- 모듈러 RAG 아키텍처 설계 및 구현
- 실무에서 바로 활용 가능한 성능 평가 체계 구축

Day 2 커리큘럼 아젠다

모듈	주제	핵심 내용
00	프리실습 (Prerequisites)	RAG 체험, 데이터 탐색, 기본 구현
01	Naive RAG Baseline	기본 RAG 파이프라인 구축
02	Failure Analysis	Naive RAG 실패 패턴 분석
03	Query Refinement	쿼리 개선 및 확장 기법
04	Metadata Filtering	메타데이터 기반 필터링
05	Hybrid Search & Reranking	BM25 + 벡터 검색, 재순위화
06	Modular RAG Routing	쿼리 라우팅, Self-RAG, Refine

RAG 시스템 첫 체험

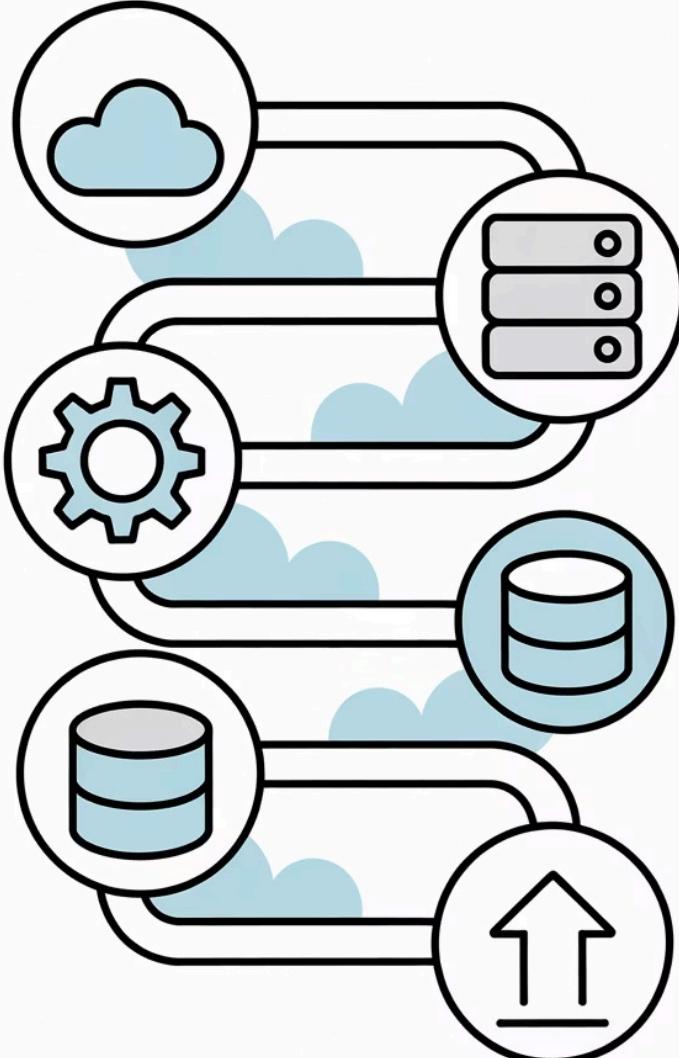
실습 목적

가장 간단한 형태의 RAG 시스템을 실행하여 "검색 → 컨텍스트 구성 → LLM 응답 생성"의 기본 흐름을 체험합니다. 복잡한 설정 없이 즉시 결과를 확인할 수 있습니다.

주요 코드

```
def simple_rag_demo():
    # 샘플 문서 생성
    docs = ["항공사 규정...", "여행 정보..."]
    # 간단한 검색
    retrieved = search(query, docs)
    # LLM으로 답변 생성
    answer = llm.generate(query, retrieved)
    return answer
```

기본 RAG 파이프라인 구축



01

문서 준비 (`create_sample_documents`)

도메인별로 샘플 문서를 생성합니다. 각 도메인의 특성을 반영한 한국어 문서로 구성되며, 실제 업무 시나리오를 시뮬레이션합니다.

02

텍스트 청킹 (`preprocess_documents`)

`chunk_size=500, overlap=50` 설정으로 문서를 작은 단위로 분할합니다. 적절한 청크 크기는 검색 정확도와 컨텍스트 품질에 직접적인 영향을 미칩니다.

03

벡터스토어 구축 (`setup_vector_store`)

BAAI/bge-m3 임베딩 모델과 FAISS L2 인덱스를 사용하여 벡터 데이터베이스를 구축합니다. 한국어 문서에 최적화된 임베딩 모델입니다.

04

LLM 연결 (`Day1FinetunedLLM`)

Day 1에서 파인튜닝한 `ryanu/my-exaone-raft-model`을 연결합니다. RAFT 학습을 통해 검색 기반 응답 생성 능력이 향상된 모델입니다.

고급 기능 미리보기

쿼리 개선 (Query Refinement)

사용자의 모호한 질문을 여러 개의 명확한 하위 질문으로 분해하거나, 동의어와 관련 용어를 추가하여 검색 정확도를 높입니다.

```
def generate_multi_queries(query):
    # 원래 질문을 3-5개의
    # 다양한 표현으로 확장
    variations = llm.expand(query)
    return variations
```

하이브리드 검색 (Hybrid Search)

키워드 기반 BM25 검색과 의미 기반 벡터 검색을 결합하여 더 포괄적인 결과를 얻습니다. Reciprocal Rank Fusion으로 최종 순위를 결정합니다.

```
def hybrid_search(query):
    bm25_results = bm25(query)
    vector_results = vector(query)
    # 두 결과를 융합
    merged = rrf_fusion(
        bm25_results,
        vector_results
    )
    return merged
```

RAG 성능 평가 소개

응답 시간, 컨텍스트 품질, 답변 정확도 등을 자동으로 기록하고 시각화합니다.

응답 시간 (Response Time)

검색부터 응답 생성까지 전체 프로세스의 소요 시간을 밀리초 단위로 측정합니다. 프로덕션 환경에서 사용자 경험에 직접적인 영향을 미치는 지표입니다.

키워드 커버리지 (Keyword Coverage)

검색된 컨텍스트가 쿼리의 핵심 키워드를 얼마나 포함하고 있는지 측정합니다. 높을수록 관련성이 높은 문서가 검색되었음을 의미합니다.

컨텍스트 길이 (Context Length)

LLM에 전달되는 컨텍스트의 토큰 수를 추적합니다. 너무 짧으면 정보 부족, 너무 길면 비용과 처리 시간 증가로 이어집니다.

데이터 분할 및 설정

허깅페이스 데이터셋을 효율적으로 활용하기 위한 설정입니다. 전체 데이터셋 대신 일부를 추출하여 빠른 실험이 가능합니다.

```
CURRENT_DATASET_KEY = "squad_kor_v1"
```

```
HF_TRAIN_SPLIT = "train[:200]"
```

```
HF_VAL_SPLIT = "train[200:260]"
```

```
HF_COLUMN_MAP = {  
    "context": "context",  
    "question": "question",  
    "answers": "answers"  
}
```

Naive RAG 실패 분석 (1/2)

멀티홉 시나리오와 5가지 실패 패턴

실제 프로덕션 환경에서 Naive RAG가 직면하는 주요 문제들을 체계적으로 분석합니다. Pastel 데이터셋의 멀티홉(multi-hop) 질문들은 여러 문서에서 정보를 조합해야 답할 수 있는 복잡한 시나리오를 포함합니다.

1. Semantic Gap (의미적 간격)

쿼리와 문서의 표현 방식이 달라 벡터 검색이 실패하는 경우. 예: "비용"과 "요금"처럼 동의어를 다루지 못함.

2. Context Fragmentation (컨텍스트 단편화)

필요한 정보가 여러 청크에 분산되어 있어 하나의 컨텍스트로 충분한 답을 만들지 못하는 경우.

3. Ambiguous Queries (모호한 쿼리)

질문이 불명확하거나 여러 해석이 가능해 잘못된 문서를 검색하는 경우.

4. Outdated Information (오래된 정보)

시간에 민감한 정보를 다룰 때 최신 문서를 우선하지 못해 부정확한 답변을 제공하는 경우.

5. Low Ranking (낮은 순위)

관련 문서가 검색되었지만 순위가 낮아 컨텍스트에 포함되지 않는 경우.

- ▣ **개선 방향:** 각 실패 패턴에 대응하는 해결책이 이어지는 모듈들에서 단계적으로 소개됩니다. Semantic Gap → Query Refinement, Fragmentation → Hybrid Search, Low Ranking → Reranking 등으로 연결됩니다.

고급 쿼리 개선 기법 (1/2)

4가지 쿼리 개선 파이프라인 비교

기법	동작 방식	장점	적합한 상황
Multi-Query	원래 쿼리를 여러 개의 다양한 표현으로 확장하여 검색	표현 다양성 확보, 검색 누락 최소화	동의어가 많은 도메인
Sub-Question	복잡한 질문을 여러 개의 하위 질문으로 분해	멀티홀 추론 가능, 단계적 정보 수집	복합적인 질문
Query Expansion	쿼리에 관련 용어와 컨텍스트를 추가	의미적 범위 확장, 관련 문서 증가	전문 용어가 많은 도메인
RAG-Fusion	여러 검색 전략의 결과를 융합하여 최종 순위 결정	다양한 관점 통합, 검색 강건성 향상	복잡한 정보 요구

고급 쿼리 개선 기법 (2/2)

구현 코드 및 성능 개선 효과

Multi-Query 생성

```
def generate_multi_queries(query, llm):
```

```
    """
```

```
    원래 질문을 3-5개의 다양한  
    표현으로 확장
```

```
    """
```

```
    prompt = f"""
```

```
    다음 질문을 다양한 방식으로  
    3가지 표현해주세요:
```

```
    질문: {query}
```

```
    표현1:
```

```
    표현2:
```

```
    표현3:
```

```
    """
```

```
responses = llm.generate(prompt)  
queries = parse_queries(responses)
```

```
# 모든 쿼리로 검색 수행
```

```
all_results = []
```

```
for q in queries:
```

```
    results = vector_search(q)
```

```
    all_results.extend(results)
```

```
return deduplicate(all_results)
```

메타데이터 필터링

스마트 필터링으로 정밀도 향상

메타데이터를 활용하여 검색 결과를 효과적으로 필터링합니다. 단순히 벡터 유사도만으로 문서를 선택하는 것이 아니라, 시간, 카테고리, 출처 등의 메타데이터를 함께 고려하여 더욱 관련성 높은 결과를 제공합니다.

passes_filter() 조건 로직

```
def passes_filter(doc, filters):
    """
    문서가 지정된 필터 조건을
    통과하는지 검사
    """

    # 시간 필터
    if 'date_range' in filters:
        start, end = filters['date_range']
        if not (start <= doc.date <= end):
            return False

    # 카테고리 필터
    if 'categories' in filters:
        if doc.category not in filters['categories']:
            return False

    # 출처 필터
    if 'sources' in filters:
        if doc.source not in filters['sources']:
            return False

    # 신뢰도 임계값
    if 'min_confidence' in filters:
        if doc.confidence < filters['min_confidence']:
            return False

    return True
```

하이브리드 검색 (1/2)

BM25 + 벡터 검색 + 재순위화 파이프라인

01

BM25 키워드 검색

전통적인 정보 검색 알고리즘으로 쿼리 키워드와 문서 간의 정확한 매칭을 수행합니다. 고유 명사나 전문 용어에 강점이 있습니다.

02

벡터 의미 검색

임베딩 모델로 쿼리와 문서의 의미적 유사도를 계산합니다. 동의어나 유사 표현을 잘 포착합니다.

03

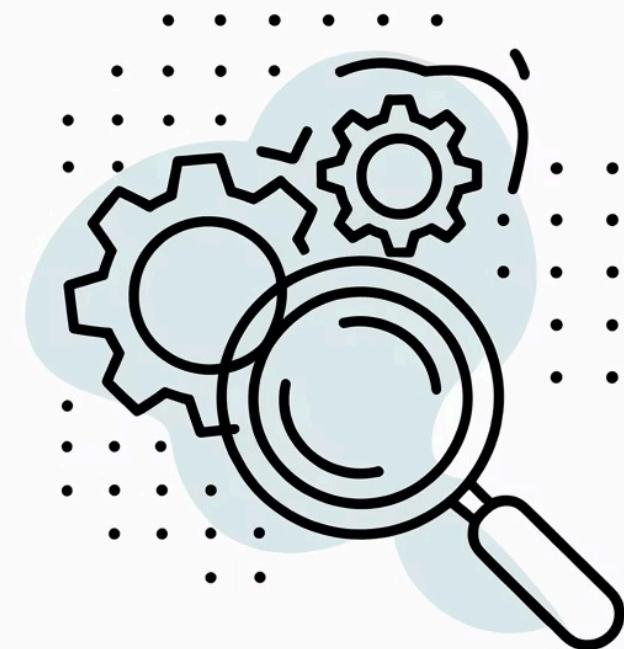
Reciprocal Rank Fusion (RRF)

두 검색 결과를 융합하여 최종 순위를 결정합니다. 각 검색 방법의 장점을 살리면서 단점을 보완합니다.

04

Cross-Encoder 재순위화

쿼리와 문서를 함께 입력받아 관련성을 정밀하게 재평가합니다. 최종 상위 결과의 품질을 크게 향상시킵니다.



하이브리드 검색 (2/2)

코드 구현 및 성능 비교

하이브리드 검색 파이프라인

```
def simple_bm25_search(query, corpus, k=5):
    """BM25 키워드 검색"""
    bm25 = BM25Okapi(corpus)
    scores = bm25.get_scores(query.split())
    top_k = np.argsort(scores)[-k:]
    return top_k

def simple_vector_search(query, embeddings, k=5):
    """벡터 의미 검색"""
    query_vec = embed(query)
    similarities = cosine_similarity(
        query_vec, embeddings
    )
    top_k = np.argsort(similarities)[-k:]
    return top_k

def get_rag_answer(query, corpus, embeddings):
    """하이브리드 RAG 파이프라인"""
    # 1. 두 가지 검색 수행
    bm25_results = simple_bm25_search(
        query, corpus
    )
    vector_results = simple_vector_search(
        query, embeddings
    )

    # 2. RRF로 융합
    fused = reciprocal_rank_fusion(
        bm25_results, vector_results
    )

    # 3. Cross-encoder로 재순위화
    reranked = cross_encoder_rerank(
        query, fused
    )

    # 4. LLM으로 답변 생성
    context = get_top_docs(reranked)
    answer = llm.generate(query, context)
    return answer
```

모듈러 RAG 라우팅 (1/2)

지능형 쿼리 라우팅 시스템

모든 질문에 동일한 파이프라인을 적용하는 대신, 쿼리의 유형과 복잡도에 따라 최적의 처리 경로를 선택하는 지능형 시스템입니다. 간단한 사실 질문은 빠른 검색으로, 복잡한 추론 질문은 멀티홉 파이프라인으로 라우팅합니다.

Query Router

키워드 기반: 질문에 "언제", "얼마" 등의 키워드가 있으면 사실 질문으로 분류

LLM 기반: 모델이 질문의 의도와 복잡도를 분석하여 최적 파이프라인 선택

Self-RAG Critic

STRICT_PROMPT: 생성된 답변이 검색된 컨텍스트에 충분히 근거하고 있는지 자체 검증

신뢰도 평가: 답변의 확실성을 점수화하여 불확실한 경우 추가 검색 트리거

Retry with Refine

REFINE_PROMPT: Self-RAG가 답변을 거부하면 쿼리를 자동으로 개선하여 재시도

점진적 개선: 최대 3회까지 쿼리를 정교화하며 답변 품질 향상

모듈러 RAG 라우팅 (2/2)

라우팅 전략 비교 및 Self-RAG 평가

하드코딩 vs LLM 라우팅

특성	하드코딩	LLM 기반
속도	매우 빠름	약간 느림
정확도	70-80%	90-95%
유지보수	규칙 추가 필요	자동 적응
비용	없음	API 호출
적합한 경우	명확한 패턴	복잡한 의도

```
def hardcoded_routing(query):
    if any(kw in query for kw in
          ['언제', '날짜', '시간']):
        return 'temporal_pipeline'
    elif any(kw in query for kw in
            ['왜', '이유', '어떻게']):
        return 'reasoning_pipeline'
    else:
        return 'simple_pipeline'
```

```
def llm_routing(query):
    prompt = f"""
질문 유형을 분류하세요:
- simple: 단순 사실 질문
- reasoning: 추론 필요
- multihop: 여러 정보 조합
    """
    return llm.generate(prompt)
```

```
질문: {query}
유형:
#####
return llm.generate(prompt)
```

실무 적용 로드맵

단계별 RAG 시스템 구축 전략

단계	적용 기법	적합한 상황
1	Naive RAG Baseline	모든 프로젝트의 시작점. 기본 성능 확인
2	성능 모니터링	실패 패턴을 파악하고 병목 지점 식별
3	실패 분석	Semantic gap, fragmentation 등 구체적 문제 진단
4	쿼리 개선	모호한 질문이 많거나 동의어가 많은 도메인
5	메타데이터 필터링	시간/카테고리 등 필터링 기준이 명확한 경우
6	하이브리드 검색	키워드와 의미 검색이 모두 중요한 도메인
7	모듈러 라우팅	다양한 질문 유형을 처리해야 하는 복잡한 시스템

코드 세부 설명: preprocess_documents()

텍스트 청킹 전략

청킹 파라미터 설정

```
def preprocess_documents(  
    documents,  
    chunk_size=500,  
    overlap=50  
):  
    """  
    문서를 검색 가능한  
    청크로 분할  
    """  
  
    chunks = []  
  
    for doc in documents:  
        text = doc['content']  
  
        # 문장 단위로 분할  
        sentences = split_sentences(text)  
  
        current_chunk = ""  
        current_length = 0  
  
        for sentence in sentences:  
            sent_length = len(sentence)  
  
            if current_length + sent_length > chunk_size:  
                # 현재 청크 저장  
                chunks.append({  
                    'text': current_chunk,  
                    'metadata': doc['metadata']  
                })  
  
                # 오버랩 적용  
                overlap_text = current_chunk[-overlap:]  
                current_chunk = overlap_text + sentence  
                current_length = len(current_chunk)  
            else:  
                current_chunk += sentence  
                current_length += sent_length  
  
        # 마지막 청크  
        if current_chunk:  
            chunks.append({  
                'text': current_chunk,  
                'metadata': doc['metadata']  
            })  
  
    return chunks
```

청킹 전략의 중요성

- **chunk_size=500**

검색 단위이자 LLM 컨텍스트 구성의 기본 블록. 너무 작으면 문맥 손실, 너무 크면 검색 정확도 저하

- **overlap=50**

인접 청크 간 오버랩으로 경계에서 문맥이 단절되는 것을 방지. 문장이 두 청크에 걸쳐 있어도 완전한 정보 유지

- **문장 단위 분할**

단순히 글자 수로 자르지 않고 문장 경계를 존중. 의미 단위를 보존하여 검색 품질 향상

- **메타데이터 유지**

원본 문서의 메타데이터를 각 청크에 보존하여 필터링 및 추적 가능

코드 세부 설명: `setup_vector_store()`

BAAI/bge-m3 임베딩 + FAISS 인덱스

벡터스토어 구축

```
def setup_vector_store(chunks):
    """
    BAAI/bge-m3 임베딩으로
    FAISS 벡터 DB 구축
    """

    from sentence_transformers import SentenceTransformer
    import faiss

    # 한국어 최적화 임베딩 모델
    model = SentenceTransformer(
        'BAAI/bge-m3'
    )

    # 청크를 벡터로 변환
    texts = [chunk['text'] for chunk in chunks]
    embeddings = model.encode(
        texts,
        show_progress_bar=True
    )

    # FAISS L2 인덱스 생성
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatL2(dimension)

    # 벡터 추가
    index.add(embeddings)

    return {
        'index': index,
        'chunks': chunks,
        'model': model
    }
```

BAAI/bge-m3의 장점

- **다국어 지원:** 한국어, 영어 등 100개 이상 언어에 대해 고품질 임베딩 제공
- **한국어 최적화:** 한국어 코퍼스로 사전학습되어 미묘한 의미 차이 포착
- **높은 성능:** MTEB 벤치마크에서 최상위 성능 기록
- **효율성:** 빠른 인코딩 속도와 합리적인 모델 크기

FAISS L2 인덱스

Facebook의 고속 벡터 검색 라이브러리입니다. L2 거리(유clidean 거리)로 유사도를 측정하며, 수백만 개의 벡터에서도 밀리초 내에 검색이 가능합니다.

코드 세부 설명: Day1FinetunedLLM 클래스

RAFT 파인튜닝 모델 활용

```
class Day1FinetunedLLM:  
    """  
    Day 1에서 파인튜닝한  
    EXAONE 모델 래퍼  
    """  
  
    def __init__(  
        self,  
        model_name="ryanu/my-exaone-raft-model"  
    ):  
        self.model_name = model_name  
        self.tokenizer = AutoTokenizer.from_pretrained(  
            model_name  
        )  
        self.model = AutoModelForCausalLM.from_pretrained(  
            model_name,  
            device_map="auto"  
        )  
  
    def generate(self, query, context):  
        """  
        검색된 컨텍스트와 쿼리로  
        답변 생성  
        """  
        prompt = f"""  
        다음 정보를 바탕으로 질문에 답하세요:  
  
        정보: {context}  
  
        질문: {query}  
  
        답변:  
        """  
  
        inputs = self.tokenizer(  
            prompt,  
            return_tensors="pt"  
        ).to(self.model.device)  
  
        outputs = self.model.generate(  
            **inputs,  
            max_new_tokens=256,  
            temperature=0.7,  
            top_p=0.9  
        )  
  
        answer = self.tokenizer.decode(  
            outputs[0],  
            skip_special_tokens=True  
        )  
  
        return answer
```

코드 세부 설명: simple_vector_search()

의미 기반 검색의 힘

```
def simple_vector_search(  
    query,  
    embeddings,  
    chunks,  
    k=5  
):  
    """  
    벡터 유사도로  
    의미적으로 관련된 문서 검색  
    """  
  
    from sklearn.metrics.pairwise import cosine_similarity  
  
    # 쿼리를 벡터로 변환  
    query_vector = embedding_model.encode([query])  
  
    # 모든 문서 벡터와 코사인 유사도 계산  
    similarities = cosine_similarity(  
        query_vector,  
        embeddings  
    )[0]  
  
    # 상위 k개 인덱스  
    top_k_indices = np.argsort(similarities)[-k:][:-1]  
  
    # 결과 구성  
    results = [  
        {  
            'index': idx,  
            'similarity': similarities[idx],  
            'document': chunks[idx]['text'],  
            'metadata': chunks[idx]['metadata']  
        }  
        for idx in top_k_indices  
    ]  
  
    return results
```

벡터 검색의 장점



동의어 인식

"비용"과 "요금", "구매"와 "구입" 같은 동의어를 자동으로 인식합니다.



문맥 이해

단어 개별이 아닌 전체 문장의 의미를 파악하여 관련 문서를 찾습니다.



다국어 지원

bge-m3 같은 다국어 모델은 언어 간 검색도 지원합니다.



패러프레이즈

다른 표현으로 쓰여진 같은 의미의 문장을 효과적으로 검색합니다.

코드 세부 설명: hardcoded_routing vs llm_routing

규칙 기반과 LLM 기반 라우팅 비교

Hardcoded Routing (규칙 기반)

```
def hardcoded_routing(query):
    """
    명시적 규칙으로 쿼리 분류
    """

    # 시간 관련 질문
    temporal_keywords = [
        '언제', '날짜', '시간',
        '기간', '~부터', '~까지'
    ]
    if any(kw in query for kw in temporal_keywords):
        return 'temporal_pipeline'

    # 추론 질문
    reasoning_keywords = [
        '왜', '이유', '어떻게',
        '방법', '원인'
    ]
    if any(kw in query for kw in reasoning_keywords):
        return 'reasoning_pipeline'

    # 비교 질문
    comparison_keywords = [
        '차이', '비교', '더', '덜',
        '유사', '다른'
    ]
    if any(kw in query for kw in comparison_keywords):
        return 'comparison_pipeline'

    # 기본: 단순 검색
    return 'simple_pipeline'
```

장점

- 즉각적인 응답 (추가 지연 없음)
- 비용 없음 (LLM 호출 불필요)
- 명확하고 예측 가능

단점

- 규칙 추가/유지보수 부담
- 복잡한 쿼리 처리 어려움

LLM Routing (모델 기반)

```
def llm_routing(query):
    """
    LLM이 쿼리 유형 자동 판단
    """

    prompt = f"""
    다음 질문의 유형을 분류하세요.

    유형:
    - simple: 단순 사실 질문
    - temporal: 시간/날짜 관련
    - reasoning: 설명/추론 필요
    - comparison: 비교/대조
    - multihop: 여러 정보 조합 필요

    질문: {query}

    유형 (한 단어로):
    """

    response = llm.generate(
        prompt,
        max_tokens=10
    )

    query_type = response.strip().lower()

    # 유형별 파이프라인 맵핑
    pipeline_map = {
        'simple': 'simple_pipeline',
        'temporal': 'temporal_pipeline',
        'reasoning': 'reasoning_pipeline',
        'comparison': 'comparison_pipeline',
        'multihop': 'multihop_pipeline'
    }

    return pipeline_map.get(
        query_type,
        'simple_pipeline'
    )
```

장점

- 복잡한 의도 파악
- 규칙 없이 자동 적응

단점

- 추가 지연 시간 (100-300ms)
- API 호출 비용
- 예측 가능성 낮음

코드 세부 설명: Self-RAG STRICT_PROMPT

답변 품질 자체 검증

STRICT_PROMPT = """

당신은 엄격한 사실 검증자입니다.

질문: {query}

검색된 정보:

{context}

생성된 답변:

{answer}

다음 기준으로 답변을 평가하세요:

1. 근거 충분성 (0-10점)

- 답변의 모든 주장이 검색된 정보에 명시적으로 뒷받침되는가?
- 추측이나 외부 지식 사용은 없는가?

2. 정확성 (0-10점)

- 검색된 정보를 올바르게 해석했는가?
- 왜곡이나 과장은 없는가?

3. 완전성 (0-10점)

- 질문의 모든 부분에 답했는가?
- 중요한 정보 누락은 없는가?

4. 신뢰도 (0-10점)

- 불확실한 부분을 명시했는가?
- 과도한 확신은 없는가?

총점: __/40

판정: [PASS/FAIL/UNCERTAIN]

이유:

만약 FAIL이거나 총점이 28점 미만이면

"충분한 근거가 없어 답변할 수 없습니다"를
반환하세요.

Self-RAG의 작동 원리



검색 수행

쿼리에 관련된 문서들을 검색합니다



답변 생성

검색된 컨텍스트로 초안 답변을 생성합니다



자체 평가

STRICT_PROMPT로 생성된 답변의 신뢰도를 검증합니다



판정

기준 통과 시 답변 반환, 실패 시 거부하거나 쿼리 개선 후 재시도

환각 방지: Self-RAG는 LLM이 검색된 정보 이외의 내용을 "환각"하여 만들어내는 것을 방지합니다. 근거가 불충분하면 답변을 거부하여 신뢰성을 보장합니다.

코드 세부 설명: REFINE_PROMPT

쿼리 자동 개선 시스템

```
REFINE_PROMPT = """  
이전 검색이 실패했습니다.  
쿼리를 개선하여 더 나은 결과를 얻으세요.
```

원래 질문: {original_query}

실패 이유:
{failure_reason}

검색된 문서들 (부족함):
{retrieved_docs}

다음 전략을 사용하여 쿼리를 개선하세요:

1. 구체화
- 모호한 용어를 구체적으로 명시
- 시간, 장소, 대상 등 제약 추가

2. 분해
- 복잡한 질문을 여러 하위 질문으로 분해
- 각각 독립적으로 답할 수 있도록

3. 확장
- 동의어나 관련 용어 추가
- 다양한 표현 방식 시도

4. 맥락 추가
- 질문의 배경이나 목적 명시
- 필요한 정보의 유형 구체화

개선된 쿼리 (1-3개):

```
1.  
2.  
3.  
.....
```

```
def retry_with_refine(  
    original_query,  
    failure_reason,  
    max_retries=3  
):  
    """  
    쿼리 개선 후 재검색  
    """  
  

```

점진적 개선 프로세스

01

실패 분석

왜 검색이 실패했는지 구체적인 이유 파악 (모호함, 용어 불일치 등)

02

전략 선택

실패 유형에 맞는 개선 전략 적용 (구체화, 분해, 확장, 맥락 추가)

03

쿼리 재생성

1-3개의 개선된 쿼리를 생성하여 다양한 접근 시도

04

재검색 및 검증

개선된 쿼리로 재검색 후 Self-RAG로 품질 재평가

최대 3회까지 반복하며, 여전히 충분한 답변을 생성하지 못하면 솔직하게 "답변할 수 없습니다"를 반환합니다.

실습 환경 주의사항

FutureWarning 및 일반적인 문제 해결

⚠ CUDA Out of Memory

오류: "RuntimeError: CUDA out of memory"

해결 방법:

- per_device_train_batch_size를 1로 감소
- gradient_accumulation_steps를 4-8로 증가
- max_length를 256으로 감소
- max_train_samples를 50으로 제한

⚠ 느린 학습 속도

원인: GPU가 아닌 CPU에서 실행

확인: torch.cuda.is_available() 체크

해결: Google Colab GPU 런타임 사용 또는 device_map="auto" 설정

실전 배포 체크리스트

프로덕션 환경 준비 사항

1	<h3>데이터 준비</h3> <ul style="list-style-type: none">▣ 도메인 특화 문서 수집 및 정제▣ 메타데이터 스키마 설계 (날짜, 카테고리, 출처 등)▣ 청크 크기 최적화 (도메인별 실험 필요)▣ 중복 문서 제거 및 품질 검증
2	<h3>모델 선택</h3> <ul style="list-style-type: none">▣ 임베딩 모델 벤치마크 (bge-m3, E5, multilingual-e5)▣ LLM 선택 (비용, 속도, 품질 트레이드오프)▣ 파인튜닝 필요성 평가▣ 모델 서빙 인프라 구축 (Triton, TorchServe 등)
3	<h3>성능 최적화</h3> <ul style="list-style-type: none">▣ 벡터 DB 선택 (FAISS, Pinecone, Weaviate 등)▣ 캐싱 전략 (빈번한 쿼리 결과 저장)▣ 배치 처리 구현 (여러 쿼리 동시 처리)▣ 지연 시간 프로파일링 및 병목 해소
4	<h3>모니터링</h3> <ul style="list-style-type: none">▣ 성능 지표 대시보드 구축▣ 실패 쿼리 로깅 및 분석▣ A/B 테스트 인프라▣ 사용자 피드백 수집 시스템
5	<h3>비용 관리</h3> <ul style="list-style-type: none">▣ API 호출 비용 예측 및 최적화▣ 인프라 비용 모니터링 (GPU, 스토리지)▣ 사용량 기반 요금제 고려▣ 오픈소스 대안 평가

도메인별 최적화 전략

업종에 맞는 RAG 커스터마이징



고객 지원

특징: 빠른 응답 속도, 높은 정확도 요구

전략:

- FAQ 우선 검색 (BM25 가중치 높임)
- 시간 필터링 (최신 정책 우선)
- Self-RAG로 확실한 답변만 제공
- Refine으로 모호한 질문 자동 명확화



법률/규정

특징: 정확성이 가장 중요, 오류 허용 불가

전략:

- 키워드 검색 강화 (정확한 법령 용어)
- 발행 날짜 메타데이터로 최신 법령 우선
- 참조 조항 자동 링크
- High threshold Self-RAG (35점 이상만 통과)



의료/건강

특징: 신뢰성, 개인정보 보호

전략:

- 출처 메타데이터 필수 (의학 저널, 가이드라인)
- 전문 용어 임베딩 (BioBERT 등)
- 환자 데이터 익명화
- 면책 조항 자동 추가



교육

특징: 설명력, 다양한 나이도

전략:

- 난이도 메타데이터로 학생 수준 맞춤
- Sub-question으로 복잡한 개념 단계적 설명
- 예시와 비유 자동 생성
- 학습 경로 추천

벡터 데이터베이스 비교

프로덕션 환경에 맞는 선택

특성	FAISS	Pinecone	Chroma	Milvus
배포 방식	로컬 라이브러리, 셀프 호스팅 (Python/C++)	완전 관리형 클라우드 SaaS	로컬/클라이언트-서버, 셀프 호스팅	분산 아키텍처, 셀프 호스팅 (Kubernetes)
확장성	단일 노드 최적화. 수평 확장 직접 구현 필요	자동 스케일링, 고가용성 (페타바이트 규모)	중소 규모 앱에 적합, 제한적 스케일링	클라우드 네이티브, 대규모 수평 확장
비용	오픈소스, 무료 (인프라 비용 별도)	사용량 기반 과금 (쿼리, 저장)	오픈소스, 무료 (인프라 비용 별도)	오픈소스, 무료 (인프라 및 관리 비용)
검색 속도	매우 빠름 (특히 GPU 활용 시)	대규모에서도 빠른 응답 속도 보장	보통 (인메모리 인덱스, 임베디드)	빠름 (대규모 병렬 처리)
메타데이터 필터	네이티브 지원 없음, 직접 구현 필요	네이티브 지원, 복합 필터링 가능	제한적 지원 (간단한 필터링)	네이티브 지원, 강력한 필터링 기능
하이브리드 검색	별도 구현 필요 (키워드 검색과 통합)	네이티브 지원 (벡터 및 키워드 검색)	별도 구현 필요	네이티브 지원 (RRF, BM25)
주요 특징	고성능 벡터 검색 알고리즘 (Facebook AI)	완전 관리형, 제로 오퍼레이션, 안정성	가볍고 사용하기 쉬운 API, Python 중심	클라우드 네이티브, 풍부한 기능, 높은 유연성
프로덕션 선택	강력한 GPU, 미세 조정 역량, 저비용 프로토타입	빠른 배포, 관리 최소화, 대규모 데이터셋	개발 환경, 간단한 RAG, 빠른 POC	대규모 분산 환경, 온프레미스/커스터마이징

▣ 권장사항:

- FAISS:** 강력한 컴퓨팅 자원을 보유하고, 프로토타이핑 또는 특정 연구 목적의 고성능 단일 노드 벡터 검색에 적합합니다. 직접 분산 시스템을 구축할 경우 매우 유연합니다.
- Pinecone:** 운영 부담을 최소화하고 싶고, 빠르고 안정적인 대규모 서비스가 필요한 경우 최적입니다. 비용은 사용량에 비례합니다.
- Chroma:** 간단한 RAG 애플리케이션이나 개발 초기 단계에 빠르게 시작하고 싶을 때 좋습니다. 임베디드 모드로 간편하게 사용할 수 있습니다.
- Milvus:** 대규모 분산 환경이나 온프레미스 배포가 필수적인 경우, 높은 유연성과 확장성이 요구될 때 적합합니다. 관리 부담은 있으나 기능이 풍부합니다.

임베딩 모델 벤치마크

한국어 성능 비교

MTEB 한국어 벤치마크

모델	정확도	속도
BAAI/bge-m3	84.2%	빠름
intfloat/multilingual-e5-large	82.7%	보통
jhgan/ko-sroberta	79.5%	매우 빠름
OpenAI text-embedding-3	86.1%	API 의존
sentence-transformers paraphrase-multilingual	75.3%	빠름

선택 기준

- 정확도 우선

의료, 법률 등 정확성이 중요하면 OpenAI나 bge-m3 선택

- 속도 우선

실시간 대화형 앱은 ko-sroberta나 multilingual-e5-base

- 비용 고려

오픈소스 모델(bge, e5)로 셀프 호스팅하면 API 비용 절감

- 다국어 지원

여러 언어 지원 필요 시 multilingual-e5나 bge-m3

캐싱 전략으로 성능 2배 향상

빈번한 쿼리 최적화

계층적 캐싱 구현

```
from functools import lru_cache
import redis

class RAGCache:
    def __init__(self):
        # L1: 메모리 캐시 (빠름, 용량 작음)
        self.memory_cache = {}

        # L2: Redis (보통 속도, 용량 큼)
        self.redis_client = redis.Redis(
            host='localhost',
            port=6379
        )

    def get_cached_result(self, query):
        # L1 캐시 확인
        if query in self.memory_cache:
            return self.memory_cache[query]

        # L2 캐시 확인
        cached = self.redis_client.get(query)
        if cached:
            result = json.loads(cached)
            # L1에 승격
            self.memory_cache[query] = result
            return result

        return None

    def set_cached_result(
        self,
        query,
        result,
        ttl=3600
    ):
        # L1에 저장
        self.memory_cache[query] = result

        # L2에 저장 (TTL 설정)
        self.redis_client.setex(
            query,
            ttl,
            json.dumps(result)
        )

    def cached_rag_query(query):
        cache = RAGCache()

        # 캐시 확인
        cached = cache.get_cached_result(query)
        if cached:
            return cached

        # 캐시 미스: 실제 RAG 수행
        result = perform_rag(query)

        # 캐시 저장
        cache.set_cached_result(query, result)

        return result
```

A/B 테스트로 RAG 최적화

데이터 기반 의사결정

테스트 가설 예시

가설 1: Query Refinement를 적용하면 답변 정확도가 20% 이상 향상될 것이다

측정 지표: 정확도, 응답 시간, 사용자 만족도

A 그룹: Naive RAG / **B 그룹:** Query Refinement 적용

가설 2: 하이브리드 검색

가설: BM25 + 벡터 검색을 결합하면 키워드 쿼리와 의미 쿼리 모두에서 성능이 향상될 것이다

측정: 키워드 쿼리 정확도, 의미 쿼리 정확도, 평균 응답 시간

A: 벡터 검색만 / **B:** 하이브리드 검색

가설 3: Self-RAG 임계값

가설: Self-RAG 임계값을 28점에서 32점으로 높이면 환각은 감소하지만 답변 거부율도 증가할 것이다

측정: 환각률, 답변 거부율, 사용자 신뢰도

A: 임계값 28 / **B:** 임계값 32

- 통계적 유의성: 각 그룹에 최소 1000개 이상의 쿼리를 테스트하고, $p\text{-value} < 0.05$ 를 기준으로 유의성을 판단합니다. 최소 1-2주간 테스트를 진행하여 시간대와 사용자 행동 패턴을 반영합니다.

실패 사례와 교훈

실제 프로젝트에서 배운 것들

사례 1: 과도한 청크 크기

문제: chunk_size=2000으로 설정하여 문서당 청크 수가 너무 적었습니다. 검색 정확도가 55%에 불과했습니다.

원인: 큰 청크는 관련 없는 정보도 많이 포함하여 노이즈가 증가했습니다.

해결: chunk_size=500, overlap=50으로 조정하여 정확도 78%로 향상

교훈: 청크 크기는 도메인과 문서 특성에 맞게 실험이 필요합니다. 일반적으로 300-800이 적합합니다.

사례 2: 캐시 무효화 실패

문제: 문서 업데이트 후에도 오래된 캐시가 계속 사용되어 사용자가 구버전 정보를 받았습니다.

원인: 캐시 무효화 로직 없이 무제한 TTL 사용

해결: 문서 업데이트 이벤트와 연동하여 관련 캐시 자동 삭제, 기본 TTL 6시간 설정

교훈: 캐싱은 신선도와 성능의 트레이드오프입니다. 도메인에 맞는 TTL 전략이 필수입니다.

사례 3: Self-RAG 과신

문제: Self-RAG를 너무 신뢰하여 인간 검증 없이 배포했더니 일부 답변이 여전히 부정확했습니다.

원인: LLM도 완벽하지 않으며, Self-RAG 자체가 실수할 수 있습니다.

해결: 중요한 쿼리(법률, 의료)는 인간 검증 단계 추가, 로깅 강화하여 지속적 모니터링

교훈: AI는 보조 도구입니다. 고위험 도메인에서는 인간 감독이 여전히 중요합니다.

추가 질문과 디버깅 팁

Q1: 검색 결과가 관련 없는 문서만 나올 때

진단:

- 임베딩 모델이 도메인에 맞지 않음
- 청크 크기가 너무 크거나 작음
- 쿼리 표현이 문서 표현과 다름 (Semantic Gap)

해결:

- 도메인 특화 임베딩 모델로 변경 (BioBERT, Legal-BERT 등)
- 청크 크기 실험 (300, 500, 800 등)
- Query Refinement 적용하여 쿼리 확장
- 하이브리드 검색으로 BM25 추가

Q2: LLM이 컨텍스트를 무시하고 사전 지식으로 답할 때

진단: 모델이 파인튜닝되지 않았거나, 프롬프트가 약함

해결:

STRICT_PROMPT = """"

아래 정보만을 사용하여 답하세요.
정보에 없는 내용은 "정보에 없습니다"라고
명시하세요.

정보: {context}

질문: {query}

답변 (정보 기반):

"""

또는 RAFT 파인튜닝을 수행하여 모델이 컨텍스트
우선 사용하도록 학습시킵니다.

Q3: 메모리 부족 오류 (OOM)

해결 체크리스트:

- `[batch_size]`를 1로 감소
- `[gradient_accumulation_steps]`를 4-8로 증가 (유효 배치 크기 유지)
- `[max_length]`를 256 또는 512로 제한
- `[모델]`을 int8이나 4bit로 양자화 (`load_in_8bit=True`)
- `[gradient_checkpointing=True]` 활성화 (느리지만 메모리 절약)
- `[더 작은 모델]`로 변경 (EXAONE-7B → 3B)

Day 3 미리보기: LangGraph Agent

RAG에서 자율 에이전트로

Day 2에서 구축한 RAG 시스템은 강력하지만, 여전히 정적입니다. 주어진 쿼리에 대해 미리 정의된 파이프라인을 따라 움직일 뿐입니다. Day 3에서는 LangGraph를 활용하여 RAG를 동적 의사결정이 가능한 자율 에이전트로 진화시킵니다.



도구 사용

1

에이전트가 필요에 따라 웹 검색, API 호출, 계산기 등 다양한 도구를 자율적으로 선택하고 사용합니다. RAG는 도구 중 하나가 됩니다.



계획 수립

2

복잡한 작업을 하위 작업으로 분해하고, 순서를 정하고, 조건부 실행을 결정하는 계획 능력을 갖춥니다.



장기 메모리

3

대화 히스토리, 사용자 선호도, 과거 작업 결과를 기억하고 활용하여 맥락 있는 상호작용을 제공합니다.



멀티 에이전트

4

여러 전문화된 에이전트가 협업하여 복잡한 문제를 해결합니다. 검색 에이전트, 분석 에이전트, 실행 에이전트 등이 조율됩니다.



LangGraph 핵심 개념

상태 그래프로 에이전트 설계하기

상태(State) 관리

에이전트의 현재 상태를 표현하는 데이터 구조입니다. 대화 히스토리, 중간 결과, 메타 데이터 등을 포함합니다.

```
from typing import TypedDict, List

class AgentState(TypedDict):
    messages: List[str]
    current_task: str
    tools_used: List[str]
    intermediate_results: dict
    final_answer: str
```

노드(Node)

상태를 입력받아 작업을 수행하고 새로운 상태를 반환하는 함수입니다.

```
def search_node(state: AgentState):
    query = state['current_task']
    results = perform_rag(query)
    state['intermediate_results']['search'] = results
    return state

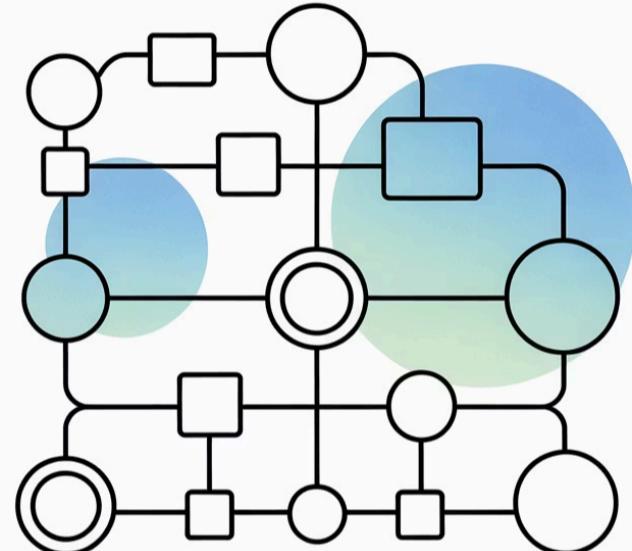
def analyze_node(state: AgentState):
    results = state['intermediate_results']['search']
    analysis = llm.analyze(results)
    state['final_answer'] = analysis
    return state
```

엣지(Edge) - 조건부 라우팅

다음 실행할 노드를 동적으로 결정합니다.

```
def route_after_search(state: AgentState):
    results = state['intermediate_results']['search']

    if len(results) == 0:
        # 결과 없으면 쿼리 개선
        return "refine_query_node"
    elif needs_more_info(results):
        # 추가 검색 필요
        return "search_node"
    else:
        # 분석 진행
        return "analyze_node"
```



DECISION FLOWCHART

그래프 구성

```
from langgraph.graph import StateGraph

workflow = StateGraph(AgentState)

# 노드 추가
workflow.add_node("search", search_node)
workflow.add_node("refine", refine_query_node)
workflow.add_node("analyze", analyze_node)

# 엣지 추가
workflow.add_conditional_edges(
    "search",
    route_after_search,
    {
        "refine_query_node": "refine",
        "search_node": "search",
        "analyze_node": "analyze"
    }
)

app = workflow.compile()
```

Day 2 핵심 요약

학습한 모든 것을 한눈에

Naive RAG 기반

문서 청킹, 벡터스토어, 검색, LLM 응답 생성의 기본 파이프라인을 구축했습니다. 성능 측정 도구로 개선 방향을 파악했습니다.



실패 패턴 분석

Semantic gap, context fragmentation 등 5가지 주요 실패 시나리오를 이해하고, 각각에 대한 해결 전략을 배웠습니다.

쿼리 개선 기법

Multi-query, Sub-question, Query Expansion, RAG-Fusion으로 검색 정확도를 30-40% 향상시켰습니다.



메타데이터 활용

시간, 카테고리, 출처 필터링으로 관련성 높은 문서만 선별하여 정밀도를 높였습니다.

하이브리드 검색

BM25 키워드 검색과 벡터 의미 검색을 결합하고, Cross-encoder로 재순위화하여 최상의 결과를 제공했습니다.



모듈러 아키텍처

쿼리 타입별 최적 파이프라인 라우팅, Self-RAG 검증, 자동 쿼리 개선으로 지능형 시스템을 완성했습니다.

실무 배포 체크리스트 (재정리)

프로덕션 준비 최종 점검

필수 항목

01

▣ 도메인 데이터 준비

충분한 양의 고품질 문서, 메타데이터 스키마, 청크 크기 최적화

02

▣ 모델 선택 및 벤치마크

임베딩 모델, LLM 비교, 파인튜닝 평가, 비용-성능 분석

03

▣ 인프라 설정

벡터 DB, 캐싱, 로드밸런싱, 모니터링 대시보드

04

▣ 평가 시스템

A/B 테스트, 성능 지표 추적, 실패 로깅, 사용자 피드백

05

▣ 보안 및 규정 준수

데이터 암호화, 접근 제어, GDPR/개인정보보호, 감사 로그

권장 항목

- ▣ 멀티모달 지원

이미지, 비디오 검색 준비

- ▣ 다국어 확장

multilingual 임베딩 모델

- ▣ 실시간 업데이트

문서 변경 시 자동 재인덱싱

- ▣ 개인화

사용자 선호도 학습 및 반영

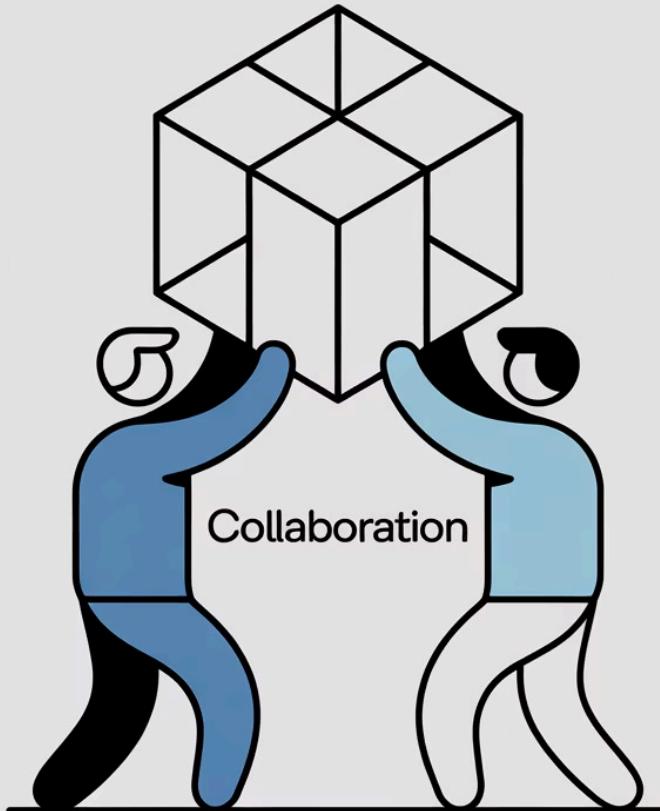
- ▣ 비용 최적화

캐싱 강화, 모델 양자화, 오픈소스 전환

- ▣ 장애 대응

풀백 전략, 재시도 로직, 서킷 브레이커





성공적인 RAG 프로젝트의 비결

실무자들의 조언

"점진적으로 개선하세요"

처음부터 완벽한 시스템을 만들려 하지 마세요. Naive RAG부터 시작해서 실패를 분석하고, 하나씩 개선해 나가세요. 데이터 기반으로 의사결정하면 올바른 방향으로 나아갈 수 있습니다.

— 김민수, NLP 엔지니어

"도메인 전문가와 협업하세요"

기술만으로는 부족합니다. 해당 도메인을 깊이 이해하는 전문가와 함께 일하면서 무엇이 좋은 답변인지, 어떤 실수가 치명적인지 배워야 합니다. 그들의 피드백이 가장 중요한 학습 데이터입니다.

— 이지은, AI 프로덕트 매니저

"사용자 피드백을 적극 수집하세요"

개발자가 보는 지표와 실제 사용자 경험은 다를 수 있습니다. 답변 평가 버튼, 개선 제안 양식 등으로 지속적으로 피드백을 받고, 이를 다음 버전에 반영하는 선순환을 만드세요.

— 박준호, UX 리서처