RAG 기반 AI 기술 실전 가이드

Ver. 2025.05

작성자: 김재우

Email: gaebalai.official@gmail.com

최근 AI의 진화에 의해 다양한 분야에서의 응용이 진행되고 있습니다. 특히, 자연어처리(NLP) 분야에서는 RAG(Retrieval-Augmented Generation, 검색증강생성)이 주목받고 있습니다.

또한, NativeRAG, GraphRAG, AgentRAG등 다양한 RAG의 변형이 나오고 있으며, 이들은 특정 유스케이스와 데이터세트에 최적화되어 있습니다.

이번에는 RAG의 기본적인 개념부터 RAG 프로젝트의 진행방법, 정밀도 향상 방법에 이르기까지 자세히 설명합니다.

여러분의 AI애플리케이션 개발에 도움이 되길 빕니다.

[목 차]

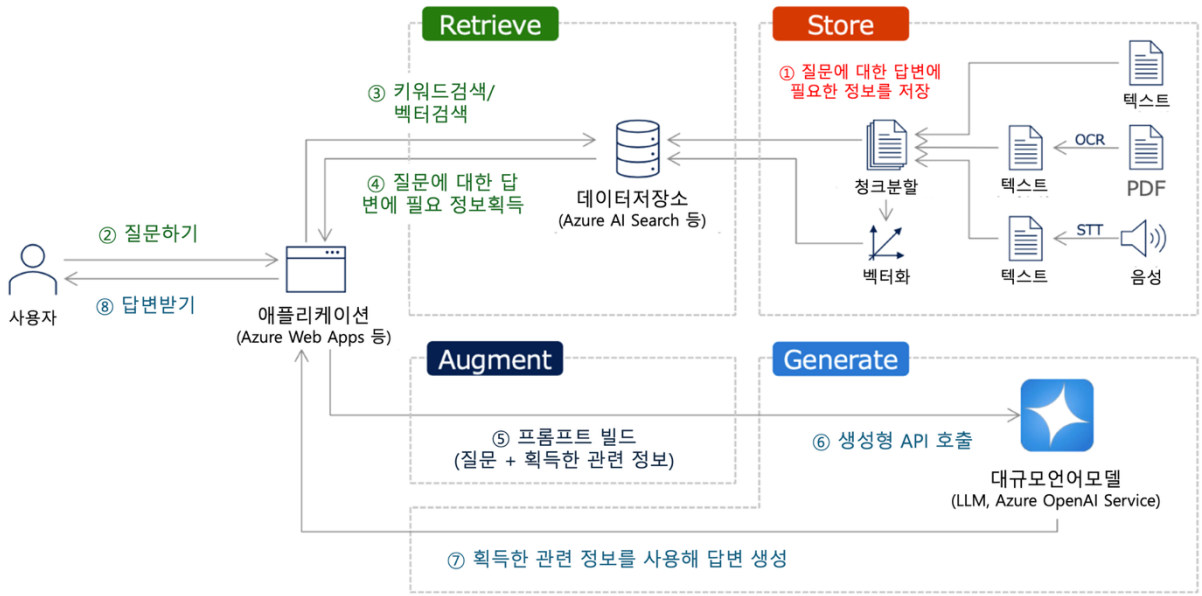
1. RAG 개요

RAG는 Retrieval-Augmented Generation(검색증강생성)의 약어로 정보검색과 생성을 결합한 기법입니다. RAG는 특히 대규모언어모델(LLM)과 결합하여 성능을 크게 향상시킬 수 있습니다.

역사적으로는 Meta의 연구자가 제안한 LLM의 할루시네이션(잘못된 정보 생성)을 낮추는 수법으로, LLM만의 사고가 아니고, 외부 자원에 검색을 걸면서 올바른 대답을 생성한다고 하면 '지식베이스의 외부화'를 실현할 수 있는 수법입니다.

사용자의 질문에 대해 백엔드 지식 기반(예: Azure AI Search)에서 검색하고 결과를 프롬프트에 추가하여 답변을 생성하는 흐름이 일반적인 흐름이 됩니다.

RAG 개요의 흐름은 아래와 같습니다.



1. Store에서 각종 데이터를 청크 분할하고 벡터화하여 데이터스토어(Azure AI Search 등)에 저장합니다.

2. 사용자의 질문을 Application측에서 받습니다.

3. Retrieve라고 표시된 위치에서 키워드 검색 및 벡터 검색을 수행합니다.

4. 관련 정보를 데이터스토어로부터 취득합니다.

5. 사용자의 질문과 데이터 스토어에서 얻은 관련 정보를 Augment에서 빌드합니다.

6. 생성형 AI의 API(예: Auzre OpenAI Service)에 사용자의 질문과 데이터스토어에서 얻은 관련 정보를 프롬프트로 전달합니다.

7. 질문과 관련 정보를 바탕으로 답변을 생성합니다.

8. 생성된 답변을 사용자에게 반환합니다.

이러한 방식으로 RAG는 다음 4가지 위치로 나눌 수 있습니다.

- Store: 데이터 저장
- Retrieve: 데이터 검색
- Augment: 프롬프트 빌드
- Generation: 답변생성

1-1. RAG 접근법

RAG의 정확성을 높이기 위한 방법에 대한 연구가 진행되고 있으며, 키워드 및 벡터검색을 기반으로 **NativeRAG**, KG(지식그래프)를 활용하는 **GraphRAG**, 그리고 두가지를 결합하는 **HybridRAG**가 주요한 옵션으로 들 수 있습니다.

또한, 요즘 주목 받고 있는 것은 AgenticRAG라고 불리는 방법으로 기존의 RAG가 가지고 있는 '정적 워크플로우', '단일단계 추론'등의 한계를 극복하기 위해 LLM 에이전트화(계획, 재시행, 도구 활용, 다른 에이전트와의 협조 등)이 있습니다. 질의별로 동적으로 검색 전략을 바꾸거나 응답 도중에 부족 정보를 재검색하거나 여러 에이전트 간에 태스크를 분담하고 협조합니다.

RAG에 대해 비교해 보았습니다.

구분	유스케이스	검색방법	장점	단점
NativeRAG	- 다양한 문서 검색 - FAQ 및 고객지원 - 표현변동에 대한 대응	- 키워드검색 - 벡터검색 - 시맨틱리랭킹	- 스키마가 필요 없고 구현하기 쉬움 - 대규모 비구조	- 관련성의 엄밀한 추론에는 약함 - 노이즈가 많은

			텍스트를 받아들이기 쉬움 - 유연성 높음	경우, 답변 정확성 떨어짐
GraphRAG	- 엔티티의 엄격한 관계추적 - 설명책임이 요구되는 경우 - 인과관계분석	- 지식그래프를 이용한 구조적 검색 - 엔티티 관계에 따른 문의	- 일관성/설명가능성이 높음 - 도메인 지식의 활용에 장점	- 지식그래프 구축 비용이 높음 - 엔티티가 직접 등장하지 않는 추상질문에 약함
HybridRAG	- 추상질문에도 대응하면서 관계도 정확하게 파악하고 싶은 경우 - 구조화되지 않은 데이터와 구조화된 데이터가 혼합된 복잡한 도메인	- NativeRAG와 GraphRAG를 모두 구현 - 벡터 검색의 맥락 및 지식그래프의 서브 그래프 정보 병용	- NativeRAG와 GraphRAG의 장점을 살려 정밀도와 유연성을 양립 - 추상적인 질문 및 엔티티 중심의 질문 모두에 강함	- 양쪽 결과를 병합하기 때문에 컨텍스트가 증가하는 경향이 있음 - 구현 및 운영이 복잡해지기 쉬움
AgenticRAG	- 멀티소스를 가로지르는 실시간 QA/데이터통합형 챗봇 - 단계를 분해해야 하는 복잡한 태스크 (예: 법무조사, 코딩 보조 등) - 고객지원 1차 응대 자동화 및 에스컬레이션 판단 - 부서 간 지식 탐색 및 데이터 자산 통합 관리	- 라우팅 및 플래닝 기능을 갖춘 에이전트가 동적으로 도구를 선택 - ReAct/Plan-and-Execute방식의 멀티스텝 추론 루프 구성 가능 - 전용 리트리버(검색기)를 여러 개 연동한 멀티 에이전트 구조 지원	높은 유연성과 적응력 자기검증 루프(self-evaluation)를 통한 노이즈 제거 및 정답률 향상 멀티 에이전트 확장을 통해 이기종 데이터, 멀티모달 대응 용이 단일 벡터DB에 의존하는 방식보다 도 높은 정보 포괄성 확보	- 추론비용 및 지연시간 증가 - 에이전트 루프 중단 또는 할루시네이션 발생 가능성 - 오케스트레이션 및 권한 관리 등의 시스템 구현이 복잡해짐 - 에이전트 간 협업 실패 시 처리 효율 저하 리스크 존재

1-2. Agent시대의 RAG (Agentic RAG)

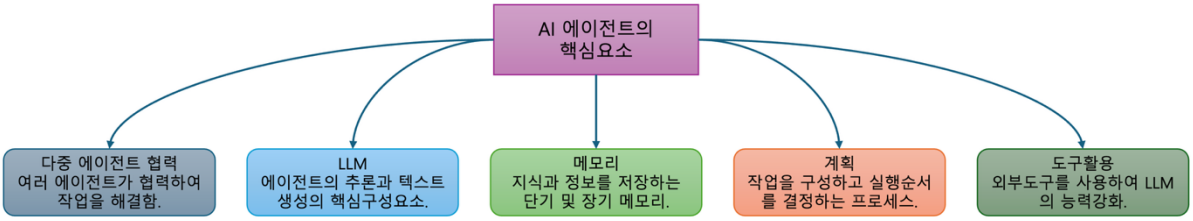
특히 최근 주목받고 있는 AgenticRAG에 대해 깊이 알아봅시다.

다음은 AgenticRAG관련 논문의 요약된 내용입니다.

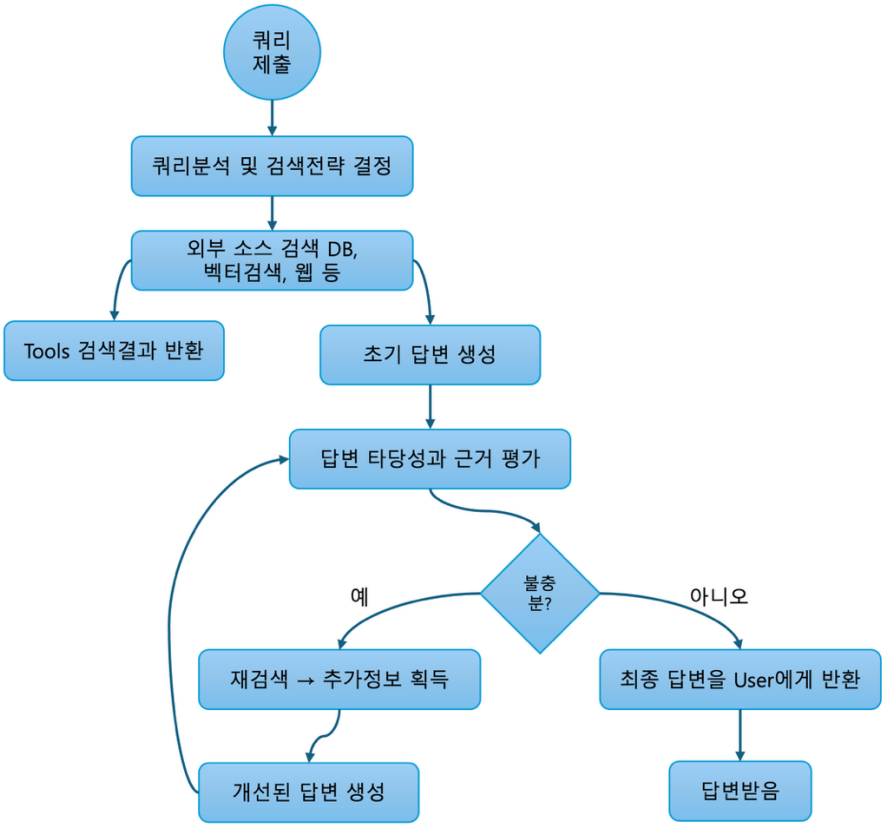
- 관련 논문: https://arxiv.org/abs/2501.09136

> 에이전트 검색확장생성(AgenticRAG)은 자율적인 AI에이전트를 RAG파이프라인에 통합하고 에이전트 설계패턴 반영, 계획, 도구사용 중 다중 에이전트 협업을 활용하여 검색전략을 동적으로 관리하고 컨텍스트 이해를 반복적으로 개선하며 복잡한 작업 요구사항을 충족하도록 워크플로우를 적응시킵니다. 이 통합을 통해 AgenticRAG시스템은 다양한 애플리케이션에서 탁월한 유연성, 확장성 및 컨텍스트 인식을 제공합니다.

AI Agent의 요소에는 다음이 포함되어 있습니다.



멀티에이전트 협업, LLM, 메모리, 계획수립, 도구활용 등의 요소를 조합하여 보다 고도화된 문제해결을 목표로 합니다. 실제로 Agentic RAG의 처리흐름 예시를 다음과 같은 방식으로 구성됩니다.



쿼리를 제출 후, 쿼리 분석 및 검색 전략을 결정하고 외부소스(데이터베이스, 벡터검색, 웹 등)에서 정보를 검색합니다. 만약 답변이 충분하지 않은 경우, 다시 검색하여 추가 정보를 얻고 개선된 답변을 생성합니다.

이전의 RAG는 Retrieve 결과를 그대로 프롬프트에 추가하여 답변을 생성했지만, Agentic RAG는 쿼리분석, 검색전략결정, 답변유효성 평가 등의 단계를 추가하여 보다 동적이고 유연한 대응을 가능하게 합니다.

Agentic RAG Core Loop라고 하지만 다음 교육이 유용합니다. URL: https://github.com/microsoft/ai-agents-for-beginners/blob/main/05-agentic-rag/README.md

그리고 Agentic RAG이라고 해도, 그 구현 방식에는 다양한 접근법이 존재합니다. 이번에는 대표적인 6가지 Agentic RAG유형을 소개하고 비교해 보겠습니다.

- 싱글에이전트 RAG (Single-Agent RAG)
- 멀티에이전트 RAG (Multi-Agent RAG)
- 계층형 에이전트 RAG (Hierarchical RAG)
- 정정기반 RAG (Corrective RAG)
- 적응형 RAG (Adaptive RAG)
- 그래프기반 RAG (Graph-based RAG)
- Agentic Document Workflows (ADW)

구분	개요	장점	주의사항	적용사례
싱글에이전트 RAG	RAG의 일련의 작업을 단일 에이전트가 함께 관리	모두 하나의 에이전트로 완성되기 때문에 에이전트 간의 연동 불필요	- 고급 병렬처리, 전문화에는 한계 - 대규모 및 복잡한 처리에는 적합하지 않음	고객지원 등 비교적 단순한 문의 대응
멀티에이전트 RAG	코디네이터역할의 상담원이 쿼리를 받고 여러 전문 상담원에게 하위작업을 할당	기능을 전문화할 수 있기 때문에 고도 및 복잡한 태스크에 대응하기 쉬움	에이전트 간 협력 설계의 복잡성	대규모 시스템에서 고급 정보 검색 및 집계

	하고 결과 통합			
계층형 에이전트 RAG	상위 에이전트가 전략 수준을 판단하고 하위 에이전트가 실무적인 검색 및 분석을 담당함	대규모 및 복잡한 의사결정을 효과적으로 관리	계층 설계가 불충분하면 병복현상을 일으키기 쉬움	분석팀, 검색팀 등 여러 부문을 가로지르는 워크플로우 제어
정정 기반 RAG(Corrective RAG)	검색결과가 부정확하거나 부족한 경우, 자동으로 수정검색 및 쿼리 재생성을 수행하는 메커니즘	할루시네이션 및 검색실수를 줄임	에이전트의 평가 및 수정 프로세스를 설계할 필요	FAQ시스템 응답 정확도 향상, 사실 기반 보고 및 고객지원
적응형 RAG(Adaptive RAG)	쿼리 난이도(간단,중간,복잡 등)에 따라 검색 및 추론의 순서를 전환함	불필요한 대규모 처리를 실시하지 않아도 되므로 효율이 좋음	쿼리 분류의 정확도가 시스템 전체의 성능에 영향을 미침	사용자의 다양한 질문에 가장 적합한 처리를 선택하는 챗봇
그래프기반 RAG	지식그래프와 텍스트 검색을 결합하여 결과를 글리틱모듈 등으로 평가하면서 정밀화	관계형 지식 구조를 활용한 다단계 추론 가능	그래프DB 구축 및 업데이트 필요	의료 및 학술분야와 같은 복잡한 엔티티 관계를 다루는 시스템
ADW(Agentic Document Workflows)	문서처리(인보이스, 계약서, 보고서 등)에 특화된 워크플로우	여러 단계의 문서처리를 중앙 집중화할 수 있음	문서형식별 세부 설정 및 템플릿 디자인	계약서 심사, 자동청구처리, 보험금 청구관리 등

AI Agent의 애플리케이션 개발에 있어서는 개발하는 제품이 달성하고자 하는 목표나 해결하고자 하는 과제에 따라 적절한 접근법을 선택하는 것이 중요합니다.

각각의 장점과 과제를 정확하게 억제하면서 Agent를 설계함으로써 보다 효과적인 AI Agent를 개발할 수 있습니다.

2. RAG의 정확도 개선 진행 방법

여기서는 Native RAG를 기반으로 RAG를 구축하는 프로젝트 진행방법을 소개합니다.

2-1. 앱 전체 프로세스

대략적인 진행방법은 아래와 같습니다.

요건정리 → PoC/개선 → 실서비스구축 → 지속적인 개선

개인적으로 가장 중요하다고 생각하는 포인트는 요구사항 정의부터 PoC(개념검증)에 들어가는 속도와 품질입니다. 절대 처음부터 높은 퀄리티를 추구하기 보다는, 일단 해보자는 정식으로 만들어가는 것이 중요하다고 느끼고 있습니다.

단계	개요	준비물	예상기간
요건정리	RA에 있어서의 요구사항 정리 예시: - 예상사용자/사용방법 - 개발일정 - 비용/평가기준(실서비스 전환을 위함)	- 히어링/요구사항 정리시트	1~2주
PoC/개선	RAG기법을 포함한 베이스라인 앱을 활용한 신속한 PoC(개념검증) - 검증/평가(질문,답변세트를 사전에 준비) - 과제파악(xx사이클) - 실서비스(운영)구축여부에 대한 의사결정	- RAG샘플앱 - 평가기능 - 과제-솔루션매핑 시트	1~2개월
실서비스 구축	실제운영을 위한 아키텍처 설계 및 구축 - 레퍼런스 아키텍처 + 고객환경정책을 바탕으로 최종 아키텍처 정리 - 구축/책임 있는 AI구현	- 레퍼런스 아키텍처 - 구축 파트너 선정	2~3개월
지속적인 개선	실서비스 운영 중 발생한 과제/요구사항에 대한 대응 - 새로운 데이터나 질문 등 신규 요구사항에 대한 지속적인 개선 - 대화로그의 지속적 분석을 통한 성능 향상	- LLMOps/정밀도 모니터링 체계 - 과제-솔루션 매핑 시트	지속적(장기적)

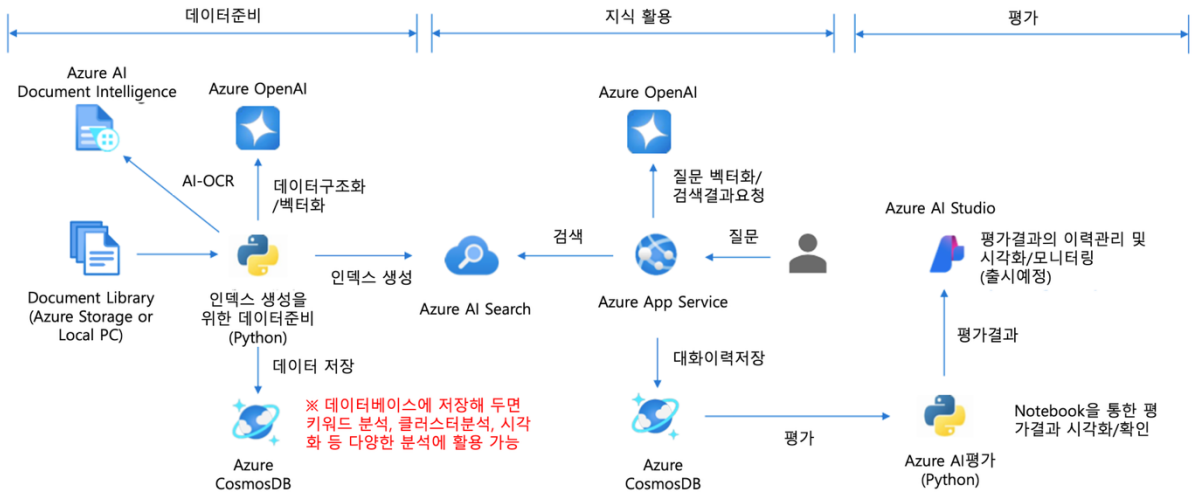
지속적인 개선을 거듭하면서 3번째 릴리스쯤에 좋은 형태의 실서비스가 되는 경우가 많습니다. 앞서 언급했듯이 처음부터 완벽함을 추구하기보다는 우선 해보는 자세가 중요합니다.

PoC 단계에서 효과측정을 위해 필요한 주요 기능은 다음과 같습니다.

▣ 샘플 애플리케이션 주요기능

- [프론트엔드 UI]
  - 파일 업로드 기능
  - 검색 파라미터 변경
  - 하이브리드 시맨틱 검색 등 다양한 검색방식
  - 시스템 규칙 변경 가능
  - 프롬프트 표시 기능 (검색결과 포함)
  - 검색 파라미터/GPT 파라미터 조정
  - AI 응답 내 참조 파일 표시
- [백엔드]
  - AI-OCR (표 구조 대응)
  - 시맨틱 청킹 (Semantic Chunking)
  - 청크 고도화 기능(제목, 요약, 키워드 추출)
  - 평가 기능

이 구성을 Azure상에서 도식화하면 다음과 같은 이미지가 됩니다.



데이터 준비 → 지식 활용 → 평가 사이클을 반복함으로써, RAG의 정밀도(정확도)를 점차 향상시켜 나갈 수 있습니다.

RAG의 정확도 개선 포인트는 크게 4가지 항목으로 나눌 수 있습니다.

- Store: 데이터 저장
- Retrieve: 검색
- Augment: 프롬프트 구성
- Generate: 응답생성

위 4가지 항목을 더 세분화하여, 정확도 개선 방안을 구체적으로 모색해 나갑니다.

각 단계에서 정밀도를 개선 할 수 있는 포인트를 간단히 정리해 보았습니다.

1. 비정형 데이터의 텍스트 추출 정확도 향상 등
2. 청크 분할 최적화 (분할 전략, 분할크기, 오버랩 크기 등)
3. 검색문서의 풍부화 (Enrichment) 등
4. 문서 검색 정확도 튜닝 등
5. 프롬프트 엔지니어링, 쿼리 확장, 최적 모델 사용 등
6. RAG 애플리케이션 UI 개선 등

즉, Store, Retrieve, Augment, Generate 각 단계별로 정확도를 높이기 위한 개선 방법은 다음과 같습니다.

Store	Retrieve	Augment	Generate
PDF파일 텍스트화	검색알고리즘	시스템메시지 정의	모델선택(응답생성용)
Office파일 텍스트화	알고리즘 파라미터 튜닝	사용자메시지 정의	모델선택(임베딩용)
음성데이터텍스트화	스코어링 프로파일	사용자에게 재질문	파인튜닝
영상데이터텍스트화	커스텀분석기(Custom Analyzer)	검색쿼리 생성	멀티모달 처리(텍스트+이미지 등 복합입력)

이미지데이터텍스트화	유사도 튜닝	가설기반문서 임베딩	-
청크분할(청크튜닝)	-	Function Calling	-
오버랩	-	-	-
테이블구조 추출	-	-	-
엔리치먼트(제목,요약, 키워드 추가)	-	-	-
카테고리분류	-	-	-

상기 내용에 추가로 4가지 항목(Store, Retrieve, Augment, Generate) 각각을 평가하면서 동시에 진행하는 것이 중요합니다. 그럼 각 항목에 대해 자세히 살펴보겠습니다.

2-2. Store의 정밀도 향상 (데이터 준비)

① 비정형 데이터의 텍스트화

Store에 데이터를 저장할 때, 텍스트화하면 정확도가 향상됩니다. 이상적인 방식은 '모든 비정형 데이터(PDF, Office, 음성, 영상, 이미지)'를 텍스트로 변환하여 RAG의 Store에 저장하는 것입니다.

자주 사용되는 데이터 유형 및 텍스트화 방식 예시입니다.

데이터 종류	주요 서비스/라이브러리
PDF파일 텍스트화	- Azure AI Vsion - OCR - Azure AI Document Intelligence (+ Azure OpenAI Service – GPT-4o) - 기타 라이브러리(PyMuPDF, PDFMiner 등)
Office 파일 텍스트화	- Azure AI Search – 문서추출스킬 - Microsoft Office SDK(Microsoft.Office.Interop) - 기타 라이브러리(python-docx, python-pptx 등)
음성 데이터 텍스트화	- Azure AI Speech – Recognition API - Azure OpenAI Service – Whisper API
영상 데이터 텍스트화	- Azure Video Indexer
이미지 데이터 텍스트화	- Azure AI Vision – Image Descriptions API

	- Azure OpenAI Service – GPT-4o

PDF나 Office등 문서분석에 자주 활용되는 서비스로는 Azure AI Document Intelligence가 대표적입니다. 이 서비스는 문서의 텍스트 추출, 구조화, 분류, 엔리치먼트 등을 수행할 수 있는 기능을 제공합니다. 또한, Azure AI Vision의 OCR 기능을 활용하면 이미지나 PDF에서 텍스트를 추출할 수 있습니다. 이 OCR기능은 비교적 저렴하게 사용할 수 있어, 가볍게 테스트하는데 적합합니다.

② 청크(Chunk) 전략

청크분할(Chunking)에 대해 먼저 전제지식을 정리해 보겠습니다. 문서를 그대로 LLM(GPT 등)에 전달하면 토큰 한도에 금방 도달하며, 정작 핵심적인 정보만 정확히 추출하기 어려워집니다. 따라서, 검색엔진이나 벡터스토어에 데이터를 저장하기 전에 반드시 '청크분할(Chunking)'이라는 전처리 과정을 거쳐야 합니다.

청크화로 해결할 수 있는 과제	설명
① LLM의 컨텍스트 제한	여러 문서를 LLM의 토큰 한도 내에서 전달할 수 있게 됨
② 검색순위 정확도	문서 내에서 가장 관련성 높은 부분이 우선적으로 검색되도록 유도할 수 있음
③ 임베딩 모델의 한계	하나의 벡터에 담을 수 있는 토큰수에 제한이 있음 (대부분 모델은 512, text-embedding-ada-002의 경우 8,192토큰까지 지원)

주요 포인트는 다음과 같습니다.

- 중간 규모의 PDF라도 수만개의 토큰에 도달하는 경우가 많습니다. 답변이 문서 후반부에 있을 경우, 앞부분에서 잘려나가 정보가 손실되는 문제가 발생합니다.
- 문서가 길수록 청크화의 효과가 더욱 커집니다.

왜 문서는 반드시 청크화해야 하는가?

- 토큰 제한 회피: LLM 입력 한도를 넘지 않기 위해

- 검색효율 향상: 문서 전체가 아닌 관련있는 단락만 검색하기 위해
- 정확도 개선: 쓸모없는 정보 없이 핵심정보만 추출가능
- 문맥단위 이해유도: 문장이나 의미 단위로 분리해 정보 손실 방지

청크화 효과 확인 (Recall@50)

문서 벡터로 처리 vs 청크화

쿼리유형	단일벡터 (최초 4,096토큰만)	청크화 (512토큰+25% 중복)	정확도 개선폭
답변이 긴 문서에 있는 경우	28.2	45.7	+17.5
답변이 문서 깊숙한 곳에 있는 경우	28.7	51.4	+22.7

청크 크기가 크면 어떻게 될까?

청크크기(토큰수)	Recall@50
512	**42.4**
1,024	37.5
4,096	36.4
8,191	34.9

청크 크기를 늘리면 문맥량이 늘어나므로 '정확도(Recall)'가 향상될 수 있지만, 너무 커지면 임베딩 표현이 흐려지거나, 벡터간 차별성이 떨어질 수 있고, LLM의 토큰한도 및 임베딩 모델의 한계(예: 512 ~ 8192토큰)에도 유의해야 합니다.

결론: 너무 큰 청크는 검색성능을 저하시킵니다.

- 512 ~ 1,000토큰 정도가 성능과 효율성의 균형이 가장 좋습니다.

경계전략 및 오버랩(중첩)의 효과

경계전략(512토큰 기준)	Recall@50
토큰 경계에서 단순 분할	40.9

문장경계를 유지하며 분할	**42.4**
+10% 중첩(오버랩) 포함	43.1
+25% 중첩(오버랩) 포함	**43.9**

- 문장 또는 단락 단위로 나누는 것이 문맥 유지에 효과적
- 10~25% 오버랩을 적용하면 Recall이 더 향상됨
- 단, 중첩률이 높아질수록 스토리지 용량 및 토큰 사용량이 증가함에 따라 평가(Evaluation) 단계에서 최적값을 찾아야 함

실무 구현에서 유용한 청킹 전략 Tips

- 청크 크기는 512 ~ 1,024토큰부터 시작하고, '측정지표(Recall 등)'로 조정
- 경계기준은 문장/단락 단위, 필요 시 표나 코드블록도 고려
- '오버랩(중첩)'은 기본적으로 10~20%, 리콜(Recall)이 부족하면 더 늘림
- 메타정보(제목, 키워드 등)를 추가하면 검색 정확도 향상에 효과적
- Precision/Recall/비용측정은 꼭 수행하며, 지속적인 튜닝이 필수

청크분할 전략은 RAG 성공의 핵심포인트입니다.

- 적절한 크기: 너무 큰 청크는 성능저하 및 비용증가 초래
- 문맥 유지되는 경계설정: 문장, 단락, 코드/표 단위 등 의미 단위 고려
- 오버랩 최적화: 중첩을 통하 Recall 향상, 단 과도한 중복은 리소스 낭비
- 지표 기반 개선 사이클: 작은 규모로 시작 → 성능 측정 → 반복개선이 최선

③ 필터링

카테고리 + 필터링이란?

RAG에서 백터 검색을 수행하기 전에 '메타데이터(category, tag, language, date 등)'을 활용해 검색 범위를 사전에 좁히는 기술입니다.

Azure AI Search, Elasticsearch와 같은 검색 시스템은 구조화 필드기반 필터링을 지원합니다.

1. LLM을 활용하여 질문을 카테고리로 분류
2. 해당 카테고리에 한정된 범위에서 벡터 검색 수행

좀더 쉽게 설명하면 무작정 전체 문서를 대상으로 검색하는 것이 아니라, LLM이 먼저 '질문에 맞는 분야를 추정(분류)'하고 해당 분야(카테고리)의 데이터만 벡터 검색에 사용하는 방식입니다.

단계	처리내용	목적
① 사용자 쿼리수신	예: "MLB의 역사를 알고 싶다"	-
② 쿼리 분류	LLM 또는 도구 기반으로 '야구'로 분류	검색범위 축소
③ 메타데이터로 필터링	예: category eq '야구'조건을 검색 API에 부여	관련없는 문서 제거
④ 벡터 검색	남은 수천-수만건 중에서 근사 최근접 검색(ANN) 수행	Recall과 속도균형 유지
⑤ RAG파이프라인으로 전달	관련 청크를 LLM에 입력	최적의 컨텍스트 구성

필터링 효과

지표	필터없음	필터있음	개선이유
검색속도	100ms	30ms	문서 후보수를 사전에 줄이기 위해
비용(Azure AI Search 호출수)	1.0x	0.3x	토큰수 및 I/O감소
응답 정확도 (Precision@10)	0.65	0.83	무관심 카테고리가 제거되어 할루시네이션 감소

보완전력 및 메타 필드 설계

- ✓ 폴백 전략(Fallback Strategy)

"검색이 맞지 않을 때, 어떻게 점진적으로 검색범위를 넓혀갈지를 사전에 정의하는 절차"

레벨	조건	예시	목적
Level0 Strict Filter	필터적용 + 벡터검색	Category eq '야구' and year > 2020	가장 빠르고 비용 효율적으로 정답찾기
Level1 Relax Filter	필터를 하나 완화	Category eq '야구'만 사용	Recall(재현율)약간 회복

Level2 No Filter + Vector	필터 해제후 벡터 검색만 수행	전체 문서 벡터 검색	검색누락방지
Level3 Lexical Search	키워드기반 또는 BM25 검색으로 전환	'MLB 역사' 키워드 검색	롱테일 질문대응
Level4 LLM Q&A	외부 검색/검색없이 직접 생성	답없음 시 웹검색 또는 LLM단독 생성	최종 백업수단

Level0이 기본 전략(속도 및 비용 최적화), 검색이 실패하거나 불충분할 경우, 단계적으로 조건을 완화, 마지막 Level4는 검색결과가 없을 경우 생성형 AI로 전화하는 최후 수단으로 실서비스에서는 각 단계에 따른 트리거 조건과 비용 및 통제 전략 설정이 중요합니다.

구현시 고려사항

1. 임계값 설정(Threshold 설정)
   A. top_k 결과가 0건이거나, 상위결과의 스코어가 임계값(threshod)미만이면 다음단계(Fallback Level)로 자동전환
2. 타임아웃/비용관리
   A. Fallback Level이 올라갈수록 비용과 응답 지연이 증가하기 때문에 각 레벨에 대해 '시간 제한(Timeout)'이나 비용상한선을 사전에 설정해야 함
      i. Level 0~1은 300ms 이하
      ii. Level 3ㅇㅣ상은 API호출 횟수 제한
3. 사용자 안내 메시지
   A. 모든 레벨에서 답변을 생성하지 못하는 경우, '관련된 문서를 찾을 수 없습니다'등 명확한 메시지를 사용자에게 반환함

메타 필드 설계(Metadata Field Design)

'어떤 메타데이터를 어떤 타입으로 어떻게 인덱싱할 것인가?'를 사전에 정의하는 것이 중요합니다.

체크항목	우수사례	예시
필드 선정	검색 축이 될 속성을 선별	category, language, year, owner, securityLevel
데이터 타입	문자열 / 숫자 / 날짜 등을 정확히 지정	year → Edm.Int32,

		publishDate → Edm.DateTimeOffset
속성 설정	filterable, sortable, facetable 등 필요한 최소만 설정	ategory = filterable + facetable title = searchable
정규화	케밥 케이스(-)나 스네이크 케이스(_)로 통일하여 철자 혼란 방지	security_level 등
계층 태그	"sports > baseball > MLB" 형식으로 저장하고 prefix 검색 활용	startswith(category, 'sports/baseball')
보안	ACL / 테넌트 ID 등도 메타필드로 포함해 검색 시 반드시 필터링	tenantId eq 'xyz'
계산 필드	토큰 수, 청크 ID 등은 인덱싱 시 미리 추가	tokenLength로 장문 제외
다중값 필드	태그 등은 Collection(Edm.String)으로 정의해 OR 조건 최적화	tags = ["MLB", "history", "US"]

설계 플로우 (메타필드 + Fallback 전략 통합 관점)

1. 설계절차

1. 유스케이스 → 쿼리패턴 나열
2. 각 쿼리 패턴에 필요한 필터 조건 추출
3. 사용하는 검색엔진의 제약사항확인
   A. 인덱스당 필드 수 제한
   B. 지원 데이터 유형 등
4. Filterable속성은 최소한으로만 적용
   A. 너무 많이 지정하면 인덱스가 커짐
5. 실제 데이터를 이용해 스키마/쿼리 성능 테스트
   A. 이후 단계적으로 필드 구조 조정

2. 핵심요약

- **Fallback 전략**
  - '검색 실패'를 대비한 보험계층
  - 각 레벨별로 속도 ↔ 정확도 ↔ 비용을 점진적으로 트레이드오프
- **메타필드 설계**

- 필터링 성능의 핵심
- 검색측 선정, 데이터유형 지정, 속성(filterable등)설정을 정밀히 설계
- 과도한 filterable설정은 인덱스 부하 유발 → "최소구성 + 향후 확장 가능성"이 중요

Fallback 전략 + 메타필드 설계를 세트로 함께 고려함으로써 빠르고 정확하며 견고한 RAG검색기반을 구축할 수 있습니다.

활용포인트 및 우수사례 (Filtering + Metadata 설계)

언제 사용하면 좋을까?

- 도메인 다양한 지식베이스에서 매우 적합
  - 예: 스포츠정보, 기술제품 카달로그, 사내문서 등
- 시계열/국가별 등 복수 기준으로 필터링이 필요한 경우
  - 예: category eq 'finance' and year gt 2020
- 분류 신뢰도가 낮은 경우
  - 필터 검색에서 결과가 0건이면 전체검색으로 폴백(Fallback)
- 적절한 인덱스 설계
  - 사전에 filterable, facetable필드를 정의해 둠
- 정확도 테스트
  - Precision과 Recall 양쪽을 평가해야 함
- 과도한 필터링 주의
  - 너무 세게 필터링하면 Recall이 급격히 떨어질 수 있음

필터링을 적절히 활용하면 RAG시스템의 '정확도(Precision)'와 '속도(성능)'를 크게 향상시킬 수 있습니다. 그리고 검색 전 단계에서 메타데이터 기반 필터링을 수행하는 구조는 성공적인 RAG시스템 구축을 위한 핵심 전략입니다.

실무 운영환경에서는 반드시 폴백 전략 + 메타필드 설계를 세트로 통합설계해야 제대로 작동합니다.

2-3. Retrieve 정밀도 향상 (검색)

RAG에서 검색에는 다음 5가지 대표적인 방식이 있으며, 번호가 올라갈수록 정확도는 높아지지만 비용과 연산량도 증가합니다.

#	검색방식	구성요소	설명
①	플텍스트 검색	텍스트 검색	키워드일치만으로 결과정렬
②	벡터검색	의미 벡터 기반 검색	문장의 의미를 벡터로 변환해 유사도 검색
③	하이브리드 검색	텍스트 검색 + 벡터 검색	키워드 점수 및 의미 유사도 점수 합산
④	시맨틱 검색	텍스트 검색 + 시맨틱 랭커	키워드로 검색후, BERT등으로 재정렬
⑤	시맨틱 하이브리드 검색	텍스트 검색 + 벡터 검색 + 시맨틱 랭커	하이브리드 검색 후, 시맨틱 랭커로 최종 재정렬

① → ⑤로 갈수록 정확도는 높아지지만, 처리비용과 연산부하는 증가합니다. 상황에 따라 정확도와 성능(속도 및 비용)' 균형을 고려하여 선택이 필요합니다.

전체 텍스트 검색 (Full-Text Search)

전체 텍스트 검색이란 입력된 쿼리와 문서의 문자열을 비고하여 관련성 점수를 부여하고 그 점수에 따라 정렬된 결과를 반환하는 방식입니다.

내부 처리 흐름

- 전체 텍스트 분석(Tokenization)
  - '공백, 구두점(쉼표, 마침표 등)'으로 문장을 나눔
  - 소문자화, 어간 추출(stemming), 정규화 등 전처리 수행
- 인덱싱
  - 각 문서를 구성하는 단어(토큰)들을 기준으로 역색인(Inverted Index) 생성
  - 하나의 문서 → 여러 단어에 대한 참조로 구성됨
- 검색 및 점수화
  - 사용자 쿼리의 토큰들과 인덱스를 비교, '일치 정도(정확도, 빈도 등)'를 기반으로 점수(스코어)를 계산하며 관련도 순으로 정렬하여 결과 반환합니다.

전체 텍스트 검색에서는 단순 매칭 외에도 다음과 같은 스코어링 기준이 존재합니다.

지표	의미	공식
TF (Term Frequency)	해당 문서에서 단어가 몇 번 등장했는가	tf = 단어의 등장 횟수
DF (Document Frequency)	단어가 등장한 문서의 수	df = 단어를 포함한 문서 수
IDF (Inverse Document Frequency)	드물게 등장하는 단어일수록 가치가 높음	idf = log(N / df) (N = 전체 문서 수)
TF-IDF	TF와 IDF의 곱으로 단어의 중요도 가중치 계산	score = tf × idf
BM25	TF-IDF를 개선한 알고리즘: 문서 길이 보정 등 포함	score = Σ idf × ((tf·(k+1)) / (tf + k·(1 - b + b·(docLen / avgLen)))) k, b는 조정 파라미터

BM25는 Azure AI Search/OpenSearch/Lucene 등에서 기본 검색 알고리즘입니다.

긴 문서일수록 특정 단어가 많이 등장할 가능성이 높기 때문에, BM25는 문서길이(docLen)를 보정하여 단어 출현빈도를 과대평가하지 않도록 설계되어 있습니다.

동작 이미지 예시

1. 검색쿼리 실행: 검색쿼리 "RAG 가이드는 생성형 AI 앱 개발에 도전하는 분들께 제공"
2. 텍스트 전처리 → 토큰 리스트로 변환
   A. ["RAG", "가이드", "생성", "AI", "앱", "개발", "도전", "여러분", "제공"]
3. 인덱스 및 매칭
   A. 문서①: "... rag 가이드 우수사례 ..."
   B. 문서②: "…. ai 앱 개발 가이드 ..."
4. TF/DF계산 예시
   A. 문서① 내 rag: tf=2, df=3
   B. 문서②: 내 ai: tf=1, df=4
   C. idf(rag) = log(N/df)=log(3/3)=0
   D. idf(ai) = log(3/4)≈-0.29

E. TF-IDF = (tf×idf) 점수 → 0 − 0.29 + 1.10 ≈ 0.81

5. 스코어 계산 → 정렬 → top_K건 반환 (점수가 높은 순서대로 정렬하여 사용자에게 상위 문서 추천)

전체 텍스트 검색에도 장점이 있습니다.

장점	단점
키워드 정확 일치 검색(예: 코드검색, 로그검색)	유의어나 장문의 의미 기반 검색
Bool연산 (AND/OR/NOT)	단어의 순서변경이나 말바꾸기 표현 대응
부분일치/와일드카드 검색(예: rag*)	모호한 질의 → 이 경우 벡터/시맨틱 검색이 더 유리함

요구사항이 전체 텍스트 검색에 잘 맞는 경우에는 선정해도 좋습니다.

벡터 검색 (Vector Search)

벡터검색 방법은 다음과 같습니다.

쿼리와 문서를 모두 '숫자 벡터'로 변환하고, 벡터 공간에 '얼마나 가까운가'를 계산해서 돌려줍니다 이것이 벡터검색(Vector Search)입니다.

벡터 검색 내부에서는 다음과 같은 처리가 이루어집니다.

- Embedding(수치화): 이미지, 음성, 동영상 및 텍스트를 고정길이 벡터로 인코딩, 예: text-embedding-ada-002, CLIP, Whisper 등
- 인덱싱(Indexing): 벡터를 근사 최근접 탐색(ANN) 구조에 저장하고 대표적인 구조는 HNSW(Hierarchical Navigable Small World), FAISS, ScaNN 등
- 검색(Similarity 계산): 사용자쿼리도 벡터로 변환, 코사인 유사도, 내적(Dot Product), ㅇㅠ클리드 거리등을 활용하여 거리 측정, 전체를 순차 검색하지 않고 빠르게 '근처 후보'만 탐색 → 고속 검색 가능

수치벡터로 변환한 후의 벡터 공간에서 얼마나 가까운지를 나타내는 유사도 함수는 다음과 같습니다.

유사도 함수	수식	특징
코사인 유사도	$1 - \cos\theta = 1 - (u \cdot v / \|u\|\|v\|)$	방향유사도, 길이에 영향을 받지않음
내적(Dot Product)	$u \cdot v$	코사인에 길이 영향도 추가(스코어가 커지는 경향)
유클리드 거리	$\|u-v\|_2$	'거리'로서 직관적, 실장에 의해 제곱근은 생략 가능

참고로

- Azure AI Search, Pinecone, Milvus등 주요 벡터DB는 코사인 유사도와 내적(Dot Product) 모두 지원
- 특히 RAG시스템의 대부분은 코사인 유사도를 기본값으로 사용 → 스케일불변이어서 다루기 용이함

벡터 검색의 동작 이미지

- 데이터소스 → 임베딩 변환
  - PDF, 이미지, 음성등 → 벡터화
  - 예: [-2, -1, 0, 0]
- 인덱싱
  - HNSW 그래프에 벡터를 노드로 등록
  - 노드간 거리(근접성) 정보를 저장
- 검색쿼리 → 임베딩 변환
  - 예: "RAG 가이드를 생성형 AI 앱개발에 도전하는 분들께 제공" → 벡터로 변환 [2, 3, 4, 5]
- ANN (근사최근접 탐색) 실행: HNSW를 통해 가까운 노드들을 단계적으로 탐색
- 유사한 문서 벡터 반환
- Top_k개의 청크를 RAG파이프라인으로 전달 → LLM이 응답생성에 활용

장점	단점
유의어, 말 바꾸기, 번역 등 의미 기반 일치	정확한 키워드 일치 (예: 특정 용어 검색)
장문 ↔ 요약, 패러프레이즈(의미만 같은 표현) 판별	AND / OR 등 논리 연산 검색
이미지 → 텍스트, 음성 → 텍스트 등 크로스모달 검색	정규표현식이나 부분 문자열 검색 ("RAG*" 등)
의미 유사도 기반의 클러스터링, 추천 시스템	특정 빈출어구 또는 고정된 표현만 찾는 경우

이 검색방법도 요건에 맞춰 이용합니다.

벡터 검색의 튜닝 포인트

벡터 검색에서 튜닝에 대해 다음 포인트를 체크합ㅂ니다.

- 검색 알고리즘 선택

알고리즘	특징	언제 선택할까?
KNN(정확 검색)	전체 데이터를 순차 검색 → 정확도는 최고 하지만 속도 느림 / 메모리·비용 많이 소모	데이터셋이 매우 작을 때 또는 PoC, 실험용 "정답 비교" 목적
HNSW(근사 최근접 탐색)	사전에 **근접 그래프 (HNSW Graph)**를 생성해 빠르게 근사 검색 수행 속도와 정확도 균형 우수	실서비스에서는 사실상 필수 대부분의 벡터 DB 기본값

실제 운영 환경에서는 대부분 HNSW가 기본 선택입니다. KNN은 정확하지만 비효율적이며, 실무에서는 PoC 수준에서만 활용됩니다.

파라미터	권장범위	역할/효과	트레이드오프
M	4 ~ 10	각 노드가 연결하는 이웃 노드 수	↑ 크면 정확도↑ / 인덱스 크기↑ / 속도↓

efConstruction	100 ~ 1000	인덱스 구축 시 근접 이웃 후보 수	↑ 크면 정확도↑ / 인덱싱 시간↑ / 메모리↑
efSearch	100 ~ 1000	검색 시 탐색 후보 수	↑ 크면 정확도↑ / 쿼리 속도↓

정확도가 부족하다면, efSearch → M → efConstruction 순서로 값을 점진적으로 증가시킵니다. 지연시간(레이턴시)이 문제일 경우, 우선은 efSearh값을 낮춰서 속도를 확보합니다.

매트릭	수식	장점	단점	주요사례
코사인 유사도 (Cosine Similarity)	$1 - \cos\theta = 1 - (u \cdot v / \|\|u\|\|\|\|v\|\|)$	- '벡터 방향(각도)'만 비교해 스케일 불변 - 임베딩 모델과 잘 맞음	- 스코어를 그대로 관련도 점수로 쓰기 어려움	- 자연어 기반 RAG - 이미지 검색 - Azure OpenAI 임베딩 기본값
내적 (Dot Product)	$u \cdot v$	- 벡터의 크기 정보까지 포함 - 추천 시스템에서 점수 직접 활용 가능	- 문서가 길수록 스코어가 커지기 쉬움	- 레코멘드 시스템 - 클릭률(CTR)/랭킹 최적화
유클리드 거리 (Euclidean Distance)	$\|\|u - v\|\|_2$	- 값이 직관적 (거리 0이면 완전 일치) - 수치 특성 데이터와 잘 맞음	- 고차원에서는 거리 분산이 작아짐 (허브니스 문제)	- 수치 기반 클러스터링 - 로그 패턴 유사도 검색 (저차원)

Azure OpenAI 임베딩(text-embedding-ada-002)은 코사인 유사도 사용을 전제로 정규화된 벡터를 반환함 → 내적 또는 유클리드 거리 사용시, 직접 정규화하지 않으면 정확도 저하 가능

내적이나 유클리드 거리 기반 검색을 원할 경우,

벡터를 직접 L2 정규화하거나 오픈소스 임베딩 모델(예: E5, BGE, Instructor, S-BERT 등)사용을 고려함

임베딩 모델을 비교합니다.

모델	차원수	정확도	추론비용/속도	특이사항
text-embedding-3-large	5,120	★★★★★	★★☆☆☆ (비용높음, 약간 느림)	최고 정확도, 최대 2,000 토큰 입력 가능
text-embedding-3-small	3,072	★★★★☆	★★★☆☆	중소형 RAG에 적합한 균형형 모델
text-embedding-ada-002	1,536	★★★☆☆	★★★★☆	저비용, 최대 8K 토큰 대량 청크 처리에 강함
multilingual-e5-large (OSS)	1,024	★★★★☆	★★★☆☆	100개 이상 다국어 지원, OSS로 커스터마이징 가능
distiluse-base-multilingual (OSS)	512	★★☆☆☆	★★★★★	초경량 다국어 모델, 실시간 API에 적합

선택가이드

- 정확도 최우선: text-embedding-3-large
- 비용·속도 중시 text-embedding-ada-002
- 다국어 or 온프레미스 환경 multilingual-e5, distiluse-base-multilingual
- 사내 GPU 클러스터 활용 가능 OSS 모델을 파인튜닝해서 자체 배포 고려 가능

임베딩 기반 RAG 튜닝 순서

페이즈	주요 작업	평가지표	Go/No-Go판단기준
① 베이스라인 구축	- text-embedding-3-small + HNSW + cosine - efSearch = 100 - 청크 크기 = 512 tok (15% 오버랩)	Recall@50, p95 지연시간, 1만건당 비용	지표가 기준 이상이면 다음 단계로 진행
② 정확도 개선	- efSearch: 300 → 500으로 단계적 증가 - m: 4 → 8로 조정 - 필요 시 모델을 small → large로 변경	Recall, Precision, 사용자 평가	성능 향상 or 비용 허용 범위면 설정 확정 (Fix)
③ 속도 / 비용 최적	- efSearch: 300 → 150	Latency < 200ms, 인	속도 ↑ ↔ 정확도 ↓

화	으로 감소시켜 P99 latency 측정 - 인덱스 샤딩, 캐시 도입 등 적용	프라 비용	트레이드오프 판단
④ 다국어 / 도메인 특화 대응	- 영어 외 언어 지원 필요 시 → e5-large 도입 - 특정 업종/도메인 → ada-002 파인튜닝 활용	다국어 Recall, 도메인 특화 BLEU 등	OSS or Fine-tuning 성과 > 기존 baseline 이면 적용
⑤ 운영 및 모니터링	- 지표 대시보드 운영: Recall@50, Latency, 비용 등 - efSearch, 모델 버전 전환을 Feature Flag로 관리	주간 KPI, 알림 기준치 설정	이상 탐지 시 자동 롤백 or 재튜닝

임베딩 운영 Tips

1. 하나씩만 바꾸며 A/B테스트: 동시에 여러 파리미터를 조정하지 말고, 1개씩 조정 후 성능 비교
2. 벡터 차원수 ↔ 인덱스 크기 비례: 벡터 차원이 커질수록 인덱스 용량도 커지므로, 스토리지 용량 예측 필수
3. 모델을 변경하면 반드시 재인덱싱 필요: 서비스 중단 방지를 위해 블루/그린(이중 시스템) 도입 권장

임베딩 모델은 아래 4가지 기준에서 선택하고 튜닝은 ①베이스라인 → ②정확도 향상 → ③속도·비용 최적화 순서로 진행하는 것이 가장 효율적입니다:

- 정확도 (Precision / Recall)
- 차원 수 (임베딩 벡터 길이)
- 다국어 지원 여부 (Multilingual)
- 비용 및 추론 속도

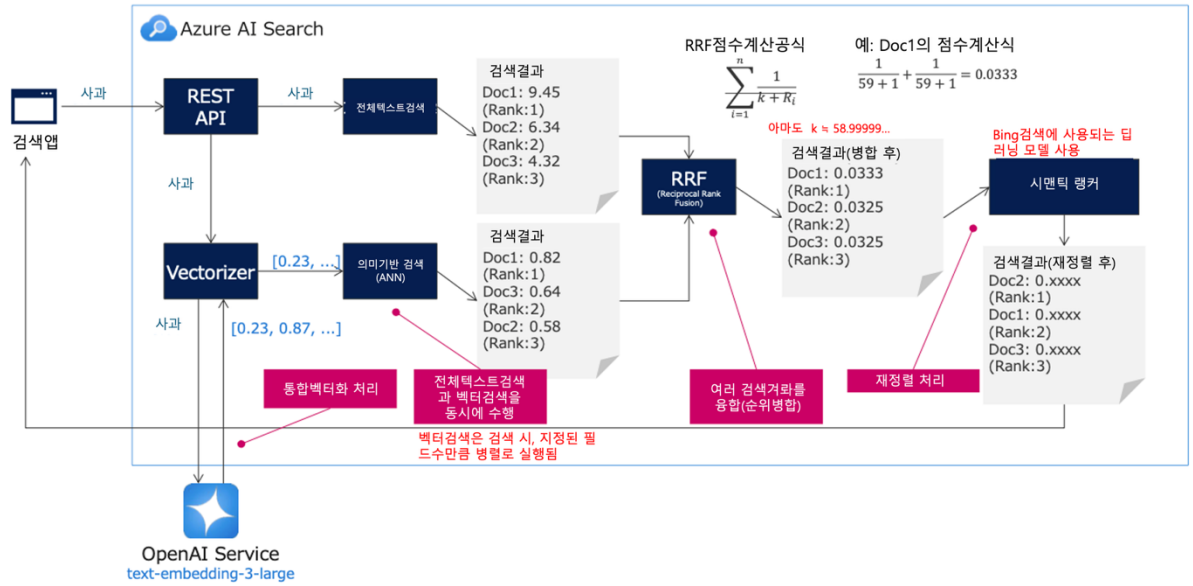
시맨틱 하이브리드 검색(Semantic Hybrid Search)

시맨틱 하이브리드 검색이란, 다음 3가지를 종합적으로 사용하는 방식입니다. 텍스트 검색, 벡터 검색, 시맨틱 랭커(BERT 기반 재정렬 모델) 3계층을 결합함으로써 Recall(재현율), Precision(정확도), Latency(속도)를 균형있게 향상시킬 수 있습니다.

계층	역할	대표모델
① 예비 필터 (키워드 기반)	- 불 연산(AND/OR/NOT), BM25, 와일드카드 - 메타데이터 필터링 예: category eq 'AI'	Lucene, Azure AI Search Text, OpenSearch
② 의미 기반 검색 (벡터)	- 쿼리 및 문서를 임베딩(벡터화) - HNSW 등 근사 최근접 탐색으로 후보 100~300건 검색	Azure AI Search Vector, Pinecone, Milvus
③ 세맨틱 랭커 (재정렬기)	- 후보 결과를 BERT 또는 Cross-Encoder로 재스코어링 - 최종 상위 10~20건을 결과로 출력	Azure Semantic Ranker, Cohere Rerank, OSS cross-encoder/ms-marco-*

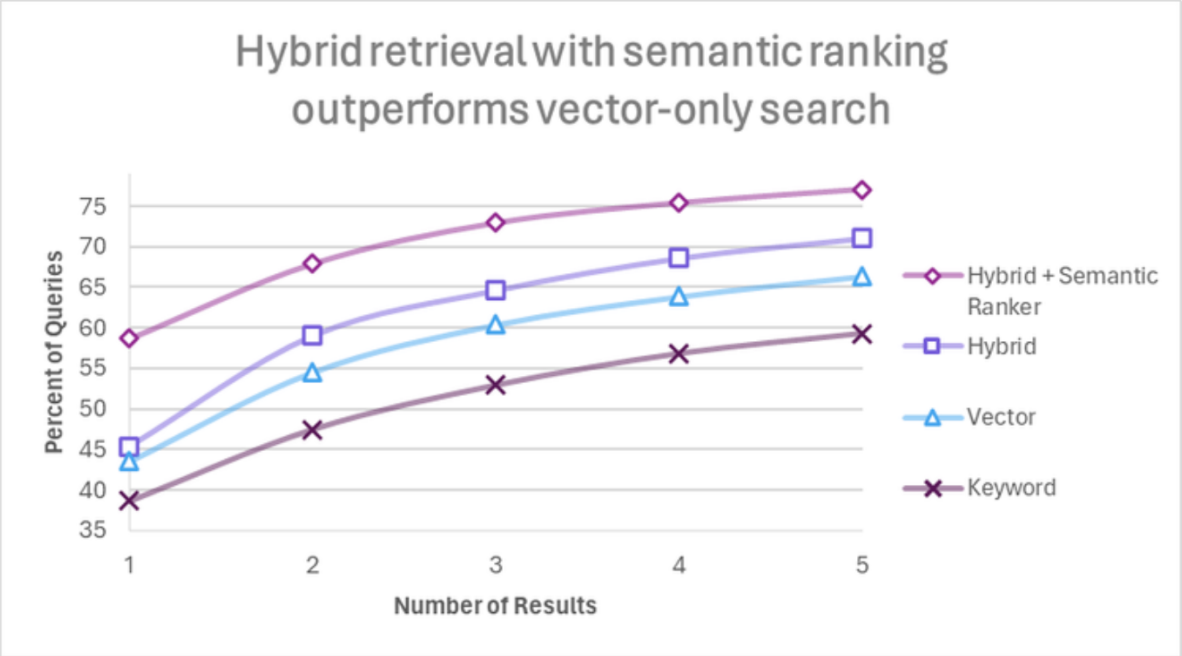
시멘틱 하이브리드 검색 동작 순서

① 쿼리입력: 'RAG 가이드 청크 분할 이유'
② Keyword Search: Lucene/Azure Search 기반 BM25 키워드 검색 수행
③ Vector Search: 쿼리를 벡터로 임베딩, FAISS, Pinecone등 의미유사, 동의어, 부분일치등을 커버
④ 결과 통합 및 중복제거: BM25결과 + 벡터검색결과를 병합
⑤ Semantic Re-ranking: BERT기반 크로스엔코더 모델 사용
⑥ Top-K 결과 반환: 검색UI에 반환 하거나 RAG파이프라인에 컨텍스트로 전달



관점	기존방식	시만틱 하이브리드 방식
Recall(재현율)	키워드 검색만 사용 → 어형 변화에 약함	벡터 검색이 동의어 및 의미 유사어를 보완
Precision(정밀도)	벡터만 사용 시 → 노이즈 문서 포함 위험	키워드로 정확히 일치시키고, 랭커로 오탐 제거
속도	키워드 ◎ / 벡터 ○ / BERT △ → BERT 느림	키워드로 선 필터링 → BERT는 소량에만 적용
비용	벡터 + BERT를 전체 문서에 적용 시 비용 과다	후보 수백 건만 재정렬하므로 저비용 운영 가능

정확도 + 재현율 + 속도 + 비용 이 4가지를 모두 만족하는 유일한 방법이 바로 '키워드 검색 + 벡터검색 + 시맨틱 랭커' 조합입니다.



2-4. Augment 정밀도 향상 (확장)

Prompt Engineering 개선포인트

RAG에서는 프롬프트 엔지니어링이 중요한 포인트가 됩니다.

사용자 메시지 구성	검색 결과를 사용자 프롬프트에 어떤 형식으로 포함할지를 설계해야 함 예: 문장1: ~ 문장2: ~ → 명확한 컨텍스트 제공
검색쿼리 생성	- 키워드 검색용 쿼리: OpenAI를 통해 Azure AI Search에 적합한 키워리치 쿼리 생성 - 벡터 검색용 쿼리: HyDE(Hypothetical Document Embedding), 쿼리 확장(Query Expansion), Query-Rewrite 사용
질문 보강 및 피드백 설계	사용자 입력에 검색에 필요한 정보가 부족할 경우, 시스템 프롬프트에 '추가 질문 유도'를 명시해 재질문하도록 구성
Function Calling 활용	질문의 목적이나 도메인에 따라 검색 인덱스를 자동 전환해야 할 경우, Function Calling을 사용해 "어떤 인덱스를, 어떤 조건으로 검색할지" 결정 및 실행

Prompt Engineering이 필요한 이유

관점	왜 중요한가	예시
품질 향상	LLM은 모호한 지시에 대해 모호한 출력을 낼 가능성이 높음. 의도를 명확히 하면 정확도가 향상됨	"○○에 대해 알려줘"보다 → "○○의 장점과 단점을 각각 3개, 200자 이내로 나열"이 더 정확
제어 가능성	출력 포맷·스타일·길이를 통일시켜 후속 처리 및 UI 연결이 쉬워짐	JSON 포맷, Markdown 목록, 표 형태 등으로 명시
업무 요구사항 적합	도메인 용어, 기업 정책, 금칙어 등을 Prompt에 명시하여 컴플라이언스 확보	반드시 "○○주식회사"를 포함, 특허 내용은 요약만 표시 등
비용 최적화	불필요한 토큰 사용을 줄여 추론 비용과 지연 시간 감소	장황한 인삿말 금지, 짧은 응답 요구 등
안정성(가드레일)	개인정보·유해 표현 방지, 예상 밖 동작 방지	"개인 정보를 출력하지 마세요" 등의 System Prompt 사용
모델 한계 보완	Chain-of-Thought, RAG 등으로 추론 분해, 검색, 근거 삽입을 유도해 LLM 보완	RAG에서 "아래 문서만 참고하여 답변하세요"라고 명시
재현성 확보	동일 프롬프트 입력 시 동일한 응답 경향 확보로 테스트 및 품질 보증 용이	A/B 테스트, 자동 평가 가능
자동화 / 시스템 연동	Function Calling 등으로 도구 선택 / 파라미터 추출 자동화	사내 KB 검색, SQL 호출 등을 자연어 기반으로 실행 지시 가능

LLM은 암묵적 맥락을 내포한 범용 모델이지만. '무엇을 어떻게 해라'를 언어로 구체적이고 명확하게 지시하지 않으면 원하는 동작이 나오지 않습니다. 프롬프트 엔지니어링은 단순한 지시문이 아니라, UI/UX, 보안, 품질, 비용을 앙우르는 명세서(사양서)역할을 합니다.

Azure OpenAI +| RAG에 적용하는 Prompt 설계는 ① 검색 쿼리 생성 → ② 근거 주입 (Augment) → ③ 응답 생성 함으로써 엔터프라이즈 품질을 달성할 수 있습니다.

2-5. Generation 정확도 향상 (생성)

LLM 모델 선택 정보

- 텍스트 생성 모델 선택

항목	내용
정확도 중시 시	최신 대형 모델 사용 권장 예: gpt-4o, 또는 복잡한 추론/정중한 응답이 필요한 경우 o1
속도 중시 시	경량 모델 사용 가능 예: gpt-4o mini, o3-mini 등
모델 다양성	AI Studio의 모델 카탈로그에서 선택 가능 (예: Llama 3.1, Mistral, Databricks Dolly 등)
중요한 주의사항	"Garbage In, Garbage Out" → 검색(Retrieve)된 문서에 답변에 필요한 정보가 없으면 어떤 모델도 정확한 답변 불가 👉 검색 정확도(Retrieval precision)가 무엇보다 중요

- 임베딩 모델 선택 (벡터 검색 정확도에 영향)

항목	내용
기본 추천	text-embedding-3-large 사용 권장 → OpenAI의 최신 임베딩 모델 (5,120차원) → 성능 최상급이지만 비용은 매우 저렴
다국어 지원 필요 시	Cohere-embed-v3-multilingual (AI Studio 카탈로그에서 선택 가능)
기타 옵션	- OSS 임베딩 모델을 Azure ML 상에서 직접 실행 - OSS 임베딩 모델을 파인튜닝하여 커스터마이징

LLM유형별 특성 비교: Inference cs Reasoning

항목	Inference 계열 (예: GPT-4o / GPT-3.5 등)	Reasoning 계열 (예: o1 / o3 / o4-mini 등)
주요 목적	빠르고 저렴한 범용 생성용 → 번역, 요약, 이메일, 일반 대화	복잡한 논리 추론, 멀티스텝 문제 해결 → 수학, 코드, 과학적 사고, 전략 기획 등
사고 방식	1패스 → 즉답 스타일 → Chain-of-Thought(CoT)는 내부적으로 최소화	"생각하고 답하는" 스타일 → 내부/외부 CoT 사용, 필요 시 reasoning effort 증가
속도 / 비용	빠름, 저렴 (낮은 토큰 비용)	느림, 비쌈 (추론 계산량 많음)

		→ 동일 토큰 수라도 3~30배 비쌀 수 있음
정확도 경향	턴 인식 기반 질문, 일상적 작업에 강함 → 예: 이메일 초안, 블로그/SNS 작성	수식, 알고리즘, 계획 수립 등에 탁월 → CoT 요구되는 문제에서 압도적 성능
기능	멀티모달(GPT-4o), 긴 컨텍스트, Function Calling	동일 기능 + reasoning_effort 조절 Parallel Tool Calling 최적화됨
대표 모델	GPT-4o, GPT-4.1, GPT-4o-mini, GPT-3.5	o1, o1-mini, o3, o3-mini, o4-mini
벤치마크	- MMLU: 85~90 - AIME: ~10%	- AIME: 최대 74% (o1) - GPQA 등에서 GPT-4o 대비 우위
대표 활용 예시	- FAQ 챗봇 - 콘텐츠 생성 - 빠른 챗UI용	- 수학 최적화 - 코드 자동 수정 / 테스트 생성 - 사내 보고서 자동화(근거 포함 전략 출력 등)

모델 선택 가이드라인

1. 기본은 Inference 모델로 시작

- 속도/비용 우선인 경우: 프론트엔드용 챗봇, RAG 응답생성 단계
- 일반적인 질문 텍스트 요약/작성에 충분한 품질 제공
- 예시: GPT-4o, GPT-4o-mini, GPT-3.5-turbo
- 즉시 쓸만한 답변을 빠르게 받고 싶다면: Inference

2. 다음과 같은 경우 Reasoning 모델로 전환

- 정확도가 부족하거나 논리적 오류(비약)가 많을 경우: o1, o3등으로 스위칭
- 수학증명, 회계로직, 복잡한 코드 생성 등 "틀리면 치명적인" 도메인: Reasoning 모델을 초기에 선택
- 멀티스텝 추론/도구호출을 에이전트에게 전적으로 맡길 경우: o3, o4-mini같은 모델이 유리

3. 하이브리드 전략: 분업구조

- Planner역은 o3(Reasoning)의 절차설계, API파라미터 생성, 질의분해
- Executor역은 GPT-4o-mini(Inference)로 응답 생성, 문장정리, 프롬프트 형식화
- Azure에서는 단일 API내에서 모델 스위칭 가능하고, model_router를 사용하면

동적 라우팅 구성도 가능합니다.

3. Evaluate (평가)

RAG의 평가에서는 크게 나누어 2종류의 정밀도 평가가 가능합니다.

- 검색정확도 평가(Store ~ Retrieve 위치평가)
- 답변정확도 평가(Store ~ Generation 전체평가)

각 항목의 평가방법에 대해 설명합니다.

검색정확도

검색 정밀도에 대해서는 평가용 데이터 준비, 평가 지표 설정이 필요합니다.

아래에 관측 포인트를 정리합니다.

- 평가용 데이터 준비
  - 검색 쿼리, 검색 문서 ID, 관련도 목록을 제공합니다(100개~).
  - 검색 문서는 검색 엔진에 저장되어 있기 때문에 청크가 될 수 있습니다.
  - 청크 크기가 변경되면 검색 문서 ID도 변경되므로 해당 위치의 정규식을 사용하여 평가하는 방법도 생각합니다.
- 평가 지표
  - Hit Rate (HitRate @ N) : 관련 문서가 1 개이더라도 상위 N 위치에 순위가 매겨진 비율
  - 평균 역수 순위 (MRR @ N) : 검색 결과에서 정답이 처음 나타나는 순위의 역수 평균
  - 평균 적합률 (mAP @ N) : 여러 검색 쿼리에 대한 적합률 평균
  - nDCG@N: 검색결과의 순위에 따라 관련성 높은 결과가 상위일수록 높은

점수를 얻는 지표

- 검색 정확도의 중요성
  - 정량적 평가가 가능 (생성 정밀도 평가는 LLM에 의존하는 평가이므로 정성적 평가에 가깝다)
  - 검색 정밀도가 응답 정밀도에 직접 연결되기 때문에 매우 중요한 평가

답변 시스템

응답 정밀도에 대해서도 평가용 데이터와 평가 방법이 필요합니다.

이쪽에 대해서도 관측 포인트를 정리합니다.

- 평가용 데이터 준비
  - [질문], [예상 답변] 목록 준비
  - 평가 처리에서 각 질문에 대한 [생성 된 답변]과 [얻은 문서]
- 정확도 평가 방법
  - ROUNGE: 생성된 텍스트의 품질을 평가하기 위한 메트릭, n-gram 사용
  - BLEU : 기계 번역 및 텍스트 생성 품질을 평가하는 메트릭 (n-gram 사용)
  - BertScore : 생성 문장과 정답 문장의 유사성을 포함하여 평가
  - LLM as a Judge: LLM을 사용하여 생성 문장 평가

LLM as a Judge에서 평가 지표의 주요 사항은 다음과 같습니다.

- 일관성(Consistency)
- 관련성 (Relevance)
- 유창함 (Fluency)
- 커버리지 (Coverage)
- 다양성 (Diversity)
- 상세도(Detail) 등

위와 같은 관점에서 RAG의 정확도를 평가할 수 있습니다.

4. RAG 및 파인튜닝

자주 묻는 질문에는 RAG와 Fine-Tuning의 차이가 있습니다.

RAG (Retrieval-Augmented Generation, 검색증강생성)와 Fine-Tuning(파인튜닝)은 모두 LLM의 정확성을 향상시키는 방법이지만 접근 방식이 다릅니다.

RAG는 외부 지식 기반 및 문서를 활용하여 생성하는 답변의 정확성을 향상시키는 방법입니다. 구체적으로는 다음과 같은 특징이 있습니다.

- 외부 지식 활용: RAG는 미리 준비된 문서와 지식 기반에서 정보를 검색하고 이를 바탕으로 답변을 생성합니다.
- 동적 정보 획득: RAG는 사용자의 질문에 따라 실시간으로 관련 정보를 검색하고 답변에 통합할 수 있습니다.

Fine-Tuning은 기존 LLM을 특정 작업 및 도메인에 적응시키는 방법입니다. 구체적으로는 다음과 같은 특징이 있습니다.

- 모델 적응 : Fine-Tuning은 특정 작업 및 도메인에 대해 모델을 다시 학습하여 모델의 매개 변수를 조정하여 특정 작업의 정확성을 향상시킬 수 있습니다.
- 데이터세트의 필요성: Fine-Tuning에는 특정 작업 및 도메인과 관련된 많은 양의 교육 데이터가 필요합니다.
- 정적 지식 강화: Fine-Tuning은 특정 도메인이나 작업에 특화된 지식을 모델에 통합할 수 있지만 RAG와 같이 실시간으로 정보를 얻을 수는 없습니다.
- 모델 재학습: Fine-Tuning은 기존 모델을 재학습하므로 계산 리소스와 시간이 필요합니다.
- 특정 작업 전문: Fine-Tuning은 특정 작업 및 도메인에 대한 모델을 최적화하므로 다용도가 낮을 수 있습니다.
- 모델 크기: Fine-Tuning은 모델의 크기를 늘릴 수 있으므로 더 많은 매개 변수를 가진 모델을 사용할 수 있습니다.
- 모델 업데이트: Fine-Tuning은 특정 작업 및 도메인에 대한 모델을 업데이트하므로 모델 버전 관리가 필요합니다.
- 모델 재사용성: Fine-Tuning은 특정 작업이나 도메인에 대한 모델을 최적화하기 때문에 다른 작업이나 도메인에 재사용하기 어려울 수 있습니다.

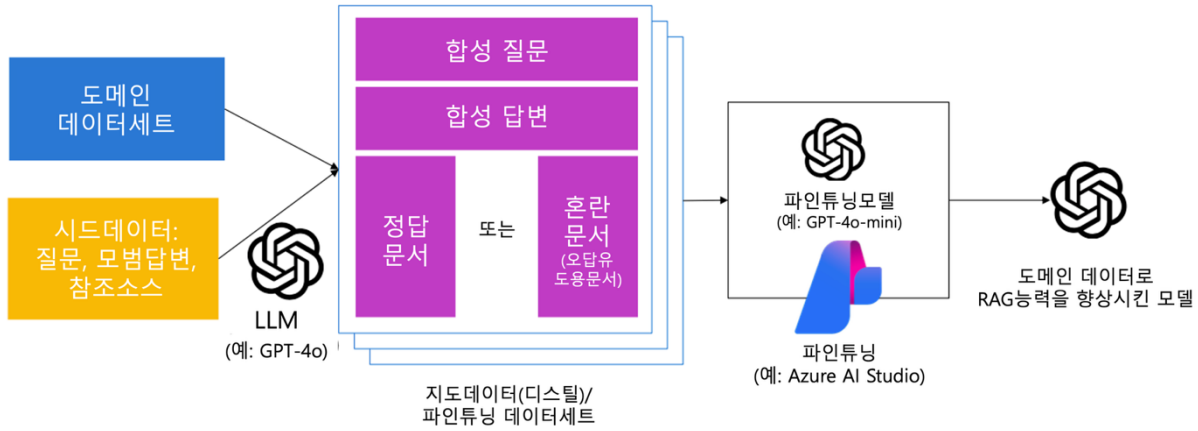
RAG와 Fine-Tuning의 차이를 정리하면 다음과 같습니다.

관점	파인튜닝(API기반)	RAG(검색증강생성)
추천 용도	① 출력 형식 / 말투 조정 ② 특정 태스크의 정확도 향상 ③ 토큰 절약 (완전 튜닝 시 어휘 확장도 가능)	✅ 지식 추가 또는 로직 삽입
비용	① GPU 학습 시간에 따른 비용 ② 튜닝된 모델 전용 엔드포인트 실행 비용	① 검색엔진 이용 요금 ② 검색 문맥 추가에 따른 요청별 토큰 비용 증가
생성 속도 영향	입력 토큰이 줄어들기 때문에 속도에 거의 영향 없음	검색 수행 + 프롬프트 확장으로 인해 Fine-tuning보다 느림
데이터 반영 시간	데이터 크기에 따라 수 분 ~ 수 시간 학습 필요	데이터 업로드 후 즉시 반영 가능
리소스 준비	GPU가 필요하므로 사용 가능 리전 제한	검색엔진은 다중 리전 대응, 인프라 확보 쉬움
필요 기술 스택	- 신경망 학습 이해 - 학습용 데이터 설계 및 품질 관리 능력	- 청크 설계, 벡터 검색, 프롬프트 설계 능력

RAFT(Retrieval Augmented Fine-Tuning) 기술

RAFT (Retrieval Augmented Fine Tuning)는 RAG 접근법을 Fine-Tuning과 결합하는 기술입니다. RAFT의 특징은 다음과 같습니다.

- RAG 활용: RAFT는 RAG의 Retrieve 단계를 활용하여 외부 지식 기반 및 문서에서 정보를 검색합니다.
- Fine-Tuning 적용: 검색된 정보를 기반으로 모델을 Fine-Tuning하여 특정 작업 및 도메인에 대한 정확성을 향상시킵니다.
- 동적 정보 획득 및 적응: RAFT는 사용자의 질문에 따라 실시간으로 관련 정보를 검색하고 해당 정보를 기반으로 모델을 Fine-Tuning합니다.



관련 URL: https://techcommunity.microsoft.com/blog/aiplatformblog/retrieval-augmented-fine-tuning-use-gpt-4o-to-fine-tune-gpt-4o-mini-for-domain-s/4248861

간단하게 말하면, RAG의 Retrieve단계에서 정보를 얻고 해당 정보를 기반으로 모델을 파인튜닝(Fine-Tuning)해서 특정 작업 및 도메인에 대한 정확성을 향상시키는 방법입니다.

RAG와 CAG

CAG(Cache-Augmented Generation)이란?

CAG는 RAG의 검색 단계를 생략하고, 사전로드된 문성들과 KV-cache만으로 생성하며 검색없이 컨텍스트 윈도우에 문서를 미리 넣고 KV캐시를 사전준비합니다. 지식이 사전에 결정되어 있고 실시간 외부 검색이 불필요한 환경에서 사용합니다.

항목	RAG	CAG
지식수입방식	실행 시 외부 벡터 DB 등에서 검색하여 문서 전달	사전에 모든 문서를 컨텍스트 + KV 캐시로 로드
속도	검색 시간이 필요 → 상대적으로 느림	검색 생략 → 매우 빠름
컨텍스트 범위	Top-k 검색 결과만 LLM에 입력	전체 문서를 한번에 넣을 수 있음 (단, 윈도우 제약 있음)
장점	최신 정보 검색 가능 도메인 확장성 뛰어남	검색 엔진 불필요 단순 구조, 지연 시간 최소
단점	검색 품질이 정확도에 직결 시스템 복잡	문서가 많을 경우 컨텍스트 초과, 동적 질의 대응 어려움

RAG + 파인튜닝 + CAG 활용전략

- 자주하는 질문(FAQ), 답변일관성: Fine-Tuning + CAG
- 실시간 문서 응답, 변화많은 정보: RAG
- 구조단순화 + 빠른 응답속도: CAG
- 정밀한 업무대응 + 문서기반근거 제공: RAG + Fine-Tuning

- RAG = 실시간 검색 기반 동적 생성
- CAG = 사전 캐시 기반 정적/고속 생성
- 파인튜닝 = 응답 일관성과 업무 특화 보장

→ 필요에 따라 이들을 조합하면 가장 강력한 도메인이 됩니다.

관련 URL: https://arxiv.org/abs/2412.15605

5. 정리

이번에 정리한 RAG 관련 지식 외에도 Microsoft에서 RAG에 관한 다양한 인사이트가 공개되어 있습니다. RAG프로젝트에 참여하시는 분이라면 꼭 한번 읽어보시길 권장합니다. 결과물 품질을 높이는데 많은 도움이 됩니다.

관련 URL: https://github.com/microsoft/RAG-Knowledge

또한, 앞서 언급했던 Agentic RAG는 앞으로도 매우 중요한 주제가 될 것으로 예상되며 전반적으로 내용을 파악하고자 하는 분들은 아래 링크를 참고하시기 바랍니다.

