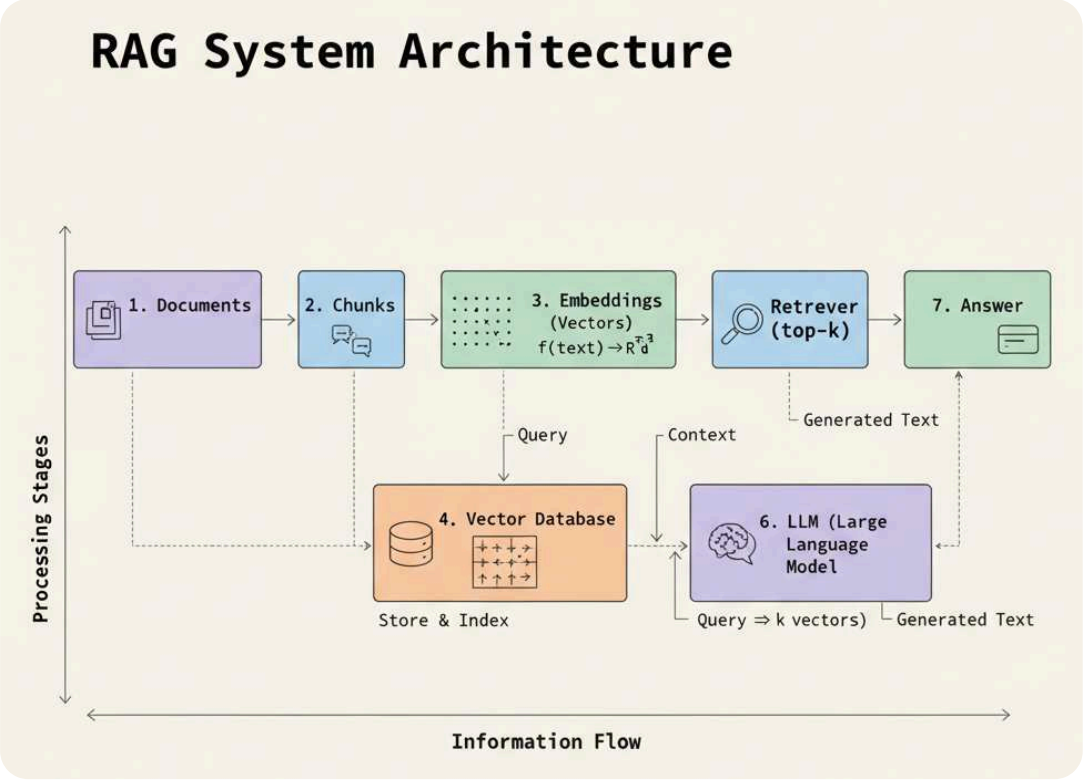


MINI BOOK

생성형 AI 기반 RAG & Agent 입문 과정

OpenAI API · LangChain RAG · 도구 호출 · 에이전트

3일 / 24시간 · 핸즈온 핸드북

젠아이랩스 (GenAI Labs)

저작권 안내

본 교재 **「생성형 AI 기반 RAG & Agent 입문 과정」 미니북**은 교육 목적의 핸즈온 핸드북입니다.

수록된 모든 예제 코드는 본 과정의 실습 저장소에서 발췌하였으며, 학습자는 강의 및 개인 학습 목적의 범위 내에서 자유롭게 실행하고 수정할 수 있습니다.

> **그 외의 무단 복제·재배포·상업적 이용은 엄격히 금지됩니다.**
>
> 본 교재에 수록된 **모든 소스 코드, 교재 내용(텍스트), 이미지(삽화 포함)**에 대한 저작권과 일체의 권리는 **젠아이랩스(GenAI Labs)**에 있습니다. 권리자의 사전 서면 동의 없이 전부 또는 일부를 복제·배포·전송·2차적저작물 작성·상업적으로 이용할 수 없습니다.

LangChain, OpenAI, ChromaDB, Google Gemini 등 본문에 언급된 제품·상표는 각 권리자에게 귀속됩니다.

목차 · Contents

3일 / 24시간 · RAG & Agent 입문

DAY 1 LLM 이해와 RAG 핵심 기술 구현

1일차 목표 LLM의 작동 원리(토큰·임베딩·어텐션)와 OpenAI API 사용법을 익힙니다. 임베딩·코사인 유사도·Top-K 검색을 프레임워크 없이 직접 구현하고, 하루의 끝에는 손으로 만든 RAG QA로 RAG의 뼈대를 체득합니다.

PART 1. LLM과 OpenAI API

PART 2. RAG 핵심 원리를 직접 구현하기

DAY 2 LangChain 기반 RAG 구축

2일차 목표 LangChain과 LCEL로 RAG 파이프라인을 표준화합니다. Document Loader·Text Splitter·임베딩·벡터 DB(Chroma)·Retriever를 조립하고, 환각(Hallucination)을 막고 출처를 다는 RAG 체인을 완성합니다.

PART 3. LangChain 기초

PART 4. 벡터 검색과 RAG 체인

DAY 3 RAG를 넘어 Agent로

3일차 목표 도구 호출(Tool Calling)과 ReAct 루프로 RAG를 에이전트화합니다. RAG 검색을 도구로 등록해 검색 여부를 스스로 판단하는 Agentic RAG를 구현하고, 멀티 도구와 대화 기억을 더해 '도구를 쓰는 RAG 에이전트'로 완성합니다.

PART 5. 에이전트와 도구 호출 기초

PART 6. Agentic RAG와 최종 프로젝트

DAY

1

LLM 이해와 RAG 핵심 기술 구현

원리를 손으로 구현하며 RAG의 뼈대를 세운다

PART 1 LLM과 OpenAI API

PART 2 RAG 핵심 원리를 직접 구현하기

PART 01

LLM과 OpenAI API

1. 생성형 AI 개요와 LLM 발전사
2. 트랜스포머와 임베딩의 직관
3. OpenAI API 기초와 CLI 챗봇
4. 대화형 챗봇: 멀티턴과 Temperature

생성형 AI 개요와 LLM 발전사

> 규칙 기반에서 ChatGPT까지, 언어모델이 어떻게 사람의 말을 다루게 되었는지 그 흐름을 짚습니다.

1. 생성형 AI란 무엇인가

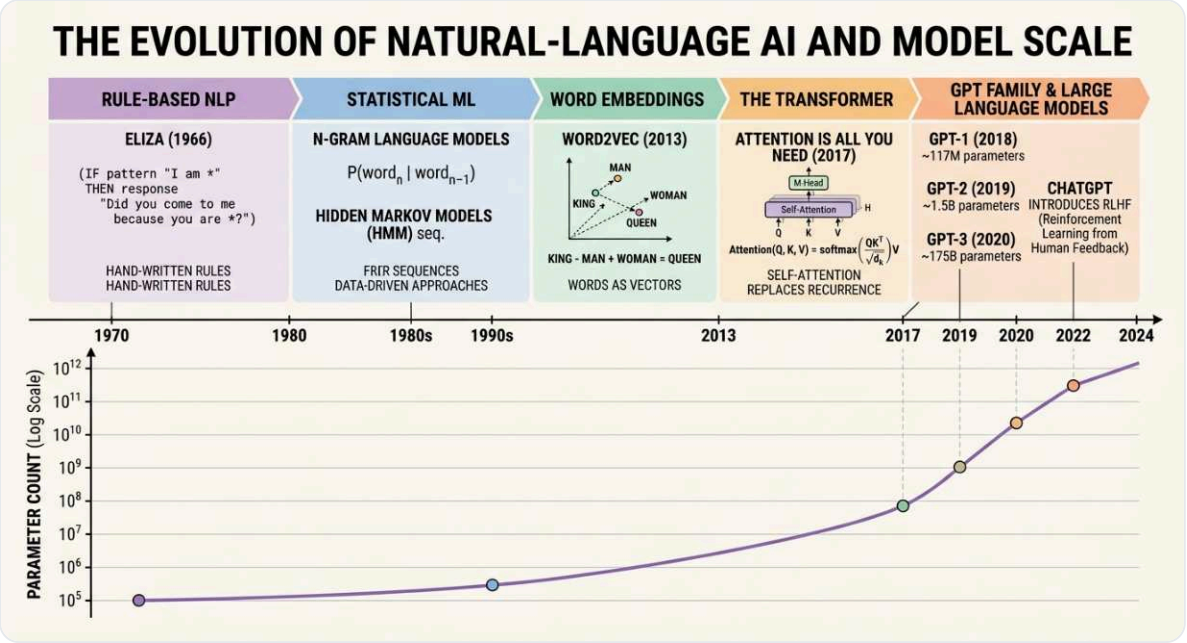
생성형 AI(Generative AI)는 기존 데이터를 분류하거나 점수만 매기는 것이 아니라, 새로운 콘텐츠를 직접 만들어 내는 인공지능입니다. 텍스트, 이미지, 음성, 코드처럼 사람이 만들던 결과물을 모델이 생성합니다. 그중에서 텍스트를 다루는 것이 대규모 언어모델(LLM, Large Language Model)이고, 이 책의 주제입니다.

전통적인 인공지능은 "이 메일은 스팸인가 아닌가"처럼 정답 라벨을 고르는 판별(discriminative) 작업에 강했습니다. 반면 생성형 모델은 "이 질문에 답을 문장으로 써 줘"처럼 출력의 경우의 수가 거의 열려 있는 작업을 수행합니다. 같은 입력에도 매번 조금씩 다른 문장이 나오고, 결과물의 품질을 단일 정답과 비교해 채점하기 어렵습니다. 이 두 가지가 생성형 AI를 다룰 때 늘 따라오는 특징입니다.

> **핵심 포인트:** 생성형 AI는 '정답을 고르는' 모델이 아니라 '그럴듯한 다음 내용을 만들어 내는' 모델입니다. 이 차이가 환각(hallucination)이라는 약점과 RAG라는 보완책으로 이어집니다.

2. 패러다임의 발전사: 규칙 기반에서 LLM까지

언어를 다루는 인공지능은 단번에 지금의 모습이 된 것이 아닙니다. 여러 단계의 패러다임 전환을 거쳤고, 각 단계는 앞선 단계의 한계를 극복하면서 등장했습니다.



인공지능 패러다임의 발전: 규칙 기반 → 통계 기반 머신러닝 → 단어 임베딩 → 트랜스포머 → 대규모 언어모델(파라미터 규모의 급증)

초기의 규칙 기반(rule-based) 시스템은 "이런 단어가 나오면 이렇게 처리하라"는 규칙을 사람이 일일이 작성했습니다. 동작이 명확하고 설명하기 쉬웠지만, 자연어의 무수한 예외와 표현 변형을 사람이 모두 규칙으로 적기란 불가능했습니다.

다음 단계인 머신러닝(machine learning)은 규칙을 사람이 짜는 대신 데이터에서 통계적 패턴을 학습하게 했습니다. 다만 어떤 특징(feature)을 입력으로 쓸지는 여전히 사람이 설계해야 했고, 이 특징 공학(feature engineering)이 성능을 좌우했습니다.

딥러닝(deep learning)은 여러 층의 신경망을 쌓아 특징 추출 자체를 모델이 학습하게 했습니다. 사람이 특징을 설계하지 않아도, 충분한 데이터와 연산만 있으면 모델이 스스로 유용한 표현을 찾아냈습니다. 자연어 처리에서는 단어를 벡터로 바꾸는 임베딩과 순차 데이터를 다루는 RNN 계열 모델이 이 시기에 발전했습니다.

결정적 전환점은 2017년 트랜스포머(Transformer) 구조의 등장입니다. 트랜스포머는 문장을 앞에서부터 순서대로 처리하는 대신, 어텐션(attention) 메커니즘으로 문장 안의 모든 단어 관계를 한 번에 봅니다. 이 구조는 대규모 병렬 학습에 적합해 모델과 데이터의 규모를 크게 키울 수 있게 했고, 그 위에서 GPT 계열의 대규모 언어모델이 자라났습니다.

단계	핵심 아이디어	한계
규칙 기반	사람이 규칙을 직접 작성	예외·변형을 모두 담을 수 없음
머신러닝	데이터에서 통계 패턴 학습	특징 설계를 사람이 해야 함
딥러닝	신경망이 특징까지 학습	긴 문맥·병렬화에 취약(RNN)
트랜스포머	어텐션으로 전체 관계 파악	큰 연산 자원 필요
LLM	대규모 사전학습 + 생성	환각·최신 정보 부재

3. GPT의 작동 원리: 다음 토큰 예측

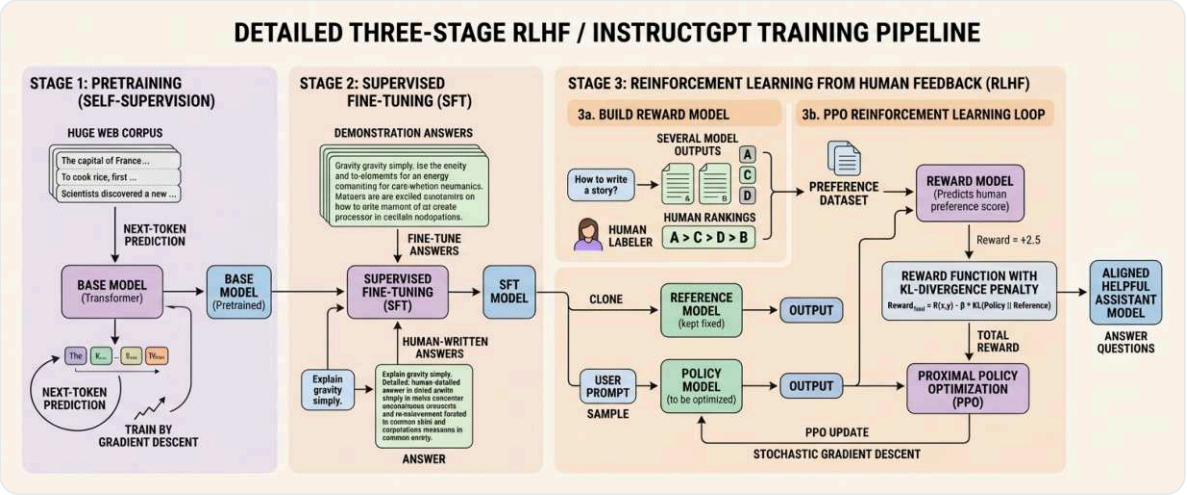
GPT(Generative Pre-trained Transformer)의 작동 원리는 한 문장으로 요약됩니다. 지금까지 주어진 텍스트 다음에 올 가장 그럴듯한 토큰(token)을 예측하는 것입니다. 토큰은 단어보다 작을 수 있는 텍스트 조각이며, 모델은 매 순간 어휘 전체에 대한 확률 분포를 계산해 그중 하나를 골라 이어 붙입니다.

예를 들어 "대한민국의 수도는"이라는 입력이 주어지면, 모델은 다음 토큰 후보 중 "서울"에 높은 확률을 부여합니다. 한 토큰을 생성하면 그 토큰을 입력 뒤에 붙여 다시 다음 토큰을 예측하고, 이 과정을 반복해 문장 전체를 완성합니다. 이 반복을 방대한 텍스트로 학습하면, 문법과 사실 지식, 추론의 흔적까지 모델 파라미터 안에 자리 잡습니다.

> **핵심 포인트:** GPT는 "의미를 이해해서 답을 안다"기보다 "학습 데이터의 통계에 따라 가장 그럴듯한 다음 토큰을 잇습니다". 그래서 통계적으로 그럴듯하지만 사실과 다른 문장을 자신 있게 만들어 내기도 합니다. 이것이 환각의 근본 원인입니다.

4. ChatGPT의 등장과 RLHF 정렬

다음 토큰 예측만으로 학습한 모델은 문장을 유창하게 잇지만, 사용자의 의도에 맞게 친절하고 안전하게 답하도록 다듬어지지는 않습니다. ChatGPT가 보여 준 대화 품질은 사전학습된 거대 모델 위에 사람의 선호(preference)에 맞추는 정렬(alignment) 과정을 더한 결과입니다. 이 정렬의 대표 기법이 인간 피드백 기반 강화학습, 즉 RLHF(Reinforcement Learning from Human Feedback)입니다.



RLHF 3단계: 사전학습 → 지도 미세조정(SFT) → 보상모델 + 강화학습으로 사람 선호에 정렬

RLHF는 크게 세 단계로 진행됩니다. 첫째, 사전학습(pretraining) 단계에서 방대한 텍스트로 다음 토큰 예측을 학습해 언어 능력의 토대를 만듭니다. 둘째, 지도 미세조정(SFT, Supervised Fine-Tuning) 단계에서 사람이 직접 작성한 모범 답변으로 모델을 추가 학습시켜, 질문에 답하는 형식과 어조를 익히게 합니다. 셋째, 보상모델(reward model)과 강화학습 단계에서 같은 질문에 대한 여러 응답을 사람이 선호 순서로 평가하고, 그 평가를 학습한 보상모델이 더 선호되는 응답에 높은 점수를 주도록 본 모델을 강화학습으로 미세 조정합니다.

이 과정을 거치면 모델은 그럴듯한 문장을 잇는 것을 넘어, 사람이 더 좋다고 느끼는 방향으로 답하도록 정렬됩니다. 같은 GPT 계열이라도 RLHF를 거친 대화형 모델이 더 도움이 되고 안전하게 느껴지는 이유가 여기에 있습니다.

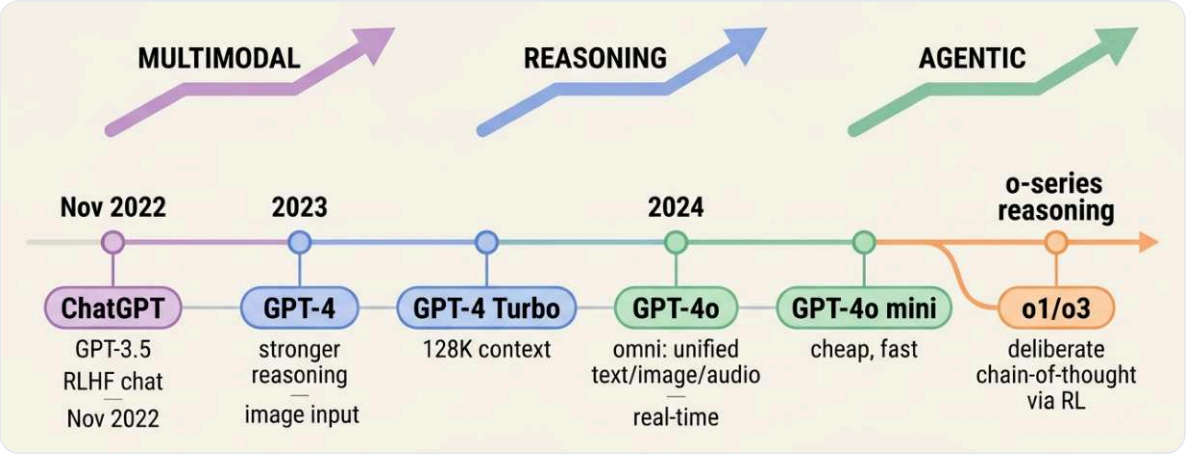
단계	입력 데이터	얻는 것
사전학습	대규모 일반 텍스트	언어 능력·지식의 토대
지도 미세조정(SFT)	사람이 쓴 모범 답변	질문에 답하는 형식과 어조
보상모델 + 강화학습	응답에 대한 사람의 선호 순위	사람 선호에 맞는 정렬

> **핵심 포인트:** RLHF는 모델에 새로운 지식을 넣는 과정이 아니라, 이미 가진 능력을 사람의 선호에 맞게 정렬하는 과정입니다. 그래서 정렬을 잘 해도 모델이 모르는 최신·사내 정보를 답하게 만들지는 못합니다. 이 빈틈을 메우는 것이 Day1 Part2에서 다룰 RAG입니다.

5. 2022년 이후 GPT 계열의 전개

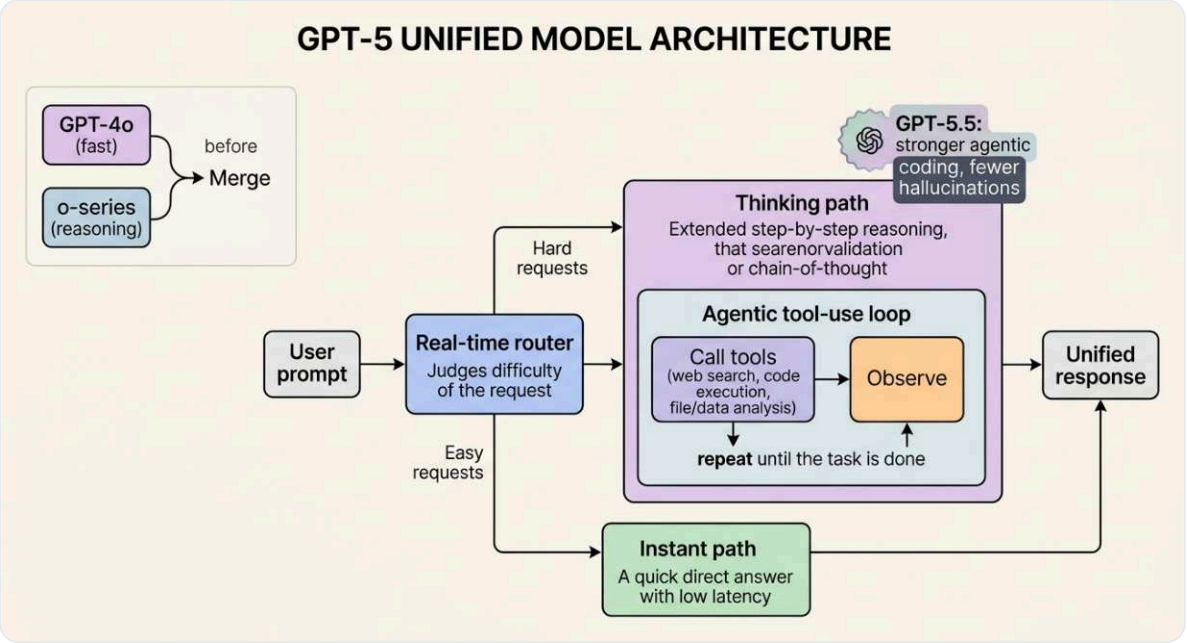
2022년 11월 ChatGPT(GPT-3.5)가 공개된 뒤 GPT 계열은 빠르게 갈라져 발전했습니다. 2023년의 GPT-4는 추론 능력과 함께 이미지 입력을 다루는 멀티모달 기능을 갖췄고, 2024년의 GPT-4o는 텍스트·이미지·음성을 한 모델에서 처리하는 옴니(omni) 멀티모달로 확장됐습니다. 같은 시기에 더 가볍고 저렴한 GPT-4o mini가 나와 대량·실시간 처리에 쓰

이고, 복잡한 추론에 특화된 o 시리즈가 별도 갈래로 자리 잡았습니다. 그리고 2025년의 GPT-5는 빠르게 답하는 모델과 깊이 추론하는 o 계열을 하나로 합쳐, 질문의 난도에 따라 즉답할지 더 오래 생각할지를 모델이 스스로 정하는 통합 구조로 나아갔습니다. 이 통합으로 사용자가 모델을 일일이 고르지 않아도 되고, 이전 세대보다 환각과 지시 위반이 줄었으며 코딩·에이전트 작업의 성능이 특히 강해졌습니다. GPT-5 역시 gpt-5·gpt-5-mini·gpt-5-nano처럼 용도별 크기로 나뉘어, 같은 계열 안에서 비용과 성능을 골라 쓸 수 있습니다. 이어 2026년의 GPT-5.5는 이 통합 구조 위에서 여러 도구를 오가며 작업을 끝까지 수행하는 에이전트·코딩 능력을 크게 끌어올렸고, 의료·법률·금융 같은 고난도 질문에서 환각을 다시 한번 큰 폭으로 줄였습니다. 이후로도 모델은 계속 갱신되지만, 큰 흐름은 멀티모달, 추론, 에이전트 세 방향으로 모입니다.



2022년 이후 OpenAI GPT 계열의 전개: ChatGPT(GPT-3.5) 공개 이후 멀티모달(GPT-4o)과 추론 특화(o 시리즈)로 분화

GPT-5의 내부 동작은 '실시간 라우터'로 요약됩니다. 사용자의 요청이 들어오면 라우터가 난도를 판단해, 쉬운 요청은 즉답 경로로 빠르게 답하고 어려운 요청은 추론 경로로 보내 단계적으로 사고하게 합니다. 추론 경로에서는 도구를 호출하고 결과를 확인하는 과정을 작업이 끝날 때까지 반복하는 에이전트 루프가 동작합니다. 과거에는 빠른 응답용 GPT-4o와 추론용 o 시리즈를 사용자가 직접 골라야 했지만, GPT-5는 이 둘을 한 모델로 합쳐 자동으로 경로를 선택합니다.



GPT-5의 통합 구조: 실시간 라우터가 난도를 판단해 쉬운 요청은 즉답 경로(빠른 응답), 어려운 요청은 추론 경로(단계적 사고와 도구 사용 반복)로 자동 분기한 뒤 하나의 응답으로 합친다. 과거 분리됐던 GPT-4o(빠름)와 o 시리즈(추론)가 한 모델로 통합되었다

실무에서 중요한 것은 작업에 맞는 모델을 고르는 일입니다. 복잡한 추론이 필요하면 상위 모델을, 대량 처리나 낮은 지연이 중요하면 경량 모델을 택합니다. RAG의 답변 생성 백본으로는 비용과 품질의 균형이 좋은 중간급 모델을 기본으로 두는 경우가 많습니다. 모델 이름과 버전은 자주 바뀌므로, 코드에서는 모델명을 상수나 환경변수로 분리해 한곳에서 관리하는 편이 좋습니다.

> **핵심 포인트:** 모델은 계속 바뀌지만 고르는 기준은 같습니다. "추론 난도, 응답 속도, 비용" 세 축으로 작업에 맞는 모델을 선택하고, 모델명은 코드 곳곳에 흩지 말고 한곳에서 관리합니다.

6. 정리

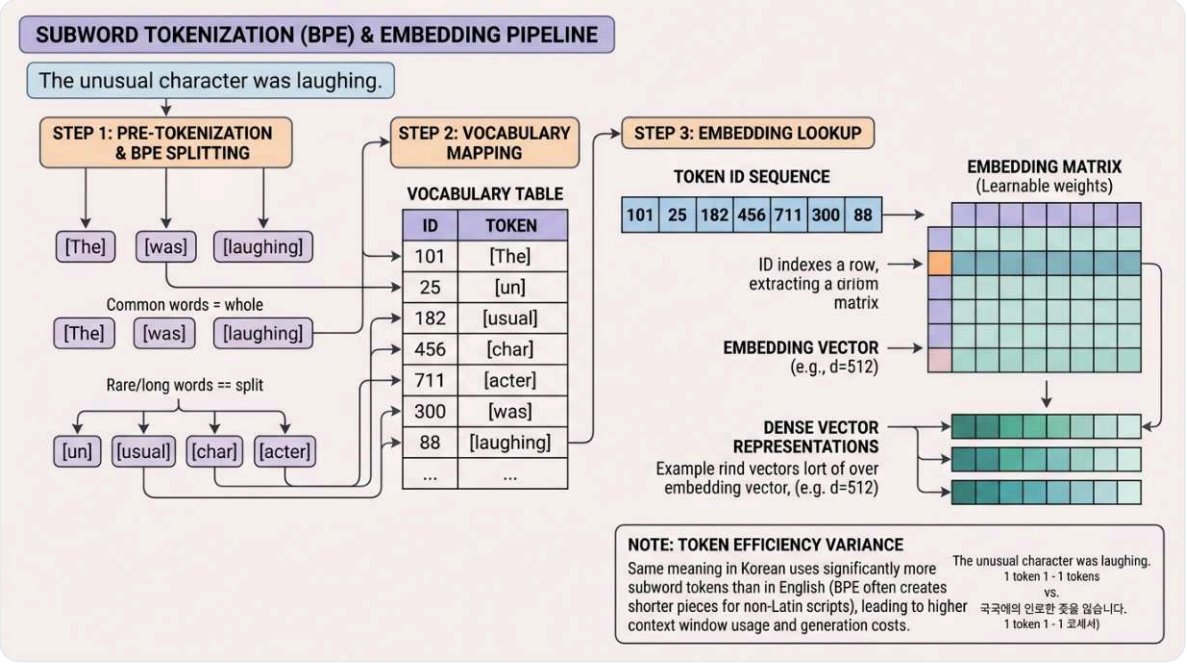
생성형 AI는 새로운 콘텐츠를 만들어 내는 인공지능이고, 그 중심에는 다음 토큰 예측으로 작동하는 대규모 언어모델이 있습니다. 규칙 기반에서 머신러닝, 딥러닝, 트랜스포머를 거쳐 LLM에 이른 발전사는 "사람이 규칙을 짜던" 방식에서 "모델이 데이터에서 스스로 배우는" 방식으로의 이동입니다. GPT는 통계적으로 그럴듯한 다음 토큰을 잇는 모델이며, ChatGPT는 그 위에 RLHF 정렬을 더해 사람에게 유용하고 안전한 대화를 가능하게 했습니다. 다만 정렬만으로는 환각과 최신 정보 부재라는 한계를 해결할 수 없으며, 뒤에서 직접 구현할 RAG가 그 해법입니다.

트랜스포머와 임베딩의 직관

> 토큰, 임베딩, 어텐션이라는 세 가지 개념을 수식 없이 직관으로 잡아 LLM의 내부 동작을 이해합니다.

1. 토큰: 모델이 읽는 최소 단위

언어모델은 문장을 글자나 단어 그대로 다루지 않습니다. 먼저 텍스트를 토큰(token)이라는 작은 조각으로 쪼개는 토크나이즈(tokenize) 과정을 거칩니다. 토큰은 단어 하나일 수도, 단어의 일부일 수도, 공백이나 기호일 수도 있습니다. 쪼갠 각 토큰에는 고유한 숫자 ID가 붙고, 모델은 이 숫자 ID의 나열을 입력으로 받습니다.



토크나이즈: 문장이 토큰 단위로 분해되고 각 토큰이 숫자 ID로 변환된다

토큰 단위로 쪼개는 이유는 어휘 크기와 표현력 사이의 균형 때문입니다. 모든 단어를 통째로 어휘에 넣으면 어휘가 너무 커지고, 처음 보는 단어(OOV)를 다룰 수 없습니다. 반대로 글자 단위로 쪼개면 어휘는 작아지지만 한 문장의 토큰 수가 지나치게 많아집니다. 그래서 요즘 토크나이저는 자주 등장하는 덩어리는 한 토큰으로, 드문 단어는 여러 서브워드로 쪼개는 서브워드(subword) 방식을 씁니다. 다음은 여러 토크나이저를 비교하는 예제의 핵심 부분입니다.

```python
# 12.study/5.tokenizer/1.tokenizer_compare.py
from transformers import AutoTokenizer

def show_tokenization(tokenizer_name, text, label=""):
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
    tokens = tokenizer.tokenize(text)
    token_ids = tokenizer.encode(text, add_special_tokens=False)
    print(f"  토큰 수: {len(tokens)}")
    print(f"  토큰: {tokens}")
    print(f"  토큰 ID: {token_ids}")
    return tokens

# 같은 의미라도 모델마다 토큰 분할이 다르다
show_tokenization("openai-community/gpt2", "Artificial intelligence is transforming the wo
show_tokenization("openai-community/gpt2", "인공지능이 세상을 변화시키고 있습니다.")
```

이 코드는 같은 문장을 여러 토크나이저로 쪼개 토큰의 개수와 ID를 보여 줍니다. 여기서 눈여겨볼 점은 같은 의미의 문장이라도 한국어가 영어보다 토큰 수가 훨씬 많다는 것입니다. 영어 중심으로 학습된 토크나이저는 한국어를 잘게 쪼개기 때문에, 같은 내용을 처리하는 데 토큰이 2~4배 더 듭니다.

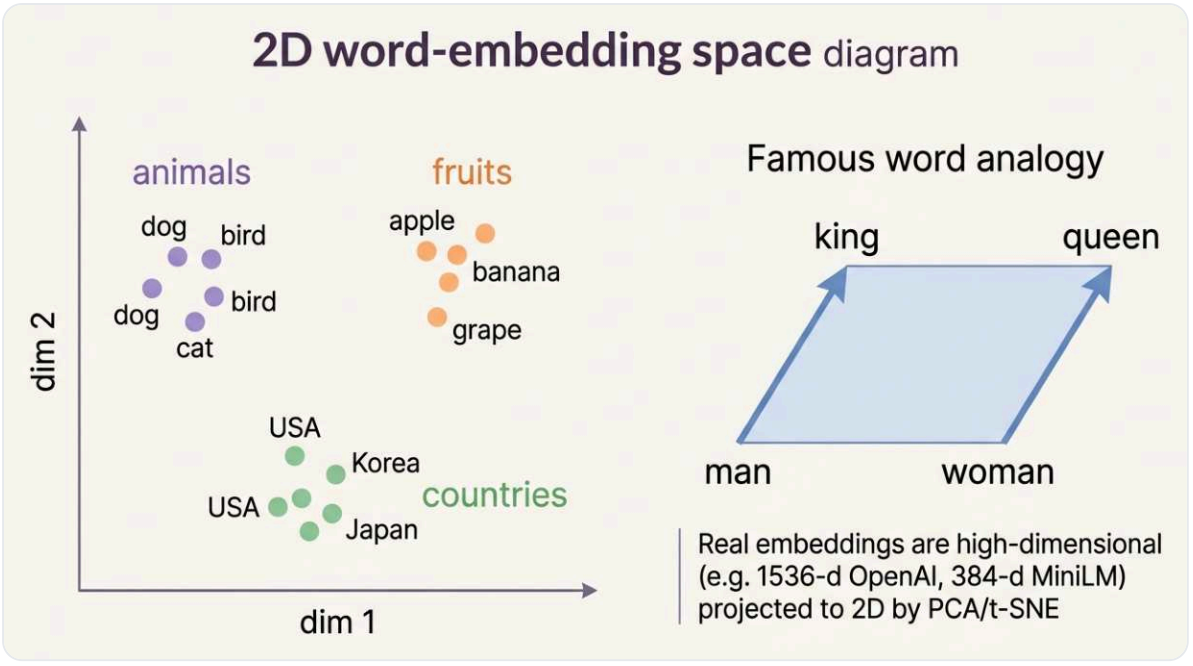
> **핵심 포인트:** OpenAI를 비롯한 API의 과금과 컨텍스트 길이 제한은 글자 수가 아니라 토큰 수 기준입니다. 한국어는 토큰이 많이 드는 언어이므로, 프롬프트와 문서 길이를 토큰 단위로 가늠하는 습관이 비용 관리의 출발점입니다.

2. 임베딩: 토큰을 의미가 담긴 벡터로

토큰 ID 자체는 사전의 일련번호일 뿐, 의미를 담고 있지 않습니다. ID 100번과 101번이 의미상 가깝다는 보장은 없습니다. 그래서 모델은 각 토큰 ID를 임베딩(embedding)이라는 고정 길이의 실수 벡터로 바꿉니다. 이 벡터의 각 차원은 토큰의 의미적 특성을 나눠 담으며, 의미가 비슷한 토큰은 벡터 공간에서 가까운 위치에 놓이도록 학습됩니다.

```python
# 12.study/6.embedding/2.embedding_visualize.py
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("paraphrase-multilingual-MiniLM-L12-v2")

sample = "인공지능이 세상을 바꾸고 있다."
emb = model.encode(sample)
print(f"  벡터 차원: {emb.shape[0]}")        # 예: 384
print(f"  처음 10개 값: {emb[:10].round(4)}")  # 실수들의 배열
```

이 코드는 한 문장을 임베딩 모델에 넣어 고정 길이(예: 384차원)의 실수 벡터로 변환합니다. 출력된 벡터는 사람 눈에는 의미 없는 숫자의 나열로 보이지만, 이 배열의 방향이 곧 문장의 의미를 나타냅니다. "인공지능이 세상을 바꾼다"와 "AI가 세계를 변화시킨다"는 표현은 다르지만 벡터가 거의 같은 방향을 가리키고, "날씨가 좋다"와 "피자를 먹고 싶다"는 서로 다른 방향을 가리킵니다.

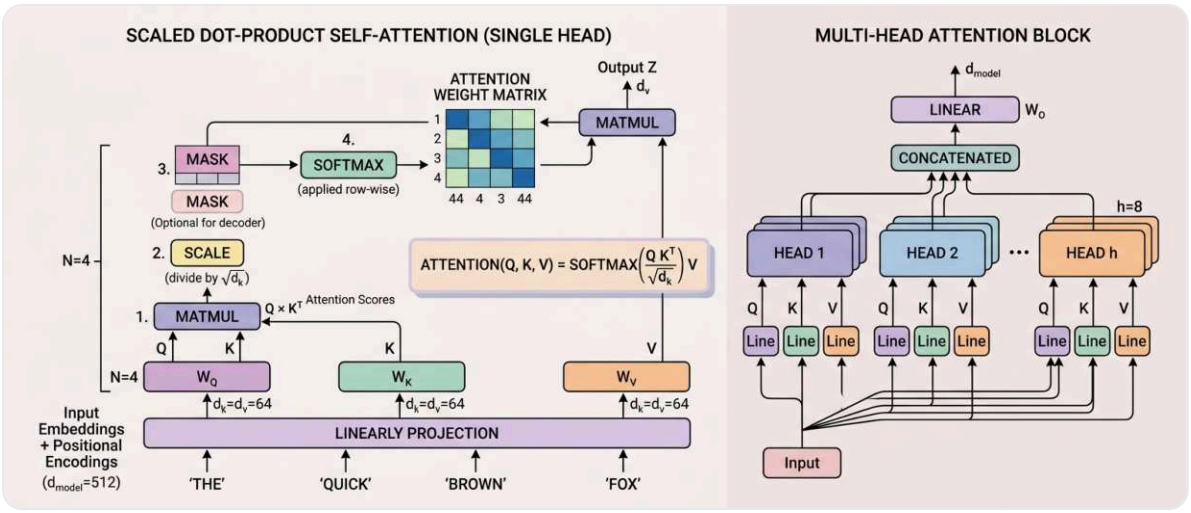


임베딩 공간: 'man→king'과 'woman→queen'처럼 의미 관계가 같으면 평행한 벡터로 나타나고, 비슷한 단어는 서로 가까이 모인다

이렇게 의미를 좌표로 바꿔 놓으면, 두 텍스트가 의미상 얼마나 가까운지를 벡터 사이의 거리나 각도로 계산할 수 있습니다. 이 성질이 RAG에서 질문과 관련된 문서를 찾아내는 검색의 토대입니다. 임베딩과 유사도 계산은 뒤의 별도 장에서 코드로 직접 구현합니다.

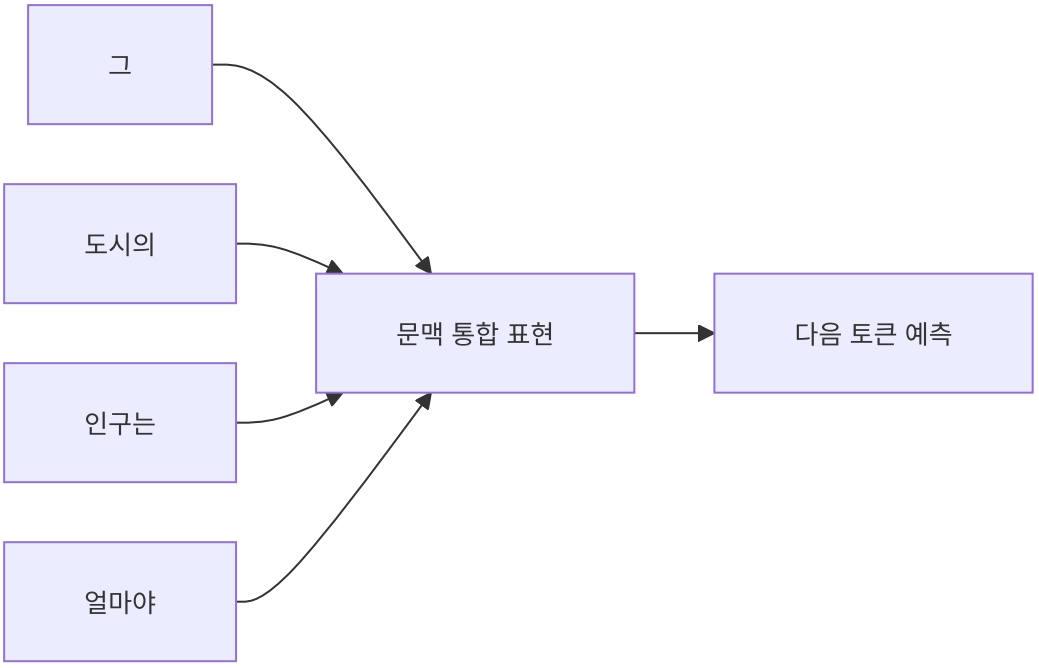
3. 어텐션: 문맥을 함께 보는 메커니즘

토큰을 임베딩으로 바꾼 뒤, 트랜스포머가 수행하는 핵심 연산이 어텐션(attention)입니다. 어텐션의 직관은 "한 단어를 이해하려면 같은 문장 안의 다른 단어들을 함께 봐야 한다"는 것입니다. 예를 들어 "그 도시의 인구는 얼마야"라는 문장에서 "그 도시"가 무엇을 가리키는지는 앞 문맥의 다른 단어를 봐야 알 수 있습니다.



셀프 어텐션: 각 단어를 Query·Key·Value로 투영해, Query–Key 유사도로 가중치를 만들어 Value를 합성한다(수식·멀티헤드는 그림 참고)

셀프 어텐션(self-attention)은 문장 안의 모든 단어가 서로를 바라보면서, 각 단어가 다른 단어에 얼마나 주목할지 가중치를 매기는 방식입니다. 어떤 단어와 강하게 관련되면 굵은 화살표(높은 가중치)로, 무관하면 가는 화살표(낮은 가중치)로 연결된다고 보면 됩니다. 이렇게 각 단어는 자신과 관련 깊은 단어들의 정보를 더 많이 끌어와 자신의 표현을 갱신합니다.



이 구조의 큰 장점은 문장을 앞에서부터 한 단어씩 순서대로 처리하지 않고, 모든 단어 쌍의 관계를 한 번에 계산한다는 것입니다. 덕분에 GPU에서 대규모 병렬 학습이 가능해졌고, 이것이 모델과 데이터를 거대하게 키워 오늘날의 LLM을 만든 결정적 요인입니다. 어텐션 층을 깊게 쌓을수록 모델은 더 복잡한 문맥 관계를 포착합니다.

> **핵심 포인트:** 셀프 어텐션은 "각 단어가 다른 단어에 가중치를 두고 주목한다"는 한 줄로 충분히 직관을 잡을 수 있습니다. 수식의 세부보다, 단어 사이 관계를 한 번에 본다는 점과 그 덕에 병렬 학습이 가능해졌다는 점이 핵심입니다.

4. 토큰·임베딩·어텐션의 연결

토큰, 임베딩, 어텐션은 따로 노는 개념이 아니라 하나의 흐름으로 이어집니다. 텍스트는 토크나이즈를 거쳐 토큰 ID 나열이 되고, 각 토큰 ID는 임베딩 벡터로 바뀌어 의미 공간에 놓입니다. 그 벡터들이 어텐션 층을 여러 번 통과하며 문맥을 반영한 표현으로 다듬어지고, 마지막에 모델은 그 표현을 바탕으로 다음 토큰의 확률 분포를 계산합니다.

개념	한 줄 직관	RAG와의 연결
토큰	텍스트를 쪼갠 최소 단위(숫자 ID)	비용·컨텍스트 길이의 단위
임베딩	의미를 담은 고정 길이 벡터	문서·질문을 같은 공간에 두고 검색
어텐션	단어가 서로에게 주목	긴 문맥을 한 번에 이해

이 장에서 잡은 직관, 특히 "의미가 비슷한 텍스트는 임베딩 벡터가 가깝다"는 성질은 뒤에서 유사도 검색을 직접 구현할 때 그대로 쓰입니다. 트랜스포머의 내부 수식을 모두 외우지 않아도, 이 세 개념의 흐름만 분명히 잡으면 RAG 파이프라인의 모든 단계를 다룰 수 있습니다.

OpenAI API 기초와 CLI 챗봇

> API 키 보안부터 Chat Completion의 구조와 역할까지, OpenAI API로 첫 챗봇을 만드는 기본기를 다집니다.

1. API 키 발급과 .env 보안

OpenAI API를 쓰려면 먼저 OpenAI 플랫폼에서 발급한 API 키가 필요합니다. 이 키는 결제 계정과 직접 연결된 비밀 자격증명이므로, 소스 코드나 깃 저장소, 로그에 직접 적어서는 안 됩니다. 키가 노출되면 타인이 내 계정으로 API를 호출해 비용을 발생시킬 수 있습니다.

안전한 방법은 키를 환경변수로 분리하는 것입니다. 프로젝트 루트에 `.env` 파일을 만들어 키를 넣고, 이 파일은 `.gitignore`에 등록해 버전 관리에서 제외합니다.

```
# .env (저장소에 커밋하지 말 것)
OPENAI_API_KEY=sk-여기에_발급받은_키
```

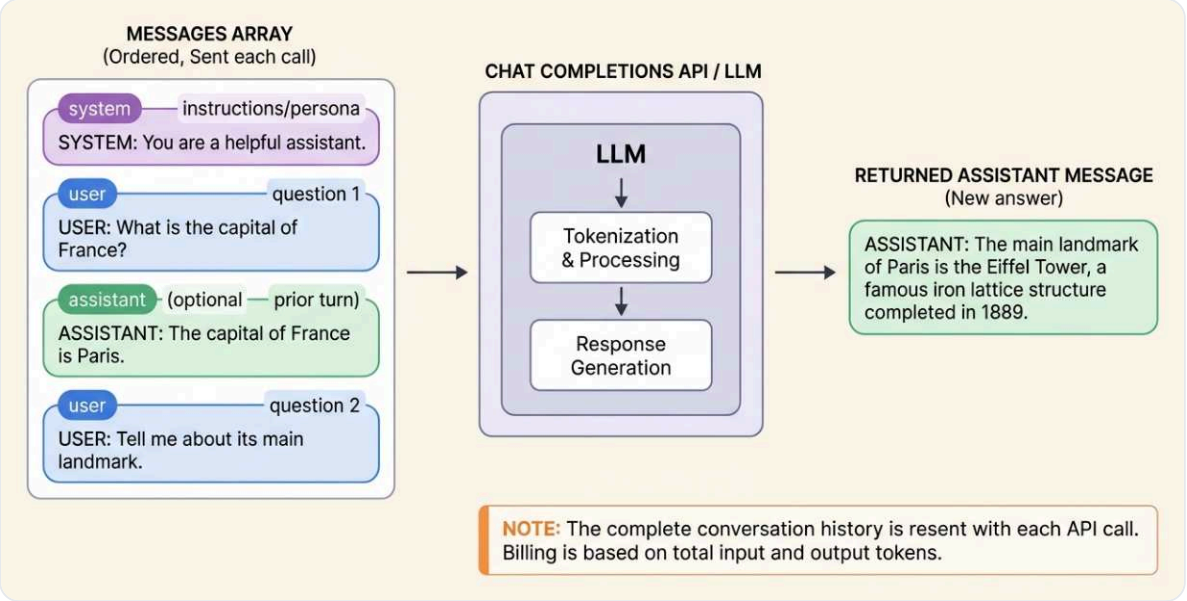
코드에서는 `python-dotenv`로 `.env`를 읽어 환경변수로 올린 뒤, `os.getenv`로 키를 꺼내 클라이언트를 만듭니다. 키 문자열이 코드 어디에도 직접 등장하지 않는 것이 핵심입니다.

```python
# 1.openai/1.intro/11.sdk_new.py
import os
from openai import OpenAI
from dotenv import load_dotenv

load_dotenv(dotenv_path='../.env')
client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))
```

> **핵심 포인트:** API 키와 개인정보(이름·이메일 등)는 코드·로그·외부 요청에 직접 포함하지 말고 반드시 `.env` 환경변수로 관리합니다. `os.getenv("OPENAI_API_KEY")` 패턴을 기본으로 삼습니다.

2. Chat Completion의 구조

OpenAI의 대화 모델은 Chat Completion이라는 형식으로 호출합니다. `messages` 리스트에 대화 메시지를 순서대로 담아 보내면, 모델이 그 뒤에 이어질 응답 메시지 하나를 만들어 돌려줍니다. v1.x SDK에서는 `OpenAI()` 로 만든 클라이언트의 `client.chat.completions.create(...)` 를 호출합니다.



Chat Completion 구조: system·user·assistant 역할의 메시지 배열을 보내면 모델이 다음 assistant 메시지를 생성

```python
# 1.openai/1.intro/11.sdk_new.py
response = client.chat.completions.create(
    model='gpt-4o-mini',
    messages=[
        {'role': 'system', 'content': '당신은 친절한 AI 도우미입니다.'},
        {'role': 'user', 'content': '안녕, 챗봇!'},
    ],
)

# 응답은 객체 → 속성으로 접근한다
print('챗봇:', response.choices[0].message.content)
```

이 코드의 흐름은 단순합니다. `model` 로 사용할 모델을 지정하고, `messages` 에 대화를 담아 `create` 를 호출하면 응답 객체가 돌아옵니다. 답변 텍스트는 `response.choices[0].message.content` 로 꺼냅니다. REST 방식에서는 같은 값을 딕셔너리 형태(`response.json()['choices'][0]['message']['content']`)로 접근했지만, SDK에서는 객체 속성으로 더 간결하게 읽습니다.

3. system / user / assistant 역할

`messages` 안의 각 메시지에는 `role`이 붙습니다. 이 역할 구분이 모델의 행동을 결정합니다.

역할	의미	쓰임
system	모델의 성격·규칙 설정	어조, 페르소나, 제약 조건 지정
user	사용자의 입력	실제 질문이나 요청
assistant	모델의 이전 답변	멀티턴 대화에서 맥락 유지

`system` 메시지는 대화 맨 앞에 한 번 두어 모델이 어떤 역할로 답할지 정합니다. 같은 질문이라도 system 프롬프트를 바꾸면 답의 어조와 관점이 달라집니다.

```
# 1.openai/1.intro/4.restapi_chat.py
# 챗봇 성격 — 한 줄만 활성화해 바꿔보세요
system_prompt = '당신은 친절한 AI 도우미입니다.'
# system_prompt = '당신은 여행 가이드입니다. 여행 정보를 제공하세요.'
# system_prompt = '당신은 초등학생도 이해하도록 쉽게 설명하는 선생님입니다.'
```

`user`는 사용자의 발화이고, `assistant`는 모델의 이전 답변입니다. 단발성 질문에는 system과 user만 있으면 되지만, 대화를 이어 가려면 모델의 답변을 assistant 역할로 다시 넣어 줘야 합니다. 이 멀티턴 처리는 별도의 장에서 자세히 다룹니다.

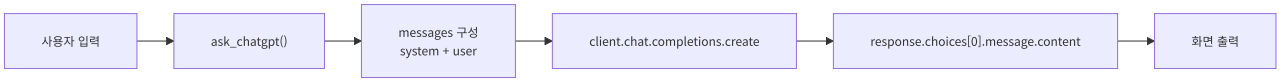
4. CLI 챗봇 만들기

지금까지의 요소를 묶으면 터미널에서 동작하는 간단한 대화 루프를 만들 수 있습니다. API 호출을 함수로 분리하고, `try/except`로 네트워크나 인증 오류에 대비하며, `input()`으로 사용자 입력을 반복해서 받습니다.

```python
# 1.openai/1.intro/4.restapi_chat.py
def ask_chatgpt(user_input):
    try:
        response = client.chat.completions.create(
            model='gpt-4o-mini',
            messages=[
                {'role': 'system', 'content': system_prompt},
                {'role': 'user', 'content': user_input},
            ],
        )
        return response.choices[0].message.content
    except Exception as error:
        print('API 요청 중 오류 발생:', str(error))
        return '응답을 가져오는 도중 오류가 발생했습니다.'

if __name__ == '__main__':
    print('챗봇과 대화를 시작합니다. (종료: exit)')
    while True:
        user_input = input('\n나: ').strip()
        if user_input.lower() == 'exit':
            break
        print('챗봇:', ask_chatgpt(user_input))
```

이 구조에서는 매 호출이 독립적입니다. system과 그때그때의 user 메시지만 보내므로, 모델은 직전에 무슨 대화를 했는지 기억하지 못합니다. 의도된 단순함이며, 대화 기억은 멀티턴 처리를 더하면서 해결합니다. 같은 로직을 웹으로 옮기면 Flask 서버의 엔드포인트 안에서 같은 `ask_chatgpt` 함수를 호출하는 형태가 됩니다.

```python
# 1.openai/2.chatbot_ui/app2_openailib.py
@app.route('/api/chat', methods=['POST'])
def chat():
    data = request.get_json()
    user_input = data.get('userInput', '')
    response = ask_chatgpt(user_input)
    return jsonify({'chatgpt': response})
```



5. 비용과 Rate Limit

API는 사용한 토큰 수에 비례해 과금됩니다. 입력(프롬프트) 토큰과 출력(응답) 토큰이 각각 계산되며, 모델에 따라 단가가 다릅니다. 가볍고 저렴한 모델(`gpt-4o-mini`)은 일상 대화와 간단한 작업에, 더 큰 모델은 품질이 중요한 작업에 쓰는 식으로 고릅니다. 한국어는 같은 내용도 토큰이 더 많이 든다는 점을 고려해 프롬프트 길이를 관리합니다.

또한 계정과 모델마다 분당 요청 수와 토큰 수에 제한(rate limit)이 있습니다. 짧은 시간에 많은 요청을 보내면 일시적으로 거부될 수 있으므로, 실제 서비스에서는 예외 처리와 재시도 로직을 함께 둡니다.

> **핵심 포인트:** 모델 선택은 곧 비용과 품질의 트레이드오프입니다. 토큰 단위 과금과 rate limit을 의식해 모델과 프롬프트 길이를 정하면, 같은 결과를 더 적은 비용으로 얻을 수 있습니다.

대화형 챗봇: 멀티턴과 Temperature

> messages 배열로 대화 이력을 관리해 맥락을 잇고, temperature로 응답의 성격을 조절하는 법을 익힙니다.

1. 단발성 호출의 한계

앞에서 만든 CLI 챗봇은 매 호출이 독립적입니다. system 메시지와 그때의 user 메시지만 보내므로, 모델은 직전에 무슨 대화를 했는지 기억하지 못합니다. 예를 들어 "대한민국의 수도는 어디야?"라고 물어 "서울"이라는 답을 받은 뒤 "그 도시의 인구는 얼마야?"라고 이어 물으면, 모델은 "그 도시"가 어디인지 알 수 없습니다. 직전 대화가 다음 요청에 함께 전달되지 않기 때문입니다.

이 한계는 모델의 결함이 아니라 호출 방식의 특성입니다. Chat Completion API는 상태를 저장하지 않는(stateless) 방식이라, 모델이 맥락을 이어 가게 하려면 지금까지의 대화를 매번 함께 보내야 합니다.

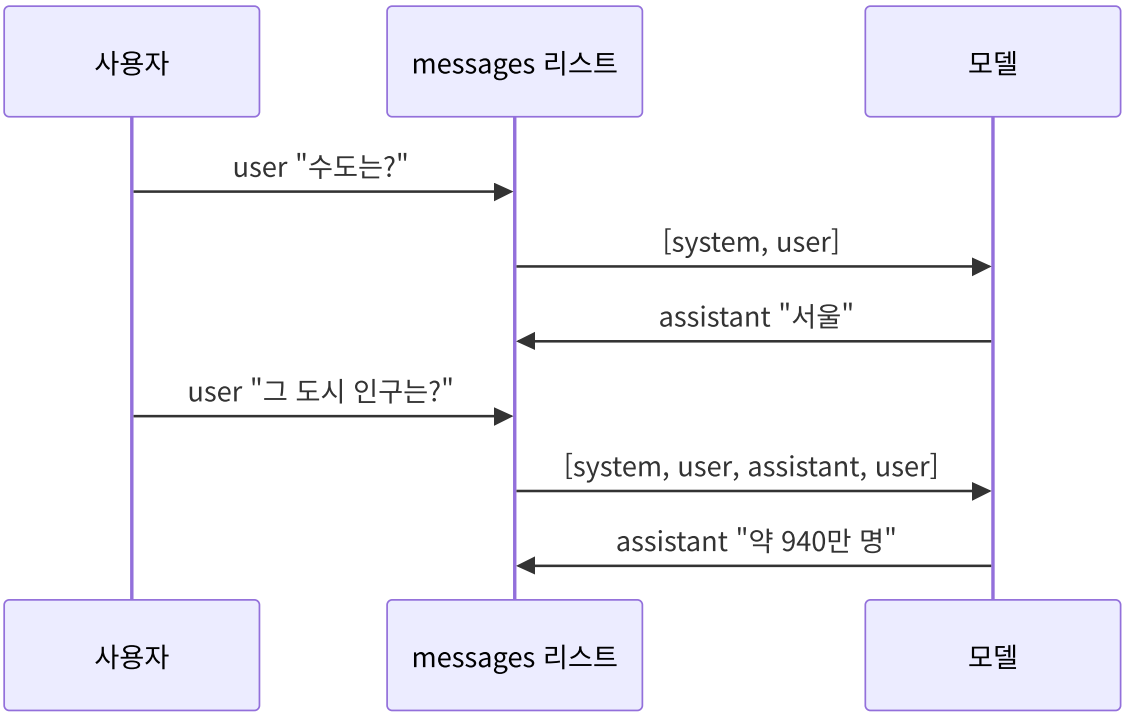
> **핵심 포인트:** 모델은 기억 장치를 따로 갖지 않습니다. 대화 맥락은 클라이언트(우리 코드)가 messages 리스트에 누적해서 매 호출에 함께 보내 줄 때만 유지됩니다.

2. messages로 대화 이력 누적하기

멀티턴(multi-turn) 대화의 원리는 단순합니다. 대화가 진행될수록 messages 리스트에 user 메시지와 모델의 assistant 응답을 차곡차곡 쌓아, 매 호출에 지금까지의 전체 대화를 보냅니다. 모델의 지난 답변을 assistant 역할로 다시 넣어 줘야, 모델이 자기가 한 답을 근거로 맥락을 이어 갑니다.

```python
# 1.openai/1.intro/13.chat_multiturn.py
# 대화 누적 리스트 – system 메시지로 시작한다.
messages = [
    {'role': 'system', 'content': '당신은 친절한 AI 도우미입니다.'},
]

def chat(user_input):
    # 1) 사용자 메시지를 리스트에 추가
    messages.append({'role': 'user', 'content': user_input})

    # 2) 지금까지의 '전체 대화'를 보낸다
    response = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=messages,
    )
    answer = response.choices[0].message.content

    # 3) 모델의 답변도 assistant 역할로 추가 → 다음 턴에서 기억된다
    messages.append({'role': 'assistant', 'content': answer})
    return answer
```

이 `chat` 함수의 세 단계가 멀티턴의 전부입니다. 사용자 입력을 user로 추가하고, 누적된 전체 messages를 보내고, 돌아온 답변을 assistant로 다시 추가합니다. 그러면 두 번째 질문에 "그 도시"라고만 적어도, 첫 답변이 messages에 남아 있어 모델이 맥락을 이어서 답합니다.



웹 서버에서도 같은 원리를 씁니다. 서버 측에 대화 이력 리스트를 두고, 요청이 올 때마다 user 메시지를 추가한 뒤 system 메시지와 함께 묶어 모델에 보냅니다.

```
# 1.openai/3.chatbot2_history/app3_history.py
conversation_history = []

@app.route('/api/chat', methods=['POST'])
def chat():
    user_input = request.json.get('userInput', '')
    conversation_history.append({'role': 'user', 'content': user_input})
    response = ask_chatgpt(conversation_history)
    conversation_history.append({'role': 'assistant', 'content': response})
    return jsonify({'chatgpt': response})

def ask_chatgpt(conversation_history):
    input_messages = [
        {'role': 'system', 'content': '당신은 도움이 되는 AI 어시스턴트입니다.'},
        *conversation_history,
    ]
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=input_messages,
    )
    return response.choices[0].message.content
```

여기서 `*conversation_history`는 누적된 대화를 system 메시지 뒤에 펼쳐 넣는 부분입니다. system은 항상 맨 앞에 한 번만 두고, 그 뒤로 user와 assistant가 번갈아 쌓입니다.

3. 대화 이력 길이 관리

대화가 길어지면 messages도 끝없이 길어집니다. 그런데 모델이 한 번에 다룰 수 있는 토큰에는 한계(컨텍스트 윈도우)가 있고, messages가 길수록 입력 토큰이 늘어 비용도 커집니다. 그래서 일정 길이를 넘으면 오래된 대화를 잘라 내는 관리가 필요합니다.

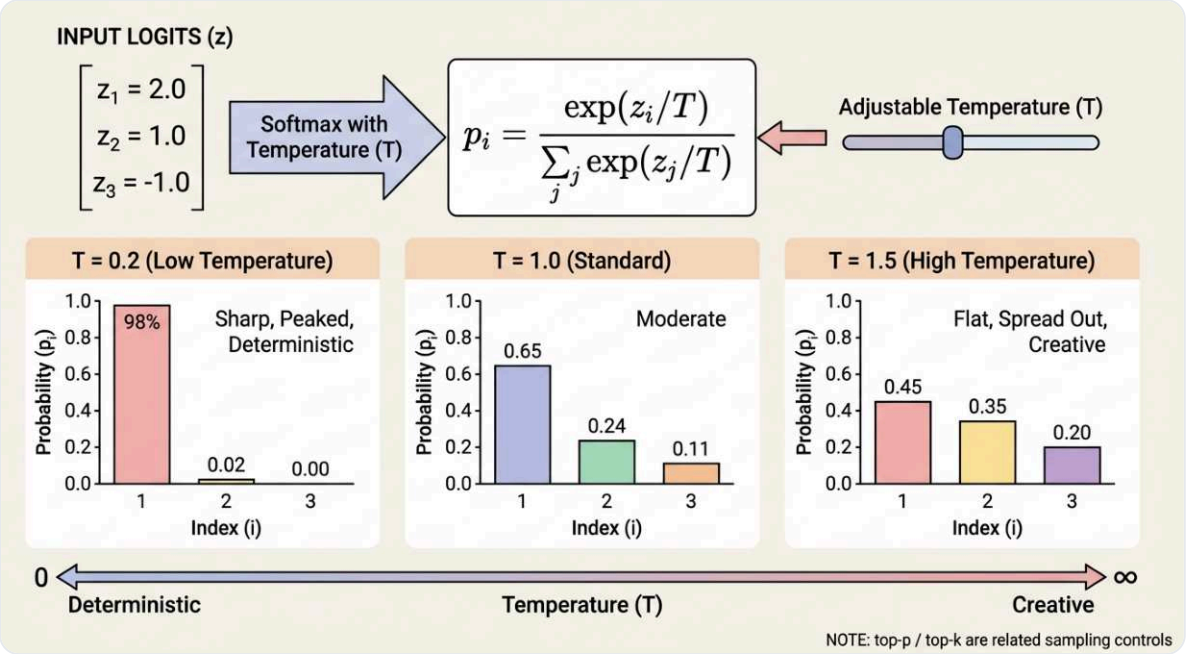
```python
# 1.openai/3.chatbot2_history/app4_historylimit.py
MAX_HISTORY_LENGTH = 10
conversation_history = []

@app.route('/api/chat', methods=['POST'])
def chat():
    user_input = request.json.get('userInput', '')
    conversation_history.append({'role': 'user', 'content': user_input})
    response = ask_chatgpt(conversation_history)
    conversation_history.append({'role': 'assistant', 'content': response})

    # 대화 히스토리 길이 관리: 오래된 대화부터 삭제
    while len(conversation_history) > MAX_HISTORY_LENGTH:
        conversation_history.pop(0)
    return jsonify({'chatgpt': response})
```

이 코드는 누적된 대화가 `MAX_HISTORY_LENGTH`를 넘으면 가장 오래된 메시지부터 제거합니다. 최근 맥락은 유지하면서 입력 토큰이 끝없이 커지는 것을 막습니다. 실제 서비스에서는 단순 개수 제한 대신 토큰 수 기준으로 자르거나, 오래된 대화를 요약해 압축하는 방식을 쓰기도 합니다.

> **핵심 포인트:** 멀티턴 대화의 비용과 길이는 messages 누적량에 비례합니다. 컨텍스트 윈도우 한계 안에서 최근 맥락을 유지하도록 이력을 적절히 잘라 내는 것이 안정적인 챗봇 운영의 핵심입니다.

4. Temperature로 응답 성격 조절하기

같은 질문에도 모델의 답을 더 일관되게, 또는 더 다양하게 만들 수 있습니다. 이를 조절하는 대표 파라미터가 `temperature`입니다. temperature가 낮으면 모델은 확률이 가장 높은 토큰에 집중해 안정적이고 반복적인 답을 내고, 높으면 확률 분포가 평평해져 더 다양한 답을 냅니다.



Temperature: 값이 낮으면 안정적·반복적, 값이 높으면 창의적·다양한 출력

```python
# 1.openai/1.intro/12.sdk_params.py
temperature = 0.7  # 0.0 (정확·일관) / 1.5 (창의·다양)

response = client.chat.completions.create(
    model='gpt-4o-mini',
    messages=[
        {'role': 'system', 'content': '당신은 친절한 AI 도우미입니다.'},
        {'role': 'user', 'content': '인공지능을 한 문장으로 설명해줘.'},
    ],
    temperature=temperature,  # 창의성 (0.0 정확 ~ 2.0 창의)
    max_tokens=100,           # 응답의 최대 토큰 수
    top_p=0.9,                # 확률 상위 토큰만 선택 (0.0 ~ 1.0)
    frequency_penalty=0.5,    # 같은 단어 반복 억제 (-2.0 ~ 2.0)
    presence_penalty=0.6,     # 새로운 주제 도입 장려 (-2.0 ~ 2.0)
)
```

함께 자주 쓰는 파라미터를 표로 정리하면 다음과 같습니다.

파라미터	역할	범위(대략)
temperature	출력의 무작위성·창의성	0.0 ~ 2.0
max_tokens	응답의 최대 토큰 수	모델 한계 내
top_p	확률 상위 토큰만 후보로 제한	0.0 ~ 1.0
frequency_penalty	같은 단어 반복 억제	-2.0 ~ 2.0
presence_penalty	새로운 주제 도입 장려	-2.0 ~ 2.0

RAG처럼 사실에 근거한 답이 중요한 작업에서는 temperature를 낮게(0에 가깝게) 두어 일관되고 보수적인 답을 유도합니다. 반대로 아이디어 브레인스토밍처럼 다양성이 필요한 작업에서는 높게 둡니다. 참고로 일부 추론 특화 모델은 temperature 같은 파라미터를 지원하지 않으므로, 사용하는 모델이 어떤 파라미터를 받는지 확인하고 적용해야 합니다.

> **핵심 포인트:** temperature는 "정확함과 창의성 사이의 다이얼"입니다. 사실 기반 응답이 중요한 RAG에서는 낮게 설정해 출력을 안정시키고, 같은 입력에 같은 답이 나오도록 재현성을 높입니다.

PART 02

RAG 핵심 원리를 직접 구현하기

1. RAG 개요: 왜 필요한가
2. 문서 전처리와 청킹 전략
3. 임베딩 이해
4. 유사도 검색 직접 구현
5. 미니 프로젝트: 손코딩 RAG QA

RAG 개요: 왜 필요한가

> 환각과 최신 정보 부재라는 LLM의 약점을 짚고, RAG가 이를 어떻게 해결하는지 Fine-tuning과 비교해 이해합니다.

1. LLM의 두 가지 약점: 환각과 최신 정보 부재

대규모 언어모델은 학습 데이터의 통계를 바탕으로 가장 그럴듯한 다음 토큰을 잇는 방식으로 작동합니다. 이 방식은 유창한 문장을 만들지만, 두 가지 근본적인 약점을 함께 지닙니다.

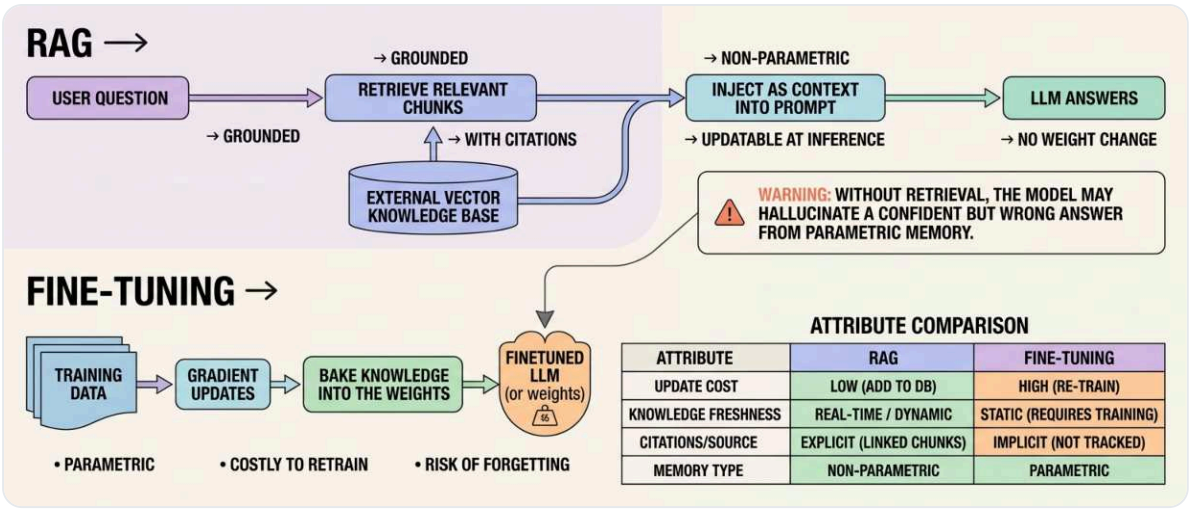
첫째는 환각(hallucination)입니다. 모델은 모르는 사실에 대해서도 "모른다"고 말하기보다, 통계적으로 그럴듯한 답을 자신 있게 지어냅니다. 존재하지 않는 논문 제목, 틀린 수치, 사실과 다른 인용을 매끄러운 문장으로 만들어 내기 때문에, 겉보기에는 그럴듯해 사용자가 오답을 알아차리기 어렵습니다.

둘째는 최신·내부 정보의 부재입니다. 모델의 지식은 학습 시점에 고정됩니다. 학습 이후에 일어난 사건이나, 애초에 학습 데이터에 없던 사내 문서, 특정 제품 매뉴얼 같은 정보는 모델이 알 수 없습니다. 정렬(RLHF)을 아무리 잘 해도 모델이 본 적 없는 정보를 답하게 만들 수는 없습니다.

> **핵심 포인트:** 환각과 최신 정보 부재는 모델을 더 크게 키우거나 더 잘 정렬한다고 사라지지 않습니다. 모델 내부 기억에만 의존하는 한 근본적으로 남는 한계입니다.

2. RAG라는 해법

RAG(Retrieval-Augmented Generation, 검색 증강 생성)는 이 약점을 외부 지식으로 메웁니다. 핵심은 모델에게 질문만 던지는 대신, 질문과 관련된 문서를 먼저 찾아서 함께 주고 그 문서에 근거해 답하게 하는 것입니다. 모델이 내부 기억을 더듬어 추측하는 대신, 눈앞에 주어진 근거 자료를 읽고 답하게 합니다.



RAG vs 파인튜닝: 모델 내부 기억(파라미터)에만 의존하면 환각, 외부 문서를 검색해 근거로 주면(RAG) 출처 있는 정확한 답

이렇게 하면 두 약점이 동시에 완화됩니다. 모델이 답의 근거를 외부 문서에서 가져오므로 환각이 줄고, 학습 이후의 최신 정보나 사내 문서도 문서만 갖춰 두면 답할 수 있습니다. 모델을 다시 학습시키지 않고, 검색해 온 문서를 프롬프트에 끼워 넣는 것만으로 지식을 갱신하는 셈입니다.

기본 RAG 예제의 주석은 이 아이디어를 간결하게 요약합니다.

```
# 1.openai/7.rag/1.rag_basic.py
# RAG(Retrieval-Augmented Generation, 검색 증강 생성):
#   LLM에게 질문만 던지는 게 아니라, '관련 문서를 먼저 찾아서' 함께 줘서 답하게 하는 방식.
#   → 모델이 모르는 최신/사내 정보도 문서만 있으면 답할 수 있다.
#
# 파이프라인:
#   문서 → 임베딩(벡터화) → 저장 → 질문 임베딩 → 유사 문서 검색 → GPT 응답
```

3. RAG vs Fine-tuning

모델에 새로운 지식을 반영하는 또 다른 방법으로 파인튜닝(Fine-tuning)이 있습니다. 둘은 목적이 겹쳐 보이지만 성격이 다르므로, 상황에 맞게 선택하거나 병행합니다.

파인튜닝은 모델을 추가 데이터로 다시 학습시켜 가중치를 바꾸는 방식입니다. 특정 도메인의 말투나 출력 형식을 모델에 깊이 새기는 데 강하지만, 지식을 바꾸려면 다시 학습해야 하므로 자주 바뀌는 정보에는 맞지 않습니다. 학습 비용과 데이터 준비 부담도 큽니다.

반면 RAG는 모델은 그대로 두고 외부 문서를 검색해 프롬프트에 넣는 방식입니다. 문서를 추가하거나 갱신하는 것만으로 지식이 즉시 바뀌고, 답의 근거가 된 문서를 출처로 제시할 수 있어 신뢰성을 확인하기 쉽습니다.

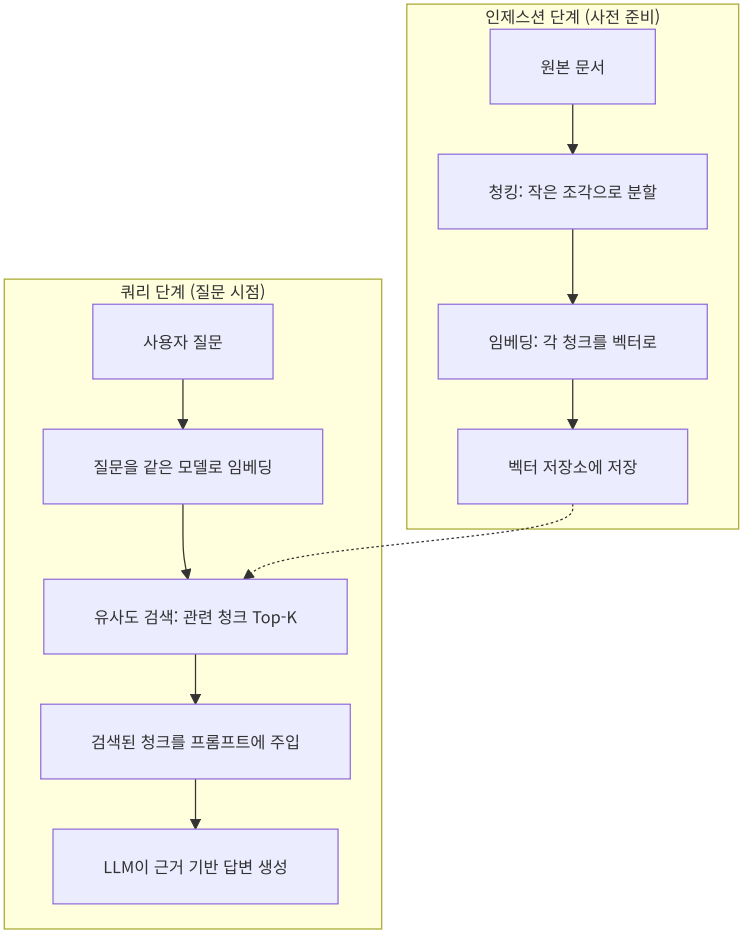
구분	RAG	Fine-tuning
바꾸는 대상	프롬프트에 넣는 외부 문서	모델 가중치
지식 갱신	문서 추가·교체로 즉시	재학습 필요
최신 정보	강함	약함(학습 시점 고정)
출처 제시	가능(검색된 문서)	어려움
말투·형식 학습	제한적	강함
초기 비용	낮음	높음

실무에서는 "지식은 RAG로, 말투와 형식은 파인튜닝으로" 나누어 병행하는 경우가 많습니다. 다만 이 책에서 집중하는 것은 자주 바뀌는 문서 기반 지식을 다루는 RAG입니다.

> **핵심 포인트:** 자주 바뀌고 출처가 중요한 지식은 RAG, 고정된 말투·형식·도메인 특화는 파인튜닝이 유리합니다. 두 기법은 경쟁이 아니라 역할 분담입니다.

4. RAG의 2단계 파이프라인

RAG 시스템은 크게 두 단계로 나뉩니다. 사용자가 질문하기 전에 문서를 미리 준비해 두는 인제스션(ingestion) 단계와, 질문이 들어왔을 때 관련 문서를 찾아 답하는 쿼리(query) 단계입니다. 이 둘을 나누는 이유는, 문서를 벡터로 바꾸어 저장하는 작업은 한 번만 해 두면 여러 질문에 재사용할 수 있기 때문입니다.



인제스션 단계에서는 원본 문서를 적당한 크기의 청크로 나누고, 각 청크를 임베딩 모델로 벡터화해 벡터 저장소에 넣습니다. 쿼리 단계에서는 사용자 질문을 같은 임베딩 모델로 벡터화하고, 저장된 청크 중 질문과 가장 가까운 것들을 유사도 검색으로 골라냅니다. 찾은 청크를 프롬프트에 근거 자료로 끼워 넣어 LLM에게 전달하면, 모델은 그 근거에 기반해 답을 생성합니다.

이 파이프라인의 각 단계, 즉 청킹과 임베딩, 유사도 검색, 검색 결과를 프롬프트에 주입해 답하는 과정은 모두 뒤이은 장들에서 프레임워크 없이 직접 구현합니다. 원리를 한 번 손으로 만들어 보면, 이후 LangChain 같은 프레임워크가 어떤 작업을 대신해 주는지 분명히 보입니다.

> **핵심 포인트:** RAG는 인제스션(문서를 벡터로 미리 준비)과 쿼리(질문 시점에 검색·생성)의 두 단계로 나뉩니다. 문서 준비는 한 번, 검색·생성은 질문마다 반복된다는 구조를 머릿속에 그려 두면 이후 모든 구현이 이 틀 위에 얹힙니다.

문서 전처리와 청킹 전략

> 긴 문서를 왜, 어떻게 작은 조각으로 나누는지 이해하고 chunk size와 overlap을 직접 다뤄 봅니다.

1. 문서 전처리: 텍스트 추출

RAG의 출발점은 원본 문서에서 순수한 텍스트를 뽑아내는 일입니다. 실제 데이터는 PDF, 워드, HTML, 일반 텍스트 등 다양한 형태지만, 임베딩 모델에 넣으려면 결국 깨끗한 문자열로 만들어야 합니다. 이 단계에서 머리글·바닥글, 페이지 번호, 의미 없는 공백 같은 잡음을 정리할수록 이후 검색 품질이 좋아집니다.

이 장에서는 원리에 집중하기 위해 이미 텍스트로 추출된 문자열을 다룹니다. PDF나 DOCX 같은 다양한 포맷을 통일된 방식으로 읽어 들이는 작업은 Day2에서 Document Loader로 표준화하므로, 여기서는 "텍스트는 이미 손에 있다"고 가정하고 시작합니다.

2. 왜 청킹이 필요한가

문서를 통째로 임베딩하지 않고 작은 조각, 즉 청크(chunk)로 나누는 데에는 분명한 이유가 있습니다.

첫째, 임베딩 모델과 LLM 모두 한 번에 처리할 수 있는 토큰 길이에 한계가 있습니다. 긴 문서를 한 벡터에 욱여넣으면 모델이 잘라 버리거나, 너무 많은 내용이 한 벡터에 뭉뚱그려져 의미가 흐려집니다.

둘째, 검색의 정밀도 때문입니다. 사용자의 질문은 보통 문서의 한 부분에만 관련됩니다. 문서 전체가 하나의 벡터라면 질문과 무관한 내용까지 함께 끌려오지만, 적당한 크기로 나누면 질문과 직접 관련된 조각만 골라낼 수 있습니다.

셋째, 비용과 프롬프트 길이 관리입니다. 검색된 청크는 그대로 프롬프트에 들어가 입력 토큰이 됩니다. 청크가 작고 관련성이 높을수록 불필요한 토큰 낭비 없이 핵심 근거만 모델에 전달할 수 있습니다.

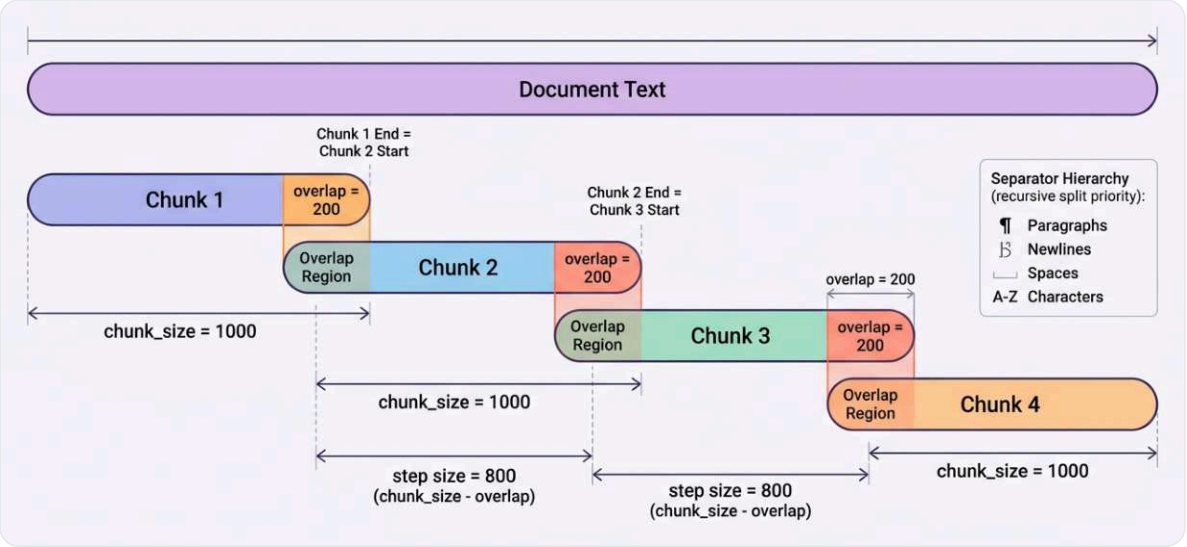
> **핵심 포인트:** 청킹의 목표는 "질문에 답하기에 딱 필요한 만큼의 의미 단위"를 만드는 것입니다. 너무 크면 잡음이 섞이고, 너무 작으면 맥락이 끊깁니다. 이 균형이 검색 품질을 좌우합니다.

3. Chunk Size와 Overlap

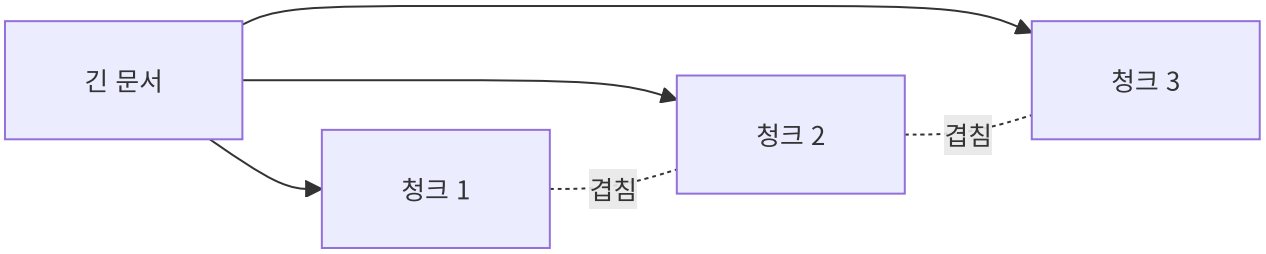
청킹에서 가장 중요한 두 가지 설정은 청크 크기(chunk size)와 겹침(overlap)입니다.

청크 크기는 한 조각에 담을 텍스트의 길이입니다. 크기가 너무 크면 한 청크에 여러 주제가 섞여 검색이 부정확해지고, 너무 작으면 문장이 끊겨 맥락을 잃습니다. 문서의 성격에 따라 적절한 크기가 다르므로, 실험으로 조정하는 값입니다.

겹침은 인접한 청크가 서로 일부 내용을 공유하도록 경계에서 조금씩 겹치게 나누는 것입니다. 겹침 없이 칼같이 자르면, 중요한 문장이 두 청크의 경계에서 반토막 나 어느 쪽에서도 온전히 검색되지 않을 수 있습니다. 약간의 겹침을 두면 경계에 걸친 내용도 최소 한 청크에는 온전히 담겨 검색 누락을 줄입니다.



청킹: 긴 문서를 의미 단위의 조각으로 나누되 경계에서 약간씩 겹치게(overlap) 분할



설정	너무 작을 때	너무 클 때
chunk size	맥락 끊김, 조각이 너무 많아짐	한 청크에 여러 주제 혼재, 검색 부정확
overlap	경계 문장 검색 누락 위험	중복 증가, 저장·비용 낭비

4. 슬라이딩 윈도우로 직접 나누기

청킹의 원리는 슬라이딩 윈도우(sliding window)로 간단히 구현할 수 있습니다. 시작 위치를 잡아 청크 크기만큼 잘라 내고, 다음 시작 위치를 (청크 크기 − 겹침)만큼 뒤로 옮기기를 문서 끝까지 반복합니다. 다음은 이 아이디어를 글자 단위로 보여 주는 최소 구현입니다.

```python
# day1 예제 구성
def chunk_text(text, chunk_size=300, overlap=50):
    chunks = []
    start = 0
    step = chunk_size - overlap          # 다음 시작 위치 이동량
    while start < len(text):
        end = start + chunk_size
        chunk = text[start:end]          # 현재 윈도우 잘라내기
        chunks.append(chunk)
        start += step                    # 겹침만큼 되돌려 다음 윈도우로
    return chunks

document = "..."  # 전처리로 추출된 긴 텍스트
chunks = chunk_text(document, chunk_size=300, overlap=50)
print(f"청크 개수: {len(chunks)}")
for i, c in enumerate(chunks):
    print(f"[{i}] {c[:30]}...")
```

이 함수는 `start`부터 청크 크기만큼 잘라 청크를 만들고, 다음 청크의 시작을 `chunk_size - overlap`만큼 전진시킵니다. 그래서 인접한 두 청크는 `overlap` 글자만큼 겹칩니다. `overlap`을 0으로 두면 겹침 없이 단순 분할이 되고, 키우면 겹침이 늘어납니다. 실제 RAG 파이프라인에서는 이렇게 만든 각 청크를 임베딩해 저장합니다.

실무에서는 글자 수로 자르는 대신 문장이나 문단 경계를 살려 자르는 편이 자연스럽습니다. 문장 중간에서 끊기지 않게 구분자(마침표, 줄바꿈 등)를 우선해 나누면 각 청크가 더 온전한 의미 단위가 됩니다. 이런 정교한 분할은 Day2에서 Text Splitter로 표준화해 다루지만, 그 안에서 일어나는 일은 결국 위 슬라이딩 윈도우의 확장입니다.

> **핵심 포인트:** 청킹은 "크기로 자르되 경계에서 겹친다"는 단순한 규칙의 반복입니다. chunk size와 overlap은 정답이 정해진 값이 아니라, 문서 성격과 검색 품질을 보며 조정하는 튜닝 파라미터입니다.

5. 정리

문서 전처리와 청킹은 RAG 인제스션 단계의 첫 관문입니다. 원본에서 깨끗한 텍스트를 뽑고 적절한 크기의 겹치는 조각으로 나누면, 이후 임베딩과 검색이 정밀하게 동작할 토대가 마련됩니다. 청크 크기와 겹침은 검색 품질에 직접 영향을 주는 핵심 설정이며, 슬라이딩 윈도우라는 단순한 원리로 직접 구현해 보면 프레임워크가 내부에서 무엇을 하는지 분명히 이해할 수 있습니다. 이렇게 만든 청크를 벡터로 바꾸는 임베딩 작업이 다음 단계입니다.

임베딩 이해

> 텍스트를 의미가 담긴 벡터로 바꾸는 임베딩의 원리를 이해하고, OpenAI API와 로컬 모델 두 방식으로 직접 생성해 봅니다.

1. 벡터란 무엇인가

벡터(vector)는 숫자들의 나열입니다. 임베딩에서 다루는 벡터는 수백에서 수천 개의 실수로 이루어진 고정 길이의 배열이고, 텍스트 하나가 배열 하나에 대응됩니다. 핵심은 이 숫자들이 무작위가 아니라 텍스트의 의미를 담도록 학습된 값이라는 데 있습니다.

임베딩 벡터의 각 차원은 사람이 이름 붙일 수 있는 명확한 속성은 아니지만, 전체로 보면 그 텍스트의 의미적 특성을 나눠 담습니다. 그래서 두 텍스트의 의미가 비슷하면 두 벡터가 비슷한 방향을, 무관하면 다른 방향을 가리킵니다. 이 성질이 검색의 토대입니다.

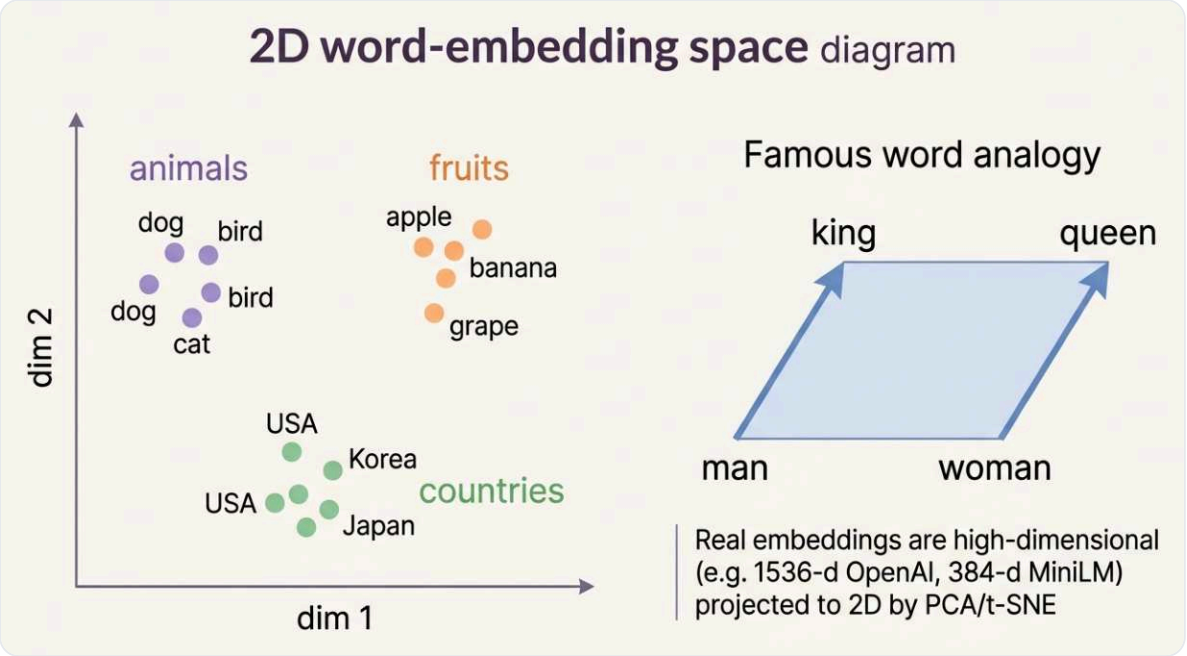
```python
# 12.study/6.embedding/2.embedding_visualize.py
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("paraphrase-multilingual-MiniLM-L12-v2")
emb = model.encode("인공지능이 세상을 바꾸고 있다.")
print(f"  벡터 차원: {emb.shape[0]}")          # 예: 384
print(f"  처음 10개 값: {emb[:10].round(4)}")  # 실수들의 배열
```

이 코드는 한 문장을 고정 길이 벡터로 바꾸고 그 차원 수와 앞부분 값을 보여 줍니다. 출력된 숫자 배열은 사람이 읽기 어렵지만, 이 배열의 방향이 곧 문장의 의미입니다.

> **핵심 포인트:** 임베딩은 텍스트의 '의미'를 좌표로 옮긴 것입니다. 의미가 비슷하면 벡터가 가깝다는 단 하나의 성질이, 이후 유사도 검색의 모든 동작을 가능하게 합니다.

2. 의미 공간

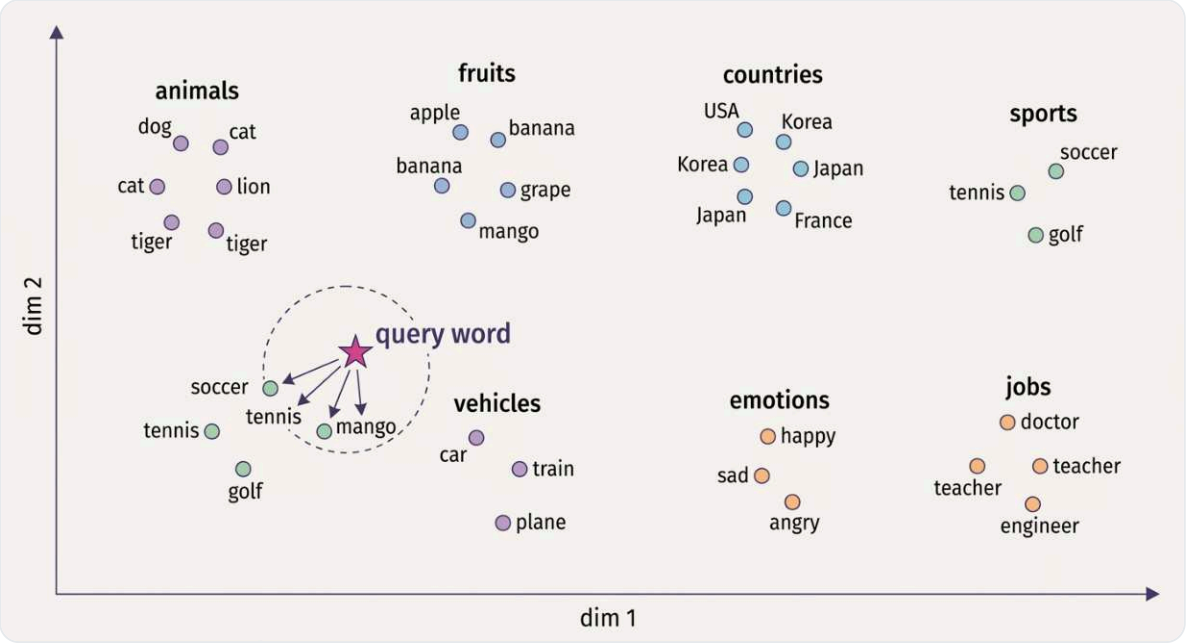
임베딩으로 만든 벡터들이 놓이는 공간을 의미 공간(semantic space) 또는 임베딩 공간이라 부릅니다. 이 공간에서는 의미가 가까운 텍스트가 서로 가까이 모이고, 무관한 텍스트는 멀리 떨어집니다. 동물에 관한 문장끼리, 과일에 관한 문장끼리 자연스럽게 군집을 이룹니다.



임베딩 공간: 'man→king'과 'woman→queen'처럼 의미 관계가 같으면 평행한 벡터로 나타나고, 비슷한 단어는 서로 가까이 모인다

예를 들어 "인공지능이 세상을 바꾸고 있다"와 "AI가 세계를 변화시키고 있다"는 표현이 다르지만 의미 공간에서 거의 같은 위치에 놓입니다. 반면 "오늘 날씨가 좋다"와 "피자를 먹고 싶다"는 멀리 떨어집니다. 좋은 임베딩 모델은 표면적인 단어가 달라도 의미가 같으면 가깝게, 단어가 비슷해도 의미가 다르면 멀게 배치합니다. 다국어 임베딩 모델은 한국어와 영어처럼 언어가 달라도 같은 의미면 가까이 두어, 교차 언어 검색까지 가능하게 합니다.

단어가 둘셋이 아니라 수만 개로 늘어나면, 의미 공간은 주제별 군집으로 가득 찬 '은하'처럼 보입니다. 질문 단어가 들어오면 같은 공간에서 가장 가까운 단어들을 찾는 것이 곧 검색의 출발점입니다.



임베딩 공간이 수많은 단어의 의미별 군집으로 가득 찬 모습. 질문 단어가 들어오면 가장 가까운(유사한) 단어 위치를 찾는다

3. OpenAI 임베딩 API

임베딩을 만드는 가장 손쉬운 방법은 OpenAI의 임베딩 API입니다. `client.embeddings.create`에 텍스트와 모델명을 넣으면 벡터가 돌아옵니다.

```python
# 1.openai/7.rag/1.rag_basic.py
import numpy as np
from openai import OpenAI

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

def get_embedding(text):
    response = client.embeddings.create(input=text, model="text-embedding-ada-002")
    return np.array(response.data[0].embedding)
```

이 함수는 텍스트를 임베딩 API에 보내고, 응답에서 벡터를 꺼내 넘파이 배열로 돌려줍니다. 응답의 `data[0].embedding`이 실제 벡터이며, OpenAI 임베딩은 보통 1536차원입니다. API 방식은 별도 모델을 내려받지 않아도 되고 품질이 안정적이지만, 호출마다 비용이 들고 인터넷 연결과 API 키가 필요합니다.

4. 로컬 임베딩 모델

임베딩을 OpenAI API 대신 로컬 모델로 만들 수도 있습니다. `sentence-transformers` 라이브러리로 모델을 한 번 내려받으면, 이후 임베딩 생성은 비용 없이 인터넷 연결과 API 키 없이 동작합니다. 응답을 생성하는 LLM은 그대로 API를 쓰더라도, 임베딩만 로컬로 돌려 비용을 줄이는 구성이 가능합니다.

```python
# 1.openai/7.rag/3.rag_local_embedding.py
import numpy as np
from sentence_transformers import SentenceTransformer

# 로컬 임베딩 모델 (최초 실행 시 자동 다운로드, 약 80MB)
embedding_model = SentenceTransformer("all-MiniLM-L6-v2")

documents = [
    "Python은 강력한 프로그래밍 언어입니다.",
    "OpenAI는 AI 연구를 선도하는 기업입니다.",
    "FAISS는 벡터 검색을 위한 라이브러리입니다.",
]

# 임베딩 차원은 모델마다 다르다 (OpenAI 1536 vs MiniLM 384)
doc_embeddings = np.array(embedding_model.encode(documents))
```

이 코드는 로컬 모델을 불러와 여러 문서를 한 번에 임베딩합니다. `encode`에 리스트를 넣으면 각 문서의 벡터를 모은 2차원 배열이 돌아옵니다. 주의할 점은 모델마다 벡터 차원이 다르다는 것입니다. OpenAI는 1536차원, MiniLM은 384차원이므로, 차원을 코드에 직접 박지 말고 임베딩 결과의 형태(`shape`)에서 받아 와야 합니다.

임베딩 모델을 더 정확한 것으로 바꾸면 검색 품질이 올라갑니다. 모델만 교체하면 같은 코드로 결과를 비교할 수 있습니다.

```python
# 1.openai/7.rag/5.rag_better_embedding_model.py
# all-MiniLM-L6-v2  : 384차원,  80MB, 빠름 / 품질 보통
# all-mpnet-base-v2 : 768차원, 420MB, 느림 / 품질 우수
model_name = "all-mpnet-base-v2"
embedding_model = SentenceTransformer(model_name)
```

방식	모델 예시	차원	특징
OpenAI API	text-embedding-ada-002	1536	안정적 품질, 호출당 비용·키 필요
로컬(가벼움)	all-MiniLM-L6-v2	384	빠름, 무료, 품질 보통
로컬(정확함)	all-mpnet-base-v2	768	느림, 무료, 품질 우수

5. 한 가지 철칙: 문서와 질문은 같은 모델로

임베딩을 다룰 때 절대 어겨선 안 되는 규칙이 하나 있습니다. 문서를 저장할 때 쓴 임베딩 모델과 질문을 임베딩할 때 쓰는 모델이 반드시 같아야 한다는 것입니다. 서로 다른 모델로 만든 벡터는 다른 의미 공간에 놓이므로, 한 공간에서 거리를 비교하는 것 자체가 무의미해집니다.

```
# 1.openai/7.rag/3.rag_local_embedding.py
def rag_query(user_query):
    # 질문도 '저장 때와 같은 로컬 모델'로 임베딩해야 한다 (필수 규칙)
    query_embedding = np.array([embedding_model.encode(user_query)])
    distances, indices = index.search(query_embedding, k=1)
    ...
```

또한 모델을 바꾸면 차원도 달라져, 저장된 벡터와 질문 벡터의 차원이 어긋나 검색 자체가 불가능해집니다. 따라서 임베딩 모델을 교체할 때는 반드시 저장소(인덱스)를 새 모델로 다시 구축해야 합니다.

> **핵심 포인트:** "문서와 질문은 같은 임베딩 모델로." 이 한 줄을 어기면 검색이 통째로 망가집니다. 모델을 바꾸면 차원과 의미 공간이 모두 달라지므로 인덱스를 반드시 재구축합니다.

유사도 검색 직접 구현

> 코사인 유사도를 넘파이로 직접 구현하고, 질문과 가장 가까운 문서 Top-K를 골라내는 검색의 핵심을 손으로 만듭니다.

1. 유사도를 재는 두 가지 척도

임베딩으로 텍스트를 벡터로 바꾸면, 두 텍스트가 의미상 얼마나 비슷한지를 벡터 사이의 관계로 계산할 수 있습니다. 대표적인 척도는 두 가지입니다. 하나는 두 벡터 사이의 직선 거리를 재는 L2(유클리드) 거리이고, 다른 하나는 두 벡터가 이루는 각도를 재는 코사인 유사도(cosine similarity)입니다.

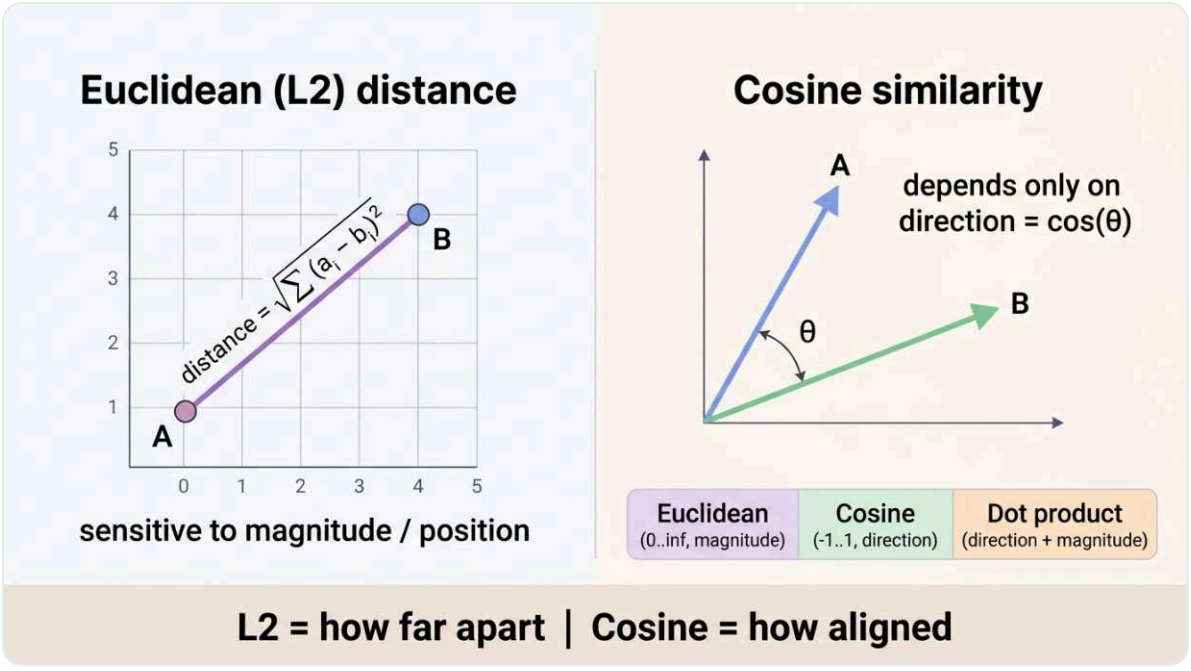
두 척도의 성격은 다릅니다. L2 거리는 벡터의 크기와 방향을 모두 보지만, 코사인 유사도는 오직 방향만 봅니다. 텍스트 검색에서는 코사인이 더 선호되는데, 이유는 문서 길이에 휘둘리지 않고, 값 범위가 −1에서 1로 고정되어 임계값을 일관되게 정할 수 있으며, 임베딩 모델 자체가 코사인 기준으로 학습되는 경우가 많기 때문입니다.

척도	보는 것	값 범위	해석
L2 거리	크기 + 방향	0 ~ ∞	작을수록 유사(상한 없음)
코사인 유사도	방향만	-1 ~ 1	1=같은 방향, 0=무관, -1=정반대

> **핵심 포인트:** 텍스트 검색에서는 코사인 유사도가 기본입니다. 방향(의미)만 보므로 문서 길이에 흔들리지 않고, 값이 −1~1로 고정돼 "유사도 0.2 미만은 무관" 같은 임계 판단이 일관되게 작동합니다.

거리·유사도 척도, 무엇을 언제 쓰나

유사도를 재는 척도는 코사인과 L2만 있는 것이 아닙니다. 실무에서 자주 만나는 네 가지를 한자리에 놓고 보면 선택 기준이 분명해집니다.



유클리드(L2) 거리와 코사인 유사도의 직관 비교 — L2는 두 점 사이의 직선거리, 코사인은 두 벡터의 각도

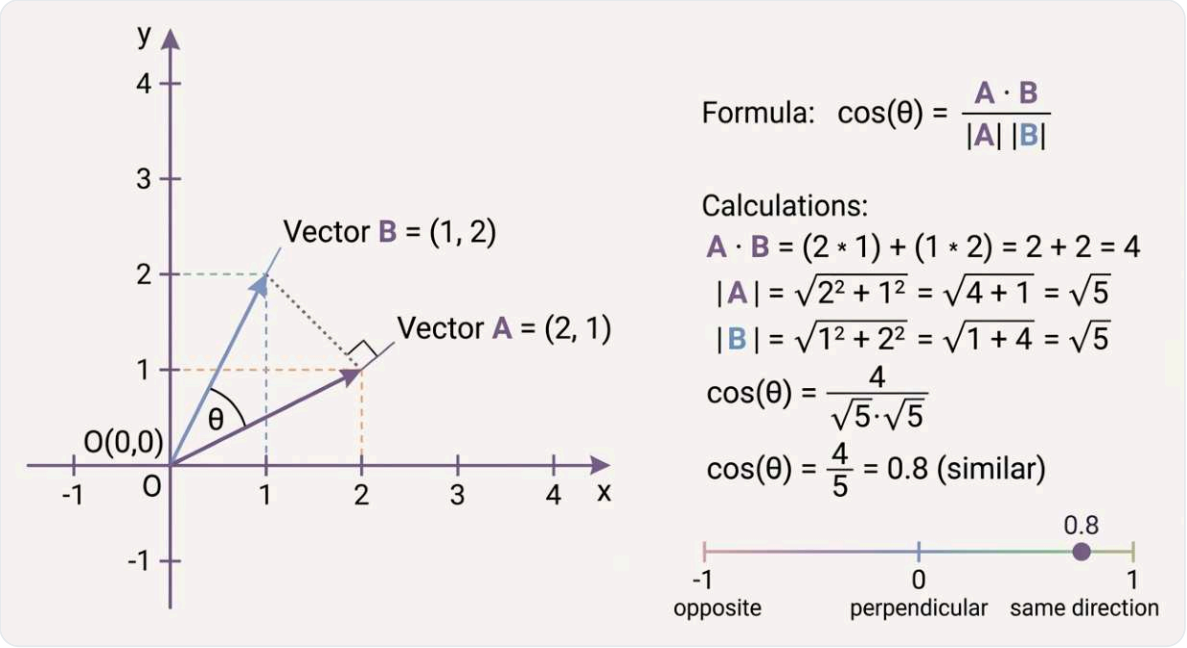
척도	재는 것	값 범위	크기에 민감?	주로 쓰는 곳
유클리드(L2)	두 점 사이의 직선거리	0 ~ ∞	예	좌표·이미지 특징처럼 절대 위치가 중요한 데이터
코사인	두 벡터의 방향(각도)	-1 ~ 1	아니오	텍스트 임베딩 검색(문서 길이에 둔감)
내적(Dot)	방향과 크기를 함께	-∞ ~ ∞	예	정규화한 벡터에서는 코사인과 동일, 추천·랭킹
맨해튼(L1)	축마다 이동한 거리의 합	0 ~ ∞	예	고차원 희소 데이터, 이상치에 덜 민감

척도를 고르는 기준은 결국 한 가지 질문으로 모입니다. "벡터의 크기가 의미를 갖는가"입니다. 좌표나 이미지 특징처럼 절대적인 위치와 거리가 중요한 데이터에서는 크기를 함께 보는 L2가 자연스럽습니다. 반면 텍스트 임베딩은 같은 주제라도 문서가 길면 벡터의 크기가 커지므로, 크기에 휘둘리지 않고 방향만 보는 코사인이 적합합니다. 내적은 방향과 크기를 동시에 반영하지만, 벡터를 길이 1로 정규화하면 내적이 곧 코사인 유사도가 되어 더 적은 연산으로 같은 결과를 얻습니다. 그래서 많은 벡터 검색 엔진이 내부적으로 벡터를 정규화한 뒤 내적으로 코사인을 계산합니다. 맨해튼 거리는 차원이 매우 높거나 값이 듬성듬성한(희소한) 데이터에서 L2보다 안정적으로 동작하는 경우가 있습니다.

> **핵심 포인트:** 텍스트 RAG에서 코사인이 표준이 된 이유는 "방향이 의미를 담고 크기는 문서 길이에 따라 달라지기" 때문입니다. 벡터를 정규화하면 내적이 곧 코사인이므로, 실무에서는 정규화 후 내적으로 빠르게 코사인을 계산하는 방식을 흔히 씁니다.

2. 코사인 유사도의 수식과 직관

코사인 유사도는 두 벡터가 이루는 각도의 코사인 값입니다. 각도가 작아 두 벡터가 같은 방향을 가리키면 1에 가깝고, 직각이면 0, 정반대면 −1이 됩니다.



코사인 유사도: 두 벡터가 이루는 각도로 의미적 유사성을 측정 (각도가 작을수록 유사)

수식으로는 두 벡터의 내적을 각 벡터 길이의 곱으로 나눈 값입니다. 즉 `cos(θ) = (A · B) / (¦¦A¦¦ × ¦¦B¦¦)`입니다. 분자의 내적은 두 벡터의 같은 차원끼리 곱해 더한 값이고, 분모는 각 벡터의 크기(노름)입니다. 이를 넘파이로 그대로 옮기면 다음과 같습니다.

```python
# day1 예제 구성
import numpy as np

def cosine_similarity(a, b):
    dot = np.dot(a, b)                          # 내적 (분자)
    norm = np.linalg.norm(a) * np.linalg.norm(b)  # 두 벡터 크기의 곱 (분모)
    return dot / norm                           # -1 ~ 1 사이 값

# 두 벡터의 방향이 비슷할수록 1에 가깝다
sim = cosine_similarity(query_vec, doc_vec)
print(f"코사인 유사도: {sim:.4f}")
```

이 함수는 코사인 유사도 정의를 그대로 코드로 옮긴 것입니다. `np.dot`으로 내적을 구하고, `np.linalg.norm`으로 각 벡터의 크기를 구해 나눕니다. 벡터를 미리 길이 1로 정규화해 두면 분모가 1이 되어, 코사인 유사도가 단순한 내적과 같아집니다.

저장소의 RAG 예제는 검색 라이브러리를 쓰되, 거리를 사람이 읽기 쉬운 점수로 바꾸는 과정을 보여 줍니다. L2 거리를 0~1 점수로 변환하고, 점수가 너무 낮으면 관련 문서가 없다는 신호로 경고합니다.

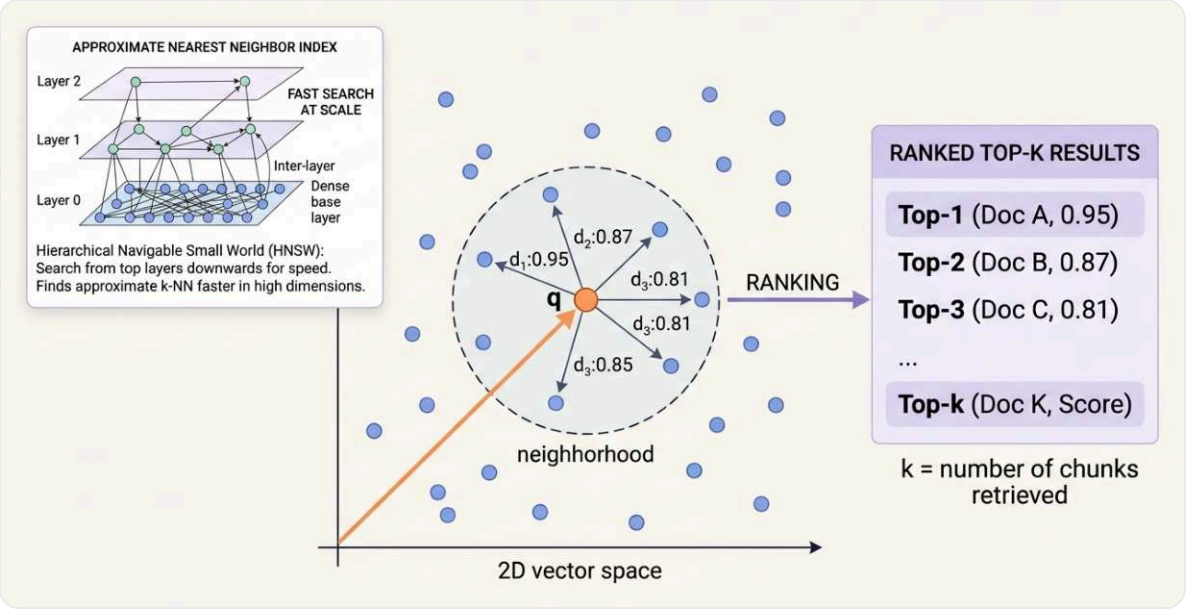
```
# 1.openai/7.rag/2.rag_similarity_score.py
# FAISS의 distance는 '제곱된 L2 거리' → 점수로 변환
true_distance = np.sqrt(distances[0][0])
similarity_score = 1 / (1 + true_distance)   # 0~1, 거리가 작을수록 1에 가깝다
print(f"유사도 점수: {similarity_score:.4f}")

# 점수가 너무 낮으면 = 질문과 관련된 문서가 없다는 신호
if similarity_score < 0.2:
    print("경고: 질문과 관련된 문서를 찾지 못했을 수 있습니다.")
```

코사인 유사도를 쓰면 이런 변환이 필요 없다는 점도 중요합니다. 벡터를 정규화하고 내적 기반으로 검색하면, 결과값이 곧 코사인 유사도라 그대로 읽으면 됩니다.

```
# 1.openai/7.rag/4.rag_cosine_similarity.py
# 코사인 유사도 = '정규화된 벡터의 내적(Inner Product)'
doc_embeddings = np.array(embedding_model.encode(documents), dtype=np.float32)
faiss.normalize_L2(doc_embeddings)                  # ① 벡터를 길이 1로 정규화
index = faiss.IndexFlatIP(doc_embeddings.shape[1])  # ② 내적 기반 인덱스
index.add(doc_embeddings)

# 질문 벡터도 똑같이 정규화 → 검색 결과값이 곧 코사인 유사도
faiss.normalize_L2(query_embedding)
similarities, indices = index.search(query_embedding, k=1)
```

3. Top-K 검색

실제 검색에서는 가장 가까운 문서 하나만이 아니라, 가까운 순서대로 상위 K개를 골라냅니다. 이를 Top-K 검색이라 합니다. 질문 벡터와 모든 문서 벡터의 유사도를 계산한 뒤, 유사도가 높은 순으로 정렬해 앞에서 K개를 취합니다.

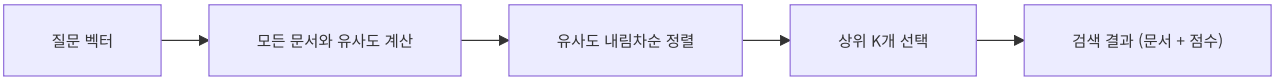


Top-K 검색: 질문 벡터와 가장 가까운 K개의 문서 벡터를 골라낸다

라이브러리 없이 넘파이만으로 Top-K를 구현하면 검색의 본질이 그대로 드러납니다. 모든 문서에 대해 유사도를 계산하고, 정렬해 상위 K개의 인덱스를 뽑는 것이 전부입니다.

```python
# day1 예제 구성
import numpy as np

def top_k_search(query_vec, doc_vecs, k=2):
    # 모든 문서에 대해 코사인 유사도 계산
    sims = [cosine_similarity(query_vec, d) for d in doc_vecs]
    # 유사도 내림차순으로 정렬한 인덱스에서 상위 k개
    ranked = np.argsort(sims)[::-1][:k]
    return [(i, sims[i]) for i in ranked]

results = top_k_search(query_vec, doc_embeddings, k=2)
for idx, score in results:
    print(f"문서 {idx}: 유사도 {score:.4f} → {documents[idx]}")
```

이 함수는 질문 벡터와 모든 문서 벡터의 유사도를 리스트로 구한 뒤, `np.argsort`로 유사도 순서를 매기고 `[::-1]`로 내림차순으로 뒤집어 상위 K개의 인덱스를 반환합니다. FAISS 같은 검색 라이브러리는 내부적으로 같은 일을 하되, 문서가 수백만 개로 늘어나도 빠르게 동작하도록 최적화한 것입니다. 저장소 예제에서는 `index.search(query_embedding, k=...)`의 `k` 인자가 바로 이 Top-K의 K에 해당합니다.



4. 검색 품질 점검

Top-K로 문서를 골랐다고 해서 그것이 항상 정답과 관련 있는 것은 아닙니다. 질문과 관련된 문서가 애초에 저장소에 없으면, 검색은 그나마 가장 가까운(그러나 여전히 무관한) 문서를 돌려줍니다. 그래서 유사도 점수를 함께 확인해, 점수가 일정 임계값보다 낮으면 관련 문서를 찾지 못했다고 판단하는 점검이 중요합니다. 무관한 문서를 그대로 LLM에 근거로 넘기면 잘못된 답이 나오기 때문입니다.

코사인 유사도는 값 범위가 고정돼 있어 이런 임계값을 정하기 쉽습니다. 검색 결과의 점수가 임계값 아래면 "관련 문서 없음"으로 처리하거나, 모델에게 "근거가 부족하면 모른다고 답하라"는 지시를 함께 주는 식으로 환각을 추가로 방지할 수 있습니다.

> **핵심 포인트:** 검색은 "가장 가까운 것"을 줄 뿐 "정말 관련 있는 것"을 보장하지 않습니다. 유사도 점수를 임계값과 비교해 무관한 검색 결과를 걸러 내는 것이, 잘못된 근거로 인한 환각을 막는 마지막 안전장치입니다.

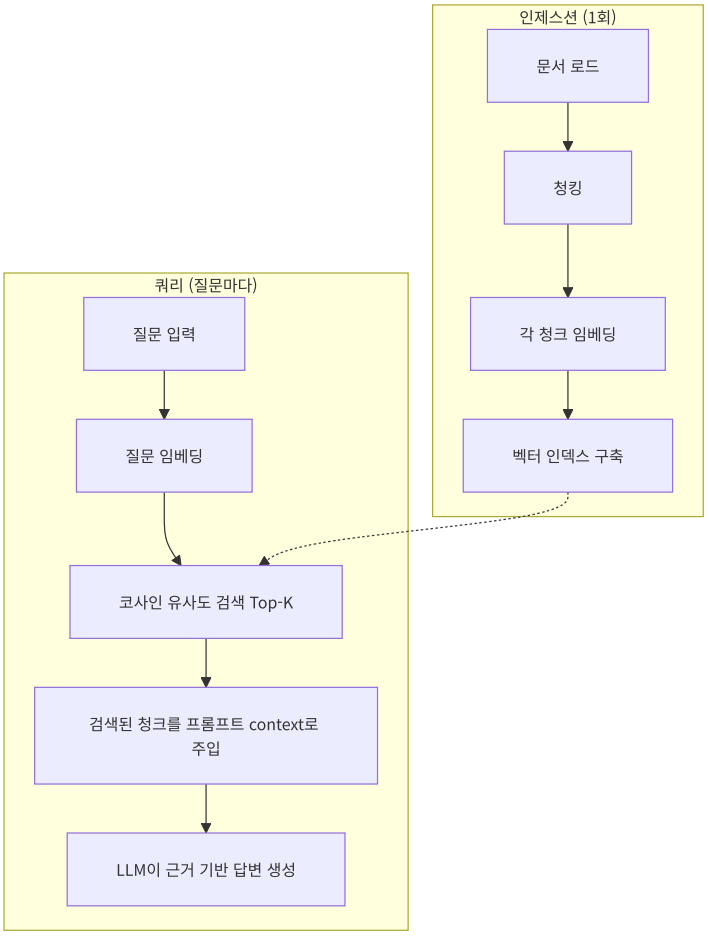
미니 프로젝트: 손코딩 RAG QA

> 앞에서 익힌 임베딩·검색·프롬프트 주입을 하나로 엮어, 프레임워크 없이 동작하는 RAG QA 파이프라인을 완성합니다.

1. 무엇을 만드는가

이 미니 프로젝트의 목표는 텍스트 문서를 근거로 질문에 답하는 RAG QA 시스템을 프레임워크 없이 직접 조립하는 것입니다. 앞에서 따로 익힌 임베딩, 유사도 검색, 그리고 검색 결과를 프롬프트에 주입해 LLM에게 답하게 하는 과정을 하나의 흐름으로 잇습니다. 라이브러리가 대신 해 주던 일을 손으로 한 번 만들어 보면, 이후 프레임워크가 어떤 단계를 자동화하는지 분명하게 보입니다.

전체 흐름은 인제스션과 쿼리 두 단계로 나뉩니다. 인제스션에서는 문서를 임베딩해 저장소를 만들고, 쿼리에서는 질문을 임베딩해 관련 문서를 찾아 프롬프트에 넣고 답을 생성합니다.



2. 준비: 클라이언트와 문서

먼저 환경변수에서 API 키를 읽어 클라이언트를 만들고, 검색 대상이 될 문서(지식 베이스)를 준비합니다. 실제로는 텍스트 파일을 읽어 청킹한 결과가 이 자리에 들어가지만, 여기서는 원리에 집중하기 위해 짧은 문서 리스트로 시작합니다. 각 문서가 하나의 청크라고 보면 됩니다.

```python
# 1.openai/7.rag/1.rag_basic.py
import os
import numpy as np
import faiss
from openai import OpenAI
from dotenv import load_dotenv

load_dotenv(dotenv_path='../.env')
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# 검색 대상 문서 (지식 베이스)
documents = [
    "Python은 강력한 프로그래밍 언어입니다.",
    "OpenAI는 AI 연구를 선도하는 기업입니다.",
    "FAISS는 벡터 검색을 위한 라이브러리입니다.",
]
```

API 키는 코드에 직접 적지 않고 `os.getenv("OPENAI_API_KEY")`로 환경변수에서 읽습니다. 텍스트 파일을 다룬다면 이 단계에서 파일을 읽어 앞 장의 청킹 함수로 잘게 나눈 뒤 `documents`에 담으면 됩니다.

3. 인제스션: 문서를 벡터로 만들어 저장

각 문서를 임베딩해 검색 가능한 인덱스를 구축합니다. 임베딩 함수는 텍스트를 OpenAI 임베딩 API에 보내 벡터를 받아오고, 모든 문서 벡터를 인덱스에 추가합니다.

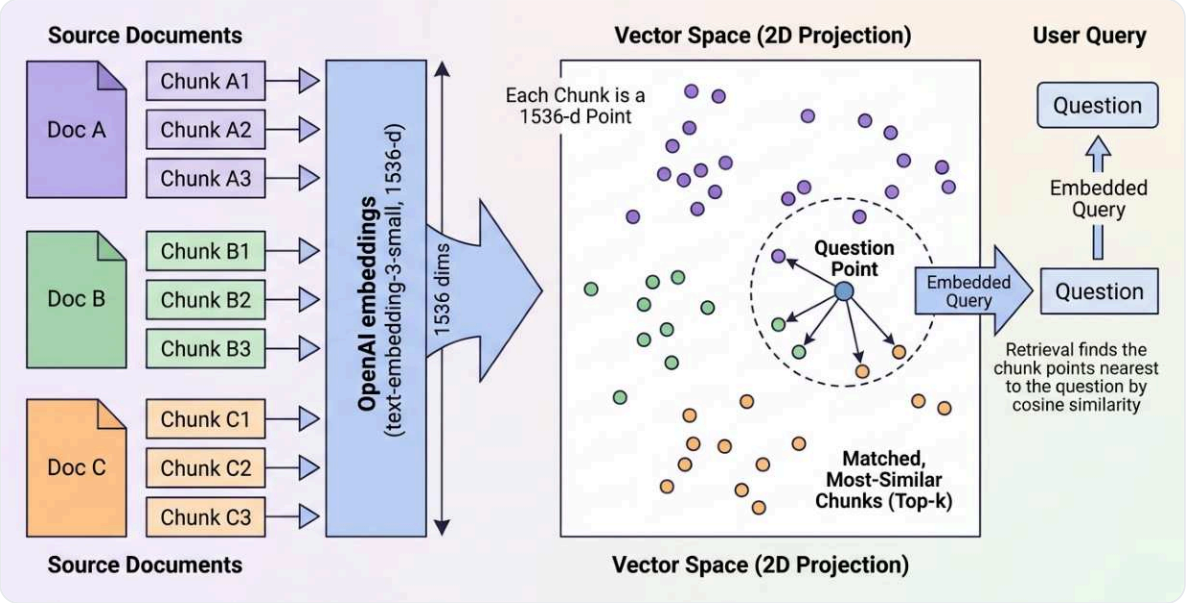
```python
# 1.openai/7.rag/1.rag_basic.py
def get_embedding(text):
    response = client.embeddings.create(input=text, model="text-embedding-ada-002")
    return np.array(response.data[0].embedding)

# 문서 벡터를 인덱스에 저장 (1536 = 임베딩 차원)
index = faiss.IndexFlatL2(1536)
doc_embeddings = np.array([get_embedding(doc) for doc in documents])
index.add(doc_embeddings)
```

여기서 `get_embedding`은 한 텍스트를 벡터로 바꾸고, 리스트 컴프리헨션으로 모든 문서를 임베딩해 한꺼번에 인덱스에 넣습니다. 이 인제스션은 한 번만 해 두면 이후 여러 질문에 재사용됩니다. 라이브러리 없이 순수 넘파이로 만들 경우, 이 인덱스 자리에는 앞 장에서 구현한 `top_k_search`가 사용할 `doc_embeddings` 배열을 그대로 보관하면 됩니다.

4. 쿼리: 검색하고 context를 주입해 답하기

질문이 들어오면 같은 임베딩 모델로 질문을 벡터화하고, 가장 가까운 문서를 검색합니다. 검색된 문서를 프롬프트에 참고 정보(context)로 끼워 넣어 LLM에게 전달하는 것이 RAG의 핵심입니다.

여러 문서를 청킹한 문장들은 모두 같은 임베딩 공간의 점이 됩니다. 질문도 같은 차원(예: 1536차원)으로 임베딩되어, 그 점과 가장 가까운 청크 점들을 골라내는 것이 검색의 실제 동작입니다.



여러 문서를 청킹한 문장을 1536차원으로 임베딩해 벡터 공간의 점으로 두고, 질문도 같은 차원으로 임베딩해 가장 유사한 청크를 매칭(Top-K)

```python
# 1.openai/7.rag/1.rag_basic.py
def rag_query(user_query):
    # ① 질문도 '같은 임베딩 모델'로 벡터화 (같은 공간에서 비교)
    query_embedding = get_embedding(user_query)

    # ② 가장 가까운 문서 k개 검색 (여기서는 k=1)
    distances, indices = index.search(np.array([query_embedding]), k=1)
    retrieved_doc = documents[indices[0][0]]

    # ③ 검색된 문서를 프롬프트에 끼워넣어 GPT에게 전달 = RAG의 핵심
    prompt = f"참고 정보: {retrieved_doc}\n\n질문: {user_query}\n답변:"
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "당신은 친절한 AI 도우미입니다."},
            {"role": "user", "content": prompt},
        ],
    )
    return response.choices[0].message.content

query = "Python은 어떤 언어인가요?"
print(rag_query(query))
```

이 `rag_query` 함수가 쿼리 단계의 전부입니다. 질문을 임베딩하고(①), 인덱스에서 가장 가까운 문서를 찾고(②), 그 문서를 `참고 정보:`로 시작하는 프롬프트에 넣어 LLM에 보냅니다(③). 모델은 자신의 내부 기억이 아니라 프롬프트에 주어진 검색 결과를 근거로 답하게 됩니다. 이것이 환각을 줄이고 최신·사내 정보를 답할 수 있게 하는 RAG의 작동 원리입니다.

5. 검색 품질을 함께 확인하기

실전에서는 검색이 제대로 됐는지 점검하는 단계를 더합니다. 검색된 문서의 유사도 점수를 확인하고, 점수가 너무 낮으면 관련 문서가 없다는 경고를 띄웁니다. 또한 프롬프트의 토큰 수를 측정해 비용과 컨텍스트 길이를 관리합니다.

```python
# 1.openai/7.rag/2.rag_similarity_score.py
import tiktoken

def rag_query(user_query):
    query_embedding = get_embedding(user_query)
    distances, indices = index.search(np.array([query_embedding]), k=1)
    retrieved_doc = documents[indices[0][0]]

    # 거리 → 0~1 유사도 점수로 변환 후 저품질 경고
    true_distance = np.sqrt(distances[0][0])
    similarity_score = 1 / (1 + true_distance)
    if similarity_score < 0.2:
        print("경고: 질문과 관련된 문서를 찾지 못했을 수 있습니다.")

    prompt = f"참고 정보: {retrieved_doc}\n\n질문: {user_query}\n답변:"

    # 프롬프트 토큰 수 측정 (비용·컨텍스트 길이와 직결)
    token_count = len(tiktoken.get_encoding("o200k_base").encode(prompt))
    print(f"프롬프트 토큰 수: {token_count}")

    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "당신은 제공된 정보만을 바탕으로 답변합니다."},
            {"role": "user", "content": prompt},
        ],
    )
    return response.choices[0].message.content
```

여기서 system 메시지가 "당신은 제공된 정보만을 바탕으로 답변합니다"로 바뀐 점에 주목합니다. 이렇게 지시하면 모델이 검색된 근거를 벗어나 추측하는 것을 억제해, 환각을 한 번 더 막을 수 있습니다. 유사도 점수 점검과 결합하면, 근거가 부족할 때 잘못 답하는 상황을 효과적으로 줄입니다.

6. 전체 파이프라인 정리

지금까지의 조각을 모으면, 프레임워크 없이도 동작하는 완전한 RAG QA가 완성됩니다. 흐름을 한눈에 정리하면 다음과 같습니다.

단계	하는 일	사용한 도구
로드·청킹	문서를 읽어 작은 조각으로	파일 입출력 + 슬라이딩 윈도우
임베딩	각 청크를 벡터로	embeddings.create / 로컬 모델
인덱스 구축	벡터를 검색 가능하게 저장	인덱스 또는 넘파이 배열
검색	질문과 가까운 Top-K 청크	코사인 유사도
주입·생성	청크를 프롬프트에 넣어 답변	chat.completions.create

이 미니 프로젝트로 RAG의 뼈대를 손으로 세웠습니다. 문서를 벡터로 만들어 저장하고, 질문과 가까운 근거를 찾아 프롬프트에 주입하고, 그 근거에 기반해 모델이 답하게 하는 흐름이 RAG의 전부입니다. 여기서 직접 구현한 임베딩, 코사인 검색, Top-K, 프롬프트 주입은 이후 LangChain 같은 프레임워크에서 한 줄의 호출로 대체되지만, 그 한 줄 뒤에서 일어나는 일은 바로 이 미니 프로젝트에서 만든 그 흐름입니다.

> **핵심 포인트:** RAG는 "검색해서 찾은 근거를 프롬프트에 넣어 답하게 한다"는 한 문장으로 요약됩니다. 그 한 문장을 프레임워크 없이 직접 구현해 본 경험이, 이후의 모든 RAG 시스템을 이해하고 디버깅하는 토대가 됩니다.

DAY

2

LangChain 기반 RAG 구축

프레임워크로 RAG 파이프라인을 표준화한다

PART 3 LangChain 기초

PART 4 벡터 검색과 RAG 체인

PART 03

LangChain 기초

1. LangChain과 LCEL
2. 프롬프트 엔지니어링
3. Document Loader
4. Text Splitter

LangChain과 LCEL

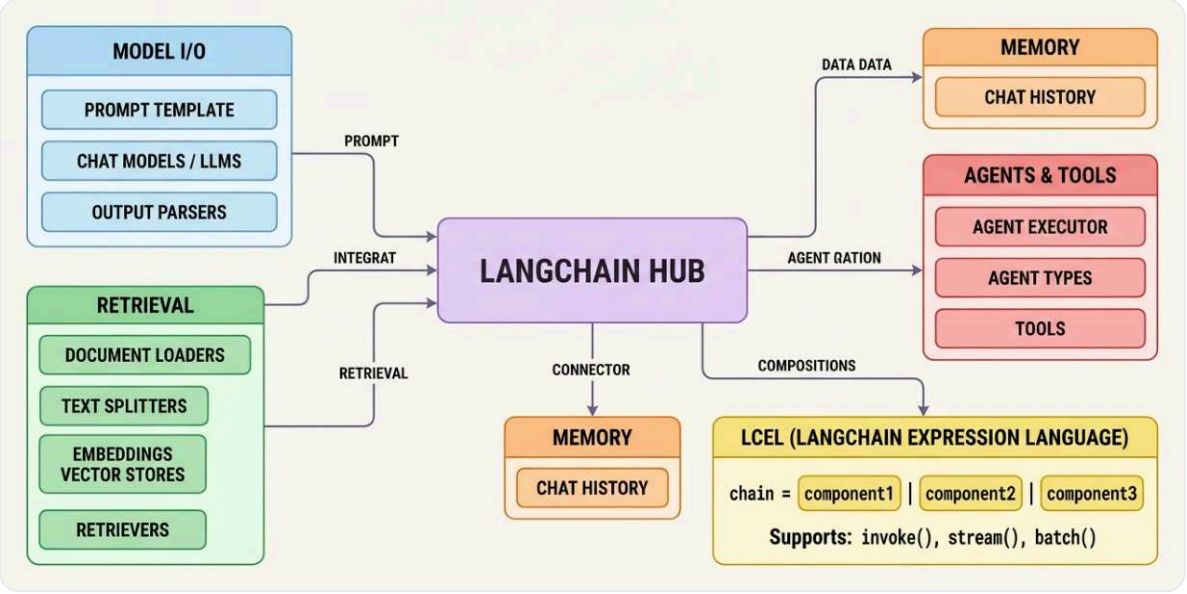
> 흩어진 LLM 호출 코드를 파이프 하나로 꿰는 표준 조립 언어, LCEL을 배웁니다.

1. 왜 LangChain인가

RAG의 핵심 원리를 직접 구현하면 임베딩 생성, 코사인 유사도 계산, Top-K 선택, 프롬프트 조립, LLM 호출이 모두 개별 함수로 흩어집니다. 검색기를 바꾸거나, 출력 파서를 끼우거나, 스트리밍을 추가할 때마다 호출 순서를 손으로 다시 엮어야 하고, 컴포넌트 사이의 입출력 형식을 매번 맞춰야 합니다. 작은 예제에서는 견딜 만하지만 프롬프트, 검색, 후처리, 메모리가 늘어나면 금세 관리가 어려워집니다.

LangChain은 이 조각들을 교체 가능한 부품으로 추상화한 프레임워크입니다. 임베딩 모델, 벡터 스토어, LLM, 프롬프트, 출력 파서가 각각 표준 인터페이스를 따르므로 부품을 갈아끼우듯 조합합니다. 직접 짠 코사인 검색을 `as_retriever()` 한 줄이 대신하고, 모델 교체는 클래스 이름 한 줄로 끝납니다.

LangChain이 제공하는 부품은 크게 모델 I/O, 검색(Retrieval), 메모리, 에이전트와 툴로 나뉘고, 이들을 LCEL로 엮습니다. 전체 구성을 그림으로 보면 다음과 같습니다.



LangChain 개요: 모델 I/O·Retrieval·메모리·에이전트/툴을 LCEL로 조립하는 프레임워크

> **핵심 포인트:** LangChain은 RAG의 각 단계를 표준 부품으로 만들어, "원리 구현"을 "부품 조립"으로 바꿔 줍니다.

2. ChatOpenAI: LLM 부품

LangChain에서 모델은 `ChatOpenAI` 같은 클래스 하나로 표현됩니다. 문자열을 그대로 넣어도 내부적으로 `HumanMessage`로 변환되어 동작하지만, 출력은 `AIMessage` 객체이므로 `.content`로 본문을 꺼냅니다.

```
# 2.langchain/1.llm_models/1.3_openai3_chatlibs.py
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage, AIMessage

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.9)

prompt = "What's a good company name that makes arcade games?"
result = llm.invoke(prompt)  # 문자열을 넣어도 내부적으로 HumanMessage 로 변환됨
print(result.content)
```

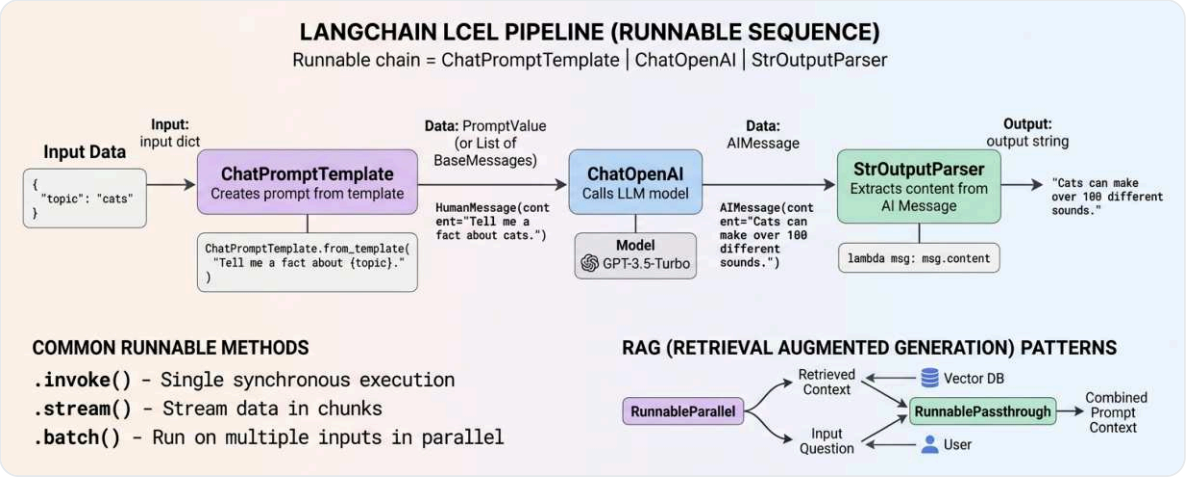
Chat 모델의 본래 입력은 메시지 리스트입니다. System / Human / AI 메시지를 교차 배치하면 멀티턴 대화의 맥락을 그대로 모델에 전달할 수 있습니다.

```
# 2.langchain/1.llm_models/1.3_openai3_chatlibs.py
messages = [
    SystemMessage(content="너는 매우 유쾌한 농담을 잘하는 AI야."),
    HumanMessage(content="재미있는 회사 이름 하나만 지어줘."),
    AIMessage(content="그럼... '퍼니컴퍼니' 어때?"),
    HumanMessage(content="하나만 더!")
]
result = llm.invoke(messages)
print(result.content)
```

`SystemMessage`는 역할과 규칙을, `HumanMessage`는 사용자 발화를, `AIMessage`는 이전 모델 답변을 나타냅니다. 마지막의 "하나만 더!"가 앞 맥락 덕분에 "재미있는 회사 이름을 하나 더"로 해석되는 이유가 여기에 있습니다.

3. LCEL: 파이프로 잇는 조립 언어

LCEL(LangChain Expression Language)은 부품을 파이프 연산자 `|`로 연결해 하나의 실행 가능한 체인을 만드는 표기법입니다. 유닉스 파이프처럼, 앞 단계의 출력이 다음 단계의 입력이 됩니다.



LCEL 파이프라인: ChatPromptTemplate | ChatOpenAI | StrOutputParser로 dict → PromptValue → AIMessage → str 타입이 흐른다

가장 기본이 되는 체인은 프롬프트, LLM, 출력 파서 세 부품을 잇는 형태입니다.

```python
# 2.langchain/4.chaining/1.basics/1.1_basic_chain.py
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)

prompt = ChatPromptTemplate.from_messages([
    ("system", "당신은 친절한 어시스턴트입니다."),
    ("user", "{question}"),
])

chain = prompt | llm | StrOutputParser()

result = chain.invoke({"question": "파이썬이 뭐야? 한 줄로 답해줘."})
print(result)
```

`chain.invoke({"question": ...})` 를 호출하면 입력 딕셔너리가 `prompt` 로 들어가 메시지 리스트로 채워지고, 그 메시지가 `llm` 에 전달되어 `AIMessage` 가 나오며, `StrOutputParser` 가 그것을 순수 문자열로 변환합니다. 세 부품의 입출력이 자동으로 맞물리므로 중간 변환 코드를 직접 쓸 필요가 없습니다.

3.1 ChatPromptTemplate의 역할

`ChatPromptTemplate.from_messages` 는 `("system", ...)` , `("user", ...)` 처럼 역할과 템플릿 문자열의 튜플 리스트를 받습니다. 중괄호 `{question}` 은 자리표시자(placeholder)로, `invoke` 에 넘긴 딕셔너리의 같은 이름 키 값으로 채워집니다. 덕분에 프롬프트의 골격은 고정하고 변하는 부분만 바깥에서 주입하는 구조가 됩니다.

> **핵심 포인트:** `prompt ¦ llm ¦ parser` 가 LCEL의 기본형입니다. 파이프는 "앞의 출력을 뒤의 입력으로" 자동 연결합니다.

4. RAG 체인: 검색을 파이프에 끼우기

RAG는 이 기본형 앞에 "검색" 단계를 끼운 것입니다. 질문이 들어오면 검색기가 관련 문서를 찾고, 그 문서를 프롬프트의 `{context}` 자리에 넣은 뒤 LLM이 답합니다. LCEL에서는 딕셔너리 형태로 두 갈래 입력을 동시에 만들어 이 흐름을 표현합니다.

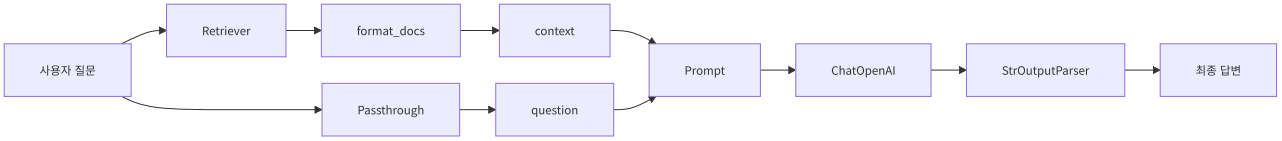
```
# 2.langchain/7.RAG/1.basics/1.4_rag_lcel_chain.py
prompt = ChatPromptTemplate.from_template(
    "다음 문서를 참고해서 질문에 답해주세요.\n\n"
    "문서:\n{context}\n\n"
    "질문: {question}"
)

def format_docs(docs):
    """검색된 Document 리스트 → 하나의 문자열"""
    return "\n\n".join(d.page_content for d in docs)

chain = (
    {
        "context": retriever ¦ format_docs,
        "question": RunnablePassthrough()
    }
    ¦ prompt
    ¦ llm
    ¦ StrOutputParser()
)

print(chain.invoke("NVMe 와 SATA 의 속도 차이?"))
```

체인 맨 앞의 딕셔너리에서 두 키가 병렬로 채워집니다. `context` 는 입력 질문이 `retriever` 로 흘러 관련 `Document` 리스트를 받고, 그것이 `format_docs` 로 하나의 문자열이 됩니다. `question` 은 `RunnablePassthrough()` 로 원본 질문을 그대로 통과시킵니다. 두 값이 모두 준비되면 `prompt` 가 `{context}` 와 `{question}` 을 채우고, 이후는 기본형과 동일합니다.

LCEL이 익숙하지 않다면 같은 흐름을 풀어 쓸 수도 있습니다. 검색을 먼저 호출해 문자열로 만든 뒤, 프롬프트 체인에 딕셔너리로 넘기면 됩니다.

```python
# 2.langchain/7.RAG/1.basics/1.4_rag_lcel_chain.py
docs = retriever.invoke(query)
context = format_docs(docs)

answer = (prompt | llm | StrOutputParser()).invoke({
    "context": context,
    "question": query,
})
print(answer)
```

두 방식은 결과가 같습니다. LCEL은 이 일련의 호출을 하나의 선언적 파이프로 묶어, 스트리밍·병렬·재시도 같은 기능을 체인 전체에 일괄 적용할 수 있게 해 줍니다.



> **핵심 포인트:** RAG 체인은 기본형 앞에 검색 단계를 딕셔너리로 끼운 형태입니다. `context` 는 검색 결과, `question` 은 통과값으로 채워져 프롬프트에 함께 들어갑니다.

프롬프트 엔지니어링

> 모델의 행동을 글로 설계하는 기술 — System Prompt, 템플릿, Few-shot으로 출력을 길들입니다.

1. 프롬프트: 모델을 다루는 인터페이스

LLM에게 작업을 시키는 유일한 통로는 프롬프트입니다. 같은 모델이라도 "감정 분석해줘"라고 막연히 시키면 자유로운 자연어로 길게 답하지만, 역할과 형식을 명확히 지정하면 파싱 가능한 일정한 출력을 냅니다. RAG에서 프롬프트는 더욱 중요합니다. 검색된 문서를 어떻게 참고할지, 문서에 없는 내용은 어떻게 처리할지, 출처를 어떻게 표기할지가 모두 프롬프트에 달려 있기 때문입니다.

LangChain은 프롬프트를 문자열이 아니라 재사용 가능한 템플릿 객체로 다룹니다. 골격은 고정하고 변하는 부분만 자리표시자로 비워 두면, 같은 프롬프트를 여러 입력에 일관되게 적용할 수 있습니다.

> **핵심 포인트:** 프롬프트는 모델의 행동을 정의하는 인터페이스입니다. 역할과 출력 형식을 명시할수록 결과가 안정됩니다.

2. System Prompt와 ChatPromptTemplate

Chat 모델은 메시지에 역할(role)을 부여합니다. `system`은 모델의 정체성과 규칙을, `human`(user)은 실제 요청을 담습니다. `ChatPromptTemplate.from_messages`로 역할별 템플릿을 한 번에 정의합니다.

```python
# 2.langchain/2.prompts/2.chat_templates/2.1_template_chat.py
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a naming consultant for new companies."),
    ("human", "What is a good name for a {company} that makes {product}?")
])

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.9)
parser = StrOutputParser()
```

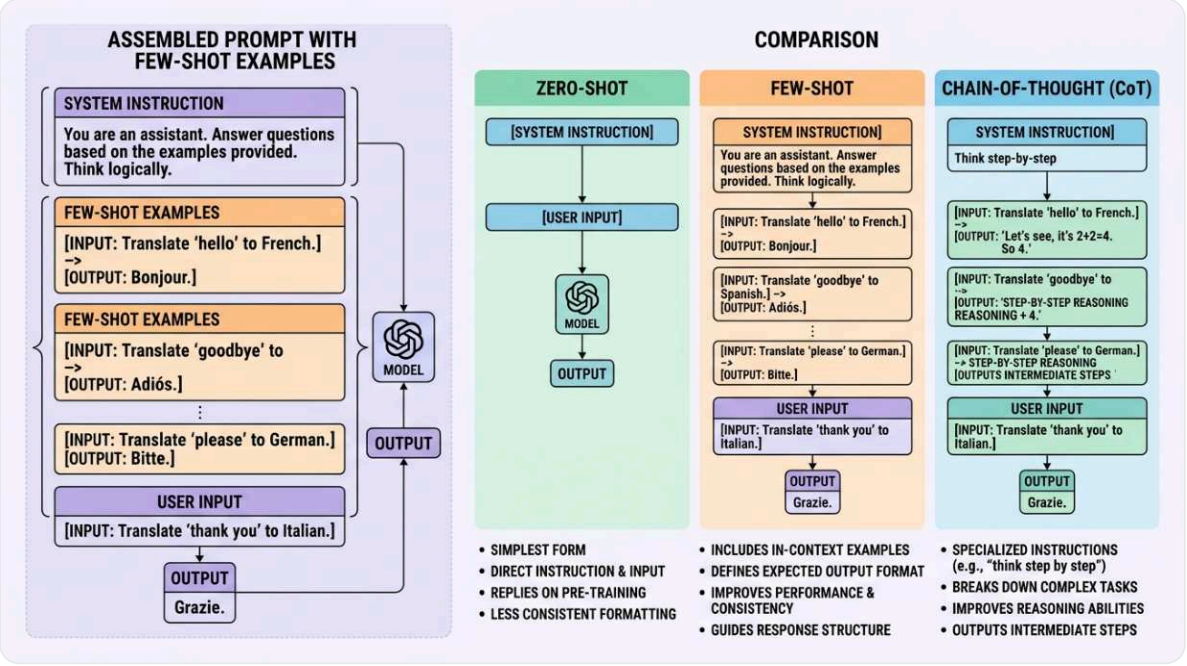
`{company}`와 `{product}`는 자리표시자입니다. 실행 시 입력 딕셔너리의 같은 이름 키로 채워집니다. `format_messages`로 실제 모델에 들어갈 메시지를 미리 확인할 수도 있습니다.

```
# 2.langchain/2.prompts/2.chat_templates/2.1_template_chat.py
inputs = {"company": "High Tech Startup", "product": "Web Game"}

# format_messages 는 ChatPromptTemplate 전용 메서드 (자리표시자를 실제 값으로 채움)
messages = prompt.format_messages(**inputs)

response = llm.invoke(messages)
cleaned_output = parser.invoke(response)  # AIMessage → 문자열
print(cleaned_output)
```

실무에서는 이 세 단계를 `chain = prompt ¦ llm ¦ parser`로 묶어 한 번에 실행하지만, 위처럼 단계별로 분리해 두면 어떤 메시지가 모델에 들어가는지 눈으로 확인하며 디버깅하기 좋습니다.

3. Few-shot: 예제로 형식을 고정하기

출력 형식을 글로 설명하기 까다로울 때는 예제 몇 개를 먼저 보여주는 Few-shot 프롬프팅이 효과적입니다. 모델은 예시의 패턴을 그대로 따라 답합니다. 감정 분석을 "감정: X / 점수: N"이라는 정해진 형식으로 강제하는 예를 보겠습니다.



프롬프트 구성: 시스템 지시 + Few-shot 예시 + 사용자 입력. 예시를 함께 주면(Few-shot) 출력 형식과 품질이 안정된다

```
# 2.langchain/2.prompts/3.fewshot/3.1_fewshot_chat.py
from langchain_core.prompts import PromptTemplate, FewShotPromptTemplate, ChatPromptTempla

examples = [
    {"sentence": "오늘 정말 최고의 하루였어!", "result": "감정: 긍정 / 점수: 9"},
    {"sentence": "이거 진짜 별로네요. 시간 낭비였어요.", "result": "감정: 부정 / 점수: 2"},
    {"sentence": "그냥 평범했어요. 특별히 좋지도 나쁘지도 않았네요.", "result": "감정: 중립 /
]

# 예제 한 건을 어떤 문자열로 만들지 정의
example_prompt = PromptTemplate(
    input_variables=["sentence", "result"],
    template="문장: {sentence}\n분석: {result}",
)
```

`FewShotPromptTemplate`은 예제들을 합쳐 하나의 큰 프롬프트 문자열로 만듭니다. `prefix`와 `suffix`로 "예시 구간"과 "진짜 질문"의 경계를 명확히 표시하는 것이 형식 추종 정확도를 높이는 요령입니다.

```
# 2.langchain/2.prompts/3.fewshot/3.1_fewshot_chat.py
fewshot_prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix="다음은 문장의 감정을 분석한 예시입니다. 같은 형식으로 새 문장을 분석하세요.\n\n===
    suffix="=== 예시 끝 ===\n\n=== 새로 분석할 문장 ===\n문장: {sentence}\n분석:",
    input_variables=["sentence"],
    example_separator="\n------------\n",
)

chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "당신은 한국어 감정 분석기입니다. 예시와 '정확히 같은 형식' 으로만 답하세요."),
    ("user",   "{fewshot_text}"),
])
```

같은 작업을 Few-shot 없이 시키면 모델은 "이 문장은 매우 긍정적이고 만족감이 느껴집니다..." 같은 자유 자연어로 답합니다. 파싱이 어렵고 점수 스케일도 매번 달라집니다. 예제 몇 개를 보여주는 것만으로 출력이 일정한 형식으로 고정됩니다.

> **핵심 포인트:** 출력 형식이 까다로울 때는 설명보다 예제가 빠릅니다. Few-shot은 분류 기준이나 회사 톤을 고정하는 데 효과적입니다.

4. Chat-native Few-shot

`FewShotPromptTemplate`은 예제 전부를 하나의 user 메시지 안에 박아 넣습니다. 반면 Chat 모델에는 각 예제를 "Human → AI" 메시지 쌍으로 분리해 진짜 대화 이력처럼 끼워 넣는 `FewShotChatMessagePromptTemplate`이 더 잘 맞습니다.

```
# 2.langchain/2.prompts/3.fewshot/3.2_fewshot_messages_chat.py
from langchain_core.prompts import ChatPromptTemplate, FewShotChatMessagePromptTemplate

# 예제 한 건을 어떤 '메시지 쌍' 으로 만들지 정의
example_prompt = ChatPromptTemplate.from_messages([
    ("human", "{sentence}"),
    ("ai",    "{result}"),
])

fewshot_messages = FewShotChatMessagePromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
)

# system + (few-shot 메시지들이 펼쳐짐) + 진짜 user 입력
final_prompt = ChatPromptTemplate.from_messages([
    ("system", "당신은 한국어 감정 분석기입니다. 사용자 문장에 대해 '감정: X / 점수: N' 형식으
    fewshot_messages,              # 예제 5쌍이 여기에 자동으로 펼쳐짐
    ("human", "{sentence}"),
])
```

두 방식은 모델에 전달되는 메시지 구조가 다릅니다. `FewShotPromptTemplate`은 한 사람이 길게 예시를 늘어놓고 마지막에 질문하는 모양이고, `FewShotChatMessagePromptTemplate`은 이미 다섯 번 대화한 뒤 여섯 번째 차례에 답하는 모양입니다. 후자가 Chat 모델이 학습한 형식에 가까워 형식 추종이 더 정확합니다.

구분	FewShotPromptTemplate	FewShotChatMessagePromptTemplate
메시지 구조	예제 전부가 하나의 user 메시지	예제마다 human/ai 메시지 쌍
모델 관점	길게 예시 늘어놓고 질문	여러 번 대화한 뒤 다음 차례
권장 모델	Legacy completion 모델	Chat 모델 (gpt-4o 등)

> **핵심 포인트:** Chat 모델에는 예제를 메시지 쌍으로 푸는 `FewShotChatMessagePromptTemplate`이 형식 추종에 유리합니다.

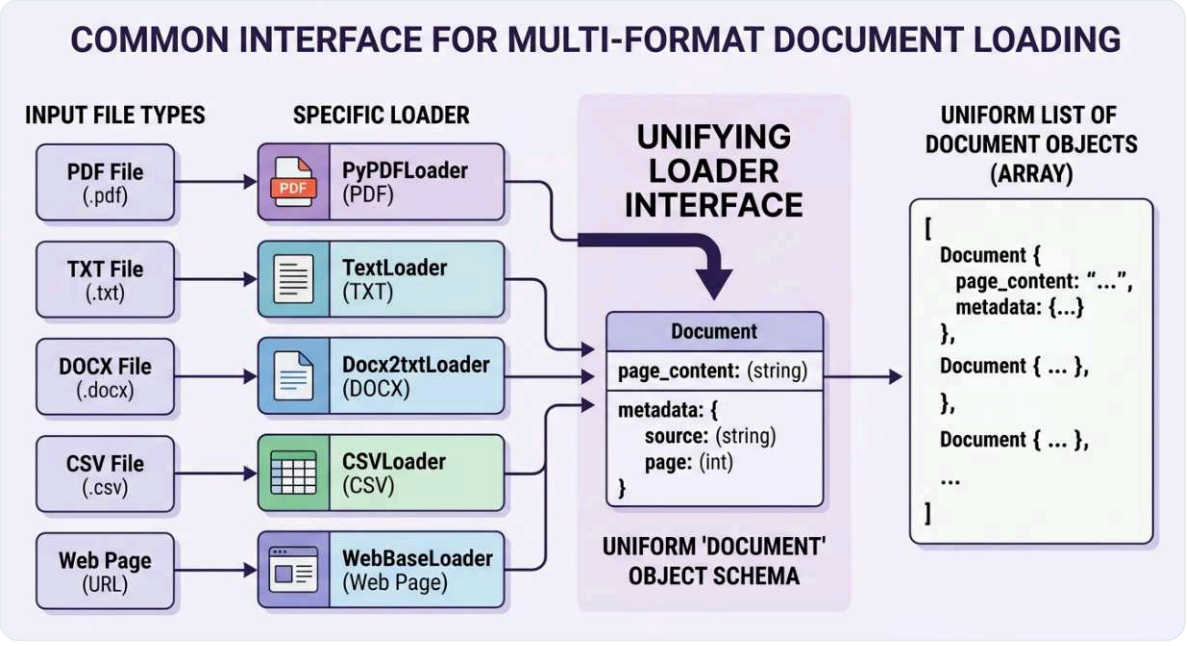
Document Loader

> 흩어진 원본 파일을 RAG가 다룰 수 있는 표준 단위, Document로 불러옵니다.

1. RAG 입력의 표준 단위, Document

RAG 파이프라인은 텍스트, PDF, 워드 등 형식이 제각각인 원본 파일에서 출발합니다. 이 다양한 입력을 하나의 표준 형태로 통일하지 않으면 이후 청킹·임베딩·검색 단계를 형식마다 따로 만들어야 합니다. LangChain은 이 문제를 `Document` 라는 단일 자료형으로 해결합니다.

`Document` 는 두 개의 필드로 이루어집니다. `page_content` 는 실제 텍스트 본문이고, `metadata` 는 출처 파일 경로, 페이지 번호 같은 부가 정보를 담는 딕셔너리입니다. Loader는 어떤 형식의 파일이든 읽어서 `Document` 리스트로 변환하는 부품입니다. 이후 모든 단계는 원본이 무엇이었는지 신경 쓰지 않고 `Document` 만 다룹니다.



Document Loader: PDF·TXT·DOCX·웹 등 다양한 포맷을 동일한 Document(page_content, metadata) 형태로 통일한다

> **핵심 포인트:** Loader는 형식이 다른 원본을 `Document(page_content, metadata)` 라는 공통 단위로 통일합니다. 이후 단계는 원본 형식을 몰라도 됩니다.

2. TextLoader: 텍스트 파일

가장 단순한 로더는 텍스트 파일을 읽는 `TextLoader` 입니다. 한글이 깨지지 않도록 인코딩을 명시하고 `load()` 를 호출하면 `Document` 리스트가 반환됩니다.

```python
# 2.langchain/7.RAG/2.loaders/2.1_text_loader.py
from langchain_community.document_loaders import TextLoader

loader = TextLoader("../DATA/nvme.txt", encoding="utf-8")
documents = loader.load()

print(f"불러온 Document 개수: {len(documents)}\n")

doc = documents[0]
print(f"page_content (앞 200자):\n{doc.page_content[:200]}...\n")
print(f"metadata: {doc.metadata}")
```

여기서 짚을 점은 **Loader 단계에서는 분할(청킹)을 하지 않는다**는 것입니다. `TextLoader` 는 파일 하나 전체를 `Document` 하나로 만듭니다. 14KB짜리 텍스트 파일을 읽으면 길이 1짜리 리스트에 파일 전체 내용이 통째로 들어갑니다. 잘게 쪼개는 작업은 다음 단계인 Text Splitter의 몫입니다.

`metadata` 의 `source` 키에는 파일 경로가 자동으로 채워집니다. 별도로 지정하지 않아도 "이 텍스트가 어디서 왔는지"가 기록되는 것입니다.

3. PyPDFLoader: PDF 파일

PDF는 페이지 단위로 구조가 나뉘므로, `PyPDFLoader` 는 **PDF 1페이지를 Document 1개**로 변환합니다. 10페이지 PDF를 읽으면 길이 10의 `Document` 리스트가 나옵니다.

```python
# 2.langchain/7.RAG/2.loaders/2.2_pdf_loader.py
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("../DATA/하계학술대회(CISC-S'24) CFP v2.pdf")
pages = loader.load()

print(f"PDF 페이지 수: {len(pages)}\n")

# 첫 번째 내용 있는 페이지 살펴보기
for p in pages:
    if p.page_content.strip():
        print(f"첫 내용 페이지 metadata: {p.metadata}")
        print(f"page_content (앞 200자):\n{p.page_content[:200]}...")
        break
```

PDF 로더의 `metadata`에는 `source`(파일 경로)뿐 아니라 `page`(페이지 번호)도 자동으로 부여됩니다. 덕분에 나중에 답변에 "이 내용은 3페이지에서 왔다"고 출처를 표기할 수 있습니다. `PyPDFLoader`를 쓰려면 `pip install pypdf`가 필요합니다.

PDF 로더 역시 청킹은 하지 않습니다. 한 페이지가 길면 그 텍스트가 통째로 한 `Document`에 들어가므로, 페이지 텍스트가 임베딩 토큰 한도를 넘을 수 있습니다. 따라서 PDF도 텍스트와 마찬가지로 이후 Splitter 단계에서 다시 잘게 쪼갭니다.

로더	분할 단위	자동 metadata
TextLoader	파일 1개 = Document 1개	source
PyPDFLoader	페이지 1개 = Document 1개	source, page

> **핵심 포인트:** TextLoader는 파일 단위, PyPDFLoader는 페이지 단위로 Document를 만듭니다. 어느 쪽도 청킹은 하지 않으므로 이후 Splitter가 필요합니다.

4. 메타데이터로 출처 추적하기

자동으로 부여되는 `source`, `page` 외에도, 필요에 따라 메타데이터를 직접 추가할 수 있습니다. 출처를 답변에 인용하거나, 특정 파일·날짜로 필터링 검색을 하려면 의미 있는 메타데이터가 필수입니다. 여러 파일을 로드하고 청킹한 뒤, 각 청크에 출처 파일명과 청크 번호를 부여하는 예를 보겠습니다.

```python
# 2.langchain/7.RAG/2.loaders/2.5_metadata.py
import os
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

FILES = ["../DATA/nvme.txt", "../DATA/ssd.txt"]
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)

all_chunks = []
for path in FILES:
    documents = TextLoader(path, encoding="utf-8").load()
    chunks = splitter.split_documents(documents)

    # 파일별로 source(파일명만) + chunk_id(파일 내부 1..N) 부여
    for i, chunk in enumerate(chunks, start=1):
        chunk.metadata["source"] = os.path.basename(path)
        chunk.metadata["chunk_id"] = i

    all_chunks.extend(chunks)
    print(f"{os.path.basename(path)}: {len(chunks)} 청크")
```

자동으로 들어가는 `source` 는 전체 경로(`../DATA/nvme.txt`)이지만, 여기서는 `os.path.basename` 으로 파일명만(`nvme.txt`) 남겨 더 깔끔하게 만들었습니다. 추가로 `chunk_id` 를 부여하면 같은 파일 안에서 몇 번째 조각인지도 추적됩니다.

이렇게 메타데이터를 부여해 두면 검색 결과에서 `doc.metadata["source"]` , `doc.metadata["chunk_id"]` 로 출처를 정확히 인용합니다. 답변의 신뢰도를 확보하는 RAG의 기본기이며, 뒤의 Retriever 메타데이터 필터링과 RAG 체인의 출처 표기로 이어집니다.

> **핵심 포인트:** 메타데이터는 출처 추적과 필터링 검색의 토대입니다. 자동 부여되는 source/page에 더해, source(파일명)·chunk_id 같은 키를 직접 부여해 두면 인용과 필터가 쉬워집니다.

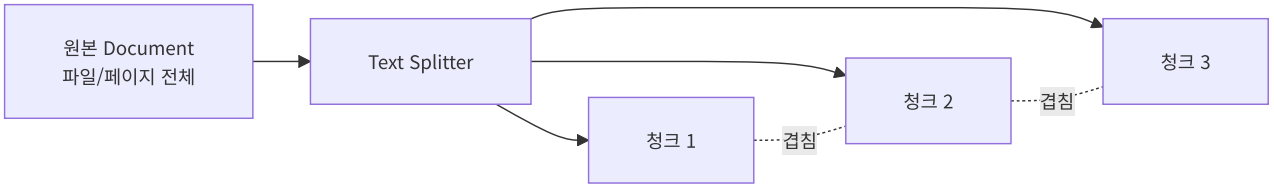
Text Splitter

> 긴 문서를 검색에 알맞은 조각으로 나누는 청킹 — 크기와 겹침이 검색 품질을 좌우합니다.

1. 왜 청킹이 필요한가

Loader가 만든 `Document`는 파일이나 페이지 전체를 담고 있어 임베딩에 그대로 쓰기에는 너무 큽니다. 임베딩 모델에는 입력 길이 제한이 있고, 한 청크에 여러 주제가 뒤섞이면 검색 정확도가 떨어집니다. 질문과 관련된 한 문단을 찾고 싶은데 책 한 권 전체가 하나의 벡터로 뭉뚱그려져 있으면, 의미가 희석되어 정밀한 매칭이 어렵습니다.

반대로 너무 잘게 쪼개면 맥락이 끊깁니다. 한 문장만 떼어 두면 앞뒤 맥락이 사라져, 검색은 되더라도 LLM이 답을 구성할 근거가 부족해집니다. 그래서 청킹은 검색 정밀도와 맥락 보존 사이의 균형을 찾는 작업입니다. 보통 200~1000 토큰 크기에 50~200 정도의 겹침(overlap)을 두는 것이 출발점입니다.



> **핵심 포인트:** 청킹은 검색 정밀도와 맥락 보존의 균형입니다. 너무 크면 의미가 희석되고, 너무 작으면 맥락이 끊깁니다.

2. 두 가지 Splitter

LangChain의 대표적인 두 분할기는 동작 방식이 다릅니다. `CharacterTextSplitter`는 정해진 구분자 하나로만 나누고, `RecursiveCharacterTextSplitter`는 여러 구분자를 순서대로 시도합니다. 같은 텍스트에 둘을 적용해 비교해 보겠습니다.

```python
# 2.langchain/7.RAG/2.loaders/2.3_chunking.py
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import (
    CharacterTextSplitter,
    RecursiveCharacterTextSplitter,
)

documents = TextLoader("../DATA/nvme.txt", encoding="utf-8").load()
original = documents[0].page_content
print(f"원본 글자수: {len(original)}\n")

# 1) CharacterTextSplitter – 단순히 정해진 separator 로 나누기
char_splitter = CharacterTextSplitter(
    separator="\n\n",      # 목표는 빈 줄 기준
    chunk_size=500,        # 최대 500글자로 합침 (넘치지 않게)
    chunk_overlap=100,     # 이전과 겹치게
)
chunks_char = char_splitter.split_documents(documents)
print(f"[CharacterTextSplitter] {len(chunks_char)} 청크")
```

`CharacterTextSplitter`는 `separator`로 지정한 `\n\n`(빈 줄)을 기준으로만 자릅니다. 단순하고 예측 가능하지만, 빈 줄이 드물게 등장하는 문서에서는 청크 크기가 들쭉날쭉해질 수 있습니다.

`RecursiveCharacterTextSplitter`는 여러 구분자를 큰 단위부터 작은 단위 순(`\n\n` → `\n` → 공백 → 글자)으로 시도합니다. 가능한 한 의미 단위를 보존하면서 `chunk_size`를 맞추므로, 일반 텍스트에 널리 권장됩니다.

```python
# 2.langchain/7.RAG/2.loaders/2.3_chunking.py
# 2) RecursiveCharacterTextSplitter – 여러 separator 를 순서대로 시도
# (\n\n → \n → 공백 → 글자 순). 가장 권장되는 일반용 splitter.
recur_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=100,
)
chunks_recur = recur_splitter.split_documents(documents)
print(f"[RecursiveCharacterTextSplitter] {len(chunks_recur)} 청크")
```

항목	CharacterTextSplitter	RecursiveCharacterTextSplitter
분할 기준	구분자 하나	여러 구분자를 순서대로
의미 단위 보존	약함	강함 (문단→줄→단어 순)
권장 용도	단순 데모/구조	일반 텍스트 (기본 선택)

> **핵심 포인트:** 일반 텍스트에는 `RecursiveCharacterTextSplitter`를 기본으로 씁니다. 여러 구분자를 순서대로 시도해 의미 단위를 최대한 보존합니다.

3. 토큰 기반 분할

`chunk_size`는 기본적으로 글자 수 기준이지만, OpenAI 모델은 토큰 단위로 과금됩니다. 글자 수와 토큰 수는 한국어·영어에서 비율이 다르므로, 실제 청구되는 토큰과 청크 크기를 일치시키려면 토큰 기반 분할을 씁니다.

```
# 2.langchain/7.RAG/2.loaders/2.3_chunking.py
# 3) Token 기반 – tiktoken 으로 실제 토큰 수 기준 (OpenAI 모델 호환)
#   ※ pip install tiktoken
token_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=200,        # 토큰 단위!
    chunk_overlap=50,
)
chunks_token = token_splitter.split_documents(documents)
print(f"[tiktoken 기반] {len(chunks_token)} 청크")
```

`from_tiktoken_encoder`로 만든 분할기는 `chunk_size=200`을 200토큰으로 해석합니다. 비용 예측과 토큰 한도 관리가 중요한 운영 환경에서 유용합니다. 선택 기준을 정리하면, 일반 텍스트는 `RecursiveCharacterTextSplitter`, OpenAI 비용 관리가 중요하면 `from_tiktoken_encoder`, 단순한 데모는 `CharacterTextSplitter`입니다.

4. PDF 청킹과 overlap의 의미

PDF도 청킹 방식은 같습니다. `PyPDFLoader`가 페이지별 `Document`를 만들면, 그 리스트를 `split_documents`에 그대로 넘기면 됩니다. 분할기는 입력이 텍스트였는지 PDF였는지 구분하지 않고 `Document` 리스트만 처리합니다.

```python
# 2.langchain/7.RAG/2.loaders/2.4_chunking_pdf.py
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

loader = PyPDFLoader("../DATA/하계학술대회(CISC-S'24) CFP v2.pdf")
pages = loader.load()
print(f"원본 페이지 수: {len(pages)}")

splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=100,
)
chunks = splitter.split_documents(pages)
print(f"청킹 후 Document 수: {len(chunks)}\n")

first = chunks[0]
print("metadata:")
print(first.metadata)
```

`split_documents`는 원본 `Document`의 메타데이터(여기서는 `source`와 `page`)를 각 청크에 그대로 물려줍니다. 따라서 청킹 후에도 "이 조각이 몇 페이지에서 왔는지"가 보존됩니다.

`chunk_overlap`은 인접한 청크가 일부 내용을 겹쳐 갖게 하는 설정입니다. 분할 경계에서 한 문장이 두 청크로 잘려 맥락이 끊기는 문제를 완화합니다. 예를 들어 `chunk_size=500, chunk_overlap=100`이면 각 청크의 끝 100글자가 다음 청크의 앞부분에 다시 등장합니다. 경계에 걸친 정보도 최소한 한쪽 청크에서는 온전히 검색될 수 있도록 보장하는 안전장치입니다.

> **핵심 포인트:** `split_documents`는 메타데이터를 청크에 그대로 물려줍니다. `chunk_overlap`은 분할 경계에서 끊기는 맥락을 완화하는 안전장치입니다.

PART 04

벡터 검색과 RAG 체인

1. 벡터 데이터베이스 이해
2. ChromaDB 구축
3. Retriever 구성
4. RAG Chain 구축: 환각 방지와 출처

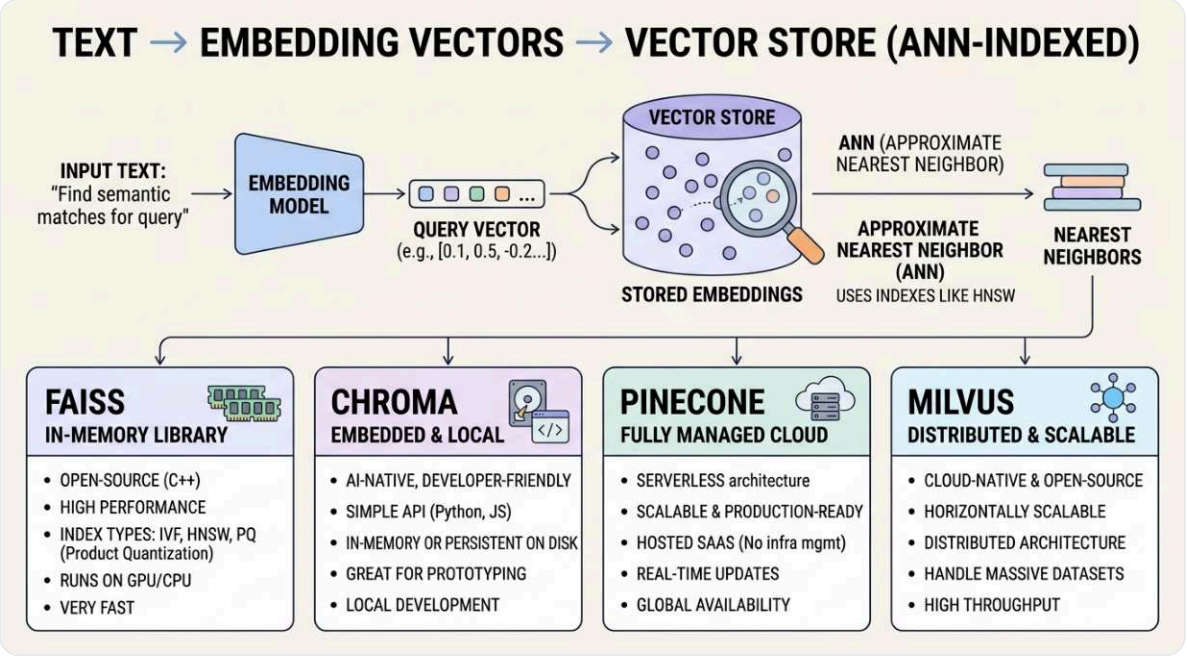
벡터 데이터베이스 이해

> 의미를 좌표로 바꾼 임베딩을 저장하고, 가장 가까운 이웃을 빠르게 찾는 전용 저장소입니다.

1. 벡터 데이터베이스란

RAG에서 각 청크는 임베딩을 거쳐 수백~수천 차원의 벡터, 즉 의미를 나타내는 좌표가 됩니다. 질문도 같은 방식으로 벡터가 되고, 검색이란 질문 벡터와 가장 가까운 문서 벡터를 찾는 일입니다. 직접 구현한다면 모든 문서 벡터와 코사인 유사도를 일일이 계산하고 정렬해야 합니다. 문서가 수천 개면 견딜 만하지만, 수백만 개가 되면 매 질문마다 전수 계산을 하는 것은 비현실적입니다.

벡터 데이터베이스는 이 문제를 전담하는 저장소입니다. 벡터와 원본 텍스트, 메타데이터를 함께 보관하고, "가장 가까운 N개"를 효율적으로 찾는 색인(index)을 제공합니다. 관계형 DB가 행과 인덱스로 정확한 일치를 빠르게 찾듯, 벡터 DB는 근사 색인으로 의미적 근접을 빠르게 찾습니다.



벡터 DB 비교: FAISS·Chroma·Pinecone·Milvus. 임베딩을 저장하고 근사 최근접(ANN) 검색으로 가장 가까운 벡터를 찾는다

> **핵심 포인트:** 벡터 DB는 임베딩과 텍스트·메타데이터를 함께 저장하고, "가장 가까운 이웃"을 빠르게 찾는 색인을 제공하는 전용 저장소입니다.

2. ANN: 근사 최근접 이웃

핵심 기술은 ANN(Approximate Nearest Neighbor), 근사 최근접 이웃 검색입니다. 모든 벡터와의 거리를 정확히 계산하는 완전 탐색(brute-force)은 데이터가 커질수록 느려집니다. ANN은 약간의 정확도를 양보하는 대신 검색 공간을 영리하게 좁혀, 수백만 벡터에서도 밀리초 단위로 "거의 가장 가까운" 이웃을 찾아냅니다.

대표적인 ANN 색인으로 HNSW(계층적 그래프 탐색), IVF(클러스터로 나눈 뒤 일부만 탐색), PQ(벡터를 압축해 메모리 절약) 등이 있습니다. 사용자는 보통 이 내부를 직접 다루지 않고, 벡터 DB가 제공하는 기본 색인을 그대로 씁니다. 다만 정확도와 속도는 맞교환 관계이며, 대규모에서는 근사 검색이 필수라는 점은 알아 둡니다.

> **핵심 포인트:** ANN은 약간의 정확도를 내주고 검색 공간을 좁혀 대규모에서도 빠른 검색을 가능하게 합니다. 정확도와 속도는 맞교환 관계입니다.

3. 주요 벡터 DB 비교

벡터 DB는 종류가 많고 성격이 다릅니다. 학습·MVP·일반 실무 RAG에서는 설치가 간단하고 메타데이터 관리가 풍부한 도구가 유리하고, 수억 벡터 규모나 매니지드 운영이 필요하면 다른 선택지가 등장합니다.

항목	Chroma	FAISS	Pinecone	Milvus
형태	임베디드(로컬)	라이브러리	매니지드(클라우드)	자체 호스팅 서버
영속화	persist_directory 한 줄	save/load 수동	자동(관리형)	클러스터 저장
메타데이터 필터	네이티브, 풍부	약함, 후처리	지원	지원
강점	학습·MVP·웹앱	대규모·GPU 가속	운영 부담 최소	대규모 자체 운영
규모	수만~수십만	수백만~수억	대규모	대규모

Chroma는 설치가 `pip` 한 번으로 끝나고 서버가 필요 없으며, 출처·페이지·날짜 같은 메타데이터 필터링을 네이티브로 지원해 학습과 일반 실무 RAG에 적합합니다. FAISS는 라이브러리로서 수억 벡터와 GPU 가속에 강합니다. Pinecone은 인프라를 직접 운영하지 않아도 되는 매니지드 서비스이고, Milvus는 대규모 자체 호스팅에 적합합니다. 이 과정에서는 Chroma를 기본으로 사용합니다.

4. FAISS로 체감하는 Chroma의 강점

같은 RAG 흐름을 FAISS로 짜 보면 Chroma가 어느 부분에서 손이 덜 가는지 분명해집니다. 인덱스를 만드는 단계까지는 거의 동일합니다.

```
# 2.langchain/7.RAG/3.vectorstore/3.10_faiss_compare.py
from langchain_community.vectorstores import FAISS

def build_index():
    docs = TextLoader("../DATA/nvme.txt", encoding="utf-8").load() \
         + TextLoader("../DATA/ssd.txt",  encoding="utf-8").load()
    chunks = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100).split_docum
    for c in chunks:
        c.metadata["source"] = os.path.basename(c.metadata.get("source", "?"))

    store = FAISS.from_documents(chunks, embeddings)

    # ★ 차이 1: 영속화는 명시적 호출. Chroma 는 persist_directory 만 주면 자동.
    store.save_local(FAISS_DIR)
    return store
```

영속화부터 차이가 납니다. Chroma는 `persist_directory` 만 주면 자동 저장되지만, FAISS는 `save_local` 을 직접 호출해야 하고, 로드할 때는 pickle 역직렬화이므로 위험 허용 옵션을 명시해야 합니다.

```
# 2.langchain/7.RAG/3.vectorstore/3.10_faiss_compare.py
def load_index():
    # ★ 차이 2: pickle 역직렬화이므로 allow_dangerous_deserialization 명시 필요.
    store = FAISS.load_local(
        FAISS_DIR,
        embeddings,
        allow_dangerous_deserialization=True,
    )
    return store
```

가장 큰 차이는 메타데이터 필터링입니다. Chroma는 `similarity_search(q, filter={"source": "nvme.txt"})` 처럼 깔끔한 네이티브 쿼리를 제공하지만, FAISS는 단순 단일 키 필터까지는 되더라도 AND/OR나 숫자 비교 같은 본격 쿼리는 부담스러워, 복잡한 조건은 보통 직접 후처리해야 합니다.

```python
# 2.langchain/7.RAG/3.vectorstore/3.10_faiss_compare.py
# (A) FAISS 가 지원하는 가벼운 filter
docs_a = store.similarity_search("PCIe 인터페이스", k=3, filter={"source": "nvme.txt"})

# (B) 복잡한 조건은 직접 후처리
all_docs = store.similarity_search("PCIe", k=10)
docs_b = [d for d in all_docs if d.metadata.get("source") in {"nvme.txt", "ssd.txt"}][:3]
```

RAG의 핵심은 메타데이터로 출처를 추적하고 삭제·필터링하는 일인데, 그 부분이 Chroma에서 훨씬 간편합니다. FAISS는 대규모 벡터 검색이나 GPU 가속이 핵심인 환경에서 강점을 보입니다.

> **핵심 포인트:** 학습과 일반 실무 RAG에서는 영속화·필터링·삭제가 간편한 Chroma가 유리하고, FAISS는 대규모·GPU가 필요한 경우에 강점이 있습니다.

ChromaDB 구축

> 임베딩을 디스크에 영속화하고, 컬렉션 단위로 저장·검사·검색하는 실전 워크플로를 익힙니다.

1. 영속화가 필요한 이유

앞서 본 `InMemoryVectorStore`는 메모리 위에서만 동작합니다. 프로세스가 끝나면 임베딩이 사라지므로, 스크립트를 다시 실행할 때마다 모든 청크를 임베딩 API로 재계산해야 합니다. 청크가 많아질수록 시간과 비용이 누적됩니다.

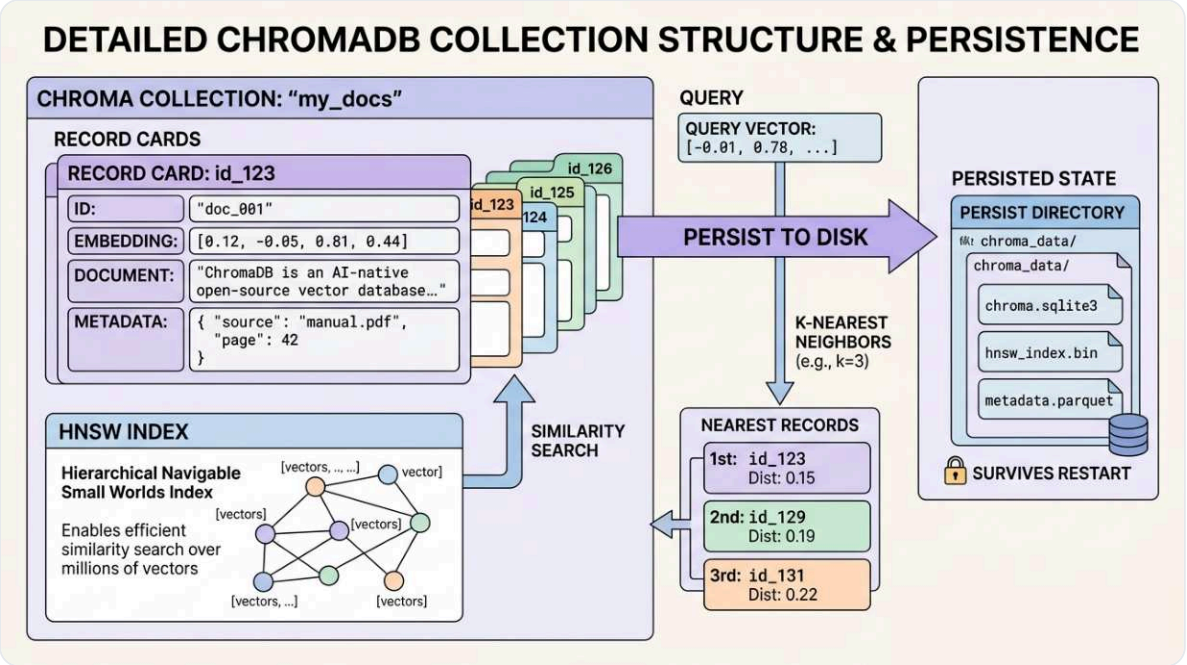
ChromaDB는 임베딩을 `persist_directory`로 지정한 폴더에 디스크 저장합니다. 한 번 만들어 두면 재실행 시 즉시 로드되어 임베딩 API를 다시 호출하지 않습니다. 같은 데이터로 반복 실험하거나, 웹 서버가 재시작되어도 인덱스를 유지해야 하는 운영 환경에서 중요한 차이입니다.

구분	InMemoryVectorStore	Chroma
저장 위치	메모리	persist_directory (디스크)
재실행 시	매번 재임베딩, API 비용 발생	즉시 로드, 추가 비용 0
용도	빠른 데모	반복 실험·운영

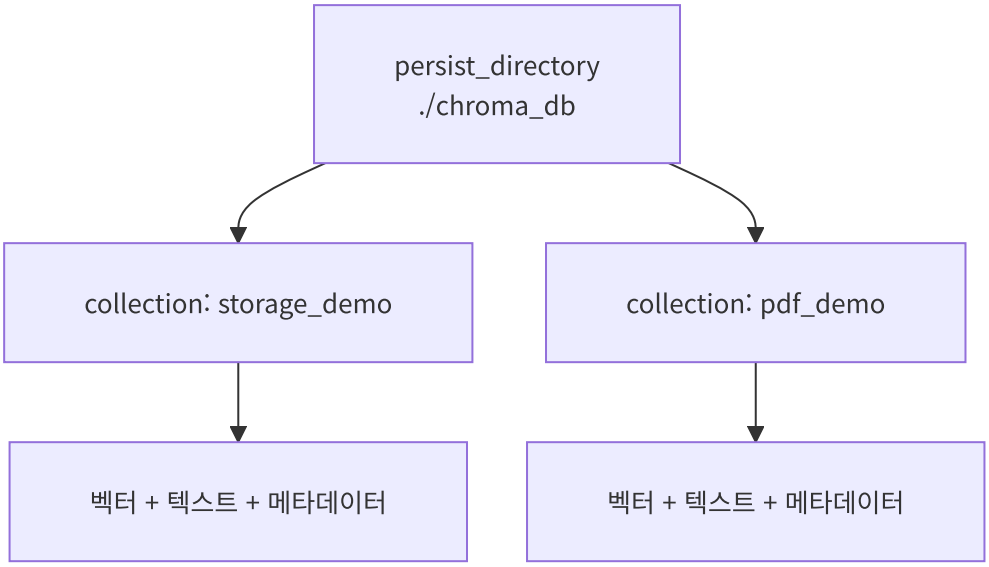
> **핵심 포인트:** Chroma는 임베딩을 디스크에 영속화해, 재실행 시 재임베딩 없이 즉시 로드합니다. InMemory와의 결정적 차이입니다.

2. 저장 구조: 컬렉션과 영속 디렉터리

Chroma는 벡터를 컬렉션(collection) 단위로 묶어 관리합니다. 하나의 `persist_directory` 안에 여러 컬렉션이 공존할 수 있고, 각 컬렉션은 `collection_name`으로 구분됩니다. 같은 디렉터리에 텍스트 문서용 컬렉션과 PDF용 컬렉션을 따로 두는 식입니다.



ChromaDB 컬렉션 구조: 각 항목이 id·임베딩 벡터·원문(document)·메타데이터로 저장되고, persist 디렉토리에 영속화된다



각 컬렉션 안에는 청크의 임베딩 벡터, 원본 텍스트, 메타데이터가 함께 저장됩니다. 내부적으로 SQLite와 Parquet 파일로 디스크에 기록되므로, 디렉터리만 백업하면 인덱스 전체를 이전할 수 있습니다.

3. 생성과 로드

핵심 패턴은 "디스크에 컬렉션이 있으면 로드, 없으면 새로 생성"입니다. 문서를 로드·청킹한 뒤 `Chroma.from_documents`로 저장하는 함수와, 기존 DB를 불러오는 함수를 나눠 둡니다.

```python
# 2.langchain/7.RAG/3.vectorstore/3.1_persist.py
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma

PERSIST_DIR     = "./chroma_db"
COLLECTION_NAME = "storage_demo"
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

def build_store():
    """문서 로드 → 청킹 → 임베딩 → Chroma 에 저장"""
    docs   = TextLoader("../DATA/nvme.txt", encoding="utf-8").load()
    chunks = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100).split_docum

    store = Chroma.from_documents(
        chunks, embeddings,
        collection_name=COLLECTION_NAME,
        persist_directory=PERSIST_DIR,
    )
    print(f"새 DB 생성 — {len(chunks)} 청크 임베딩 저장")
    return store
```

`from_documents`는 청크 리스트와 임베딩 함수를 받아, 각 청크를 임베딩하고 컬렉션에 저장합니다. 반면 이미 만들어진 컬렉션을 다시 쓸 때는 `Chroma(...)` 생성자로 로드만 합니다. 이때는 임베딩 API 호출이 일어나지 않습니다.

```python
# 2.langchain/7.RAG/3.vectorstore/3.1_persist.py
def load_store():
    """디스크에서 기존 DB 로드 (임베딩 재계산 없음)"""
    store = Chroma(
        collection_name=COLLECTION_NAME,
        embedding_function=embeddings,
        persist_directory=PERSIST_DIR,
    )
    print(f"기존 DB 로드 — {store._collection.count()} 청크 있음")
    return store

# 디스크에 컬렉션이 이미 있으면 로드, 없으면 새로 만들기
if os.path.exists(PERSIST_DIR) and os.listdir(PERSIST_DIR):
    store = load_store()
else:
    store = build_store()
```

이 스크립트를 두 번 연속 실행하면 첫 실행은 "새 DB 생성", 두 번째는 "기존 DB 로드"가 출력됩니다. 두 번째 실행에서는 임베딩 API가 전혀 호출되지 않는 것이 핵심입니다.

> **핵심 포인트:** `from_documents` 는 임베딩 후 저장, `Chroma(...)` 생성자는 재임베딩 없는 로드입니다. "있으면 로드, 없으면 생성" 패턴으로 비용을 아낍니다.

4. PDF도 동일한 흐름

PDF를 저장하는 코드는 텍스트와 거의 같습니다. 로더만 `TextLoader`에서 `PyPDFLoader`로 바뀌고, 컬렉션 이름을 따로 둔다는 점만 다릅니다. PDF는 "1페이지 = Document 1개"로 먼저 쪼개진 뒤 청킹됩니다.

```
# 2.langchain/7.RAG/3.vectorstore/3.2_persist_pdf.py
from langchain_community.document_loaders import PyPDFLoader

PDF_PATH        = "../DATA/하계학술대회(CISC-S'24) CFP v2.pdf"
COLLECTION_NAME = "pdf_demo"

def build_store():
    docs   = PyPDFLoader(PDF_PATH).load()          # ★ 페이지별 Document 리스트
    chunks = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100).split_docum
    store = Chroma.from_documents(
        chunks, embeddings,
        collection_name=COLLECTION_NAME,
        persist_directory=PERSIST_DIR,
    )
    return store

# 검색 결과의 PDF 청크에는 page 메타데이터가 붙어있음
results = store.similarity_search("논문 제출 일정?", k=2)
for d in results:
    page = d.metadata.get("page", "?")
    print(f"  [p.{page}] {d.page_content[:60]}...")
```

PDF 청크에는 `page` 메타데이터가 보존되므로, 검색 결과에서 몇 페이지에서 왔는지 바로 표시할 수 있습니다.

5. 컬렉션 검사하기

저장이 제대로 되었는지, 안에 무엇이 들어있는지 들여다보는 것은 디버깅의 기본입니다. `_collection`의 `count`로 문서 개수를, `get`으로 검색 없이 일부 문서를 직접 꺼낼 수 있습니다.

```python
# 2.langchain/7.RAG/3.vectorstore/3.3_inspect.py
store = Chroma(
    collection_name=COLLECTION_NAME,
    embedding_function=OpenAIEmbeddings(model="text-embedding-3-small"),
    persist_directory=PERSIST_DIR,
)

# 1) 전체 문서 개수
count = store._collection.count()
print(f"컬렉션 '{COLLECTION_NAME}' 의 문서 수: {count}\n")

# 2) 일부 문서 샘플 – get(limit=N) 으로 그냥 가져오기 (검색 아님)
results = store._collection.get(include=["documents", "metadatas"], limit=3)
for i, (doc_id, content, meta) in enumerate(
    zip(results["ids"], results["documents"], results["metadatas"]), start=1):
    print(f"[{i}] id={doc_id[:8]}...")
    print(f"    내용: {content[:80]}...")
    print(f"    메타: {meta}\n")
```

`get` 은 유사도 검색이 아니라 단순 조회입니다. 저장된 청크의 ID, 내용, 메타데이터를 그대로 확인할 수 있어 "내가 의도한 청크와 메타데이터가 실제로 들어갔는지" 검증할 때 유용합니다.

> **핵심 포인트:** `count` 로 개수를, `get` 으로 내용을 직접 들여다봅니다. 검색이 이상할 때는 먼저 컬렉션에 무엇이 저장됐는지부터 확인합니다.

Retriever 구성

> 벡터 스토어를 검색 부품으로 감싸고, 검색 모드와 메타데이터 필터로 결과를 조율합니다.

1. Retriever란

벡터 스토어는 검색 메서드를 직접 노출하지만, RAG 체인에 끼우려면 표준화된 검색 인터페이스가 필요합니다. `as_retriever()`는 벡터 스토어를 Retriever라는 표준 부품으로 감쌉니다. Retriever는 질문 문자열을 받아 관련 `Document` 리스트를 반환하는 한 가지 역할만 맡으며, LCEL 파이프에 그대로 연결됩니다.

Retriever를 만들 때 `search_kwargs`로 검색 동작을 설정합니다. 가장 기본은 `k`로 반환할 문서 개수를 지정하는 것입니다. `k`가 너무 작으면 답에 필요한 근거가 빠지고, 너무 크면 관련 없는 청크가 섞여 프롬프트가 길어지고 답변 품질이 떨어집니다.

```python
# 2.langchain/7.RAG/4.rag_chain/4.1_standard_chain.py
retriever = store.as_retriever(search_kwargs={"k": 3})
```

> **핵심 포인트:** Retriever는 벡터 스토어를 "질문 → 관련 문서" 표준 부품으로 감싼 것입니다. `search_kwargs={"k": N}`으로 반환 개수를 조절합니다.

2. 검색 모드: similarity와 MMR

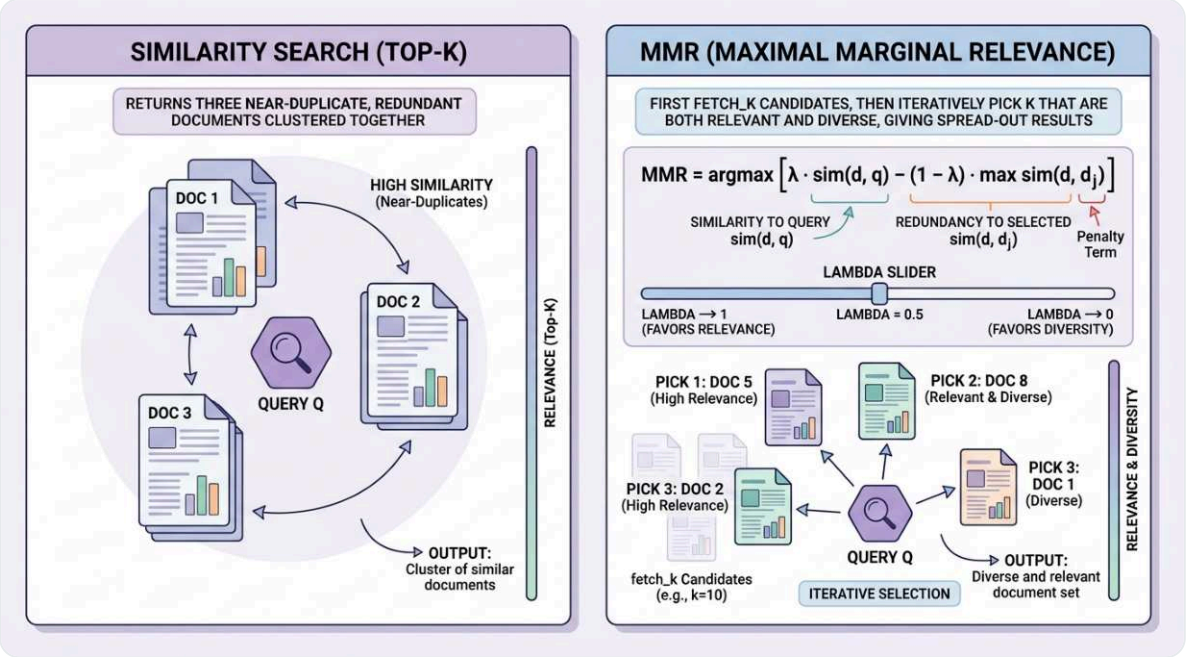
같은 질문이라도 검색 방식에 따라 결과 구성이 달라집니다. 벡터 스토어는 세 가지 모드를 제공합니다. `similarity_search`는 가장 가까운 N개를 반환하는 기본 방식이고, `similarity_search_with_score`는 거리 점수를 함께 돌려주며, MMR은 관련성에 다양성을 더합니다.

```python
# 2.langchain/7.RAG/3.vectorstore/3.4_search_modes.py
query = "NVMe 의 속도와 인터페이스"

# 1) similarity_search – 가장 단순 / 기본
for d in store.similarity_search(query, k=3):
    print(f"  → {d.page_content[:80]}...")

# 2) similarity_search_with_score – 점수 포함 (Chroma 는 거리, 작을수록 유사)
for d, score in store.similarity_search_with_score(query, k=3):
    sim_pct = round((1 - score) * 100, 1)
    print(f"  거리 {score:.3f} (유사도 ≈ {sim_pct}%)  {d.page_content[:60]}...")
```

점수가 함께 나오면 임계값으로 필터링할 수 있습니다. Chroma의 점수는 "거리"이므로 작을수록 가깝습니다. 일정 거리 이상으로 멀면 무관한 결과로 보고 버리는 식의 후처리가 가능합니다.

기본 유사도 검색의 약점은 "쏠림"입니다. 가장 가까운 N개를 그대로 뽑으면, 거의 같은 내용을 담은 청크들이 상위를 모두 차지해 정작 다양한 관점이 빠질 수 있습니다. MMR(Maximal Marginal Relevance)은 첫 결과는 가장 관련성 높은 것을 고르되, 그 다음부터는 이미 뽑은 결과와 다른 것을 우선해 중복을 피합니다.

```python
# 2.langchain/7.RAG/3.vectorstore/3.4_search_modes.py
# 3) MMR – 다양성 보장
# 같은 의미의 비슷한 청크들이 top-k 를 다 차지하지 않게,
# 첫 결과는 가장 관련성 높은 것, 그 다음부터는 이전 결과들과 다른 것 우선.
for d in store.max_marginal_relevance_search(query, k=3, fetch_k=10, lambda_mult=0.5):
    print(f"  → {d.page_content[:80]}...")
```



MMR 검색: 유사도만 보면 비슷한 청크가 중복되지만, MMR은 관련성과 다양성을 함께 고려해 중복을 줄인다

MMR의 `fetch_k` 는 먼저 가져올 후보 개수, `k` 는 최종 반환 개수입니다. `fetch_k=10` 으로 넓게 후보를 뽑은 뒤 그 중 다양성을 고려해 `k=3` 을 선택합니다. `lambda_mult` 는 관련성과 다양성의 가중치로, 1에 가까우면 관련성 위주, 0에 가까우면 다양성 위주입니다. 0.5는 균형점입니다.

모드	특징	적합한 경우
similarity_search	가장 가까운 N개	일반적인 기본 검색
with_score	거리 점수 포함	임계값 필터링이 필요할 때
MMR	관련성 + 다양성	긴 문서, 중복 청크 회피

> **핵심 포인트:** 기본은 유사도 검색, 중복 청크가 상위를 독점할 때는 MMR로 다양성을 확보합니다. `fetch_k` 로 후보를 넓게 잡고 `k` 로 최종 선택합니다.

3. 메타데이터 필터링

벡터 유사도만으로는 "최신 문서만", "특정 출처만" 같은 정형 조건을 걸 수 없습니다. 의미 유사도(벡터)에 정형 조건(메타데이터)을 함께 거는 것이 실무 RAG의 기본기입니다. Chroma는 `filter` 인자에 `where` 연산자를 받아 후보를 먼저 좁힙니다. 문서마다 풍부한 메타데이터를 붙여 저장하는 것이 출발점입니다.

```
# 2.langchain/7.RAG/3.vectorstore/3.11_metadata_filter.py
from langchain_core.documents import Document

# page_content(벡터화 대상) + 풍부한 metadata(필터 대상)
docs = [
    Document(page_content="NVMe SSD 는 PCIe 로 약 7GB/s 의 시퀀셜 속도를 낸다.",
             metadata={"category": "nvme", "year": 2022, "tier": "high"}),
    Document(page_content="SATA SSD 는 약 0.5GB/s 로 NVMe 보다 느리지만 호환성이 좋다.",
             metadata={"category": "sata", "year": 2018, "tier": "mid"}),
    Document(page_content="HDD 는 회전 디스크 기반이라 IO 가 느린 편이다.",
             metadata={"category": "hdd", "year": 2015, "tier": "low"}),
    Document(page_content="최신 PCIe 5.0 NVMe 는 약 14GB/s 까지 도달한다.",
             metadata={"category": "nvme", "year": 2024, "tier": "high"}),
]
store = Chroma.from_documents(docs, embeddings, collection_name="meta_filter_demo")
```

Chroma 메타데이터 값은 문자열·정수·실수·불리언만 가능합니다(리스트·딕셔너리 불가). 저장 후에는 `similarity_search(query, filter=where)` 로 같은 질문에 조건만 바꿔 가며 후보 집합을 좁힙니다.

```python
# 2.langchain/7.RAG/3.vectorstore/3.11_metadata_filter.py
query = "저장장치 속도"

def show(title, where=None):
    for d in store.similarity_search(query, k=4, filter=where):
        m = d.metadata
        print(f"  [{m['category']:4s} {m['year']} {m['tier']:4s}] {d.page_content[:38]}...

# 2) 동등 — category 가 nvme 인 것만
show("category == 'nvme'", {"category": "nvme"})

# 3) 숫자 비교 — 2020년 이후 문서만
show("year >= 2020", {"year": {"$gte": 2020}})

# 5) 복합 AND — nvme 이면서 2023년 이후
show("nvme AND year >= 2023", {"$and": [{"category": "nvme"}, {"year": {"$gte": 2023}}]})
```

`where` 연산자는 동등(`{"category": "nvme"}`), 숫자 비교(`$gt` / `$gte` / `$lt` / `$lte` / `$ne`), 목록 포함(`$in` / `$nin`), 복합 조건(`$and` / `$or`)을 지원합니다. 같은 질문 "저장장치 속도"인데도 `where` 조건만으로 후보 집합이 정확히 좁혀집니다. 예를 들어 `year >= 2020` 필터는 2022·2024년 문서만, `nvme AND year >= 2023` 은 2024년 한 건만 남깁니다.

이 필터링은 정확도 향상을 넘어 보안에도 쓰입니다. 사용자가 권한 있는 문서만, 최신 버전만, 특정 부서 문서만 검색하도록 제한하는 것을 메타데이터 필터로 구현합니다.

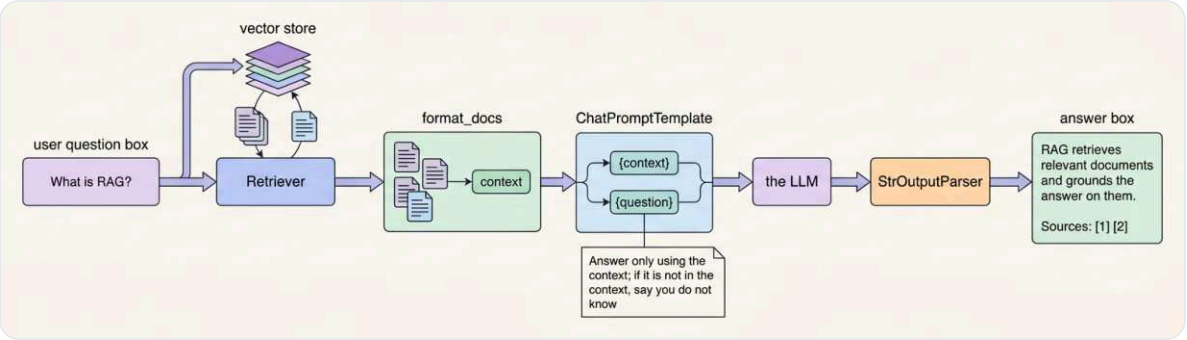
> **핵심 포인트:** 메타데이터 필터는 벡터 유사도에 정형 조건을 더해 후보를 먼저 좁힙니다. 최신 문서만·특정 출처만·권한 있는 문서만 검색하는 RAG 정확도와 보안의 핵심입니다.

RAG Chain 구축: 환각 방지와 출처

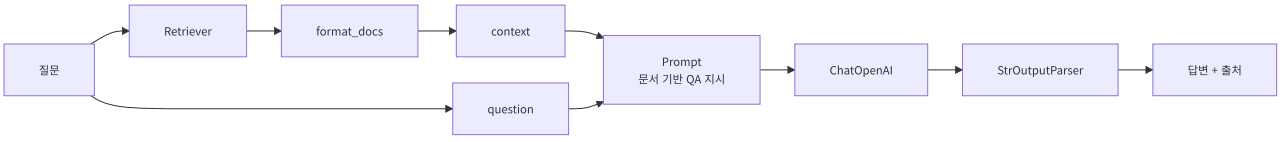
> Retriever·Prompt·LLM을 하나의 체인으로 잇고, 환각을 막고 출처를 표기하는 신뢰할 수 있는 RAG를 완성합니다.

1. RAG 체인의 전체 그림

지금까지 만든 부품들이 하나로 합쳐지는 단계입니다. 질문이 들어오면 Retriever가 관련 청크를 찾고, 그 청크들을 Prompt의 `{context}` 자리에 끼우며, LLM이 그 컨텍스트만 근거로 답합니다. 이 흐름을 LCEL 파이프 하나로 선언합니다.



RAG 체인: 질문 → Retriever → 검색된 context를 프롬프트에 결합 → LLM → 답변+출처. '문서에 없으면 모른다'는 지시로 환각을 막는다



> **핵심 포인트:** RAG 체인은 "검색 → 컨텍스트 조립 → 프롬프트 → LLM"을 LCEL 파이프로 연결한 것입니다. 프롬프트가 LLM의 행동 규칙을 정합니다.

2. 표준 RAG 체인과 환각 방지

RAG의 목적은 LLM이 자기 기억이 아니라 제공된 문서만 근거로 답하게 만드는 것입니다. 그런데 LLM은 문서에 없는 내용도 그럴듯하게 지어내는 경향(환각, hallucination)이 있습니다. 이를 막는 가장 직접적인 수단이 프롬프트의 명시적 지시입니다. "아래 문서만 참고하고, 문서에 없으면 모르겠다고 답하라"는 규칙을 System Prompt에 박아 넣습니다.

```
# 2.langchain/7.RAG/4.rag_chain/4.1_standard_chain.py
retriever = store.as_retriever(search_kwargs={"k": 3})

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
prompt = ChatPromptTemplate.from_messages([
    ("system",
        "당신은 문서 기반 QA 시스템입니다. 아래 문서만 참고해서 답하세요. "
        "문서에 없으면 '모르겠습니다'라고 답하세요.\n\n문서:\n{context}"),
    ("user", "{question}"),
])
```

`temperature=0`도 환각 억제의 일부입니다. 무작위성을 없애 모델이 가장 확실한 답에 수렴하게 합니다. 체인은 `RunnablePassthrough.assign`으로 입력 질문을 보존하면서 검색 결과를 `context` 키로 추가하는 표준형으로 구성합니다.

```
# 2.langchain/7.RAG/4.rag_chain/4.1_standard_chain.py
def format_docs(docs):
    return "\n\n".join(d.page_content for d in docs)

chain = (
    RunnablePassthrough.assign(context=lambda x: format_docs(retriever.invoke(x["question"
    | prompt
    | llm
    | StrOutputParser()
)

print(chain.invoke({"question": "NVMe 와 SATA SSD 의 차이?"}))
```

`RunnablePassthrough.assign`은 입력 딕셔너리 `{"question": ...}`를 그대로 통과시키면서 `context` 키만 새로 채웁니다. 덕분에 프롬프트가 `{question}`과 `{context}`를 모두 받을 수 있습니다. 이 구조에서 문서에 없는 질문을 던지면 모델은 지어내는 대신 "모르겠습니다"라고 답합니다.

> **핵심 포인트:** 환각 방지의 핵심은 "문서만 참고, 없으면 모른다고 답하라"는 프롬프트 지시와 `temperature=0`입니다. 모르는 것을 모른다고 답하는 것이 신뢰의 출발점입니다.

3. 출처 인용 부착

운영 환경에서는 왜 이렇게 답했는지를 추적할 수 있어야 합니다. 답변 끝에 참고한 문서의 출처를 자동으로 붙이면 신뢰도가 올라갑니다. 검색 결과를 두 곳에 동시에 쓰는 것이 요령입니다. 하나는 프롬프트의 컨텍스트로, 다른 하나는 출처 목록으로 활용합니다. 이를 위해 Retriever를 한 번만 호출하고 결과를 재사용합니다.

```python
# 2.langchain/7.RAG/4.rag_chain/4.2_with_citations.py
def format_docs_with_index(docs):
    """청크에 [1], [2] 번호를 붙여서 인용 가능하게"""
    return "\n\n".join(f"[{i}] {d.page_content}" for i, d in enumerate(docs, 1))

def extract_sources(docs):
    """검색된 청크의 출처를 중복 제거해서 리스트로"""
    seen, sources = set(), []
    for d in docs:
        src = d.metadata.get("source", "unknown")
        if src not in seen:
            seen.add(src)
            sources.append(src)
    return sources

# 검색 결과를 두 곳에 동시에 사용 — context(문자열), sources(리스트)
def retrieve_and_split(inputs):
    docs = retriever.invoke(inputs["question"])
    return {
        "question": inputs["question"],
        "context":  format_docs_with_index(docs),
        "sources":  extract_sources(docs),
    }
```

`retrieve_and_split`이 검색을 한 번 수행해 `context`(번호 붙은 문자열)와 `sources`(중복 제거한 출처 리스트)를 함께 만듭니다. 이후 체인은 컨텍스트로 답변을 생성하고, 그 답변에 출처를 덧붙입니다.

```python
# 2.langchain/7.RAG/4.rag_chain/4.2_with_citations.py
# answer 생성 후 sources 와 합쳐서 최종 텍스트 만들기
def append_sources(d):
    src_lines = "\n".join(f"  - {s}" for s in d["sources"])
    return f"{d['answer']}\n\n📚 참고 문서:\n{src_lines}"

chain = (
    RunnableLambda(retrieve_and_split)
    | RunnablePassthrough.assign(
        answer=(prompt | llm | StrOutputParser())
    )
    | RunnableLambda(append_sources)
)

print(chain.invoke({"question": "NVMe 의 인터페이스가 뭐고 어떤 장점이 있어?"}))
```

`RunnablePassthrough.assign(answer=...)` 은 `retrieve_and_split` 이 만든 딕셔너리를 보존하면서 `answer` 키에 LLM 답변을 추가합니다. 마지막 `append_sources` 가 답변과 출처를 하나의 텍스트로 합칩니다. 메타데이터의 `source` 를 활용하므로, 앞서 로더 단계에서 출처를 부여해 둔 것이 여기서 효과를 발휘합니다.

4. 번호형 인라인 인용

논문이나 리포트처럼 "이 문장의 근거가 몇 번 문서인지"를 정확히 표기하려면 인라인 인용으로 확장합니다. 핵심은 프롬프트가 각 문장 끝에 `[1]`, `[2]` 같은 근거 번호를 달도록 명시적으로 지시하는 것입니다.

```python
# 2.langchain/7.RAG/4.rag_chain/4.3_numbered_citations.py
# ★ 프롬프트가 '인라인 인용' 을 명시적으로 지시 — 이게 4.2 와의 핵심 차이
prompt = ChatPromptTemplate.from_messages([
    ("system",
        "다음 번호가 매겨진 문서들만 참고해 한국어로 답하세요.\n"
        "각 문장 끝에, 근거가 된 문서 번호를 [1], [2] 처럼 표기하세요.\n"
        "한 문장이 여러 문서를 근거로 하면 [1][2] 처럼 모두 표기합니다.\n"
        "문서에 없으면 '모르겠습니다'.\n\n"
        "문서:\n{context}"),
    ("user", "{question}"),
])

def numbered_sources(docs):
    """청크 번호별 출처 — 중복 제거 안 함 (인라인 [n] 과 1:1 대응)"""
    return [f"[{i}] {d.metadata.get('source', 'unknown')}" for i, d in enumerate(docs, 1)]
```

여기서는 출처를 중복 제거하지 않습니다. 같은 파일이라도 청크가 다르면 번호도 달라야, 답변 속 `[1]`, `[2]` 인라인 표기와 출처 목록이 1:1로 대응되기 때문입니다. 체인 구조는 4.2와 동일하며, `numbered_sources` 만 교체됩니다.

```python
# 2.langchain/7.RAG/4.rag_chain/4.3_numbered_citations.py
def retrieve_and_split(inputs):
    docs = retriever.invoke(inputs["question"])
    return {
        "question": inputs["question"],
        "context":  format_docs_with_index(docs),
        "sources":  numbered_sources(docs),
    }

chain = (
    RunnableLambda(retrieve_and_split)
    | RunnablePassthrough.assign(answer=(prompt | llm | StrOutputParser()))
    | RunnableLambda(append_sources)
)
```

이렇게 하면 답변의 각 문장이 어느 청크에서 비롯됐는지 추적할 수 있습니다. 두 인용 방식의 차이를 정리하면 다음과 같습니다.

방식	출처 표기	중복 처리	적합한 경우
4.2 끝부분 인용	답변 끝에 파일명 목록	중복 제거	일반 QA, 출처 확인
4.3 번호형 인라인	문장마다 [n] + 번호별 출처	청크별로 유지	논문·리포트, 문장 단위 근거 추적

> **핵심 포인트:** 출처 표기는 검색 결과를 한 번 받아 컨텍스트와 출처 양쪽에 쓰는 구조로 만듭니다. 끝부분 인용은 간결한 신뢰 확보용, 번호형 인라인 인용은 문장 단위 근거 추적용입니다.

DAY

3

RAG를 넘어 Agent로

스스로 검색을 결정하고 도구를 쓰는 에이전트로 진화한다

PART 5 에이전트와 도구 호출 기초

PART 6 Agentic RAG와 최종 프로젝트

PART 05

에이전트와 도구 호출 기초

1. 에이전트란 무엇인가: RAG의 한계와 Agent
2. 도구 호출의 원리: bind_tools
3. 커스텀 도구 만들기: @tool과 Pydantic
4. 첫 에이전트: ReAct 루프와 create_agent

에이전트란 무엇인가: RAG의 한계와 Agent

> 앞서 만든 RAG는 "무조건 검색하고 답하는" 고정 파이프라인이었습니다. 이제 스스로 판단해 도구를 고르는 에이전트로 확장합니다.

앞서 RAG의 원리를 손으로 구현하고, LangChain으로 표준화하며 문서 기반 QA를 만들어 왔습니다. 그런데 지금까지의 RAG에는 공통된 한 가지 전제가 있었습니다. **어떤 질문이 들어와도 항상 "검색 → 생성"이라는 같은 순서를 밟는다**는 것입니다. 이번 챕터에서는 이 고정 흐름의 한계를 짚고, 그 한계를 넘는 **에이전트(Agent)** 라는 개념을 소개합니다.

1. 고정 RAG 파이프라인의 한계

앞서 만든 RAG의 질의 처리를 떠올려 봅시다. 사용자가 "안녕하세요"라고 인사해도, "2 더하기 3은?"이라고 물어도, 코드는 무조건 벡터 DB를 검색한 뒤 그 결과를 컨텍스트로 LLM을 호출합니다. 문서와 무관한 질문에도 검색이 일어나고, 한 번 검색해서 결과가 부실해도 그대로 답을 만들어 냅니다.

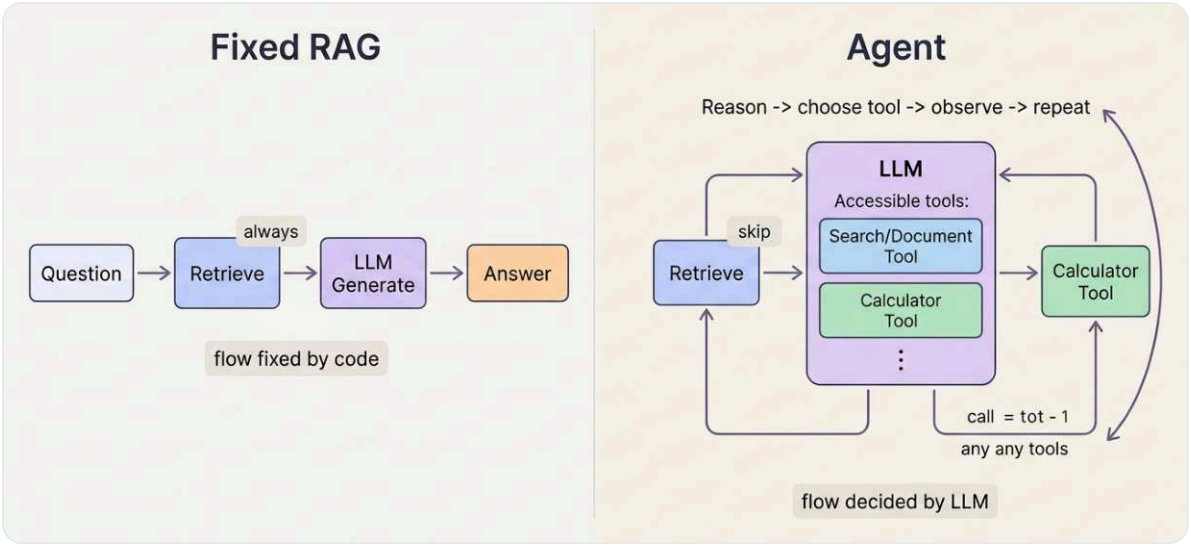
```
# 고정 RAG 의 질의 흐름 – 질문 종류와 무관하게 항상 같은 순서
docs = vectorstore.similarity_search(question, k=3)   # 무조건 검색
context = "\n\n".join(d.page_content for d in docs)
answer = llm.invoke(f"문서:\n{context}\n\n질문: {question}")  # 무조건 생성
```

이 방식은 "문서 기반 QA"라는 좁은 목적에는 충분하지만, 다음과 같은 상황에서 한계를 드러냅니다.

- **검색이 불필요한 질문**에도 검색 비용과 지연이 든다 (인사, 단순 계산, 일반 상식).
- **한 번의 검색으로 부족한 질문**에 재시도하지 못한다 (검색어를 바꿔 다시 찾아야 하는 경우).
- **검색 외의 능력**이 필요한 질문에 무력하다 (계산, 오늘 날씨, 최신 뉴스 등 문서에 없는 정보).

2. 에이전트: 흐름을 LLM이 결정한다

에이전트는 이 고정 순서를 뒤집습니다. **무엇을 할지(검색할지, 계산할지, 그냥 답할지)를 코드가 아니라 LLM이 매 순간 판단**합니다. 개발자는 순서를 짜는 대신, LLM이 쓸 수 있는 **도구(Tool)** 목록을 제공합니다. 그러면 LLM은 질문을 보고 "지금 어떤 도구가 필요한가"를 스스로 결정합니다.



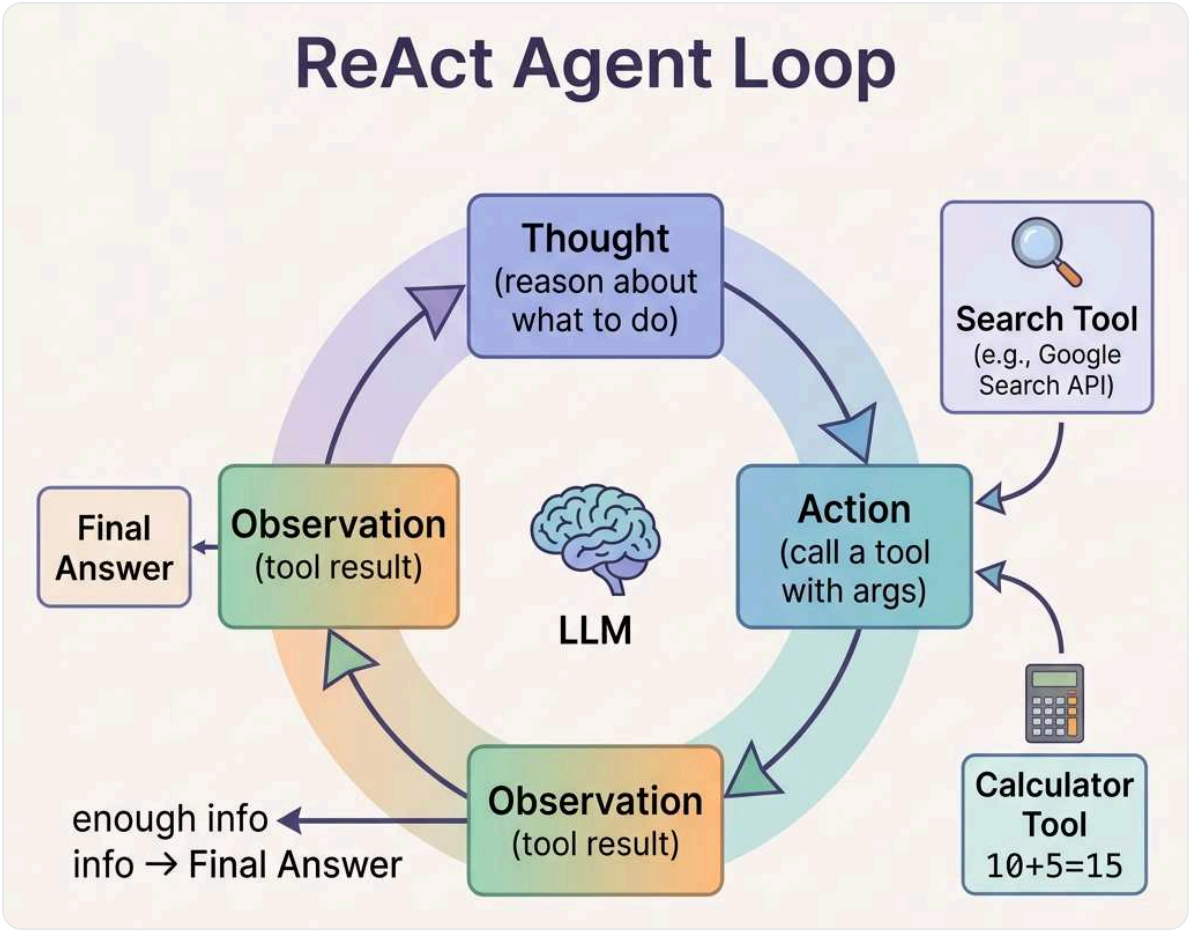
고정 RAG 파이프라인과, 도구를 쥐고 스스로 흐름을 정하는 에이전트 루프의 비교

구분	고정 RAG	에이전트
흐름 결정	개발자가 코드로 고정	LLM이 매 단계 판단
검색	항상 1회	필요할 때만, 여러 번 가능
능력	문서 검색 1가지	도구 여러 개(검색·계산·API…)
종료	1회 생성 후 끝	답할 수 있을 때까지 반복

3. ReAct: 생각하고, 행동하고, 관찰한다

에이전트가 흐름을 결정하는 대표 패턴이 **ReAct(Reasoning + Acting)** 입니다. LLM은 한 번에 답을 내놓는 대신, "생각(Thought) → 행동(Action, 도구 호출) → 관찰(Observation, 도구 결과)"을 충분할 때까지 반복합니다.

```
질문: "우리 회사 연차는 며칠이고, 그게 작년 15일보다 며칠 늘어난 거야?"

  Thought: 연차 규정을 모른다. 핸드북을 검색하자.
  Action:  search_handbook("연차 일수")
  Observation: "입사 1년 후 15일 부여, 이후 매년 1일씩 증가"
  Thought: 올해 16일이라 가정. 15일과의 차이를 계산하자.
  Action:  calculator("16 - 15")
  Observation: 1
  Thought: 이제 답할 수 있다.
  최종 답변: 올해 연차는 16일로, 작년보다 1일 늘었습니다.
```



ReAct 루프: 생각(Thought)→행동(Action, 도구 호출)→관찰(Observation)을 반복하다 답이 충분해지면 종료

핵심은 **루프**입니다. 한 번의 도구 호출로 끝나지 않고, 결과를 보고 다음 행동을 다시 정합니다. 이 루프 덕분에 에이전트는 검색을 건너뛰거나, 여러 번 검색하거나, 검색과 계산을 조합하는 등 질문에 맞춰 유연하게 움직입니다.

4. 이번 과정에서 만들 것

목표는 분명합니다. **앞서 만든 RAG 검색을 "도구"로 바꿔 에이전트에 등록하는 것**입니다. 그러면 우리가 만든 RAG는 "항상 검색하는 파이프라인"에서 "필요할 때 스스로 검색하는 에이전트"로 바뀝니다. 이를 위해 다음 순서로 학습합니다.

1. **도구 호출의 원리**(`bind_tools`): LLM이 도구를 부르는 저수준 메커니즘
2. **커스텀 도구 만들기**(`@tool`, Pydantic): 내 함수를 LLM이 쓸 수 있는 도구로
3. **첫 에이전트**(`create_agent`): ReAct 루프를 한 줄로
4. **Retriever를 도구로**: RAG 검색을 에이전트에 연결하는 Agentic RAG
5. **검색 판단과 재시도 루프**: 검색 여부와 재질의를 스스로 결정
6. **대화 기억**: 멀티턴 에이전트
7. **최종 프로젝트**: 도구를 쓰는 RAG 에이전트

> **핵심 포인트:** RAG는 "검색해서 답하는" 고정 파이프라인이고, 에이전트는 "무엇을 할지 LLM이 판단하는" 동적 루프입니다. 이 과정의 목표는 **RAG 검색을 에이전트의 도구로 바꿔, 검색 여부를 스스로 판단하는 RAG를 만드는 것**입니다.

도구 호출의 원리: bind_tools

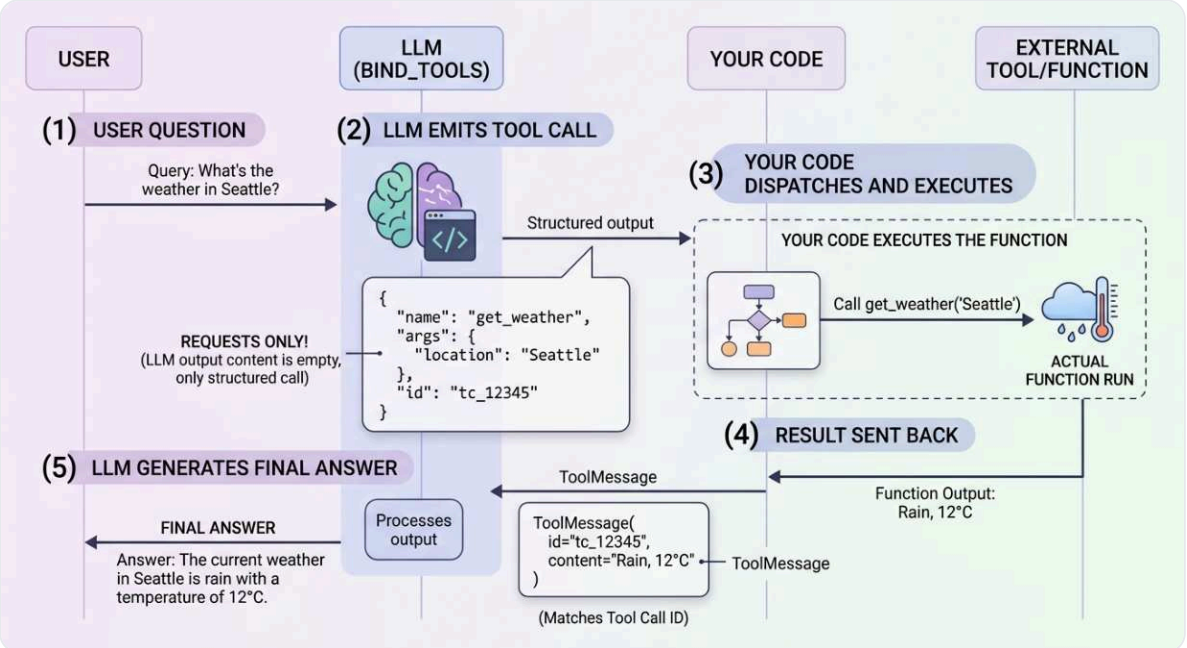
> 에이전트가 도구를 쓴다는 것은 결국 LLM이 "어떤 함수를 어떤 인자로 부를지" JSON으로 알려 주는 일입니다.
> 그 저수준 메커니즘을 손으로 한 번 따라가 봅니다.

에이전트를 한 줄로 만들어 주는 `create_agent` (다음 챕터)는 편리하지만, 그 안에서 무슨 일이 벌어지는지 모르면 디버깅이 어렵습니다. 이 챕터에서는 자동 루프를 걷어 내고, **도구 호출 → 결과 회수 → 최종 응답**의 한 사이클을 직접 손으로 구현합니다. 이 한 사이클이 모든 에이전트 동작의 기본 단위입니다.

1. LLM은 도구를 "직접 실행"하지 않는다

가장 먼저 바로잡아야 할 오해가 있습니다. LLM은 함수를 직접 실행하지 못합니다. LLM이 할 수 있는 일은 **"이 함수를, 이 인자로 불러 주세요"라는 구조화된 요청(tool call)을 내놓는 것**뿐입니다. 실제 실행은 우리 코드가 담당합니다. 그 결과를 다시 LLM에 돌려주면, LLM이 그것을 근거로 최종 답을 만듭니다.

```
LLM이 하는 일:  "add(a=3, b=5) 를 불러줘"   ← JSON 요청만 생성
우리 코드가 할 일: 실제로 add(3, 5)=8 실행 → 결과를 LLM에 회신
LLM이 하는 일:  "결과는 8입니다"           ← 결과 보고 최종 답
```



도구 호출 흐름: 질문→LLM이 tool_call(JSON) 생성→코드가 실행→ToolMessage로 결과 회신→LLM 최종 답변

2. bind_tools로 LLM에 도구를 알린다

먼저 도구가 될 함수를 정의하고(자세한 작성법은 다음 챕터), `bind_tools`로 LLM에 "이런 도구들이 있다"고 알려 줍니다.

```python
# 2.langchain/8.agents/4.internals/4.1_bind_tools.py
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage, ToolMessage

@tool
def add(a: int, b: int) -> int:
    """두 정수 a 와 b 를 더한다."""
    return a + b

@tool
def multiply(a: int, b: int) -> int:
    """두 정수 a 와 b 를 곱한다."""
    return a * b

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
llm_with_tools = llm.bind_tools([add, multiply])    # LLM 에 도구 목록 알리기
```

`bind_tools`는 LLM 자체를 바꾸지 않고, "이 도구들을 쓸 수 있다"는 정보가 덧붙은 새 객체를 돌려줍니다. 이제 이 `llm_with_tools`에 질문을 던지면, LLM은 답 대신 **도구 호출**을 내놓을 수 있습니다.

3. 1차 호출: LLM이 어떤 도구를 부를지 결정

```python
# 2.langchain/8.agents/4.internals/4.1_bind_tools.py
question = "3 더하기 5 를 계산하고, 그 결과에 7 을 곱해줘."
response = llm_with_tools.invoke([HumanMessage(content=question)])

print(response.content)        # 보통 빈 문자열 – 답 대신 도구를 호출하므로
for call in response.tool_calls:
    print(f"{call['name']}({call['args']})  [id={call['id']}]")
    # → add({'a': 3, 'b': 5})  [id=call_abc...]
```

응답의 `.content`는 비어 있고, 대신 `.tool_calls`에 호출 요청이 담깁니다. 각 호출에는 도구 이름(`name`), 인자(`args`), 그리고 결과를 짝지어 돌려보낼 때 쓸 고유 ID(`id`)가 들어 있습니다. LLM은 docstring과 타입 힌트만 보고 `add`에 `a=3, b=5`를 넣어야 한다는 걸 스스로 판단했습니다.

4. 도구 실행 후 결과를 ToolMessage로 회신

LLM은 실행을 못 하므로, 우리가 직접 함수를 호출하고 그 결과를 `ToolMessage`로 감싸 대화에 덧붙입니다. 이때 `tool_call_id`로 어떤 호출에 대한 결과인지 짝지어 줍니다.

```
# 2.langchain/8.agents/4.internals/4.1_bind_tools.py
tool_map = {"add": add, "multiply": multiply}

messages = [HumanMessage(content=question), response]   # 질문 + LLM의 도구호출
for call in response.tool_calls:
    result = tool_map[call["name"]].invoke(call["args"])   # 실제 실행
    messages.append(ToolMessage(content=str(result), tool_call_id=call["id"]))
```

5. 2차 호출: 결과를 보고 최종 답변

도구 결과가 붙은 대화를 다시 LLM에 넣으면, LLM은 그 결과를 근거로 사람이 읽을 답을 만듭니다.

```
# 2.langchain/8.agents/4.internals/4.1_bind_tools.py
final = llm_with_tools.invoke(messages)
print(final.content)   # → "8에 7을 곱하면 56입니다."
```

여기서 중요한 점: 만약 두 번째 응답에 **또** `tool_calls`가 있으면(예: `add` 다음 `multiply`가 필요하면), 같은 과정을 한 번 더 반복해야 합니다. 이 "도구 호출이 더 없을 때까지 반복"이 바로 **ReAct 루프**이고, 다음 챕터의 `create_agent`가 자동으로 돌려 주는 부분입니다.

```
1차 호출 → tool_calls 있음? → 실행·회신 → 2차 호출
2차 호출 → tool_calls 또 있음? → 실행·회신 → 3차 호출 → ...
       → tool_calls 없음? → 그것이 최종 답변 (루프 종료)
```

> **핵심 포인트:** 도구 호출은 4단계입니다. ① `bind_tools`로 도구 알리기 → ② LLM이 `.tool_calls`로 호출 요청 → ③ 우리가 실행하고 `ToolMessage`로 회신 → ④ LLM이 결과 보고 최종 답. 이 한 사이클을 "더 부를 도구가 없을 때까지 반복"하면 에이전트가 됩니다.

커스텀 도구 만들기: @tool과 Pydantic

> 도구의 docstring과 타입 힌트는 사람이 읽는 주석이 아니라 LLM이 읽는 명세입니다. 잘 쓴 도구 설명이 곧 에이전트의 정확도입니다.

앞 챕터에서 `add`, `multiply`를 `@tool`로 도구화해 썼습니다. 이번에는 그 `@tool` 데코레이터를 자세히 살펴봅니다. 여기서 가장 중요한 점은, 함수의 docstring과 타입 힌트가 "사람을 위한 설명"이 아니라 "LLM에게 보여 줄 명세"라는 사실입니다. LLM은 이 명세만 보고 도구를 언제 어떤 인자로 부를지 결정하므로, 명세의 품질이 곧 에이전트의 품질을 좌우합니다.

1. @tool: 함수를 도구로 만드는 가장 단순한 방법

평범한 파이썬 함수에 `@tool`을 붙이면 LLM이 쓸 수 있는 도구가 됩니다. 규칙은 세 가지뿐입니다.

```python
# 2.langchain/8.agents/2.custom_tools/2.2_at_tool_basic.py
from langchain_core.tools import tool

@tool
def get_word_length(word: str) -> int:
    """단어의 글자 수를 센다."""
    return len(word)
```

- **함수에 `@tool` 붙이기** → 도구로 등록
- **docstring** → LLM이 보는 "이 도구가 무엇을 하는가" 설명
- **타입 힌트**(`word: str`, `-> int`) → LLM이 보는 "인자 스키마"

세 가지가 모여 LLM에게 전달되는 명세를 이룹니다. docstring이 비어 있거나 모호하면 LLM은 도구를 언제 써야 할지 몰라 헤맵니다. **docstring은 사용 설명서라고 생각하고 또렷하게** 적습니다.

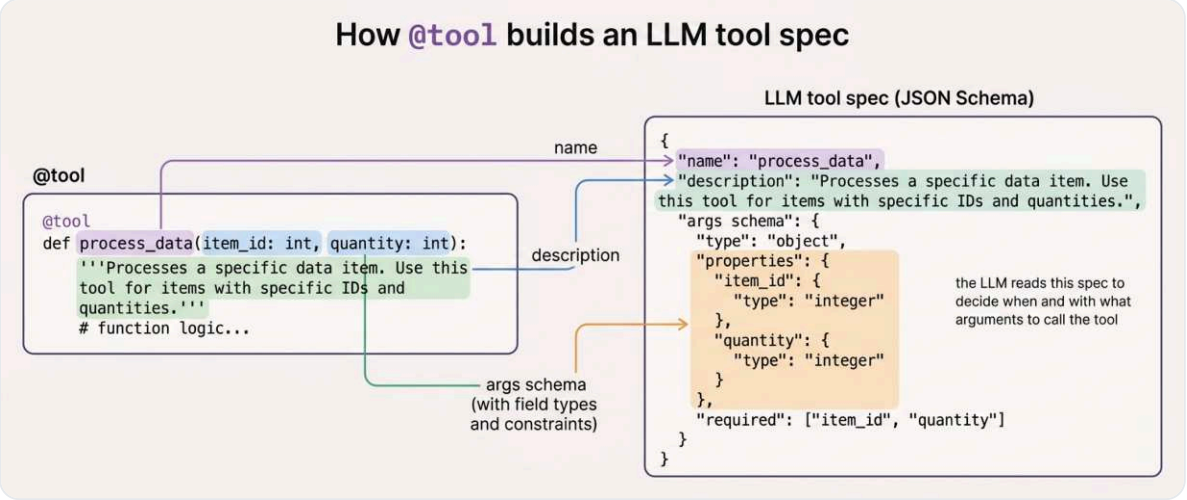
```python
# 2.langchain/8.agents/2.custom_tools/2.2_at_tool_basic.py
@tool
def calculate_tip(amount: float, percent: float) -> float:
    """음식점 영수증 금액과 팁 비율(%) 을 받아 팁 금액을 계산한다."""
    return amount * percent / 100
```

2. 도구 명세는 LLM에게 이렇게 보인다

`@tool`이 함수에서 자동으로 뽑아낸 명세는 `.name`, `.description`, `.args`로 확인할 수 있습니다.

```
print(get_word_length.name)          # get_word_length
print(get_word_length.description)   # 단어의 글자 수를 센다.
print(get_word_length.args)          # {'word': {'type': 'string', ...}}
```

LLM은 이 정보를 JSON Schema로 받아, 사용자의 말에서 어떤 도구가 맞고 인자에 무엇을 넣을지 추론합니다. 즉 **함수 이름·docstring·타입 힌트를 잘 짓는 것이 프롬프트 엔지니어링의 일부**입니다.



@tool 데코레이터가 함수에서 LLM 명세를 뽑아내는 구조: 함수 이름→도구 이름, docstring→설명, 타입 힌트·Pydantic→인자 스키마(JSON Schema)

3. Pydantic으로 인자를 강하게 명세하기

타입 힌트만으로 충분한 경우가 많지만, 인자가 복잡하거나 LLM이 자주 헷갈리는 값이라면 **Pydantic 모델**로 인자 하나하나에 설명과 제약을 붙일 수 있습니다.

```python
# 2.langchain/8.agents/2.custom_tools/2.4_pydantic_args.py
from typing import Literal
from pydantic import BaseModel, Field
from langchain_core.tools import tool

class SendEmailInput(BaseModel):
    """이메일 전송 도구의 인자."""
    to: str = Field(description="수신자 이메일 주소 (반드시 유효한 이메일 형식)")
    subject: str = Field(description="이메일 제목 (50자 이내, 간결하게)")
    body: str = Field(description="이메일 본문 (반드시 한국어로 작성)")
    priority: Literal["low", "normal", "high"] = Field(
        default="normal",
        description="우선순위. urgent 한 경우에만 high 사용",
    )

@tool(args_schema=SendEmailInput)
def send_email(to: str, subject: str, body: str, priority: str = "normal") -> str:
    """사용자가 요청하면 이메일을 보낸다."""
    return f"이메일이 {to} 에게 전송되었습니다 (priority={priority})."
```

Pydantic이 주는 무기는 세 가지입니다.

- `Field(description=...)`: 인자마다 가이드를 달아 LLM이 정확히 채우도록 유도합니다.
- `Literal["low", "normal", "high"]`: 값을 enum으로 강제해, LLM이 정해진 값 외에는 넣지 못하게 합니다.
- `Field(ge=1, le=20)`: 범위와 검증을 자동 적용합니다(예: `max_results`를 1~20으로 제한).

4. description이 정확도를 좌우한다

description을 잘 달면 LLM이 자연어의 뉘앙스를 인자에 정확히 반영합니다.

```python
# 2.langchain/8.agents/2.custom_tools/2.4_pydantic_args.py
queries = [
    "alice@example.com 에게 '회의 일정 변경' 제목으로 이메일 보내줘. 긴급해.",
    "파이썬 비동기 자료 5개만 날짜순으로 검색해줘.",
]
# "긴급해" → priority="high"   (Literal enum 인식)
# "날짜순" → sort_by="date"    (Field description 인식)
```

사용자가 "긴급해"라고만 해도 LLM은 `priority="high"`를 골라냅니다. 이는 코드가 아니라 **명세에 적힌 설명** 덕분입니다. 도구가 자주 틀린 인자를 받는다면, 함수 본문을 고치기 전에 먼저 description부터 다듬어야 합니다.

> **핵심 포인트:** `@tool` + docstring + 타입 힌트면 도구가 됩니다. docstring/타입 힌트는 LLM이 읽는 명세이므로 또렷하게 적습니다. 복잡한 인자는 Pydantic의 `Field(description=...)`·`Literal`·검증으로 강하게 명세하면 LLM의 인자 정확도가 크게 올라갑니다.

첫 에이전트: ReAct 루프와 create_agent

> 앞서 손으로 짠 "도구 호출 → 실행 → 재호출" 루프를, 이제 `create_agent` 한 줄에 맡깁니다. 우리는 도구만 주면 됩니다.

02 챕터에서 도구 호출 한 사이클을 손으로 돌렸고, "도구가 더 없을 때까지 반복하면 에이전트"라고 했습니다. 그 반복 루프를 직접 짤 필요는 없습니다. LangChain의 `create_agent`가 ReAct 루프 전체를 자동으로 처리합니다. 이번 챕터에서 실제로 동작하는 첫 에이전트를 만듭니다.

1. create_agent: 도구 쓰는 LLM을 한 줄로

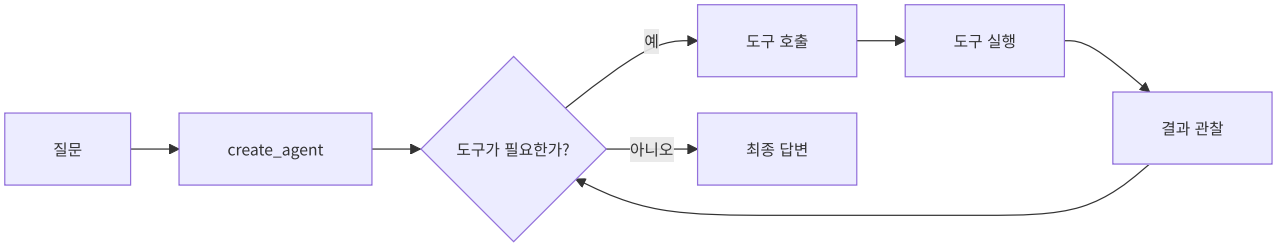
도구를 정의하고, LLM과 함께 `create_agent`에 넘기면 끝입니다.

```python
# 2.langchain/8.agents/2.custom_tools/2.1_first_agent.py
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langchain.agents import create_agent

@tool
def calculator(expression: str) -> str:
    """수학 식을 계산한다. 예: '53 * 7 + 2'"""
    try:
        return str(eval(expression))
    except Exception as e:
        return f"계산 오류: {e}"

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
agent = create_agent(llm, [calculator])   # 도구 목록만 주면 ReAct 루프 자동 구성
```

에이전트가 내부에서 하는 일은 02 챕터에서 손으로 짠 그대로입니다.

1. 질문을 받고
2. 도구가 필요한지 LLM이 판단
3. 필요하면 도구 호출 → 결과 → 다시 LLM 호출 → … 반복 (ReAct 루프)
4. 도구 없이 답할 수 있게 되면 최종 답변

차이는 단 하나, **그 루프를 우리가 직접 짜지 않아도 된다**는 점입니다. 자동 루프는 내부(LangGraph)가 처리합니다.



2. 실행: messages 형식으로 입력

에이전트는 `{"messages": [...]}` 형식으로 입력받고, 전체 대화 흐름을 messages 리스트로 돌려줍니다.

```
# 2.langchain/8.agents/2.custom_tools/2.1_first_agent.py
result = agent.invoke({
    "messages": [("user", "(53 * 7 + 2) / 5 는 얼마야?")]
})
print(result["messages"][-1].content)   # 최종 답변
```

3. 에이전트 내부 흐름 들여다보기

`result["messages"]`에는 사용자 메시지, 에이전트의 도구 호출, 도구 결과, 최종 답변이 순서대로 쌓입니다. 이 흐름을 출력하면 ReAct 루프가 실제로 어떻게 돌았는지 한눈에 보입니다.

```
# 2.langchain/8.agents/2.custom_tools/2.1_first_agent.py
for m in result["messages"]:
    if hasattr(m, "tool_calls") and m.tool_calls:
        for c in m.tool_calls:
            print(f"  [도구 호출]  {c['name']}({c['args']})")
    if m.content:
        prefix = {"human": "[사용자]", "ai": "[AI]", "tool": "[도구 결과]"}.get(m.type, m.t
        print(f"  {prefix} {m.content}")
```

```
[사용자] (53 * 7 + 2) / 5 는 얼마야?
[도구 호출] calculator({'expression': '(53 * 7 + 2) / 5'})
[도구 결과] 74.6
[AI] (53 × 7 + 2) ÷ 5 는 74.6 입니다.
```

4. system_prompt로 에이전트의 성격 정하기

`system_prompt`로 에이전트가 도구를 언제·어떻게 쓸지 지침을 줄 수 있습니다. 다음 챕터에서 본격적으로 쓰게 될 패턴입니다.

```
agent = create_agent(
    llm,
    [calculator],
    system_prompt="너는 계산 도우미다. 계산이 필요하면 반드시 calculator 도구를 쓰고, "
                  "직접 암산하지 마라.",
)
```

5. API 변경 이력: create_react_agent에서 create_agent로

LangChain 1.0부터 에이전트 생성 함수가 정리되었습니다. 오래된 글이나 예제에서 본 `create_react_agent`는 이제 `create_agent`로 바뀌었습니다. 이 교재와 저장소 예제는 모두 **현행** `create_agent` 를 씁니다.

```
# 이전(deprecated)                          현행 (이 교재)
# from langgraph.prebuilt import           from langchain.agents import
#     create_react_agent                        create_agent
# create_react_agent(llm, tools,           create_agent(llm, tools,
#     prompt=system_prompt)                    system_prompt=system_prompt)
#                                           # ↑ 인자 이름 prompt → system_prompt
```

반환값은 둘 다 LangGraph 그래프라서 `.invoke({"messages": [...]})`, `.stream()`, `checkpointer=` 사용법은 동일합니다. 인터넷 예제에서 `create_react_agent`를 보면 `create_agent`로 바꿔 읽으면 됩니다.

> **핵심 포인트:** `create_agent(llm, tools)` 한 줄이 02 챕터에서 손으로 짠 ReAct 루프를 자동으로 돌려 줍니다. 우리는 좋은 도구와 명확한 `system_prompt`만 준비하면 됩니다. `result["messages"]`를 출력해 도구 호출·결과·답변의 흐름을 확인하는 습관이 디버깅의 핵심입니다.

PART 06

Agentic RAG와 최종 프로젝트

1. Retriever를 도구로: RAG를 에이전트에 연결
2. Agentic RAG: 검색 판단과 재시도 루프
3. 대화를 기억하는 에이전트: LangGraph 메모리
4. 도구 쓰는 RAG 에이전트 만들기
5. 최종 프로젝트: 통합 웹 데모로 에이전트 들여다보기

Retriever를 도구로: RAG를 에이전트에 연결

> 앞서 만든 검색기를 `@tool` 로 한 번 감싸면, 우리가 만든 RAG가 에이전트의 도구가 됩니다. 이것이 Agentic RAG의 출발점입니다.

지금까지 도구로 계산기나 이메일 같은 예시를 썼습니다. 이제 이 과정의 핵심으로 들어갑니다. **앞서 만든 벡터 검색(retriever)을 도구로 등록**하는 것입니다. 그러면 에이전트는 사용자의 질문을 보고 문서를 찾아봐야 한다고 스스로 판단해 검색을 호출하고, 문서와 무관한 질문에는 검색 없이 바로 답합니다. 이것이 고정 RAG를 **Agentic RAG**로 바꾸는 지점입니다.

1. 검색을 @tool로 감싸기

벡터스토어의 `similarity_search` 를 함수 하나로 감싸고 `@tool` 을 붙이면, 검색이 에이전트의 도구가 됩니다.

```python
# 2.langchain/8.agents/3.applied_agents/3.3_retriever_tool.py
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_core.tools import tool
from langchain_core.vectorstores import InMemoryVectorStore
from langchain.agents import create_agent

DOCS = [
    "회사 연차는 입사 1년 후 15일 부여되며, 이후 매년 1일씩 증가한다.",
    "재택근무는 주 3일까지 가능하고, 사전에 팀장 승인이 필요하다.",
    "경조사 휴가는 본인 결혼 5일, 직계가족 사망 5일이다.",
    "사내 카페는 평일 오전 8시부터 오후 6시까지 운영한다.",
]
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vectorstore = InMemoryVectorStore.from_texts(DOCS, embedding=embeddings)

@tool
def search_handbook(query: str) -> str:
    """사내 규정 핸드북에서 관련 내용을 검색한다 (연차/재택/휴가/카페 등)."""
    hits = vectorstore.similarity_search(query, k=2)
    return "\n".join(f"- {d.page_content}" for d in hits) or "관련 규정 없음"
```

여기서 docstring이 특히 중요합니다. **"이 도구가 무엇을 검색하는지"를 구체적으로 적어야** LLM이 어떤 질문에 이 도구를 써야 할지 안다. "연차/재택/휴가/카페 등"처럼 다루는 주제를 명시한 이유입니다.

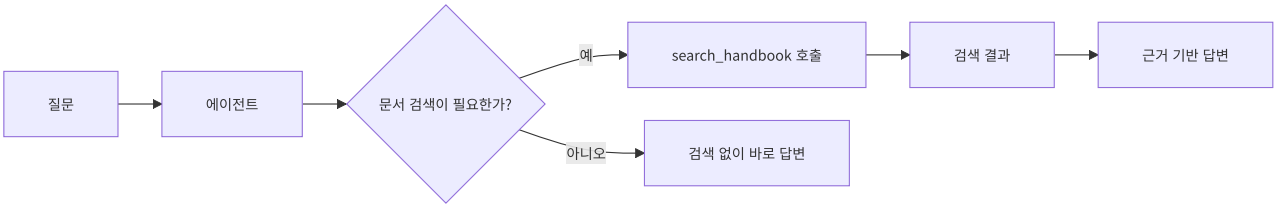
2. 에이전트에 검색 도구 연결하기

검색 도구를 `create_agent`에 넘기고, system_prompt로 "규정 질문은 검색하고, 무관한 질문은 그냥 답하라"고 지시합니다.

```
# 2.langchain/8.agents/3.applied_agents/3.3_retriever_tool.py
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
agent = create_agent(
    llm,
    [search_handbook],
    system_prompt="사내 규정 질문은 search_handbook 로 근거를 찾아 답하라. "
                  "규정과 무관한 질문은 검색 없이 그냥 답하라.",
)
```

3. 검색 여부를 에이전트가 스스로 결정한다

같은 에이전트에 성격이 다른 두 질문을 던져 봅니다.




```
# 2.langchain/8.agents/3.applied_agents/3.3_retriever_tool.py
def run(q: str):
    print(f"\nQ: {q}")
    result = agent.invoke({"messages": [("user", q)]})
    for m in result["messages"]:
        for c in getattr(m, "tool_calls", []) or []:
            print(f"  └─ 검색: {c['args'].get('query')}")
    print(f"A: {result['messages'][-1].content}")

run("연차는 며칠 받을 수 있어?")   # → search_handbook 호출 후 근거 기반 답
run("2 더하기 3은?")               # → 검색 없이 바로 답
```

```
Q: 연차는 며칠 받을 수 있어?
  └ 검색: 연차 일수
A: 입사 1년 후 15일이 부여되고, 이후 매년 1일씩 늘어납니다.

Q: 2 더하기 3은?
A: 5입니다.            ← 검색 도구를 부르지 않음
```

같은 코드, 같은 에이전트인데 첫 질문에는 검색이 일어나고 둘째 질문에는 일어나지 않았습니다. **이 판단을 코드의 if문이 아니라 LLM이 내렸다**는 점이 핵심입니다.

4. 고정 RAG와 무엇이 다른가

	고정 RAG	Retriever Tool (Agentic RAG)
검색 시점	모든 질문에 항상	LLM이 필요하다고 판단할 때만
검색 횟수	1회 고정	0회 또는 여러 회 (LLM이 결정)
무관한 질문	불필요한 검색 발생	검색 건너뛰고 바로 답
다른 도구와 조합	불가	계산·API 등과 자유롭게 섞음

앞서 쓰던 `vectorstore.similarity_search(question, k=3)` 를 `@tool` 로 감싼 것뿐인데, 검색이 "무조건 실행되는 코드"에서 "LLM이 필요할 때 부르는 도구"로 바뀌었습니다. 실제 RAG(청킹·리랭킹·출처 표시)는 앞서 배운 그대로 도구 안에 넣으면 되고, 검색 도구를 여러 개 만들면(예: 사내 규정용·기술문서용) 에이전트가 질문에 맞는 도구를 골라 씁니다.

> **핵심 포인트:** `retriever.similarity_search` 를 `@tool` 로 감싸면 그것이 곧 Agentic RAG입니다. LLM이 검색 여부와 횟수를 스스로 판단하므로, 무관한 질문에는 검색을 건너뛰고 필요한 질문에는 근거를 찾아 답합니다. 도구의 docstring에 무엇을 검색하는지 또렷이 적는 것이 정확도의 핵심입니다.

Agentic RAG: 검색 판단과 재시도 루프

> "검색할지 말지"를 넘어 "검색 결과가 부족하면 쿼리를 바꿔 다시 찾는다". 에이전트의 판단 루프를 단계별로 직접 들여다봅니다.

앞 챕터에서 `create_agent`가 검색 여부를 알아서 판단하는 모습을 확인했습니다. 이번에는 그 판단 과정을 직접 구현해 봅니다. LangGraph나 `create_agent` 없이 평범한 `if`와 `for` 루프만으로 agentic 흐름을 짜 보면, 에이전트가 내부에서 어떤 결정을 내리는지 분명하게 드러납니다.

1. 1단계: 검색이 필요한 질문인가? (if 분기)

가장 단순한 Agentic RAG는 "검색이 필요한지 LLM에게 먼저 물어보는 것"입니다.

```python
# 2.langchain/7.RAG/6.agentic/6.1_agentic_rag.py
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

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

def needs_retrieval(question: str) -> bool:
    resp = llm.invoke([
        SystemMessage(content="질문이 외부 문서 검색이 필요하면 YES, 아니면 NO. 한 단어만."),
        HumanMessage(content=question),
    ])
    return "YES" in resp.content.upper()
```

```
기존 RAG:       질문 → (무조건) 검색 → 응답
Agentic (기본):  질문 → [검색 필요?] → 필요하면 검색 후 응답 / 아니면 곧장 응답
```

이미 LLM이 흐름을 결정하기 시작했습니다. 하지만 한 번 검색해서 결과가 부실하면? 그대로 답을 만들 수밖에 없습니다. 다음 단계가 이를 해결합니다.

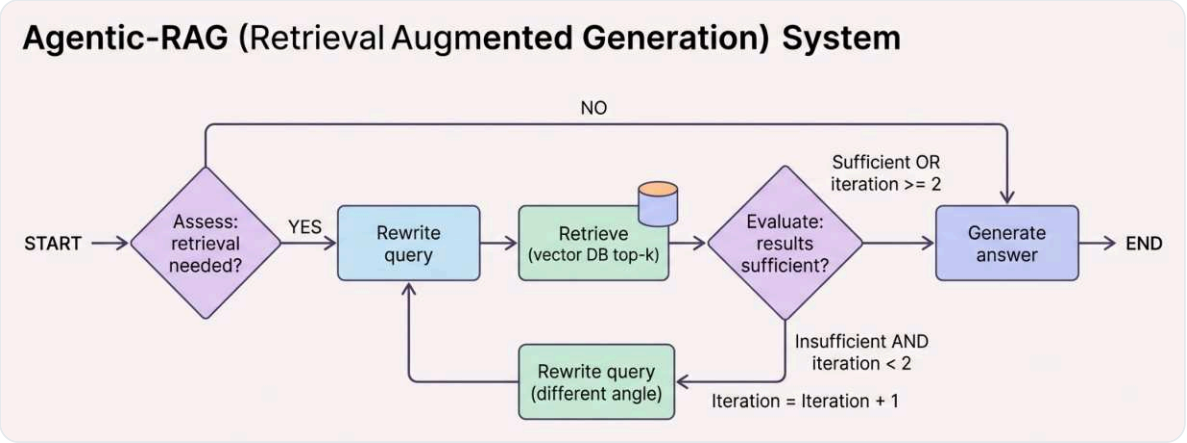
2. 2단계: 쿼리 재작성 + 충분성 평가 + 재시도 (루프)

검색 결과가 부족하면 **쿼리를 다른 각도로 바꿔 다시 검색**합니다. 이를 위해 두 도우미 함수를 더합니다. 검색용 쿼리를 다시 쓰는(쿼리 재작성, query rewriting) `rewrite_query`와, 결과가 충분한지 판단하는 `is_sufficient`입니다.

```python
# 2.langchain/7.RAG/6.agentic/6.2_agentic_rag_loop.py
def rewrite_query(question, prev_query, attempt):
    """검색용 쿼리로 재작성. 재시도면 직전 쿼리를 피해 다른 각도로."""
    if prev_query is None:
        prompt = f"다음 질문을 벡터 검색용 키워드 쿼리로 재작성. 쿼리만:\n{question}"
    else:
        prompt = (f"이전 검색이 부족했습니다. 다른 각도로 쿼리 재작성. 쿼리만.\n"
                  f"질문: {question}\n이전 쿼리: {prev_query}")
    return llm.invoke([HumanMessage(content=prompt)]).content.strip()

def is_sufficient(question, docs):
    resp = llm.invoke([
        SystemMessage(content="검색 결과가 질문에 답하기 충분하면 SUFFICIENT, 아니면 INSUFFI
        HumanMessage(content=f"질문: {question}\n\n결과:\n" + "\n".join(docs)),
    ])
    return "SUFFICIENT" in resp.content.upper()
```

이제 이 조각들을 루프로 엮습니다. 검색 → 평가 → (부족하면) 재작성 → 재검색을 최대 2회까지 반복합니다.

```python
# 2.langchain/7.RAG/6.agentic/6.2_agentic_rag_loop.py
MAX_TRIES = 2

def answer(question: str) -> str:
    if not needs_retrieval(question):
        return llm.invoke([HumanMessage(content=question)]).content    # 검색 없이 곧장

    query, documents = None, []
    for attempt in range(1, MAX_TRIES + 1):
        query = rewrite_query(question, query, attempt)
        documents = [d.page_content for d in retriever.invoke(query)]
        if is_sufficient(question, documents):    # 충분하면 루프 탈출
            break

    context = "\n".join(documents)
    prompt = f"자료만 보고 답하세요. 자료에 없으면 추측 금지.\n\n자료:\n{context}\n\n질문: {q
    return llm.invoke([HumanMessage(content=prompt)]).content
```



Agentic RAG 루프: 검색 필요 판단 → 쿼리 재작성 → 검색 → 충분성 평가 → 부족하면 재시도, 충분하면 응답

이 루프가 에이전트의 핵심 사고 패턴입니다. **결과를 평가하고, 부족하면 접근을 바꿔 다시 시도**합니다. 사람이 검색 엔진에서 검색어를 바꿔 가며 원하는 자료를 찾는 과정과 똑같습니다.

3. 3단계: 같은 로직을 LangGraph 그래프로

`if/for/break`로 짠 흐름이 복잡해지면, **LangGraph**로 옮겨 분기와 순환을 선언적으로 표현할 수 있습니다. 로직은 6.2와 동일하고 표현 방식만 바뀝니다. 함수는 **노드(node)**가 되고, `if/break`는 **조건부 엣지(conditional edge)**가 됩니다.

```
# 2.langchain/7.RAG/6.agentic/6.3_agentic_rag_langgraph.py
from langgraph.graph import StateGraph, START, END

g = StateGraph(State)
g.add_node("assess", assess_need)      # 검색 필요?
g.add_node("rewrite", rewrite_query)   # 쿼리 재작성
g.add_node("retrieve", retrieve)       # 벡터 검색
g.add_node("evaluate", evaluate)       # 결과 충분?
g.add_node("generate", generate)       # 최종 답변

g.add_edge(START, "assess")
g.add_conditional_edges("assess", route_after_assess, {"rewrite": "rewrite", "generate": "
g.add_edge("rewrite", "retrieve")
g.add_edge("retrieve", "evaluate")
g.add_conditional_edges("evaluate", route_after_eval, {"rewrite": "rewrite", "generate": "
g.add_edge("generate", END)
app = g.compile()
```

```
# 분기 함수 – 6.2 의 if 조건이 그대로 옮겨옴
def route_after_eval(s):
    # 충분하거나 2번 이상 시도했으면 답변, 아니면 다시 검색
    return "generate" if (s["is_sufficient"] or s["iteration"] >= 2) else "rewrite"
```

그래프로 그리면 "evaluate에서 부족하면 rewrite로 돌아간다"는 순환이 코드 구조에 그대로 드러나, 흐름을 한눈에 보고 단계를 추가·수정하기 쉽습니다.

> **핵심 포인트:** Agentic RAG는 세 단계로 깊어집니다. ① 검색 필요 판단(if) → ② 쿼리 재작성 + 충분성 평가 + 재시도(루프) → ③ 같은 로직을 LangGraph 그래프로. 어느 방식이든 본질은 "결과를 평가하고 부족하면 접근을 바꿔 다시 시도하는" 루프이며, `create_agent` 는 이 루프를 자동으로 돌려 주는 것입니다.

대화를 기억하는 에이전트: LangGraph 메모리

> "내가 사는 곳 날씨 어때?" 같은 후속 질문에 답하려면 에이전트가 이전 대화를 기억해야 합니다. 체크포인터(checkpointer) 한 줄로 멀티턴(multi-turn) 에이전트를 만듭니다.

지금까지의 에이전트는 매 호출이 독립적이었습니다. 한 번 질문하고 답하면 그것으로 끝이고, 다음 질문은 앞선 대화를 전혀 기억하지 못합니다. 하지만 실제 챗봇은 "내 이름은 앨리스야" 다음에 "내 이름이 뭐였지?" 같은 멀티턴 대화를 이어가야 합니다. 대화형 RAG에 Chat History를 붙이듯, 에이전트에도 **메모리**를 붙입니다. LangGraph 에이전트에서는 이 작업이 매우 간단합니다.

1. checkpointer 한 줄로 메모리 활성화

`create_agent`에 `checkpointer=MemorySaver()`를 넘기면, 에이전트가 대화와 도구 결과를 자동으로 보존합니다.

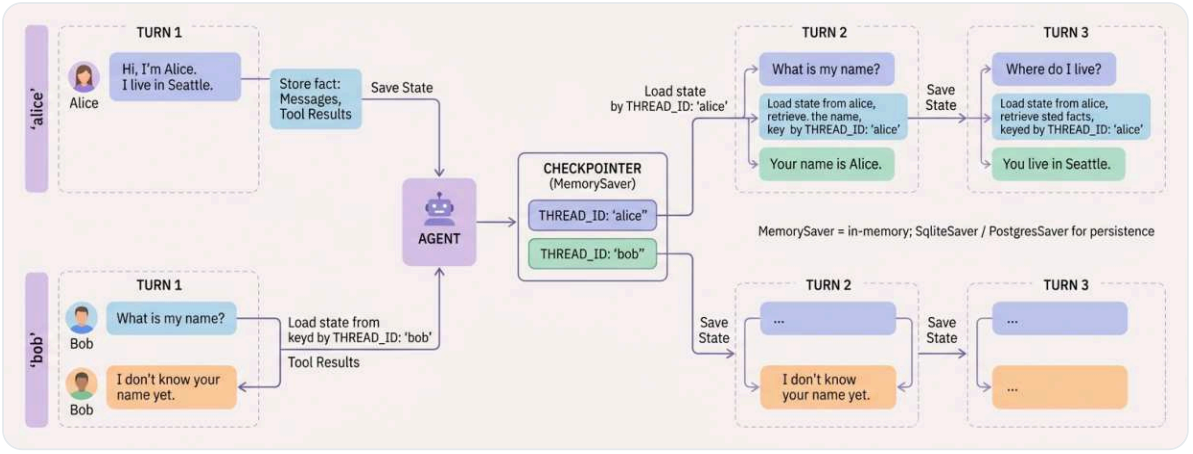
```python
# 2.langchain/8.agents/5.langgraph_memory/5.1_with_memory.py
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool
from langchain.agents import create_agent
from langgraph.checkpoint.memory import MemorySaver

@tool
def get_weather(city: str) -> str:
    """도시 날씨 조회."""
    return {"서울": "맑음, 22도", "부산": "흐림, 25도"}.get(city, "정보 없음")

checkpointer = MemorySaver()
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
agent = create_agent(
    llm,
    [get_weather],
    checkpointer=checkpointer,    # ← 이 한 줄로 메모리 활성화
)
```

2. thread_id로 세션을 구분한다

메모리를 켰으면, 호출할 때마다 **어느 대화에 속하는지**를 `thread_id`로 알려 줍니다. 같은 `thread_id`는 같은 대화로 이어지고, 다른 `thread_id`는 완전히 격리된 새 대화입니다.

```python
# 2.langchain/8.agents/5.langgraph_memory/5.1_with_memory.py
def chat(user_input: str, thread_id: str):
    result = agent.invoke(
        {"messages": [("user", user_input)]},
        config={"configurable": {"thread_id": thread_id}},    # ← 세션 구분
    )
    print(f"[{thread_id}] A: {result['messages'][-1].content}")
```

> 대화형 RAG의 Chat History에서 쓰는 `session_id`와 같은 역할입니다. LangGraph에서는 이를 `thread_id`라 부릅니다.



thread_id별 메모리 격리: 같은 thread는 이전 대화·도구 결과를 이어받고, 다른 thread는 독립된 세션으로 분리. MemorySaver가 체크포인트로 상태를 보존

3. 멀티턴: 같은 thread 안에서 맥락 유지

같은 `thread_id`로 대화를 이어가면, 에이전트가 앞선 발화를 기억합니다.

```python
# 2.langchain/8.agents/5.langgraph_memory/5.1_with_memory.py
chat("내 이름은 앨리스야. 기억해줘.",  "alice")
chat("나는 서울에 살아.",            "alice")
chat("내가 사는 곳 날씨 어때?",       "alice")  # 도구가 자동으로 "서울" 사용
chat("내 이름이 뭐였지?",            "alice")  # 첫 메시지 기억
```

```
[alice] A: 네, 앨리스님 기억하겠습니다.
[alice] A: 서울에 사시는군요!
[alice] A: 서울은 맑음, 22도입니다.   ← "사는 곳"=서울을 기억해 get_weather("서울") 호출
[alice] A: 앨리스님이세요.            ← 첫 메시지를 기억
```

세 번째 질문 "내가 사는 곳 날씨 어때?"가 핵심입니다. 에이전트는 두 번째 발화에서 "서울에 산다"는 걸 기억하고 있다가, `get_weather`를 호출할 때 인자로 `"서울"`을 스스로 채워 넣었습니다. **기억 + 도구 사용이 결합**된 것입니다.

4. 다른 thread는 완전히 격리

```python
# 2.langchain/8.agents/5.langgraph_memory/5.1_with_memory.py
chat("내 이름이 뭐였지?", "bob")   # bob 은 alice 의 정보를 전혀 모름
# [bob] A: 죄송하지만 이름을 알려주신 적이 없습니다.
```

`thread_id`가 다르면 메모리가 분리되므로, 여러 사용자가 각자의 대화를 안전하게 이어갈 수 있습니다. 웹 서비스에서는 사용자/세션마다 `thread_id`를 부여하면 됩니다.

5. 메모리 저장소 선택

`MemorySaver`는 프로세스 메모리에만 저장하므로 서버를 재시작하면 사라집니다. 영속화가 필요하면 저장소만 교체하면 되고, 에이전트 코드는 그대로 둡니다.

체크포인터	저장 위치	용도
`MemorySaver()`	프로세스 메모리	개발·데모 (재시작 시 사라짐)
`SqliteSaver(path)`	SQLite 파일	단일 서버 영속화
`PostgresSaver(...)`	PostgreSQL	프로덕션·다중 서버

> **핵심 포인트:** `checkpointer=MemorySaver()` 한 줄로 에이전트가 대화를 기억하고, `config` 의 `thread_id` 로 세션을 구분합니다. 같은 thread는 맥락을 잇고 다른 thread는 격리됩니다. 기억과 도구 사용이 결합되면 "내가 사는 곳 날씨"처럼 이전 대화를 근거로 도구 인자를 채우는 대화형 에이전트가 됩니다. 영속화는 체크포인터만 교체하면 됩니다.

도구 쓰는 RAG 에이전트 만들기

> 지금까지 배운 조각(검색 도구·계산 도구·create_agent·메모리)을 하나로 모아, 스스로 도구를 골라 쓰는 RAG 에이전트를 완성합니다. 외부 의존 없이 이 한 파일로 동작합니다.

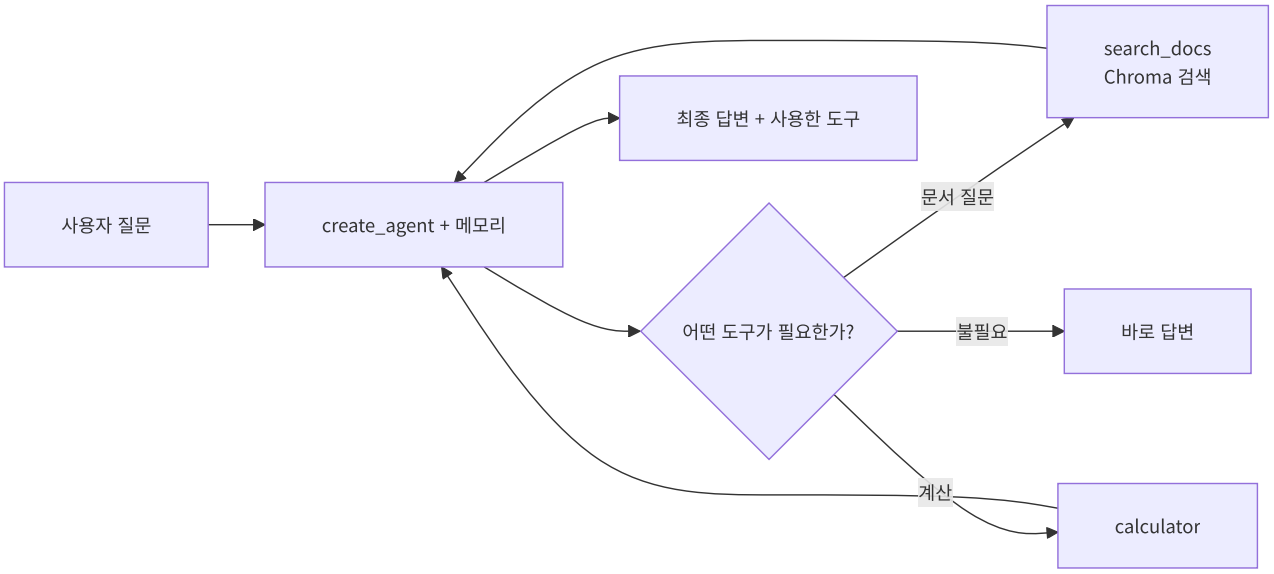
앞 여섯 챕터에서 도구 호출의 원리, 커스텀 도구, ReAct 에이전트, Retriever 도구, Agentic RAG 루프, 대화 기억을 차례로 익혔습니다. 이제 이들을 **하나의 자기완결 프로그램**으로 통합합니다. 작은 RAG를 직접 구성해 검색을 도구로 달고, 계산 도구를 더하고, 메모리를 붙인 에이전트를 만들어 콘솔에서 멀티턴으로 대화합니다. 별도의 웹 서버 없이 이 파일 하나로 돌아가며, 다음 챕터에서 이 에이전트를 그대로 웹 데모로 감쌉니다.

1. 무엇을 만드는가

질문에 따라 에이전트가 도구를 **스스로 골라** 움직이는 콘솔 챗봇입니다.

```
Q> NVMe가 뭐야?              → search_docs 호출 → 문서 근거로 답변
Q> 550 * 6 은?               → calculator 호출 → 3300
Q> 방금 그게 SATA보다 빨라?   → (메모리로 맥락 유지) search_docs 재호출
Q> 고마워                     → 도구 없이 바로 답변
```

흐름을 그림으로 보면 다음과 같습니다.



2. 작은 RAG를 직접 구성

먼저 검색 대상이 될 작은 지식베이스를 벡터 스토어로 만듭니다. 문서를 청킹하고 임베딩해 Chroma에 넣는, 앞서 배운 RAG 인제스션 그대로입니다.

```
# 최종 프로젝트 – 자기완결 RAG 에이전트
import os
from dotenv import load_dotenv
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_chroma import Chroma
from langchain_core.documents import Document
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.tools import tool
from langchain.agents import create_agent
from langgraph.checkpoint.memory import MemorySaver

load_dotenv()

DOCS = [
    "NVMe는 PCIe 버스를 사용하는 SSD 인터페이스로, SATA보다 훨씬 빠르다.",
    "SATA SSD의 최대 대역폭은 약 550MB/s로, NVMe(수 GB/s)보다 낮다.",
    "RAID 0은 속도를 위한 스트라이핑, RAID 1은 안정성을 위한 미러링 구성이다.",
]
splits = RecursiveCharacterTextSplitter(chunk_size=300, chunk_overlap=50).split_documents(
    [Document(page_content=d) for d in DOCS])
vectorstore = Chroma.from_documents(
    splits, OpenAIEmbeddings(model="text-embedding-3-small"), collection_name="agent_final
```

3. 검색과 계산을 도구로

벡터 검색을 `@tool`로 감싸 에이전트의 검색 도구로 만들고(Retriever를 도구로 — 05장), 계산기 도구를 더합니다(커스텀 도구 — 03장). docstring에 "무엇을 하는 도구인지"를 또렷이 적는 것이 핵심입니다.

```python
# 최종 프로젝트 – 도구 정의
@tool
def search_docs(query: str) -> str:
    """기술 문서에서 질문과 관련된 내용을 검색한다 (SSD, RAID 등 저장장치 주제)."""
    hits = vectorstore.similarity_search(query, k=3)
    return "\n".join(f"- {d.page_content}" for d in hits) or "관련 문서를 찾을 수 없습니다."


@tool
def calculator(expression: str) -> str:
    """수학 식을 계산한다. 예: '550 * 6'."""
    try:
        return str(eval(expression, {"__builtins__": {}}, {}))
    except Exception as e:
        return f"계산 오류: {e}"


TOOLS = [search_docs, calculator]
```

4. 메모리를 가진 에이전트 구성

`create_agent`로 도구를 묶고(첫 에이전트 — 04장), `MemorySaver`로 대화를 기억하게 합니다(대화 기억 — 07장). system_prompt로 도구 사용 지침을 줍니다.

```python
# 최종 프로젝트 – 에이전트 구성
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
agent = create_agent(
    llm,
    TOOLS,
    system_prompt=(
        "너는 문서 기반 도우미다. 저장장치 관련 질문은 search_docs 로 근거를 찾아 답하고, "
        "계산이 필요하면 calculator 를 써라. 문서에 없는 내용은 추측하지 말고 모른다고 답하라."
    ),
    checkpointer=MemorySaver(),
)
```

5. 콘솔 멀티턴 루프

마지막으로 사용자 입력을 받아 에이전트를 호출하는 루프를 만듭니다. `thread_id` 하나를 고정해 한 세션의 대화 맥락을 잇고, 어떤 도구를 썼는지도 함께 보여 줍니다.

```
# 최종 프로젝트 – 실행 루프
def main():
    config = {"configurable": {"thread_id": "console"}}
    print("질문을 입력하세요 (종료: quit)")
    while True:
        q = input("\nQ> ").strip()
        if q in {"quit", "exit"}:
            break
        result = agent.invoke({"messages": [("user", q)]}, config=config)
        used = [c["name"]
                for m in result["messages"]
                for c in (getattr(m, "tool_calls", []) or [])]
        print(f"A> {result['messages'][-1].content}")
        if used:
            print(f"   (사용한 도구: {', '.join(used)})")


if __name__ == "__main__":
    main()
```

같은 챗봇이 질문에 따라 다르게 동작합니다. 문서 질문에는 `search_docs`를, 계산에는 `calculator`를 부르고, 인사에는 도구 없이 답합니다. 이 선택을 전부 에이전트가 스스로 판단하며, 메모리 덕분에 "방금 그거"처럼 앞선 대화를 가리키는 후속 질문도 이어집니다.

6. 실행과 기능 체크리스트

```
pip install langchain langchain-openai langchain-chroma langchain-text-splitters langgraph
# .env 에 OPENAI_API_KEY=sk-... 를 설정 (코드에는 키를 적지 않는다)
python final_agent.py
```

기능	출처 챕터	구현
작은 RAG 구성(청킹·임베딩·Chroma)	RAG 파트	`Chroma.from_documents`
검색을 도구로 (`search_docs`)	05	`@tool` + `similarity_search`
추가 도구 (`calculator`)	03	`@tool`
에이전트 구성 (`create_agent`)	04	ReAct 루프 자동
대화 기억 (`thread_id`)	07	`MemorySaver`
검색/계산 자동 선택	06	LLM이 흐름 판단

7. 다음 단계: 웹으로 감싸기

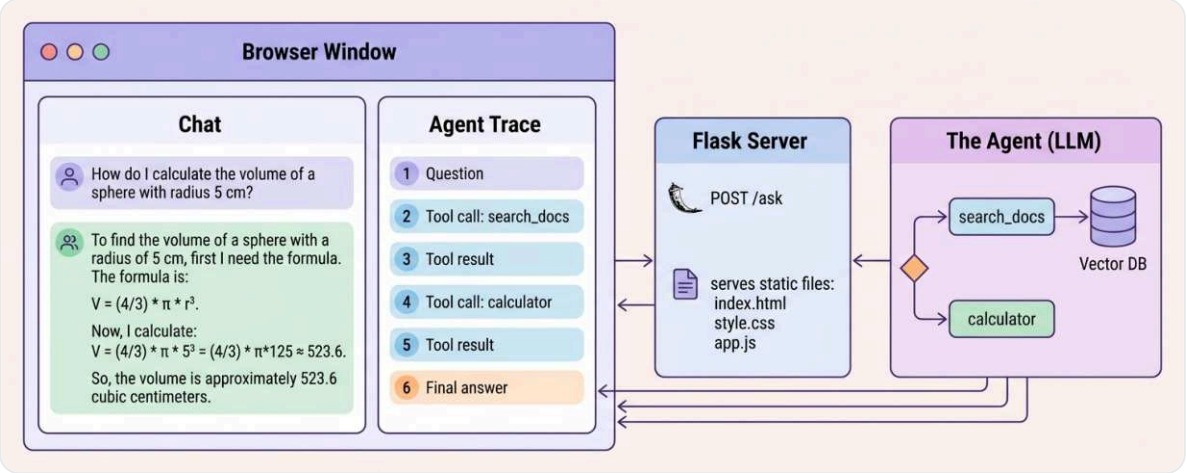
이 에이전트의 핵심은 `agent.invoke(...)` 한 줄입니다. 콘솔 입력 대신 웹 요청으로 이 한 줄을 호출하면 그대로 웹 서비스가 됩니다. 사용자/세션마다 `thread_id`를 부여하면 멀티 사용자 대화 기억까지 자연스럽게 얻습니다. 다음 챕터에서는 바로 이 에이전트를 작은 Flask 웹앱으로 감싸, 에이전트가 어떤 판단으로 어떤 도구를 불렀는지를 화면으로 들여다봅니다.

> **핵심 포인트:** 도구 쓰는 RAG 에이전트는 새로운 개념이 아니라 **조립**입니다. 작은 RAG를 만들어 검색을 도구로 달고, 계산 도구를 더하고, `create_agent`로 묶고, `MemorySaver`로 기억을 붙이면, "항상 검색하는 RAG"가 "필요할 때 검색하고 계산도 하며 대화를 잇는 에이전트"가 됩니다. 핵심 호출은 `agent.invoke(...)` 한 줄이므로, 이를 그대로 웹으로 옮기면 통합 데모가 됩니다.

최종 프로젝트: 통합 웹 데모로 에이전트 들여다보기

> 앞에서 만든 에이전트를 웹으로 감쌉니다. 질문을 던지면 오른쪽 패널에 "에이전트가 어떤 판단으로 어떤 도구를 불렀고, 무엇을 관찰해 어떻게 답했는지"가 단계별로 펼쳐집니다.

CLI 에이전트는 동작은 같지만 내부에서 무슨 일이 일어나는지 눈에 잘 들어오지 않습니다. 이번 마지막 프로젝트에서는 같은 에이전트를 작은 Flask 웹앱으로 감싸, **에이전트의 사고 과정(trace)을 화면 오른쪽에 시각화**합니다. 사용자는 왼쪽에서 질문하고, 오른쪽에서 "질문 → 판단·도구 호출 → 도구 결과 → 최종 답변"의 흐름을 그대로 봅니다.



통합 웹 데모 구조: 브라우저(좌측 질문/답변 · 우측 트레이스 패널) ↔ Flask /ask ↔ 에이전트가 search_docs·calculator 중 도구를 선택, 실행 단계를 trace 로 반환

1. 무엇을 보여 주는가

```
┌──────────────────────────────┬──────────────────────────────┐
│  RAG 에이전트                │  에이전트 트레이스           │
│  ┌────────────────────┐      │  질문    NVMe와 SATA 속도 더하면?│
│  │ Q: NVMe와 SATA 속도.. │   │  판단·도구 호출  search_docs   │
│  │ A: 3500+550=4050MB/s │   │  판단·도구 호출  search_docs   │
│  └────────────────────┘      │  도구 결과  3500MB/s · 550MB/s │
│  [ 질문 입력...    ] [전송] │  판단·도구 호출  calculator    │
│                              │  도구 결과  4050             │
│                              │  최종 답변  4050MB/s 입니다     │
└──────────────────────────────┴──────────────────────────────┘
```

오른쪽 패널이 핵심입니다. 에이전트가 **검색이 필요하다고 판단해** `search_docs` **를 불렀는지, 계산이라** `calculator` **를 골랐는지**, 그리고 두 도구를 어떻게 이어 썼는지가 매 질문마다 그대로 드러납니다. "블랙박스"였던 판단 과정이 눈에 보이는 것이 이 데모의 목적입니다.

2. 에이전트와 지식: 숫자를 담아 도구를 잇게 한다

백엔드의 에이전트 구성은 앞 챕터와 똑같습니다(`search_docs` · `calculator` + `create_agent` + `MemorySaver`). 다만 문서에 **수치**를 담아, 검색과 계산을 이어 쓰는 질문이 가능하게 합니다.

```
# 09_web_demo — 검색+계산 조합을 유도하는 지식베이스
DOCS = [
    "NVMe SSD의 순차 읽기 속도는 약 3500MB/s이다.",
    "SATA SSD의 순차 읽기 속도는 약 550MB/s이다.",
    "이 서버에는 NVMe SSD가 4개 장착되어 있다.",
    "RAID 0은 속도를 위한 스트라이핑, RAID 1은 안정성을 위한 미러링 구성이다.",
]
```

이렇게 두면 "NVMe와 SATA 속도를 더하면?"이나 "NVMe 속도에 장착된 개수를 곱하면?" 같은 질문에서, 에이전트가 **먼저** `search_docs` **로 숫자를 찾고** → `calculator` **로 계산**하는 흐름을 트레이스로 확인할 수 있습니다.

3. 핵심: 실행 과정을 trace로 풀어내기

에이전트를 호출하면 `result["messages"]` 에 전체 흐름(질문 · 도구 호출 · 도구 결과 · 최종 답변)이 순서대로 담깁니다. 이를 화면이 그릴 수 있는 단계 목록으로 변환하는 것이 이 프로젝트의 새 코드입니다.

```
# 09_web_demo — 메시지를 화면용 trace 로 변환
def build_trace(messages):
    trace = []
    for m in messages:
        t = getattr(m, "type", "")
        if t == "human":
            trace.append({"step": "질문", "text": m.content})
        elif t == "ai":
            for c in (getattr(m, "tool_calls", []) or []):
                trace.append({"step": "판단·도구 호출", "tool": c["name"], "args": c["args"
            if m.content:
                trace.append({"step": "최종 답변", "text": m.content})
        elif t == "tool":
            trace.append({"step": "도구 결과", "tool": getattr(m, "name", ""), "text": m.c
    return trace
```

`/ask` 는 질문을 에이전트에 넣고, 최종 답변과 trace를 JSON으로 돌려줍니다. 세션마다 `thread_id` 를 부여해 멀티턴 맥락도 유지합니다.

```
# 09_web_demo — 질의 엔드포인트
@app.route("/ask", methods=["POST"])
def ask():
    question = (request.json or {}).get("question", "").strip()
    if not question:
        return jsonify({"error": "질문을 입력하세요"}), 400
    if "tid" not in session:
        session["tid"] = str(uuid.uuid4())
    result = agent.invoke(
        {"messages": [("user", question)]},
        config={"configurable": {"thread_id": session["tid"]}},
    )
    return jsonify({"answer": result["messages"][-1].content,
                    "trace": build_trace(result["messages"])})
```

4. 화면은 static으로 분리한다 (템플릿 엔진 없이)

HTML을 파이썬 문자열에 박아 넣지 않고, **정적 파일로 분리**합니다. 템플릿 엔진(Jinja 등)도 쓰지 않습니다. 백엔드는 `/` 에서 `index.html` 만 그대로 내려보내고, 나머지(`style.css` · `app.js`)는 Flask가 `/static/` 으로 자동 서빙합니다.

```
09_web_demo_실습.py        # 백엔드: 에이전트 + /ask + '/' 서빙
static/
├─ index.html              # 화면 구조 (좌: 대화 / 우: 트레이스)
├─ style.css               # 디자인 (분리 — 나중에 교체)
└─ app.js                  # 질문 전송 + 트레이스 렌더
```

```
# 09_web_demo — 정적 파일 서빙 (send_from_directory, 템플릿 엔진 미사용)
from flask import send_from_directory

@app.route("/")
def index():
    return send_from_directory(app.static_folder, "index.html")
```

`index.html` 은 평범한 정적 HTML이고, `app.js` 를 불러옵니다.

```
<!-- static/index.html (발췌) -->
<main class="app">
  <section class="chat">
    <div id="log" class="log"></div>
    <form id="f" class="ask"><input id="q"><button>전송</button></form>
  </section>
  <aside class="trace"><h2>에이전트 트레이스</h2><div id="trace" class="steps"></div></asid
</main>
<link rel="stylesheet" href="/static/style.css">
<script src="/static/app.js"></script>
```

`app.js` 는 질문을 `/ask` 로 보내고, 답변과 트레이스를 그립니다. 백엔드 호출 부분이 핵심입니다.

```
// static/app.js — 백엔드 /ask 호출 (실습에서 직접 채우는 부분)
async function ask(question) {
  const res = await fetch('/ask', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ question }),
  });
  return await res.json();
}
```

받은 `trace` 는 단계 카드로 렌더합니다(도구 단계는 색을 달리 표시).

```
// static/app.js — 트레이스 렌더 (요지)
tr.innerHTML = data.trace.map(s => `
  <div class="step ${s.step.includes('도구') ? 'tool' : ''}">
    <span class="k">${s.step}</span>${s.tool ? ` <code>${s.tool}</code>` : ''}
    <div class="v">${s.text ? s.text : JSON.stringify(s.args || {})}</div>
  </div>`).join('');
```

5. 디자인은 분리한다 (나중에 바이브코딩)

`style.css` 는 일부러 **최소 레이아웃만** 둡니다. HTML 구조(`.app` , `.chat` , `.log` , `.msg` , `.trace` , `.step`)는 그대로 두고 이 파일만 갈아끼우면 디자인이 바뀝니다.

```
/* static/style.css — 최소 스타일. 이 파일만 교체하면 디자인이 바뀐다. */
.app { display: flex; height: 100vh; }
.chat { flex: 2; display: flex; flex-direction: column; padding: 16px; border-right: 1px s
.trace { flex: 1; padding: 16px; background: #fafafa; overflow: auto; }
.step { margin: 8px 0; padding: 8px 10px; border-left: 3px solid #bbb; background: #fff; }
.step.tool { border-left-color: #6c5ce7; }
```

> 디자인을 입히고 싶다면, ChatGPT 같은 도구에 *"이 HTML 구조(`.app` / `.chat` / `.log` / `.msg` / `.trace` / `.step`) 에 어울리는 모던한 채팅 UI CSS를 만들어 줘"* 라고 요청해 `style.css` 만 통째로 바꾸면 됩니다. 백엔드·HTML·JS는 손대지 않아도 됩니다.

6. 멀티툴 질문: 검색과 계산을 잇는다

실제로 "NVMe와 SATA 속도를 더하면?"이라고 물으면, 트레이스에 다음이 그대로 찍힙니다.

```
{
  "answer": "NVMe(3500MB/s)와 SATA(550MB/s)를 더하면 4050MB/s입니다.",
  "trace": [
    {"step": "질문", "text": "NVMe와 SATA 속도를 더하면?"},
    {"step": "판단·도구 호출", "tool": "search_docs", "args": {"query": "NVMe 속도"}},
    {"step": "판단·도구 호출", "tool": "search_docs", "args": {"query": "SATA 속도"}},
    {"step": "도구 결과", "tool": "search_docs", "text": "- NVMe ... 3500MB/s ... / - SATA
    {"step": "판단·도구 호출", "tool": "calculator", "args": {"expression": "3500 + 550"}},
    {"step": "도구 결과", "tool": "calculator", "text": "4050"},
    {"step": "최종 답변", "text": "... 더하면 4050MB/s입니다."}
  ]
}
```

이어서 "그럼 NVMe 속도에 장착된 개수를 곱하면?"이라고 물으면, 메모리로 앞 맥락(3500MB/s)을 유지한 채 `calculator("3500 * 4") → 14000` 을 호출합니다. 검색·계산·기억이 한 화면에서 눈에 보입니다.

7. 실행

```
pip install langchain langchain-openai langchain-chroma langchain-text-splitters langgraph
# .env 에 OPENAI_API_KEY=sk-... 설정
python 09_web_demo_실습.py
# → 브라우저에서 http://localhost:5001 접속, 질문을 던지며 오른쪽 트레이스를 관찰
```

> **핵심 포인트:** 에이전트 로직은 그대로 두고, `result["messages"]` 를 `build_trace` 로 풀어내 화면에 흘리면 "에이전트가 왜 그렇게 답했는지"가 눈에 보입니다. 화면은 `static/` (index.html·style.css·app.js)으로 분리하고 `send_from_directory` 로 서빙해, 백엔드·디자인·동작을 깔끔히 나눴습니다. 디자인(CSS)만 따로 발전시킬 수 있습니다. 이것으로 손코딩 RAG에서 시작해, 스스로 도구를 골라 쓰고 그 과정을 보여 주는 에이전트 웹 서비스까지 완성했습니다.

여기서 출발해 도구를 늘리고(웹 검색·날씨·사내 API), 멀티 에이전트(multi-agent)로 분업시키고, 영속 메모리와 가드레일(guardrail)을 더하면 실무 수준의 AI 에이전트 서비스로 발전시킬 수 있습니다.

부록 · 학습 성취 체크리스트

> 과정 종료 후, 수강생은 아래 항목을 직접 구현할 수 있습니다.

역량 항목	구현	관련 실습 (저장소 경로)
OpenAI API 사용 (Chat Completion)	O	`1.openai/1.intro`
CLI / 멀티턴 챗봇	O	`1.openai/2.chatbot_ui`, `3.chatbot2_history`
Embedding 생성	O	`1.openai/7.rag/3.rag_local_embedding.py`
Vector Search (cosine, Top-K) 직접 구현	O	`1.openai/7.rag/4.rag_cosine_similarity.py`
ChromaDB 사용	O	`2.langchain/7.RAG/3.vectorstore`
LangChain RAG (LCEL)	O	`2.langchain/7.RAG/4.rag_chain`
PDF / TXT / DOCX QA	O	`2.langchain/7.RAG/2.loaders`
도구 호출 (bind_tools) / @tool 커스텀 도구	O	`2.langchain/8.agents/2.custom_tools`, `4.internals/4.1_bind_tools.py`
ReAct 에이전트 (create_agent)	O	`2.langchain/8.agents/2.custom_tools/2.1_first_agent.py`
Retriever Tool / Agentic RAG	O	`2.langchain/8.agents/3.applied_agents/3.3_retriever_tool.py`, `7.RAG/6.agentic`
대화를 기억하는 에이전트 (LangGraph 메모리)	O	`2.langchain/8.agents/5.langgraph_memory`
도구 쓰는 RAG 에이전트 (최종 프로젝트)	O	`2.langchain/8.agents/3.applied_agents/3.3_retriever_tool.py`, `5.langgraph_memory`

실습 환경 요약

```
# Python 3.10+ 권장. 가상환경 후 설치
pip install openai langchain langchain-openai langchain-community \
            langchain-chroma chromadb pypdf docx2txt flask python-dotenv \
            sentence-transformers numpy
# 에이전트 파트까지 실습하려면 추가로:
pip install langgraph

# .env 에 키 설정 (코드에 직접 적지 말 것)
# OPENAI_API_KEY=sk-...
```

> **보안 주의:** API 키와 개인정보(이름·이메일 등)는 코드·로그·외부 요청에 직접 포함하지 말고 반드시 `.env` 환경 변수로 관리합니다.

수고하셨습니다

3일 동안 LLM과 RAG의 원리에서 시작해 LangChain으로 RAG를 표준화하고, 마지막에는 스스로 도구를 골라 쓰는 에이전트까지 만들어 보았습니다. 이 미니북이 문서 기반 AI와 에이전트를 직접 설계하고 구현하는 든든한 출발점이 되기를 바랍니다.

교육 · 강의 · 강연 문의

생성형 AI · LLM · RAG · 로컬 모델 · 클라우드 · 도커/쿠버네티스 · 보안

20년 이상의 풀스택 개발 경험과 글로벌 기업의 개발 문화·프로세스를 바탕으로,
입문부터 실무 적용까지 수준에 맞춘 교육을 제공합니다.

contact@genailabs.kr

젠아이랩스 (GenAI Labs)