생성형 AI 기반 RAG 개발자 과정

Vector DB 이해 + FAISS

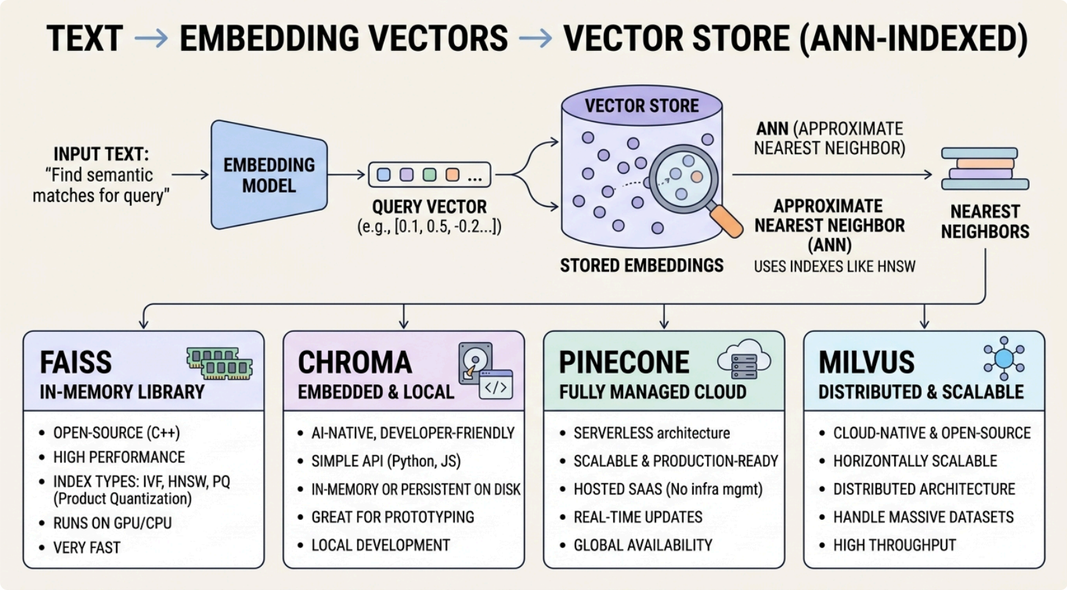
Session 13 / 33 — Day 2

FAISS · Chroma · Pinecone · Milvus · ANN · HNSW

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

FAISS · Chroma · Pinecone · Milvus



벡터 DB 비교 · FAISS(인메모리)·Chroma(임베디드 영속)·Pinecone(매니지드)·Milvus(분산) · 임베딩 저장 + ANN 검색

이번 시간을 마치면

원리 (이론)

- ✅ **벡터 DB 의 정의와 필요성** — 임베딩 저장 + ANN 검색 전용 저장소
- ✅ **ANN/인덱스 구조** — HNSW · IVF 가 근사 최근접을 빠르게 해결하는 원리
- ✅ **FAISS** 가 brute-force 보다 1만 배 빠른 근거 — ANN 알고리즘
- ✅ 인덱스 3종 — **Flat · IVF · HNSW** 의 정확도/속도/메모리 트레이드오프
- ✅ 청킹의 의의와 sweet spot (chunk_size · overlap)

> 💡 임베딩(텍스트→벡터·코사인 유사도)은 **Day 1(s2·s7)에서 다뤘다**. 이번 회차의 주제는 그 벡터를 **저장·검색하는 DB 구조**다.

실습 (코드)

- ✅ OpenAI Embedding API 직접 호출 + numpy 유사도 계산
- ✅ FAISS 첫 인덱스 — 1만 문서 색인 + 검색 (1ms 이하)
- ✅ LangChain `FAISS.from_documents()` 단일 호출 추상화
- ✅ 청킹 4종 — Character / Recursive / Token-aware / Semantic
- ✅ 미니 RAG — 문서 → 임베딩 → 검색 → LLM 답변

> 📂 소스: `2.langchain/7.RAG/1.basics/`, `7.RAG/3.vectorstore/`, `1.openai/7.rag/1.faiss/`

PART A

PART A
벡터 DB — 무엇·왜

> 임베딩을 저장하고 가장 가까운 이웃을 찾는 전용 저장소 — 약 15분
>
> 🔗 임베딩(텍스트→벡터, 코사인 유사도)의 원리는 **Day 1 s2·s7** 에서 다뤘다. 여기서는 그 벡터를 **저장·검색하는 DB 구조**에 집중한다.

키워드 검색만으로 부족한 이유

❌ 키워드 매칭의 한계 4가지

사례	키워드 검색 결과
"차" 검색	자동차·홍차·차이·바둑돌 모두 매치
"노트북 추천"	"추천 노트북" 못 찾음 (단어 순서)
"심장마비"	"myocardial infarction" 못 찾음 (언어·동의어)
"AI 가 위험한가"	"인공지능의 잠재적 리스크" 못 찾음

본질

- 키워드 검색은 **문자열 일치**만 판별
- **의미·맥락·동의어**를 인식하지 못함
- 사용자는 동일 의미를 다양한 표현으로 질의

✅ 의미 기반 검색의 동작

```
사용자 질의: "심장마비 응급처치"
                      ↓
                      ↓ 임베딩
                      ↓
          [0.21, -0.83, 0.15, ..., 0.07]
                      ↓
            벡터 공간에서 가까운 문서 찾기
                      ↓
검색됨: "Cardiac arrest first aid" (영문)
검색됨: "심근경색 응급조치 매뉴얼" (동의어)
검색됨: "Heart attack emergency response"
```

이를 가능하게 하는 핵심 원리

> "**의미가 유사한 텍스트는 벡터 공간에서 가깝게 위치한다**"

Text → Vector (Day 1) → 어디에 저장·검색? (오늘)

복습 (Day 1 s2·s7) — 요약

```
"강아지" → [0.21, -0.83, 0.15, ...]
"개"     → [0.20, -0.81, 0.16, ...]   ← 가까움
"자동차" → [-0.55, 0.32, -0.78, ...]  ← 멀리
```

- 임베딩 = 의미를 N차원 좌표로 표현 (1536d / 3072d)
- 의미가 유사할수록 **코사인 유사도**가 높다
- **이 원리는 Day 1 에서 다뤘다** → 재설명 생략

핵심 질문 — 그 벡터를 어디에 두나

```
청크 수천~수백만 개 → 벡터 수천~수백만 개
질문도 벡터 1개

검색 = 질문 벡터와 가장 가까운 문서 벡터 탐색
```

- 직접 구현하면: 매 질문마다 **전체 벡터와 전수 계산** → 수백만 개 규모에서는 비현실적
- 그래서 **벡터 DB** 가 필요하다 — 다음 슬라이드

> 🔗 임베딩 모델·차원·코사인 수식·Contrastive 학습은 **Day 1 의 범위**다. 이번 회차는 "그 벡터를 저장·검색하는 구조"를 다룬다.

벡터 DB = 임베딩 전용 저장소 + 근사 검색 색인

두 가지 핵심 역할

1. **저장** — 벡터 + 원본 텍스트 + 메타데이터를 함께 보관
2. **검색** — "가장 가까운 N개"를 효율적으로 찾는 **색인(index)** 제공

```
관계형 DB : 행 + B-tree 인덱스 → 정확 일치 빠르게
벡터  DB  : 벡터 + ANN 색인     → 의미 근접 빠르게
```

전용 저장소가 필요한 이유

- 의미 검색은 "정확히 같은 값"이 아니라 **"가장 가까운 값"**을 찾는 문제
- B-tree·해시 인덱스로는 해결 불가 → **ANN 색인**이 필요

ANN — 근사 최근접 이웃 (Approximate NN)

- **완전 탐색(brute-force)**: 모든 벡터와 거리 계산 → 데이터 증가 시 속도 저하
- **ANN**: 정확도를 소폭 양보하는 대신 **검색 공간을 효율적으로 축소**해 수백만 벡터에서도 밀리초 단위로 "거의 최근접" 이웃을 탐색

색인	한 줄 원리
HNSW	계층적 그래프 탐색
IVF	클러스터로 나눈 뒤 일부만 탐색
PQ	벡터를 압축해 메모리 절약

> 🎯 정확도 ↔ 속도는 **맞교환** 관계다. 대규모에서는 ANN 이 사실상 필수이며, 구조는 PART C 에서 상세히 다룬다.

FAISS · Chroma · Pinecone · Milvus

항목	Chroma	FAISS	Pinecone	Milvus
형태	임베디드(로컬)	라이브러리(인메모리)	매니지드(클라우드)	자체 호스팅 서버
영속화	`persist_directory` 자동	`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

본 코스의 선택

- **Day 2 학습·실습**: FAISS 로 ANN·인덱스 **구조를 직접 체감** → 다음 세션에서 Chroma 로 영속화·필터
- **일반 실무 RAG**: 설치 한 줄, 메타데이터 필터를 네이티브 지원하는 **Chroma**
- **수억 벡터·GPU 가속**: **FAISS**, **매니지드 SaaS**: Pinecone, **대규모 자체 운영**: Milvus

> 💡 FAISS 는 순수 **인메모리 검색 라이브러리**로 속도가 가장 빠르지만 영속화·메타 필터는 직접 구현해야 한다. 그 공백을 메우는 것이 Session 14 의 Chroma 다.

PART B

PART B
Brute-force 검색의 한계 체감

> 벡터를 numpy 로 직접 검색하며 ANN 색인이 필요한 이유를 실측으로 확인한다 — 약 15분

`client.embeddings.create()` — REST 호출과 동일

호출 코드

```
from openai import OpenAI

client = OpenAI()

# 1개 문장
resp = client.embeddings.create(
    model="text-embedding-3-small",
    input="강아지가 공원에서 뛰논다",
)
vec = resp.data[0].embedding
print(f"차원: {len(vec)}")        # 1536
print(f"앞 5개: {vec[:5]}")        # [0.012, -0.045, ...]
print(f"norm: {sum(x*x for x in vec)**0.5:.4f}")
# → 1.0000 (이미 unit-norm)

# 여러 문장 배치 (8192 토큰 한도까지)
resp = client.embeddings.create(
    model="text-embedding-3-small",
    input=["강아지", "고양이", "자동차", "비행기"],
)
vectors = [d.embedding for d in resp.data]
```

가격 (2026 기준)

모델	차원	1M 토큰	적합
3-small	1536	**$0.02**	가성비 — 본 코스 기본
3-large	3072	$0.13	정밀 RAG
ada-002	1536	$0.10	레거시

> 💡 **dimensions 파라미터**: `dimensions=512` 로 Matryoshka 잘라 쓰기 — 정확도 ~5% 손실로 저장 1/3, 검색 3배 빠름.

코사인 유사도 직접 구현

두 함수 — embed + cosine_sim

```
import numpy as np
from openai import OpenAI

client = OpenAI()

def embed(texts: list[str]) → np.ndarray:
    """문장 리스트를 (N, 1536) 행렬로."""
    resp = client.embeddings.create(
        model="text-embedding-3-small",
        input=texts,
    )
    return np.array(
        [d.embedding for d in resp.data])
    # shape (N, 1536)

def cosine_sim(a, b) → float:
    """단위 벡터끼리는 dot product = cosine."""
    return float(
        np.dot(a, b) /
        (np.linalg.norm(a) * np.linalg.norm(b)))
```

실험 + 결과

```
texts = [
    "강아지가 공원에서 뛰논다",
    "개가 산책 중이다",             # 의미 매우 가까움
    "Dog playing in the park",   # 영문 같은 의미
    "주식 시장이 폭락했다",          # 무관
    "강아지는 정말 정말 정말 귀엽다",# 같은 주제
]
M = embed(texts)
q = M[0]
for i, t in enumerate(texts):
    print(f"{cosine_sim(q, M[i]):.3f}  {t}")
```

```
1.000  강아지가 공원에서 뛰논다
0.842  개가 산책 중이다
0.781  Dog playing in the park
0.123  주식 시장이 폭락했다
0.512  강아지는 정말 정말 정말 귀엽다
```

> 🎯 **관찰 포인트**: ① 다국어 → 의미 ② 동의어 ③ 무관한 주제 = 낮은 점수

for 루프 vs numpy 행렬 연산

```
import numpy as np, time

N = 10_000
D = 1536
db = np.random.randn(N, D).astype(np.float32)
db /= np.linalg.norm(db, axis=1, keepdims=True)   # 정규화
q = np.random.randn(D).astype(np.float32);  q /= np.linalg.norm(q)

# ❌ 1만 개 for 루프
t = time.time()
scores = [np.dot(q, db[i]) for i in range(N)]
print(f"for 루프:   {(time.time()-t)*1000:.1f} ms")   # ~80ms

# ✅ 행렬 연산 1줄
t = time.time()
scores = db @ q                                  # shape (N,)
print(f"행렬 연산:  {(time.time()-t)*1000:.1f} ms")   # ~0.5ms
```

Top-K 추출 — brute-force 가 한계에 이르는 지점

Top-K 검색

```
k = 5
top_k_idx = np.argpartition(-scores, k)[:k]      # 부분 정렬 (O(N))
top_k_idx = top_k_idx[np.argsort(-scores[top_k_idx])]   # 정확 정렬 (O(k log k))
print(f"Top-{k} 인덱스:", top_k_idx)
```

문제 — N=1억 일 때

```
1억 × 1536 차원 × float32 = 약 600GB 메모리
단순 행렬 곱: 약 100초 (싱글 코어)
```

→ **FAISS 가 해결하는 문제.**

PART C

PART C
FAISS — Facebook AI Similarity Search

> 1억 벡터 검색을 1ms 로 — ANN 알고리즘 — 약 25분

N 증가에 따른 brute-force 의 병목

Brute-force 확장성

N (문서 수)	검색 시간	메모리 (1536d × float32)
1,000	< 1ms	6MB
10,000	~ 5ms	60MB
100,000	~ 50ms	600MB
1,000,000	~ 500ms	6GB
10,000,000	~ 5초	60GB
100,000,000	~ 50초	600GB

복잡도

- 검색: **O(N · D)** — 모든 벡터와 q 의 내적
- 메모리: **O(N · D · 4 bytes)** (float32)

ANN (Approximate Nearest Neighbor)

측면	Brute-force	ANN (FAISS)
정확도	100%	95~99% (튜닝)
검색 시간	O(N)	**O(log N)** / O(√N)
메모리	풀 벡터	압축 가능 (PQ 등)
색인 시간	0	색인 구축 비용 있음

트레이드오프

> "정확도를 소폭 양보하고 **속도·메모리 1000배**를 확보한다."

- 1억 벡터 → brute 50초, FAISS 1~10ms
- 메모리도 PQ 압축으로 1/16
- 운영 환경에서는 사실상 필수

인덱스 타입 — 메모리·속도·정확도 트레이드오프

인덱스	원리	정확도	검색 속도	메모리	적합
IndexFlat	Brute-force (정답 보장)	100%	O(N)	풀 벡터	< 10만 벡터
IndexIVF	Inverted File — 클러스터 후 일부만 검색	95~99%	O(√N)	풀 벡터 + 클러스터 ID	10만 ~ 1억
IndexHNSW	Hierarchical NSW Graph	97~99%	**O(log N)**	풀 벡터 + 그래프	1억 ~ 10억
IndexPQ	Product Quantization (벡터 압축)	90~95%	O(N) (빠른 곱)	**풀의 1/16**	메모리 절약
IndexIVFPQ	IVF + PQ 조합	90~98%	O(√N)	1/16	운영 RAG

본 코스에서 사용하는 것

- **Day 2 학습용**: `IndexFlat` (정답 보장, 작은 데이터)
- **운영 시나리오**: `IndexHNSW` (가장 균형)
- **메모리 제약 환경**: `IndexIVFPQ` (예: 라즈베리파이)

Small World 그래프 — 6단계 분리 원리

핵심 아이디어

- 모든 벡터를 **그래프 노드**로 구성
- 인접 벡터끼리 **엣지**로 연결
- 검색은 그래프를 **greedy traversal** — 더 가까운 이웃으로 이동

다층 구조 (Hierarchical)

- **Layer N (상위)**: 희소 연결, 장거리 이동 (지하철 노선)
- **Layer 0 (하위)**: 조밀 연결, 미세 탐색 (마을 골목)
- 검색은 상위 계층에서 시작해 점차 하위 계층으로 진행

시간 복잡도

- 색인: **O(N · log N · M)** — M = 노드당 엣지 수
- 검색: **O(log N · ef_search)** — ef_search = 후보 풀 크기

튜닝 파라미터

파라미터	의미	트레이드오프
`M` (default 16)	노드당 엣지 수	↑ = 정확도 ↑, 메모리 ↑
`ef_construction` (200)	색인 시 후보 수	↑ = 정확도 ↑, 색인 시간 ↑
`ef_search` (50)	검색 시 후보 수	↑ = 정확도 ↑, 검색 시간 ↑

최소 구성 코드 — IndexFlatIP

1~2. 데이터 준비 + 색인

```
import faiss, numpy as np
from openai import OpenAI

client = OpenAI()

# 1. 데이터 준비
docs = [
    "강아지가 공원에서 뛰논다",
    "Dog playing in the park",
    "주식 시장이 폭락했다",
    "Stock market crashed today",
    "강아지는 귀엽다",
]
emb = client.embeddings.create(
    model="text-embedding-3-small", input=docs)
M = np.array([d.embedding for d in emb.data]
             ).astype('float32')
print(M.shape)   # (5, 1536)

# 2. FAISS 인덱스 생성 (내적 = 코사인)
index = faiss.IndexFlatIP(1536)
index.add(M)                              # 1줄로 색인
print(f"색인된 벡터 수: {index.ntotal}")  # 5
```

인덱스 종류

- `IndexFlatIP` : 내적 (정규화된 벡터)
- `IndexFlatL2` : 유클리드 거리 (비정규화)
- 그 외: HNSW, IVF, PQ — 다음 슬라이드

검색 결과 — `D` 와 `I`

3. 검색 코드

```
# 3. 검색
query = client.embeddings.create(
    model="text-embedding-3-small",
    input=["강아지 산책"])
q = np.array([query.data[0].embedding]
             ).astype('float32')

D, I = index.search(q, k=3)           # top-3
for rank, (score, idx) in enumerate(zip(D[0], I[0])):
    print(f"#{rank+1}  score={score:.3f}  {docs[idx]}")
```

출력

```
#1  score=0.832  강아지가 공원에서 뛰논다
#2  score=0.741  Dog playing in the park
#3  score=0.523  강아지는 귀엽다
```

`D` 와 `I`

반환값	의미
`D`	distances — IP 인덱스는 inner product, **높을수록 가까움**
`I`	indices — 원본 배열에서의 위치

1만 벡터 색인 + 검색 벤치마크

Flat vs HNSW 벤치마크 코드

```
import faiss, numpy as np, time

N, D = 10_000, 1536
np.random.seed(42)
data = np.random.randn(N, D).astype('float32')
faiss.normalize_L2(data)         # 단위 벡터로

# === Flat (정답 기준) ===
flat = faiss.IndexFlatIP(D)
t = time.time(); flat.add(data)
add_flat = time.time() - t
q = data[:1]
t = time.time()
D_flat, I_flat = flat.search(q, 10)
s_flat = time.time() - t

# === HNSW ===
hnsw = faiss.IndexHNSWFlat(D, 32)   # M=32
hnsw.hnsw.efConstruction = 200
hnsw.hnsw.efSearch = 50
t = time.time(); hnsw.add(data)
add_hnsw = time.time() - t
t = time.time()
D_h, I_h = hnsw.search(q, 10)
s_hnsw = time.time() - t

# === Recall@10 ===
recall = len(set(I_flat[0]) & set(I_h[0])) / 10
```

측정 결과 + 트레이드오프

```
Flat  add=22ms     search=2.34ms
HNSW  add=1820ms   search=0.18ms
Recall@10 = 0.90
```

트레이드오프 한 줄

- 색인은 **80배 느림** (한 번만)
- **검색은 13배 빠름** (매번)
- 정확도 90%

운영 가이드

N 규모	권장 인덱스
< 10만	`IndexFlatIP`
10만~1000만	`IndexHNSWFlat`
1000만~	`IndexIVFPQ` (메모리 압축)

앞의 30줄을 LangChain 으로 압축

색인 + 검색 (3줄)

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document

# 1. Document 객체 (metadata 포함)
docs = [
    Document(page_content="강아지가 공원에서 뛰논다",
             metadata={"src": "doc1.txt"}),
    Document(page_content="Dog playing in the park",
             metadata={"src": "doc2.txt"}),
    Document(page_content="주식 시장이 폭락했다",
             metadata={"src": "doc3.txt"}),
]

# 2. 임베딩 + FAISS 색인 — 한 줄!
vectorstore = FAISS.from_documents(
    docs,
    OpenAIEmbeddings(model="text-embedding-3-small"))

# 3. 검색 (metadata 함께 반환)
results = vectorstore.similarity_search_with_score(
    "강아지 산책", k=3)
for doc, score in results:
    print(f"score={score:.3f}  "
          f"src={doc.metadata['src']}")
```

저장·로드 (영속화)

```
# 디스크 저장
vectorstore.save_local("./faiss_index")

# 다음번에 로드
vectorstore = FAISS.load_local(
    "./faiss_index",
    OpenAIEmbeddings(),
    allow_dangerous_deserialization=True,
    # FAISS 는 pickle 사용
)
```

LangChain vs raw FAISS

항목	raw FAISS	LangChain
코드 줄 수	~30줄	~3줄
메타데이터	직접 매핑	자동
vectorstore 교체	재작성	한 줄
인덱스 세부 튜닝	직접 가능	한 단계 추상화

FAISS 는 라이브러리 — 다음 단계는 Chroma

FAISS 로 수행한 작업

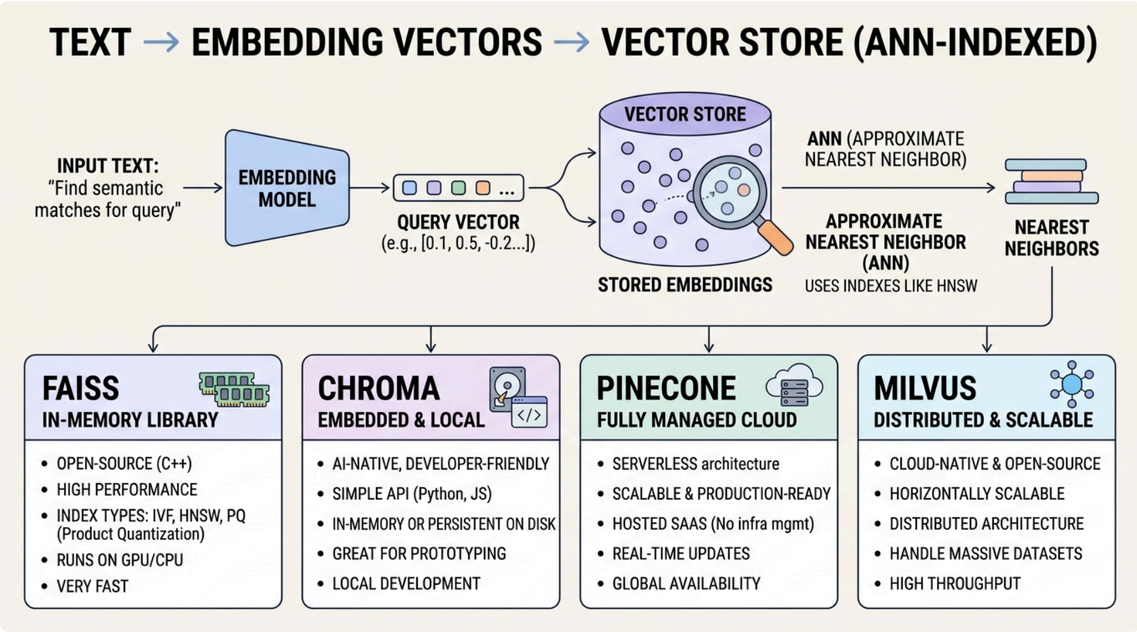
- 임베딩 벡터를 **인메모리 색인**에 적재하고 ANN 검색을 직접 실행 (`IndexFlatIP` → `IndexHNSWFlat`)
- 인덱스 타입·`ef_search`·Recall 트레이드오프를 **직접 측정**

FAISS 가 직접 처리해야 하는 것

- **영속화** — `save_local` 수동 직렬화 (pickle)
- **메타데이터 필터** — 래퍼가 후처리로만 지원 → AND/숫자 비교 부담
- **동시 쓰기·삭제** — 단일 프로세스 mutation

> 💡 4대 DB 성격 비교는 **PART A 슬라이드 7** 참고. 여기서는 "FAISS 로 구조를 체감한 뒤 Chroma 로 이행"하는 흐름이 핵심이다.

벡터 DB 지형 — FAISS 다음은 Chroma



벡터 DB 비교 · FAISS(인메모리)·Chroma(임베디드 영속)·Pinecone(매니지드)·Milvus(분산) · 임베딩 저장 + ANN 검색

🎯 **FAISS → Chroma 로 이어지는 맥락**: FAISS 로 인덱스·검색 원리를 체감한 다음 단계는 **영속화·메타데이터 필터링·삭제를 한 줄로 처리하는** Chroma. 학습·일반 실무 RAG 의 기본은 Chroma, 수억 벡터·GPU 가속이 핵심이면 FAISS.

PART D

PART D
청킹 — 문서 분할 전략

RAG 성능을 결정하는 숨은 변수 — 약 20분

문서 전체를 단일 임베딩하면 안 되는 이유 4가지

① 의미 희석 (semantic dilution)

- 1만 자 문서 → 단일 1536d 벡터 → 챕터별 주제가 평균값으로 혼합 → 검색 정확도 저하

② 임베딩 모델의 입력 한도

- `text-embedding-3-small`: 8191 토큰 (~12,000 자 한국어) — PDF 한 페이지로도 종종 초과

③ 비용

- 답변에 필요한 부분은 한 문단인데 문서 전체를 LLM 에 주입 → 토큰 급증

④ Context window 압박

- 128K 토큰에 가까워질수록 **응답 품질 저하** (Lost in the Middle 현상)
- 작은 chunk → 정밀 retrieval → 작은 context → 좋은 답변

청킹의 목표

> **"하나의 chunk = 하나의 답변 가능한 의미 단위"** — 보통 300~1500자

CharacterTextSplitter vs RecursiveCharacterTextSplitter 외

전략	작동 방식	장점	단점
Character	N자 단위 단순 절단	가장 단순	문장·문단 중간 자름
Recursive ⭐	단락(`\n\n`) → 문장(`\n`) → 단어 → 글자 순으로 시도	문맥 보존 최선	약간 느림
Token-aware	tiktoken 으로 정확한 토큰 수	비용 예측 정확	한국어는 비효율적
Semantic	임베딩으로 의미 경계 탐지	정확한 의미 단위	색인 시간 ↑↑
MarkdownHeader	`#` / `##` 헤더 기준	구조화 문서에 최적	Markdown 전용
Code	AST 기반 함수·클래스 단위	코드 RAG 에 최적	언어별 파서 필요

본 코스 권장

- **Markdown · 일반 텍스트**: `RecursiveCharacterTextSplitter`
- **PDF**: `RecursiveCharacterTextSplitter.from_tiktoken_encoder` (토큰 정확도)
- **코드**: `Language.PYTHON` splitter (AST 기반)

가장 널리 쓰이는 분할기

코드 + 실행

```
from langchain_text_splitters \
    import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len,
    separators=["\n\n", "\n", "。", ".",
                "?", "!", " ", ""],
    is_separator_regex=False,
)

text = """1장. 인공지능의 역사

인공지능(AI)은 1950년대에 시작되었다. 앨런 튜링은 ...

2장. 머신러닝

머신러닝은 데이터에서 패턴을 학습하는 ...
"""

chunks = splitter.split_text(text)
for i, c in enumerate(chunks):
    print(f"[chunk {i}, {len(c)}자]")
    print(c[:80].replace('\n', ' '), "...")
```

작동 원리 (재귀)

1. 텍스트를 `\n\n` 으로 분할 시도
2. 한 조각이 1000자 넘으면 다시 `\n` 으로 분할
3. 그래도 넘으면 `.` 으로
4. 결국 `""` 또는 글자 단위까지 내려감

파라미터 의미

파라미터	의미
`chunk_size`	각 chunk 최대 글자
`chunk_overlap`	인접 chunk 와 겹치는 글자
`length_function`	한국어는 `len()`, 영어는 토큰 권장
`separators`	우선순위 분할 기준 리스트

> 🎯 **chunk_overlap** 200자가 핵심 — chunk 경계에 답이 걸쳐 있을 때 검색 정확도 ↑

비용을 정확히 추정하려면 토큰 단위 분할

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

# 토큰 단위 분할 (LLM context window 기준)
splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    encoding_name="cl100k_base",   # GPT-4 계열
    chunk_size=500,                # 토큰 단위!
    chunk_overlap=100,
    separators=["\n\n", "\n", ".", " "],
)

chunks = splitter.split_text(long_pdf_text)
print(f"chunk 수: {len(chunks)}")
print(f"평균 토큰: {sum(len(splitter._tokenizer.encode(c)) for c in chunks) / len(chunks):.0f}")
```

토큰 단위 분할 — 크기 선택과 한국어 고려

chunk_size 선택 가이드

용도	chunk_size (토큰)	이유
FAQ · QA	100~300	짧고 정밀한 답
일반 문서 RAG	500~1000	본 코스 권장
긴 문맥 추론	1500~2000	chapter 단위
chunk_overlap	chunk_size × 0.1~0.2	경계 답 누락 방지

한국어 주의

- "안녕하세요" = 5자, 약 6 토큰 (영어보다 비효율)
- 본 코스에서는 **한국어 =** `len()` **글자 단위**가 더 직관적

chunk 별 부여할 메타데이터

권장 메타데이터 스키마

```
{
    "source": "ml_intro.pdf",          # 출처 파일
    "page": 12,                         # 페이지 번호 (PDF)
    "chunk_id": "ml_intro_p12_c3",     # 고유 ID
    "chunk_idx": 3,                     # chunk 순번
    "total_chunks": 47,                 # 전체 청크 수 (디버그)
    "doc_title": "ML 입문",            # 문서 타이틀
    "section": "1.2 지도학습",         # 챕터/섹션 (있다면)
    "char_start": 12450,                # 원문 내 시작 위치 (citation)
    "char_end": 13420,
    "embedded_at": "2026-05-05"         # 임베딩 생성일 (재색인 추적)
}
```

각 필드의 용도

- `source` + `page` : RAG 답변 출처 인용 (Session 16 핵심)
- `chunk_id` : 중복 색인 방지, 업데이트 시 정확한 삭제
- `char_start` / `char_end` : 원문 하이라이트 (UI)
- `embedded_at` : 임베딩 모델 버전 변경 시 재색인 판별 기준

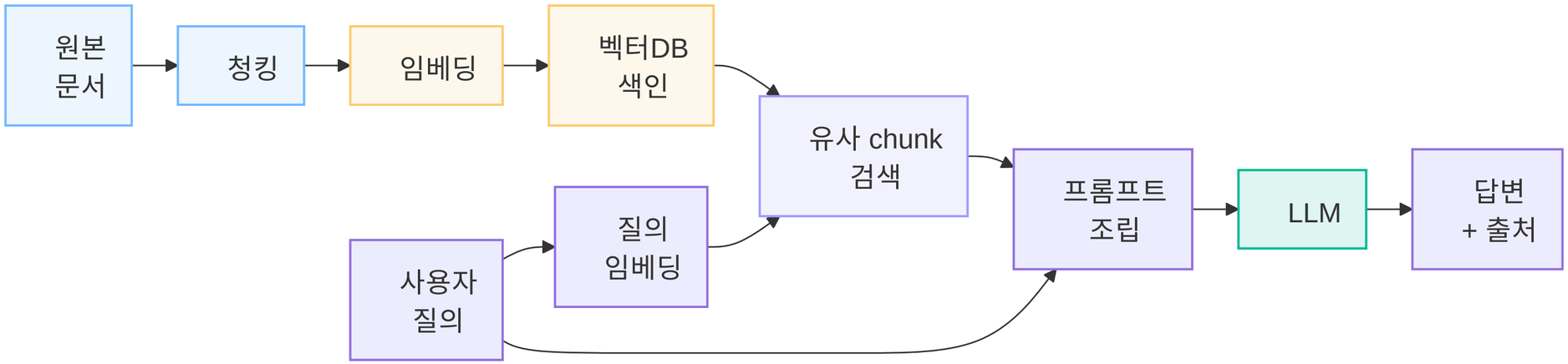
> 🎯 **추적성 = 신뢰성**. 메타데이터 없는 RAG 은 출처 불명의 답변이며, 곧 사용자 불신으로 이어진다.

PART E

PART E
첫 미니 RAG — 풀스택

30줄로 문서 Q&A — 약 20분

Retrieval → Augmented → Generation



3단계 핵심 함수

1. **Indexing** (한 번): chunks → embeddings → vectorDB
2. **Retrieval** (매 질의): query → embed → top-K chunks
3. **Generation** (매 질의): query + chunks → LLM → answer

단일 파일 문서 Q&A 구현

1. 색인 (한 번만)

```
from dotenv import load_dotenv
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_community.vectorstores import FAISS
from langchain_text_splitters \
    import RecursiveCharacterTextSplitter
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_community.document_loaders import TextLoader

load_dotenv()

loader = TextLoader("./my_notes.txt", encoding="utf-8")
docs = loader.load()

splitter = RecursiveCharacterTextSplitter(
    chunk_size=800, chunk_overlap=150)
chunks = splitter.split_documents(docs)
print(f"chunk 수: {len(chunks)}")

embeddings = OpenAIEmbeddings(
    model="text-embedding-3-small")
vectorstore = FAISS.from_documents(chunks, embeddings)
vectorstore.save_local("./faiss_idx")
```

2. 질의 + RAG 체인 (매번)

```
retriever = vectorstore.as_retriever(
    search_kwargs={"k": 4})

prompt = ChatPromptTemplate.from_messages([
    ("system",
     "다음 문서를 근거로만 답하세요. 모르면 '모릅니다'."),
    ("user", "문서:\n{context}\n\n질문: {question}"),
])

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

def format_docs(docs):
    return "\n\n".join(
        f"[chunk {i+1}] {d.page_content}"
        for i, d in enumerate(docs))

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

# 3. 사용
answer = rag_chain.invoke("이 문서의 핵심 주장은?")
print(answer)
```

🎯 **이 50줄이 Day 2 전체의 기반**이다. Session 14~16 은 이를 개선·확장한다.

PART F

PART F
미니 실습

문서 RAG — 약 25분

실습 과제 (25분)

1단계 — 데이터 준비 (5분)

- 대상 도메인 텍스트/마크다운 1개 (3000자 이상)
- 예시: 학과 노트 / 회사 위키 / 책 챕터 / GitHub README

2단계 — 색인 + 첫 검색 (10분)

```
# 위 슬라이드의 30~40줄 그대로
chunks = splitter.split_documents(docs)
vectorstore = FAISS.from_documents(chunks, embeddings)

# 직접 retriever 만 호출 (LLM 생략)
results = vectorstore.similarity_search_with_score(
    "질문", k=4)
for d, s in results:
    print(f"score={s:.3f}")
    print(d.page_content[:200])
```

3단계 — RAG 전체 (LLM 답변까지, 5분)

- 위 슬라이드 코드 그대로
- 대상 도메인 질문 3개로 실험

4단계 — 비교 분석 (5분)

- 동일한 3개 질문을 **RAG 없이**(`llm.invoke(질문)`)도 실행
- 비교 항목:
- RAG 가 답한 것 / 일반 LLM 이 답하지 못한 것
- RAG 가 출처 없이 부정확하게 답한 것

📤 보고

- chunk_size 를 500 → 1500 → 3000 으로 바꿔가며 검색 품질 변화 관찰
- RAG 의 가치가 가장 두드러진 질문은 무엇인가

핵심 9가지

1. ✅ 임베딩(텍스트→벡터·코사인 유사도)은 **Day 1 의 범위** — 이번 회차는 그 벡터의 **저장·검색 구조**
2. ✅ **벡터 DB** = 임베딩 + 텍스트·메타데이터 저장 **+ ANN 검색 색인**
3. ✅ **ANN** = 정확도를 소폭 양보, 검색 공간을 좁혀 밀리초 검색 (정확도↔속도 맞교환)
4. ✅ Brute-force = O(N) — 1억 벡터에서 한계 → ANN 이 해결하는 문제
5. ✅ **FAISS** 인덱스 3종 — Flat / IVF / HNSW (속도·메모리·정확도)
6. ✅ HNSW = **다층 작은 세계 그래프** (O(log N) 검색), IVF = 클러스터 후 일부만
7. ✅ 4대 DB — **Chroma**(학습·필터)·FAISS(대규모·GPU)·Pinecone(매니지드)·Milvus(분산)
8. ✅ 청킹은 RAG 성능을 가르는 숨은 변수 (chunk_size · overlap · Recursive/token-aware)
9. ✅ **메타데이터 = 추적성 = 신뢰성** — 출처 인용의 기반

실습 과제

필수

- [ ] 문서 1개로 FAISS 미니 RAG 동작 확인
- [ ] chunk_size 3종 실험 → 문서에 적합한 값을 한 줄로 기록

옵션

- [ ] `2.langchain/7.RAG/3.vectorstore/` 실행 — 영속화 + 검색 모드 비교
- [ ] 임베딩 모델 비교 — `text-embedding-3-small` vs `-large` 같은 질의 결과 비교

📤 정리

- chunk_size 실험 결과 — 어떤 값에서 가장 좋은 답이 나왔으며, 그 원인은 무엇으로 추정되는가