AI 바이브/페어 개발 — CLAUDE CODE/CODEX 과정

파일 인덱서 + RAG 검색 MCP 300줄로 만드는 개인 지식 검색

Session 11 · RAG 검색 MCP 서버로 여는 개인 지식 검색

sqlite · OpenAI Embeddings · FastMCP Tool·Resource · watchdog · Claude·Codex 등록

강사 박수현 · 🚀 젠아이랩스(GenAI Labs)

바이브코딩 2일 집중 · 프롬프트·컨텍스트·하네스 엔지니어링

PART 1

왜 이 실습인가 — 문제 · RAG · MCP 노출 이유

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

세션 목표 — 노트북에서 네 가지가 동시에

이 세션이 끝나면 참여자 노트북에서 다음 네 가지가 동시에 일어난다.

① 인덱스가 생긴다

자기 문서 폴더가 인덱싱된 sqlite 파일이 생긴다.

② 실시간 갱신

폴더에 새 파일을 떨어뜨리면 3초 안에 인덱스가 갱신된다.

③ 자연어 검색

Claude Code 에서 '회의록 중에 배포 실패 원인 언급된 것 찾아줘' 라고 물으면 관련 파일 3개가 검색 결과로 반환된다.

④ 클라이언트 무관

같은 서버를 Codex 에서 붙여도 그대로 동작한다.

📌 만드는 코드는 **300줄 남짓**. 검색 UI 를 새로 만드는 대신, MCP 서버 하나로 자기가 이미 쓰고 있는 AI 도구에 검색을 붙인다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

사내 문서 검색이 안 풀리는 이유 — 세 겹의 벽

컨플루언스 검색을 켰고, 노션 검색을 켰고, GitHub 리포에서 grep 을 돌렸다. 그런데도 '이 정보 어디 있더라' 는 사라지지 않는다.

🔍 개념 검색이 필요

'결제 실패' 로 검색하는데 문서에는 'payment gateway timeout' 이라고만 적혀 있다.

키워드 검색은 이걸 못 잡는다.

🧩 파편이 흩어져 있다

회의록은 노션, 결정 사항은 지라 티켓, 상세 스펙은 GitHub 마크다운.

검색 인터페이스가 도구마다 다르다.

🖥️ AI 도구 안에서 쓰고 싶다

결과를 브라우저에서 확인하고 Claude Code 로 돌아와 붙여 넣는 흐름은 마찰이 크다.

컨텍스트 스위칭 비용.

📌 세 겹을 한 파이프로 잇는 접근이 **RAG 를 MCP 서버로 노출** 하는 방식이다. 개념 검색 (임베딩) · 여러 소스 통합 (파일 인덱서) · AI 도구 안에서 호출 (MCP).

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

RAG 3단계 — Retrieval · Augmentation · Generation

RAG · Retrieval-Augmented Generation 은 이름과 달리 흐름이 단순하다.

🔎 Retrieval

하는 일 — 질의와 관련된 문서 조각을 찾는다.

이 세션에서 — 임베딩 유사도 검색.

➕ Augmentation

하는 일 — 찾은 조각을 프롬프트에 붙인다.

이 세션에서 — MCP Tool 리턴 → 클라이언트가 컨텍스트로.

✍️ Generation

하는 일 — LLM 이 그 조각을 근거로 답한다.

이 세션에서 — Claude Code · Codex 가 자연어 답변 생성.

📌 핵심은 임베딩

- **감각 먼저** — 임베딩은 **문장을 좌표로 바꾼 것**이다. 1,536 개 축을 가진 공간에 각 문장이 점 하나로 찍힌다. 의미가 비슷하면 점끼리 가깝고, 무관하면 멀다.
- 텍스트를 고정 길이 벡터로 바꾼 것. OpenAI 의 `text-embedding-3-small` 은 **1,536 차원** 벡터를 낸다.
- 의미가 비슷한 두 문장은 벡터 공간에서 **가까이 위치**한다. '결제 실패' 와 'payment gateway timeout' 의 코사인 유사도가 0.8 이상이 되는 식.
- **인덱싱 시점**에 문서를 조각내고 각 조각을 벡터로 저장 → **검색 시점**에 질의를 같은 방식으로 임베딩해 저장된 벡터들과 코사인 유사도 계산 → 상위 K 개 반환.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

MCP 로 노출하는 이유 — 세 가지 이득

같은 RAG 파이프라인을 웹서비스로 만들 수도 있다. FastAPI 로 검색 API 를 노출하고 프론트를 붙이는 흐름. 그 결과물은 **브라우저 창 하나가 더 늘어날 뿐**이다. 개발자는 이미 하루 종일 Claude Code · Codex 창에 있다.

🗣️ 자연어 인터페이스가 공짜

검색 UI 를 안 만든다.

'지난 달 데이터베이스 관련 결정 찾아줘' 를 LLM 이 알아서 툴 호출로 매핑.

🔄 클라이언트 무관

Claude Code 에서 만들고 Codex 에서 그대로 쓴다.

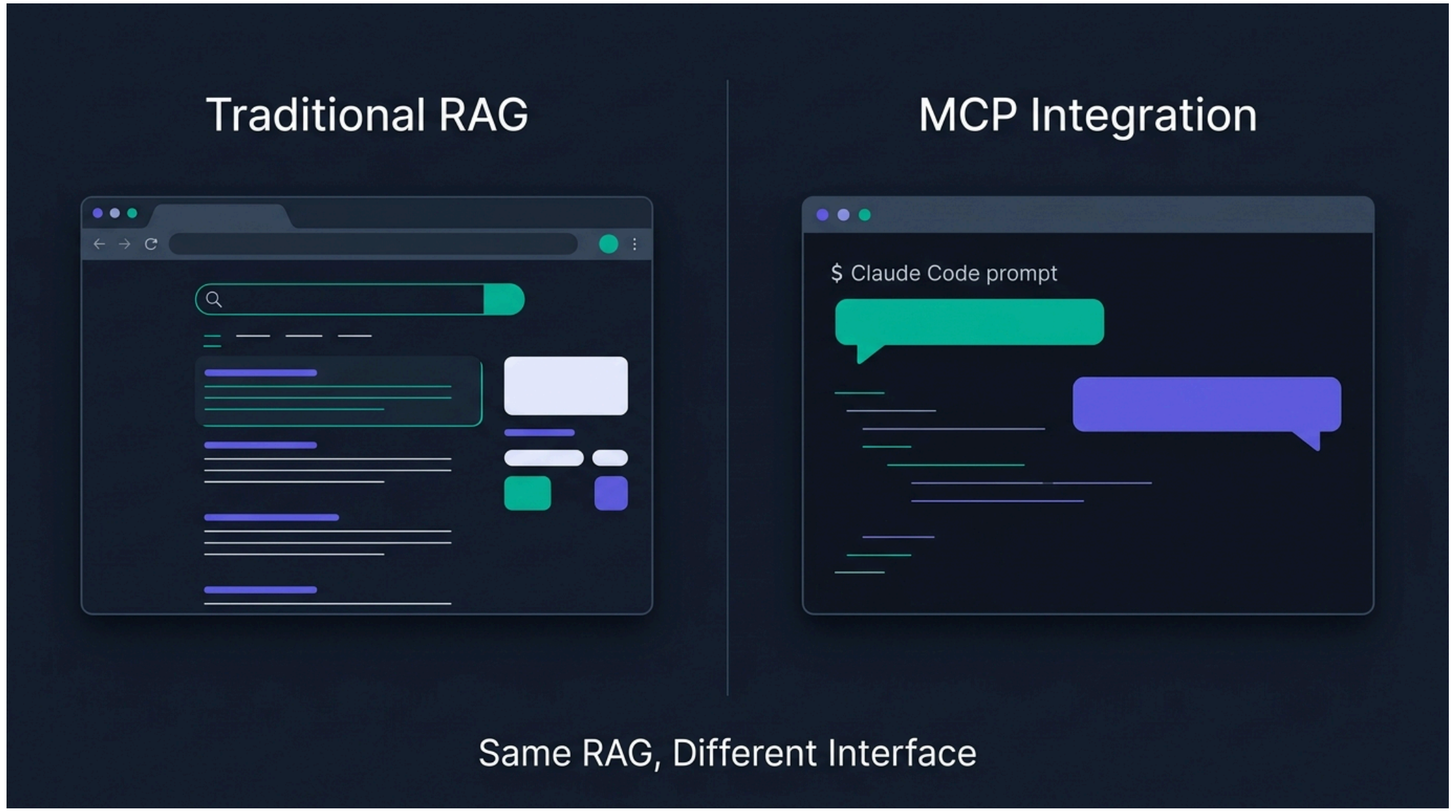
Cursor · Windsurf 도 같은 서버가 붙는다.

🎯 LLM 후처리가 보너스

검색 결과 5개를 받으면 LLM 이 '이 세 개가 질문에 맞다' 고 재순위화해 준다.

재순위화 코드를 짜지 않아도 된다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG



MCP 로 노출하는 이유 — 세 가지 이득

PART 2

아키텍처 — 4개 컴포넌트 · sqlite · 청킹 · 임베딩

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

4개 컴포넌트 — 파일 하나 = 책임 하나

서버는 네 조각. 각 조각의 책임을 처음부터 분리해 두면 구조가 단순하게 유지된다.

🔧 Indexer · `indexer.py`

폴더 훑고 **청킹 → 임베딩 → sqlite 저장**.

👁️ Watcher · `watcher.py`

폴더 변경 **실시간 감지 → 인덱스 갱신**.

🔍 Searcher · `searcher.py`

질의 **임베딩 → 유사도 상위 K 반환**.

🌐 MCP Server · `server.py`

위 셋을 **Tool · Resource** 로 노출.

📌 **설계 원칙.** 각 파일은 하나의 책임. 인덱서·검색기·워처는 서로 직접 호출하지 않고 **sqlite 를 매개로 통신**한다. MCP 서버는 얇게 — Tool 함수는 Searcher 를 감싸는 얇은 래퍼.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

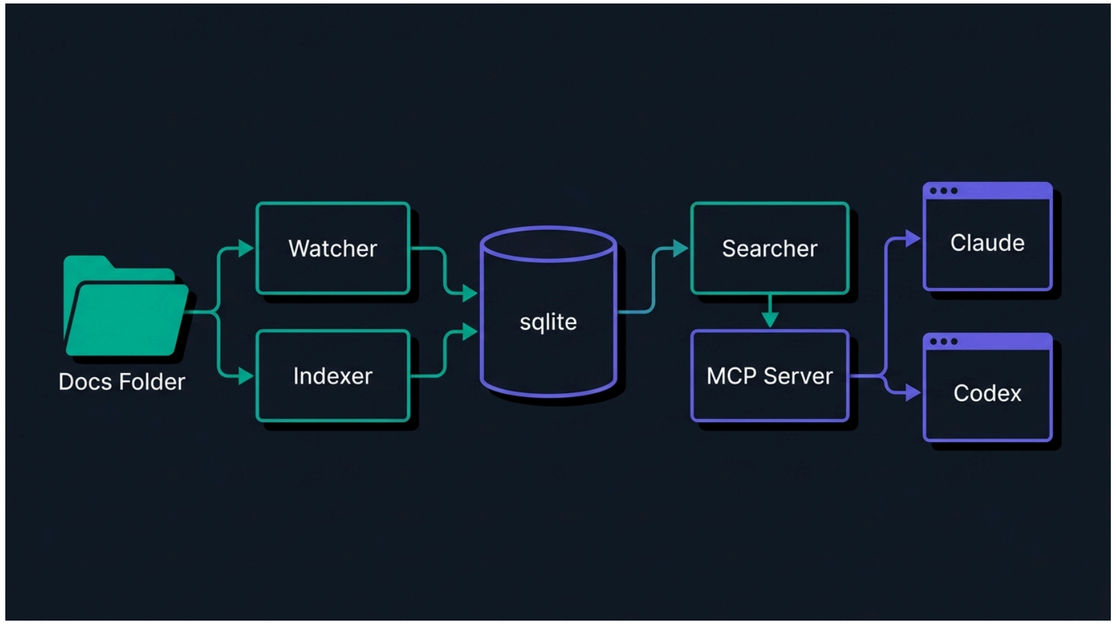
아키텍처 — 왜 sqlite 를 매개로

📌 컴포넌트가 서로 직접 호출하지 않는다

- Watcher · Indexer 는 sqlite 에 **쓰기만**
- Searcher 는 sqlite 를 **읽기만**

🧵 이 구조의 이득

- **상태 공유가 없다** — 어느 조각이든 독립 재기동
- 인덱싱 스크립트를 배치로 따로 돌려도 그대로
- 각 파일 단위로 **테스트가 붙는다**



Docs → Watcher · Indexer → sqlite → Searcher → MCP Server → Claude · Codex

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

sqlite 스키마 — sources · chunks · embeddings

sqlite 로 충분하다. **개인 문서 수만 개 규모까지 성능 문제 없다.** 3개 테이블로 나눈다.

```
CREATE TABLE sources (
    id INTEGER PRIMARY KEY, path TEXT UNIQUE NOT NULL,
    mtime REAL NOT NULL, doc_type TEXT NOT NULL, indexed_at TEXT NOT NULL
);
CREATE TABLE chunks (
    id INTEGER PRIMARY KEY,
    source_id INTEGER NOT NULL REFERENCES sources(id) ON DELETE CASCADE,
    ord INTEGER NOT NULL, start_line INTEGER, end_line INTEGER, text TEXT NOT NULL
);
CREATE TABLE embeddings (
    chunk_id INTEGER PRIMARY KEY REFERENCES chunks(id) ON DELETE CASCADE,
    dim INTEGER NOT NULL, vec BLOB NOT NULL   -- float32 × dim
);
CREATE INDEX idx_chunks_source ON chunks(source_id);
```

📌 **세 가지 결정.** - `sources.mtime` **저장** — 파일 수정 시간 비교로 변경된 파일만 다시 인덱싱. 재실행 시 API 비용 절감. - `chunks` · `embeddings` **분리** — 청크 텍스트는 결과 반환용, 벡터는 유사도 계산용. 벡터만 메모리에 올려야 할 때 쿼리가 간단. - `ON DELETE CASCADE` — 파일 삭제 시 `sources` 만 지우면 나머지 자동 정리. 예: `DELETE FROM sources WHERE id=5;` 한 줄 → 해당 파일의 `chunks` 행 · `embeddings` BLOB 이 함께 사라진다 (수동으로 두 테이블 청소할 필요 없음).

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

청킹 전략 — 문서 유형별

문서를 통째로 임베딩하면 안 된다. 임베딩 모델의 컨텍스트 한계보다 **검색 정밀도** 가 더 큰 이유다. 문서 전체를 한 벡터로 압축하면 문서의 특정 문단만 관련된 질문에서 유사도가 낮아진다.

📝 마크다운

청킹 기준 — H1·H2 헤딩 경계.

최대 크기 — 800 토큰.

이유 — 헤딩이 자연스러운 의미 단위.

💻 코드

청킹 기준 — 함수·클래스 단위.

최대 크기 — 400 라인.

이유 — 함수 하나가 의미 단위.

📄 일반 텍스트

청킹 기준 — 200 단어 슬라이딩 윈도우.

오버랩 — 30 단어.

이유 — 문단 경계가 불명확할 때.

📌 **오버랩 이유.** 청크 경계에 걸린 문장은 앞뒤 청크 어디에도 온전히 안 들어간다. 30 단어 오버랩을 두면 경계 문장이 **최소 한쪽에는 문맥과 함께 남는다**. 검색 재현율이 크게 오른다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

임베딩 · 배치 · float32 BLOB

청크 100개를 임베딩한다고 API 를 100번 호출하지 않는다. **OpenAI 는 배치 요청을 지원한다.**

📊 세 가지 지표

- **모델** — `text-embedding-3-small`. 1,536 차원. 2026년 기준 **100만 토큰당 $0.02**. 문서 1만 개를 인덱싱해도 비용은 몇 달러 수준이다.
- **배치 크기** — 요청당 청크 **100개까지 안전**.
- **유사도** — 코사인 유사도. numpy 로 한 줄.

💾 sqlite BLOB 로 저장

float32 배열을 그대로 바이트로.

1,536 차원 × 4바이트 = 6,144 바이트. 청크 1만 개면 60MB.

`numpy.tobytes()` · `numpy.frombuffer()` 로 왕복. JSON 직렬화보다 빠르고 저장 공간도 **3배 이상 절약**.

왜 3배? JSON 은 float 하나를 `"0.12345678"` 처럼 **문자열**로 저장한다 (원소당 12~18 바이트, float64 정밀도 유지). BLOB 은 float32 로 **원소당 4바이트 고정**. 1,536 차원 기준 **대략 20KB → 6KB** (3배 이상 절약).

```
vec = np.array([0.1, 0.2, ...], dtype=np.float32)
blob = vec.tobytes()                              # 저장
back = np.frombuffer(blob, dtype=np.float32)    # 복원
```

PART 3

Python 구현 — indexer · searcher

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

인덱서 · sqlite 초기화 — `indexer.py`

`indexer.py` 부터. 폴더 경로를 받아 청킹·임베딩·저장까지 처리한다.

```
"""파일 인덱서 · 청킹 · 임베딩 · sqlite 저장."""
import os, sqlite3
from datetime import datetime
from pathlib import Path

DB_PATH = os.environ.get("RAG_DB_PATH", "./rag_index.db")
SCHEMA = """<PART 2 의 스키마 그대로>"""

def get_db() -> sqlite3.Connection:
    conn = sqlite3.connect(DB_PATH)
    conn.execute("PRAGMA foreign_keys = ON")   # CASCADE 활성화
    conn.executescript(SCHEMA)
    return conn
```

📌 **두 가지 결정.** - `foreign_keys = ON` — sqlite 는 `foreign_keys` 가 기본 OFF. `ON DELETE CASCADE` 를 쓰려면 반드시 켠다. - `executescript` — 여러 CREATE 문을 한 번에.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

파일 유형 판별 · 청킹 함수

확장자로 유형을 판별하고 그에 맞는 청킹 함수를 부른다.

```
def detect_doc_type(path: Path) → str:
    ext = path.suffix.lower()
    if ext in {".md", ".markdown"}: return "markdown"
    if ext in {".py", ".ts", ".js", ".go", ".rs", ".java"}: return "code"
    return "text"

def chunk_markdown(text: str, max_tokens: int = 800) → list[dict]:
    """H1·H2 헤딩 경계로 청킹."""
    import re
    lines = text.splitlines()
    chunks, current, current_start = [], [], 1
    heading_pat = re.compile(r"^#{1,2}\s")
    for i, line in enumerate(lines, start=1):
        if heading_pat.match(line) and current:
            chunks.append({"text": "\n".join(current),
                           "start_line": current_start, "end_line": i - 1})
            current, current_start = [line], i
        else:
            current.append(line)
    if current:
        chunks.append({"text": "\n".join(current),
                       "start_line": current_start, "end_line": len(lines)})
    return [c for c in chunks if c["text"].strip()]
```

📌 **관찰.** 헤딩 경계로 자를 때 **직전 헤딩 라인 번호**를 `current_start` 로 잡아 둔다. 검색 결과에 라인 범위를 함께 실어 주면 참여자가 파일 안에서 바로 점프할 수 있다.

텍스트 청킹 · 라우터 함수

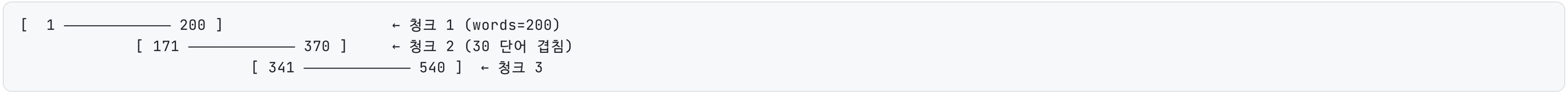
일반 텍스트와 코드에 쓰이는 슬라이딩 윈도우 · 라우터 함수.

```
def chunk_text(text: str, words: int = 200, overlap: int = 30) → list[dict]:
    """단어 슬라이딩 윈도우 + 오버랩."""
    tokens, chunks, step = text.split(), [], words - overlap
    for i in range(0, len(tokens), step):
        seg = tokens[i:i + words]
        if not seg: break
        chunks.append({"text": " ".join(seg), "start_line": None, "end_line": None})
    return chunks

def chunk_file(path: Path) → list[dict]:
    text = path.read_text(encoding="utf-8", errors="ignore")
    doc_type = detect_doc_type(path)
    if doc_type == "markdown": return chunk_markdown(text)
    if doc_type == "code":     return chunk_text(text, words=400, overlap=50)
```

⚠️ `errors="ignore"` — 인코딩이 깨진 파일 하나로 배치가 중단되는 것을 방지. PDF 는 이 세션에서 다루지 않는다 (확장 파트에서 언급).

📌 슬라이딩 윈도우 30 단어 겹침 — 3단계 도해



step = words − overlap = 170 단어씩 전진. 경계에 걸린 문장이 한쪽 청크에서 잘려도 옆 청크에서 온전히 남아 검색 리콜이 유지된다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

OpenAI 임베딩 배치 호출 — `embed_batch`

청크 100개를 한 요청으로 벡터 100개를 받는다.

```
import numpy as np
from openai import OpenAI

client = OpenAI(api_key=os.environ["OPENAI_API_KEY"])
EMBED_MODEL, EMBED_DIM = "text-embedding-3-small", 1536

def embed_batch(texts: list[str]) → np.ndarray:
    """(len, 1536) float32 배열 반환."""
    if not texts:
        return np.zeros((0, EMBED_DIM), dtype=np.float32)
    resp = client.embeddings.create(model=EMBED_MODEL, input=texts)
    return np.stack([np.array(d.embedding, dtype=np.float32) for d in resp.data])
```

📌 **세 가지 결정.** - `dtype=np.float32` — OpenAI 응답은 float64 리스트. 그대로 두면 저장 크기가 2배. - **빈 배열 조기 리턴** — API 는 빈 입력에서 에러. - **재시도는 SDK 에 위임** — `openai` SDK 가 자동 재시도·백오프 담당.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

파일 인덱싱 · `index_file`

파일 하나를 받아 청킹·임베딩·저장까지 한 함수에서 처리한다. **mtime 비교로 skip** 이 핵심 결정.

```
def index_file(conn: sqlite3.Connection, path: Path) → int:
    """이미 최신이면 skip. 반환은 저장된 청크 수."""
    mtime = path.stat().st_mtime
    doc_type = detect_doc_type(path)

    # 1) 최신이면 skip
    row = conn.execute("SELECT id, mtime FROM sources WHERE path = ?",
                       (str(path),)).fetchone()
    if row and row[1] ⩾ mtime:
        return 0
    # 2) 변경분이면 기존 source 삭제 (CASCADE)
    if row:
        conn.execute("DELETE FROM sources WHERE id = ?", (row[0],))

    # 3) 청킹 · 임베딩
    chunks = chunk_file(path)
    if not chunks: return 0
    vecs = embed_batch([c["text"] for c in chunks])

    # 4) 저장 · 청크·벡터 삽입 (다음 슬라이드)
    ...
    conn.commit()
```

📌 **세 가지 관찰.** - **mtime 비교로 skip** — 폴더를 다시 인덱싱해도 변경 없는 파일은 API 를 호출하지 않는다. - **기존 source 삭제 후 재삽입** — 수정 파일은 청크 개수가 달라진다. 부분 업데이트보다 전체 재삽입이 안전. - `commit()` **은 파일 단위** — 청크 단위는 느리고, 폴더 단위는 중간 실패 시 앞서 처리한 파일까지 전부 롤백된다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

청크·벡터 삽입 — `index_file` 뒷부분

앞 슬라이드의 `# 4) 저장` 이하가 아래처럼 이어진다.

```
    # 4) 저장
    cur = conn.execute(
        "INSERT INTO sources (path, mtime, doc_type, indexed_at) VALUES (?, ?, ?, ?)",
        (str(path), mtime, doc_type, datetime.utcnow().isoformat()))
    source_id = cur.lastrowid
    for ord_, (chunk, vec) in enumerate(zip(chunks, vecs)):
        cur = conn.execute(
            "INSERT INTO chunks (source_id, ord, start_line, end_line, text) "
            "VALUES (?, ?, ?, ?, ?)",
            (source_id, ord_, chunk["start_line"], chunk["end_line"], chunk["text"]))
        conn.execute("INSERT INTO embeddings (chunk_id, dim, vec) VALUES (?, ?, ?)",
                     (cur.lastrowid, EMBED_DIM, vec.tobytes()))
    conn.commit()
    return len(chunks)
```

📌 **관찰.** `cur.lastrowid` 로 방금 INSERT 한 소스·청크 ID 를 즉시 회수. `vec.tobytes()` 로 float32 배열을 BLOB 으로 직행. **JSON 직렬화 우회.**

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

폴더 인덱싱 · 삭제 감지 — `index_folder`

폴더 전체 훑기와 삭제된 파일 정리를 한 함수에.

```
def index_folder(conn: sqlite3.Connection, root: Path,
                 patterns: list[str] = None) → dict:
    patterns = patterns or ["*.md", "*.markdown", "*.txt",
                            "*.py", "*.ts", "*.js"]
    files = set()
    for pat in patterns:
        files.update(root.rglob(pat))

    stats = {"added": 0, "skipped": 0, "removed": 0, "chunks": 0}
    for f in files:
        n = index_file(conn, f)
        if n: stats["added"] += 1; stats["chunks"] += n
        else: stats["skipped"] += 1

    # 디스크에 없는데 sqlite 에 남은 것 정리 (orphans)
    known = {r[0] for r in conn.execute("SELECT path FROM sources")}
    disk = {str(f) for f in files}
    for path in known - disk:
        conn.execute("DELETE FROM sources WHERE path = ?", (path,))
        stats["removed"] += 1
    conn.commit()
    return stats
```

📌 **관찰.** `known - disk` 로 **디스크에 없는데 sqlite 에 남은 것** (orphan) 을 잡아 정리. 파일 삭제·이동을 폴더 인덱싱 한 번에 흡수.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

유사도 검색 — `searcher.py` · numpy 벡터화

`searcher.py` — 질의를 받아 상위 K 개 청크 반환.

```
"""유사도 검색."""
import numpy as np, sqlite3
from indexer import embed_batch

SQL = ("SELECT e.chunk_id, e.vec, c.text, c.start_line, c.end_line, s.path, s.doc_type "
       "FROM embeddings e JOIN chunks c ON c.id = e.chunk_id "
       "JOIN sources s ON s.id = c.source_id")

def search(conn: sqlite3.Connection, query: str, top_k: int = 5) → list[dict]:
    if not query.strip(): return []
    q_vec = embed_batch([query])[0]                        # (1536,)
    rows = conn.execute(SQL).fetchall()
    if not rows: return []

    vecs = np.stack([np.frombuffer(r[1], dtype=np.float32) for r in rows])
    q_norm  = q_vec / (np.linalg.norm(q_vec) + 1e-8)      # 0 벡터 방어
    v_norms = vecs / (np.linalg.norm(vecs, axis=1, keepdims=True) + 1e-8)
    scores  = v_norms @ q_norm                             # (N,) 코사인 유사도
    top_idx = np.argsort(-scores)[:top_k]
    return [{"score": float(scores[i]), "path": rows[i][5], "doc_type": rows[i][6],
```

📌 **네 가지 관찰.** - **전 벡터 메모리 로드** — 1만 청크 × 6KB = 60MB. 개인 규모는 무리 없음. 임계 초과 시 pgvector·Qdrant 로 이관. - `v_norms @ q_norm` — numpy `@` = **행렬 곱**. `(N, 1536) @ (1536,)` = `(N,)`, for 루프 없이 청크별 유사도 한 줄. - `np.argsort(-scores)` — 내림차순 정렬 관용구. - `+ 1e-8` — **0 벡터 방어**. 빈 질의로 `norm` 이 0 이 되어도 `nan` 없이 나눗셈.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

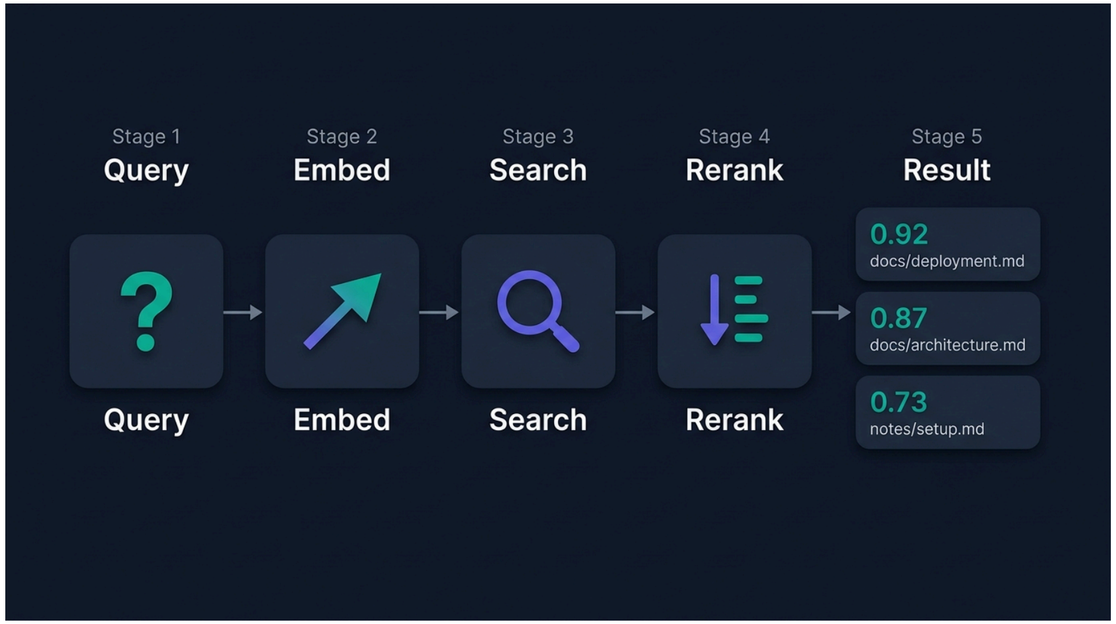
검색 파이프라인 — 한눈 요약

📌 흐름 5단계

1. **자연어 질의** → `search(query, top_k)`
2. **질의 임베딩** — 1,536 차원 벡터 1개
3. **sqlite 로드** — 전체 벡터 → numpy 배열
4. **코사인 유사도** — `v_norms @ q_norm` 한 줄
5. **상위 K 개 반환** — score · path · text

💡 관찰

- 한 질의당 **OpenAI API 콜은 1회** (질의 임베딩만)
- 유사도 계산은 로컬 numpy — **지연 없음**



Query → embedding vector → sqlite → top-K chunks

PART 4

MCP 서버 노출 — Tool · Resource · Watchdog · 등록

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

Tool · Resource 설계 — 두 조각의 역할

서버는 Tool 하나, Resource 하나를 노출한다.

🔧 Tool — `search(query, top_k)`

성격 — 클라이언트가 호출하는 함수. 부작용 있는 액션 (임베딩 API 호출).

반환 — 요약된 청크 5개.

이유 — 파일 전체를 리턴하면 프롬프트 폭발.

📄 Resource — `file://{path}`

성격 — 클라이언트가 URI 로 참조하는 읽기 전용 데이터.

반환 — 검색 결과 파일의 원문 전체.

이유 — LLM 이 청크 내용만으로 부족해 원문 확인이 필요하다고 판단할 때 열린다.

📌 **설계 이유.** 검색 → LLM 관련성 판단 → 필요한 파일만 원문 로드. **컨텍스트 낭비 없이 확장.** Tool 이 청크로 힌트를 주고, Resource 가 원문 접근을 연다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

FastMCP 서버 뼈대 — `server.py`

`server.py` 의 도입부. 로거·루트 폴더 확정·FastMCP 인스턴스.

```
"""파일 시스템 인덱서 + RAG 검색 MCP 서버."""
import os, sys, logging
from pathlib import Path
from mcp.server.fastmcp import FastMCP
from indexer import get_db, index_folder, index_file
from searcher import search as search_impl

logging.basicConfig(level=logging.INFO, stream=sys.stderr,
                    format="%(asctime)s [%(levelname)s] %(message)s")
log = logging.getLogger("rag-indexer")
ROOT = Path(os.environ["RAG_INDEX_ROOT"]).expanduser().resolve()
mcp = FastMCP("rag-indexer")
```

📌 **두 가지 결정.** - **인덱싱 대상 폴더는 환경 변수** (`RAG_INDEX_ROOT`) — 하드코딩하면 재사용할 때 매번 수정. - `expanduser().resolve()` — `~/notes` 를 절대 경로로 확정.

⚠️ **로그는 stderr 로만.** stdio 트랜스포트는 stdout 이 JSON-RPC 전용. `print()` 한 줄이 서버 전체를 죽인다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

`search` Tool — docstring 이 곧 LLM 스펙

```
@mcp.tool()
def search(query: str, top_k: int = 5) → list[dict]:
    """파일 인덱스에서 질의와 관련된 문서 조각을 유사도 상위 K개 반환한다.

    Args:
        query: 검색 질의. 자연어 문장.
        top_k: 반환할 조각 개수. 기본 5, 최대 20 권장.

    Returns:
        각 조각은 dict — score·path·doc_type·start_line·end_line·text 키.
        결과는 유사도 내림차순.
    """
    log.info("search: %r / top_k=%d", query, top_k)
    conn = get_db()
    try:
        return search_impl(conn, query, top_k=min(top_k, 20))
    finally:
        conn.close()
```

📌 **세 가지 결정.** - **docstring 이 곧 LLM 스펙** — `Args:` · `Returns:` 를 명시하면 Claude 가 파라미터 의미를 정확히 잡는다. - `top_k` **상한 20** — 사용자가 100 을 넣어도 20 으로 자른다. 컨텍스트 폭발 방지. - `try/finally` **로 커넥션 정리** — 예외 상황에서도 sqlite 커넥션이 새지 않는다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

`file://` Resource — 경로 탈출 방어

Resource 는 URI 로 참조되며, `{path}` 파라미터가 자동 매핑된다.

```
@mcp.resource("file://{path}")
def file_content(path: str) → str:
    """인덱스에 있는 파일의 원문. LLM 이 검색 청크의 path 를 그대로 열어
    파일 전체를 컨텍스트로 가져갈 수 있다."""
    from urllib.parse import unquote
    target = Path(unquote(path)).resolve()
    try:
        target.relative_to(ROOT)                # target 이 ROOT 하위가 아니면
    except ValueError:                          # ValueError 발생 → 경로 탈출 시도로 판정
        raise PermissionError(f"인덱스 밖 파일 접근 금지: {target}")
    if not target.is_file():
        raise FileNotFoundError(str(target))
    return target.read_text(encoding="utf-8", errors="ignore")
```

⚠️ **경로 탈출 방어가 없으면 LLM 이 `file:///etc/passwd` 를 요청해도 통과.** `relative_to(ROOT)` 로 **인덱싱 폴더 하위인지 검사**한다. `target` 이 `ROOT` 의 상위 폴더거나 형제 폴더면 `ValueError` 가 던져진다 (예: `ROOT=/home/me/notes`, `target=/etc/passwd`). 이를 `PermissionError` 로 재포장해 명시적으로 실패시킨다.

📌 **에러는 raise.** FastMCP 가 클라이언트 응답으로 자동 변환한다. `try/except` 로 삼키지 말 것.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

watchdog · 실시간 인덱싱

서버가 뜬 동안 폴더 변경을 즉시 반영한다. `watchdog` 의 **Observer** 가 별도 스레드로 파일 이벤트를 감지해 Handler 로 재인덱싱을 넘긴다 — 메인 서버는 MCP 응답에 전념.

```
from watchdog.events import PatternMatchingEventHandler
from watchdog.observers import Observer

class RagHandler(PatternMatchingEventHandler):
    def __init__(self):
        super().__init__(patterns=["*.md", "*.txt", "*.py", "*.ts", "*.js"],
                         ignore_directories=True)

    def _reindex(self, path: str):
        try:
            conn = get_db()
            log.info("reindexed %s (%d chunks)", path, index_file(conn, Path(path)))
            conn.close()
        except Exception as e:
            log.error("reindex failed: %s / %s", path, e)

    def on_created(self, event):  self._reindex(event.src_path)
    def on_modified(self, event): self._reindex(event.src_path)
    def on_deleted(self, event):
        conn = get_db()
```

📌 **관찰.** 세 이벤트 (`on_created` · `on_modified` · `on_deleted`) 만 훅해도 충분. `PatternMatchingEventHandler` 의 확장자 필터가 `.git` 변경 재인덱싱을 막는다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

watcher 시작 · 서버 부트스트랩

Observer 를 데몬 스레드로 띄우고, 서버 진입점에서 초기 인덱싱 + 워처 시작을 한 번에 처리한다.

```
def start_watcher(root: Path):
    observer = Observer()
    observer.schedule(RagHandler(), str(root), recursive=True)
    observer.daemon = True
    observer.start()
    log.info("watcher started on %s", root)

def bootstrap():
    conn = get_db()
    stats = index_folder(conn, ROOT)
    log.info("initial index: %s", stats)
    conn.close()
    start_watcher(ROOT)

if __name__ == "__main__":
    bootstrap()
    mcp.run()
```

📌 **세 가지 결정.** - `daemon = True` — 워처 스레드가 메인 프로세스와 함께 죽는다. - `recursive=True` — 하위 폴더까지 감지. - **초기 인덱싱 후 워처 시작** — 서버 기동 순간에 폴더 상태를 한 번 정합. 이후는 이벤트 기반.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

Claude Code 등록 — `claude mcp add`

Claude Code 에서 서버를 붙이는 명령은 하나다. **환경 변수 3개를 함께 담는다.**

```
claude mcp add rag-indexer \
  --env OPENAI_API_KEY=sk-... \
  --env RAG_INDEX_ROOT=/home/me/notes \
  --env RAG_DB_PATH=/home/me/.rag/index.db \
  -- python /absolute/path/to/server.py

claude mcp list          # rag-indexer  running  1 tool · 1 resource
```

🔍 확인 순서

1. `claude mcp list` 에 `rag-indexer` 가 `running` 으로 뜨는가.
2. `1 tool · 1 resource` 카운트가 정확한가.
3. Claude Code 를 새로 띄우고 `/mcp` 로 `search` 툴이 잡히는가.

🎯 자연어 첫 호출

Claude Code 대화창에서

```
회의록 중에 배포 실패
원인 언급된 것 찾아줘
```

`search` 툴이 호출되고 결과 청크 5개가 응답. LLM 이 재정리해 답을 낸다.

⚠️ **절대 경로 강제.** `python server.py` 처럼 상대 경로를 쓰면 클라이언트가 어느 디렉터리에서 스크립트를 찾을지 모른다. 반드시 `/absolute/path/to/server.py`.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

Codex 등록 · 한 서버 두 클라이언트

Codex 는 `~/.codex/config.toml` 에 정의한다. **서버 코드는 동일**, 등록 방식만 다르다.

```
[mcp.servers.rag-indexer]
command = "python"
args = ["/absolute/path/to/server.py"]

[mcp.servers.rag-indexer.env]
OPENAI_API_KEY = "sk-..."
RAG_INDEX_ROOT = "/home/me/notes"
RAG_DB_PATH = "/home/me/.rag/index.db"
```

📌 **한 서버 · 두 클라이언트 — 이 구조가 MCP 의 핵심 이득이다.**

- Claude Code 에서 만든 서버가 Codex 에 그대로 붙는다
- 클라이언트가 늘어도 서버 코드 변경 없음
- 팀 리포에 서버 하나만 두면 도구 선택이 팀 표준을 흔들지 않는다

⚠️ **Codex 설정 파일 위치.** macOS·Linux 는 `~/.codex/config.toml`. Windows 는 `%APPDATA%\codex\config.toml`. 참여자마다 다르니 사전 확인.

PART 5

재순위화 · 확장 — 실전 시나리오 · rerank · 확장 5선

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

실전 시나리오 3종 — 개념·심볼·크로스 파일

Claude Code 대화창에서 실제로 써 본다.

① 개념 검색

```
결제 실패 관련
회의록 찾아줘
```

`search(query="결제 실패 관련 회의록", top_k=5)` 호출. Claude 가 상위 결과의 파일 경로를 뽑아 `file://` 리소스를 읽어 원문 확인 후 답변.

② 코드 심볼 찾기

```
인덱서에서 mtime
비교하는 부분 어디야?
```

`search(query="mtime 비교 파일 최신")` 호출. 상위 결과에 `indexer.py` 청크. Claude 가 파일 경로와 라인 번호로 응답.

③ 크로스 파일 요약

```
이번 달 회의록 3개 훑고
배포 관련 결정 사항만
정리해줘
```

Claude 가 여러 번 `search` 를 호출한다. 질의를 다양화해 상위 결과를 모아 요약. **LLM 이 쿼리를 재구성.**

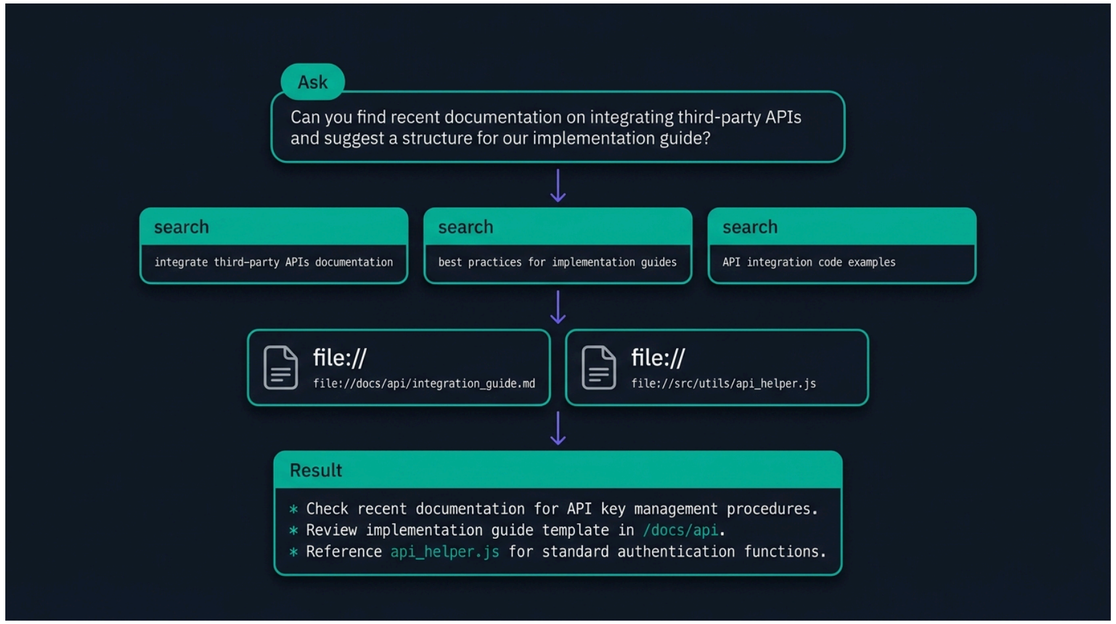
📌 **관찰.** 서버 API 는 단일 `search` 하나. **조합의 지능은 LLM 이 가져간다.** 서버 코드가 얇으니 유지보수가 쉽다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

시나리오 흐름 — 자연어 → 툴 → 리소스 → 답변

📌 클라이언트 내부 흐름

1. **Ask** — 참여자가 자연어로 질문
2. **search 툴 호출** — LLM 판단으로 **여러 번 재질의**
3. **file:// 리소스 로드** — 원문이 필요한 파일만
4. **Result** — 마크다운 답변 생성



Ask → search × N → file:// × M → Result

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

재순위화 — 필요와 함정

코사인 유사도 상위 5개가 항상 최선은 아니다. **'SLA 관련 논의' 검색에서 SLA 라는 단어가 지나가는 말로 한 번 언급된 청크가 상위에 뜨는 식.**

> 🧠 **왜** — 임베딩은 단어의 **존재만으로도** 유사도가 오른다. 'SLA' 를 스쳐 언급한 회의록 각주가 상위 5에 섞인다. 재순위화는 이 노이즈를 LLM 의 의미 판단으로 걸어내는 후처리.

```
RERANK_PROMPT = """질의 '{query}' 의 검색 후보 {n}개를 0~10 점으로 평가.
10점 직접 답변 · 5점 부수적 · 0점 무관.
후보 번호·점수·이유(한 문장) 를 JSON 배열로 반환.
--- 후보 ---
{candidates}"""

def rerank(query: str, results: list[dict]) → list[dict]:
    from anthropic import Anthropic
    client = Anthropic()
    candidates = "\n\n".join(
        f"[{i}] {r['path']}\n{r['text'][:500]}"
        for i, r in enumerate(results))
    resp = client.messages.create(
        model="claude-haiku-4-5", max_tokens=1000,
        messages=[{"role": "user",
                   "content": RERANK_PROMPT.format(
                       query=query, n=len(results), candidates=candidates)}])
    # 점수 파싱 · 재정렬 (파싱 실패 시 원 순위 유지)
```

⚠️ **이 세션은 재순위화를 서버가 아닌 클라이언트에 맡긴다.** Claude Code · Codex 가 검색 결과를 자체 후처리하므로, 서버에서 또 하면 **이중 처리.** 위 함수는 참조용.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

확장 아이디어 5선 — 같은 뼈대에서 방향만

📓 Notion 통합

Notion API 로 페이지·데이터베이스를 주기적으로 pull.

파일 대신 페이지 ID 를 소스로. **청킹은 마크다운 방식 재사용.**

📑 PDF 지원

`pypdf` 로 텍스트 추출 → `chunk_text`.

페이지 번호를 청크 메타에 저장하면 인용 정확도가 오른다.

🎙️ 오디오 노트

Whisper API 로 트랜스크립트 → 텍스트로 인덱싱.

결과에 원본 오디오 **타임스탬프**까지 실을 수 있다.

👥 팀 지식베이스

개인용 sqlite → 팀용 **pgvector**.

임베딩·검색 로직은 그대로, **저장소만 교체**. 인증 계층 추가.

🔀 하이브리드 검색

BM25 (키워드) + 임베딩 (의미) 조합.

정확한 고유명사와 개념 검색을 동시에.

🧩 공통 확장 규칙

1. **소스 로더**만 바꾼다 (파일 → API · DB · 오디오)
2. 청킹 전략을 소스에 맞게
3. 저장소는 규모에 맞게
4. 검색 로직 · MCP Tool 정의는 재사용

📌 뼈대가 같기 때문에 **두 번째 인덱서부터는 훨씬 빠르게 만든다.** 한 번 구현해 본 경험이 그대로 자산이 된다.

PART 6

정리 · 코스 마무리

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

만든 것 · 얻은 것 — 300줄의 네 가지 자산

85분 동안 만든 것은 `indexer.py` · `searcher.py` · `watcher.py` · `server.py` **네 파일**. 실제 코드 **300줄 남짓**. 그 300줄로 다음을 얻었다.

🔧 개인용 RAG 파이프라인

인덱싱 · 유사도 검색 · 실시간 갱신 완결.

🌐 MCP 서버로 노출된 검색

Claude Code · Codex 어디서든 **자연어로 호출**.

👥 팀 이관 가능한 뼈대

소스 로더 · 저장소만 교체하면 **팀 지식베이스**로.

🗣️ UI 없는 검색 경험

검색 UI 제작 시간을 통째로 아꼈다.

📌 재사용 지점 4가지

자산	어디에 다시 쓰나
청킹 전략 (문서 유형별)	Notion · PDF · 이메일 인덱서
sqlite BLOB 임베딩 패턴	개인 규모의 어떤 벡터 저장소 프로토타입
MCP Tool + Resource 조합	검색 → 원문 접근이 필요한 어떤 서버
watchdog 실시간 인덱싱	로그 모니터 · 파일 동기화 · 백업 트리거

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

코스 완주 · 수고 많으셨습니다

🏁 이 코스에서 얻은 것

AI 도구는 도구다. 도구의 힘은 도구 자체가 아니라 **도구를 감싸는 규약과 확장 코드**에서 온다. 프롬프트 · 컨텍스트 · 하네스 · MCP — 네 축은 모두 '도구를 어떻게 확장할 것인가'에 대한 답이다.

🧰 실전 조작 감각

Claude Code · Codex 두 도구를 실전 수준으로 조작한다.

명령어 · 모드 · CLI · 승인 정책까지.

📐 하네스 직접 구성

`CLAUDE.md` · `AGENTS.md` · hooks · subagent 를 직접 작성하고 적용했다.

🧩 MCP 서버 3종 실제 조립

다음에 만들 서버는 **훨씬 빠르게 조립**할 수 있다.

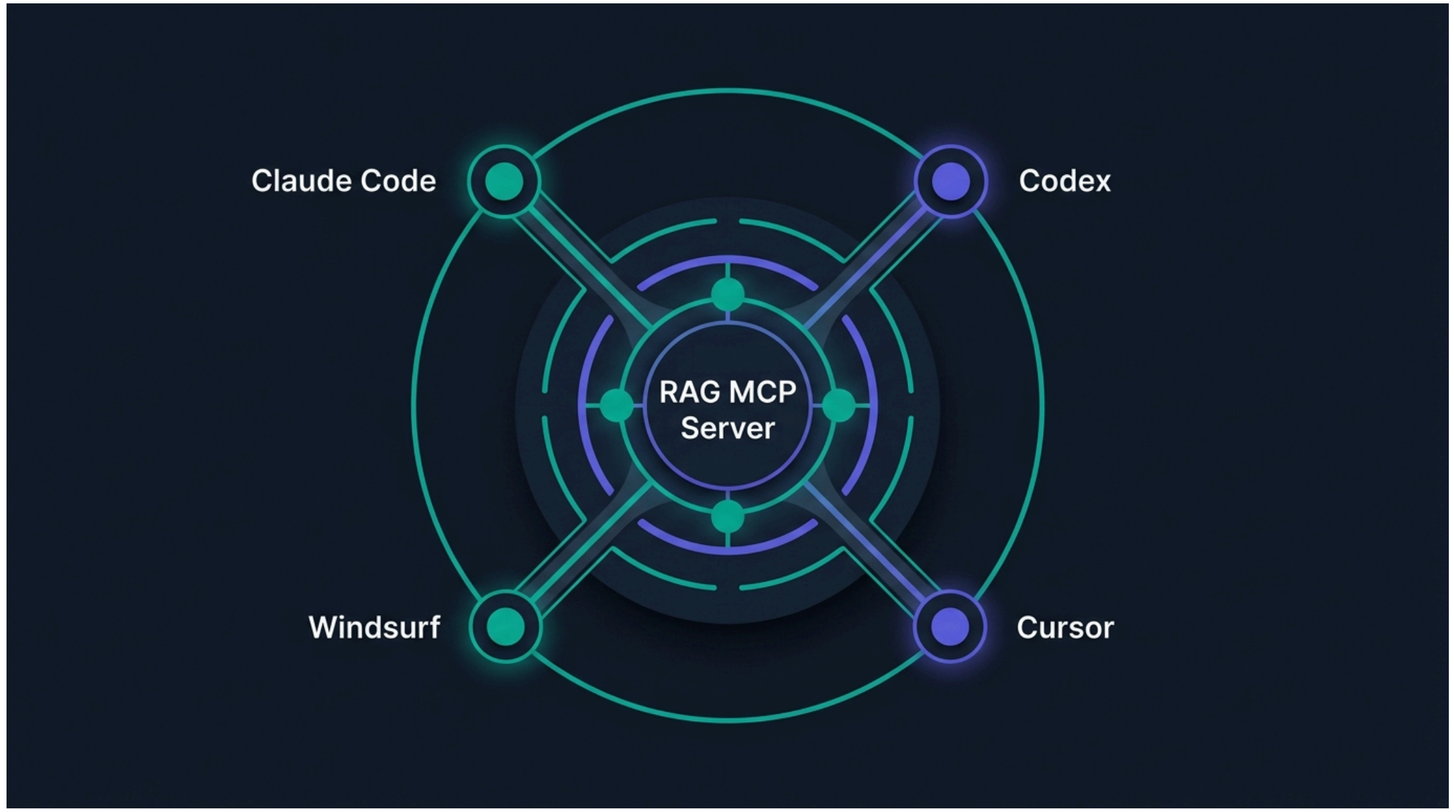
🌱 바이브코딩 방법론

커밋 단위 · 테스트 우선 · 회귀 방지 · 팀 규약을 **손에 잡히는 형태**로 정리했다.

오늘 만든 300줄의 RAG 서버가 실무에서 계속 쓰이게 만드는 것은 코드 실력이 아니라 **실험 습관**이다. '이 검색을 매일 쓰는 문서 폴더에 붙여 보자' 는 한 번의 결정이 다음 실험으로 이어지고, 확장은 그 과정에서 자연스럽게 따라온다.

여기까지 오신 노력, 정말 수고 많으셨습니다. 🎉

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG



코스 완주 · 수고 많으셨습니다

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

하네스 다음 — 에이전트 시스템 엔지니어링

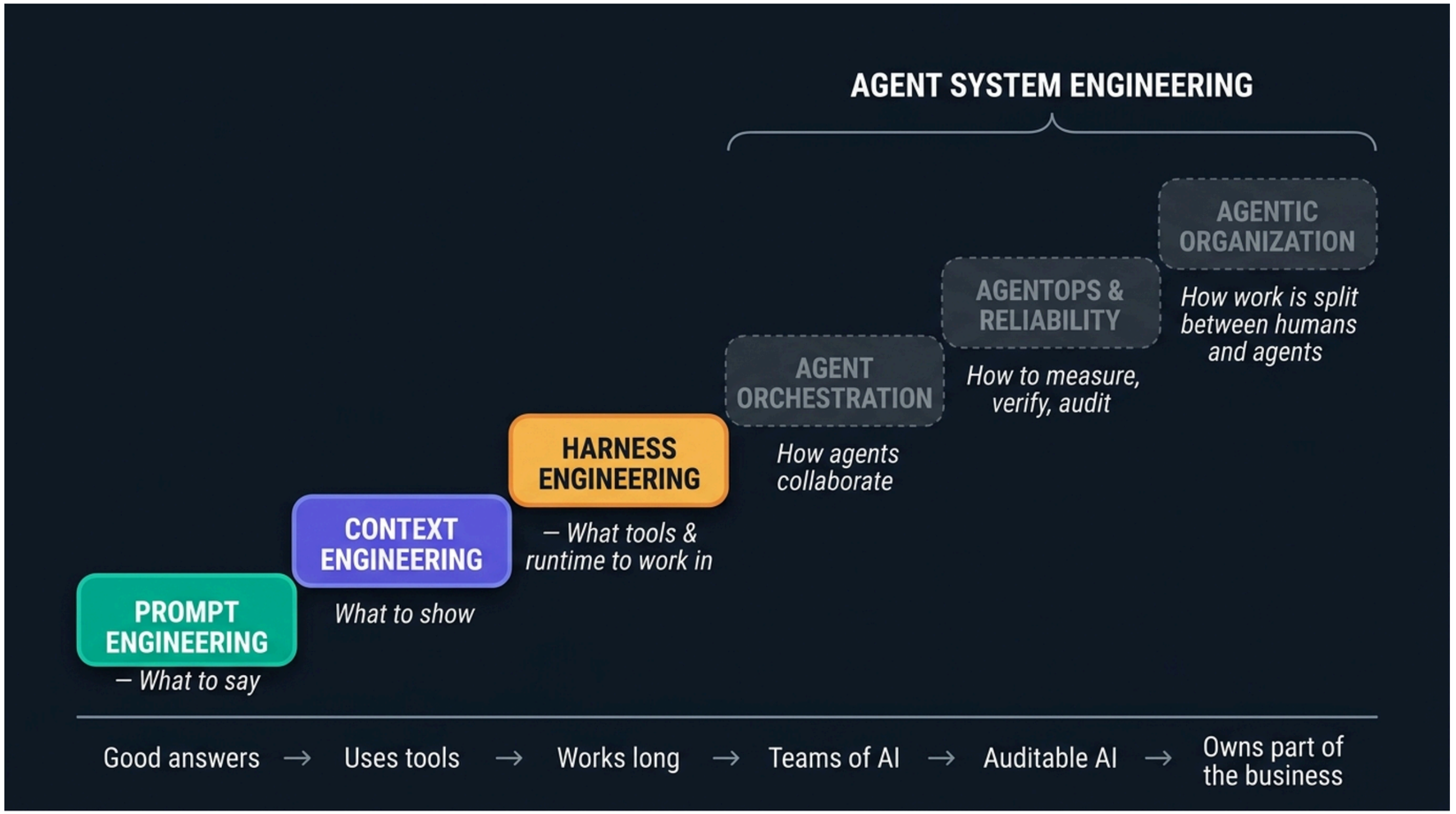
하네스 다음이 무엇인지에 대해서는 아직 하나의 용어가 정착하지 않았다. 여러 흐름으로 분화하는 중이고, 각 진영이 서로 다른 이름을 붙이고 있다. 한 우산으로 묶으면 **에이전트 시스템 엔지니어링** 이다.

📐 정의

여러 AI 에이전트와 도구 · 데이터 · 권한 · 평가 · 검증, 그리고 사람의 승인 절차를 **하나의 신뢰 가능한 업무 시스템으로 설계하는 기술.**

프롬프트 → 컨텍스트 → 하네스 — 이 코스가 채운 세 축. **→ 에이전트 오케스트레이션 → 운영 · 신뢰성 → 에이전틱 조직** — 그 너머로 이어지는 세 축. 앞의 세 축은 그 자체로 완결이 아니라 더 긴 사슬의 앞부분이다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG



선명한 세 계단은 이 코스가 밟은 자리 · 점선 세 계단은 아직 형성 중인 자리

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

다섯 갈래 ①~③ — 나누고 · 지켜보고 · 검증한다

🧭 에이전트 오케스트레이션

하네스가 에이전트 하나에 도구를 달아 주는 일이라면, 그다음은 **역할별로 나누고 조정하는** 일이다. 관리자 에이전트 아래 조사 · 코딩 · 테스트 · 보안검토 · 문서화 에이전트를 둔다.

핵심 문제 — 작업 분배 · 병렬과 순차의 선택 · 결과 충돌 해소 · 실패 재시도 · **비용과 토큰 상한**.

📊 AgentOps · 운영

Trace (어떤 판단과 도구 호출을 했나) · Evaluation (정확했나) · Observability (어디서 실패했나) · 비용과 지연 모니터링 · 회귀 테스트.

계보로 보면 DevOps → MLOps → LLMOps → **AgentOps** 의 연장선이다.

🛡️ 신뢰성 · 검증

에이전트는 답만 만들지 않는다. 이메일 발송 · 코드 수정 · 서버 배포 · DB 변경 · 결제를 **실제로 실행한다.** 그래서 "대체로 잘한다" 는 운영 기준이 되지 못한다.

구조 — 작업 에이전트 → 결과 → 검증 에이전트 → 규칙 · 테스트 · 정책 검사 → 승인 또는 실행.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

다섯 갈래 ④~⑤ — 잇고 · 배분한다

🔌 상호운용성 · 프로토콜

MCP 가 `에이전트 ↔ 도구 · 데이터` 를 잇는다면, **A2A (Agent2Agent)** 는 `에이전트 ↔ 에이전트` 를 잇는다.

서로 다른 회사 · 플랫폼의 에이전트가 상대를 발견하고 · 능력을 설명하고 · 업무를 위임하고 · 상태를 공유하고 · 인증과 권한을 주고받아야 한다.

A2A 는 2025.06.23 Linux Foundation 프로젝트로 이관됐고 AWS · Cisco · Google · Microsoft · Salesforce · SAP · ServiceNow 가 참여한다.

📌 **이 코스에서 직접 만든 MCP 서버가 그 한쪽 축이다.**

🏛️ 에이전틱 조직 · 업무 설계

기술보다 **조직 업무의 재설계**에 가깝다. 사람이 목표와 정책을 설정하고 → 관리 에이전트가 업무를 분해하고 → 병렬로 수행하고 → 검증하고 → 사람이 중요 판단과 승인을 맡는다.

요점은 사람을 에이전트로 **교체하는 것이 아니라 배분을 설계하는 것** 이다.

설계해야 할 것 — 무엇을 완전 자동화할지 · 무엇을 AI 초안으로 두고 사람이 손볼지 · 무엇에 사람 승인을 강제할지 · **실패했을 때 책임은 누가 지는지** · 권한 범위를 어디까지 열지.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

커뮤니티 생태계 — 기본기를 프레임워크로 엮다

기본기 다음, 남들은 이 확장들(Skills · CLAUDE.md · subagent)을 어떻게 프레임워크로 묶었나. 각기 다른 실패 모드를 겨냥한 셋을 본다.

🧭 gstack

Garry Tan · Y Combinator

성격 — 스캐폴딩 · 거버넌스

겨냥하는 실패: 열린 맥락에서의 표류. 스택을 미리 정하고 역할별 모드(엔지니어 · 디자이너 · QA · 보안)로 경계 지어진 맥락에서 추론하게 한다.

🩹 GSD

긴 세션의 지속

성격 — 안정화

겨냥하는 실패: 긴 세션에서 하던 일을 잊음. 맥락을 붙들어 작업을 끝까지 이어 간다.

✅ Superpowers

검증 우선

성격 — 규율 · 검증

겨냥하는 실패: 성급하게 그럴듯한 오답을 냄. 검증 단계를 강제해 확인 없이 넘어가지 못하게 한다.

📌 커뮤니티 한 줄 — **"gstack 은 생각하고 · GSD 는 안정화하고 · Superpowers 는 실행한다."** 새 기술이 아니라 이 코스에서 배운 Skills · CLAUDE.md · subagent 를 조합 · 규율화한 것이다. 기본기를 알면 이런 프레임워크를 읽고 · 고르고 · 자기 것으로 만들 수 있다.

⚠️ **주의 —** 빠르게 변하는 커뮤니티 생태계다. 여기 셋은 고정 목록이 아니라 예시이며, 셋 다 Anthropic 공식이 아니라 커뮤니티 · 개인 프로젝트다. 설치법과 최신 상태는 각 GitHub 저장소에서 직접 확인한다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

한 겹 더 — 어떻게 쓰나 · 주요 기능 · repo

겉모습은 저마다 다르다. 안을 뜯어보면 Skills · 플러그인 · subagent · worktree · TDD 의 조합이다.

🧭 gstack

역할 기반 스프린트

스킬을 이어 흐르는 한 스프린트 — `/office-hours` → `/plan-ceo-review` → `/plan-eng-review` → 구현 → `/review` → `/qa` → `/ship` → `/retro`.

대표 — `/browse`(실브라우저) · `/cso`(보안 OWASP+STRIDE) · `/codex`(2차 의견)

설치 — `~/.claude/skills/` 에 스킬 clone

`github.com/garrytan/gstack`

✅ Superpowers

자동 발동 · 강제 TDD

별도 명령 없이 자동 발동한다. 7단계 강제 워크플로 — 테스트 전에 쓴 코드는 삭제해 TDD 를 건너뛸 수 없게 만든다.

설치 — 공식 마켓플레이스 `/plugin install superpowers@claude-plugins-official`

`github.com/obra/superpowers`

🩹 GSD

컨텍스트 엔지니어링 · 스펙 주도

메타 프롬프팅 + 스펙 주도 개발. 전용 서브에이전트 · 태스크마다 새 컨텍스트 · 자동 검증으로 규모가 커져도 안정적으로 만든다.

워크플로 — Discuss → Plan → Execute → Verify → Ship. 긴 세션 컨텍스트 로트를 새 컨텍스트 서브에이전트로 해결.

설치 — npx 설치기 `npx @opengsd/gsd-core@latest`

`github.com/open-gsd/gsd-core`

원조 `gsd-build/get-shit-done` 은 2026년 6월 아카이브 → `open-gsd/gsd-core` 로 이전 — 생태계는 빠르게 변한다.

📌 설치 진입은 셋 다 다르다 — 스킬(gstack) · 공식 플러그인(Superpowers) · npx(GSD). 그 안은 Skills · 플러그인 · subagent · worktree · TDD. 겉은 달라도 안은 같은 재료다.

출처: MCP Resource/Tool | sqlite BLOB | OpenAI/Voyage 임베딩 | RAG

코스와의 연결 — 여기까지 만들었다 · 여기서부터 이어진다

다섯 갈래는 먼 이야기가 아니다. 이 2일 동안 손으로 만져 본 것들이 그 입구다.

이 코스에서 직접 만든 것	이어지는 입구
subagent 위임 · 워크트리 병렬 세션	**에이전트 오케스트레이션**
hooks · permissions · 승인 게이트	**신뢰성 · 검증 · 거버넌스**
MCP 서버 3종 (도구 · 데이터 연결)	**상호운용성 프로토콜**
CI 워크플로 · PR 자동 리뷰어	**AgentOps · 운영**

🎯 가져갈 문장 하나

AI 가 일을 할 수 있는가보다,
AI 가 한 일을 증명할 수 있는가가 중요해진다.

바이브코딩은 개인 생산성 기술로 시작해 **시스템 설계 기술**로 넘어가는 중이다. 이 코스가 채운 것은 그 앞의 세 축이고, 그 세 축이 없으면 뒤의 세 축은 세우지 못한다.