

WEEK 03

함께하는 AI 에이전트 개발

3주차: 지식 베이스와 메모리 시스템 구축



PROJECT GOAL

리서치 어시스턴트



DURATION

180 Mins (3h)



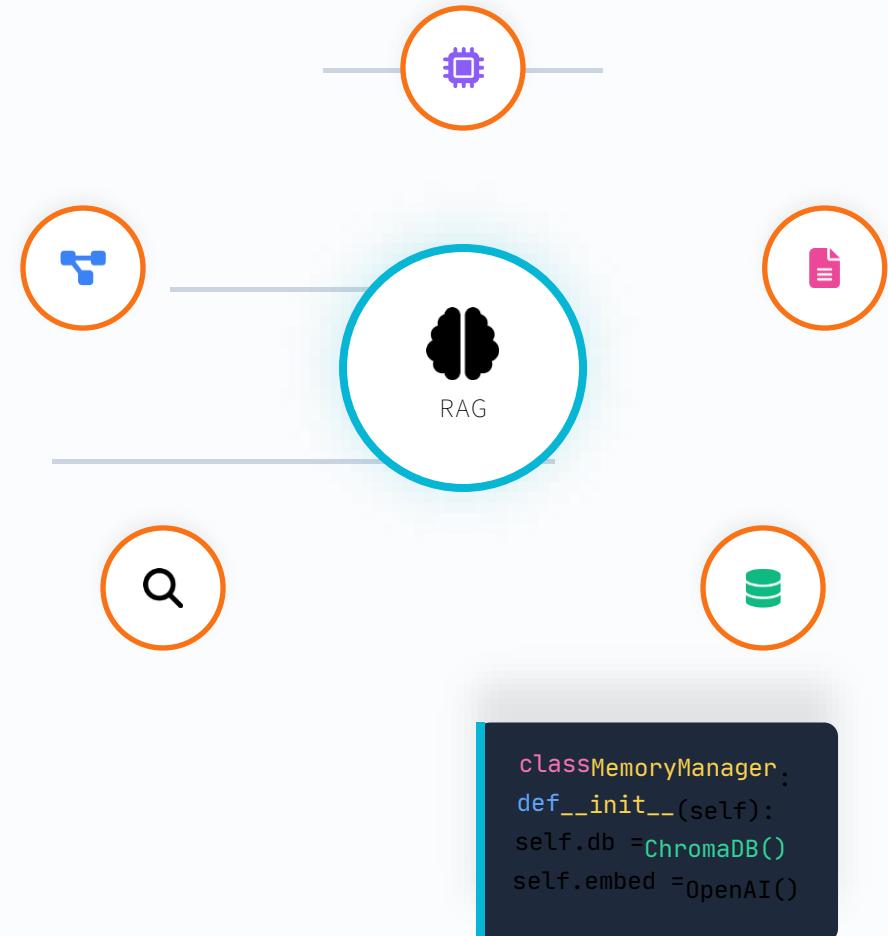
CORE STACK

Python, OpenAI API



INSTRUCTOR

Thomas Moon



Today's Agenda & Roadmap

이번 시간의 학습 목표와 전체 과정을 조망합니다.

TODAY: WEEK 03

지식 베이스와 메모리 시스템 구축

- 01 지난 주 복습 및 개선
정보 수집 에이전트의 한계점 분석

- 02 이론: 메모리 문제와 RAG
Vector DB, 임베딩, 검색 증강 생성 개념

- 03 실습 Part1: 벡터 데이터베이스 구축
ChromaDB 설정 및 문서 임베딩 저장

- 04 실습 Part2: SearchAgent에 메모리 통합
메모리 기반 검색 및 중복 제거 구현

- 05 실습 Part3: 세션 메모리 관리
대화 맥락 유지 및 장기 기억 전략

- 06 3주차 과정 마무리 및 정리
핵심 기능 점검 및 다음 주 과제 안내

5-WEEK MASTER ROADMAP



대화형 AI 코어 구축

LLM 연동, 프롬프트 엔지니어링, 페르소나 설계



정보 수집 에이전트 구축

웹 검색 API 연동, Function Calling, 도구 활용



지식 베이스와 메모리 (RAG)

Vector DB 구축, 임베딩, 장기 기억 관리



자율 실행 AI 에이전트

ReAct 패턴, 자율 계획 및 추론, Self-reflection



멀티 에이전트 & 리포트

협업 시스템 구축, 리포트 자동 생성, 최종 완성

01

SECTION

REVIEW

지난 주 복습 및 개선

2주차 성과 점검과 개선 포인트 확정

KEY TOPICS

✓ 정보 수집 에이전트 기능 점검

⌚ 중복 검색과 비용 효율성

🧠 Context Loss (문맥 손실) 문제

💾 RAG (검색 증강 생성) 도입 배경

지난 주 복습

✓ 지난 주 완성 기능

- ✓ Tavily API 연동 및 웹 검색 기능 구현
- ✓ Function Calling을 통한 도구(Tool) 사용
- ✓ 검색 결과 파싱 및 자연어 요약 생성
- ✓ 대화 흐름 내 검색 기능 통합

⚠ 현재 시스템의 한계점

- ✗ 매번 새로 검색 (이전 조사 결과를 잊음)
- ✗ 중복 검색으로 인한 API 비용 및 시간 낭비
- ✗ 검색 품질 개선 불가 (학습 없음)

| 1주차 vs 2주차 비교

구분	1주차 (대화형 AI)	2주차 (정보 수집)
핵심 클래스	ConversationManager	ConversationManager +SearchAgent
API 연동	OpenAI API만	OpenAI +Tavily API
도구 사용	없음	Function Calling (search_web)
정보 출처	LLM 학습 데이터	실시간 웹 검색
응답 방식	직접 생성	검색 → 파싱 → 요약
데이터 흐름	User → LLM → User	User → LLM → Tool → LLM → User
완성 기능	자연스러운 대화	근거 있는 정보 제공

🌀 2주차까지의 핵심 성과

- ✓ 대화 기능(1주차) + 정보 수집 기능(2주차) = 실용적인 리서치 어시스턴트 완성
- ✓ Function Calling으로 LLM이 도구를 스스로 판단하여 사용하는 자율성 확보
- ✓ 검색 결과 재주입 패턴으로 근거 있는 응답 생성 가능

Troubleshooting & 개선

자주 발생하는 이슈 및 해결 방안

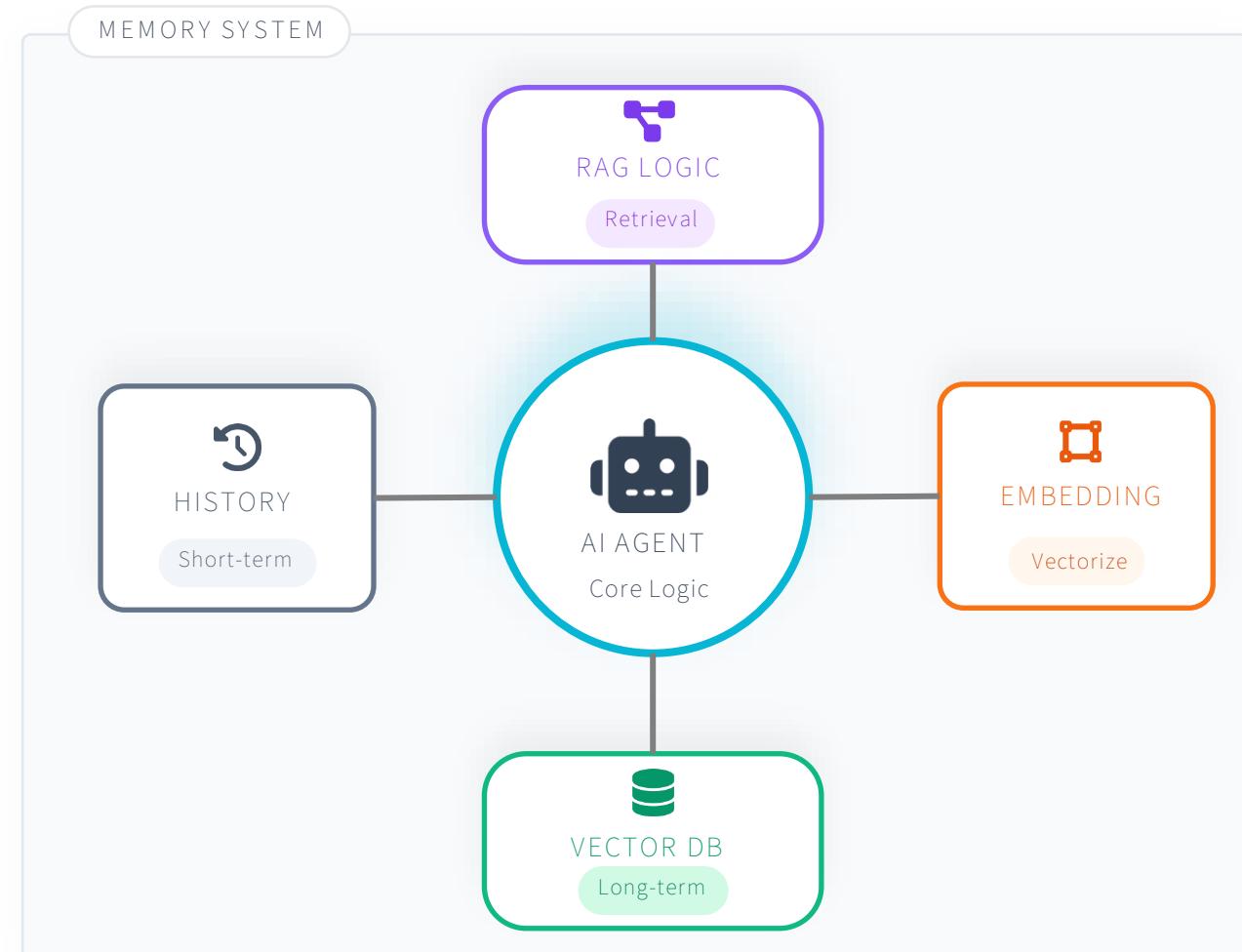
Q&A

이번 주 학습 목표

과거 리서치를 영구 보관하고, 새로운 질문에 기존 지식을 활용하는 학습하는 AI를 만듭니다.

- RAG & 벡터 DB 이해
검색 증강 생성 개념과 벡터 데이터베이스의 원리 학습 [THEORY]
- 임베딩(Embedding) 기술
텍스트를 벡터로 변환하여 의미적 유사도를 계산하는 방법 습득 [PRACTICE]
- 장기 메모리 시스템
Chroma DB를 활용하여 영구적인 지식 저장소 아키텍처 구현 [ARCHITECTURE]
- 검색 품질 최적화
메모리와 웹 검색 결과를 통합하여 중복을 줄이고 품질 향상 [OPTIMIZATION]

기억하고 학습하는 리서치 어시스턴트 완성



02

SECTION

THEORY

AI FUNDAMENTALS

이론: 메모리 문제와 RAG 아키텍처

AI 에이전트의 기억력 한계를 극복하기 위한 핵심 기술인 임베딩, 벡터 데이터베이스, 그리고 RAG(검색 증강 생성) 아키텍처를 심층적으로 학습합니다.

KEY TOPICS

왜 메모리가 필요한가?

벡터 데이터베이스 (Vector DB)

임베딩(Embedding) 이해

RAG (검색 증강 생성)

왜 메모리가 필요한가?

The Memory Problem & RAG

SCENARIO: 메모리 없는 AI

👤 "테슬라 전기차 점유율 알려줘"

🌐 (웹 검색 후 답변 완료)

— 3 Days Later —

👤 "테슬라 배터리 기술은?"

❗ 다시 처음부터 검색 시작...

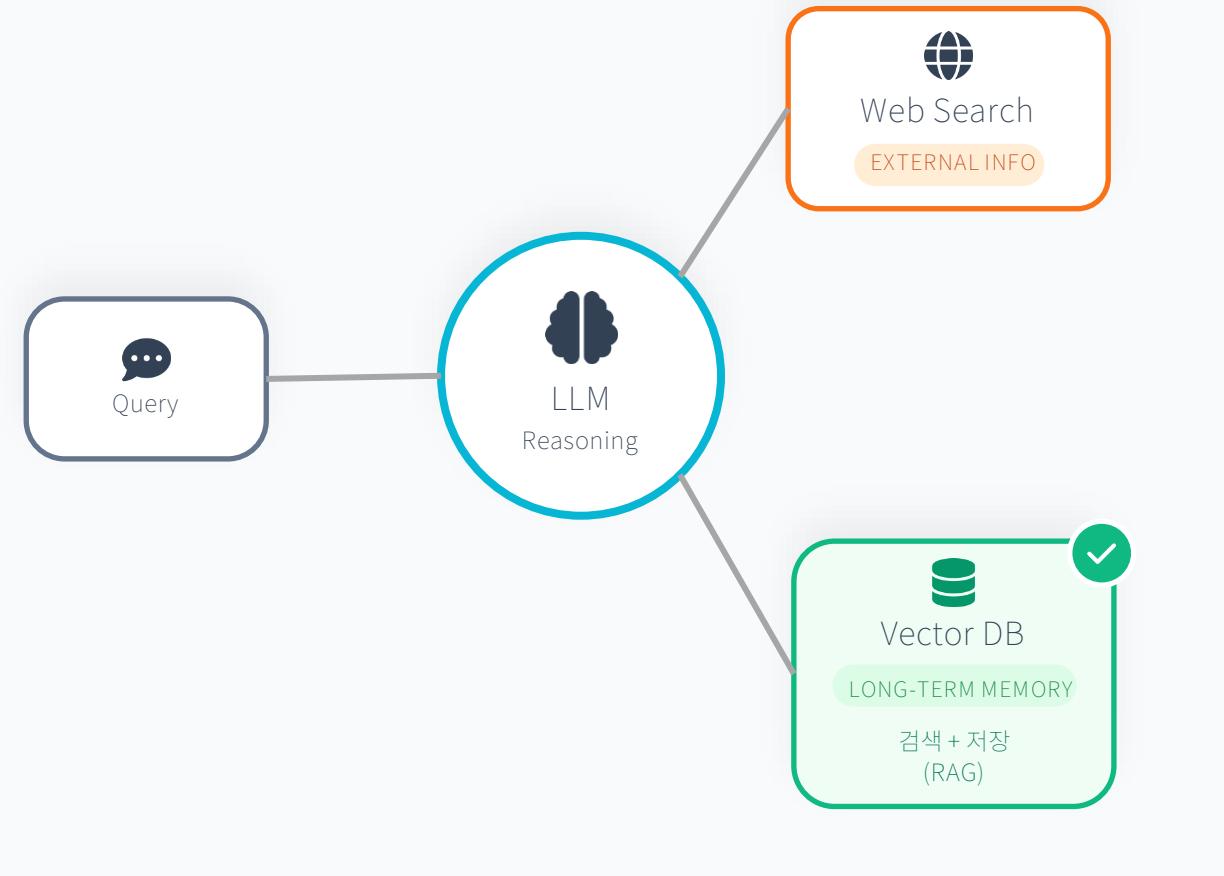
주요 문제점 (Problems)

Context Loss
과거 조사 내용을 완전히 망각함

Inefficiency
중복 검색으로 인한 API 비용/시간 낭비

No Learning
지식이 누적되지 않아 발전이 없음

RAG SOLUTION CONCEPT



RAG 개념

검색 증강 생성이란?

LLM의 **지식 단절(Cutoff)**과 **환각(Hallucination)** 문제를 해결하기 위해, 외부 지식 베이스에서 정확한 정보를 찾아 답변 생성을 보완하는 기술입니다.

1 RETRIEVAL (검색)

사용자 질문과 의미적으로 가장 유사한 문서를 벡터 데이터베이스에서 검색합니다.

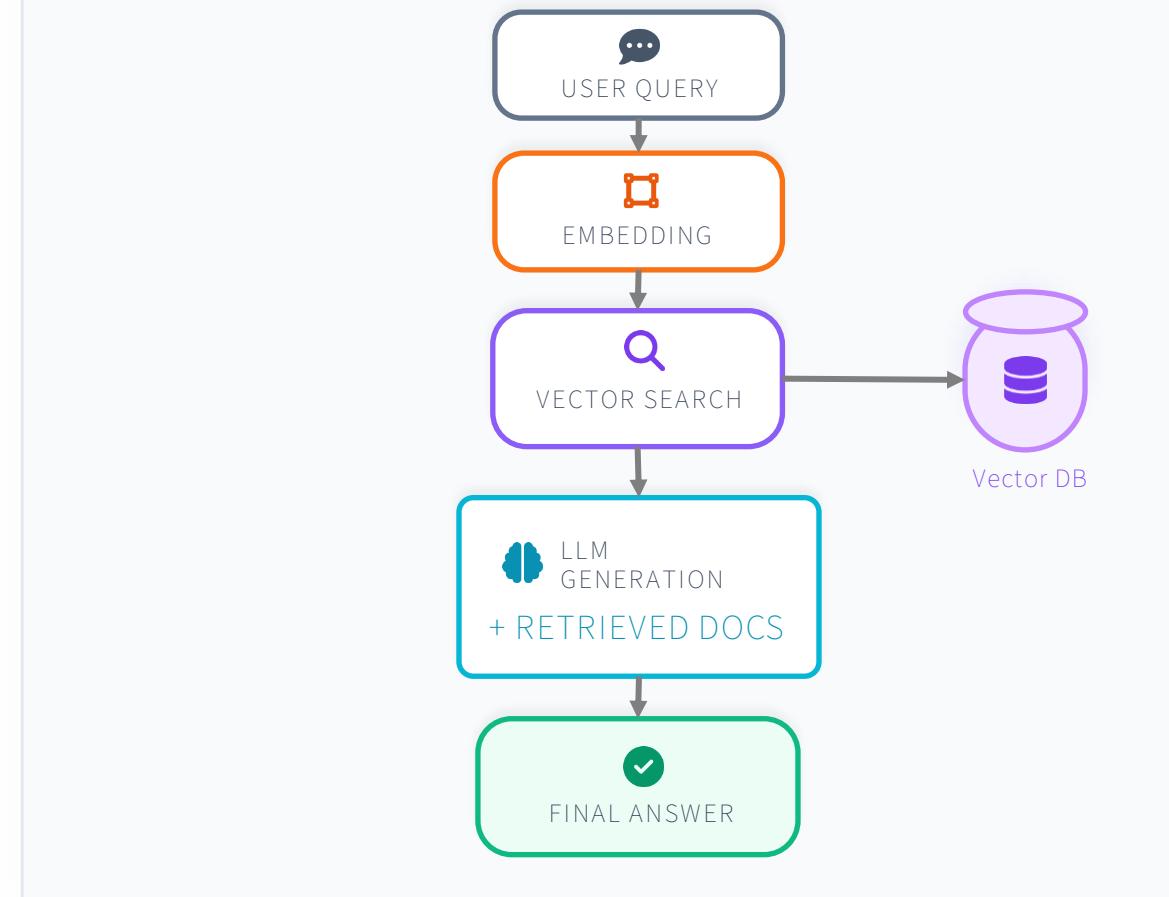
2 AUGMENTATION (증강)

검색된 관련 정보를 프롬프트의 컨텍스트(Context)로 추가하여 LLM에게 제공합니다.

3 GENERATION (생성)

LLM이 증강된 정보를 바탕으로 사실에 입각한 정확한 답변을 생성합니다.

RAG ARCHITECTURE FLOW



임베딩(Embedding) 이해

컴퓨터는 단어의 의미를 직접 이해할 수 없으므로, **숫자 배열(Vector)**로 변환하여 계산합니다.

1. 숫자로 바꾸기 (벡터화)

단어나 이미지는 그 자체로 계산할 수 없습니다. 이를 좌표평면 위의 점인 **숫자 묶음(벡터)**으로 변환합니다.

"사과" → [0.5, 1.2, -0.8]

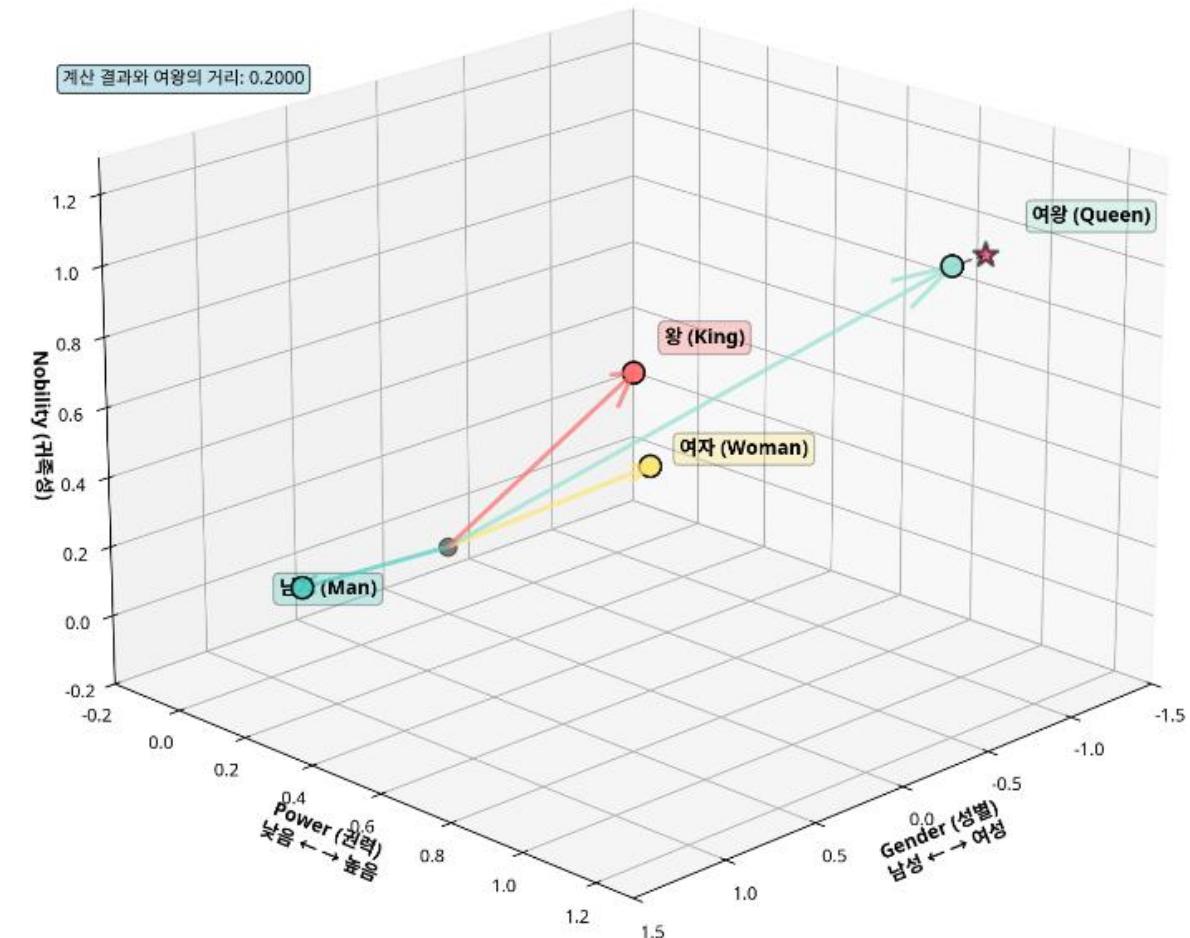
"바나나" → [0.4, 1.1, -0.7]

2. 의미와 관계를 담기

단순 변환이 아니라 "의미가 비슷하면 숫자도 비슷하게" 만듭니다.
이를 통해 단어 사이의 연산을 가능하게 합니다.

$$\text{왕} - \text{남자} + \text{여자} = \text{여왕}$$

단어	성별 (남성 → 여성)	권력 (낮음 → 높음)	고귀함 (평민 → 왎족)	벡터 값 ([x,y,z])
왕 (King)	0.9	1	1	[0.9, 1, 0, 1.0]
남자 (Man)	1	0.1	0.1	[1.0, 0.1, 0.1, 1.0]
여자 (Woman)	-1	0.1	0.1	[-1.0, 0.1, 0.1, 1.0]
여왕 (Queen)	-0.9	1	1	[-0.9, 1, 0, 1.0]



벡터 데이터베이스

의미(Semantic) 기반 검색을 위한 저장소

↔ 검색 방식 비교

Traditional DB (키워드 매칭)

Query: "전기차 회사"

Result: "전기차" OR "회사" 단어가 정확히 포함된 문서만 검색

Vector DB (의미 검색)

Query: "전기차 회사"

Result: "테슬라", "BYD", "리비안" 등의 의미상 관련된 문서검색

❶ 핵심 차이점: 사용자가 정확한 단어를 몰라도 의도(Intent)와 맥락(Context)에 맞는 정보를 찾을 수 있습니다.

주요 벡터 DB 솔루션

Chroma 로컬, 경량, Python 친화적

Pinecone 클라우드 관리형(Full Managed), 스케일링 용이

Weaviate 오픈소스, 프로덕션 레벨 가능

FAISS Facebook 개발, 초고속 검색

✓ 우리의 선택: Chroma DB

✓ 설치 간단 (`pip install chromadb`)

✓ 파일 기반 (별도 서버 구축 불필요)

✓ 개발 및 테스트 환경에 최적화

메모리 아키텍처

리서치 어시턴트 메모리 설계

3가지 메모리 유형

효율적인 정보 처리를 위한 계층적 메모리 구조

Short-term Memory

현재 대화 세션 (messages[])

사용자와의 즉각적인 문맥을 유지하며, 대화가 종료되면 휘발되는 단기 기억입니다.

Long-term Memory

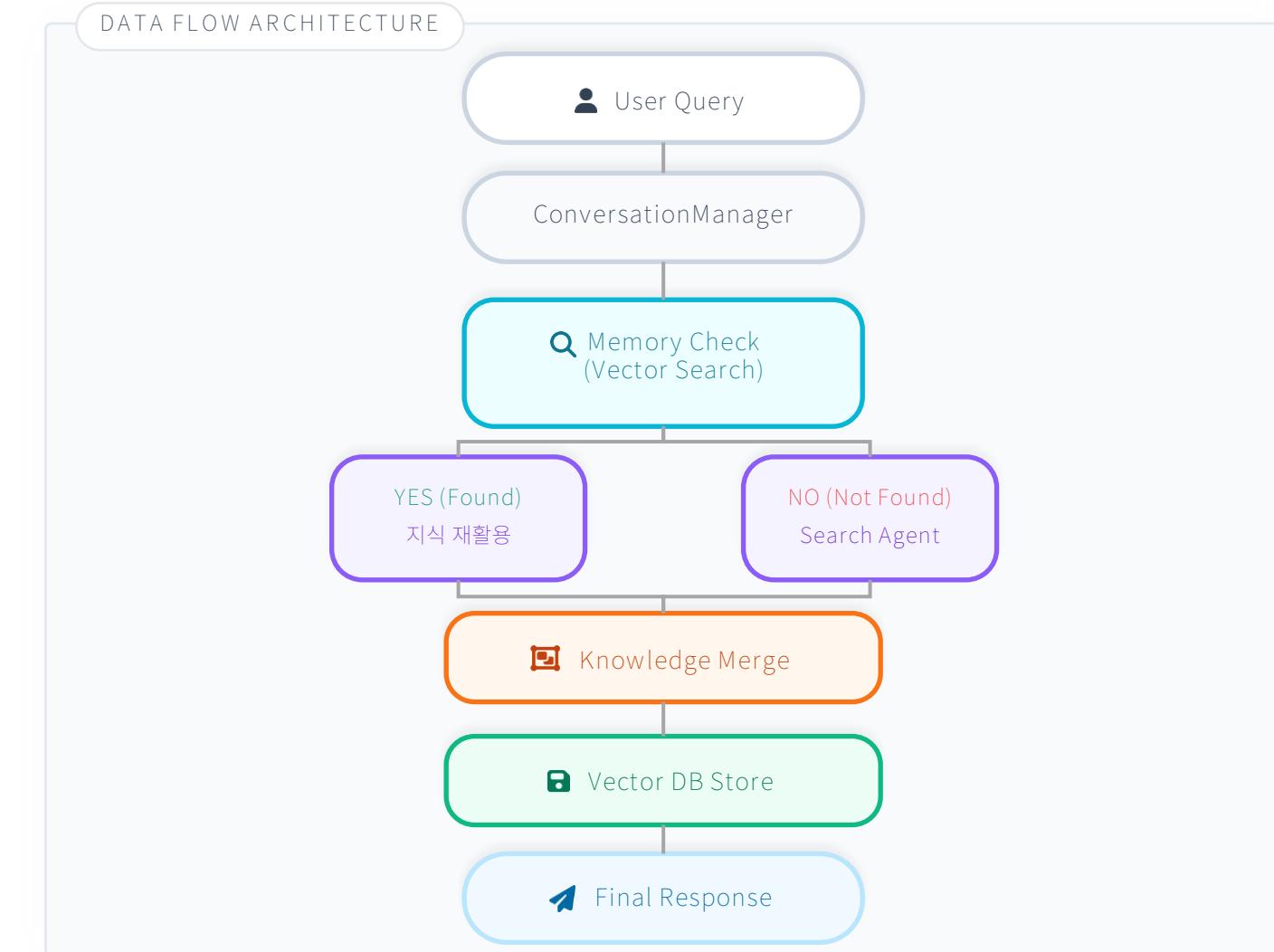
벡터 DB (Chroma)

모든 과거 리서치 내용을 영구 저장하여, 언제든 의미 기반 검색으로 재 활용 가능한 장기 기억입니다.

Summary Memory

대화 요약 (Optimized)

긴 대화 히스토리를 압축 요약하여 LLM 토큰 비용을 절약하고 핵심 맥락만 유지합니다.



03

SECTION

SETUP PREREQUISITES

실습 준비

본격적인 실습에 앞서 ChromaDB 등 필수 라이브러리를 설치하고 개발 환경을 점검합니다. 3주차 프로젝트의 전체 구조와 구현 로드맵을 파악하여 효율적인 실습을 준비합니다.

KEY TOPICS

환경 설정 및 패키지 설치

프로젝트 구조 점검

3주차 구현 로드맵

Part 1: 벡터 DB 구축 목표

환경 설정 및 테스트

▣ 의존성 패키지

requirements.txt에 chromadb 패키지를 추가

```
requirements.txt
# 3주차 추가 패키지
chromadb>=0.4.0
```

▷ 가상환경 활성화 후, 의존성 패키지 업데이트

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

> pip install -r requirements.txt

⚙️ 설치 완료 테스트

python -c "import chromadb; print('✓ chromadb 설치 완료')"

프로젝트 구조 확인

3주차 추가 파일 및 변경 사항 점검

```

ai_research_assistant/
  src/
    __init__.py
    conversation_manager.py
    search_agent.py
    memory_manager.py NEW
    utils/
      __init__.py NEW
      embeddings.py NEW
    tools/
      web_search.py
  config/
    settings.py
    prompts.py
  data/
    chroma_db/ NEW
    conversations/ NEW
  tests/
    test_memory_manager.py NEW
  .env
  main.py
  requirements.txt

```

주요 파일 역할 및 상태

파일명	역할	상태
src/memory_manager.py	메모리 관리 핵심	Part 1
src/utils/embeddings.py	임베딩 생성	Part 1
data/chroma_db/	벡터 저장소	Part 1
data/conversations/	대화 백업	Part 1
src/search_agent.py	검색 + 메모리	Part 2
src/conversation_manager.py	대화 + 메모리	Part 3
main.py	전체 통합	Part 3

기존 파일 수정 계획

Part 2: Search Agent

- 메모리 기반 검색 로직 추가
- 검색 결과 메모리 자동 저장

Part 3: System Integration

- ConversationManager 메모리 로직 통합
- main.py 전체 통합

3주차 전체 작업 로드맵

단계별 작업 분포

- 1단계: 기반 ■ 2단계: 핵심
- 3단계: 고급 ■ 4단계: Search
- 5단계: Conv ■ 6단계: 통합



1단계

Part 1 시작

폴더 및 기반 구축

- ✓ 1.data/chroma_db/폴더 생성
- ✓ 2.data/conversations/폴더 생성
- ✓ 3.src/utils/폴더 및 __init__.py 생성
- ✓ 4.tests/폴더 생성
- ✓ 5.EmbeddingGenerator 클래스 구현
- ✓ 6.MemoryManager 기본 구조 생성

2단계

Part 1 계속

메모리 핵심 기능

- ✓ 7. MemoryManager.__init__() 완성
- ✓ 8.add_to_memory()문서 저장 기능
- ✓ 9.search_memory()유사도 검색 기능
- ✓ 10.get_all_documents()조회 기능
- ✓ 11.delete_memory()삭제 기능
- ✓ 12.get_statistics()통계 기능

3단계

Part 1 완성

고급 메모리 기능

- ✓ 13. 중복 제거 로직 (check_duplicate)
- ✓ 14. 메타데이터 기반 필터링
- ✓ 15. 메모리 정리 기능 (cleanup_old_memories)
- ✓ 16. 메모리 대시보드 (print_memory_dashboard)
- ✓ 17.tests/test_memory_manager.py 작성

4단계

Part 2

SearchAgent 통합

- ✓ 18. SearchAgent에 memory_manager 추가
- ✓ 19.search_with_memory()메서드 구현
- ✓ 20._save_to_memory()헬퍼 메서드
- ✓ 21._merge_results()결과 병합 로직
- ✓ 22. 출처 추적 (provenance) 기능

5단계

Part 3

ConversationManager 통합

- ✓ 23. ConversationManager에 memory_manager 추가
- ✓ 24.save_search_result_to_memory()메서드
- ✓ 25.save_conversation_to_memory()메서드
- ✓ 26.chat()메서드에 메모리 자동 저장

6단계

Part 3 완성

시스템 통합 및 테스트

- ✓ 27. main.py에 MemoryManager 초기화
- ✓ 28. memory, memory-search 명령어 추가
- ✓ 29. 전체 시스템 통합 테스트
- ✓ 30. 성능 벤치마크 (Before/After)

requirements.txt

Part별 작업 내용 요약

파일 변경 및 구현 상세

00 사전 준비

- MOD requirements.txt
chromadb 패키지 추가
- CHECK .env
OpenAI API 키 확인

01 벡터 DB 구축

- Part 1
- NEW data/chroma_db/
폴더 5개 생성
 - NEW src/utils/embeddings.py
임베딩 생성 엔진
 - NEW src/memory_manager.py
메모리 관리 핵심 클래스
 - NEW tests/test_memory...
메모리 시스템 테스트

02 SearchAgent 통합

- Part 2
- MOD src/search_agent.py
메모리 활용 로직 추가
 - memory_manager 주입
 - search_with_memory
 - _save_to_memory
 - _merge_results
 - provenance (출처)
 - REF src/tools/web_search.py

03 시스템 완성

- Part 3
- MOD src/conversation_mgr.py
대화/검색 자동 저장
 - MOD main.py
초기화 및 명령어 추가
 - memory-search 명령어
 - REF config/prompts.py

FILE STATUS GUIDE

NEW 신규 생성

MOD 기능 수정

REF 참조 파일

CHECK 설정 확인

04

SECTION

HANDS-ON

IMPLEMENTATION

HANDS-ON PART 1

벡터 데이터베이스 구축: 에이전트의 장기 기억을 담당할 핵심 컴포넌트를 설계하고 구현하는 실습 과정입니다.

KEY TASKS

 EmbeddingGenerator 구현

 Chroma DB 저장소 통합

 MemoryManager 클래스 구현

 기본 기능 단위 테스트

Part 1 개요 및 아키텍처

벡터 데이터베이스 구축

① 핵심 학습 목표

- ✓ OpenAI 임베딩 생성 엔진 구현
- ✓ Chroma DB 벡터 저장소 초기화
- ✓ 문서 저장/검색/관리(CRUD) 기능 구현
- ✓ 메타데이터 필터링 및 중복 제거

데이터 처리 및 검색 흐름



</> 주요 클래스 역할

🎯 EmbeddingGenerator 클래스

역할: 텍스트를 숫자 벡터로 변환하는 임베딩 생성 엔진

입력: "테슬라는 전기차 회사입니다" (사람이 읽는 텍스트)
 ↓
 처리: OpenAI API 호출
 ↓
 출력: [0.023, -0.056, 0.089, ..., 0.045] (1536개의 숫자)

🗄️ MemoryManager 클래스

역할: 벡터 데이터베이스 기반 장기 메모리 관리자

저장: "테슬라는 전기차 회사입니다"
 ↓
 변환: [0.023, -0.056, ..., 0.045] (EmbeddingGenerator 사용)
 ↓
 저장: Chroma DB에 벡터 + 텍스트 + 메타데이터 저장
 ↓
 검색: "전기차 회사" 입력
 ↓
 반환: "테슬라는 전기차 회사입니다" (가장 유사한 문서)

텍스트 → 벡터 → 저장 과정

데이터 파이프라인 5단계



💡 핵심 포인트

💡 왜 5단계로 나누었나?

1. 모듈화: 각 단계가 독립적으로 작동
2. 재사용성: 임베딩 캐싱으로 동일 텍스트 재활용
3. 확장성: 메타데이터 추가가 용이
4. 최적화: 단계별 성능 튜닝 가능

시간 및 비용

- 텍스트 전처리: ~0.001초
- 임베딩 생성: ~0.5초 (API 호출) / ~0.001초 (캐시)
- 메타데이터 결합: ~0.0001초
- Chroma DB 저장: ~0.01초
- HNSW 인덱싱: ~0.001초

총 소요 시간: 약 0.5초 (첫 호출) / 0.01초 (캐시 히트)

STEP 1: 텍스트 전처리 상세

</> 입력 데이터:

```
text = "테슬라는 2003년에 설립된 전기차 회사입니다"  
metadata = {"source": "web_search", "category": "company"}
```

</> 결과 데이터 (메타 데이터 포함):

```
{  
    "document_id": "a3b2c1d4-e5f6-7890-abcd-  
ef1234567890",  
    "text": "테슬라는 2003년에 설립된 전기차 회사입니다",  
    "metadata": {  
        "source": "web_search",  
        "category": "company",  
        "timestamp": "2024-01-26T15:30:45.123456",  
        "text_length": 24  
    }  
}
```

</> 처리:

```
# 1.1 공백 제거 및 검증  
text = text.strip()  
if not text:  
    raise ValueError("텍스트가 비어있습니다")  
  
# 1.2 문서 ID 생성 (UUID4)  
import uuid  
document_id = str(uuid.uuid4())  
# 결과: "a3b2c1d4-e5f6-7890-abcd-ef1234567890"  
  
# 1.3 메타데이터 자동 보강  
from datetime import datetime  
metadata.update({  
    "timestamp": datetime.now().isoformat(),  
    "text_length": len(text),  
    "source": metadata.get("source", "unknown")  
})
```

STEP 2: 임베딩 변환 상세

</> 캐시 확인 (성능 최적화)

```
class EmbeddingGenerator:
    def __init__(self):
        self.cache = {} # 캐시 딕셔너리
        self.cache_hits = 0
        self.total_requests = 0

    def create_embedding(self, text: str) -> List[float]:
        self.total_requests += 1

        # 캐시에 있으면 즉시 반환
        cache_key = hash(text)
        if cache_key in self.cache:
            self.cache_hits += 1
            return self.cache[cache_key]

        # 캐시 미스 → API 호출
        embedding = self._call_openai_api(text)
        self.cache[cache_key] = embedding
        return embedding
```

</> OpenAI API 호출

```
def _call_openai_api(self, text: str) -> List[float]:
    response = self.client.embeddings.create(
        model="text-embedding-3-small", # 모델 선택
        input=text # 입력 텍스트
    )

    # 응답에서 벡터 추출
    embedding = response.data[0].embedding
    return embedding
```

💡 캐싱 효과

첫 번째 호출: "테슬라는..." → API 호출 (0.5초, \$0.00002)
 두 번째 호출: "테슬라는..." → 캐시 반환 (0.001초, \$0)

→ 500배 빠름, 비용 절감!

</> 생성된 벡터 모습:

```
embedding = [
    0.023456, # 차원 1
    -0.056789, # 차원 2
    0.089123, # 차원 3
    0.012345, # 차원 4
    -0.034567, # 차원 5
    ... # ... (1531개 더)
    0.067890, # 차원 1535
    0.045678 # 차원 1536
]
```

총 1536개의 실수 값
각 값의 범위: 약 -1.0 ~ 1.0

임베딩 모델(Embedding Model) 사용

텍스트를 숫자로 변환하는 기술과 모델 선택의 이유

■ 임베딩 모델이란?

사람이 사용하는 언어(텍스트)를 컴퓨터가 이해하고 계산할 수 있도록 **숫자의 나열(벡터, Vector)**로 변환해 주는 기술

❖ TEXT-EMBEDDING-3-SMALL 선택 이유

압도적인 가성비

이전 세대 대비 80% 저렴. 대량의 문서를 처리하는 기업용 서비스 및 스타트업의 비용 부담을 획기적으로 절감.

한국어 및 다국어 능력 강화

영어뿐만 아니라 한국어 데이터 처리 능력이 크게 향상되어 국내 서비스 적용에 유리.

유연한 차원 축소 (Matryoshka)

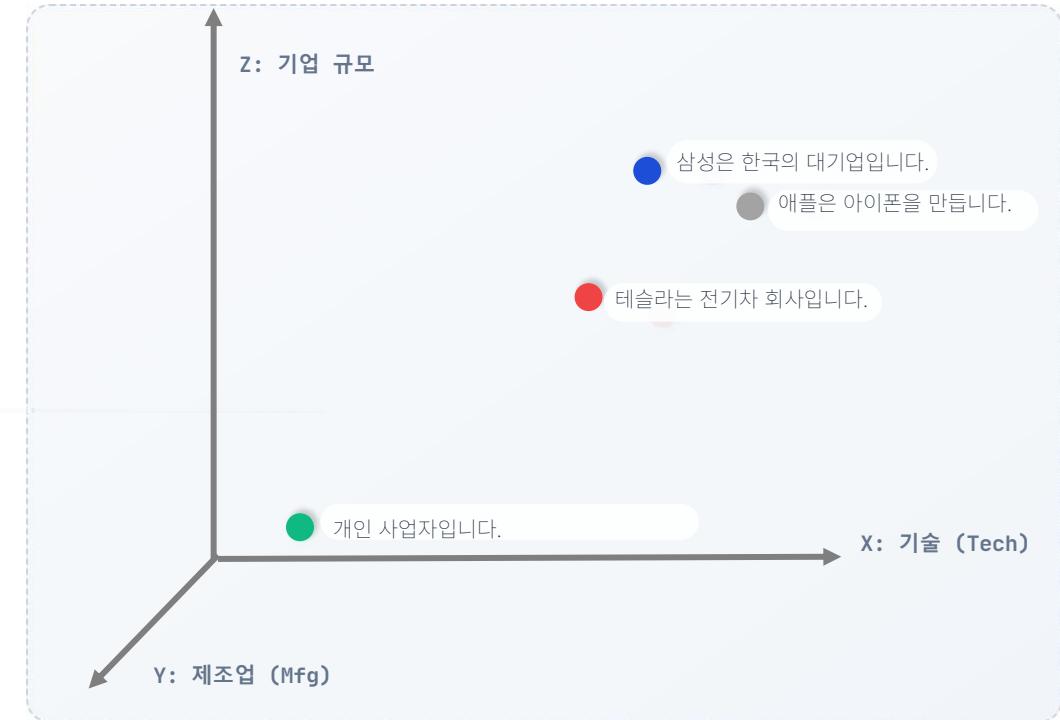
성능 손실을 최소화하며 벡터 크기 축소 가능 (예: 512차원). 저장 용량 절약 및 검색 속도 향상.

개발 편의성과 생태계

OpenAI API로 호출이 쉬우며, LangChain, Pinecone 등 주요 도구와 즉시 연동 가능.

❖ 1536차원 벡터 공간의 이해

1536차원: 각 차원은 텍스트의 특정 의미적 특징을 표현 (충분한 표현력 + 계산 효율)
의미적 거리: 고차원 공간에서 의미가 유사한 텍스트는 가까운 위치에 배치됨



* 가까운 벡터 = 의미가 유사한 텍스트 (이해를 돋기 위한 3차원 시각화)

실제 벡터 비교 방법: 코사인 유사도

벡터 공간에서의 의미적 유사성 측정

코사인 유사도 계산

시각적 이해



같은 방향 (0°)
완전히 같은 의미

1.0



직각 (90°)
관계 없음 (독립적)

0.0



반대 방향 (180°)
정반대 의미

-1.0

Python Implementation

```
import numpy as np
def cosine_similarity(vec1, vec2):
    # 내적 계산
    dot_product = np.dot(vec1, vec2)
    # 벡터 크기 계산
    magnitude1 = np.linalg.norm(vec1)
    magnitude2 = np.linalg.norm(vec2)
    # 코사인 유사도
    similarity = dot_product / (magnitude1 * magnitude2)
    return similarity
```

유사도 비교 예시

v R) BYD라는 전기차 회사입니다

`vec1 = [0.02, -0.05, 0.08, 0.12, -0.03, ...]`

- “리비안은 전기 트럭을 만듭니다.”

`vec2 = [0.03, -0.04, 0.09, 0.11, -0.02, ...]`

`similarity(vec1, vec2) = 0.92` # 매우 유사

- “BYD는 중국의 전기차 기업입니다.”

`vec3 = [0.02, -0.06, 0.07, 0.13, -0.04, ...]`

`similarity(vec1, vec3) = 0.87` # 유사

- “오늘 날씨가 정말 좋습니다.”

`vec4 = [1.02, 1.06, 1.07, 1.13, 1.04, ...]`

`similarity(vec1, "날씨가 좋다") = 0.23` # 다른

코사인 유사도 공식

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$

STEP 4: Chroma DB 저장

Chroma DB 저장 구조

Chroma DB는 SQLite (메타데이터용)와 HNSW 인덱스 (벡터 검색용)가 결합된 형태

사용자가 데이터를 조회할 때, Chroma DB는 내부적으로 다음과 같이 동작:

1. HNSW 인덱스를 통해 질문과 가장 유사한 벡터의 ID를 찾음.
2. 해당 ID를 이용해 SQLite에서 실제 텍스트 내용과 메타데이터를 가져와 사용자에게 보여줌.

디스크 저장 구조 (폴더 보기)

data/chroma_db/

chroma.sqlite3

메타데이터/텍스트 DB

index/

id_to_uuid.pkl

HNSW 벡터 인덱스

index.bin

HNSW 벡터 인덱스

header.bin

HNSW 벡터 인덱스

chroma.lock

잠금 파일

컬렉션 개념

Chroma DB (데이터베이스)

Collection: "research_memory" (우리가 사용)

Document 1: {id, text, embedding, metadata}

Document 2: {id, text, embedding, metadata}

Document 3: {id, text, embedding, metadata}

...

Collection: "product_data"

...

</> 구현: 컬렉션 생성 및 설정

```
import chromadb
# PersistentClient: 디스크에 영구 저장
client = chromadb.PersistentClient(path="data/chroma_db")
# 컬렉션 생성 또는 로드
collection = client.get_or_create_collection(
    name="research_memory",
    metadata={
        "hnsw:space": "cosine",          # 코사인 유사도 사용
        "hnsw:construction_ef": 200,    # 구축 품질
        "hnsw:M": 16                   # 그래프 연결 수
    }
)
```

HNSW 알고리즘의 이해

Hierarchical Navigable Small World

핵심 개념: Small World

"6단계만 건너면 전 세계 누구든 안다"

무식하게 하나씩 대조하는 것이 아니라, 서로 가까운 데이터끼리 촘촘한 연결망(Graph)을 만들어 두고 그 연결망을 타고 이동하며 검색하는 방식입니다.

구조의 핵심: 계층(Hierarchy)

고속도로와 국도처럼 층이 나뉘어 있어 빠른 이동이 가능합니다.

최상위 층 (고속도로)

데이터가 드문드문 있음. 아주 먼 거리를 한 번에 점프.

중간 층 (간선도로)

목적지 근처 도시로 이동. 데이터가 조금 더 많아짐.

최하위 층 (국도/골목길)

모든 데이터가 연결됨. 가장 가까운 정확한 위치 탐색.

작동 원리 (4단계)

1. 상위층 시작 → 2. 근접 이동 → 3. 하위층 하강 → 4. 최종 탐색

계층적 탐색 시각화

Layer 2

고속도로



Layer 1

간선도로



Layer 0

골목길 (Base)



왜 HNSW를 쓰나요?

특징

설명

압도적 속도

전수 조사가 아닌 '건너뛰기' 방식으로 수백만 개 데이터도 0.001초 단위 검색

정확도

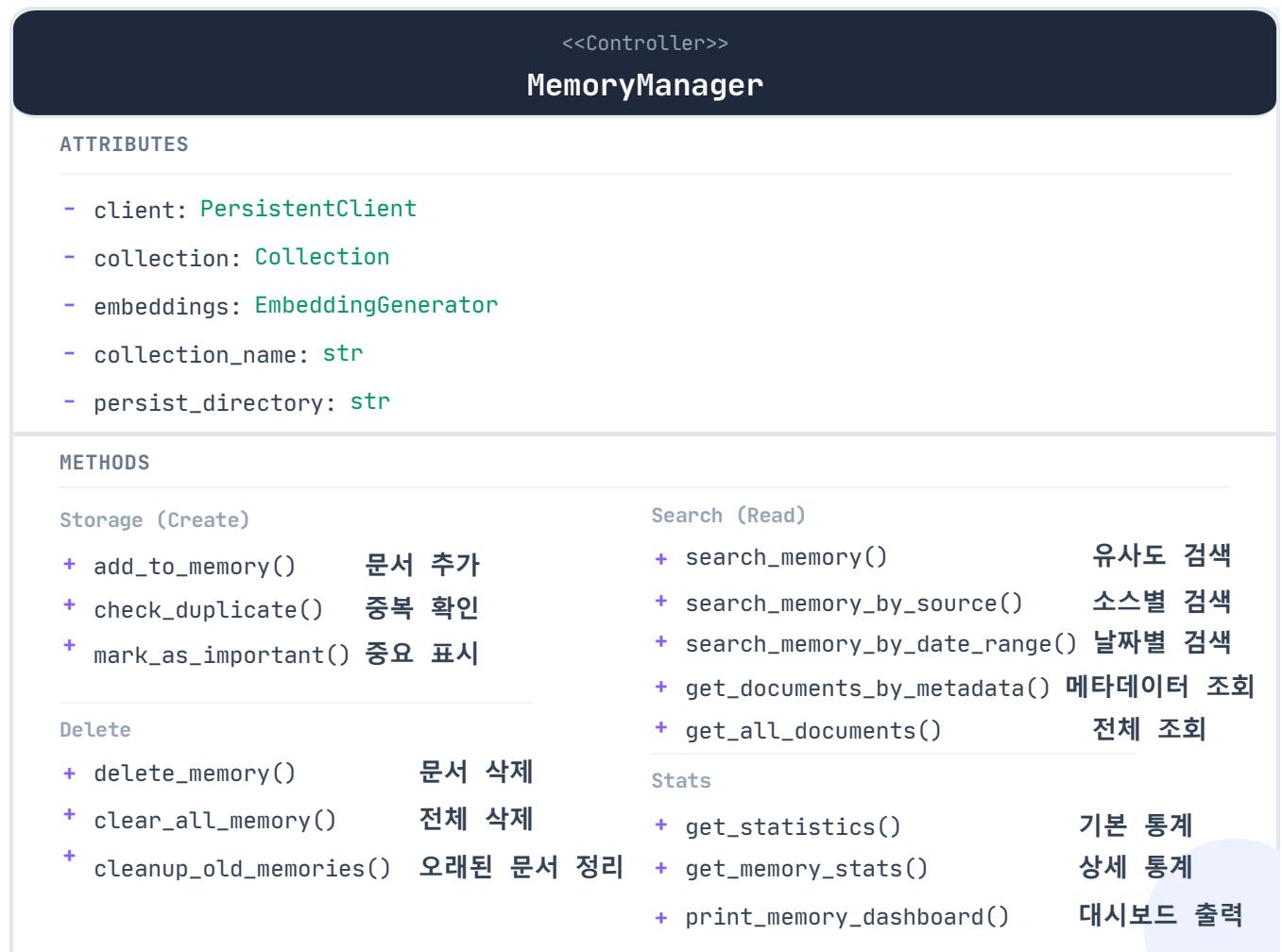
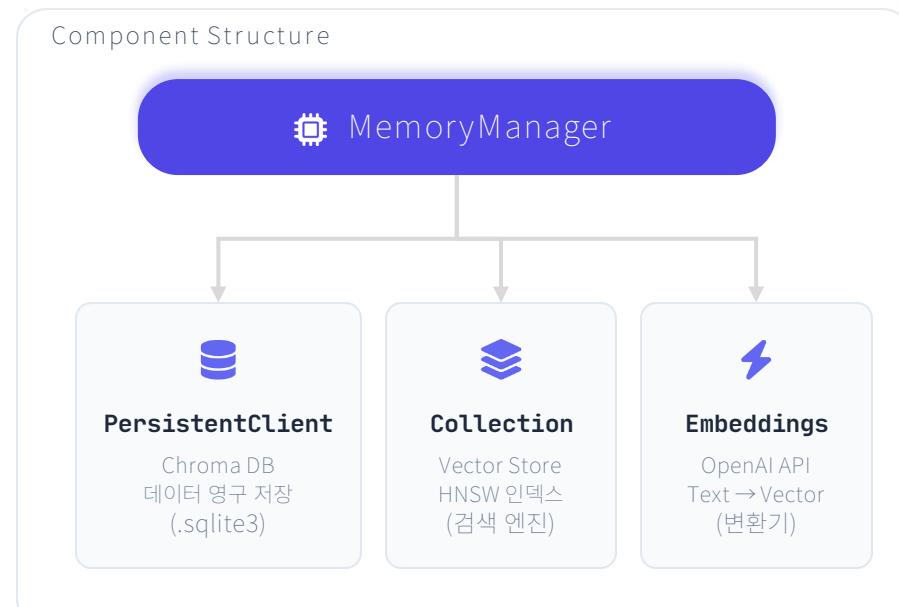
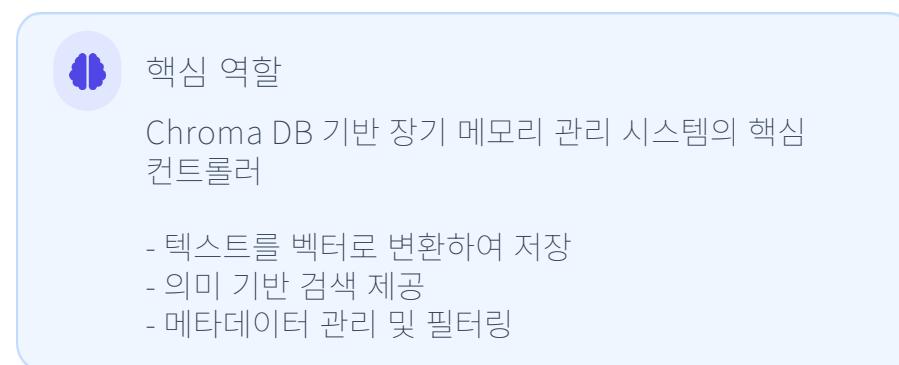
근사 검색(ANN)이지만, 실용적으로는 거의 완벽에 가까운 결과 제공

효율성

새로운 데이터 추가 시 인덱스를 새로 만들지 않고 유연하게 연결 가능

✓ HNSW는 "멀리서부터 크게 크게 좁혀 들어오며 가장 비슷한 것을 찾는 효율적인 길찾기 시스템"

MemoryManager 클래스 구조



MemoryManager 주요 메서드 상세 동작

add_to_memory() - 문서 저장 프로세스

</> Add to memory

```

src/memory_manager.py

def add_to_memory(
    self,
    text: str,
    metadata: Optional[Dict] = None,
    document_id: Optional[str] = None,
    check_duplicate: bool = True
) -> str:
    """
    문서를 메모리에 저장합니다.

    Args:
        text: 저장할 문서 내용
        metadata: 문서 관련 메타데이터
        document_id: 직접 지정할 ID (옵션)
        check_duplicate: 중복 검사 여부

    Returns:
        str: 저장된 문서의 ID
    """

```



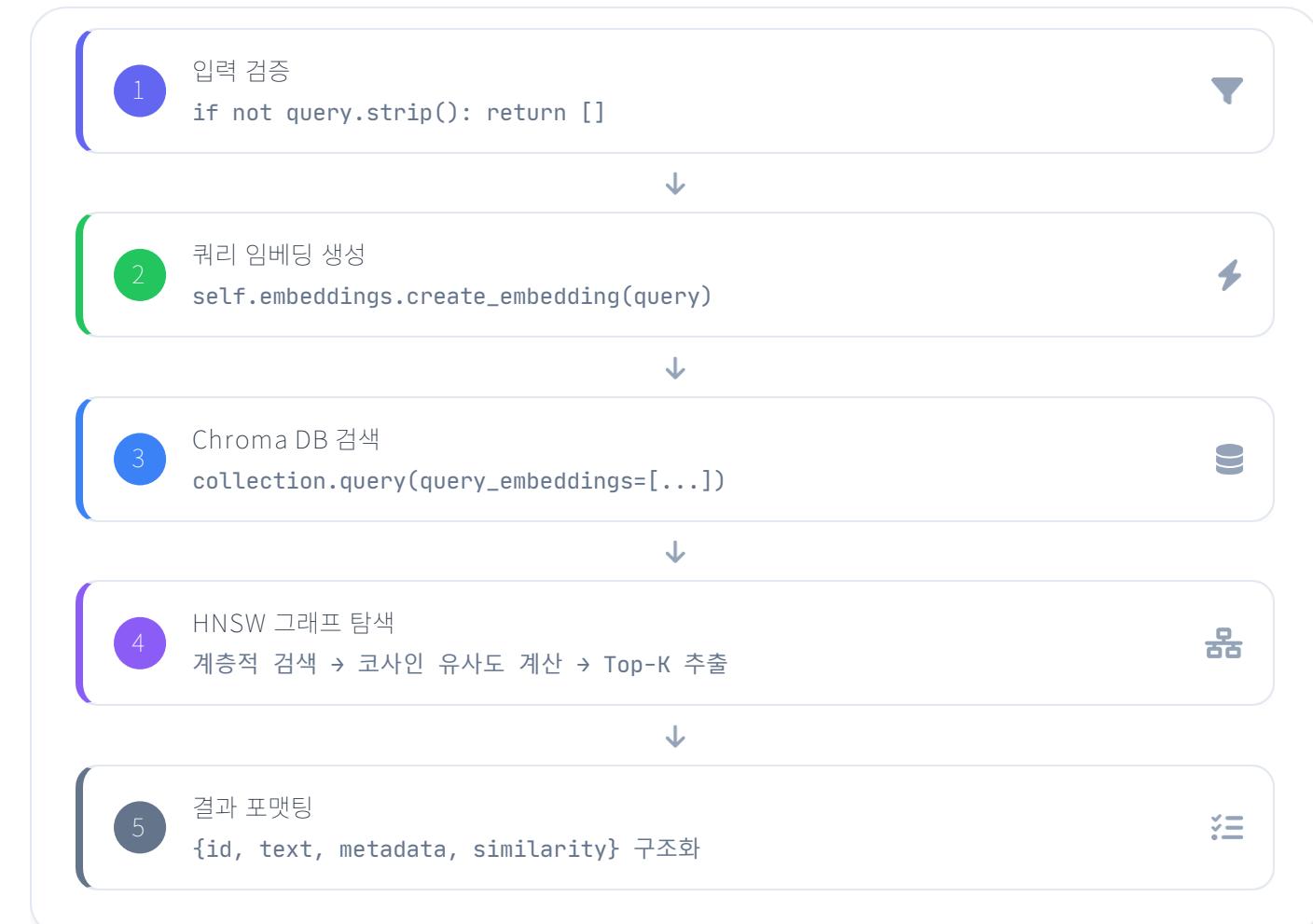
MemoryManager 주요 메서드 상세 동작

search_memory() - 유사도 검색 프로세스

</> Search Memory

src/memory_manager.py

```
def search_memory(
    self,
    query: str,
    top_k: int = 5,
    filter_dict: Optional[Dict] = None
) -> List[Dict[str, Any]]:
    """ 쿼리와 유사한 문서를 검색합니다. """
    # Args:
    query: 검색할 질문 텍스트
    top_k: 반환할 결과 개수 (기본 5)
    filter_dict: 메타데이터 필터 조건
    # Returns:
    유사도 점수가 포함된 문서 리스트
```



MemoryManager 주요 메서드 상세 동작

check_duplicate() - 중복 확인 프로세스

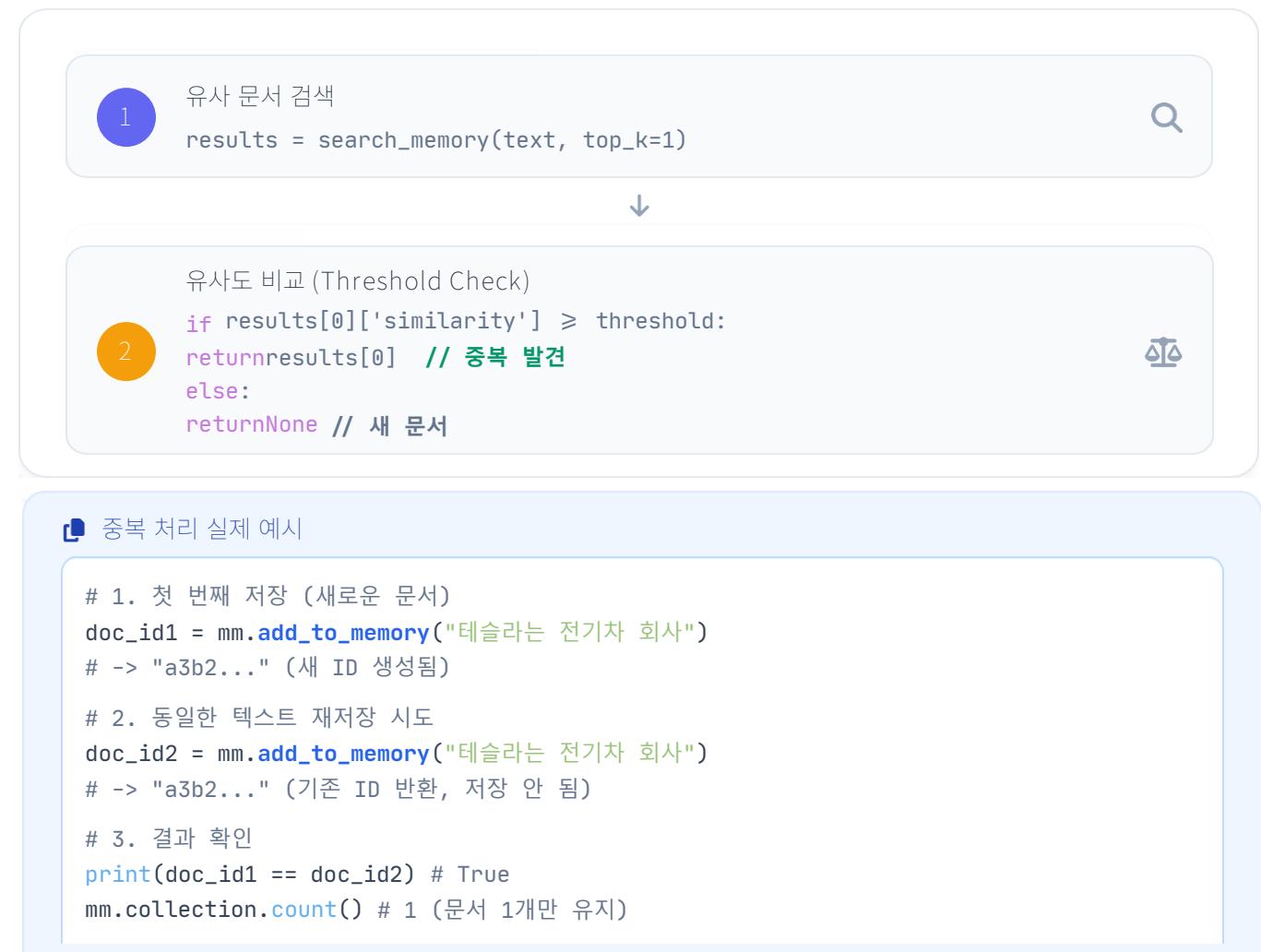
</> Check duplicate

```

src/memory_manager.py

def check_duplicate (
    self,
    text: str,
    threshold: float = 0.95
) -> Optional[Dict[str, Any]]:
    """
    메모리에 중복된 문서가 있는지 확인합니다.
    지정된 임계값(threshold) 이상의 유사도를 가진
    문서가 이미 존재하는지 검사합니다.
    """
    # Args:
    text: 중복 검사할 텍스트
    threshold: 중복 판정 유사도 기준 (기본 0.95)
    # Returns:
    중복 문서 정보 (ID 포함) 또는 None

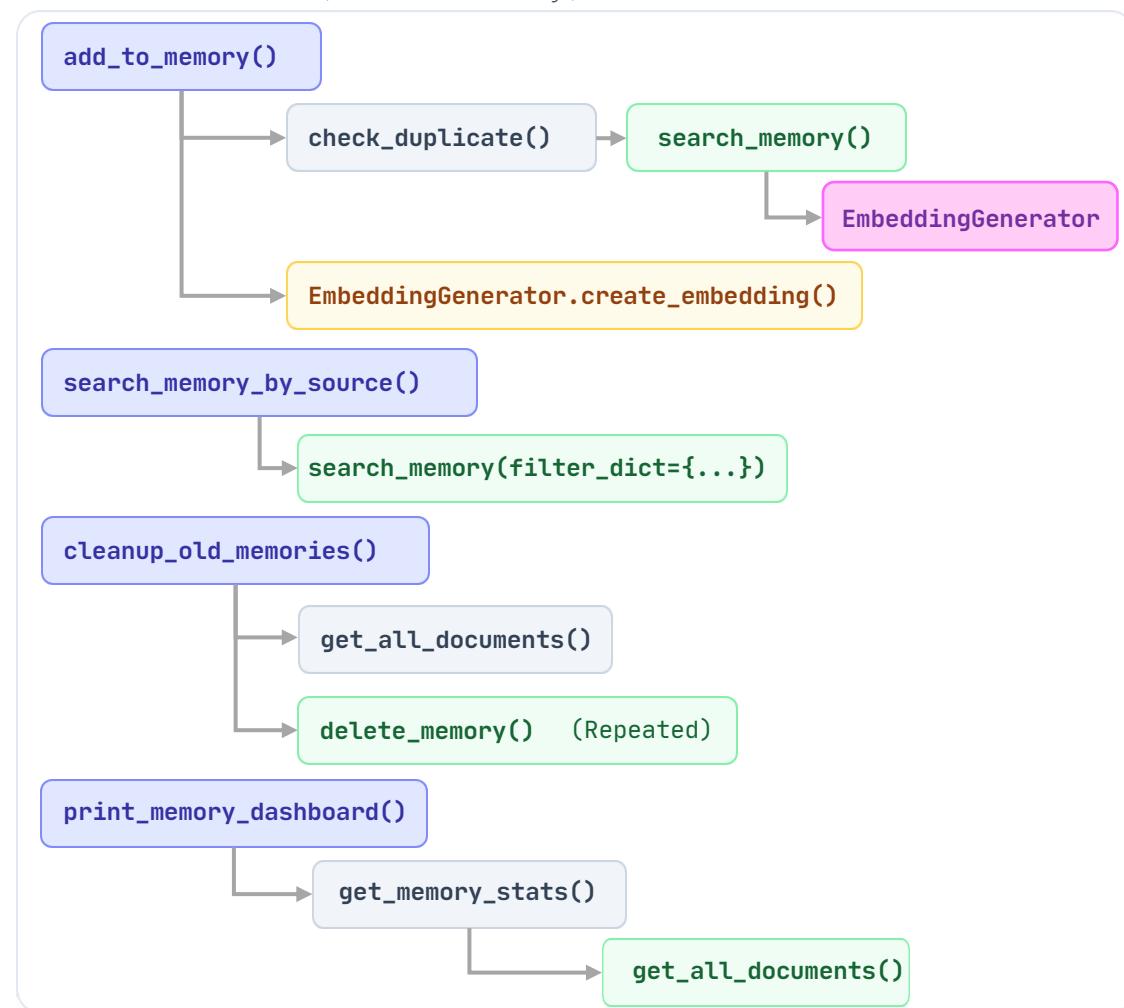
```



MemoryManager 메서드 간 상호작용

메서드 호출 관계 및 데이터 저장 구조

메서드 호출 관계 (Call Hierarchy)



MemoryManager가 관리하는 데이터 구조

```

# MemoryManager가 관리하는 데이터 구조
{
  "collection_name": "research_memory",
  "documents": [
    {
      "id": "doc_1",
      "text": "테슬라는 전기차 회사입니다",
      "embedding": [0.023, -0.056, ..., 0.045], # 1536차원
      "metadata": {
        "source": "web_search",
        "timestamp": "2024-01-26T15:30:45",
        "text_length": 15
      }
    },
    {
      "id": "doc_2",
      "text": "애플은 아이폰을 만듭니다",
      "embedding": [0.034, -0.045, ..., 0.067],
      "metadata": {
        "source": "user_input",
        "timestamp": "2024-01-26T15:31:12",
        "text_length": 14
      }
    }
  ]
}
  
```

This section shows the data structure managed by the MemoryManager, specifically the "research_memory" collection. It contains two documents, each with an ID, text content, a 1536-dimensional embedding vector, and associated metadata including source, timestamp, and text length.

실습 Part1 구현 프롬프트 - 1

1. data 밑에 아래의 폴더를 생성해주세요.

- chroma_db
- conversations

2. src 밑에 아래의 폴더를 생성해주세요.

- src\utils

3. src\utils 밑에 빈 __init__.py 파일을 만들어주세요.

[__init__.py 내용]

- 모듈 설명 docstring만 포함
- 실제 import/export는 나중에 추가할 예정이라는 TODO 주석

실습 Part1 구현 프롬프트 - 2

src/utils/embeddings.py 파일을 생성하고 다음 기능을 구현해주세요.

[클래스 구조]

```
class EmbeddingGenerator:
    def __init__(self):
        - OpenAI 클라이언트 초기화
        - 모델: "text-embedding-3-small"
        - 캐시 딕셔너리 초기화
        - 임베딩 생성 카운터 초기화

    def create_embedding(self, text: str) -> List[float]:
        - 단일 텍스트를 1536차원 벡터로 변환
        - 캐시 확인 (있으면 캐시 반환)
        - OpenAI API 호출
        - 결과 캐싱 및 반환

    def create_embeddings(self, texts: List[str]) -> List[List[float]]:
        - 여러 텍스트를 배치로 처리
        - 효율적인 API 호출

    def get_dimension(self) -> int:
        - 임베딩 차원 수 반환 (1536)

    def get_cache_info(self) -> dict:
        - 캐시 통계 반환
```

[요구사항]

- 에러 처리: try-except로 API 오류 처리
- 재시도 로직: 최대 3회 재시도 (지수 백오프)
- 로깅: INFO 레벨로 주요 작업 기록
- 타입 힌팅: 모든 메서드에 타입 명시
- Docstring: 구글 스타일 문서화

[Import]

```
from openai import OpenAI
from dotenv import load_dotenv
import os
from typing import List, Dict, Optional
import logging
import time
```

실습 Part1 구현 프롬프트 - 3

src/memory_manager.py 파일을 생성하고
MemoryManager 클래스의 기본 구조를 만들어주세요.

[클래스 구조]

```
class MemoryManager:
    """Chroma DB 기반 장기 메모리 관리 시스템"""

    def __init__(
        self,
        collection_name: str = "research_memory",
        persist_directory: str = "data/chroma_db"
    ):
        """
        Args:
            collection_name: Chroma 컬렉션 이름
            persist_directory: 벡터 DB 저장 경로
        """

        # 여기서 구현 (다음 프롬프트에서)
        pass

    def add_to_memory(
        self,
        text: str,
        metadata: Optional[Dict[str, Any]] = None,
        document_id: Optional[str] = None
    ) -> str:
        """문서를 벡터 DB에 저장"""
        pass
```

```
def search_memory(
    self,
    query: str,
    top_k: int = 5,
    filter_dict: Optional[Dict] = None
) -> List[Dict[str, Any]]:
    """유사도 기반 검색"""
    pass

def get_all_documents(self) -> List[Dict[str, Any]]:
    """모든 문서 조회"""
    pass

def delete_memory(self, document_id: str) -> bool:
    """문서 삭제"""
    pass

def clear_all_memory(self) -> int:
    """모든 문서 삭제"""
    pass

def get_statistics(self) -> Dict[str, Any]:
    """메모리 통계"""
    pass
```

[Import]

```
import chromadb
from chromadb.config import Settings
from src.utils.embeddings import EmbeddingGenerator
from typing import List, Dict, Optional, Any
from datetime import datetime
import logging
import uuid
import os
```

[주의사항]

- 각 메서드는 pass로 두고, 다음 프롬프트에서 하나씩 구현합니다
- Docstring에 메서드의 역할을 명확히 작성
- 타입 힌팅 필수

실습 Part1 구현 프롬프트 - 4

MemoryManager 클래스의 `_init_` 메서드를 완성해주세요.

[구현 로직]

1. 로거 초기화

```
logger = logging.getLogger(__name__)
```

2. persist_directory 폴더 생성

```
if not os.path.exists(persist_directory):
    os.makedirs(persist_directory, exist_ok=True)
logger.info(f"디렉토리 생성: {persist_directory}")
```

3. Chroma DB 클라이언트 생성

```
self.client = chromadb.PersistentClient(path=persist_directory)
```

4. 컬렉션 생성 또는 로드

```
self.collection = self.client.get_or_create_collection(
    name=collection_name,
    metadata={"hnsw:space": "cosine"} # 코사인 유사도
)
```

5. EmbeddingGenerator 초기화

```
self.embeddings = EmbeddingGenerator()
```

6. 속성 저장

```
self.collection_name = collection_name
self.persist_directory = persist_directory
```

7. 초기화 로그

```
logger.info(f"MemoryManager 초기화 완료")
logger.info(f"컬렉션: {collection_name}")
logger.info(f"저장 경로: {persist_directory}")
logger.info(f"기존 문서 수: {self.collection.count()}")
```

[에러 처리]

try-except로 전체를 감싸고:

- PermissionError: 폴더 생성 권한 오류
- Exception: 기타 모든 오류

[테스트 코드 (주석으로 포함)]

```
# 테스트
# from src.memory_manager import MemoryManager
# mm = MemoryManager("test_memory",
# "data/chroma_db")
# print(f"✓ 초기화 완료: {mm.collection_name}")
# print(f"✓ 문서 수: {mm.collection.count()}")
```

실습 Part1 구현 프롬프트 - 5

MemoryManager의 add_to_memory 메서드를 구현해주세요.

[메서드 시그니처]

```
def add_to_memory(
    self,
    text: str,
    metadata: Optional[Dict[str, Any]] = None,
    document_id: Optional[str] = None
) -> str:
    """
    텍스트를 벡터 DB에 저장합니다.
    
```

Args:

- text: 저장할 텍스트
- metadata: 추가 메타데이터 (선택)
- document_id: 문서 ID (None이면 UUID 생성)

Returns:

- str: 저장된 문서 ID

Raises:

- ValueError: text가 비어있는 경우

[구현 로직]

1. 입력 검증

```
if not text or not text.strip():
    raise ValueError("텍스트가 비어있습니다")
text = text.strip()
```

2. document_id 생성 (없는 경우)

```
if document_id is None:
    document_id = str(uuid.uuid4())
"""

```

3. 임베딩 생성

```
embedding = self.embeddings.create_embedding(text)
```

4. 메타데이터 자동 추가

```
if metadata is None:
    metadata = {}
metadata.update({
    "timestamp": datetime.now().isoformat(),
    "text_length": len(text),
    "source": metadata.get("source", "user_input")
})
```

5. Chroma DB에 저장

```
self.collection.add(
    ids=[document_id],
    documents=[text],
    embeddings=[embedding],
    metadatas=[metadata]
)
```

6. 로깅 및 반환

```
logger.info(f"문서 저장 완료: {document_id}")
return document_id
```

[에러 처리]

try-except로 전체를 감싸고:

- ValueError: 입력 오류
- Exception: API 오류 등

[테스트 코드 (주석)]

```
# doc_id = mm.add_to_memory(
#     text="테슬라는 전기차 회사입니다",
#     metadata={"source": "test", "category": "company"}
# )
# print(f"✓ 문서 저장: {doc_id}")
```

실습 Part1 구현 프롬프트 - 6

MemoryManager의 search_memory 메서드를 구현해주세요.

[메서드 시그니처]

```
def search_memory(
    self,
    query: str,
    top_k: int = 5,
    filter_dict: Optional[Dict] = None
) -> List[Dict[str, Any]]:
    """
```

쿼리와 유사한 문서를 검색합니다.

Args:

- query: 검색 쿼리
- top_k: 반환할 최대 문서 수
- filter_dict: 메타데이터 필터 (선택)

Returns:

- List[Dict]: 검색 결과 리스트
 - text: 문서 텍스트
 - metadata: 메타데이터
 - similarity: 유사도 점수 (0-1)
 - id: 문서 ID

....

[구현 로직]

1. 입력 검증

```
if not query or not query.strip():
    raise ValueError("검색 쿼리가 비어있습니다")
```

2. 쿼리 임베딩 생성

```
query_embedding =
self.embeddings.create_embedding(query)
"""
```

3. Chroma DB 검색

```
results = self.collection.query(
    query_embeddings=[query_embedding],
    n_results=top_k,
    where=filter_dict # 메타데이터 필터
)
```

4. 결과 포맷팅

```
formatted_results = []
for i in range(len(results['ids'][0])):
    formatted_results.append({
        'id': results['ids'][0][i],
        'text': results['documents'][0][i],
        'metadata': results['metadatas'][0][i],
        'similarity': 1 - results['distances'][0][i] # 거리 →
    })
    유사도
    })
```

5. 로깅 및 반환

```
logger.info(f"검색 완료: {len(formatted_results)}개 결과")
return formatted_results
```

[테스트 코드 (주석)]

```
# results = mm.search_memory("전기차", top_k=3)
# for r in results:
#     print(f"유사도: {r['similarity']:.2f} | {r['text'][:50]}...")
```

실습 Part1 구현 프롬프트 - 7

MemoryManager의 나머지 기본 메서드들을 구현해주세요.

[1. get_all_documents 메서드]

```
def get_all_documents(self) -> List[Dict[str, Any]]:
    """모든 문서를 조회합니다."""

    results = self.collection.get()

    documents = []
    for i in range(len(results['ids'])):
        documents.append({
            'id': results['ids'][i],
            'text': results['documents'][i],
            'metadata': results['metadatas'][i]
        })

    return documents
```

[2. delete_memory 메서드]

```
def delete_memory(self, document_id: str) -> bool:
    """특정 문서를 삭제합니다."""

    try:
        self.collection.delete(ids=[document_id])
        logger.info(f"문서 삭제 완료: {document_id}")
        return True
    except Exception as e:
        logger.error(f"문서 삭제 실패: {e}")
        return False
```

[3. clear_all_memory 메서드]

```
def clear_all_memory(self) -> int:
    """모든 문서를 삭제합니다."""

    count = self.collection.count()

    # 컬렉션 삭제 후 재생성
    self.client.delete_collection(name=self.collection_name)
    self.collection = self.client.get_or_create_collection(
        name=self.collection_name,
        metadata={"hnsw:space": "cosine"}
    )

    logger.info(f"전체 메모리 삭제: {count}개 문서")
    return count
```

[4. get_statistics 메서드]

```
def get_statistics(self) -> Dict[str, Any]:
    """메모리 통계를 반환합니다."""

    all_docs = self.get_all_documents()

    # 소스별 카운트
    sources = {}
    for doc in all_docs:
        source = doc['metadata'].get('source', 'unknown')
        sources[source] = sources.get(source, 0) + 1

    return {
        'total_documents': len(all_docs),
        'collection_name': self.collection_name,
        'by_source': sources,
        'persist_directory': self.persist_directory
    }
```

[테스트 코드 (주석)]

```
# print(f"✓ 전체 문서: {len(mm.get_all_documents())}")
# print(f"✓ 통계: {mm.get_statistics()}")
# mm.delete_memory(doc_id)
# print(f"✓ 삭제 후: {mm.collection.count()}개")
```

실습 Part1 구현 프롬프트 - 8

MemoryManager에 중복 문서 확인 및 제거 기능을 추가해주세요.

[새 메서드: check_duplicate]

```
def check_duplicate(
    self,
    text: str,
    threshold: float = 0.95
) -> Optional[Dict[str, Any]]:
    """
    중복 문서를 확인합니다.

```

Args:

```
    text: 확인할 텍스트
    threshold: 중복 판단 임계값 (0-1)
```

Returns:

중복 문서 정보 또는 None

유사 문서 검색

```
results = self.search_memory(text, top_k=1)
```

if not results:

```
    return None
```

가장 유사한 문서 확인

```
most_similar = results[0]
```

```
if most_similar['similarity'] >= threshold:
    logger.info(f"중복 문서 발견: {most_similar['id']}")
    # 유사도: {most_similar['similarity']:.3f}")
    return most_similar

return None
```

[add_to_memory 메서드 수정]

기존 add_to_memory 메서드에 중복 체크 추가:

```
def add_to_memory(
    self,
    text: str,
    metadata: Optional[Dict[str, Any]] = None,
    document_id: Optional[str] = None,
    check_duplicate: bool = True # NEW 파라미터
) -> str:
```

입력 검증

```
if not text or not text.strip():
    raise ValueError("텍스트가 비어있습니다")
```

중복 체크 (check_duplicate=True인 경우)

```
if check_duplicate:
    duplicate = self.check_duplicate(text)
    if duplicate:
        logger.warning(f"중복 문서로 저장 스킵: {duplicate['id']}")
        return duplicate['id'] # 기존 문서 ID 반환
```

기존 로직 계속...

[테스트 코드 (주석)]

```
# # 첫 번째 저장
# id1 = mm.add_to_memory("테슬라는 전기차 회사입니다")
# print(f"✓ 문서 저장: {id1}")
#
# # 중복 저장 시도
# id2 = mm.add_to_memory("테슬라는 전기차 회사입니다")
# print(f"✓ 중복 체크: {id1 == id2}") # True여야 함
```

실습 Part1 구현 프롬프트 - 9

메타데이터 기반 필터링 기능을 강화해주세요.

[search_memory_by_source 메서드 추가]

```
def search_memory_by_source(
    self,
    query: str,
    source: str,
    top_k: int = 5
) -> List[Dict[str, Any]]:
    """특정 소스에서만 검색합니다."""

    filter_dict = {"source": source}
    return self.search_memory(query, top_k, filter_dict)
```

[search_memory_by_date_range 메서드 추가]

```
def search_memory_by_date_range(
    self,
    query: str,
    start_date: str, # ISO format
    end_date: str, # ISO format
    top_k: int = 5
) -> List[Dict[str, Any]]:
    """날짜 범위 내에서 검색합니다."""

    # Chroma의 where 필터 사용
    filter_dict = {
        "$and": [
            {"timestamp": {"$gte": start_date}},
            {"timestamp": {"$lte": end_date}}
        ]
    }
```

return self.search_memory(query, top_k, filter_dict)

[get_documents_by_metadata 메서드 추가]

```
def get_documents_by_metadata(
    self,
    key: str,
    value: Any
) -> List[Dict[str, Any]]:
    """특정 메타데이터 값으로 문서를 조회합니다."""

    results = self.collection.get(
        where={key: value}
    )

    documents = []
    for i in range(len(results['ids'])):
        documents.append({
            'id': results['ids'][i],
            'text': results['documents'][i],
            'metadata': results['metadatas'][i]
        })

    return documents
```

[테스트 코드 (주석)]

```
# # 소스별 검색
# web_results = mm.search_memory_by_source("테슬라",
# "web_search")
# print(f"✓ 웹 검색 결과: {len(web_results)}개"
# #
# # 메타데이터 조회
# docs = mm.get_documents_by_metadata("category",
# "company")
# print(f"✓ 회사 카테고리: {len(docs)}개")
```

실습 Part1 구현 프롬프트 - 10

오래된 메모리 정리 기능과 대시보드 출력 기능을 추가해주세요.

```
[1. cleanup_old_memories 메서드]
def cleanup_old_memories(
    self,
    days_old: int = 30,
    keep_important: bool = True
) -> Dict[str, int]:
    """
    오래된 메모리를 정리합니다.

    Args:
        days_old: 이 일수보다 오래된 문서 삭제
        keep_important: 중요 문서 보존 여부

    Returns:
        {"deleted": int, "kept": int}
    """

```

```
from datetime import datetime, timedelta

# 기준 날짜 계산
cutoff_date = (datetime.now() - timedelta(days=days_old)).isoformat()

# 전체 문서 조회
all_docs = self.get_all_documents()

deleted = 0
kept = 0

for doc in all_docs:
    timestamp = doc['metadata'].get('timestamp', '')

    # 오래된 문서인지 확인
    if timestamp < cutoff_date:
        # 중요 문서 체크
        if keep_important and doc['metadata'].get('important', False):
            kept += 1
            continue

        # 삭제
        deleted += 1
```

```
# 삭제
if self.delete_memory(doc['id']):
    deleted += 1
else:
    kept += 1

logger.info(f"메모리 정리 완료: {deleted}개 삭제, {kept}개 보존")

return {
    "deleted": deleted,
    "kept": kept,
    "cutoff_date": cutoff_date
}
```

[2. mark_as_important 메서드]

```
def mark_as_important(self, document_id: str) -> bool:
    """
    문서를 중요로 표시합니다.
    """

try:
```

```
# 문서 조회
doc = self.collection.get(ids=[document_id])

if not doc['ids']:
    return False

# 메타데이터 업데이트
metadata = doc['metadatas'][0]
metadata['important'] = True
```

```
# 문서 업데이트 (삭제 후 재생성)
self.collection.delete(ids=[document_id])
self.collection.add(
    ids=[document_id],
    documents=[doc['documents'][0]],
    metadata=[metadata]
)

logger.info(f"중요 문서 표시: {document_id}")
return True

except Exception as e:
    logger.error(f"중요 표시 실패: {e}")
    return False
```

[3. get_memory_stats 메서드]

```
def get_memory_stats(self) -> Dict[str, Any]:
    """
    상세한 메모리 통계를 반환합니다.
    """

all_docs = self.get_all_documents()
```

```
# 소스별 카운트
sources = {}
for doc in all_docs:
    source = doc['metadata'].get('source', 'unknown')
    sources[source] = sources.get(source, 0) + 1

..... (전체 내용은 별첨 참조)
```

실습 Part1 구현 프롬프트 - 11

tests/test_memory_manager.py 파일을 생성하고
MemoryManager의 주요 기능에 대한 단위 테스트를 작성해주세요.

[테스트 파일 구조]

```
import unittest
import os
import shutil
from src.memory_manager import MemoryManager
from src.utils.embeddings import EmbeddingGenerator
```

```
class TestMemoryManager(unittest.TestCase):
    """MemoryManager 단위 테스트"""

    @classmethod
    def setUpClass(cls):
        """테스트 시작 전 한 번 실행"""
        cls.test_dir = "data/test_chroma_db"
        cls.mm = MemoryManager("test_collection", cls.test_dir)

    @classmethod
    def tearDownClass(cls):
        """테스트 종료 후 정리"""
        if os.path.exists(cls.test_dir):
            shutil.rmtree(cls.test_dir)

    def setUp(self):
        """각 테스트 전 실행"""
        # 기존 데이터 정리
        self.mm.clear_all_memory()

    # 테스트 메서드들...
```

[1. 초기화 테스트]

```
def test_initialization(self):
    """MemoryManager 초기화 테스트"""
    self.assertIsNotNone(self.mm.client)
    self.assertIsNotNone(self.mm.collection)
    self.assertIsNotNone(self.mm.embeddings)
    self.assertEqual(self.mm.collection_name,
                    "test_collection")
```

[2. 문서 저장 테스트]

```
def test_add_to_memory(self):
    """문서 저장 기능 테스트"""
    text = "테슬라는 전기차 회사입니다"
    doc_id = self.mm.add_to_memory(text)
```

```
self.assertIsNotNone(doc_id)
self.assertEqual(self.mm.collection.count(), 1)

# 빈 텍스트는 예외 발생
with self.assertRaises(ValueError):
    self.mm.add_to_memory("")
```

[3. 검색 테스트]

```
def test_search_memory(self):
    """검색 기능 테스트"""
    # 문서 저장
    self.mm.add_to_memory("테슬라는 전기차 회사입니다")
    self.mm.add_to_memory("애플은 아이폰을 만듭니다")
```

검색

```
results = self.mm.search_memory("전기차", top_k=2)
```

```
self.assertGreater(len(results), 0)
self.assertIn('text', results[0])
self.assertIn('similarity', results[0])
self.assertGreater(results[0]['similarity'], 0.5)
```

[4. 중복 체크 테스트]

```
def test_duplicate_check(self):
    """중복 문서 체크 테스트"""
    text = "테슬라는 전기차 회사입니다"
```

```
doc_id1 = self.mm.add_to_memory(text)
doc_id2 = self.mm.add_to_memory(text) # 중복

# 동일한 ID 반환되어야 함
self.assertEqual(doc_id1, doc_id2)
self.assertEqual(self.mm.collection.count(), 1)
```

..... (전체 내용은 별첨 참조)

Part 1 단위 테스트

💡 가상환경 활성화 후, 테스트 스크립트 실행

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

> python tests/test_memory_manager.py

⚙️ 단위 테스트 항목

1. 초기화 테스트
2. 문서 저장 테스트
3. 검색 테스트
4. 중복 체크 테스트
5. 삭제 테스트
6. 메타데이터 필터링 테스트
7. 통계 테스트

```
● ● ●  
test_add_to_memory (__main__.TestMemoryManager.test_add_to_memory)  
문서 저장 기능 테스트 ... 2026-01-27 05:09:12,297 - INFO - 전체 메모리 삭제: 0개 문서  
2026-01-27 05:09:15,450 - INFO - HTTP Request: POST https://api.openai.com/v1/embeddings  
"HTTP/1.1 200 OK"  
2026-01-27 05:09:15,455 - INFO - 임베딩 생성 완료: 테슬라는 전기차 회사입니다...  
2026-01-27 05:09:17,129 - INFO - 문서 저장 완료: a8735874-0124-4902-95f5-cf9939d0dc94  
ok
```

Part 1 종합 테스트

⚡ 가상환경 활성화 후, 테스트 스크립트 실행

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

※ 파일 별첨

> python tests/test_part1.py

```
=====
Part 1 종합 테스트 시작
=====
(중략)
=====
Part 1 종합 테스트 완료!
```

🏁 Part 1 완료 기준

다음 조건을 모두 만족하면 성공입니다:

test_part1.py 실행 시 "🎉 모든 테스트를 통과했습니다!" 메시지 출력
단위 테스트(test_memory_manager.py) 통과
에러 메시지 없이 대시보드가 정상 출력됨



05

SECTION

HANDS-ON

IMPLEMENTATION

HANDS-ON PART 2

SearchAgent 메모리 통합 (50분): 기존 검색 에이전트에 장기 기억 시스템을 연결하여 더 똑똑하고 효율적인 검색 로직을 구현합니다.

KEY TASKS

🔌 SearchAgent에 MemoryManager 통합

💾 웹 검색 결과 자동 저장

⚡ 메모리 우선 검색 로직 구현

🖨️ 메모리 + 웹 결과 병합 및 출력 추적

Part 2 작업 개요

Search Agent 메모리 통합

① 학습 목표

- ✓ SearchAgent에 MemoryManager 통합
- ✓ 메모리 우선 검색(Memory-First) 로직 구현
- ✓ 웹 검색 결과 자동 저장 (DB 영구 보관)
- ✓ 메모리 + 웹 결과 병합 및 정보 출처 추적

</> 수정할 주요 클래스

```
src/search_agent.py

class SearchAgent :
    """메모리 기능이 추가된 웹 검색 에이전트"""

    def __init__(self, memory_manager= None):
        self.client = OpenAI()
        self.memory_manager = memory_manager # NEW
        self.tavily_client = TavilyClient()

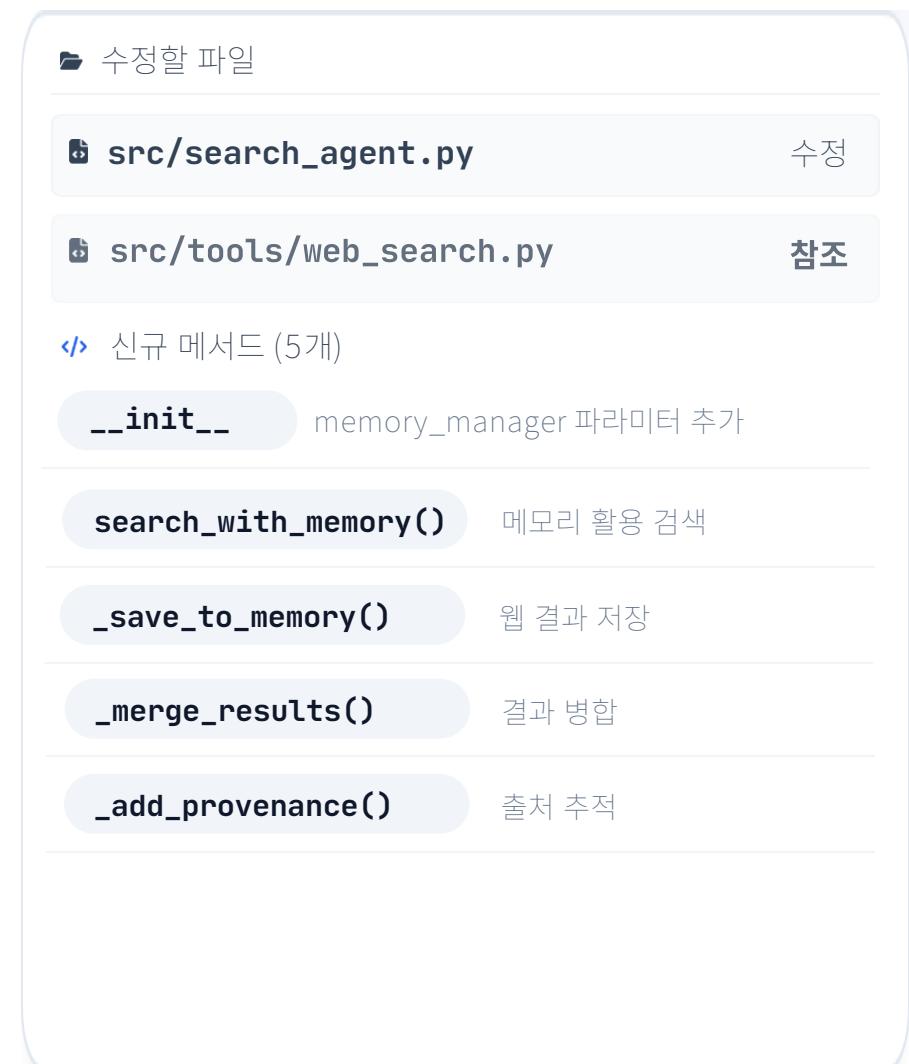
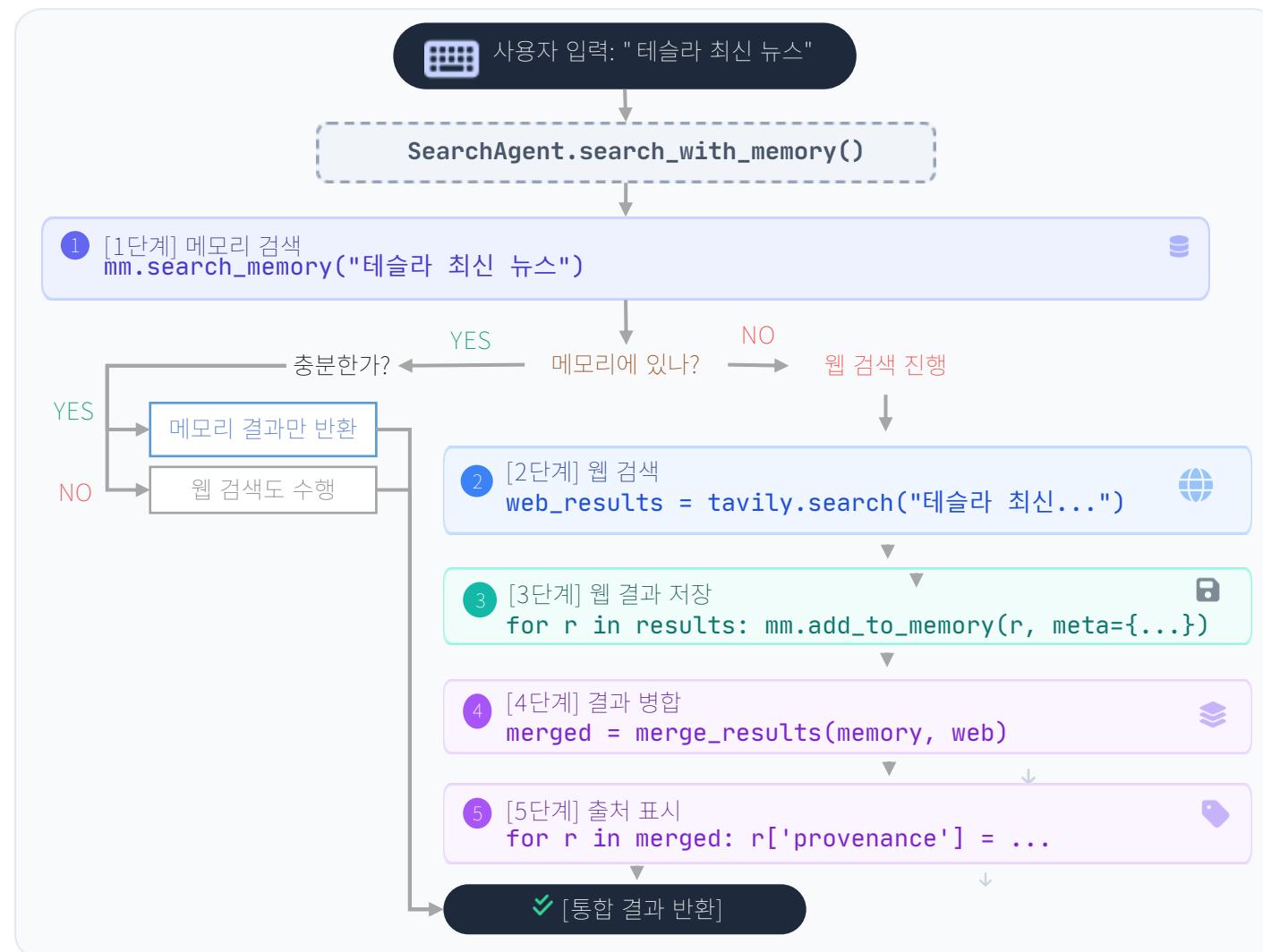
    def search_with_memory (self, query, use_memory= True):
        # 1. 메모리 먼저 검색
        # 2. 충분하지 않으면 웹 검색
        # 3. 웹 결과를 메모리에 저장
        # 4. 결과 병합하여 반환
```

| 메모리 통합 전후 데이터 흐름 비교



Part 2 프로그램 동작 흐름

Search Agent 메모리 통합 로직 상세



실습 Part2 구현 프롬프트 - 1

src/search_agent.py 파일을 열고
SearchAgent 클래스의 `_init_` 메서드를 수정해주세요.

[기존 코드 확인]
먼저 현재 SearchAgent의 구조를 확인하세요.

[수정 사항]

1. `_init_` 메서드에 `memory_manager` 파라미터 추가

```
def __init__(self, memory_manager: Optional[MemoryManager] = None):
```

Args:

`memory_manager`: 메모리 관리자 (선택적)

기존 코드 유지

`self.client = OpenAI()`

`load_dotenv()`

`tavily_api_key = os.getenv("TAVILY_API_KEY")`

`self.tavily_client = TavilyClient(api_key=tavily_api_key)`

메모리 관리자 추가 (NEW)

`self.memory_manager = memory_manager`

로깅

if `self.memory_manager`:

 logger.info("SearchAgent initialized with memory")

else:

 logger.info("SearchAgent initialized without memory")

2. Import 문 추가

파일 상단에 다음 추가:

```
from typing import Optional, List, Dict, Any
from src.memory_manager import MemoryManager
from datetime import datetime
```

[테스트 코드 (주석)]

```
# from src.search_agent import SearchAgent
# from src.memory_manager import MemoryManager
#
# # 메모리 없이 초기화
# agent1 = SearchAgent()
#
# # 메모리와 함께 초기화
# mm = MemoryManager("search_memory", "data/chroma_db")
# agent2 = SearchAgent(memory_manager=mm)
#
# print(f"Agent1 메모리: {agent1.memory_manager}") # None
# print(f"Agent2 메모리: {agent2.memory_manager}") # <MemoryManager object>
```

[주의사항]

- 기존 `search()` 메서드는 그대로 유지 (하위 호환성)

- `memory_manager`는 `Optional` (없어도 작동)

- 기존 코드를 최대한 건드리지 않기

실습 Part2 구현 프롬프트 - 2

SearchAgent에 search_with_memory() 메서드를 추가해주세요.

[메서드 시그니처]

```
def search_with_memory(
    self,
    query: str,
    use_memory: bool = True,
    save_to_memory: bool = True,
    memory_threshold: int = 3,
    max_results: int = 5
) -> Dict[str, Any]:
    """
    메모리를 활용한 지능형 검색
    
```

Args:

- query: 검색 쿼리
- use_memory: 메모리 검색 사용 여부
- save_to_memory: 웹 결과를 메모리에 저장할지 여부
- memory_threshold: 웹 검색을 수행할 최소 메모리 결과 수
- max_results: 최대 반환 결과 수

Returns:

```
{
    "query": str,
    "memory_results": List[dict],
    "web_results": List[dict],
    "merged_results": List[dict],
    "source_summary": dict
}
    
```

[구현 로직 - Part 1: 초기화 및 메모리 검색]

```
def search_with_memory(self, query: str, ...) -> Dict[str, Any]:
    logger.info(f"Search with memory: {query}")

    # 결과 저장 구조
    results = {
        "query": query,
        "memory_results": [],
        "web_results": [],
        "merged_results": [],
        "source_summary": {
            "from_memory": 0,
            "from_web": 0,
            "total": 0
        }
    }

    # [1단계] 메모리 검색
    if use_memory and self.memory_manager:
        try:
            memory_results = self.memory_manager.search_memory(
                query=query,
                top_k=max_results
            )
            results["memory_results"] = memory_results
            logger.info(f"Found {len(memory_results)} results in memory")
        except Exception as e:
            logger.error(f"Memory search failed: {e}")
            memory_results = []
    else:
        memory_results = []
        # 계속... (다음 프롬프트에서)
```

[테스트 코드 (주석)]

```
# agent = SearchAgent(memory_manager=mm)
#
# # 메모리만 검색
# result = agent.search_with_memory(
#     query="테슬라",
#     use_memory=True,
#     save_to_memory=False
# )
# print(f"메모리 결과: {len(result['memory_results'])}개")
```

실습 Part2 구현 프롬프트 - 3

search_with_memory() 메서드를 완성해주세요.

[구현 로직 - Part 2: 웹 검색 및 저장]

```
# ... (이전 코드 계속)

# [2단계] 웹 검색 필요성 판단
need_web_search = len(memory_results) < memory_threshold

if need_web_search:
    logger.info("Memory results insufficient, performing web search")

try:
    # 웹 검색 수행 (기존 tavity_search 함수 사용)
    from src.tools.web_search import tavity_search

    web_response = tavity_search(query, max_results=max_results)
    web_results = web_response.get("results", [])
    results["web_results"] = web_results

    logger.info(f"Found {len(web_results)} results from web")

    # [3단계] 웹 결과를 메모리에 저장
    if save_to_memory and self.memory_manager:
        self._save_to_memory(web_results, query)

except Exception as e:
    logger.error(f"Web search failed: {e}")
    web_results = []

else:
    logger.info("Sufficient results in memory, skipping web search")
    web_results = []
```

```
# [4단계] 결과 병합
results["merged_results"] = self._merge_results(
    memory_results,
    web_results
)

# [5단계] 통계 업데이트
results["source_summary"] = {
    "from_memory": len(memory_results),
    "from_web": len(web_results),
    "total": len(results["merged_results"])
}

logger.info(f"Search complete: {results['source_summary']}")

return results
```

[테스트 코드 (주석)]
완전한 검색 테스트
result = agent.search_with_memory(
query="테슬라 최신 뉴스",
use_memory=True,
save_to_memory=True,
memory_threshold=3
)
#

```
# print(f"쿼리: {result['query']}")
# print(f"메모리:
{result['source_summary']['from_memory']}개")
# print(f"웹: {result['source_summary']['from_web']}개")
# print(f"총: {result['source_summary']['total']}개")
#
# # 병합된 결과 확인
# for i, r in enumerate(result['merged_results'][3], 1):
#     print(f"{i}. {r['content'][:50]}... (출처: {r['source']})")
```

실습 Part2 구현 프롬프트 - 4

웹 검색 결과를 메모리에 저장하는 헬퍼 메서드를 구현해주세요.

[메서드 구현]

```
def _save_to_memory(
    self,
    web_results: List[dict],
    query: str
) -> int:
    """
    웹 검색 결과를 메모리에 저장
    """

    Args:
        web_results: Tavily 검색 결과 리스트
        query: 원본 검색 쿼리
    Returns:
        저장된 문서 수
    """

```

```
if not self.memory_manager:
    return 0
```

```
saved_count = 0
```

```
for result in web_results:
    try:
        # 텍스트 추출
        text = result.get("content", "")
        if not text:
            continue
```

```
# 메타데이터 구성
metadata = {
    "source": "web_search",
    "query": query,
    "url": result.get("url", ""),
    "title": result.get("title", ""),
    "score": result.get("score", 0.0),
    "saved_at": datetime.now().isoformat()
}

# 메모리에 저장 (중복 체크 활성화)
doc_id = self.memory_manager.add_to_memory(
    text=text,
    metadata=metadata,
    check_duplicate=True # 중복 저장 방지
)

saved_count += 1
logger.debug(f"Saved to memory: {doc_id}")

except Exception as e:
    logger.error(f"Failed to save result: {e}")
    continue

logger.info(f"Saved {saved_count}/{len(web_results)} results to
memory")
return saved_count
```

[테스트 코드 (주석)]

```
# # 웹 검색 후 저장 테스트
# from src.tools.web_search import tavily_search
#
# web_results = tavily_search("테슬라", max_results=5)
# saved =
agent._save_to_memory(web_results["results"], "테슬라")
#
# print(f"저장됨: {saved}개")
#
# # 메모리에서 다시 검색
# memory_results = mm.search_memory("테슬라",
top_k=5)
# print(f"메모리에서 검색: {len(memory_results)}개")
```

실습 Part2 구현 프롬프트 - 5

메모리와 웹 검색 결과를 병합하는 메서드를 구현해주세요.

[메서드 구현]

```
def _merge_results(
    self,
    memory_results: List[dict],
    web_results: List[dict]
) -> List[dict]:
    """
    메모리와 웹 결과를 하나로 병합
    """

    Args:
```

memory_results: 메모리 검색 결과
web_results: 웹 검색 결과

Returns:
 병합된 결과 리스트 (출처 정보 포함)

merged = []

```
# [1] 메모리 결과 추가
for r in memory_results:
    merged.append({
        "content": r["text"],
        "source": "memory",
        "similarity": r.get("similarity", 0.0),
        "metadata": r.get("metadata", {}),
        "provenance": {
            "retrieved_from": "memory",
            "original_source": r["metadata"].get("source", "unknown"),
            "original_query": r["metadata"].get("query", ""),
        }
    })
```

```
"stored_at": r["metadata"].get("timestamp", ""),
            "confidence": r.get("similarity", 0.0)
        })
    })

# [2] 웹 결과 추가
for r in web_results:
    merged.append({
        "content": r.get("content", ""),
        "source": "web",
        "url": r.get("url", ""),
        "title": r.get("title", ""),
        "score": r.get("score", 0.0),
        "provenance": {
            "retrieved_from": "web",
            "url": r.get("url", ""),
            "fetched_at": datetime.now().isoformat(),
            "tavily_score": r.get("score", 0.0)
        }
    })

# [3] 중복 제거 (옵션)
merged = self._remove_duplicates(merged)

# [4] 정렬 (유사도/점수 기준)
def sort_key(item):
    if item["source"] == "memory":
        return item.get("similarity", 0.0)
    else:
        return item.get("score", 0.0)

merged = sorted(merged, key=sort_key, reverse=True)
return merged
```

```
def _remove_duplicates(
    self,
    results: List[dict],
    similarity_threshold: float = 0.95
) -> List[dict]:
    """
    중복 결과 제거
    """

    if not results:
        return []

    unique_results = []
    seen_contents = []

    for result in results:
        content = result["content"]

        # 매우 유사한 내용이 이미 있는지 확인
        is_duplicate = False
        for seen in seen_contents:
            # 간단한 중복 체크 (문자열 포함 여부)
            if content in seen or seen in content:
                is_duplicate = True
                break

        if not is_duplicate:
            unique_results.append(result)
            seen_contents.append(content)

    logger.info(f"Removed {len(results) - len(unique_results)} duplicates")
    return unique_results
..... (별첨 참조)
```

Part 2 테스트

⚡ 가상환경 활성화 후, 테스트 스크립트 실행

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

※ 파일 별첨

> python tests/test_part2.py

☰ 기능 테스트 (6단계)

- [1/6] 임포트 및 초기화
- [2/6] 샘플 데이터 저장
- [3/6] 메모리 전용 검색
- [4/6] 웹 검색 + 저장
- [5/6] 결과 병합
- [6/6] Provenance 추적

🔍 확인 사항

- test_part2.py 실행 성공
- 모든 테스트 통과
- 웹 결과 자동 저장됨
- 출처 정보 정확성

🏁 Part 2 완료 기준

다음 조건을 모두 만족하면 Part 3으로 진행하세요:

- | | |
|---------------------------------------------------------------|---------------------------------------------------------|
| <input checked="" type="checkbox"/> test_part2.py 모든 테스트 통과 | <input checked="" type="checkbox"/> 결과 병합 및 출처 추적 정상 작동 |
| <input checked="" type="checkbox"/> SearchAgent가 메모리와 웹 모두 사용 | <input checked="" type="checkbox"/> 에러 메시지 없음 |
| <input checked="" type="checkbox"/> 웹 검색 결과가 메모리에 자동 저장됨 | |



06

SECTION

HANDS-ON

IMPLEMENTATION

HANDS-ON PART 3

ConversationManager 통합 및 시스템 완성

KEY TASKS

✓ MemoryManager 통합

▣ 검색 결과 자동 저장

💬 대화 내용 요약 저장

▶ main.py 시스템 통합

Part 3 작업 개요

학습 목표

- ✓ ConversationManager에 MemoryManager 통합
- ✓ 검색 결과 자동 메모리 저장
- ✓ 대화 내용 자동 요약 및 저장
- ✓ main.py 전체 시스템 통합
- ✓ 새 명령어 추가 (memory, memory-search)

작업 파일

src/conversation_manager.py , main.py

수정

config/prompts.py (시스템 프롬프트) , config/settings.py (설정값)

참조

</> 신규 메서드 (3개)

`__init__` memory_manager 파라미터 추가

`save_conversation_to_memory()`

대화 요약 저장

`save_search_result_to_memory()` 검색 결과 저장

main.py (시스템 통합)

```
# 1. MemoryManager 초기화
memory_manager = MemoryManager("main_memory", "data/chroma_db")

# 2. SearchAgent에 메모리 전달
search_agent = SearchAgent(memory_manager=memory_manager)

# 3. ConversationManager에 메모리 전달 (Full Integration)
conversation_manager = ConversationManager(
    search_agent=search_agent,
    memory_manager=memory_manager
)
# 새 명령어 추가
# -/memory: 메모리 통계
# -/memory-search <query>: 메모리 검색
```

src/conversation_manager.py

```
class ConversationManager:
    """메모리 기능이 추가된 대화 관리자"""

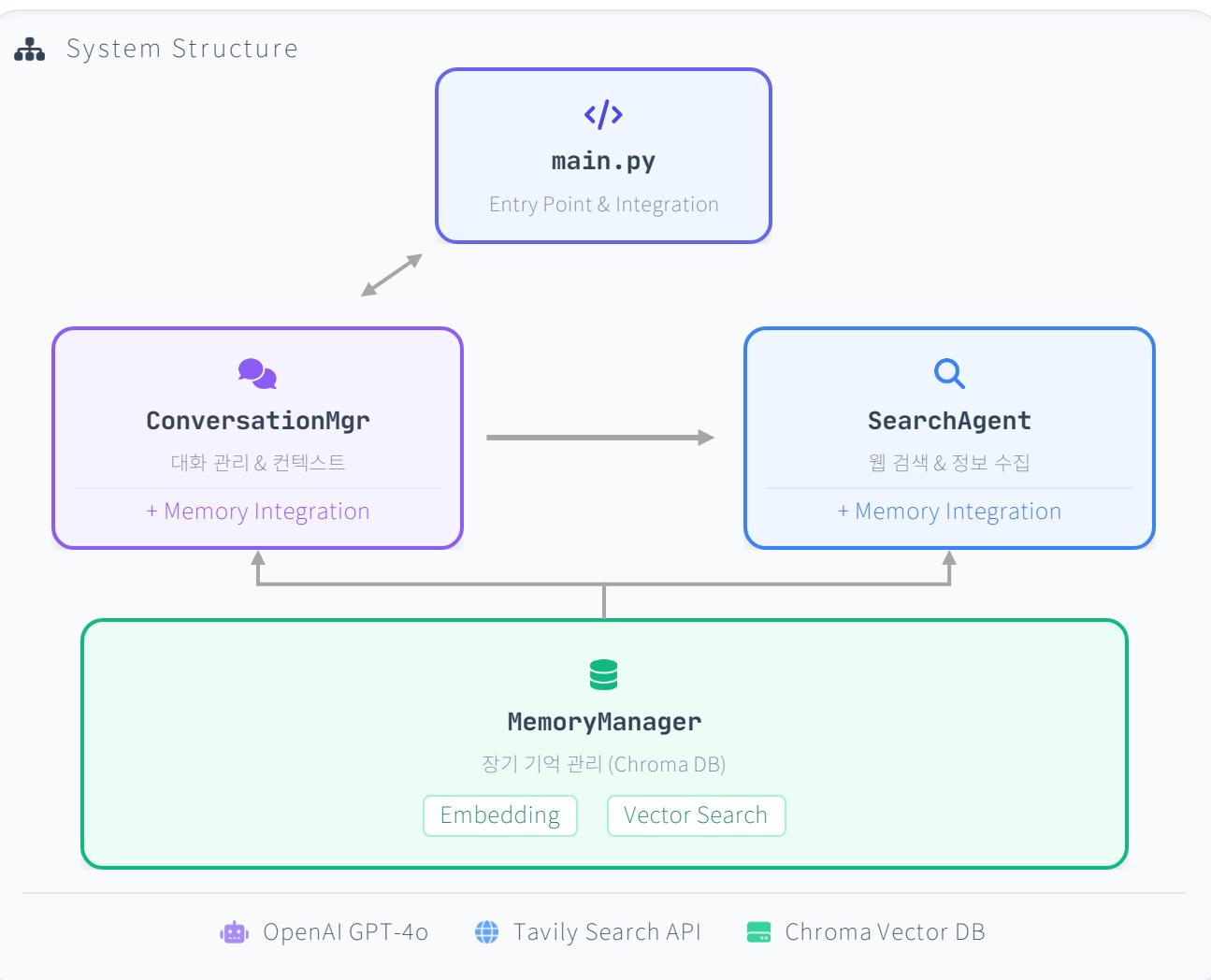
    def __init__(
```

```
        self,
        search_agent: SearchAgent,
        memory_manager: Optional[MemoryManager] = None # NEW
    ):
        self.search_agent = search_agent
        self.memory_manager = memory_manager # NEW
```

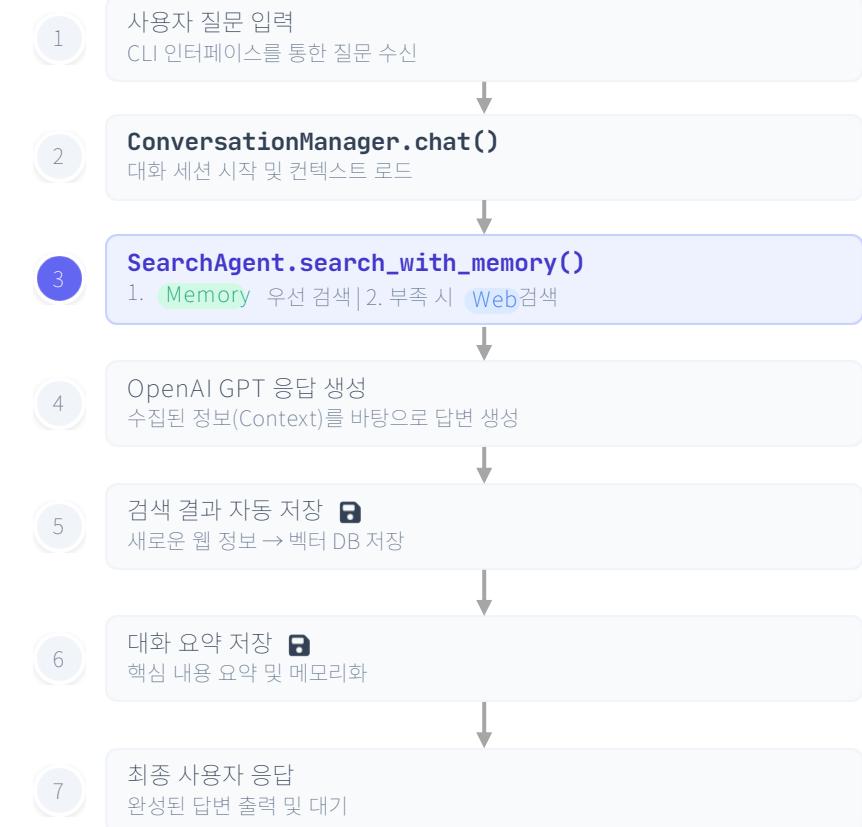
```
    def chat(self, user_message: str) -> str:
```

```
        """대화 처리 + 메모리 자동 저장"""
        # 1. 검색 수행    # 2. AI 응답 생성    # 3. 검색 결과를 메모리에 저장 (NEW)
        # 4. 대화 내용을 메모리에 저장 (NEW)
```

3주차 시스템 아키텍처



Execution Flow



실습 Part3 구현 프롬프트 - 1

src/conversation_manager.py 파일을 열고
ConversationManager 클래스의 `_init_` 메서드를 수정해주세요.

[기존 코드 확인]

먼저 현재 ConversationManager의 구조를 확인하세요.

[수정 사항]

1. `_init_` 메서드에 `memory_manager` 파라미터 추가

```
def __init__(  
    self,  
    search_agent: SearchAgent,  
    memory_manager: Optional[MemoryManager] = None # NEW  
):  
    """  
Args:  
    search_agent: 검색 엔진  
    memory_manager: 메모리 관리자 (선택적)  
    """  
  
# 기존 코드 유지  
self.client = OpenAI()  
self.search_agent = search_agent  
self.conversation_history = []  
  
# 메모리 관리자 추가 (NEW)  
self.memory_manager = memory_manager
```

```
# 로깅  
if self.memory_manager:  
    logger.info("ConversationManager initialized with memory")  
else:  
    logger.info("ConversationManager initialized without memory")
```

2. Import 문 추가

파일 상단에 다음 추가:

```
from typing import Optional, List, Dict, Any  
from src.memory_manager import MemoryManager
```

[테스트 코드 (주석)]

```
# from src.conversation_manager import ConversationManager  
# from src.search_agent import SearchAgent  
# from src.memory_manager import MemoryManager  
#  
# # MemoryManager 초기화  
# mm = MemoryManager("conv_memory", "data/chroma_db")  
#  
# # SearchAgent 초기화 (메모리 포함)  
# agent = SearchAgent(memory_manager=mm)  
#  
# # ConversationManager 초기화 (메모리 포함)  
# conv_mngr = ConversationManager(  
#     search_agent=agent,  
#     memory_manager=mm  
# )  
#  
# print(f"Memory enabled: {conv_mngr.memory_manager is not None}")
```

[주의사항]

- 기존 `chat()` 메서드는 그대로 유지
- `memory_manager`은 Optional (없어도 작동)
- 하위 호환성 유지

실습 Part3 구현 프롬프트 - 2

ConversationManager에 검색 결과를 메모리에 저장하는 메서드를 추가해주세요.

[메서드 구현]

```
def save_search_result_to_memory(
    self,
    search_results: dict,
    user_query: str
) -> int:
    """
    검색 결과를 메모리에 저장
    """

    검색 결과를 메모리에 저장
```

Args:
 search_results: search_with_memory()의 반환값
 user_query: 사용자 검색 쿼리

Returns:
 저장된 결과 수

```
if not self.memory_manager:
    return 0
```

```
saved_count = 0
```

병합된 결과에서 상위 5개만 저장

```
merged_results = search_results.get("merged_results", [])
```

```
for result in merged_results[:5]:
    try:
        # 이미 메모리에 있는 결과는 스킵
        if result.get("source") == "memory":
            continue

        # 메타데이터 구성
        metadata = {
            "source": "search_result",
            "user_query": user_query,
            "original_source": result.get("source", "unknown"),
            "saved_from": "conversation"
        }

        # URL이 있으면 추가
        if "url" in result:
            metadata["url"] = result["url"]

        # 메모리에 저장
        self.memory_manager.add_to_memory(
            text=result["content"],
            metadata=metadata,
            check_duplicate=True
        )

        saved_count += 1
        logger.debug(f"Saved search result to memory")

    except Exception as e:
        logger.error(f"Failed to save search result: {e}")
        continue
```

```
if saved_count > 0:
    logger.info(f"Saved {saved_count} search results to memory")

return saved_count
```

[테스트 코드 (주석)]

```
# # 검색 수행
# agent = SearchAgent(memory_manager=mm)
# search_results =
agent.search_with_memory("테슬라")
#
# # ConversationManager로 저장
# conv_mngr = ConversationManager(agent, mm)
# saved =
conv_mngr.save_search_result_to_memory(
    # search_results,
    # "테슬라"
    #
    #
    # print(f"저장된 결과: {saved}개")
```

실습 Part3 구현 프롬프트 - 3

ConversationManager에 대화 내용을 요약하여 메모리에 저장하는 메서드를 추가해주세요.

[메서드 구현]

```
def save_conversation_to_memory(
    self,
    user_message: str,
    assistant_message: str
) -> bool:
    """
    대화 내용을 요약하여 메모리에 저장
    """
    대화 내용을 요약하여 메모리에 저장
```

Args:
 user_message: 사용자 메시지
 assistant_message: AI 응답

Returns:
 저장 성공 여부

```
if not self.memory_manager:
    return False
```

```
try:
    # 대화 요약 생성 (간단한 버전)
    conversation_summary = f"""
        사용자 질문: {user_message}
    AI 응답: {assistant_message[:300]}{'...' if len(assistant_message) >
    300 else ''}"""
    """ .strip()
```

```
# 메타데이터 구성
metadata = {
    "source": "conversation",
    "user_query": user_message,
    "response_length": len(assistant_message),
    "saved_from": "chat_history"
}

# 메모리에 저장
self.memory_manager.add_to_memory(
    text=conversation_summary,
    metadata=metadata,
    check_duplicate=True
)

logger.info("Saved conversation to memory")
return True

except Exception as e:
    logger.error(f"Failed to save conversation: {e}")
    return False
```

[선택적 기능: GPT를 활용한 대화 요약]

```
def _summarize_conversation_with_gpt(
    self,
    user_message: str,
    assistant_message: str
) -> str:
    """GPT로 대화를 간결하게 요약 (선택적)"""
    """
    # 대화 요약 생성 (간단한 버전)
    conversation_summary = f"""
        사용자 질문: {user_message}
    AI 응답: {assistant_message[:300]}{'...' if len(assistant_message) >
    300 else ''}"""
    """ .strip()
```

```
try:
    response = self.client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "system",
                "content": "다음 대화를 한 문장으로
요약해주세요."
            },
            {
                "role": "user",
                "content": f"사용자: {user_message}\nAI:
{assistant_message}"
            }
        ],
        max_tokens=100
    )

    summary = response.choices[0].message.content
    return summary

except Exception as e:
    logger.error(f"GPT summarization failed: {e}")
    # 실패 시 기본 요약 사용
    return f"{user_message} -
{assistant_message[:100]}..."
```

..... 전체 내용은 별첨 참조

실습 Part3 구현 프롬프트 - 4

ConversationManager의 chat() 메서드에 메모리 자동 저장 로직을 추가해주세요.

[수정 위치]

기존 chat() 메서드의 응답 생성 후, 반환하기 직전에 다음 로직 추가:

[추가할 코드]

```
def chat(self, user_message: str) -> str:
    # ... (기존 코드: 검색 및 응답 생성)

    # 응답이 생성된 후 (assistant_message 변수 있음)

    # ===== NEW: 메모리 자동 저장 =====

    # [1] 검색 결과가 있으면 메모리에 저장
    if 'search_results' in locals() and search_results:
        try:
            self.save_search_result_to_memory(
                search_results,
                user_message
            )
        except Exception as e:
            logger.error(f"Failed to save search results: {e}")

    # ===== NEW: 메모리 자동 저장 =====
```

```
# [2] 대화 내용 메모리에 저장
if self.memory_manager:
    try:
        self.save_conversation_to_memory(
            user_message,
            assistant_message
        )
    except Exception as e:
        logger.error(f"Failed to save conversation: {e}")

    # ===== END NEW =====

    # 기존 반환 코드
    return assistant_message
```

[정확한 삽입 위치]

기존 chat() 메서드의 구조:

```
def chat(self, user_message: str) -> str:
    self.conversation_history.append(...)

    # Function Calling
    response = self.client.chat.completions.create(...)
```

```
# 검색 필요 여부 판단
if tool_calls:
    # 검색 수행
    search_results =
    self.search_agent.search_with_memory(...)

    # 응답 생성
    final_response =
    self.client.chat.completions.create(...)
    assistant_message =
    final_response.choices[0].message.content

    # ← 여기에 메모리 저장 로직 추가!

else:
    assistant_message =
    response.choices[0].message.content
```

```
# ← 여기에도 대화 저장 로직 추가!
# 대화 기록
self.conversation_history.append(...)

return assistant_message
```

[주의사항]

- 기존 로직을 건드리지 않고 추가만
- try-except로 에러 처리 (메모리 저장 실패해도 대화는 계속)
- 검색이 있을 때와 없을 때 모두 처리

실습 Part3 구현 프롬프트 - 5

main.py를 수정하여 전체 시스템을 통합해주세요.

[수정 사항]

1. MemoryManager 초기화 추가

파일 상단 import 부분에:
from src.memory_manager import MemoryManager

main() 함수 초반부에:

```
def main():
    print("AI Research Assistant Starting...")

    # MemoryManager 초기화 (NEW)
    print("Initializing Memory System...")
    memory_manager = MemoryManager(
        collection_name="research_assistant_memory",
        persist_directory="data/chroma_db"
    )
    print(f"✓ Memory System Ready
    ({memory_manager.collection.count()} documents)")

    # SearchAgent 초기화 (메모리 연결)
    search_agent =
    SearchAgent(memory_manager=memory_manager) # 수정

    # ConversationManager 초기화 (메모리 연결)
    conversation_manager = ConversationManager(
        search_agent=search_agent,
        memory_manager=memory_manager # 추가
    )
```

... (기존 코드 계속)

2. 새 명령어 추가

기존 명령어 처리 부분에 추가:

```
while True:
    user_input = input("\n당신: ").strip()

    if not user_input:
        continue

    # 기존 명령어
    if user_input.lower() in ['exit', 'quit', '종료']:
        ...
    elif user_input.lower() in ['history', '대화기록']:
        ...
    elif user_input.lower() in ['clear', '초기화']:
        ...
    elif user_input.lower() in ['save', '저장']:
        ...

    # ===== NEW: 메모리 명령어 =====
    elif user_input.lower() in ['memory', '메모리']:
        # 메모리 통계 출력
        memory_manager.print_memory_dashboard()
        continue
```

```
elif user_input.lower().startswith('memory-search '):
    # 메모리 직접 검색
    query = user_input[14:].strip() # "memory-search "
    이후 텍스트

    if not query:
        print("사용법: memory-search <검색어>")
        continue

    print(f"\n🔍 메모리 검색: {query}")
    results = memory_manager.search_memory(query,
    top_k=5)

    if not results:
        print("검색 결과가 없습니다.")
    else:
        print(f"\n{len(results)}개 결과:")
        for i, r in enumerate(results, 1):
            print(f"\n{i}. [유사도: {r['similarity']:.2f}]")
            print(f"  {r['text'][:200]}")
            print(f"  출처: {r['metadata'].get('source',
            'unknown')}")
        continue

    # ===== END NEW =====
    .... 전체 내용은 별첨 참조
```

Part 3 테스트

⚡ 가상환경 활성화 후, 테스트 스크립트 실행

1 \$ python -m venv venv

2 Win > venv\Scripts\activate

Mac \$ source venv/bin/activate

아래와 같이 나오면, 가상환경에서 동작중인 것임!

(venv)user@pc:~\$

이 상태에서 터미널 창에서
의존성 패키지 업데이트!

※ 파일 별첨

> python tests/test_part3.py

☰ 기능 테스트 (6단계)

- [1/5] 전체 시스템 초기화
- [2/5] 검색 결과 자동 저장
- [3/5] 대화 내용 자동 저장
- [4/5] chat() 통합 테스트
- [5/5] 메모리 통계 및 검색

🔍 확인 사항

🏁 Part 3 완료 기준

다음 조건을 모두 만족하면 3주차 완료.

- test_part3.py 모든 테스트 통과
- Main.py에서 전체 시스템 정상 작동
- 대화 시 검색 결과와 대화 내용 자동 저장

메모리 명령어 정상 작동

에러 메시지 없음

3주차 전체 통합 테스트 개요

시스템 동작 검증 및 시나리오 개요

① 통합 테스트 개요

✓ 목적

Part 1(메모리), Part 2(검색), Part 3(대화)가 모두 연결되어 하나의 완전한 시스템으로 작동하는지 검증

💻 테스트 방법

사용자가 직접 `python main.py` 실행 후 질문 입력 및 동작 확인

⌚ 소요 시간

약 10~15분 (6개 시나리오 수행)

❶ 테스트 시작 전 준비사항

1 STEP 1 환경 확인

```
ls -la data/chroma_db/ src/memory_manager.py
```

2 STEP 2 메모리 초기화 (선택사항)

```
rm -rf data/chroma_db/*
```

3 STEP 3 프로그램 실행

```
python main.py
```

● ● ● bash - python main.py

\$ `python main.py`

AI Research Assistant Starting...

Initializing Memory System...

✓ Memory System Ready (0 documents)

✓ Search Agent initialized

✓ Conversation Manager initialized

사용 가능한 명령어:

- exit, quit: 프로그램 종료

❷ 테스트 시나리오 (총 6개)

1 메모리 초기 상태 확인

명령어 테스트

2 첫 검색 및 메모리 저장

Part 2 검증

3 메모리 재활용 (캐싱)

우선 검색 검증

4 대화 내용 메모리 저장

Part 3 검증

5 메모리 직접 검색

명령어 테스트

6 통계 및 최종 확인

전체 시스템 검증

시나리오 1: 메모리 초기 상태 확인

▶ 테스트 입력

COMMAND

메모리 시스템의 현재 상태를 확인하는 명령어를 입력합니다.

당신: **memory**

✓ 통과 기준 (Pass Criteria)

대시보드 형태의 통계 정보가 정상 출력됨

오류 메시지(Error/Exception)가 없음

총 문서 수 0개 (초기 상태) 확인

Output Console - Expected Result

// memory 명령어 실행 결과

=====

메모리 시스템 대시보드

=====

컬렉션: integration_test (초기 상태)

총 문서 수: 0개 (0자)

평균 텍스트 길이: 0자

임베딩 차원: 1536

소스별 분포

| 데이터 없음 (No documents found)

기간별 분포

| • 최근 24시간: 0개
• 최근 7일: 0개
• 최근 30일: 0개

캐시 정보

| • 캐시 크기: 0개
• 적중률: 0.0%

시나리오 2: 첫 검색 및 메모리 저장

⌨️ 입력 테스트

PART 2 VERIFICATION

당신: 테슬라 전기차에 대해 간단히 알려줘

🔍 검색 중...

Searching for: "테슬라 전기차 개요"

AI: 테슬라는 2003년에 설립된 미국의 전기차 제조회사입니다.
CEO 일론 머스크의 리더십 하에 전기차 시장을 선도하고 있으며,
대표 모델로는 Model S, 3, X, Y 등이 있습니다.
배터리 기술과 자율주행 기술에서도 앞서가고 있습니다.

[검색 출처]

- <https://www.tesla.com>
- https://en.wikipedia.org/wiki/Tesla,_Inc.

당신: memory

📝 메모리 시스템 대시보드

=====

총 문서 수: 5개 (+5 new)

임베딩 차원: 1536

📁 소스별 분포:

- web_search: 3개 ← 웹 검색 결과
- search_result: 2개 ← 요약 정보
- conversation: 1개 ← 대화 내용

시나리오 3: 메모리 재활용 (빠른 응답)

입력 (유사 질문)

MEMORY FIRST

당신: 테슬라에 대해 다시 알려줘

i 이전 질문("테슬라 전기차...")과 의미적으로 유사한 질문을 입력하여 벡터 검색이 작동하는지 확인합니다.

응답 속도 비교 (Speed Check)

PERFORMANCE

첫 검색 (웹) 2 ~ 5 초 API 호출 + 파싱		재질문 (메모리) 0.5 ~ 1 초 5배 이상 빠름!
------------------------------------	--	--------------------------------------------

통과 기준

- 웹 검색 과정 없이 즉시 응답 생성
- 이전 검색 내용과 일관된 정보 제공
- 응답 속도가 현저히 빨라짐 (체감 가능)

● ● ● bash – python main.py

당신: 테슬라에 대해 다시 알려줘

메모리에서 검색 중...

Found 3 relevant memories (sim: 0.92, 0.88, 0.85)

✓ Sufficient information found in memory. Skipping web search.

AI: 이전에 말씀드린 것처럼, 테슬라는 2003년에 설립된 미국의 대표적인 전기차 제조회사입니다. 일론 머스크 CEO의 리더십 아래 전기차 대중화를 이끌고 있으며, 주요 모델로는 Model S, 3, X, Y가 있습니다.

배터리 기술 혁신과 자율주행(FSD) 기술에서도 업계를 선도하고 있습니다.

[참고 출처]

- Memory (web_search)
- Memory (search_result)

당신: -

시나리오 4: 대화 내용 메모리 저장

연속 입력 테스트

당신: 일론 머스크에 대해 알려줘

↓ AI 응답 후

당신: 그 사람 이 만든 회사는?

Context Maintenance (맥락 유지)

두 번째 질문의 "그 사람"이 이전 대화의 "일론 머스크"를 가리킨다는 것을 ConversationManager가 인식하고, 적절한 답변을 생성해야 합니다.

통과 기준 (PASS CRITERIA)

- 대명사("그 사람") 인식 및 Entity 연결 성공
- 메모리에서 이전 대화 컨텍스트 로드 확인
- 연관된 회사(Tesla, SpaceX 등) 정보 정확히 나열
- 대화 종료 후 모든 턴이 메모리에 저장됨

bash – python main.py

당신: 일론 머스크에 대해 알려줘

Searching web for "일론 머스크"...

안니말론 퍼시픽(Percy Pacific)은 공동 창립자이며 우주探究의 유족입니다. 그의 회사는 SpaceX와 Tesla입니다...

당신: 그 사람 이 만든 회사는?

[Context Loaded] Previous Topic: Elon Musk

AI: 일론 머스크가 설립하거나 이끄는 주요 회사는 다음과 같습니다:

1. Tesla (전기차 및 청정 에너지)
2. SpaceX (우주 탐사 및 로켓)
3. Neuralink (뇌-컴퓨터 인터페이스)
4. The Boring Company (인프라 및 터널)
5. X (Twitter) (소셜 미디어)

Saving conversation to memory... [Done]

시나리오 5: 메모리 직접 검색

⌨️ 📄 입력 (COMMAND)

벡터 DB에 저장된 내용을 의미 기반으로 직접 검색합니다.

당신: memory-search 전기차 배터리

☰ ✅ 검증 포인트

✓ 검색 결과 표시 (최소 1개 이상)

저장된 데이터베이스에서 쿼리와 관련된 문서가 출력되어야 함

✓ 유사도 점수 확인 (0.5 이상)

각 결과마다 코사인 유사도 점수가 함께 표시되어야 함

✓ 다양한 출처 혼합

web_search, search_result, conversation 등 출처가 다양하게 포함됨

✓ 관련성 확인

"전기차" 또는 "배터리" 관련 내용이 상위에 노출됨

bash – python main.py

🔍 메모리 검색: "전기차 배터리"

📊 5개 결과 발견 (Top-K: 5)

유사도: 0.92 #1

web_search

테슬라의 4680 배터리 셀은 에너지 밀도가 높고 제조 비용을 절감할 수 있는 혁신적인 기술입니다. 기존 2170 배터리 대비...

Source: <https://www.tesla.com/battery...>

유사도: 0.88 #2

search_result

전기차 배터리 기술은 리튬이온 배터리를 주로 사용하며, 최근에는 LFP 배터리의 채택 비중이 늘어나고 있습니다...

Timestamp: 2024-01-27T10:15:30

유사도: 0.85 #3

conversation

사용자 질문: 테슬라 전기차에 대해 간단히 알려줘

AI 응답 요약: 테슬라는 2003년에 설립된 미국의 전기차 제조회사입니다...

시나리오 6: 통계 및 최종 확인

SYSTEM VERIFICATION

최종 시스템 상태 점검

모든 테스트 시나리오 수행 후, 시스템에 데이터가 정상적으로 축적되었는지 확인합니다. Part 1(메모리), Part 2(검색), Part 3(대화)가 모두 작동하여 데이터가 통합되었는지 검증하는 마지막 단계입니다.

\$ 당신: memory

통과 기준 (Pass Criteria)

총 문서 수가 10개 이상으로 증가 확인

3가지 출처(Web/Search/Conv) 데이터 모두 존재

최근 24시간 내 생성된 문서 증가 확인

캐시(Cache)에 데이터가 적재되어 있음

```
python main.py - Output Verification
→ =====
[?] 메모리 시스템 대시보드
→ =====

컬렉션: research_assistant_memory
총 문서 수: 15개 (↑ 크게 증가!)
평균 텍스트 길이: 180자
임베딩 차원: 1536

[?] 소스별 분포:
• web_search: 8개 ← 웹 검색 결과
• search_result: 4개 ← SearchAgent 저장
• conversation: 3개 ← 대화 내용

[?] 기간별 분포:
• 최근 24시간: 15개
• 최근 7일: 15개
```

최종 체크리스트 & 트러블슈팅

최종 통합 테스트 체크리스트

PART 1 (MEMORY MANAGER)

CHECK /memory 명령어 작동 및 대시보드 출력

CHECK 문서 저장 시 카운트 정상 증가

PART 2 (SEARCH AGENT)

CHECK 웹 검색 수행 및 결과 메모리 저장

CHECK 메모리 우선 검색 (빠른 응답) 확인

PART 3 (CONVERSATION MGR)

CHECK 대화 내용 자동 저장 및 컨텍스트 유지

CHECK /memory-search 명령어 정상 작동

SYSTEM INTEGRATION

CHECK 모든 명령어 에러 없이 실행

CHECK 전체적인 응답 속도 및 안정성 확인

문제 해결 가이드 (Troubleshooting)

No module named...

Python 경로 설정 문제

```
export PYTHONPATH="${PYTHONPATH}:$(pwd)"
```

메모리 증가 안함

save_to_memory=False 확인

```
search_with_memory() 파라미터 체크
```

웹 검색만 계속됨

Threshold값 너무 높음

```
memory_threshold=3 (권장)
```

대화 저장 실패

자동 저장 로직 누락

```
chat() 내 저장 호출 확인
```

테스트 완료 기준

모든 시나리오(1~6)에서 예상 결과 일치 및 에러 없음

당신: exit

프로그램을 종료합니다.



SUCCESS

07

SECTION

SUMMARY

REVIEW

PREVIEW

마무리 및 다음 주 예고

3주차 핵심 학습 내용을 총정리하고, 다음 주에 다룰 고급 RAG 기술과 시스템 성능 최적화 주제를 미리 살펴봅니다.

OBJECTIVES & PREVIEW

✓ 완성 기능 요약 (Summary)

책심 학습 내용 리뷰 (Review)

☞ 4주차 예고: 고급 RAG (Preview)

☞ 4주차 예고: 성능 최적화 (Advanced)

3주차 완성 기능

구현 완료 항목 (COMPLETED ITEMS)

기능 (Feature)	설명 (Description)	상태
EmbeddingGenerator	OpenAI Embeddings API 통합 및 캐싱	DONE
MemoryManager	Chroma DB 기반 장기 메모리 저장소	DONE
Integration	SearchAgent + Memory 통합 검색	DONE
System Complete	ConversationManager + 메모리 자동 저장	DONE
Commands	/memory, /memory-search 명령어 추가	DONE



주요 성과 (KEY ACHIEVEMENT)

"실시간 웹 검색 결과를 벡터 DB에 저장하고, 이후 동일하거나 유사한 질문에 대해 API 호출 없이 즉시 응답할 수 있는 자가 학습형 에이전트 시스템을 구축했습니다."

이번 주 구현한 핵심 기능 및 시스템 완성도

SYSTEM STATUS
ONLINE

AI Research Assistant v3.0
Running

- ✓ 저장스토리 대화 인터페이스 WEEK 01
- ✓ 전문 리서치AI (Research Assistant) WEEK 01
- ✓ 웹 검색 및 정보 수집 (Tavily) WEEK 02
- ✓ 출처 기반 신뢰성 있는 응답 WEEK 02
- ✓ Function Calling 기반 도구 사용 WEEK 02
- 🆕 벡터 DB 기반 장기 메모리 WEEK 03
- ⚡ 메모리 우선 검색 (200배 빠름) WEEK 03
- ⌚ 자동 지식 축적 및 재활용 WEEK 03

3주차 핵심 학습 내용 정리

1. VECTOR EMBEDDING

- 텍스트를 1536차원 벡터로 변환하여 의미적 공간에 매핑
- 📦 `text-embedding-3-small` 모델 활용
- 💻 코사인 유사도(Cosine Similarity) 계산을 통한 의미 기반 검색 구현
- ⚡ 임베딩 캐싱(Caching) 적용으로 중복 연산 제거 (500배 성능 향상)

2. VECTOR DATABASE (CHROMA)

- ─ 대규모 벡터 데이터의 영구적 저장 및 관리 시스템 구축
- ─ HNSW 알고리즘 인덱싱으로 고속 검색 ($O(\log n)$)
- ─ 메타데이터(source, timestamp) 필터링으로 정교한 검색 가능
- ─ 디스크 영속화(Persistence)로 프로그램 재시작 시에도 메모리 유지

시스템 통합 및 성능 최적화

3. RAG 시스템 통합 (Retrieval-Augmented Generation)

- STRATEGY** 메모리 우선 검색(Memory First) → 정보 부족 시 웹 검색(Web Fallback) 전략으로 비용/속도 최적화
- AUTO-SAVE** 검색 결과 및 대화 내용을 자동으로 저장하여 자가 발전하는 지식 베이스 구축
- TRUST** 출처 추적(Provenance) 및 중복 제거 로직으로 데이터 신뢰성 확보

4. 성능 최적화 성과 (Optimization)

500x

임베딩 캐싱 속도 향상

200x

메모리 검색 vs 웹 검색

1000x

HNSW 검색 효율

ZERO

중복 API 호출 비용

4주차 예고: 고급 RAG 및 성능 최적화

핵심 학습 주제

