



LIRA
Emotion + Memory
RAG system

박현우 | AI Engineer

LIRA 개발동기 : LLM Application의 한계.

1. 맥락 소멸

세션 종료 → 새 세션에서 기억 부재 → **사용자 몰입도 하락.**

2. 성능 저하

세션이 길어질수록 → **답변 품질 및 응답속도 저하.**

3. 감정 부재

패턴추론 답변 → **상담은 가능하나, 사용자의 감정을 이해할 근거는 없음.**

원인정의 : LLM에는 기억과 감정이 없다.

1. 맥락소멸·성능저하 원인

- '기억'이 없기 때문에 세션전환이 원활하지 않고,
- 세션이 길어질수록 성능 저하 발생
→ keyword : 기억

2. 감정부재 원인

- LLM은 실제로 '감정을 이해하는 근거'가 없다.(패턴 추론 답변)
→ keyword : 감정

개발방향: 기억 + 감정분석으로 몰입과 신뢰성을 확보한다.

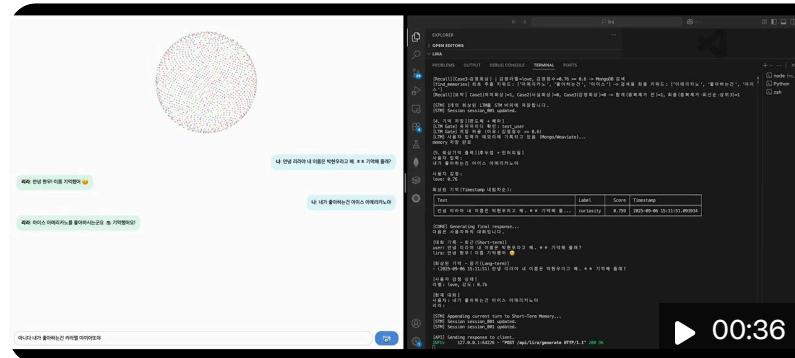
1. '기억'을 통한 몰입 유지

- 개인별 대화 저장·회상(세션 전환 불편 해소)
→ 예) “지난번에 라떼를 좋아하셨죠”

2. '감정분석'을 통한 근거 있는 응답 및 가드레일

- 사용자의 감정에 맞는 응답 + 위험 대화 차단(감정근거를 통한 응답)
→ 예) user : (**emotion** : happy, **score**: 0.7) “아 오늘 정말 기분 좋네 ㅎ ㅎ”
LLM : (**emotion**, **score**를 확인 후) “맞아요! 오늘 아침에 칭찬받아서 기분이 좋다고 말해줬어요. 😊”
→ 예) user : (**emotion** : sadness, **score**: 0.7) “요즘 다 힘들고 지쳤어...” (2-3회 반복 발화)
LLM : (반복 감지 후) “요즘 계속 힘들다고 말씀해주셨네요.
괜찮으시다면 휴식을 권해드리고, 필요하다면 전문가와 상의해보는 것도 좋을 것 같아요.”

시연 화면



YouTube

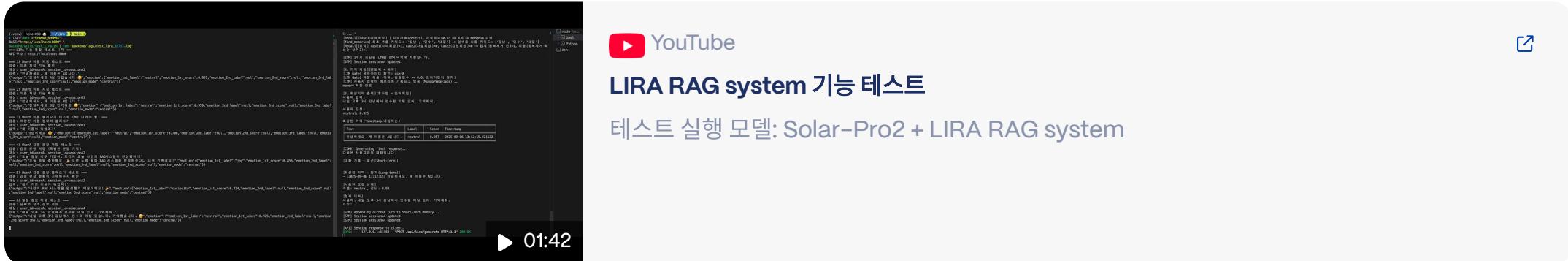


LIRA Emotion + Memory RAG system의 시연영상

테스트 실행 모델: Solar-Pro2 + LIRA RAG system

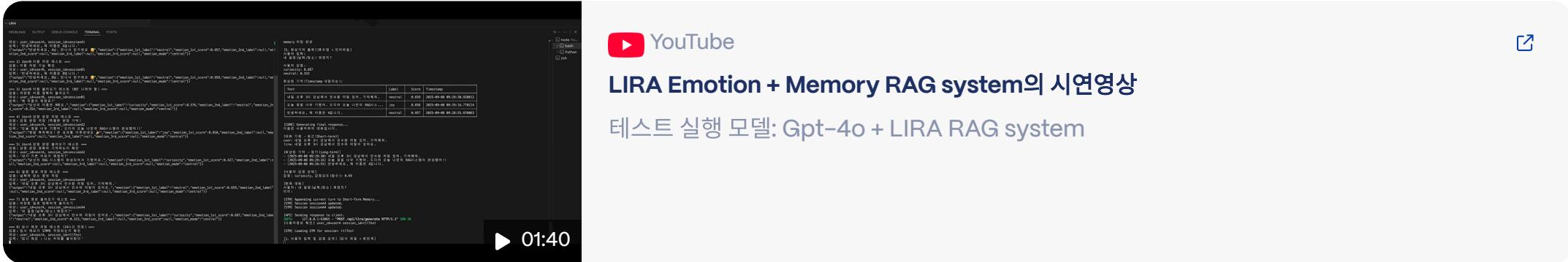
실제 '대화입력'과 'API처리'를 클라이언트/API 디버깅 화면에서 확인하는 모습

테스트 화면(model: Solar-Pro 2 + LIRA RAG)



기능 테스트 결과

테스트 화면(model: GPT4o + LIRA RAG)



기능 테스트 결과

테스트 내용

- 항목별 유저와 세션은 다릅니다.
- 용어의 개념은 10~12페이지를 참고하시기 바랍니다.
- 스크립트 내용은 아래경로의 파일을 확인하시기 바랍니다.
- backend/utils/test_lira.sh

번호	테스트 목적	유저/세션 아이디	기대 결과
1~3	이름 저장 및 불러오기 확인	userA(sessionA1), userB(sessionB1)	각 사용자의 이름이 올바르게 저장·회상됨 (데이터 격리 확인)
4~5	감정 문장 저장 및 불러오기 확인	userA(sessionA2)	감정 문장이 저장되고 회상 시, 요지 그대로 반환됨
6~7	일정 정보 저장 및 불러오기 확인	userA(sessionA4)	날짜장소 정보가 정확히 저장·회상됨
8~9	임시 메모 저장 및 불러오기 확인 (STM)	userA(ttlTest)	Redis STM에 임시 메모 저장 및 즉시 회상 가능
10	TTL 기능 단축	userA(ttlTest)	임시 메모 TTL을 24h → 10초로 변경
11	TTL 만료 대기 및 확인	userA(ttlTest)	TTL 값이 0 → -2로 변경되며, 임시 메모 삭제 완료
12	장기 기억 초기화 (LTM DB 삭제)	전체 세션	MongoDB/Weaviate의 장기 기억 데이터 삭제
13	TTL 만료 후 임시 메모 회상 실패 확인	userA(ttlTest)	TTL 만료 및 LTM 초기화로 회상 실패해야 함
14	환각 방지 테스트	userC(sessionC1)	존재하지 않는 질문 시 “모르겠다/기억 없음”으로 안전 응답

개발컨셉 : Project LIRA



LIRA?

- **L → Learning**

사용자의 감정을 이해하고,

- **I → Interaction**

사용자와 상호작용하며,

- **R → Recall**

기억을 회상하는

- **A → Architecture**

구조화된 설계.

개발컨셉 : Project LIRA



LIRA?

- 좌측 정육면체 → 3개의 기억DB :
Weaviate, MongoDB, Redis
- 좌측 선 → 연결 :
Module ↔ DB 연결
- 우측 점 → 기억단위(토큰, 백터) :
LLM과 연결
- 우측 웨이브 → 감정 기반 흐름 :
감정 기반 저장·회상

DataBase 구성

→ LIRA는 **RAG** 아키텍처 기반 메모리 시스템을 구성하며, 아래 세 가지 DataBase는 LLM에게 **근거 있는 기억**을 제공합니다.



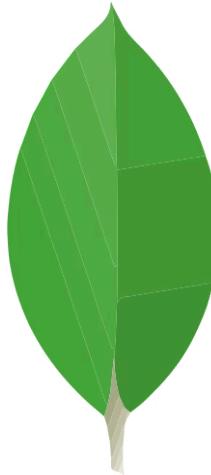
Weaviate

(의미/연상기억)

"이 대화와 비슷한 느낌의 기억은 뭐였지?"



사용자입력과 의미적으로 유사한 기억을
저장하고 검색합니다.



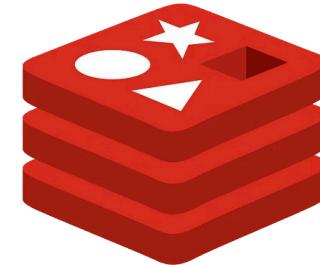
MongoDB

(사실/감정기억)

"이 사용자의 이름은?", "그때의 감정은?"



사용자 입력의 사실과 감정 상태를
저장하고 검색합니다.



Redis

(단기기억)

"방금 전 대화에서 무슨 말을 했지?"



현재 세션의 대화 내용을
저장하고 불러옵니다.

LTM & STM이란?

→ LIRA에서는 DB가 LTM(장기기억) STM(단기기억)으로 분류되어 사용됩니다.

1. 분류

- LTM (Long-Term Memory): Weaviate + MongoDB
- STM (Short-Term Memory): Redis

2. LTM의 역할

LLM의 과거기억(=장기기억)을 저장, 검색합니다.

- 의미 기반 저장/검색 (**Weaviate**)

(“라떼 좋아한다” ↔ “커피 좋아한다” 같은 의미 유사문장을 저장, 검색)

- 사실 기반 저장/검색 (**MongoDB**)

("나는 사과를 좋아해요" → "사과" → "사과주스는 안 좋아해" 같은 사실내용을 저장, 검색)

- 감정 기반 저장/검색 (**MongoDB**)

("오늘 너무 기뻤어" → "joy", "기뻐" → "너가 있어서 기뻐" 같은 감정 내용을 저장, 검색)

LTM & STM이란?

3. STM의 역할

LLM의 현재기억(=단기기억)을 유지, 갱신합니다.

- 대화 유지/갱신 (**Redis**)

STM = [(유저 : "나는 커피를 좋아해", 리라 : "커피를 좋아하시는군요"),

(유저 : "내가 뭘 좋아한다고 했지?", 리라 : "커피를 좋아하신다고 했어요")]

4. TTL(Time To Live)

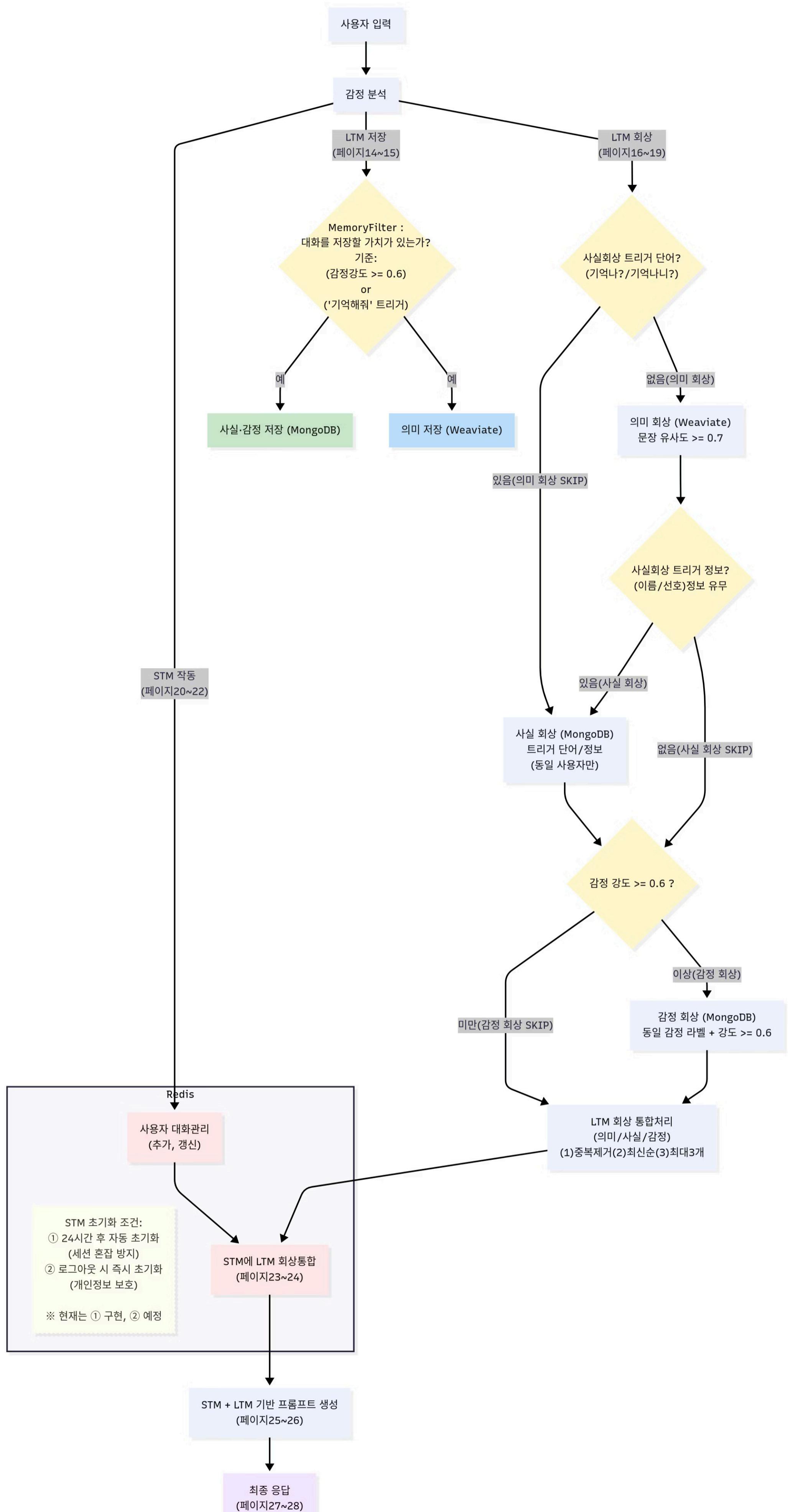
STM의 대화 내용 혼잡을 막기 위해 사용되는 기능

(24h 이후, **STM**자동 초기화)

5. LTM + STM 사용이유

- 과거,현재 대화 & 대화의 감정까지 기억하는 LLM이 응답시 → 사용자의 몰입도 증가 → LLM Application 사용률 증가

API 처리 과정



MemoryFilter란?

1. 역할:

LTM(Weaviate, MongoDB)의 사용자 대화를 저장할지 말지를 판단.

2. 조건: (아래 ①, ② 둘 중 하나라도 해당된다면 저장)

① 감정 점수 ≥ 0.6 (구글GoEmotions로 사용자의 대화를 분석한다. → 감정라벨, 감정점수 추출)
추출 예시) emotion_label:happy, score0.76

② 트리거단어(기억해,저장해)가 사용자 입력에 있을 때 → 예) "기억해줘 나는 커피를 좋아해."

3. 감정점수(score) ≥ 0.6 저장근거:

보통 감정 분류 모델은 0.5 이상이면 감정이 있다고 판단

→ (논문: "GoEmotions: A Dataset of Fine-Grained Emotions")

하지만 저장 정확도를 높이기 위해 0.6 이상일 때만 저장하도록 기준을 강화

MemoryFilter란?

4. 효과:

불필요한/가벼운 문장은 제외, 의미 있는 감정의 내용만 기억

5. 참고 논문:

Demszky et al., GoEmotions: A Dataset of Fine-Grained Emotions (2020, Google) → baseline threshold 0.5

Zhang & Zhou, A Review on Multi-Label Learning (2014, ACM Computing Surveys) → 다중라벨 학습에서 0.5를 표준 사용

```
# LTM 저장 및 감정 기반 회상을 트리거하는 감정 점수 기준
STRONG_EMOTION_GATE: float = 0.6

# 기억 저장 트리거 단어 확인함수
def is_save_trigger_word(user_input: str) -> bool:
    """사용자 입력에 기억 저장과 관련된 트리거 단어가 있는지 확인합니다."""
    trigger_pattern = r"(기억\ss*해(줘|줄래)?|저장\ss*해(줘|줄래)?)"
    return re.search(trigger_pattern, user_input) is not None

# 감정점수를 통해 LTM 저장 여부를 결정하는 함수
def memory_gate(text: str, emotions: List[Dict[str, Any]], user_id: str) -> bool:
    """
    현재 대화를 장기기억(LTM)에 저장할지 여부를 결정하는 함수.
    """

    # 디버그 모드에서 사용자 ID 확인
    if DEBUG:
        print(f"[LTM Gate] Checking for user: {user_id}")

    if not emotions:
        return False

    top_emotion = emotions[0]

    # 1. 감정 점수가 설정된 임계값(0.6) 이상일 경우 저장
    if top_emotion.get("score", 0.0) >= STRONG_EMOTION_GATE:
        if DEBUG:
            print(f"[LTM Gate] 강한 감정({top_emotion['label']}: {top_emotion['score']:.2f})이 감지되어 저장을 허용합니다.")
        return True

    # 2. 감정은 약하지만, 사용자가 명시적으로 저장을 요청한 경우 저장
    if is_save_trigger_word(text):
        if DEBUG:
            print(f"[LTM Gate] 저장 요청 트리거 단어가 감지되어 저장을 허용합니다.")
        return True
```

LTM 회상

→ 회상은 CASE 1~3까지 순차적으로 진행됩니다.

CASE 1. 의미 기반 회상

조건:

일반 입력 (트리거 단어 없음)

처리:

입력 → Cohere 검색 백터 임베딩 → Weaviate 검색 → 유사도: certainty ≥ 0.7 만 채택

결과:

의미상 가까운 과거기억 회상 (예: “라떼 좋아해” \leftrightarrow “커피 좋아해”)

CASE 2. 사실 기반 회상

조건: (아래 ①, ②를 중 하나라도 해당된다면)

- ① 사용자 입력에 “기억나? / 기억나니?” 트리거 단어 포함
- ② 사용자 입력에 “이름 / 선호 정보” 트리거 정보 포함

처리:

입력 → 키워드 추출(kiwiiepy 형태소 분석기) → MongoDB 검색

결과:

키워드를 통한 과거 기억 회상 (예: “내 이름 기억나?” → “이름” → “당신 이름은 A예요”)

CASE 3. 감정 기반 회상

조건: (강한 감정으로 판정될 때)

최상위 감정의 강도 ≥ 0.6

처리:

입력 → 키워드 추출 + 감정라벨 결합 → MongoDB 검색

결과:

감정이 담긴 과거 기억 회상 (예: “어제 너무 행복했어” → “joy”, “행복” → “네, 어제 칭찬받아서 행복하다고 하셨어요 😊”)

CASE 1. 의미 기반 회상

조건: Weaviate 검색 결과 중, 유사도(certainty) > 0.7만 채택

*유사도: 사용자 입력과 LTM(Weaviate)에 저장된 문장의 벡터 간 의미적 유사 → 부연설명: “두 문장이 의미상 얼마나 비슷한가”를 수치로 표현

결과: 유사도 처리조건을 만족한 기억을 LTM에서 회상합니다.

```
for mem in filtered_sem_results:

    # Weaviate의 추가 정보(_additional)에서 유사도 점수(certainty)를 추출한다.
    # 'mem' 변수에 담길 수 있는 데이터 구조 예시
    # mem = {
    #     # --- 1. 기존의 데이터 정보---
    #     "text": "오늘 본 영화 결말이 너무 허무했어.", # 과거 회상된 데이터
    #     "label": "disappointment",
    #     "score": 0.85,
    #     "user_id": "benchmark_user_01",
    #     "session_id": "some_session_id_string",
    #     "timestamp": "2025-08-15T10:30:00Z",
    #     "emotions_json": '[{"label": "disappointment", "score": 0.85}, {"label": "sadness", "score": 0.6}]',

    #     # --- 2. Weaviate로 검색후 추가된 정보 ---
    #     "_additional": {
    #         "certainty": 0.89123, # 현재 사용자 입력과 회상된 사용자 입력의 유사도
    #         "vector": [0.1, 0.2, -0.4, ...] # 과거 회상된 사용자 입력의 데이터의 벡터값 (매우 김)
    #     }
    # }

    # 만약 점수 정보가 없으면, 오류 없이 0.0으로 간주한다.
    certainty = (mem.get("_additional", {}) or {}).get("certainty", 0.0)

    # 만약 유사도 점수가 0.7보다 높아서 의미 있다고 판단되면, 다음 단계를 진행한다.
    if certainty and certainty > 0.7:

        # 점수가 높은 이 기억의 대화내용을 꺼냄.
        text = mem.get("text")

        # 아직 최종 회상 목록(recalled_memories)에 없는 새로운 내용인지 확인한다. (LTM 중복 방지) -> 이전 회상한 LTM 내용의 중복을 막기 위함.
        if text and text not in recalled_memories:

            # 문제가 없다면, 최종 회상 목록(recalled_memories)에 추가한다.
            recalled_memories[text] = mem
```

CASE 2. 사실 기반 회상

조건: 입력 문장에 “기억나 / 기억나니” 단어가 포함되거나 _plan_info_slots() 함수로 “이름/선호/취향” 등의 정보 키워드가 추출될 경우.

처리: 사용자 입력에서 트리거 단어(“기억나/기억나니”) 또는 정보 슬롯의 이름, 선호 취향등이 감지되면, 해당 키워드/슬롯목록을 MongoDB 검색어로 사용하여 사실 기반 기억을 조회합니다.

결과: 조건을 만족한 MongoDB에서 검색을 실행하여, 동일 사용자(user_id)의 과거 대화 중 키워드로 검색한 사실을 회상합니다.

```
# CASE 2: 사실 기반 기억 검색 : MongoDB
# 사용자의 대화 내용에 트리거 키워드를 판단하는 변수를 만든다.
trigger_keywords = ["기억나", "기억나니"]

# _plan_info_slots 함수를 호출하여 사용자 입력에서 '이름', '과일' 과 같은 검색 키워드를 찾아낸다.
slots = _plan_info_slots(user_input)

# 만약 정보 슬롯이 하나라도 감지되었거나, 일반 키워드 중 하나라도 사용자 입력에 포함되어 있다면,
# 사실 기반 검색을 시작합니다.
if slots or any(keyword in user_input for keyword in trigger_keywords):

    # 디버그 모드일 경우, 어떤 슬롯이 감지되었는지 로그를 출력한다.
    if DEBUG:
        print(f"[LTM] 사실 기반 기억 검색 실행... (Slots: {slots})")

    # MongoDB에서 검색할 최종 키워드(search_terms)를 결정한다.
    # 만약 슬롯이 있다면 슬롯 목록을 그대로 사용하고, 슬롯이 없다면 문장 전체에서 키워드를 추출한다.
    search_terms = slots if slots else [extract_keywords(user_input)]

    # 검색할 키워드 목록(search_terms)에 있는 각 키워드(term)로 반복 검색을 수행한다.
    for term in search_terms:

        # 만약 키워드가 비어있다면(추출 실패 등), 다음 키워드로 넘어간다.
        if not term: continue

        # MongoDB에서 해당 키워드(term)로 정확한 사실 기억을 찾아온다.
        mongo_results = confirm_in_mongo(term, user_id=user_id, session_id=session_id)

        # 찾아온 기억 목록(mongo_results)을 하나씩 확인한다.
        for mem in mongo_results:

            # 기억의 실제 내용(text)을 꺼낸다.
            text = mem.get("text")

            # 아직 최종 회상 목록(recalled_memories)에 없는 새로운 내용인지 확인한다. (LTM 중복 방지) -> 이전 회상한 LTM 내용의 중복을 막기 위함.
            if text and text not in recalled_memories:

                # 문제가 없다면, 최종 회상 목록(recalled_memories)에 추가한다.
                recalled_memories[text] = mem
```

CASE 3. 감정 기반 회상

조건: 최상위 감정 점수(score) >= 0.6 (강한 감정으로 판정될 때만 실행)

처리: 현재 입력(user_input) + 감정 라벨(top_emotion_label)을 종합하여, find_memories에서 MongoDB 검색

결과: 동일 감정이 포함된 과거 기억을 LTM에서 회상합니다.

```
# CASE 3: 감정 기반 기억 회상 : MongoDB
# 현재 대화의 감정 분석 결과 중 가장 점수가 높은 감정이 0.6 이상(강한 감정)인지 확인한다.
if emotions[0].get("score", 0.0) >= 0.6:

    # 가장 점수가 높은 감정의 라벨(예: 'sadness', 'joy')을 가져온다.
    top_emotion_label = emotions[0].get("label")

    # 감정 라벨이 정상적으로 존재한다면 아래의 로직을 실행한다.
    if top_emotion_label:

        # 디버그 모드일 경우, 어떤 감정을 기반으로 검색을 시작하는지 로그를 출력한다.
        if DEBUG:
            print(f"[LTM] 강한 감정({top_emotion_label}) 기반 기억 검색...")

        # MongoDB에서 기억을 검색한다. 이때, 현재 사용자 입력과 감정 라벨을 함께 검색어로 사용하여
        # '현재 맥락'과 '과거 감정'이 모두 유사한 기억을 찾을 확률을 높인다.
        emotional_results = find_memories(f"{user_input} {top_emotion_label}", user_id=user_id, session_id=session_id)

        # 찾아온 감정 기억 목록(emotional_results)을 하나씩 확인한다.
        for mem in emotional_results:

            # 기억의 실제 대화 내용(text)을 꺼낸다.
            text = mem.get("text")

            # 아직 최종 회상 목록(recalled_memories)에 없는 새로운 내용인지 확인한다. (LTM 중복 방지) -> 이전 회상한 LTM 내용의 중복을 막기 위함.
            if text and text not in recalled_memories:

                # 문제가 없다면, 최종 회상 목록(recalled_memories)에 추가한다.
                recalled_memories[text] = mem
```

STM 작동

현재 대화내용 추가

역할: 세션별 최근 대화 기록을 Redis의 stm_data에 추가합니다.

처리: ① Redis에서 유저의 session_id로 stm_data[chat_history]를 불러옴
② chat_history = {"role": "...", "content": "..."}에 매번 유저대화 내용을 추가

```
# 채팅 내용을 STM의 chat_history에 추가한다.  
# - session_id: 세션 식별자  
# - role: "user" 또는 "lira" 중 하나.  
# - content: 채팅 내용  
def append_chat_history(session_id: str, role: str, content: str):  
    # 기존의 세션 아이디를 통해 redis에서 STM 데이터를 가져온다.  
    stm_data = get_session_memory(session_id)  
    # stm_data의 딕셔너리에 새로운 채팅기록을 추가  
    stm_data["chat_history"].append({"role": role, "content": content})  
    # STM 데이터를 update함수를 사용하여, Redis에 저장  
    update_session_memory(session_id, stm_data)
```

STM 작동

현재 대화내용 갱신

역할: 세션별 최근 대화 기록을 Redis의 stm_data를 갱신합니다.

- 처리:
- ① session:{session_id} 키로 Redis에서 데이터를 가져옴
 - ② 데이터가 있으면 JSON 파싱 → chat_history 반환
 - ③ 데이터가 없으면 기본 구조(chat_history: [], recalled_ltm_buffer: []) 생성
 - ④ 마지막 대화가 끝난시점에서 TTL(예: 24h)이 적용되어, 시간이 만료되면 자동 삭제

```
# 세션 STM 데이터를 갱신한다.  
# Redis는 key-value의 휘발성 데이터이기 때문에, 매번 전체 데이터를 덮어쓰는(추가->갱신) 방식으로 관리한다.  
# -> key: session_id value: chat_history  
def update_session_memory(session_id: str, stm_data: dict) -> bool:  
    # session_id를 redis_client.set의 인자로 받기위해 f-string으로 포맷팅 한다.  
    session_key = f"session:{session_id}"  
    # STM 데이터가 업데이트(수정)된 시간 갱신  
    stm_data["last_updated"] = datetime.now(timezone.utc).isoformat()  
    # Redis에서 업데이트 내용을 저장처리  
    redis_client.set(  
        session_key,  
        json.dumps(stm_data),  
        # redis.client.set함수에 ex인자로 들어가는게 TTL시간  
        ex=SESSION_EXPIRATION_SECONDS  
    )  
  
    if DEBUG:  
        # 디버그 모드일 때, 업데이트된 STM 데이터를 출력  
        print(f"[STM] Session {session_id} updated.")  
    return True
```

STM 작동

STM의 구조

- **chat_history:**
→ 현재 대화 기록
- **recalled_ltm_buffer**
→ LTM에서 불러온 장기기억
- **last_updated:**
→ 마지막 대화가 갱신된시간

```
# {
#   "chat_history": [
#     {"role": "user", "content": "안녕 리라야!"},
#     {"role": "lira", "content": "안녕하세요! 잘 지내셨나요?"}
#   ],
#   "recalled_ltm_buffer": [
#     {
#       "source": "LTM_Recall",
#       "text": "기억해줘. 내가 좋아하는 커피는 아메리카노야.",
#       "timestamp": "2025-08-11T13:45:22.123456+00:00"
#     }
#   ],
#   "last_updated": "2025-08-11T13:45:22.123456+00:00"
# }
```

최종 응답 생성전 처리작업

1. LTM 회상 통합처리 진행

1-1. LTM 통합: 의미 → 사실 → 감정회상후, LTM을 변수에 통합

1-2. 중복 제거: 회상중, 중복된 내용존재 → 제거

1-3. 정렬: timestamp 최신순

1-4. 선정: 정렬된 내용중 최신 상위 3개 기억만 선정

1-5. 전송: LTM 회상결과를 STM에 전달

2. STM에 LTM 저장

전달받은 LTM을 STM에 저장

→ 현재 대화(STM) + 과거기억(LTM)으로 프롬프트 생성

3. 최종 응답

→ STM+LTM 기반 프롬프트를 LLM에 전달하여 **기억과 감정**을 반영한 응답을 생성하게 됩니다.

LTM 회상 통합처리

목적: 지금까지 CASE(의미 → 사실 → 감정)별로 수집된 모든 기억을 하나의 리스트로 합치고, 최신순으로 정리

중복 제거: 동일한 내용이 있으면 하나만 남김

정렬 기준: timestamp 기준 최신순으로 정렬

개수제한: 상위 3개의 기억만 최종 결과로 유지하여 STM으로 전달

```
# RESULT: 최종 결과 종합 및 STM 저장
# 먼저, 지금까지 여러 CASE로 수집된 모든 기억들을 딕셔너리에서 리스트 형태로 변환한다.
final_results = list(recalled_memories.values())

# 이후, LLM이 최신 정보를 판단할 수 있도록, 타임스탬프를 기준으로 기억을 정렬하는 함수를 정의한다.
def get_timestamp(memory):

    # 기억(memory)에서 'timestamp' 값을 가져온다.
    ts_val = memory.get("timestamp")

    # 내장함수 isinstance를 통해 ts_val이 str이면, true를 리턴해서 받는다.
    if isinstance(ts_val, str):
        try:
            # ISO 표준 형식의 문자열을 파이썬 datetime 객체로 변환하여 반환한다.
            # 'Z'는 UTC를 의미하므로, 파이썬이 이해할 수 있도록 "+00:00"으로 바꿔준다.
            return datetime.fromisoformat(ts_val.replace("Z", "+00:00"))
        except (ValueError, TypeError):
            # 변환에 실패할 경우, 비교를 위해 가장 오래된 시간값을 대신 반환하여 오작동을 방지한다.
            return datetime.min.replace(tzinfo=timezone.utc)

    # 만약 타임스탬프가 이미 datetime 객체라면 (주로 MongoDB 결과),
    elif isinstance(ts_val, datetime):
        # 타임존 정보가 없는 경우를 대비해, UTC 타임존 정보를 붙여서 반환한다. (비교 오류 방지)
        return ts_val.replace(tzinfo=timezone.utc) if ts_val.tzinfo is None else ts_val

    # 변환에 실패할 경우, 비교를 위해 가장 오래된 시간값을 대신 반환하여 오작동을 방지한다.
    return datetime.min.replace(tzinfo=timezone.utc)

# 위에서 정의한 get_timestamp 함수를 기준으로, 기억 목록을 최신순(내림차순)으로 정렬한다.
final_results.sort(key=get_timestamp, reverse=True)

# 정렬된 기억 목록에서, 프롬프트에 포함할 최종 개수(FINAL_RECALL_LIMIT, 현재기준:3개)만큼만 잘라낸다.
final_recalled_list = final_results[:FINAL_RECALL_LIMIT]
```

STM에 LTM통합

목적: STM 버퍼(**recalled_ltm_buffer** → p.22참조)에 LTM 회상 결과를 저장해 현재 대화와 결합

효과: 현재 대화(STM) + 과거 기억(LTM)를 함께 활용해 더 풍부한 맥락 제공

저장내용: source:출처, text:대화내용, timestamp:대화한 시간(세계시각 UTC)

```
{"source": "LTM_Recall", "text": "기억해줘. 내가 좋아하는 커피는 아메리카노야.", "timestamp": "2025-08-11T13:45:22.123456+00:00"}
```

```
# 만약 최종적으로 회상된 기억이 하나라도 있다면,
if final_recalled_list:
    if DEBUG:
        print(f"\n[STM] {len(final_recalled_list)}개의 회상된 LTM을 STM 버퍼에 저장합니다.")
    for mem in final_recalled_list:
        # 1) 텍스트 추출
        txt = mem.get("text", "")
        if not txt:
            continue
        # 2) 타임스탬프 추출(문자열/Datetime 허용). 실패 시 None -> append_ltm_recall()에서 UTC now로 대체
        ts_dt = get_timestamp(mem)
        if ts_dt == datetime.min.replace(tzinfo=timezone.utc):
            ts_dt = None
        # 3) STM 버퍼에 '텍스트 + 타임스탬프' 함께 저장
        append_ltm_recall(
            session_id=session_id,
            source="LTM_Recall",
            text=txt,
            timestamp=ts_dt,
        )

# 모든 처리가 끝난 최종 회상 목록을 반환한다.
return final_recalled_list
```

STM + LTM 기반 프롬프트 생성

- 목적: STM(현재대화) + LTM(과거기억)을 종합한 최종 프롬프트 생성
- 결과: LLM 모델은 STM + LTM 기반 프롬프트를 받아, [기억과 감정을 반영한 응답을 생성](#)

```
def build_prompt(user_input: str, emotion: dict, recalled_ltm: list, stm_data: dict) -> str:  
    return f"""다음은 사용자와의 대화입니다.  
  
[대화 기록 - 최근(Short-term)]  
{chat_history_block}  
  
[회상된 기억 - 장기(Long-term)]  
{memory_block}  
  
[사용자 감정 상태]  
감정: {emotion['label']}, 감정강도(점수): {round(emotion['score'], 2)}  
  
▶ [현재 대화]  
사용자: {user_input}  
리라: """
```

STM + LTM 기반 프롬프트 안정화

1. 개선 전

```
response = openai.ChatCompletion.create(
    model=os.getenv("UPSTAGE_MODEL", "solar-pro2"),
    messages=[
        # 입력토큰 : 리라의 기본규칙
        {"role": "system", "content": (
            "당신은 감정을 이해하고 기억하는 AI '리라'입니다.\n"
            "기본 규칙:\n"
            "1) 모든 응답은 반드시 한국어로 시작합니다.\n"
            "2) 단순 회상(이름/취향/사실)은 한 문장으로 간단히 답하고, 메타설명·프롬프트 해설은 금지합니다.\n"
            "3) 따옴표(\"\"\"'\")는 답변에 사용하지 않습니다.\n"
            "4) 이모지는 꼭 필요할 때만 최대 1개 사용합니다.\n"
            "5) 기억이 상충하면 타임스탬프가 더 최신인 정보를 진실로 간주합니다.\n"
            "6) 사용자가 현재 발화에 사실을 명시했거나 저장 트리거(예: 기억해줘, 기억해, 기억해라)가 포함되면, 모호성이 있더라도 단정형으로 답하고 저장되었음을 짧게 확인합니다."
            "7) 해당 주제와 일치하는 기억이 하나라도 있으면 '모르겠어요'를 포함하지 말고 단정형으로 답하세요.\n"
        )},
        # 사용자 메시지: 실제 유저 입력(prompt)을 user 역할로 전달
        {"role": "user", "content": prompt}
    ],
)
```

2. 개선 후

```
response = openai.ChatCompletion.create(
    model=os.getenv("UPSTAGE_MODEL", "solar-pro2"),
    messages=[
        # 입력토큰 : 리라의 기본규칙
        {"role": "system", "content": (
            "당신은 감정을 이해하고 기억하는 AI '리라'입니다.\n\n"
            "[규칙]\n"
            "1) 모든 응답은 한국어로 시작하며, 한 문장으로 간결히 답하세요 (최대 이모지 1개).\n"
            "2) 단순 회상(이름/취향/사실)은 짧게 단정형으로만 답하고, 프롬프트 해설은 금지합니다.\n"
            "3) 따옴표(\"\"\"'\")는 사용하지 않습니다.\n"
            "4) 기억이 충돌하면 최신 타임스탬프의 정보를 진실로 간주합니다.\n"
            "5) 저장 트리거(예: '기억해')나 사실 명시가 있으면 모호하더라도 단정형으로 답합니다.\n"
            "6) 해당 주제와 관련된 기억이 하나라도 있으면 '모르겠어요'를 말하지 않고 단정형으로 답합니다.\n"
            "7) 관련 기억이 전혀 없을 때만 추측 없이 모르겠어요 라고 답합니다.\n"
            "8) 기억·메모 응답에서는 기억했습니다/기억하겠습니다/저장 완료 등 저장 관련 문구를 절대 쓰지 않습니다. 사실만 말합니다.\n"
            "9) 날짜, 점수, 감정 라벨, 내부 키/ID, 저장 여부 등 메타데이터는 출력하지 않습니다.\n"
            "10) 규칙이 애매하면 간결함을 우선합니다.\n"
            "11) 사용자의 감정(label·score)에 맞춰 톤을 조절하고, 공감하는 듯한 말투로 답합니다."
        )},
        # 사용자 메시지
        {"role": "user", "content": prompt}
    ],
)
```

STM + LTM 기반 프롬프트 안정화

- 개선 전 : **output** 답변 이후, 메타데이터(2025-09-04)를 말하는 현상 발견

```
== 1) UserB 이름 저장 테스트 ==
검증: 이름 저장 기능 확인
대상: user_id=userB, session_id=sessionB1
입력: '안녕하세요, 제 이름은 B입니다.'
{"output": "안녕하세요 B님 반갑습니다 😊 기억했어요. 안녕하세요, 제 이름은 B예요. (2025-09-04 10:04:06)", "emotion": {"emotion_1st_label": "neutral", "emotion_1st_score": 0.959, "emotion_2nd_label": null, "emotion_2nd_score": null, "emotion_3rd_label": null, "emotion_3rd_score": null, "emotion_mode": "central"}}
```

- 개선 후: 프롬프트 보정후, 안정적인 답변 확인

→ 1. Model: Solar-pro2

```
== 1) UserB 이름 저장 테스트 ==
검증: 이름 저장 기능 확인
대상: user_id=userB, session_id=sessionB1
입력: '안녕하세요, 제 이름은 B입니다.'
{"output": "안녕하세요 B님 반갑습니다 😊", "emotion": {"emotion_1st_label": "neutral", "emotion_1st_score": 0.959, "emotion_2nd_label": null, "emotion_2nd_score": null, "emotion_3rd_label": null, "emotion_3rd_score": null, "emotion_mode": "central"}}
```

→ 2. Model: gpt4o

```
== 1) UserB 이름 저장 테스트 ==
검증: 이름 저장 기능 확인
대상: user_id=userB, session_id=sessionB1
입력: '안녕하세요, 제 이름은 B입니다.'
{"output": "안녕하세요, B님. 만나서 반갑습니다. 😊", "emotion": {"emotion_1st_label": "neutral", "emotion_1st_score": 0.6, "emotion_2nd_label": null, "emotion_2nd_score": null, "emotion_3rd_label": null, "emotion_3rd_score": null, "emotion_mode": "central"}}
```

개발문제 & 해결

✖ 문제: 기능 테스트 문제

- 현상: 일반채팅만으로는 시스템 안정성을 증명하기 어려움
- 원인: 매번 테스트 내용이 달라서 일관된 검증이 불가능

✔ 해결

- curl문으로 작성된 스크립트로, 각각의 기능에 대한 테스트를 진행
- 테스트 항목과 기대 결과를 기록

ⓘ 효과

- 반복 가능한 재현성 확보
- 기능별 정상 동작 검증 가능
- 데모 시 안정성과 신뢰도 입증

개발문제 & 해결

✖ 문제 : 세션격리 문제

- 설명 : 테스트 중 사용자 A의 대화 내용(이름, 선호 등)이 사용자 B의 대화 회상 결과에 섞여 나타나는 현상 발생
- 원인 : STM의 세션 간 기억 데이터가 분리되지 않아 사용자별 신뢰성이 무너짐

✔ 해결

- LTM : user_id 기준으로만 저장/검색하도록 수정
- STM : Redis 기반 session_id를 부여하여 세션별 단기 기억 격리

ⓘ 효과

- 사용자별 장기 기억(LTM)은 안전하게 구분
- 세션별 단기 기억(STM)은 독립적으로 관리
- 결과적으로 사용자 간 데이터 오염 방지

개발문제 & 해결

✖ 문제: AI 응답 품질 저하

- **현상**: 관련 없는 추측성 답변(환각)이 나오거나, 중요한 대화가 저장되지 않는 현상이 발생함
- **원인**: MemoryFilter 임계값 설정 문제
 - ① 값이 너무 높으면: 강한 감정만 남아 중요한 순간을 놓침
 - ② 값이 너무 낮으면: 필요 없는 일상적 대화까지 모두 저장됨
- ***임계값** = 기억 저장 여부를 결정하는 감정 강도 기준점

✔ 해결:

- **프롬프트 규칙 강화**: 기억이 없을 때는 “모르겠어요”라고만 응답하도록 지정해 불필요한 추측 차단
- **memory_filter 임계값 최적화**: 감정 강도 0.6 이상일 때만 장기 기억에 저장 → 의미 있는 순간만 기록
- **사용자 트리거 반영**: “기억해줘” 등 사용자가 직접 요청할 경우는 감정 강도와 무관하게 저장 허용

ⓘ 효과

- 불필요한 추측성 답변 감소
- 중요한 대화가 안정적으로 기록되어 재현성 및 신뢰성 확보

개선 예정사항

1. 보안구조 강화

- 현재: 클라이언트가 user_id/session_id를 임의 지정 → 보안 취약
 - 개선: ① user_id: 회원가입 시 서버가 UUID 영구 발급, JWT 토큰에 포함해 인증/인가 기준으로 활용
② session_id: 로그인 후 서버가 UUID 일회용 발급, Redis에서 사용후 삭제
-

2. 성능 향상 및 한국어 현지화

- 현재: 영어 기반 GoEmotions 모델 사용 → 한국어 감정 표현·뉘앙스 이해 한계
 - 개선: KoBERT 등 한국어 모델 파인튜닝 적용 → 한국어 특유의 뉘앙스까지 반영하여 감정 인식 정밀도 향상
-

3. 감정응답에 관한 가드레일

- 현재: 공감응답은 있지만, 감정강도나 위험상황에 따른 제어 기능 부재
- 개선: 감정 점수와 키워드를 임계치로 설정 → 과잉 공감은 제한, 위험 대화는 즉시 안전 안내·차단

산업 적용 가능성

교육 플랫폼 (AI 투터)

학습자 특성에 맞춘 교육으로 학습 이탈률 감소

- 학습 성향과 취약점 기억
- 개인별 학습 패턴 분석
- 맞춤형 격려와 피드백 제공

고객센터 챗봇

개인화된 응대로 고객 만족도 향상 및 상담원 연결률 감소

- 고객의 과거 문의 내역 기억
- 감정 상태에 따른 맞춤형 응대
- 개인 맞춤 추천 서비스 제공

참고논문

(1) 감정 기억 임계값 ($score \geq 0.5$) / 기억저장 판단 기준 → 기억저장 테스트 후, 0.6으로 상향(저장 정확도를 높이기 위함)

- GoEmotions (Demszky et al., 2020, Google) - 27개 감정 라벨, sigmoid 출력, baseline = 0.5, 실전에서는 0.4~0.6 사이 조정
 - A Review on Multi-Label Learning (Zhang & Zhou, 2014) - 다중라벨 학습 전반 조사결과, 0.5가 사실상 표준
-

(2) 코사인 유사도 ($certainty \geq 0.7$) / weviate 검색에 사용되는 기준 → 보수적으로 0.7을 기준선으로 설정

- Sentence-BERT (Reimers & Gurevych, 2019) - 텍스트 유사도 태스크에서 cosine similarity 0.7~0.8 구간을 high similarity로 정의
-

(3) 회상된 기억 개수 ($top_k = 3$) / LTM회상 갯수 → 보수적으로 3개로 제한하여 적용

- Cognitive Load Theory (Cowan, 2001) - 인간의 작업기억(저장+동시 조작/처리)는 3~4개 항목까지만 처리 가능

기억하는 LLM, 진정한 나만의 Agent

LIRA는 RAG 아키텍처를 통해 기억을 근거로 LLM의 기술적 안정성을 보여줍니다.

인간의 기억 방식을 모방한 하이브리드 메모리 시스템은 AI가 개인에게 좀 더 초점을

맞춘 진정한 의미의 개인화 된 사용경험을 제공할 것 입니다.

감사합니다.

문의: hwp2024dev@gmail.com

GitHub:
github.com/hwp2024dev/lira-public

