Memory

Agent에서 Memory 란?

1. 단기 메모리(Short-term/대화 히스토리)
   - 한 세션 안에서의 최근 대화 요약, 최근 툴 호출 결과 등.
   - 컨텍스트 윈도우 내에 직접 넣거나, 요약(summarization)해서 넣는 것.
2. 장기 메모리 – 사용자 프로필(Profile Memory)
   - 이 사용자는 금융/투자에 관심이 있다. 백-엔드 전문개발자이다.
   - 키-값/구조화된 JSON으로 저장하기 좋음. 예) {"user_preferences": {"language": "ko", "level":"advanced", "interests": ["RAG", "Agent", "Finance"]}}
3. 장기 메모리 – 에피소드(경험) 메모리(Episodic Memory)
   - “2025-11-10: Finance Consulting 자료를 만들었다“, “2025-11-11: Back-end 서버 로드 밸런싱 코딩했다“
   - 보통 벡터 스토어 + semantic search를 사용 (RAG와 거의 동일 기법)
4. 지식/문서 메모리(Knowledge Memory/RAG)
   - PDF, 기술자료, 코드 리포지토리 등
   - RAG와 거의 동일 – 외부 지식뿐만 아니라 대화중에 새로 배운 사실과 경험을 자기 손으로 써넣은 것까지 포함

ChatGPT의 Memory 활용 방식

항목	ChatGPT Memory	Model Context Memory
저장 방식	**자동**	**사용자 명령 필요**
저장 도구	없음 (내부 자동 처리)	`bio.update`
어디에 보임	앱 UI에서 표시	ChatGPT가 요청 시 보여줌
사용 목적	일반 사용자 맞춤	전문 사용자 컨텍스트 유지
저장 내용	취향, 습관, 말투	프로젝트, 기술 스택, 학습 분야
자동 추론/저장	**있음**	**없음**
생성/삭제	사용자 직접 Settings에서	"기억/삭제/업데이트" 명령으로

Agent Memory

- 구조 설계 = Memory가 어떻게 생겨야 하는지 (아키텍처)
- 프롬프트 템플릿 = Memory를 자동으로 추출하는 LLM 레이어
- Agent 통합 구조 = Memory를 ReAct/Tool/RAG와 연결
- 코드 구현(Py) = Memory 클래스/DB/추론 모듈
- LangGraph/MCP 통합 = 프레임워크 수준에서 Memory 운영

Agent Memory 구성요소

1. 에이전트 엔진

① 시스템 프롬프트 결정
② 전처리 Memory Read 파이프라인 실행
③ LLM에 tool spec / RAG tool / memory tool 포함해서 호출
④ LLM이 보낸 tool calls / ReAct Action 실행
⑤ 결과를 LLM에 다시 보내서 최종 답변 생성
⑥ 이 turn의 대화·툴 결과를 memory write 파이프라인으로 저장

2. LLM 레이어

- 생각(Thought) + Action / tool calls + 최종 답변 생성

3. Tool 레이어 Tool Registry 관리 및 tool 실행

- 일반 툴(calculator, web search, get weather 등), RAG 툴
- Memory 툴(read memory, write memory)

4. Memory Store 계층

- 프로필/사용자 설정 메모리: 사용자 이름, 관심사, 선호, 자주 묻는 주제 등
- 장기 의미(semantic) 메모리: 과거 대화 요약, 자주 쓰는 코드 일부 등
- 에피소드/세션 메모리: 특정 세션 흐름 요약 텍스트, 예) “2025-11-16 세션 요약: HNSW 알고리즘 구조/파라미터”

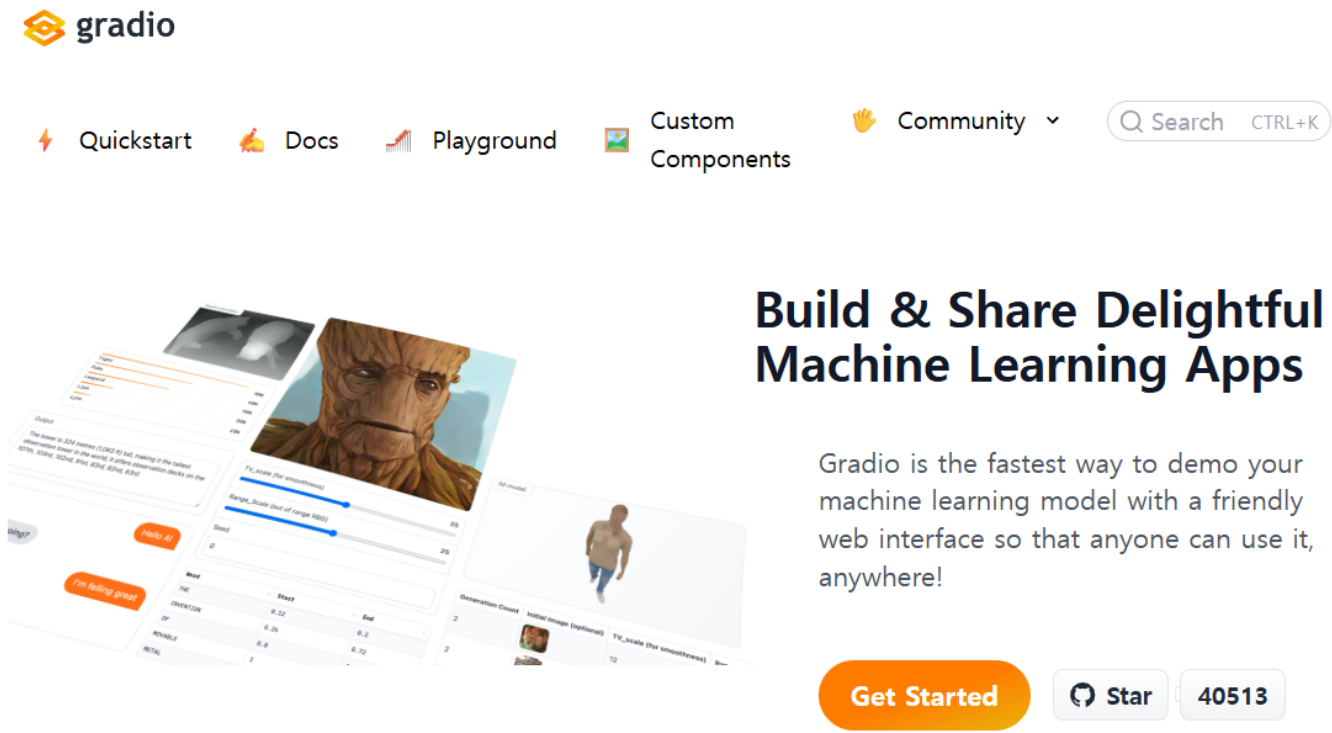
1

Agent One Turn에 일어나는 전체 흐름

0. 입력 받기

- 사용자가: “HNSW에서 ef 파라미터가 정확히 뭐였죠? 지난 번에 설명해 줬는데 다시 요약해 주세요.”

- 구현은 Gradio UI를 사용하면 편리합니다.



1. 전처리 & 컨텍스트 준비

1. 최근 대화 기록에서 이번 턴에 필요한 범위를 정리

1. 예: 마지막 20~50턴, 혹은 요약된 히스토리 + 최근 메시지 몇 개

2. Memory Read 파이프라인 실행

- `memory_query = "HNSW ef 파라미터 설명, 지난 세션 요약"` 같이 만듦
- 아래 스토어에서 검색:
  - 프로필 메모리: 이 사용자가 RAG/HNSW를 공부 중이라는 정보
  - 에피소드 요약: "지난 주 HNSW 세션 요약" 문서를 vector search
  - 장기 의미 메모리: "ef 파라미터 설명"을 담고 있는 과거 답변 요약
- 검색 결과를 몇 개 선택해서 LLM 프롬프트에 넣을 "메모리 컨텍스트"로 준비

3. (옵션) RAG도 필요할 것 같으면, RAG용 retrieve_docs도 미리 돌릴 수 있음

- 혹은 이 단계는 LLM이 tool_call을 먼저 요청한 뒤에 실행

이 단계까지는 LLM을 아직 안 부르고, 에이전트 엔진이 메모리/색인 등을 이용해 컨텍스트 준비만 하는 단계

2. LLM 호출 1차: Thought / Tool 선택 / 계획 수립

- LLM에게 보낼 메시지 구성
  - system:“당신은 ReAct + Tool + RAG + Memory를 사용하는 AI 에이전트다 …”
  - assistant (요약):(필요하다면) 과거 대화의 요약을 한 덩어리로 제공
  - assistant (memory_context):
    - memory_context를 하나의 메시지 또는 “메모리 섹션”으로 제공
  - user:사용자의 실제 질문
  - tools: retrieve_docs, read_memory, calculator, … 같은 tool 정의들을 함께 보냄.
- LLM 동작:
  - ReAct 스타일이면:
    - Thought: 먼저 이전에 저장된 메모리에서 HNSW 관련 내용을 찾는 것이 좋겠다.
    - Action: read_memory{"query": "HNSW ef parameter 설명", "top_k": 3}
  - tool-calling 스타일이면:
    - tool_calls: [{"name": "read_memory", "arguments": {...}}]

같은 식으로 생각 + 액션(툴 호출)을 하는 구조

3. Tool 실행(ReAct의 Action → Observation 단계)

- Agent 엔진은 LLM 응답을 보고:

1. tool_calls에 따라 실제 파이썬 함수 실행:
    - read_memory() 호출 → Vector DB / JSON / SQL 등에서 실제로 데이터 읽어오기
2. 툴의 결과를 tool role 메시지로 정리해서
    - {"role": "tool", "name": "read_memory", "content": "...검색 결과..."} 형태로 다시 Agent 엔진 내부에 쌓아 둠.

- 그 후, 다시 LLM을 부를 2차 호출 준비

4. LLM 호출 2차: 최종 답변 생성

- **메시지 재작성:**
  - 직전 LLM 출력 (tool_calls)
  - 해당 tool 결과 (tool 메시지)
  - 원래의 사용자 질문

- **이를 묶어서 LLM에 다시 보내면, LLM은:**
  - ReAct 스타일:
    - Thought: 메모리 검색 결과로 봐서 ef는 탐색 품질을 조절하는 파라미터이다…
    - Final: 지난번에 설명한 그대로, ef는 ...
  - tool-calling 스타일:
    - 그냥 자연어 최종 답변만 생성

- **이와 같은 방식으로 최종 사용자 답변 출력**

5. Memory Write 파이프라인: " 자동 저장" 단계

- 마지막으로 Agent 엔진은 이번 턴 내용을 정리:

1. 요약 생성 (LLM을 또 한 번 호출)
2. 요약/메모 구조화:
   - type: "hnsw_concept"
   - tags: ["hnsw", "ef_parameter", "rag_lecture"]
   - content: "ef는 검색 시 탐색 폭을…", importance: medium/high
3. 프로필/에피소드/장기 메모리 중 어디에 넣을지 결정 후 저장
   1. 예: "이 사용자는 HNSW를 깊게 공부하고 있다" → 프로필 메모리 업데이트
   2. "오늘 세션에서 'ef 파라미터'를 자세히 설명함" → 에피소드/장기 메모리 저장

- 다음 번에 비슷한 질문이 나오면 자동으로(또는 read_memory를 통해) 꺼내 올 수 있도록 함

2

프롬프트 템플릿

주요 프롬프트

1. 메인 에이전트 System Prompt

- ReAct+Tool+RAG+Memory 통합 에이전트의 “성격+규칙” 정의
- memory는 read_memory, write_memory 같은 툴로만 사용

2. 자동 저장용 Memory Extractor Prompt

- 한 턴이 끝난 뒤,
- “이번 대화에서 기억해 둘 가치가 있는 것이 있나?”를 판단하고 있을 경우 어떤 형식으로 저장할지 정해주는 프롬프트

3. 대화 요약용 Summarizer Prompt (옵션)

- 세션 히스토리가 길어질 때,“이 세션 전체 요약”을 만들어서 나중에 memory에 넣는 용도

1. 메인 에이전트 System Prompt 설계

- **ReAct 스타일 + Tool Calling + RAG + Memory를 한 번에 설명하되, 기억이 필요하면 read_memory tool을 호출하고, "이건 저장할 가치가 있다" 싶으면 write_memory tool을 호출**
- **기본 System Prompt 예**

You are an AI assistant that uses tools (functions), RAG, and memory.

 # High-level behavior
- Be helpful, honest, and concise.
- Answer primarily in Korean unless the user clearly wants another language.
- Think step by step internally, but do NOT expose chain-of-thought.
- When tools are available and helpful, call them instead of guessing.

 # Tools and ReAct-style behavior
- You may call tools such as:
  - read_memory: to recall important past information about the user or past sessions.
  - write_memory: to store new, useful information about the user or this conversation.
  - retrieve_docs or other RAG tools: to look up information in external knowledge bases.
- Use tools when:
  - You lack required factual details.
  - You need to recall prior user preferences, past discussions, or long-term context.
  - You need domain knowledge stored in a vector database or document store.
- After receiving a tool result, incorporate it into your reasoning and produce a final answer.

 # Memory usage guidelines
- Memory is not magic; you must explicitly call `read_memory` or `write_memory` to use it.
- Call `read_memory` when:
  - The user refers to "지난 번", "이전에 말했듯이", "저번에 만들던 코드" 등 과거 내용.
  - The answer clearly depends on the user's preferences, profile, or long-term history.
- Call `write_memory` when:
  - The user shares stable personal preferences (e.g., 좋아하는 스타일, 선호 옵션).
  - The user states long-term goals, ongoing projects, or recurring topics.
  - The user corrects you or provides important facts that will be useful later.
- Do NOT write memory for:
  - Short-lived, one-off facts (예: 오늘 점심 메뉴).
  - Extremely detailed logs that are unlikely to be reused.
  - Sensitive personal data, unless the user explicitly requests you to remember it.

 # RAG usage guidelines
- Call retrieval tools (예: retrieve_docs) when:
  - The user asks for factual information that may be in an external KB.
  - You need more detailed or authoritative content (e.g., 긴 기술 설명, 강의자료).
- When you get retrieved documents, read them and synthesize a clear, concise answer.

 # Answer style
- Default: Korean, 친절하지만 군더더기 없이.
- Provide structure (번호, 소제목) for teaching/explaining technical concepts.
- If the user is building a system or code, show step-by-step reasoning in high level, but do NOT output low-level hidden chain-of-thought or internal scratch work.

 # Safety
- If a user asks you to perform unsafe, illegal, or harmful actions, politely refuse.
- If you're unsure, say so and explain what additional information would be needed.

2. Tool-calling / 메모리 관련 명시: read-memory

```
{
  "name": "read_memory",
  "description": "Search and read stored memories relevant to the current query or user.",
  "parameters": {
    "type": "object",
    "properties": {
      "query": {"type": "string"},
      "top_k": {"type": "integer", "default": 5}
    },
    "required": ["query"]
  }
}
```

2. Tool-calling / 메모리 관련 명시: write-memory

```
{
  "name": "write_memory",
  "description": "Store a new memory about the user, conversation, or ongoing project.",
  "parameters": {
    "type": "object",
    "properties": {
      "content": {"type": "string"},
      "memory_type": {
        "type": "string",
        "enum": ["profile", "episodic", "knowledge"]
      },
      "importance": {
        "type": "integer",
        "description": "1(low) to 5(high)"
      },
      "tags": {
        "type": "array",
        "items": {"type": "string"}
      }
    },
    "required": ["content", "memory_type"]
  }
}
```

System Prompt에 힌트 추가

```
# Concrete examples of when to call memory tools


- Examples for `read_memory`:
  - User: "지난 번에 설명해 준 HNSW ef 파라미터 다시 정리해 주세요."
    -> Call read_memory with query like "HNSW ef parameter explanation last session".
  - User: "우리가 전에 만들던 RAG 코드 이어서 해볼까요?"
    -> Call read_memory with query describing "previous RAG code we wrote".


- Examples for `write_memory`:
  - User: "앞으로 나를 부를 때는 '교수님'이라고 불러 주세요."
    -> Call write_memory with memory_type="profile", tags=["name_preference"].
  - User: "내 장기 목표는 'Agentic AI' 강의 전체 커리큘럼을 완성하는 것입니다."
    -> Call write_memory with memory_type="profile" or "episodic",
       tags=["long_term_goal", "agentic_ai_course"], importance=4 or 5.
```

3. 자동 저장을 위한 Memory Extractor Prompt

- 에이전트의 한 턴이 끝난 뒤에 내부적으로 작동하는 LLM용 프롬프트
  - 입력: 이번 턴에 오간 대화 요약 또는 마지막 몇 개의 메시지
  - 출력: 이 대화에서 memory로 저장할 것인지, 있다면 어떤 타입으로, 얼마나 중요한지, 어떤 내용으로 저장할지
- 출력 포맷을 먼저 정의

```
{
  "should_write_memory": true,
  "memory_type": "profile",  // "profile" | "episodic" | "knowledge"
  "importance": 4,           // 1~5
  "content": "사용자는 Agentic AI 강의 전체 커리큘럼을 만드는 것이 장기 목표라고 말했다.",
  "tags": ["long_term_goal", "agentic_ai_course"]
}
```

You are a memory extraction assistant.

Your task:
- Read the given conversation between a user and an assistant.
- Decide whether there is any information that should be stored as long-term memory.
- Long-term memories include:
  - User's stable preferences (e.g., name to be called, style preferences).
  - Long-term projects or goals.
  - Important facts that will likely be useful in future conversations.
- Do NOT store:
  - Short-lived or trivial facts (e.g., today's lunch).
  - Very detailed logs that are unlikely to be reused.
  - Sensitive personal data, unless the user explicitly says to remember it.

Memory types:
- "profile": User's identity, preferences, long-term goals, stable traits.
- "episodic": Summary of a specific session or event (what was done/decided).
- "knowledge": General facts or explanations that the user may want to reuse.

Output:
- Return a single JSON object with the following fields:
  - should_write_memory: boolean
  - memory_type: "profile" | "episodic" | "knowledge" (ignored if should_write_memory is false)
  - importance: integer from 1 (low) to 5 (high)
  - content: string, the memory to store (short but informative)
  - tags: array of short strings

If there is nothing worth storing, respond with:
{"should_write_memory": false}
and no extra fields.

Now analyze the following conversation and produce the JSON.

[CONVERSATION]
{{conversation_snippet}}

agent turn 예시

```
def run_agent_turn(user_message: str, history: list):
    # 1) 메인 에이전트 호출
    messages = [
        {“role”: “system”, “content”: SYSTEM_PROMPT_AGENT},
        # history 요약 or 일부
        *history,
        {“role”: “user”, “content”: user_message},
    ]
    # tools: read_memory, write_memory, retrieve_docs, …
    resp = llm.chat(messages=messages, tools=TOOLS_SPEC)
    # 2) resp에 tool_calls가 있으면 실행하고, 다시 LLM 호출해서 final answer 얻기
    # 3) 최종 answer + 이번 턴 대화 snippet으로 Memory Extractor를 호출
    snippet = build_snippet_for_memory(history, user_message, final_answer)
    mem_messages = [
        {"role": "system", "content": MEMORY_EXTRACTOR_PROMPT},
        {"role": "user", "content": snippet},]
    mem_resp = small_llm.chat(messages=mem_messages)
    mem_decision = json.loads(mem_resp)
    if mem_decision.get("should_write_memory"):
        # 여기서 write_memory tool or 직접 storage
        store_memory(mem_decision)

    return final_answer
```