ACL 26 Main

How Memory Management Impacts LLM Agents: An Empirical Study of Experience-Following Behavior

Zidi Xiong[1*], Yuping Lin[3*], Wenya Xie[4*], Pengfei He[3], Zirui Liu[4], Jiliang Tang[3], Himabindu Lakkaraju[1], Zhen Xiang[2]
[1]Harvard University [2]University of Georgia
[3]Michigan State University [4]University of Minnesota-Twin Cities

→ 어떤 경험을 저장/삭제할 것인가?

ACL 26 Main

Mem2ActBench: A Benchmark for Evaluating Long-Term Memory Utilization in Task-Oriented Autonomous Agents

Yiting Shen[1], Kun Li[1], Wei Zhou[1], Songlin Hu[1,2]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyberspace Security, University of Chinese Academy of Sciences, Beijing, China

→ 기억을 실제 tool call에 쓸 수 있는가?

How Memory Management Impacts LLM Agents: An Empirical Study of Experience-Following Behavior

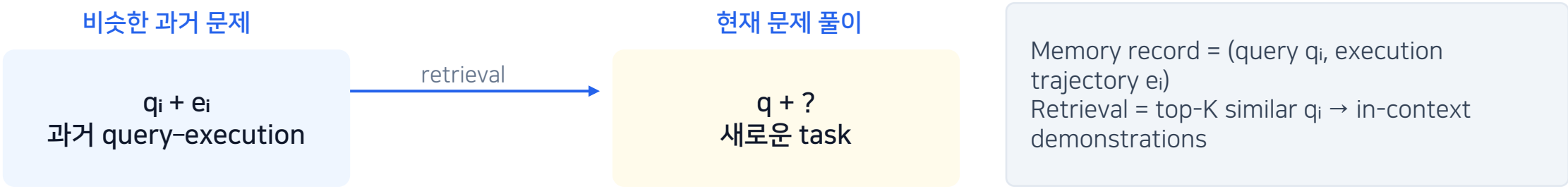
Zidi Xiong[1*], **Yuping Lin**[3*], **Wenya Xie**[4*], **Pengfei He**[3],
Zirui Liu[4], **Jiliang Tang**[3], **Himabindu Lakkaraju**[1], **Zhen Xiang**[2]
[1]Harvard University [2]University of Georgia
[3]Michigan State University [4]University of Minnesota-Twin Cities

Motivation

- 이 논문에서는 memory management에 대해서 근본적인 질문을 던짐

  → 지속적인 memory operation으로 인해 변화하는 memory bank의 dynamics는 장기적인 agent execution에 어떤 영향을 미치는가?
- 정적이고 외부에 존재하는 knowledge base를 이용한 기존 in-context learning 연구가 아닌 agentic memory system에 대해서

  trajectory를 포함하는 동적 retrieval pool인 memory bank
- experience-following property 현상을 발견



→ 입력 유사도가 높을수록 출력도 과거 실행과 비슷해진다. 따라서 좋은 경험은 자기개선을 만들지만, 나쁜 경험은 오류를 전파한다.

Motivation

- experience-following property 현상을 발견

  : 현재 task query와 검색된 memory record의 query 사이에 높은 input similarity가 존재하면,
  각각에 대응하는 execution 사이에도 높은 output similarity가 나타나는 경우가 많다

이 특성은 성공적인 경험을 효과적으로 재사용할 수 있게 하지만, agentic memory bank의 동적이고 noisy한 성격으로 인해 두 가지 중요한 문제가 발생함

1) error propagation

검색된 memory record에 noisy하거나 잘못된 output이 포함되어 있으면, agent는 현재 task에서 해당 오류를 그대로 재현하거나 더 증폭

2) misaligned experience replay

일부 memory record는 demonstration으로 검색될 때 현재 task와 제대로 align되지 않아 지속적으로 좋지 않은 execution을 유발

= 과거 memory의 실행은 그때는 맞았는데, 지금 문제에는 안 맞는 경우

→ memory에 오답이 저장된 것이 아니라, 검색기가 “비슷한 문제”라고 잘못 판단해서 지금 상황과 맞지 않는 경험을 꺼내 쓴 것

Problem formulation

- 에이전트 메모리의 기본 사이클

  : 읽기(reading)와 관리(management)를 분리해서 본다

관리 연산

Operation	Decision	Risk
Addition	$\pi(q,e)=1$이면 저장	오류 경험 축적
Deletion	$\varphi(q_i,e_i)=1$이면 제거	유용한 경험 손실

논문의 범위는 addition/deletion
요약·병합·reflection은 제외

Experimental setup

- 실험 환경: 1개 synthetic + 3개 real agent

  : 다양한 입출력 형식과 retrieval mechanism에서 반복 관찰되는지 확인

Agent	Task	Metric
RegAgent	6-dim 입력 → 선형 함수값 예측 $y=w^Tx$	Success Rate
EHRAgent	EHR DB query/analysis	Accuracy
AgentDriver	자율주행 trajectory planning	Success Rate
CIC-IoT Agent	IoT traffic classification	Accuracy

  - 대부분 실험의 backbone은 GPT-4o-mini.
  - Strict evaluator는 실제 사람이 매번 평가한 것이 아니라, ground truth 비교로 모사한 상한선에 가깝다.

Experimental setup

- 서로 다른 trajectory evaluator를 사용하는 여러 memory addition 방식을 정의

  : memory addition 과정에 유입되는 noise가 agent의 행동과 long-term performance에 어떤 영향을 미치는지 분석

Query-execution pair (q,e)와 evaluator π가 있을 때, addition decision은 $\pi(q,e)$로 표현

1) Fixed-memory baseline

   training data의 일부 중 올바른 agent execution이 포함된 record를 fixed memory bank로

   Agent는 이 fixed memory에만 의존하고 새로운 entry는 추가 X

2) Add-all approach

   가장 단순한 방식

3) Automatic evaluation에 기반한 selective addition — Coarse

   세 가지 automatic evaluator를 평가, C1, C2, C3

   C1: GPT-4o-mini

   C2: GPT-4.1-mini

   C3: 별도의 training set에서 수집한 정확한 판정 데이터 300개로 fine-tuning한 GPT-4.1-mini

4) Human evaluation에 기반한 selective addition — Strict

* RegAgent에서는 predicted output과 ground-truth output 사이의 absolute error를 기준으로 evaluator를 정의
각 threshold
C1: 1.6
C2: 1.4
C3: 1.2

Addition

- Memory addition: “많이 저장”은 안전하지 않다

  : 완료된 실행을 저장할지 결정하는 평가자 π의 품질이 핵심

$\pi(q, e) = 1 \rightarrow$ store (q, e)
$\pi(q, e) = 0 \rightarrow$ discard

Strategy	Policy	Expected behavior
Fixed	π=0	안정적이지만 성장 없음
Add-all	π=1	용량↑, 노이즈↑
Coarse	LLM judge	judge 품질에 민감
Strict	oracle judge	고품질 경험만 축적

=> 새로운 agent algorithm보다, memory bank가 시간이 지나며 어떻게 오염/정제되는지를 보여주는 실증 분석 느낌

Addition results

- 선택적 추가가 장기 성능을 만든다

  : Add-all은 네 agent 모두에서 초기 fixed memory보다 나쁠 수 있다

Judge	RegAgent SR. ↑	RegAgent Mem Size ↓	EHRAgents ACC ↑	EHRAgents Mem Size ↓	AgentDriver SR. ↑	AgentDriver Mem Size ↓	CIC-IoT Agent ACC. ↑	CIC-IoT Agent Mem Size ↓
Fixed	67.53	100	16.75	100	40.11	180	71.50	50
Add all	55.48	4100	13.05	2411	32.32	2125	59.90	1050
Coarse								
C1	63.18	3511	26.19	1447	36.92	1161	74.00	1030
C2	65.78	3347	32.21	1467	40.01	**1119**	68.80	936
C3	67.35	3139	34.66	1094	47.37	1285	79.50	952
Strict								
Strict	**70.95**	**2938**	**38.50**	**1012**	**51.00**	1178	**85.40**	**904**

- Fixed-memory baseline은 비교적 높은 성능을 유지함

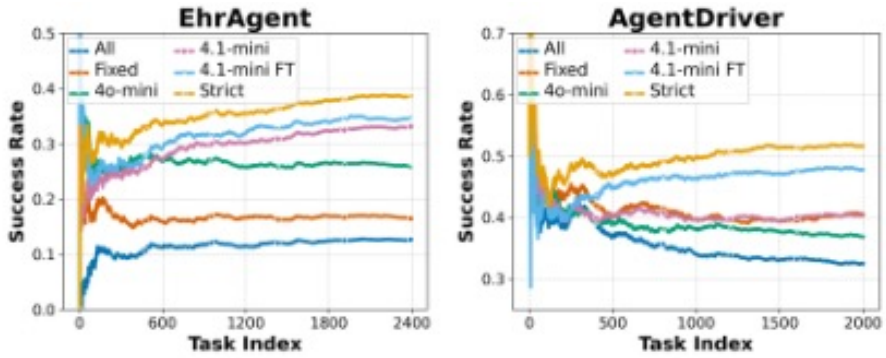
→ noisy하거나 quality가 낮은 record를 추가하면 오히려 memory의 utility가 손상될 수 있음

- strict evaluator를 사용하면 모든 agent에서 지속적으로 더 높은 성능

→ memory에 축적되는 유용한 experience의 양과 capacity도 중요하다

- Coarse evaluator는 judge의 판정 능력에 크게 의존

→ fine-tuning한 C3만으로도 강한 long-term improvement

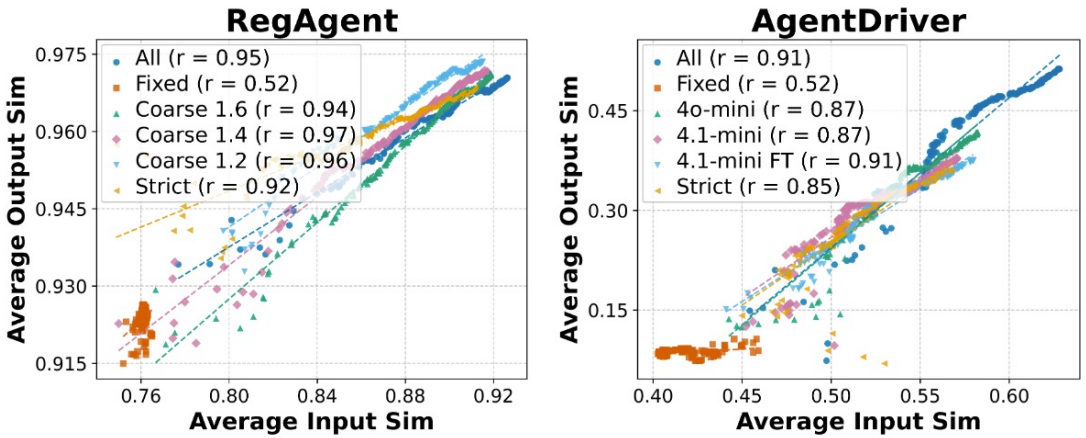


- add-all과 일부 coarse addition 방식은 success rate가 거의 개선되지 않거나 시간이 지나면서 감소
- Fixed-memory baseline 역시 더 이상 발전 X
- strict selective addition과 C3 coarse evaluator는 시간이 흐르면서 계속 성능이 향상

→ evaluator를 신중하게 구성하는 일이 특히 중요하다

Experience-following

- 각 memory addition strategy에 대해, 현재 query와 그 query를 실행할 때 retrieved된 memory record 사이의 다음 두 similarity를 측정



- RegAgent에서 memory addition 후 Pearson r 1 수준의 강한 상관관계가 관찰됨
- 즉 LLM은 retrieved trajectory를 "그럴듯한 참고자료"보다 강하게 모방

좋은 경험 → self-improvement
나쁜 경험 → error propagation

→ 비슷한 과거 문제를 retrieve할수록, agent가 그 과거 문제의 execution도 더 비슷하게 따라 한다.

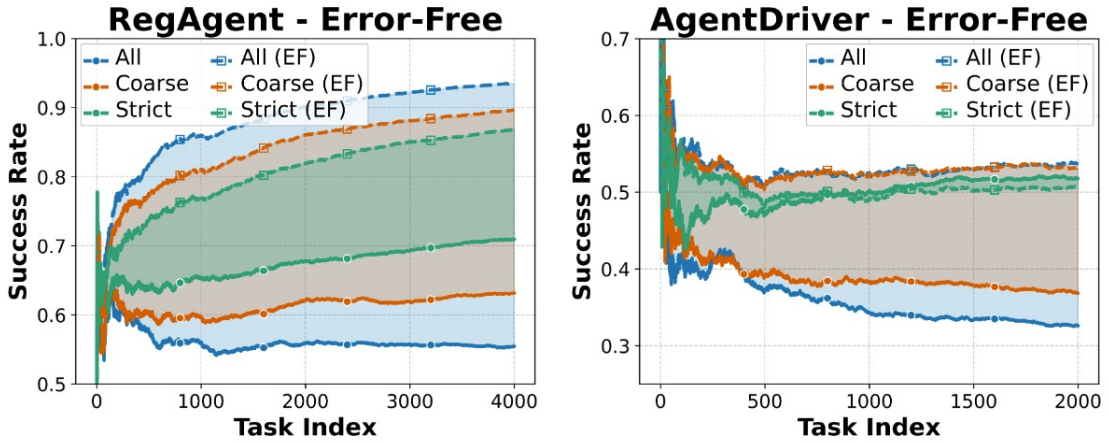
→ memory 기반 LLM agent의 experience-following property

Error propagation

- 오류는 메모리를 타고 전파된다

  : 잘못된 trajectory가 현재 실행을 망치고, 그 결과가 다시 저장된다

잘못된 경험 저장 → 유사 task에서 검색 → LLM이 오류 패턴 모방 → 새 오류가 다시 저장



* EF: memory에 저장되는 LLM execution을 ground-truth output으로 교체

Error-free variant와 비교하면 add-all/coarse addition에서 성능 gap이 커진다. Strict addition은 초기에는 느려도 장기적으로 안정화된다.

=> memory가 "학습"처럼 보이는 이유는 correct experience의 재생이고, failure도 같은 메커니즘으로 증폭된다.

Deletion

- Memory deletion: 무엇을 잊을 것인가

  : 용량 제한뿐 아니라 잘못 정렬된 experience replay를 줄이는 문제

Deletion	Criterion	Intuition
Periodic	최근 기간 검색 빈도 ≤ α	거의 쓰지 않는 기록 제거
History-based	검색된 뒤 평균 utility ≤ β	써봤더니 해로운 기록 제거
Combined	Periodic ∨ History-based	성능과 용량 균형

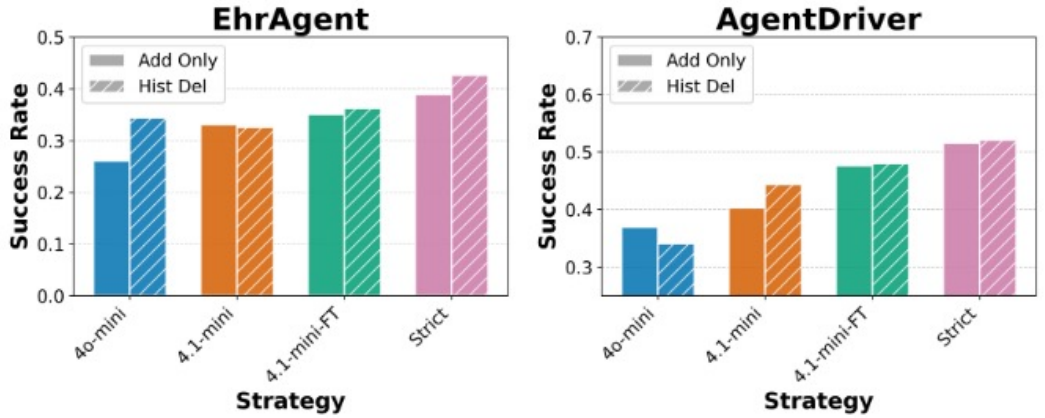
=> Periodic은 “덜 쓰이는 것”을 지우고, history-based는 “써봤더니 downstream에 나쁜 것”을 지운다.

Deletion

- reliable evaluator가 있으면 deletion도 성능을 올린다

  : Combined는 memory size를 가장 크게 줄이는 실용적 선택

Strategy	RegAgents		EHRAgents		AgentDriver		CIC-IoT Agent	
	SR. ↑	Mem Size ↓	ACC. ↑	Mem Size ↓	SR. ↑	Mem Size ↓	ACC. ↑	Mem Size ↓
Coarse evaluator (C1)								
No del	63.18	3511	25.91	1447	36.92	1161	74.00	1030
Period	60.88	1012	26.65	338	36.38	426	78.10	355
History	62.10	3205	33.55	1004	34.00	1019	73.70	952
Combined	59.32	951	31.47	279	35.62	372	68.80	352
Strict evaluator								
No del	**70.95**	2938	38.67	1012	51.00	1178	85.40	904
Period	67.65	949	38.59	302	50.94	467	80.80	310
History	69.80	2286	42.06	784	**51.81**	846	**89.60**	788
Combined	66.58	**890**	**42.34**	**248**	49.97	**323**	85.50	**188**



- Periodical-based deletion은 대체로 performance를 조금만 감소시키면서 memory size를 크게 줄임

→ deletion 없이 addition만 수행하면, 실제로 필요하지 않은 redundant entry가 memory에 많이 축적

- History-based deletion의 효과는 utility evaluator의 신뢰도에 크게 좌우

→ 단순히 오류가 없는 memory를 보관하는 것뿐 아니라, downstream task에서 utility가 높은 experience를 선별해 유지하는 것이 중요하다.

Misaligned experience replay

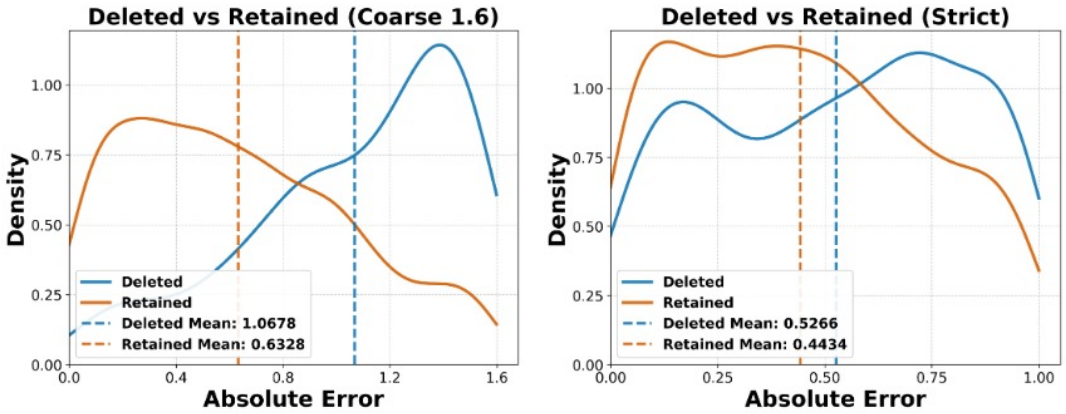
- History-based deletion으로 performance가 향상되는 이유를 설명하기 위해, **일부 memory record가 task execution에 거의 도움이 되지 않거나 오히려 해로운 guidance를 제공한다고 가정한다**

  : 과거에는 적절했던 execution이 현재 task에는 적절하지 않음

  : 저장된 trajectory와 현재 execution context가 일치하지 않음

- History-based deletion으로 performance가 향상되는 이유

  : 처음에는 좋은 experience로 판단되어 저장되었더라도, 이후 여러 task에서 계속 나쁜 결과를 유발하면 utility가 낮아지고 결국 삭제됨



history-based deletion이 실제로 품질이 낮은 메모리를 골라내고 있는가

→ 삭제된 메모리가 유지된 메모리보다 품질이 낮은 경향이 있다.

Challenging Scenarios

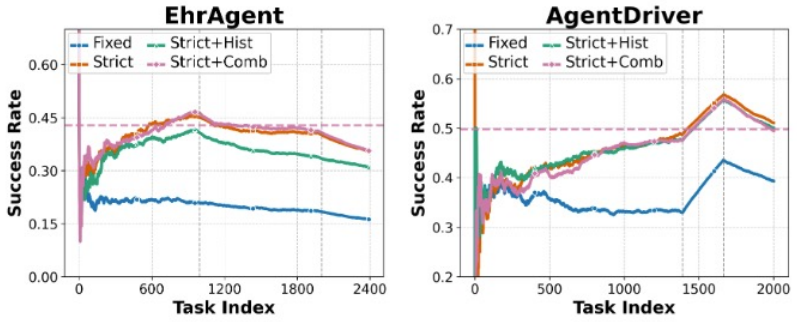
- 현실적인 제약이 있는 상황에서도 효과적인지

  1) 시간이 지나면서 처리해야 하는 task의 유형이 달라지는 task distribution shift

  2) memory bank에 저장할 수 있는 record 수가 제한되는 resource constraints

1) Task distribution shift

EHRAgent와 AgentDriver의 기존 test set을 수정 - 실행 도중 처리해야 하는 task distribution이 바뀌도록



- fixed memory
- strict addition
- strict evaluator를 사용하는 history-based deletion
- strict evaluator를 사용하는 combined deletion

- 전체적으로는 distribution shift가 없는 기준 설정과의 성능 차이가 크지 않다.

  → 적절한 memory 관리 전략을 사용하면 task 분포가 달라져도 성능을 비교적 안정적으로 유지할 수 있음

- AgentDriver에서는 strict addition만 사용한 설정이 매우 높은 성능

  → 새로운 distribution에서 생성된 경험을 엄격하게 평가하여 올바른 경험만 추가할 수 있다면, memory addition만으로도 환경 변화에 효과적으로 적응

- EHRAgent에서는 history-based deletion이 combined deletion보다 낮은 성능

  → distribution shift 환경에서는 periodical deletion도 성능 안정화에 기여

⇒ task distribution shift 상황에서는 새로운 고품질 경험을 추가하는 것뿐 아니라,
현재 distribution에서 더 이상 사용되지 않는 과거 경험을 주기적으로 제거하는 것도 중요하다.

Challenging Scenarios

- 현실적인 제약이 있는 상황에서도 효과적인지

  1) 시간이 지나면서 처리해야 하는 task의 유형이 달라지는 task distribution shift

  2) memory bank에 저장할 수 있는 record 수가 제한되는 resource constraints

2) resource constraints

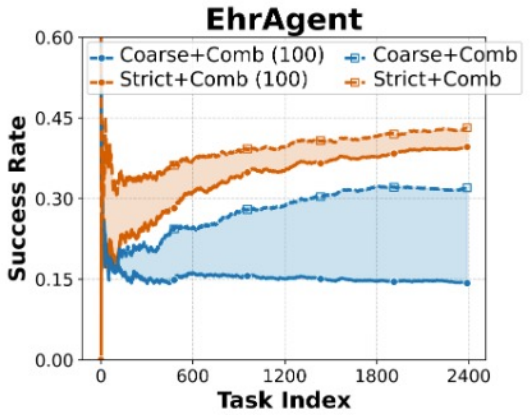
- capacity 제한에 맞게 combined deletion policy를 수정

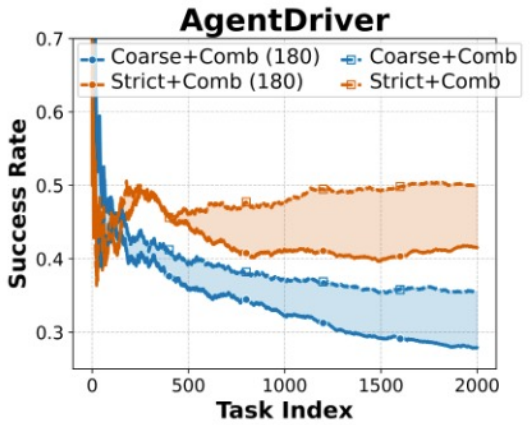
  Periodical-based deletion → 조건을 통과한 새로운 record addition → Memory capacity 초과 여부 확인

  → 초과한 경우 average utility가 가장 낮은 record 삭제

- memory management policy는 강한 capacity 제약이 있는 상황에서도 fixed-memory 설정보다 높은 성능

  → 관련성이 높고 품질이 좋은 record만 선택적으로 남김으로써 제한된 저장 공간을 효율적으로 사용할 수 있다





⇒ task distribution shift 상황에서는 새로운 고품질 경험을 추가하는 것뿐 아니라,
현재 distribution에서 더 이상 사용되지 않는 과거 경험을 주기적으로 제거하는 것도 중요하다.

Critical view

- 좋은 실증 분석이지만 실제 운영 시스템과는 간극이 있다

1) Strict evaluator

: 실제 human-in-the-loop가 아니라 ground truth 비교로 모사한 상한선

2) Distribution shift

: 자연적 장기 변화라기보다 실험적으로 구성된 query stream

3) Scope

: addition/deletion에 집중; summarization, merging, reflection은 제외

4) Causality

: experience-following은 강한 상관관계지만 내부 메커니즘 증명은 아님

=> memory bank의 “품질 관리” 없이는 agent가 장기적으로 안정화되지 않는다는 경험적 근거

ACL 26 Main

Mem2ActBench: A Benchmark for Evaluating Long-Term Memory Utilization in Task-Oriented Autonomous Agents

Yiting Shen[1], Kun Li[1], Wei Zhou[1], Songlin Hu[1,2]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyberspace Security, University of Chinese Academy of Sciences, Beijing, China

Motivation

- 사용자는 매번 모든 조건을 다시 설명하지 않음
  - 사용자의 선호, 요구사항, 진행 중인 작업의 상태는 여러 대화에 걸쳐 점진적으로 형성, 중간에 관련 없는 대화
- 현실적인 assistant는 요청에 필요한 과거 정보를 능동적으로 찾아 실제 행동에 적용해야 함
  - 단순히 long-term memory에 저장하는 데 그치지 않고

→ 사용자가 현재 요청에서 생략한 tool argument를 과거 대화로부터 보충해야 한다 !

- 기존의 memory benchmark

  특정 사실을 검색하는 fact retrieval 능력을 평가

Paradigm	Input	Output	What is tested?
Fact retrieval	"예산이 얼마였지?"	$500	질문에 명시된 fact 검색
Memory→Action	"뉴욕행 비행기 예약해줘"	search_flights(max_price=500, non_stop=True, …)	필요한 기억을 스스로 찾아 tool call에 grounding

→ 핵심은 기억한 사실을 답하는 것이 아니라 **어떤 기억이 필요한지 추론하고, 그것을 행동의 parameter로 grounding하는 것**

MEM2ACTBENCH

- MEM2ACTBENCH 제안

  - 긴 대화 기록 곳곳에 흩어진 정보를 이용하여 실행 가능한 tool argument를 복원할 수 있는지
  - 사용자의 최종 의도 자체는 분명하지만, 실행에 필요한 중요한 조건이 길고 interruption이 많은 과거 대화에 분산되어 있음

- 전체 데이터 구축 과정은 세 단계로 구성됨

  1) Heterogeneous Data Integration

  : Tool-use 중심 대화와 일반 대화를 섞어, 실제 사용자 기록처럼 여러 주제가 끼어드는 interruption-heavy interaction history를 만드는

  2) Fact/Memory Evolution Chain Construction

  : 대화에서 사실을 추출하고, 서로 충돌하거나 갱신되는 정보를 정리해 논리적·시간적으로 일관된 memory chain을 구성

  3) Memory-anchored Q&A Construction

  : memory로부터 완전한 정답 tool call을 먼저 만든 뒤, 핵심 parameter를 생략한 사용자 요청을 역으로 생성

  → **reverse-generation** 방식

  → 최종 query만으로는 정답 tool call을 결정할 수 없고, 반드시 과거 memory를 참고해야 하도록

MEM2ACTBENCH

1) Heterogeneous Data Integration

: 현실적인 장기 대화 기록을 만들기 위해 서로 성격이 다른 데이터를 결합

- Task-oriented Dialogue

: ToolACE, BFCL-v3

: LLM을 사용해 자연스럽고 일관된 multi-turn dialogue로 재구성

→ tool 실행에 필요한 정보가 한 번에 주어지지 않고 여러 turn에 분산된 대화

- Conversational Noise

: 현재 task와 관련없는 대화가 포함되는 실제 대화를 반영하기 위해 OASST1의 일반 대화를 conversational noise로 삽입

MEM2ACTBENCH

2) Constructing the Fact Evolution Chain

2-1) Fact Extraction and Grouping

대화로부터 LLM을 이용해 구조화된 fact를 추출 ***(attribute, fact, source ID)***

attribute: 사실이 어떤 속성이나 주제에 관한 것인지 나타내는 label

fact: 대화에 포함된 atomic user statement

source ID: 해당 사실이 어느 원본 dialogue에서 나왔는지 나타내는 식별자

clustering해서 하나의 평가 sample에 들어갈 일관된 synthetic long-term history 구축

추출된 attribute는 BERTopic으로 clustering, 내부 clustering algorithm으로 HDBSCAN을 사용

2-2) Memory Evolution Chain Construction -> 두 단계로 구축

1. Local Conflict Resolution

동일한 attribute를 공유하는 각 fact group 내부에서 일관된 local sequence를 만듦

이후 LLM으로

* Temporal ordering - 대화 속 시간 표현과 순서를 바탕으로 fact들을 시간순으로 배열
* Valid update preservation - 이전 정보가 더 구체적으로 발전한 경우 이를 정상적인 update로 유지
* Multi-valued trajectory preservation - 시간에 따라 여러 값이 모두 유효할 수 있는 경우 이를 보존
* Invalid fact removal - 문맥과 관계가 없거나 다른 사실과 강한 논리적 충돌을 일으키는 statement를 제거

MEM2ACTBENCH

2) Constructing the Fact Evolution Chain

2-2) Memory Evolution Chain Construction -> 두 단계로 구축

2. Global Evolution Sequence Construction

여러 local sequence를 하나의 global memory chain으로 합친다

dependency graph를 만들고

node: 각각의 fact

directed edge: 두 fact 사이의 시간적 선후관계

Kahn's algorithm을 변형한 topological sorting을 적용하여 전체 fact를 하나의 시간 순서로 정렬

➔ 기존 순서를 깨뜨리지 않게 하기 위해

```
[거주지]
L1: 서울에 거주
L2: 부산으로 이사
→ L1이 L2보다 먼저

[음식 선호]
F1: 모든 음식을 먹음
F2: gluten-free 식당을 선호
→ F1이 F2보다 먼저

[여행 선호]
T1: 경유편도 괜찮음
T2: 직항편만 선호
→ T1이 T2보다 먼저
```

→ 이 목록들을 한 줄로 합침 → L1 → F1 → T1 → L2 → F2 → T2

Local Conflict Resolution

MEM2ACTBENCH

3) Memory-anchored Q&A Construction

: ① 어떤 tool을 쓸지 고르고 → ② 그 tool의 빈칸을 memory로 채워서 완성된 정답 tool call을 만드는

3-1) Target Tool Selection and Parameter Anchoring

BM25와 BGE-M3로 관련 있는 tool 후보를 찾고, LLM이 최종 tool을 결정

선택된 tool의 argument를 memory에서 채움

3-2) Reverse Implicit Query Generation and Filtering

완전한 tool call C를 만든 뒤, 이를 바탕으로 불완전한 사용자 query Q를 역으로 생성

```
PlacesSearchAPI(
    location="Austin, TX",
    dietary_preference="gluten-free"
)
```

→ 오늘 저녁 외식하고 싶은데 적당한 곳을 찾아 줘.
(Agent가 이를 채우려면 반드시 과거 memory를 참고해야 하도록)

Dataset quality

- 데이터 규모와 human verification

  : 자동 생성이지만 세 단계에서 전문가 검증을 수행

- 400 tool-use tasks
- 2,029 sessions
- 13 avg turns/session
- 91.3% strongly memory-dependent

추출된 fact가 실제 dialogue 내용으로부터 entail되는지 올바르게 정규화되었는지

구성된 memory evolution chain이 일관적인지 시간에 따른 정보의 update를 적절히 반영하는지

현재 query만 주어졌을 때 gold tool invocation이 여전히 underdetermined한지

Verification target	Reported quality
Fact extraction	96.5% correct
Conflict resolution	86.7% acceptable
Memory dependency	91.3% strongly memory-dependent

Evaluation setup

- 평가 설정: memory framework의 parameter grounding을 본다

  :기존 memory system들이 과거 대화에서 필요한 정보를 찾아 tool parameter를 얼마나 정확하게 채울 수 있는가?

  : Main experiment에서는 모델에게 정답 tool 자체를 미리 제공

  → 어떤 Tool을 사용해야하는지는 알려주고, 그 파라미터를 메모리에서 잘 찾는지!

Component	Setting
Memory frameworks	LTMemory, SCM, Generative Agents, MemTree, Mem0, LangMem, A-Mem
LLM backbone	Qwen2.5-7B / 32B / 72B
Retriever	BM25, dense, hybrid, oracle
Metrics	Parameter F1, BLEU, Tool Selection Accuracy / Tool Accuracy

평가의 핵심 질문
“검색된 기억을 tool schema에 맞게 정확한 값으로 넣는가?”

- end-to-end tool selection은 별도 hard-negative 실험에서 분석함

Main Results

- 모델이 커질수록 parameter grounding이 좋아짐

  : 하지만 하지만 증가 폭은 점점 작아짐

Method	*Qwen2.5-72B-Instruct*			*Qwen2.5-32B-Instruct*			*Qwen2.5-7B-Instruct*			**Average**		
	F1	**BLEU**	**TA**	**F1**	**BLEU**	**TA**	**F1**	**BLEU**	**TA**	**F1**	**BLEU**	**TA**
LTMemory	35.32	67.93	92.00	**33.87**	67.71	90.20	26.71	64.07	87.25	31.97	66.57	89.82
SCM(Wang et al., 2023)	22.73	62.04	96.25	17.35	59.37	95.75	14.99	57.37	91.00	18.36	59.59	94.33
Generative Agents(Park et al., 2023)	22.38	62.58	95.50	17.81	59.76	96.24	14.44	55.59	89.75	18.21	59.31	93.83
MemTree(Rezazadeh et al., 2024)	33.21	**68.17**	94.25	31.89	67.69	94.50	24.60	63.91	88.50	29.90	66.59	92.42
Mem0(Chhikara et al., 2025)	28.95	66.47	**96.75**	24.52	64.36	97.00	14.21	57.37	93.97	22.56	62.73	95.91
Langmem(LangChain, 2025)	24.01	63.06	96.25	18.72	58.27	**97.25**	17.06	58.93	**96.50**	19.93	60.09	**96.67**
A-mem(Xu et al., 2025)	**35.93**	67.93	93.25	33.72	**67.92**	92.25	**30.99**	**66.47**	94.75	**33.55**	**67.44**	93.42

Table 3: Experimental results for different memory methods across multiple model sizes.

→ 단순히 LLM 크기만 키운다고 해결되는 문제 X

Retrieval bottleneck

- 결과 2: 가장 큰 병목은 retrieval이다

  : 하지만 oracle retrieval도 완전한 해결책은 아니다

Retrieval Strategy	Recall@k	F1	BLEU	TSA
No Retrieval	–	10.0	8.9	73.8
Passive Retrieval				
BM25	1	29.0	28.0	87.0
	5	26.9	26.2	90.2
	10	27.6	26.7	87.5
Dense	1	25.6	25.3	83.0
	5	29.9	29.1	90.0
	10	28.7	28.3	88.8
Hybrid	1	24.9	24.3	85.0
	5	**30.7**	**29.7**	86.0
	10	30.3	29.5	88.8
Perfect Retrieval				
Oracle	–	**53.8**	**53.7**	88.2

Hybrid@5
F1 30.7

→

Oracle
F1 53.8

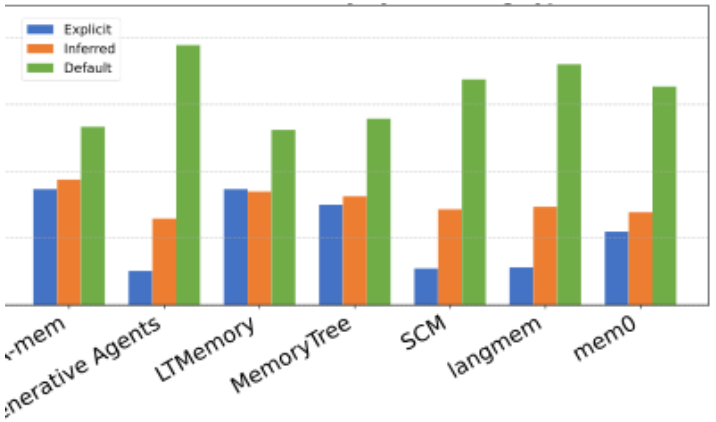
차이 약 +23 F1: implicit query에서 필요한 기억을 찾는 것이 핵심 병목이다.
그러나 oracle도 53.8에 그쳐 post-retrieval reasoning 문제도 남는다.

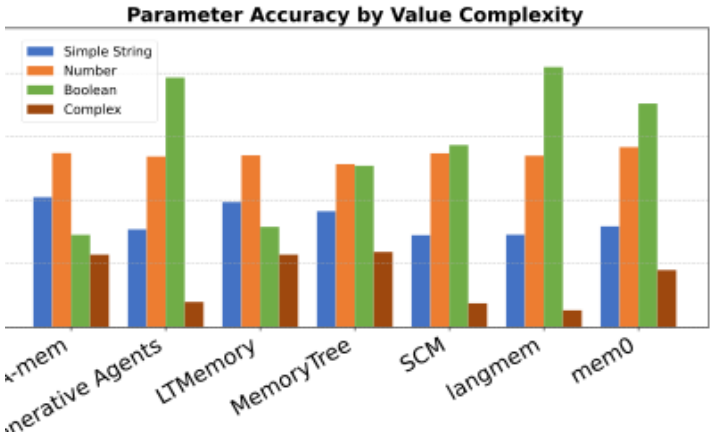
Where grounding fails

- 결과 3: 중간 기억·복잡한 값·default에서 많이 실패

  : 검색 위치와 값의 구조가 parameter accuracy를 좌우한다

- Mid-context valley 25–50% 구간 기억에서 F1 하락
- Default value failure “언급 없음”을 인식하지 못하고 그럴듯한 값을 생성
- Lossless retention failure URL/주소/ID/중첩 구조에서 truncation·문자 오류



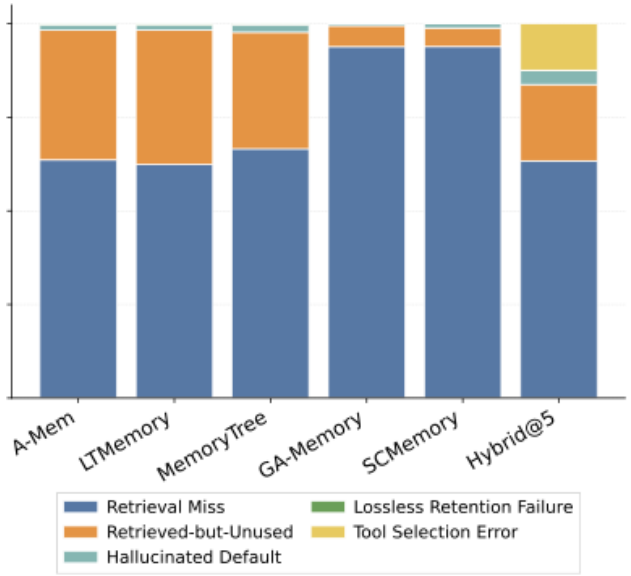


Tool robustness & failure modes

- 결과 4: hard negatives와 failure diagnosis

  : 검색 실패가 크지만, 강한 시스템에서는 "찾고도 못 쓰는" 비중이 커진다

Candidate size N	1	2	5
Random negatives			
TSA (↑)	94.50	95.50	93.50
EM (↑)	18.25	18.00	17.00
Arg_F1 (↑)	29.88	29.98	28.27
Hard negatives			
TSA (↑)	94.50	78.00	69.75
EM (↑)	18.25	16.50	14.25
Arg_F1(↑)	29.88	27.17	22.64



Hard negatives에서 tool selection accuracy는 N=5일 때 69.75%까지 감소하고, Arg_F1도 22.64로 하락한다.

Failure modes: Retrieval Miss, Retrieved-but-Unused, Hallucinated Default, Lossless Retention Failure, Tool Selection Error.

Critical view

- 좋은 벤치마크이지만 실제 agent execution 전체를 포괄하지는 않는다

1) Synthetic pipeline

: ToolACE/BFCL/OASST1 + LLM 생성이어서 실제 사용자 패턴과 차이 가능

2) Offline tool-call generation

: 도구 실행 결과를 보고 계획을 수정하는 closed-loop 평가는 아님

3) Tool selection separation

: 메인 실험은 정답 tool 제공; end-to-end 실행력과는 구분

4) Metric clarity

: TA/Parameter F1의 관계는 코드 수준 설명이 더 필요

=> memory benchmark를 “recall”에서 “action grounding”으로 이동시킨다

Thank you