Agent Memories: A Database Perspective

Week 8 – Machine Learning Systems (Fall 2026)

Yang Cao

University of Edinburgh

Recall: from retrieval to shared memory



- Week 2: encode text/image as vectors for learning and retrieval
- Week 7: efficient vector search in full ML stack (RAG, VDBMS)
- Today: how LLM-backed agents access and share data (a.k.a. memories)

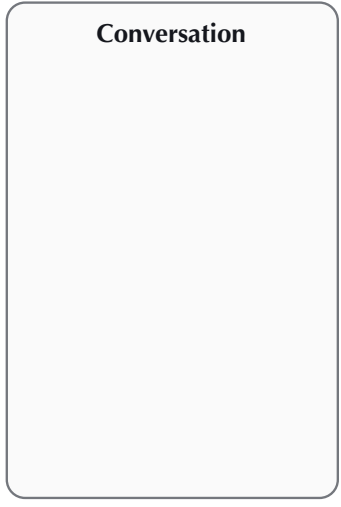
Goal: a database view of agent memory management, to reveal underlying principles

Learning points

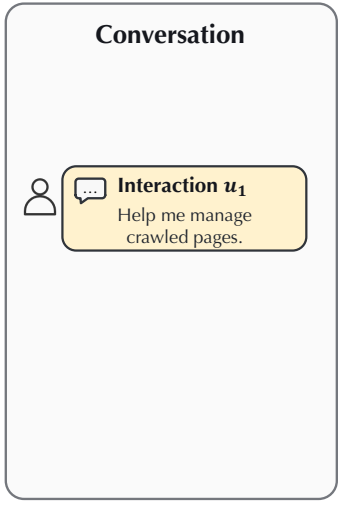
1. What are LLM calls and agents and why do we need agent memory?
2. Agent memories more than markdown files
3. Consistency on agent memories, a 'C'-view from ACID's pespective
4. Exemplar agent memory systems
5. connection with LLM KV cache

Sharing memory across agents is just like a transactional database shared by concurrent users

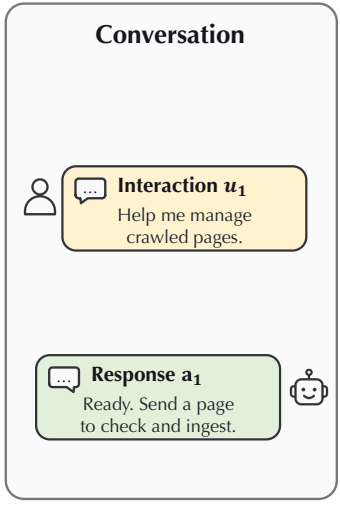
Running example: a conversation with an agent



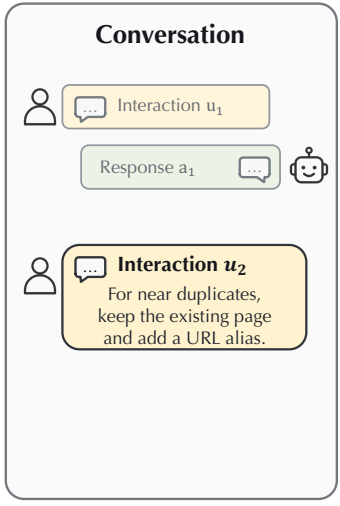
Running example: a conversation with an agent



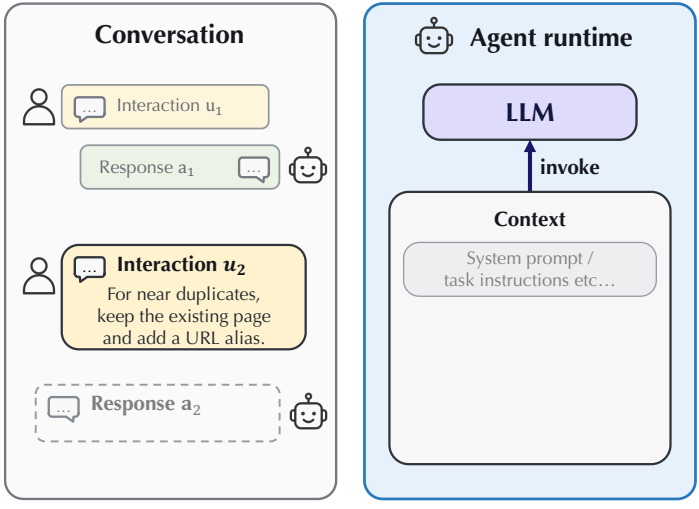
Running example: a conversation with an agent



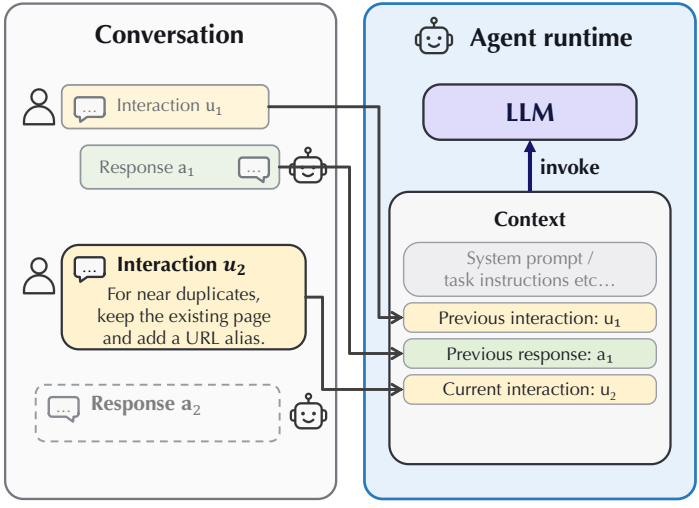
Running example: a conversation with an agent



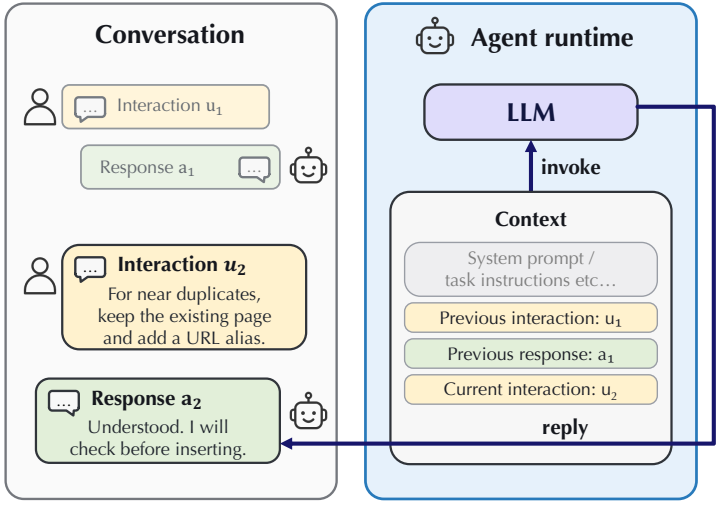
Running example: the runtime builds the context



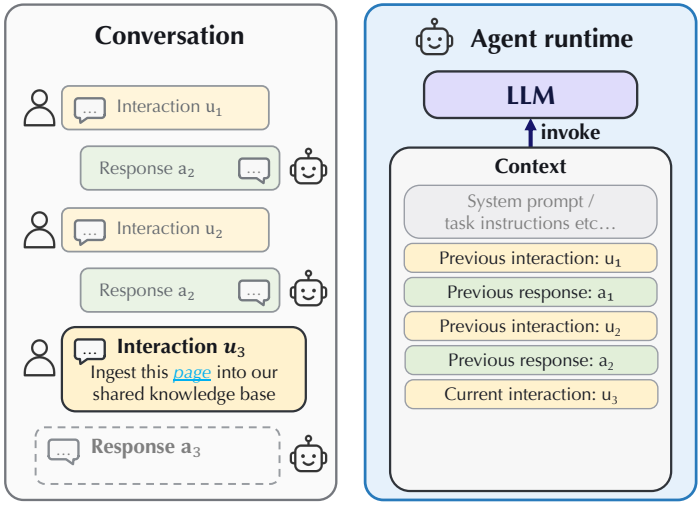
Running example: the runtime builds the context



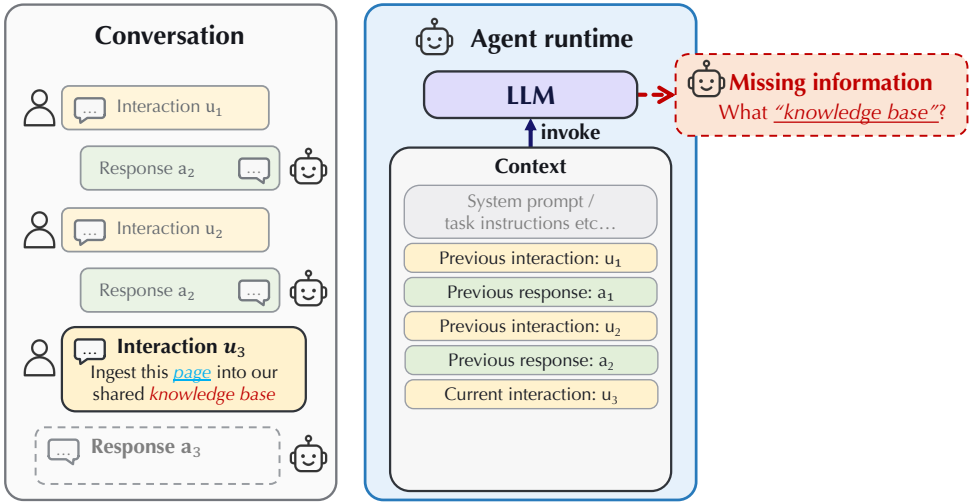
Running example: the runtime builds the context



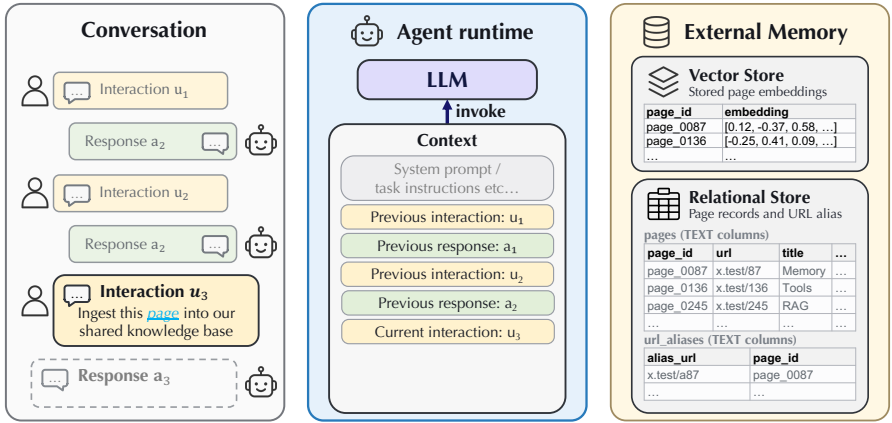
Running example: missing information in turn 3



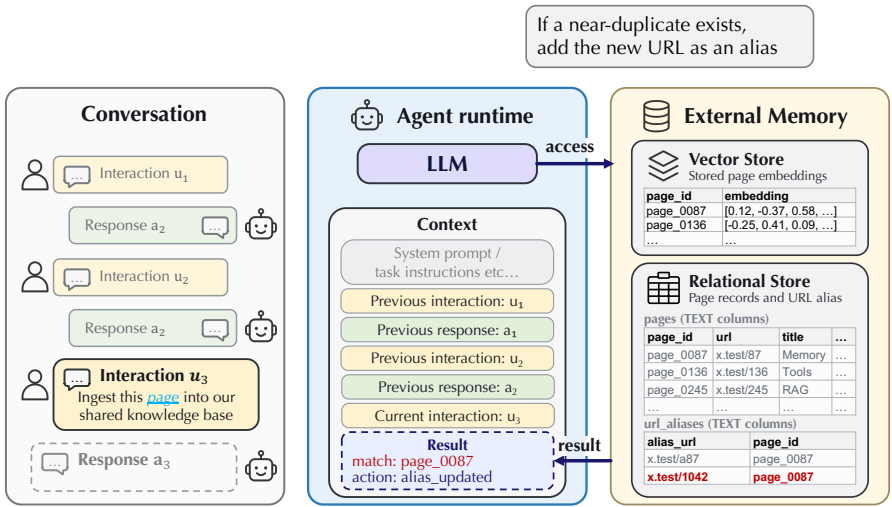
Running example: missing information in turn 3



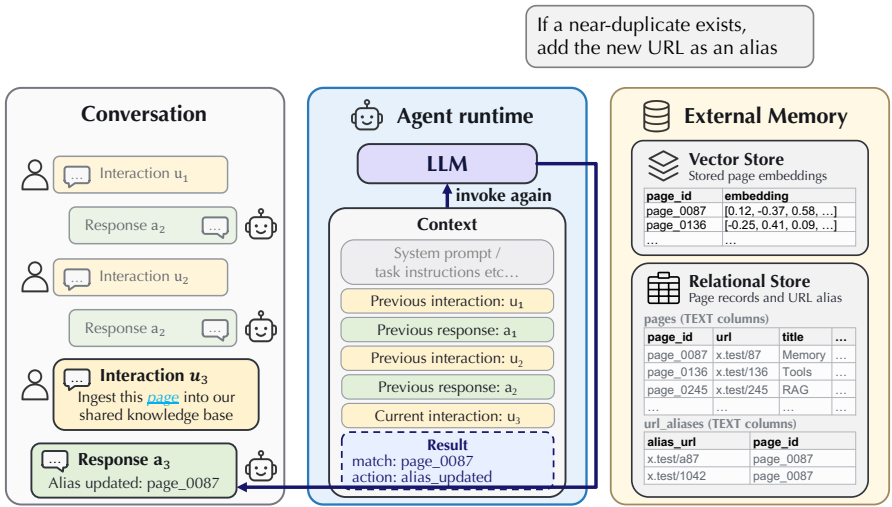
Running example: external memory (near-duplicate case)



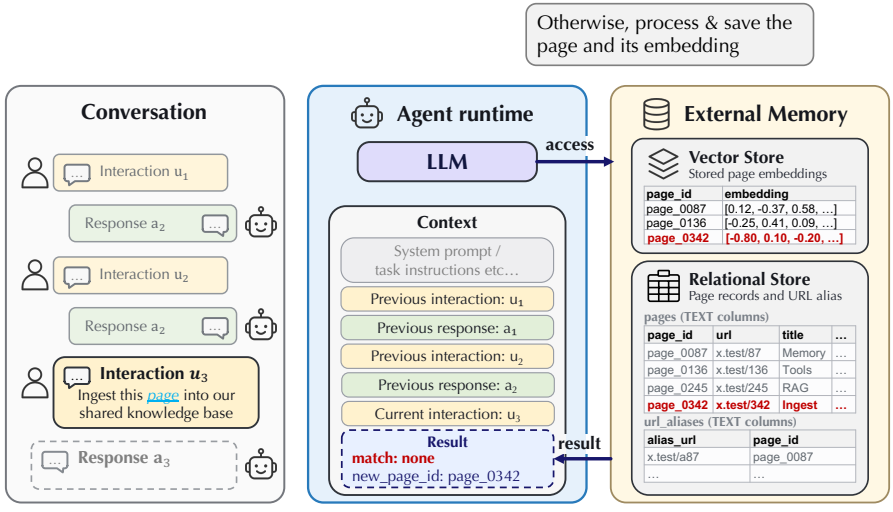
Running example: external memory (near-duplicate case)



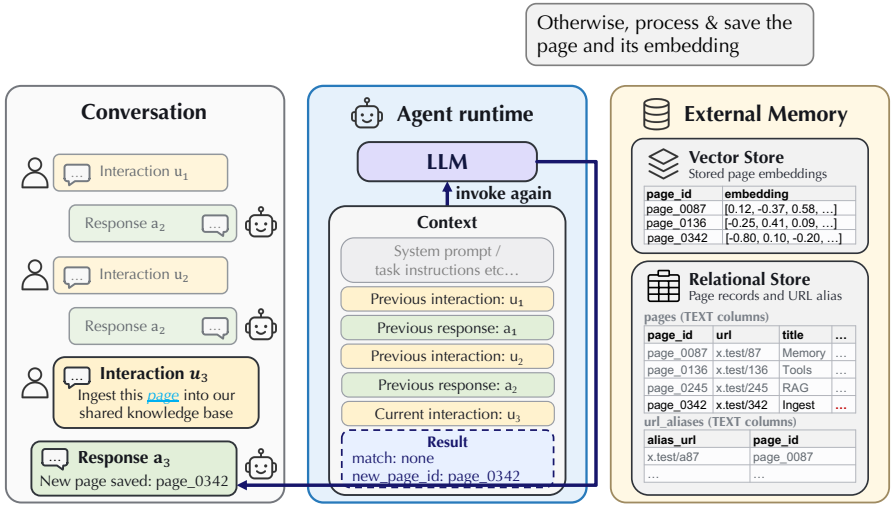
Running example: external memory (near-duplicate case)



Running example: external memory (new-page case)



Running example: external memory (new-page case)



What does one LLM call receive?

```
Call 1 input:  Our knowledge base lives at kb.x.test.
Call 2 input:  Where does our knowledge base live?
```

Call 2 is a separate request. Its input does not contain the address; so can't answer.

abstract view of LLM call: $x_t \sim p_\theta(\cdot \mid x_1, \ldots, x_{t-1})$.

- $x_1, \ldots, x_{t-1}$: the tokens of this call's input. We call them the *context*.
- θ: the model parameters (fixed during inference).

Call 2 can use the address only if the runtime stores it and supplies it again.

An example agent: web page crawl assistant

A page crawl example: a user asks an assistant to ingest the crawled web page x.test/1042 into a shared knowledge base. The knowledge base should not store a page twice, even when the two copies differ a little. Such copies are *near-duplicates*.

Arrangement	On the web page x.test/1042	Next step set by
One model call	Summarise this page	Application code
Tool calling	Emit `ingest_page("x.test/1042")`	Model output; code runs it
Fixed workflow	Fetch, embed, search, insert if new	Application code
Agent loop	Read the tool result, emit the next call	Model output, each step

- A *tool call* is structured text in model output; runtime checks it and runs the function.
- An *agent* in this lecture is a loop of model calls and tool results.

interpretation taken from "Building effective agents" (2024), by Anthropic

Interact with the agent: turns 1 and 2

Speaker	Message
User 1	Help me manage crawled pages.
Assistant	Ready. Send a page to check and ingest.
User 2	For near-duplicates, keep the existing page and add a URL alias.
Assistant	Understood. I will check before inserting.

- After turn 2 the conversation holds four messages, and no tool has run.

Where is the near-duplicate rule from turn 2 stored at this point?

Interact with the agent: turns 1 and 2

Speaker	Message
User 1	Help me manage crawled pages.
Assistant	Ready. Send a page to check and ingest.
User 2	For near-duplicates, keep the existing page and add a URL alias.
Assistant	Understood. I will check before inserting.

- After turn 2 the conversation holds four messages, and no tool has run.

Where is the near-duplicate rule from turn 2 stored at this point?

Only in the conversation; starting a new session would lose the rule

Turn 3

What the model sees for turn 3 (user 3):

```
system:  You manage crawled pages. Tool: ingest_page(url).
user 1:  Help me manage crawled pages.
reply 1: Ready. Send a page to check and ingest.
user 2:  For near-duplicates, keep the existing page and add a URL alias.
reply 2: Understood. I will check before inserting.
user 3:  Ingest x.test/1042 into our shared knowledge base.
```

- Context: all these six messages (the stored pages are not part of model input/context)
- The model output is a tool call: `ingest_page("x.test/1042")`.

The stored pages are outside the context (part of the agent memory)

The shared knowledge base before turn 3

Relational store: `pages`

page_id	url	title
page_0087	x.test/87	Memory
page_0136	x.test/136	Tools
page_0245	x.test/245	RAG

Relational store: `url_aliases`

just one row: x.test/a87 $\rightarrow$ page_0087.

Vector store: embeddings

page_id	toy embedding (E)
page_0087	-0.70
page_0136	0.60
page_0245	0.30

E: toy embeddings of the page for duplication (similarity) checking

ingest_page tool call

```
def ingest_page(page):
    cands = memory.read_vector(query=page.emb, top_k=k)    # 1. search
    match = verify_near_duplicate(page, cands)
    if match is None:                                      # 2. new page
        record = process_page(page)
        memory.insert_page(record)
        memory.insert_vector(page_id=record.id, emb=page.emb)
        return {"action": "page_inserted", "page_id": record.id}
    memory.upsert_alias(url=page.url, page_id=match.id)    # 3. alias
    return {"action": "alias_updated", "match": match.id}
```

insert page only when the search finds no near-duplicate

Tool outcome A: x.test/1042 is a near-duplicate

- Incoming page: x.test/1042, toy $E = -0.67$. Threshold: $\tau = 0.10$.
- Nearest stored page: page_0087 at -0.70. Distance $0.03 \leq 0.10$, so it is a match.

Store	Change
pages	none
embeddings	none
url_aliases	add x.test/1042 $\rightarrow$ page_0087

Tool result: `action: alias_updated, match: page_0087`

The result enters the context. The next model call replies: Alias updated: page_0087.

Tool outcome B: x.test/342 is a new page

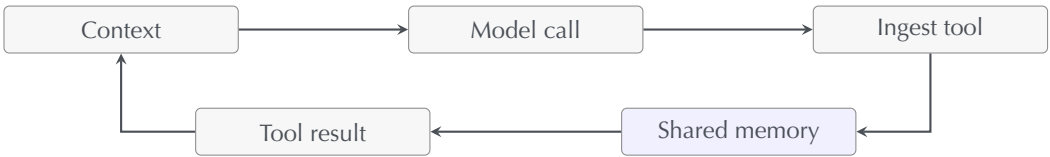
- Incoming page: x.test/342 (`page_0342`), title Ingest, toy $E = -0.25$.
- Nearest stored pages: page_0087 at 0.45, page_0245 at 0.55. Both exceed $\tau = 0.10$.

Store	Change
pages	add page_0342, x.test/342, Ingest
embeddings	add page_0342, -0.25
url_aliases	none

Tool result: `action: page_inserted, page_id: page_0342`

Reply: New page saved: page_0342; two writes to `pages` and `embeddings` tables.

Turn 3 end to end: two model calls and one tool run



- Model call 1 emits `ingest_page("x.test/1042")`.
- The tool result becomes message 7 of the context.
- Model call 2 writes the reply from that result.

What did this interaction have to retain?

	In our example	source
Policy	Keep the existing page, add an alias	Conversation, turn 2
Shared records	3 pages, 1 alias, 3 vectors	External store
Tool result	alias_updated, page_0087	Context, message 7
Source text	The page at x.test/1042	Crawler output

agent memory is the state that an agent system (system@LLM) keeps outside context to affect later model calls and the operations that read and write it.

outside the chat context/history.

Why not just text files (e.g., AGENTS.md and CLAUDE.md)?

```
# AGENTS.md
Before inserting a crawled page, check for a near-duplicate.
If one exists, keep it and add the new URL as an alias.
```

Property	Codex: AGENTS.md	Claude Code: CLAUDE.md
Written by	User	User; Claude writes auto memory
Loaded	Once per run	Start of every session
Size limit	32 KiB by default	Auto memory: 200 lines or 25 KB
Role	Guidance in the prompt	Context, not enforced configuration

can record rule from turn 2, but it's insufficient/incomplete

Memory as markdown files

```
kb/page_0087.md: title Memory, url x.test/87
kb/page_0136.md: title Tools, url x.test/136
kb/page_0245.md: title RAG, url x.test/245
kb/aliases.md: x.test/a87 -> page_0087
```

Turn 3 needs three **operations** on these files:

1. Find a stored page that is a near-duplicate of x.test/1042.
2. If one exists, add a line to `aliases.md`.
3. Otherwise create a new page file.

can we use md files and filesystem to serve as memory of this agent (webpage crawl)?

Operation 1: find the near-duplicate (via md files)

Assume 10,000 stored pages of 1,000 tokens each. The incoming page is x.test/1042.

Method	Work per ingest	Finds page_0087?
`grep` for the URL	One scan of the files	No. The URLs differ
`grep` for the title	One scan of the files	Only for an equal title
Model reads every file	10,000,000 input tokens	Yes, at that cost
Compare stored embeddings	10,000 distances	Yes. Distance 0.03
Vector index from Week 7	Far fewer distances	Usually. ANN can miss

needs embeddings and indices (RAG vs. filesystem)

Operation 2: two agents add an alias (coordination over md files)

```
t1: A reads aliases.md, 1 line: x.test/a87 -> page_0087
t2: B reads aliases.md, 1 line: x.test/a87 -> page_0087
t3: A saves 2 lines, adding x.test/1042 -> page_0087
t4: B saves 2 lines, adding x.test/2042 -> page_0136
```

- Final file: the first line and B's line. A's alias is gone.
- Neither agent received an error.
- In database terms this is a *lost update*. A lock or a version check on the file prevents it.

We need conflict resolution to coordinate conflicting operations

Connecting the dots: what turn 3 needs from storage

need in turn 3	dealt by
keep the page text	files or a document store
look up a page by ID or URL	relational table with keys
find similar pages	embeddings and a vector index
link pages to the entities they mention	graph store
apply several writes as one unit	transactions

agent memory for this example: a hybrid data system

files, relational records, embeddings and indexes.

Connecting the dots: what turn 3 needs from storage

need in turn 3	dealt by
keep the page text	files or a document store
look up a page by ID or URL	relational table with keys
find similar pages	embeddings and a vector index
link pages to the entities they mention	graph store
apply several writes as one unit	transactions

agent memory for this example: a hybrid data system

files, relational records, embeddings and indexes.

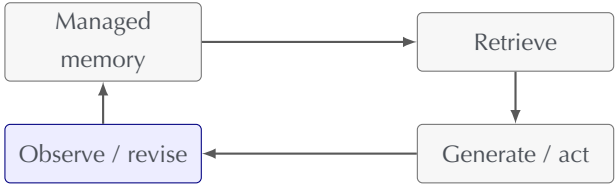
Question: Can we use git?

Example agent memory systems

System	Relational	Vector search	Graph
Mem0 OSS	SQLite history	Qdrant; pgvector	Not in OSS
Letta V1	PostgreSQL	pgvector	None
LangMem + LangGraph	PostgreSQL	pgvector	None
Graphiti	None	Inside the graph database	Neo4j; FalkorDB
A-Mem	None	ChromaDB	None
MIRIX	PostgreSQL	pgvector; Redis cache	Neo4j, optional
MemOS	None	Qdrant	Neo4j
Cognee OSS	SQLite; PostgreSQL	LanceDB; pgvector	Ladybug; Neo4j

- All eight systems have a vector search component.
- Seven of the eight pair it with a relational store or a graph store.

Memory vs. RAG (Week7)



- RAG's read path: chunk, embed, index, retrieve, generate (the corpus is given).
- In turn 3 the agent's own tool call changed the corpus (e.g., one alias row added).

RAG is the read path of memory. Memory needs both read and write.

And write path causes troubles for multi-agent interaction

One ingest as a sequence of reads and writes

Ingest of x.test/342, toy $E = -0.25$:

Step	Operation	On our data
BEGIN	Start a transaction	
R_V	Read vectors near the page	Nearest distance 0.45: no match
W_R	Write the relational row	page_0342, x.test/342, Ingest
W_V	Write the vector	page_0342, -0.25
COMMIT	Make both writes permanent	Store holds 4 pages

- A *transaction* is the group of operations between BEGIN and COMMIT
- adding alias: R_V, then W_A (the write of one alias row)

The no-duplicate memory rule

the memory (store) never holds two pages that are near-duplicates of each other.

Let P be the set of stored pages, E the encoder, d the distance and τ the threshold.

$$I(P) \equiv \neg\exists\, a, b \in P :\ a \neq b \ \wedge\ d(E(a), E(b)) \leq \tau.$$

- Our toy case: d is the absolute difference and $\tau = 0.10$
- In database terms such a rule is an *invariant*: P is consistent if $I(P)$ holds
- a commit (consistent write) that changes P into P': if $I(P)$ holds, then $I(P')$ must hold

Two new pages arrive: p and q

```
p: x.test/345, title llm, toy E = 0.00
    LLM agents store facts in external memory.
q: x.test/346, title LLM, toy E = 0.08
    LLM agents store facts in an external memory.
```

- p and q are near-duplicate pages: their distance is $0.08 \leq 0.10$ threshold
- Each doesn't have duplicate in `pages` (memory):

  closest stored page is 'RAG': 0.30 from p and 0.22 from q).

If p is stored first, what should `ingest(q)` do?

Two new pages arrive: p and q

```
p: x.test/345, title llm, toy E = 0.00
    LLM agents store facts in external memory.
q: x.test/346, title LLM, toy E = 0.08
    LLM agents store facts in an external memory.
```

- p and q are near-duplicate pages: their distance is $0.08 \leq 0.10$ threshold
- Each doesn't have duplicate in `pages` (memory):

  closest stored page is 'RAG': 0.30 from p and 0.22 from q).

If p is stored first, what should `ingest(q)` do?

add x.test/346 as an alias of p (don't store q as a second page)

serial ingests

Step	Operation	Result
1	A: R_V for p	Nearest is RAG at 0.30. No match
2	A: W_R, W_V, COMMIT	p is stored
3	B: R_V for q	Finds p at 0.08. Match
4	B: W_A, COMMIT	add alias x.test/346 $\rightarrow$ p. No new page

- Final store (`pages` relation): Memory, RAG, Tools, p. Every pair is at least 0.30 apart.
- first q then p: stores q and makes x.test/345 an alias (the rule holds again)

any *serial* transactional execution of multiple ingests keeps memory always consistent

Case 1: two agents ingest at the same time

Time	Agent A: ingest(p)	Agent B: ingest(q)
t1	R_V: nearest 0.30, no match	
t2		R_V: nearest 0.22, no match
t3	W_R, W_V: insert p	
t4	COMMIT	
t5		W_R, W_V: insert q
t6		COMMIT

- Final memory: p and q stored with distance $0.08 \leq \tau$ (inconsistent memory state)

B's search at t2 was correct for the memory at t2 but incorrect at t6?

Question: would a unique key constraint on `pages` have stopped this?

Learning from the past (DBMS): transactions and ACID

Property	A transaction guarantees	In our `ingest`
Atomicity	All of its writes happen, or none	Page row plus vector
Consistency	Each commit keeps the declared rules	No pair within τ
Isolation	Concurrent ones do not interfere	A and B in Case 1
Durability	Committed data survives a crash	p stays stored

- In Case 1 each transaction alone kept the rule. Their interleaving broke it.
- So Case 1 is a failure of *isolation*.

Learning from the past (DBMS): isolation levels

Assume pages and vectors live in one PostgreSQL database. Replay Case 1 at each level:

Level	A transaction reads	Case 1 result
Read Committed	Commits before each statement	p and q stored (inconsistent)
Repeatable Read	One snapshot per transaction	p and q stored (inconsistent)
Serializable	As if run one at a time	One aborts, then retries

retry runs after the other commit; its search finds the stored page and adds an alias

Connecting the dots: one transaction or three?

In one transaction

```
BEGIN
  R_V: search
  W_R: insert page
  W_V: insert vector
COMMIT
```

Separated transactions

```
BEGIN; R_V: search; COMMIT

BEGIN; W_R: insert page; COMMIT

BEGIN; W_V: insert vector; COMMIT
```

- In one: both stores are in one database, such as PostgreSQL with pgvector.
- Separated: lines 1 and 3 run in the vector store, line 2 in the relational store.

An isolation level protects only what runs inside one transaction.

Case 2: the vector store lags behind `pages` table

A finishes before B starts. A's vector write is queued. The columns list only p and q.

Time	pages table	Vector store	Event
t1	p	none	A commits p. Its vector is queued
t2	p	none	B: R_V for q finds no match
t3	p, q	none	B commits q
t4	p, q	p, q	The queue is applied

- At t4 both stores agree again. The pair p, q stays stored at distance 0.08.

The two ingests did not overlap. The vector store was one write behind the table.

Learning from the past (DBMS): index consistency

Derived structure	Updated	Can lag?
Index in PostgreSQL	Inside the INSERT transaction	No
Materialized view	At the next REFRESH	Yes
Asynchronous replica	When the log is replayed	Yes
Separate vector store	By a second write or a queue	Yes

- index, view and replica are *derived* data; each can be rebuilt from the table.
- pgvector index is a PostgreSQL index (so always consistent)

A derived structure agrees with its table at every commit
only if
both change in one transaction

A baseline routine

```
vector = encode(page)                        # fixed model, before locking
with transaction():                          # one database, READ COMMITTED
    lock_scope_guard(scope)                  # one ingest at a time
    match = exact_nearest(scope, vector)     # scans the pages table
    if match is not None and match.distance <= tau:
        add_alias(page.url, match.id)
    else:
        insert_page_and_vector(page, vector)
    enqueue_index_update()                   # ANN index catches up later
```

- The lock orders overlapping ingests: Case 1
- transactional scan reads vectors in `pages`: Cases 2 (and potential ANN errors)

Replay Case 1 with the baseline



B searches after A's commit. It finds p at 0.08 and adds the alias x.test/346.

How does baseline address memory consistency?

Case	What the baseline does	Result for q
case 1: overlap	B waits for the lock, then searches	Finds p. Alias
case 2: lagging index	Check reads the pages table	Finds p. Alias
(ANN miss)	Check computes all 4 distances	Minimum 0.08. Alias

- Final `pages` in all three replays: Memory, RAG, Tools, p.
- Final aliases: x.test/a87 $\rightarrow$ page_0087 and x.test/346 $\rightarrow$ p.

baseline is inefficient:

lock forces ingests of the same scope run one at a time

exact scan does not exploit ANN indices and is impractical

Case study: Cognee

Cognee by default: SQLite for rows, LanceDB for vectors, Ladybug for the graph.

Our rule needs	system guarantees	left to the application
Detect identical bytes	Yes (content_hash lookup)	Nothing
Detect a near-duplicate	No	Distance test against τ
Page plus URL aliases	No	pages and url_aliases tables
Case 1: one writer at a time	lock in one process	Locking across processes
Case 2: vector with its page	separate writes	A check that sees every page

In Cognee, 1st stage commits a row of SQL table `data`. Graph nodes and vectors follow later.

Memory consistency rules and their enforcement still rely on application

Discussion: are RAG and the KV cache part of agent memory?



State	For page_0087	Reusable by
Stored record	Row, file and vector	Any agent with access
Retrieved context	Its text inside one prompt	That model call
KV cache	Attention state of that prompt	Same model, same prefix

- RAG is the arrow from the store to the context: read path of agents
- MemOS activation memory: runtime hidden states and KV-cache.