The Living Craft · Resources · agentic system design 05

Agent Memory Audit Kit

Every fact your agent remembers needs three answers: where it came from, where it applies, and what happens when someone corrects it.

Sunil Mathew · The Living Craft

https://learning.thelivingcraft.ai/resources/agent-memory-audit-kit

Published 17 September 2026 · Content CC BY 4.0 · Code MIT · © The Living Craft

Why this exists

The stale memory never reached the client. The fix did.

An expense agent remembered a project code Priya typed once, for one trip, and reused it on the next trip. The team shipped a fix: learn from user corrections. Three weeks later, Priya corrected a code for a client dinner. That Friday, at the same restaurant, the agent put her own team's dinner on the client's bill.

The stale memory never reached the client. The fix did. A correction is a memory too, and it needs the same discipline as any other memory.

Illustrative scenario. The failure pattern is real.

The three questions

Source, scope, correction.

Source

Who said it?

"Typed ATL-2291 for this trip" and "uses ATL-2291" are different facts. Store what the user stated, what a system record says, and what the agent inferred as different things.

Scope

Where, and until when?

One request, one project, one quarter. A fact without a boundary quietly becomes a rule nobody wrote.

Correction

What exactly changes?

This expense, this client, or every future dinner? A correction without a scope is the next wrong memory.

01 The memory record schema

Every field exists because a real failure needed it.

Copy this into your memory layer's design review. Every field exists because a real failure needed it.

Field	Type or values	Purpose
id	string	Stable identifier. Never reused, even after deletion.
subject	{ type: user | project | client | org, id }	Who or what the fact is about.
key	string, e.g. expense.project_code	What the fact is.
value	any, nullable	The remembered value. Null for policy pointers and tombstones.
authority	preference | fact | policy	How much weight it carries.
source.type	user_stated | system_record | agent_inferred | user_correction	Source Who said it.
source.evidence	{ kind: message | record | trace, ref }	Evidence Where you can check it.
source.observations	integer ≥ 1	How many observations support it. The key number for inferred facts.
source.captured_at, source.captured_by	datetime, agent@version	Provenance. A bad batch of memories can be traced to the build that wrote it.
source.source_of_truth	URI, required when authority = policy	Where the real value lives.
scope.level	instance | task | session | project | client | user | org	Scope How wide it applies.
scope.bindings	object, e.g. { "trip_id": "TRIP-PUNE-0304" }	The concrete context it is bound to.
scope.valid_from, scope.valid_until	date, nullable	Expiry When it starts and stops being true.

Field	Type or values	Purpose
scope.revalidate_on	array of event types, e.g. ["project_closed"]	Expiry Events that force a fresh check.
correction.route	string, required	Correction route How a human fixes it.
correction.supersedes, correction.superseded_by	ids	Correction history.
correction.broadened	boolean	True only if the user explicitly confirmed a wider scope.
lineage.derived_from	array of ids	What this fact was inferred from.
lifecycle.status	active | superseded | expired | deleted	Whether it can be served.
lifecycle.sensitivity	none | personal | sensitive	Retention rules.
lifecycle.retain_value_after_expiry	boolean, default false	Whether an expired record keeps its value.
usage.last_used_at, usage.use_count, usage.last_confirmed_at	datetime, integer, datetime	Staleness signals.

Seven invariants

I1	An inferred record cannot have a scope wider than its evidence supports, unless the user confirmed it.
I2	A policy record stores a pointer, never a value. Policy is fetched from its source of truth at use time.
I3	Every record names a correction route.
I4	A user correction defaults to the scope of the instance corrected. Broadening it needs broadened: true plus evidence that the user confirmed the wider scope.
I5	Superseded, expired and deleted records are never served.
I6	A sensitive record must have a valid_until date.
I7	Deleting a record invalidates every record whose lineage.derived_from includes it. The tombstone keeps id, key, deletion time and reason. The value is removed.

I2, I3, I4, I6 and the tombstone shape are encoded in the JSON Schema with if/then. I1, I5 and the cascade in I7 are rules across records, so the schema cannot hold them; the reference store in the kit enforces them.

The example record

A fact Priya stated, bound to one trip. It answers all three questions: msg_8812 is the evidence, trip TRIP-PUNE-0304 is the scope, and the override on the expense form is the correction route.

```
{
  "id": "mem_01",
  "subject": {
    "type": "user",
    "id": "priya"
  },
  "key": "expense.project_code",
  "value": "ATL-2291",
  "authority": "fact",
  "source": {
    "type": "user_stated",
    "evidence": {
      "kind": "message",
      "ref": "msg_8812"
    },
    "observations": 1,
    "captured_at": "2026-03-04T09:12:00Z",
    "captured_by": "expense-agent@1.4.0"
  },
  "scope": {
    "level": "task",
    "bindings": {
      "trip_id": "TRIP-PUNE-0304"
    },
    "valid_from": "2026-03-04",
    "valid_until": null,
    "revalidate_on": [
      "trip_closed",
      "project_closed"
    ]
  },
  "correction": {
    "route": "inline_override_on_expense_form",
    "supersedes": [],
    "superseded_by": null,
    "broadened": false
  },
  "lineage": {
    "derived_from": []
  },
  "lifecycle": {
    "status": "active",
    "sensitivity": "none",
    "retain_value_after_expiry": false
  },
  "usage": {
    "last_used_at": null,
    "use_count": 0,
    "last_confirmed_at": "2026-03-04T09:12:00Z"
  }
}
```


Download the full JSON Schema (draft 2020-12, 11 top-level fields, invariants encoded with if/then). It is also in the ZIP, with four example records and the harness that validates them.

02 12 audit questions

Any red flag is a design task.

Run these against one remembered fact in your system. Any red flag is a design task.

Source

01 **Can you point to the exact evidence for this fact: a message, a record, or a trace?**

Red flag "The agent just knows."

02 **Is it stored as user-stated, system record, or inferred, and does the agent behave differently for each?**

Red flag All three reach the prompt looking identical.

03 **If inferred, how many observations support it, and would the user agree with the generalisation?**

Red flag One observation became a preference.

04 **Is anything policy-like (limits, approvals, permissions) being served from memory instead of its source of truth?**

Red flag A cached policy value.

Scope

05 **What is the narrowest context this fact is true for, and is that what you store?**

Red flag Everything is stored at user level.

06 **When does it stop being true: a date, an event, or never?**

Red flag "Never" for anything tied to a project, role, price or person.

07 **Which event forces revalidation, and is that event actually wired to the memory store?**

Red flag Expiry exists only in a design doc.

08 **Can the agent tell a user why it applied this memory here?**

Red flag No trace from output back to a memory record.

Correction

09 **How does a user correct it, both in the moment and later?**

Red flag Only an engineer editing a database can.

10 **When corrected, what exactly changes (this instance, this scope, future behaviour), and who decided?**

Red flag "The agent learns from it."

11 **After deletion, does anything derived from it survive, such as summaries, embeddings or inferred preferences?**

Red flag Nobody knows.

12 **What sensitive content does your correction history retain, and do you need it?**

Red flag Full values kept forever "for audit".

03 7 failure tests

Each one is a way memory hurts a real user.

Each test is one way memory hurts real users. Run them against the naive store and watch all seven fail, then against the reference store, then against your own.

#	Test	Given	When	Then (pass criteria)
1	Scope bleed `test_scope_bleed`	ATL-2291 remembered at task scope, bound to trip_id T1.	Recall expense.project_code for trip_id T2, a different client.	Decision is not apply.
2	Stale fact `test_stale_fact`	A project-scoped code with revalidate_on: ["project_closed"].	Emit project_closed, then recall in that project.	Decision is none. The record's status is expired.
3	Inferred as explicit `test_inferred_as_explicit`	An agent_inferred fare class "economy" with observations: 1.	Recall.	Decision is ask. source_type is agent_inferred.
4	Correction bleed `test_correction_bleed`	The user corrects the code to MER-0417 on EXP-5521: merchant Olive Grove, client attendees.	Recall for EXP-5530: same merchant, internal attendees only.	MER-0417 is not applied to EXP-5530. Recall for EXP-5521 still returns MER-0417 with apply.

#	Test	Given	When	Then (pass criteria)
5	Policy shadowing `test_policy_shadowing`	Memory tries to hold the meal limit 75 as policy. The policy source now returns 60.	Recall the meal limit.	Value is 60. Decision is revalidate. No policy value was persisted.
6	Zombie memory `test_zombie_memory`	Record X (default cost centre) and record Y (approver) with derived_from: [X].	Forget X, then recall Y's key.	Decision is none. X's tombstone carries no value.
7	Unexplainable recall `test_unexplainable_recall`	Any active, user-stated record in scope.	Recall returns apply.	The result carries record_id, source_type and evidence_ref. explain() returns source, evidence, scope and correction route.

Run them

```
pip install -e .
pytest                  # reference store: 7 passed
pytest --store=naive    # naive store: 7 failed (on purpose)
```

The harness is Python 3.11 or newer, with pytest and jsonschema as its only dependencies. Its README, in the ZIP, shows how to write a six-method adapter for your own memory layer and run the same seven tests against it. Download the ZIP.

04 Decision table

Remember, Revalidate, Ask, Forget.

`Remember`	apply it, with a visible trace to the record
`Revalidate`	check the source of truth before using it
`Ask`	confirm with the user before using it
`Forget`	expire or delete it (tombstone only)

"Consequential" means the action moves money, grants access, contacts someone outside the organisation, or is hard to undo.

Situation	Outcome
User-stated, in scope, not expired, low-stakes action	`Remember`
User-stated, in scope, consequential action	`Ask`
Came from a system record	`Revalidate`
Policy, limit or permission	`Revalidate` every time; never store the value
Inferred from a single observation	`Ask`
Inferred, repeated, and confirmed by the user	`Remember` within the confirmed scope
Any fact used outside its bound context (new trip, project, client)	`Ask`
A user correction	`Remember` at the corrected instance's scope; Ask before broadening
Past valid_until, or a revalidation event fired	`Forget`
User asked to delete	`Forget` including everything derived from it
Sensitive and no longer needed for the task	`Forget` the value; keep a minimal audit entry

Closing

Memory is a set of promises

Memory isn't a feature you add to an agent. It is a set of promises about what the agent may believe, where, and for how long. This kit turns those promises into things you can review and test.

This is one episode of how we teach agentic system design to senior engineering leaders and architects at The Living Craft.