Contents

The six kinds of memory mnueron stores

A field guide to what's in your memory store, who reads it, and how it gets used.

TL;DR — the elevator pitch

mnueron is a persistent memory layer for AI agents. It stores six different *kinds* of memory, because human memory isn't one thing either. You don't remember "yesterday's coffee with Maya" the same way you remember "Maya works at Stripe" or "the Slack-to-Linear sync runbook." We store them differently too, because they're used differently.

Here are the six, plain-English:

#	Type	What it is	Closest human analogue
1	**Episodic**	Raw conversation chunks, time-ordered	"I remember that call we had"
2	**Semantic**	Standalone facts you've learned	"I know that the capital of France is Paris"
3	**Entity**	Canonical people, projects, places, decisions	Recognising Maya's name and knowing it's the same Maya you met last month
4	**Relational**	Edges between entities (Maya *works at* Stripe)	The mental map of who-knows-whom
5	**Procedural**	Step-by-step runbooks for how-to tasks	"Here's how I deploy to staging"
6	**Consolidation**	Proposed cleanups (this is a duplicate of that)	The "wait — didn't I write this down before?" feeling

The dashboard surfaces each of these on a dedicated page. The rest of this doc is one section per type — what it is, why it exists, what it looks like in the database, and a real example you can probably find in your own store.

---

1 · Episodic memory · the chat log

Non-technical

This is the raw stuff — the conversation as it actually happened. When you save a Claude or ChatGPT thread (via the Chrome extension or `mnueron-import`), every turn becomes one episodic memory. They're time-stamped, in order, and grouped by the conversation they came from.

Technical

Lives in the `memories` table. Each row carries:

- `content` — the message body
- `namespace` — the bucket the memory lives in (`claude-web`, `slack-imports`, etc.)
- `source` + `source_ref` — *where it came from* (e.g. `claude-export`) and a stable id that lets us group chunks back into a thread
- `metadata.role` — `user` / `assistant` / `system`
- `metadata.parent_ref` — the conversation id, so chunks from the same chat cluster
- `metadata.chunk_index` + `chunk_count` — turn ordering inside a thread
- `created_at`, `updated_at` — bi-temporal anchors

These are *NOT* deduped or rewritten — they're the corpus. Everything else in this doc is *derived* from them.

Real example

```
content:        "what's the latest on the Stripe-to-Mercury migration?"
namespace:      claude-web
source:         claude-export
source_ref:     conv_2025-02-14_abc123
metadata.role:  user
metadata.parent_ref: conv_2025-02-14_abc123
metadata.chunk_index: 14
```

How it's used

- The dashboard list shows one row per *thread* (clustered by `parent_ref`).
- Search hits this table via Postgres full-text search.
- The **Context builder** lets you cherry-pick episodic chunks and paste them into a new conversation.
- Future: the foundation for the **recall index** — every time an agent pulls one of these into context, we'll log it.

---

2 · Semantic memory · standalone facts

Non-technical

If episodic memory is "what happened," semantic memory is "what's true." It's the facts you can pull out of a conversation and reuse without needing the conversation itself. "Maya's parents live in Glasgow" is a semantic memory — you don't need to remember the dinner where she told you to use it.

Technical

Also lives in `memories`, but distinguished by:

- `source: "fact-extraction"`
- `tags: ["extracted-fact"]`
- `metadata.derived_from` — the parent episodic memory id
- `metadata.category` — `personal_info`, `preference`, `commitment`, `claim`, etc.

Created automatically by a background fact-extraction pass after each episodic save (paid tier; free tier can opt in via BYOK).

Real example

```
content:            "Maya prefers Mercury over Stripe for personal accounts because of the 1099 expo
source:                  fact-extraction
tags:                    [extracted-fact, preference]
metadata.derived_from:   <id of the original chat memory>
metadata.category:       preference
```

How it's used

- Faster recall — the agent doesn't have to re-read the whole conversation.
- Cleaner search — when you ask "what does Maya think about Mercury?", semantic memories score higher than the raw chat.
- The basis for **entity extraction** — facts are where canonical names get pulled out.

---

3 · Entity memory · canonical things

Non-technical

People, projects, places, technologies, decisions. The proper nouns. mnueron resolves "Maya," "Maya P.," and "maya@stripe.com" to one canonical entity so the rest of your memory store can point at it instead of duplicating the name everywhere.

Technical

Lives in the `entities` table (added in migration `014_p2_entities.sql`). Each row carries:

- `id` — UUID
- `display_name` — the canonical form
- `entity_type` — `person`, `organization`, `project`, `technology`, `place`, `decision`
- `aliases` — every surface form we've ever seen for this entity
- `mention_count` — how many times it's been referenced
- `first_seen_at`, `last_seen_at` — when the entity entered and was last touched in your store

A second table, `memory_entities`, joins each entity to the memories that mention it. So given the entity "Stripe" you can find every memory that talks about Stripe, with the actual surface form ("stripe.com," "Stripe Inc.," "the Stripe team") preserved.

Real example

```
display_name:    Maya Patel
entity_type:     person
```

```
aliases:         [Maya, Maya P., maya@stripe.com, Maya Patel]
mention_count:   47
first_seen_at:   2024-09-12
last_seen_at:    2026-05-21
```

How it's used

- **/dashboard/entities** lists every canonical entity, sortable, searchable.
- Click into one and you see *every memory* that mentions it, *every alias* we've matched, and the *graph neighbourhood* it sits in.
- Per-entity **visibility** controls (private / team / public) make this the unit of sharing.
- Entity extraction is **paid-tier server-key** by default; free-tier users can BYOK their own Claude or OpenAI key.

---

4 · Relational memory · the graph

Non-technical

Once you have canonical entities, you can store *relationships between them*. "Maya works at Stripe." "The auth-rewrite project depends on Postgres 16." "The Mercury migration was approved by Jordan."

Each of those is one edge in a knowledge graph. Together they form the map of your world.

Technical

Lives in the `relations` table (migration `029_relations_and_consolidation.sql`). Each row:

- `from_entity_id` → `to_entity_id`
- `predicate` — `works_at`, `depends_on`, `approved_by`, etc.
- `confidence` — 0..1, from the LLM that extracted it
- `valid_from`, `valid_to` — **bi-temporal** validity window. "Maya works at Stripe" might have `valid_from: 2024-06-01` and `valid_to: null` (still true). When she switches jobs, we close the window with `valid_to` instead of deleting the edge.
- `memory_id` — the episodic source that produced this edge

Real example

```
from_entity_id:  <Maya's id>
predicate:       works_at
to_entity_id:    <Stripe's id>
confidence:      0.94
valid_from:      2024-06-01
valid_to:        null            ← still true today
memory_id:       <id of the chat where she mentioned it>
```

How it's used

- **/dashboard/graph** renders this as a force-directed graph. Click a node to re-anchor.
- The date slider on that page is **point-in-time querying**: "show me what was true on 2024-12-01." Closed-window edges drop out; open ones stay.
- Down the road, relations feed agent reasoning — when an agent needs "people at Stripe," it walks edges, not full-text.

---

5 · Procedural memory · how-to runbooks

Non-technical

This is the kind of memory that says *"here's how to do X."* "Here's how I deploy to Vercel." "Here's how I onboard a new contractor." Step-by-step, reusable, tested.

This is the type Mem0 doesn't have — it's the leapfrog feature for mnueron.

Technical

Stored as memories with `source: "procedural"` and a structured `metadata.steps` array. Each step has a description and (optionally) a verification check.

Today this lives in the local CLI (`mnueron procedural`); the hosted version is on the roadmap. The local schema:

- `title` — short label ("Deploy Elevizio to Lightsail")
- `trigger_phrase` — what someone might say that should retrieve this ("deploy to lightsail," "ship api")
- `steps` — ordered array of `{ description, command?, check? }`
- `last_used_at`, `success_count`, `failure_count` — track how reliable the runbook is

Real example

```
title:           Deploy Elevizio API to Lightsail
trigger_phrase:  ship api
steps:
  1. ssh ubuntu@<lightsail-ip>
  2. cd /var/www/elevizio-api
  3. git pull && dotnet publish -c Release
  4. sudo systemctl restart elevizio-api
  5. curl https://api.elevizio.com/health   ← check
success_count:   23
failure_count:   1
```

How it's used

- When an agent gets a request that matches a `trigger_phrase`, it pulls the runbook into context — no fuzzy reasoning about "how does deploy work for this user."
- The success/failure counter lets you spot runbooks that have drifted (high failure rate = the steps don't match reality anymore).
- Coming to hosted as `/dashboard/procedural` in a follow-up.

---

6 · Consolidation memory · "this is a duplicate"

Non-technical

After enough conversations, you end up saving the same fact more than once. "Maya prefers Mercury." "Maya likes Mercury better than Stripe." Same fact, different words.

Consolidation memory is *memory about memory* — proposals that say "row #4827 and row #5912 might be the same thing, want to merge them?" You approve or reject; over time the store gets cleaner without you needing to audit every save.

Technical

Lives in `consolidation_proposals` (migration `029`). Each row:

- `kind` — `duplicate`, `contradiction`, or `stale`
- `memory_a_id`, `memory_b_id` — the two memories under suspicion
- `score` — similarity (trigram-based today, embedding-based later)
- `status` — `pending`, `approved`, `rejected`
- `proposed_at`, `reviewed_at`, `reviewed_by`

A detection scan walks recent memories and uses Postgres `pg_trgm` similarity to surface candidates. Above-threshold matches get inserted; the user reviews them in **/dashboard/consolidate**.

Real example

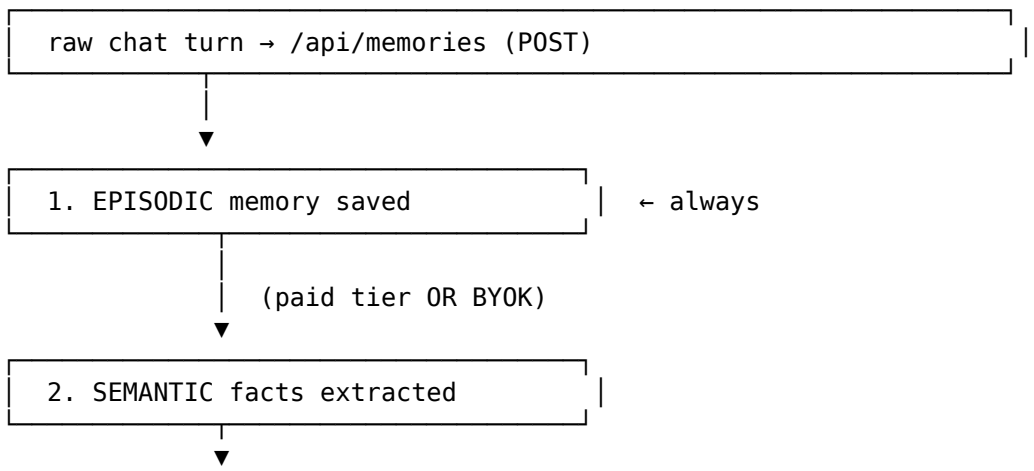
```
kind:           duplicate
memory_a_id:    <Maya prefers Mercury, saved Feb 14>
memory_b_id:    <Maya likes Mercury better, saved Apr 3>
score:          0.94
status:         pending
proposed_at:    2026-05-20
```

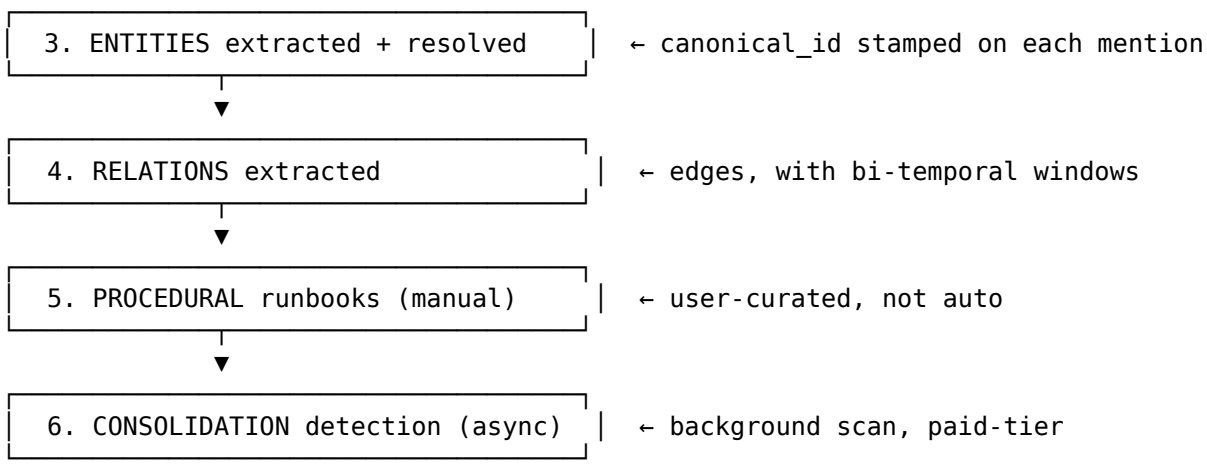
How it's used

- The Consolidate page is the review queue. Approve = mark for merge (phase 5b ships the actual merge); reject = keep both, never re-flag.
- Approved proposals teach the detector — over time it learns which phrasings are "the same thing."
- Detection is **paid-tier** (it costs Postgres CPU + LLM calls); review is free for everyone.

How the six fit together

Picture a pipeline running every time you save a chat:





Each layer is *derived* from the one above it, and each layer is *queryable* on its own. The dashboard has one page per layer — and one Analytics page that summarises across all of them.

What the dashboard surfaces

Page	Shows	Backed by
/dashboard (Memories)	Episodic chat list, paginated, freshness-dotted, searchable	`memories`
/dashboard/analytics	KPIs, 12-month volume chart, top-entities leaderboard	`/api/stats/*`
/dashboard/entities	Canonical entity browser, type filters	`entities`
/dashboard/entities/[id]	Per-entity detail: aliases, memories, relations, visibility controls	`entities` + `memory_entities` + `relations`
/dashboard/graph	Force-directed knowledge graph with bi-temporal slider	`relations` + `entities`
/dashboard/consolidate	Duplicate review queue, run-detection button	`consolidation_proposals`
/dashboard/context-builder	Cherry-pick + format memories for paste into ChatGPT/Claude	`memories`

What's *not* tracked yet (and where we're going)

mnueron is opinionated about what to add when. Today we **do not** track:

- **Recall counts** — how many times a given memory has been pulled into context. The infrastructure (`/api/recall` + audit table) is queued.

- **Decay scores** — a column exists (`memories.decay_score`) but no logic maintains it. Today the dashboard derives a "freshness" bucket from age only.
- **Embedding similarity** — search is BM25 (Postgres FTS) today; we'll add a vector column once a hosted embedder is wired.
- **Cross-org entity dedup** — every org has its own canonical "Maya." That's correct for privacy; it's not currently a roadmap reversal.

When recall tracking ships, the **Analytics** page will gain a "most-recalled memories" leaderboard and the list view will gain a "popularity" sort. The list-row freshness dot will mix recall-recency into the bucket — so a heavily used "old" memory still reads as fresh.

Quick reference for builders

If you're integrating with mnueron and want to surface the right type:

- **"Find what we said about X"** → search episodic (`/api/memories?q=...`)
- **"What do I know about X"** → load entity (`/api/entities/<id>`) + its relations
- **"What's the state of X right now"** → walk relations with `?asOf=<today>`
- **"How do I do X"** → look for procedural memories with matching `trigger_phrase`
- **"What memories are stale / duplicated"** → poll `/api/consolidate?status=pending`
- **"Show me my month"** → `/api/stats/timeseries?bucket=month&months=12`

Generated 2026-05-22 · mnueron memory product · randy@mnueron.com