EIDARA

A Compiled Memory System for AI Agents

Author: Javier Rotllant Miras

Version: 1.0 — May 2026

License: MIT

Repository: github.com/jrotllant/eidara

Web: eidara.dev

Open-source persistent memory for AI agents. No vector database. No cloud. No vendor lock-in.

Abstract

EIDARA is a file-based persistent memory system for AI agents. Unlike existing approaches that rely on vector databases, graph structures, or cloud APIs, EIDARA uses a **compiler-based architecture**: AI agents write structured markdown files to a source directory, and a compiler transforms them into a single optimized document (BRAIN.md) that any AI can read. The system is governed by a 15-rule constitution (W1-W15) that standardizes how any AI—Claude, GPT, DeepSeek, or others—interacts with the memory. It includes self-healing mechanisms, distributed content governance via consensus flags (3/3 votes), SHA256 tamper detection, git-based version control, and two reading modes (Standard and Light) to accommodate models with different capabilities. The system was stress-tested across 4 different AI models with 52 autonomous tests each, achieving an average score of 92.6% with zero system failures.

1. The Problem

1.1 AI Amnesia

Every AI conversation starts from zero. Large language models are stateless by design—they process a context window and produce a response, but nothing persists between sessions. This creates a fundamental limitation for any professional who uses AI as a daily tool:

- **Context repetition:** Users paste the same project information into every new conversation.
- **Session boundaries:** Long conversations exhaust token limits. Starting a new session means losing all accumulated context.
- **Cross-platform gaps:** Information discussed in Claude is unavailable to DeepSeek, and vice versa.
- **Context drift:** Multiple versions of the same context file accumulate, and it becomes unclear which is current.

1.2 Existing Approaches

Several approaches have emerged to address AI memory:

Mem0 is a popular open-source memory project that uses a hybrid vector + graph architecture. It extracts facts from conversations, stores them in a searchable format, and retrieves relevant context at query time. It requires a cloud API or self-hosted infrastructure with a vector database.

Letta / MemGPT takes an OS-inspired tiered memory approach. Agents actively manage their own memory via built-in tools, with core/recall/archival storage tiers. It requires PostgreSQL and a runtime server.

Zep builds temporal knowledge graphs using their open-source Graphiti library. Every memory is stored with time anchoring, tracking how facts change over time. It requires a knowledge graph infrastructure.

Andrej Karpathy's LLM-Wiki concept describes markdown files that AI agents maintain as persistent memory. It introduces the concept of AI-maintained wikis but, as a concept rather than a reference implementation, does not include a compiler, validation, governance, or multi-agent consensus mechanisms.

Common limitations across existing approaches:

- Require infrastructure (vector DB, PostgreSQL, cloud API)
- Are platform-specific (API-bound or framework-bound)
- Lack content governance (any AI can write without oversight)
- Store raw facts rather than compiling structured knowledge

2. Design Principles

EIDARA was built on three non-negotiable principles that emerged from multiple architecture iterations:

2.1 Rules Control, Not Roles

Any AI can read and write if it follows the protocol. There are no special agents with special permissions. Security lies in the rules of entry (a constitution with two gates), not in who has access.

> *"In an efficient company you don't tell people 'only these three can enter the system'. You tell them 'to enter the system you do it like this' and you put controls. Controls create bottlenecks when they're about WHO, not about HOW."*

2.2 Errors Self-Heal

~10% imprecision is acceptable in exchange for scalability. If an AI writes something incorrect, the next AI that reads it and knows it's wrong will fix it. The system cleans itself through use.

2.3 The System Scales Infinitely

The structure doesn't change whether there are 25 memories or 2,000. No folders to reorganize, no hierarchies to break. Just new memories, a growing index, and connections between them.

3. Architecture

3.1 Two-Gate Design

EIDARA operates on a two-gate architecture:

```
DARA.md (Constitution)
    |
    +-- Gate 1: VAULT/ (Write)
    |   +-- NEURONS/     -> Project and context knowledge
    |   +-- ENABLERS/    -> Agents, tools, credentials
    |   +-- INBOX/       -> Feedback channel
    |   +-- BACKUP/      -> Timestamped snapshots
    |
    +-- Gate 2: LIBRARY/ (Read)
        +-- BRAIN.md     -> Compiled output, single file
```

Gate 1 — VAULT (Write): Any AI writes information as individual markdown files. Each file represents one topic: a project, a person, a tool, or a configuration. Files follow a compressed encoding format designed for AI consumption.

Gate 2 — LIBRARY (Read): Any AI reads a single compiled file (BRAIN.md). This file is auto-generated by the compiler and contains all active entries compressed, validated, and indexed. It fits in a modern context window — typically ~3,500 tokens for an active VAULT with 20+ entries.

Why two gates: Writers deal with individual, well-structured source files. Readers get a single optimized file. The compiler ensures consistency between the two.

3.2 File Categories

NEURONS: Project and context knowledge. Each file is one topic. Named with a domain prefix (e.g., project-alpha.md, profile-technical.md).

ENABLERS: Executable agents, tools, and credentials. Agents start with a summary: line and contain full execution protocols. Any AI that reads an enabler file can execute it.

INBOX: Feedback channel for structural issues. Any AI can write here; only the system architect processes it.

3.3 Encoding Language

All VAULT files use a compressed notation designed to minimize token usage while preserving semantic content:

Element	Format	Example
Dates	YYYY-MM-DD	2026-04-24
Money (thousands)	€NNK	€15K
Status/Relevance	[A/H], [C/L], [P/M]	[A/H] = Active + High

People	First.Last(role)	Alice.M(Marketing)
References	→refs:	→refs: project-alpha, profile-tech
Tags	→tags:	→tags: #saas #typescript
Consensus flags	→flag:	→flag: "reason" | 2/3 | voters
Word count	~NNNw	~300w

3.4 Two Reading Modes

Standard Mode: AI reads full BRAIN.md and opens VAULT files as needed. For Claude, GPT with plugins, and models with reliable tool access.

Light Mode: AI reads only BRAIN.md (INDEX + →brain: summaries). For DeepSeek, ChatGPT web, and models without filesystem access. Same data, smaller context. The protocol adapts to the model's capabilities—not the other way around.

4. The Constitution (W1-W15)

The constitution is the core innovation. It defines 15 writing rules that govern every AI interaction with the system:

Rule	Description
W1	Check first—scan BRAIN.md INDEX before creating a new neuron
W1(b)	External-session writes—read DARA.md fully before writing from any external context
W2	Fix errors—self-healing. Spot it, fix it, log it
W3(a)	Never delete files without explicit owner instruction
W3(b)	5 protected files (Architect-only): DARA.md, compile.py, validators.py, watcher.py, agent-architect.md
W4	Update existing when: new info, data changed, error detected
W5	Create new only when topic doesn't fit anywhere existing
W6	After any write: regenerate BRAIN.md (auto-compiled by watcher in ~8s)
W7	Log everything in changelog.md—append only
W8	File naming: kebab-case, always
W9	Session closing trigger—ask "anything DARA should remember?"
W10	Live memory detection—flag significant info as it arises
W11	Write lean, review everything—consensus flags for ambiguous content
W12	Verify writes via shell after edits (avoids tool truncation bugs)
W13	Structure for BRAIN efficiency—large neurons start with →brain: summary
W14	Enabler summaries—every agent file starts with summary: line
W15	Platform-aware reading—Standard vs Light mode

4.1 Consensus Flags (W11)

The consensus flag system is a distributed governance mechanism. When an AI is unsure whether content still adds value, it adds a flag. When another AI independently agrees, it adds its vote. At 3/3 votes, the next AI removes the flagged content.

- Each AI votes only once per flag
- No voting on own flags
- Disagreement resets the flag
- Flags older than 30 days with <3 votes are removed (consensus didn't form, content stays)

4.2 Signal System

Don't ask the user. Fix or flag:

1. **If certain it's wrong:** Fix directly (W2). Log the fix.
2. **If uncertain:** Leave a feedback file in INBOX with evidence.

3. **If the entry has pending feedback:** Treat with caution.

4.3 The Librarian — Quality Guardian

The Librarian is the system's autonomic maintenance agent. It is itself an executable document (agent-librarian.md) that any AI can read and execute. Its job is enforcement, not design: the Architect designs the rules, the Librarian enforces quality within those rules.

Cadence: Every 3 days. The compiler tracks the last run and embeds an OVERDUE alert in BRAIN.md when more than 3 days have passed.

7-phase execution protocol: Orientation, System backup, Health diagnostics, Separate durable from dated, Execute corrections, System verification, Report.

4.4 The Architect — System Designer

The Architect is the only role authorized to modify the 5 W3(b) protected files. It requires an explicit activation phrase from the system owner (the 'Golden Door'). Without that explicit instruction, any AI reading agent-architect.md treats it as read-only documentation.

This dual gating (Golden Door + W3(b) checksums) prevents any AI from spontaneously deciding to modify the Constitution.

4.5 Self-Healing in Practice

The self-healing principle operates at three layers:

- **Layer 1 — In-flight by any AI (W2):** When any AI reads VAULT data and spots an error, it fixes it directly. No permission needed.
- **Layer 2 — At compile time:** The compiler auto-fixes broken refs, header mismatches, ASCII arrows, and null bytes.
- **Layer 3 — Across sessions via consensus (W11):** Ambiguous content is flagged, voted on, and removed at 3/3.

5. The Compiler (compile.py v3.1)

The compiler is the heart of the system. It transforms raw VAULT files into a single, optimized, verified BRAIN.md.

5.1 Pipeline

The compiler executes a 10-step pipeline:

1. **Backup:** Copy current BRAIN.md to timestamped snapshot
2. **Prune:** Remove backups older than 180 days; batch-prune when approaching limit
3. **Read:** Scan all NEURONS and ENABLERS markdown files
4. **Auto-purge:** Soft-delete archived entries ([C/L]) older than 60 days
5. **Deduplicate:** Name overlap + Jaccard content similarity (>0.60 threshold)
6. **Validate + auto-fix:** 16 validator functions covering refs, headers, encoding, monetary format, kebab-case, orphans, changelog, misplaced agents, token budget
7. **Check INBOX:** Count and report pending feedback
8. **Compile:** Generate BRAIN.md with active entries, archived entries (title only), PII stripping
9. **Delta report:** Compare per-entry sizes vs previous compile
10. **Git auto-commit:** Stage all changes and commit with descriptive message

5.2 Validators

16 pure functions with no global state: get_entry_date, fix_refs, fix_header, fix_ascii_arrows, check_creds, check_stale, dedup, content_similarity_check, check_kebab_case, check_orphan_files, check_changelog_format, check_monetary_format, check_misplaced_agents, check_size_regression, check_protected_files, check_brain_tamper.

5.3 Protection Mechanisms

• **SHA256 checksums** on 5 critical files

• **Tamper detection:** BRAIN.md has an embedded checksum; modifications detected on next compile

• **Warning header:** BRAIN.md starts with machine-readable warnings against direct editing

• **Backup rotation:** Timestamped snapshots, max 60, max 180 days

5.4 CLI

```
python compile.py               Full compilation
python compile.py --dry-run     Validate without writing
python compile.py --stats       Quick health check
python compile.py --validate-only  Validate structure only
python compile.py --no-git      Compile without git commit
```

5.5 Watcher (Auto-Compiler)

watcher.py monitors VAULT/NEURONS/ and VAULT/ENABLERS/ every 5 seconds. When a file change is detected, it waits 3 seconds (debounce) and runs compile.py. It runs silently on system startup.

5.6 Three-Layer Compression

A populated VAULT can hold 30K+ characters of source content. BRAIN.md keeps the compiled output under 4K tokens through three compression layers:

- **Layer 1 — →brain: summary lines (W13):** For neurons larger than 1500 chars, a one-line summary compresses ~7,000 chars to ~120 chars in BRAIN.md.
- **Layer 2 — Enabler one-liners (W14):** Every enabler renders as a compact name | summary line. A 10,000-char agent becomes ~150 chars.
- **Layer 3 — strip_noise() PII redaction:** IBAN, NIF/CIF, filesystem paths, UUIDs, and API tokens are replaced with placeholders.

Compression ratio: ~27K source chars → ~3.5K BRAIN tokens (approximately 7:1 for active content).

5.7 Health Monitoring

Every compile produces SYSTEM/health.json with metrics: active neurons/enablers, source/brain tokens, warnings, inbox count, backup count, and alerts when thresholds are crossed. Token budget alerts fire at 50K (INFO), 100K (WARNING), and 150K (CRITICAL) source tokens.

5.8 Recovery: Multi-Layer Defense

- **Layer 1 — Git:** Every successful compile auto-commits. git checkout restores any file instantly.
- **Layer 2 — System backups:** 3 most recent copies of each protected file in VAULT/BACKUP/SYSTEM/.
- **Layer 3 — BRAIN.md backups:** Timestamped snapshots before each compile. Up to 60, max 180 days.
- **Layer 4 — Manual reconstruction:** RECOVERY.md documents 6 emergency scenarios.

6. Agents as Documents

One of the most distinctive features of EIDARA is that **agents are documents, not programs**.

A file agent-librarian.md contains complete instructions for vault maintenance. Any AI that reads that file becomes the librarian—it doesn't matter if it's Claude, DeepSeek, or GPT. The intelligence is in the file, not in who executes it.

This eliminates platform dependency for configuring agents. To create a new agent, one writes a markdown file. To share an agent, one shares the file. To modify an agent, one edits the file.

This concept extends to sharing knowledge across teams. A neuron file can be dropped in shared Drive. A colleague opens it with their AI—any AI—and the information is instantly accessible. No install needed.

7. Testing

7.1 System Tests (Phase 1B)

4 different AI models each ran 52 autonomous tests across 7 evaluation blocks. Each model operated independently, with no human intervention.

Model	Platform	Score	%
Opus 4.6	Claude Cowork	670/700	95.7%
Sonnet 4.6	Claude Cowork	652/700	93.1%
DeepSeek V4	TypingMind (no shell)	650/700	92.9%
Opus 4.7	Claude Cowork	613/700	87.6%

Average: 92.6% · Zero system failures · Zero data corruption

The spread reflects interpretive choices on ambiguous edge cases. Variance across a 9-point band is itself evidence that the protocol, not the model, is what carries the system.

7.2 Unit Tests

103 pytest tests across 3 files, 100% pass rate:

- **test_validators.py:** 70 tests (597 lines), covering all 16 validator functions
- **test_compiler.py:** 33 tests (347 lines), covering checksums, noise stripping, backups, and the full pipeline
- **conftest.py:** Shared fixtures for testing

8. Design Origins

8.1 Three Iterations

v1 — The Org Chart Trap: Six specialized agents with a supervisor. Killed because task experts don't understand domain context. A 'mail agent' reading vendor emails doesn't know the vendor's history.

v2 — The Roles Trap: Domain agents with specific write permissions. Killed because it creates bottlenecks. Any AI might have the right information at the right time.

v3 — The Breakthrough: Eliminate all hierarchical folders. Design for the AI, not for the human. If the human never opens the folder, who are you organizing for?

8.2 The MECE Trap

Javier comes from strategy consulting—13 years total, 9 of them at Bain & Company—where MECE (Mutually Exclusive, Collectively Exhaustive) is the central dogma. The realization: MECE is a thinking tool, not a memory tool.

> *"That's how I think but not how I remember."*

9. Getting Started

9.1 Requirements

Required: Python 3.10+ and a folder on the local filesystem.

Recommended: Git (for automatic version history) and an AI assistant with filesystem access.

Not required: GitHub account, cloud sync, a specific AI platform, or any Python package beyond stdlib.

9.2 Installation

```
cd /path/to/your/EIDARA/folder
python3 --version                          # Check Python 3.10+
git init && git add -A && git commit -m "EIDARA v1.0 - initial install"
python3 SYSTEM/compile.py                  # First compile
python3 SYSTEM/watcher.py                  # Auto-watcher
```

9.3 Daily Workflow

1. Start a session with any AI. Give it LIBRARY/BRAIN.md.
2. Work normally — the AI now has full context across all your projects.
3. When something significant happens, the AI flags it (W10) or you say 'update DARA'.
4. The watcher recompiles BRAIN.md automatically within ~8 seconds.
5. The Librarian self-triggers every 3 days for maintenance.

6. The Architect activates only on explicit owner command.

10. Future Directions

- **Semantic search:** Finding information by meaning, not just by filename
- **MCP Server:** Connecting DARA to tools without native filesystem access
- **Conflict detection:** Flagging when two neurons cover the same topic differently
- **Web dashboard:** Visualizing system status, neurons, and enablers
- **Agent marketplace:** Community-built and shared ENABLERS
- **Optional cloud sync:** Team synchronization without compromising local-first

11. Conclusion

EIDARA demonstrates that persistent AI memory does not require vector databases, cloud infrastructure, or complex graph systems. A compiler-based approach with a clear constitution, self-healing mechanisms, and distributed governance can produce a system that is simpler, more transparent, and more portable than existing alternatives.

The system is production-ready, tested across multiple AI models, and available as open-source software.

References

- Mem0: github.com/mem0ai/mem0
- Letta (MemGPT): github.com/letta-ai/letta
- Zep / Graphiti: github.com/getzep/graphiti
- Andrej Karpathy (LLM-Wiki concept): proposed in public posts, no canonical repository

---

EIDARA v1.0 — MIT License — Created by Javier Rotllant Miras