Cortex

Four-Layer AI Cognitive Architecture

Author: Jonathan Domian
Version: 1.0
Date: April 2026
Status: Production (deployed on Claude Server, 10.0.0.99)

"AI agents forget between sessions, load too many tokens, and never learn from mistakes. Cortex fixes all three."

Table of Contents

1. Executive Summary

1.1 The Problem

AI coding agents like Claude Code are powerful within a single session. But between sessions, they forget everything. Every conversation starts from zero. The workarounds are crude: dump thousands of lines of markdown into the system prompt, hope the model reads all of it, and accept that the agent will never remember what went wrong last time.

Three specific failures repeat across every AI agent deployment:

- **Amnesia:** No cross-session memory. The agent cannot recall what it did yesterday, what decisions were made, or what context matters now.
- **Token bloat:** To compensate for amnesia, users load massive context files into every session. A typical setup burns 2,000+ lines and 70,000+ tokens before the first question is asked.
- **No learning:** When something breaks, the agent fixes it in the moment but never internalizes the lesson. The same mistake happens again next week. There is no feedback loop from error to permanent behavioral change.

1.2 The Solution

Cortex is a four-layer cognitive memory architecture for AI agents. It replaces the "dump everything into the prompt" approach with a structured system that mirrors how biological brains actually work: always-on identity, on-demand recall, searchable knowledge, and continuous learning from mistakes.

The result: 74% fewer tokens loaded per session, semantic search across all past context, a knowledge graph for temporal fact tracking, and an evolution layer that turns recurring errors into permanent rules.

1.3 The Brain Metaphor

Each layer of Cortex maps to a region of the human brain. This is not decoration. The metaphor drives the architecture, defines the boundaries between layers, and explains why each layer exists.

LAYER	NAME	BRAIN REGION	FUNCTION
L1	Identity	Prefrontal Cortex	Personality, executive function, rules, active priorities
L2	Recall	Hippocampus	Episodic memory, decisions, conversations, facts
L3	Knowledge	Neocortex	Learned knowledge, reference docs, external material
L4	Evolution	Cerebellum	Error correction, motor learning, habit formation
DMN	Default Mode Network	Default Mode Network	Background maintenance, consolidation, decay

2. The Four Layers

L1

Identity (Always Loaded)

L1 is the agent's sense of self. It contains the system prompts, personality definition, behavioral rules, active priorities, and configuration that must be present in every single session. Without L1, the agent is a blank slate. With L1, it knows who it is, who it serves, and what matters right now.

Brain Analogy: Prefrontal Cortex

The prefrontal cortex governs executive function, personality, decision-making, and social behavior. It is "always on" in a healthy brain. Damage to the prefrontal cortex changes who a person is, not what they know. L1 is the same: it defines the agent's identity, not its knowledge.

Sub-system: Mirror Neurons

Mirror neurons in the biological brain fire when observing others' actions and emotions, enabling social awareness and empathy. In Cortex, the Mirror Neuron sub-system reads user state from conversation cues. It detects frustration, urgency, confusion, and satisfaction from language patterns and adjusts the agent's behavior accordingly. If the user sends short, clipped messages after a failed deployment, the agent shifts to concise, action-oriented responses without being told.

Token Budget

L1 is constrained to approximately 750 lines of markdown. This is the only layer that burns tokens every session, so it must be lean. Anything that can be recalled on demand belongs in L2, not L1.

Implementation

Markdown files: `CLAUDE.md`, configuration files, priority files, and context documents. These are loaded by the AI agent framework (Claude Code) at session start. No database, no API, just flat files read into the prompt.

L2

Recall (On Demand, Semantic Search)

L2 is everything that has ever happened. Facts learned, conversations held, decisions made, solutions discovered, errors encountered. Unlike L1, L2 is not loaded into every session. It is searched on demand when the agent needs context it does not currently have.

Brain Analogy: Hippocampus

The hippocampus is the brain's memory formation and retrieval center. It encodes episodic memories (events that happened) and enables recall by association. You do not load every memory into consciousness at all times. You recall them when a cue triggers retrieval. L2 works identically: content is stored in vector space and retrieved when a query is semantically similar.

Sub-system: Amygdala (Emotional Weighting)

The biological amygdala tags memories with emotional significance. A traumatic event is remembered more vividly than a routine Tuesday. In Cortex, the Amygdala sub-system assigns importance tags to memories. A "critical" bug that bricked the server gets higher retrieval priority than a routine configuration change. Importance levels influence search ranking: when multiple results match a query, higher-importance memories surface first.

Technology: ChromaDB + Sentence Transformers

L2 uses ChromaDB as a persistent vector database with the `all-MiniLM-L6-v2` sentence transformer model for embeddings. Here is how the pipeline works:

1. **Ingestion:** Content (text from files, conversations, manual entries) is split into chunks of manageable size.
2. **Embedding:** Each chunk is passed through the all-MiniLM-L6-v2 model, which produces a 384-dimensional vector. This vector captures the semantic meaning of the text, not just keywords.
3. **Storage:** The vector is stored in ChromaDB alongside metadata: the wing (category), room (sub-category), source file, timestamp, and importance level.
4. **Query:** When the agent needs to recall something, the query text is embedded using the same model, producing another 384-dimensional vector.
5. **Retrieval:** ChromaDB compares the query vector against all stored vectors using cosine similarity. The top-k most similar results are returned with their relevance scores.

Implementation

MCP (Model Context Protocol) server exposing tools: `cortex_search`, `cortex_add`, `cortex_kg_query`. The agent calls these tools as naturally as it calls any other tool. No special syntax, no manual retrieval. The agent decides when it needs more context and searches for it.

L3

Knowledge (On Demand, RAG)

L3 is external knowledge: vendor documentation, API references, hardware manuals, protocol specifications. This is not memory of what happened. This is reference material the agent has been given access to but should not memorize wholesale.

Brain Analogy: Neocortex

The neocortex is where learned knowledge lives. A mechanic does not recite every torque spec from memory. They know the specs exist, know where to find them, and can retrieve the right one when working on a specific engine. L3 gives the agent the same capability: awareness that knowledge exists and the ability to retrieve the relevant piece on demand.

Technology

Currently, L3 shares the same ChromaDB instance as L2 but uses different "wings" (collections) to keep reference material separate from episodic memory. Queries can target L3 specifically when the agent recognizes it needs reference material rather than personal recall.

Future plans include a dedicated vector index optimized for large document corpora, enabling the ingestion of full vendor manuals (hundreds of pages) without polluting the recall layer.

Use Case

When the agent encounters a question like "What is the UniFi CLI command for blocking a client?", it searches L3 rather than relying on its training data. The answer comes from indexed documentation that may be more current, more specific, or more authoritative than general training knowledge.

L4

Evolution (Continuous Learning)

L4 is what makes Cortex different from every other AI memory system. Most architectures have some form of L1 (system prompt) and L3 (RAG). Some have L2 (vector recall). Nobody has L4: a systematic feedback loop that turns mistakes into permanent behavioral changes.

> **Why L4 is novel:** The AI industry has largely ignored the "learning from errors" problem. RAG systems retrieve knowledge. Memory systems store facts. But no mainstream architecture closes the loop from "this broke" to "this will never break again because the agent's behavior permanently changed." L4 does exactly that.

Brain Analogy: Cerebellum

The cerebellum handles motor learning and error correction. When you learn to ride a bicycle, each fall sends an error signal. The cerebellum adjusts motor patterns until falling stops. You do not consciously think about balance after learning. The behavior became automatic through repeated error correction. L4 does the same for an AI agent's operational patterns.

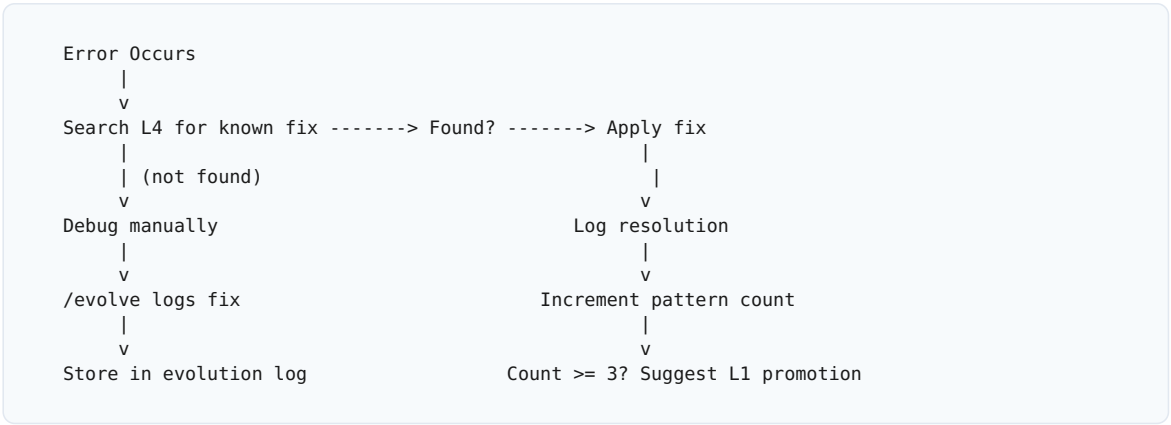
Sub-systems

SUB-SYSTEM	BRAIN REGION	FUNCTION
Basal Ganglia	Basal Ganglia	Habit formation. When a pattern repeats enough times, it becomes an automatic behavior rather than a conscious decision.
Reticular Activating System	RAS (Brainstem)	Attention filter. Determines what is worth paying attention to versus what is noise. Prevents the agent from drowning in low-priority information.
Thalamus	Thalamus	Smart router. Directs incoming information to the correct layer. A new fact goes to L2. A vendor doc goes to L3. An error pattern goes to L4.

The Feedback Loop

L4 operates through a structured feedback cycle:

1. **Error occurs:** Something breaks. A deployment fails, a command produces unexpected output, a configuration is wrong.
2. **Search existing fixes:** L4 first checks whether this error has been seen before. If a known solution exists, it applies it immediately.
3. **Debug if new:** If the error is novel, the agent debugs it normally, working through the problem with the user.
4. **/evolve logs the fix:** After resolution, the `/evolve` command captures the error, root cause, solution, and affected files into the evolution log.
5. **Pattern counting:** Each error type is tracked. The system counts occurrences across sessions.
6. **Promotion threshold:** After 3 occurrences of the same pattern, L4 suggests promoting the fix to an L1 permanent rule. This means the agent will proactively avoid the error in every future session, not just react to it.


Error Occurs
Search L4 for known fix
Found?
Apply fix
(not found)
Debug manually
Log resolution
/evolve logs fix
Increment pattern count
Store in evolution log
Count >= 3? Suggest L1 promotion

3. DMN: Default Mode Network

DMN

Background Maintenance System

The Default Mode Network is not a layer you query. It is a maintenance system that runs between sessions, keeping the four layers healthy, consolidated, and optimized. It burns zero tokens because it runs as pure Python and bash scripts via cron.

Brain Analogy

In neuroscience, the Default Mode Network activates when the brain is "at rest" but not truly idle. It consolidates memories, makes connections between disparate experiences, and performs maintenance on neural pathways. Sleep is when most memory consolidation happens. The Cortex DMN is the same: it runs when the agent is not in an active session, performing the housekeeping that keeps the memory system healthy.

Schedule

SCHEDULE	TASK	DESCRIPTION
Nightly (3:00 AM)	Incremental Remine	Scan for new or modified files since last run. Chunk, embed, and store new content in ChromaDB.
Nightly (3:00 AM)	Consolidation	Merge related memories, deduplicate near-identical entries, update metadata timestamps.
Nightly (3:00 AM)	Pattern Detection	Scan the evolution log for recurring error patterns. Flag patterns that have reached the promotion threshold.
Nightly (3:00 AM)	Connection Building	Identify links between memories in different wings. If a decision in L2 references a document in L3, create a knowledge graph edge.
Weekly (Sunday, 6:00 AM)	Decay Management	Reduce the relevance score of memories that have not been accessed in a configurable window. Prevents stale information from crowding out current context.
Weekly (Sunday, 6:00 AM)	Promotion Report	Generate a summary of patterns ready for L1 promotion, memories approaching decay threshold, and overall palace health statistics.
Weekly (Sunday, 6:00 AM)	Palace Health Check	Verify ChromaDB integrity, check SQLite knowledge graph consistency, report storage usage and index sizes.

Cortex Boundary Rule

> **Cortex stores, retrieves, learns, and maintains.** It does NOT call external APIs. Cortex is self-contained and open-sourceable. No cloud dependencies, no API keys required for core functionality. The DMN runs entirely on the local machine using Python, bash, and cron. External integrations (MCP tools, agent frameworks) sit outside the Cortex boundary.

4. Technology Stack

Cortex is deliberately minimal. The entire system runs on three core dependencies plus standard library tooling. There is no Kubernetes, no cloud, no GPU required.

TECHNOLOGY	ROLE	VERSION
Python 3.9+	Runtime for all Cortex components	3.9+
ChromaDB	Vector embeddings, semantic search	0.4+
SQLite	Knowledge graph (temporal triples)	3.x (stdlib)
sentence-transformers	Embedding model (all-MiniLM-L6-v2)	2.x
cron	DMN scheduling	System
Markdown	L1 personality files	N/A

ChromaDB: How It Works

ChromaDB is a persistent vector database designed for embedding storage and similarity search. It runs as an embedded library (no separate server process) and stores data to disk in a configurable directory.

The storage model uses **collections**, which Cortex maps to "wings" of the memory palace. Each wing contains documents (chunks of text), embeddings (384-dimensional vectors), and metadata (key-value pairs like source file, timestamp, importance level).

When content is added:

1. The text is split into chunks (typically 500-1000 characters, split at sentence boundaries).
2. Each chunk is passed through the all-MiniLM-L6-v2 model, producing a 384-dimensional float vector.
3. The vector and its metadata are inserted into the appropriate ChromaDB collection.
4. ChromaDB maintains an HNSW (Hierarchical Navigable Small World) index for fast approximate nearest neighbor search.

When content is queried:

1. The query text is embedded using the same model.
2. ChromaDB traverses the HNSW index to find the k nearest vectors by cosine similarity.
3. Results are returned with their original text, metadata, and distance scores.
4. Distance scores are converted to relevance percentages (1.0 minus normalized distance).

Knowledge Graph: How It Works

The knowledge graph uses SQLite to store subject-predicate-object triples with temporal validity windows. This enables queries that are time-aware, something vector search alone cannot do.

Each triple has four fields:

FIELD	TYPE	EXAMPLE
subject	Text	"Jonathan"
predicate	Text	"works_at"
object	Text	"KPMG"
valid_from	Datetime	2026-04-01
valid_to	Datetime (nullable)	NULL (still current)

This structure enables powerful temporal queries:

- "Where did Jonathan work on January 15, 2026?" returns "Integrity XD" (valid_from before Jan 15, valid_to after Jan 15 or NULL at that time).
- "Invalidate: Jonathan no longer works at Integrity XD" sets valid_to on the old triple and creates a new one for the current employer.
- "What changed in March 2026?" returns all triples with valid_from or valid_to timestamps in March.

Sentence Transformers: How It Works

The `all-MiniLM-L6-v2` model is a distilled version of BERT, specifically fine-tuned for producing semantically meaningful sentence embeddings. Key characteristics:

- **384-dimensional output:** Each text input produces a vector of 384 floating-point numbers that capture its semantic meaning.
- **Semantic similarity:** Texts with similar meanings produce vectors that are close together in 384-dimensional space, as measured by cosine similarity.
- **CPU performance:** Embedding a single sentence takes approximately 50ms on a modern CPU. No GPU required.
- **Fully local:** The model runs entirely on the local machine. No API calls, no cloud, no cost per query. The model weights are downloaded once (approximately 90MB) and cached locally.
- **Cross-lingual:** Trained on multilingual data, though Cortex primarily uses English.

5. Operations Classification

Cortex operations fall into four categories based on when and how they fire. Understanding this classification is essential for predicting token costs and system behavior.

#	OPERATION	TYPE	TRIGGER	TOKEN COST
1	Identity load	Passive	Session start	~750 lines
2	Auto-save (diary)	Passive	Session end hook	0 (background)
3	Error detection	Passive	Command failure	Minimal
4	Known-fix lookup	Passive	Error detected	Search result only
5	Importance tagging	Passive	Content ingestion	0 (background)
6	Smart routing (Thalamus)	Passive	Content arrives	0 (routing logic)
7	Mirror neuron read	Passive	User message	Minimal
8	Semantic search	On demand	Agent decides to search	Query + results
9	Knowledge graph query	On demand	Temporal or relational question	Query + results
10	/evolve (log fix)	On demand	User or agent invokes	Log entry
11	Incremental remine	Scheduled	Nightly cron (3 AM)	0
12	Consolidation	Scheduled	Nightly cron (3 AM)	0
13	Pattern detection	Scheduled	Nightly cron (3 AM)	0
14	Connection building	Scheduled	Nightly cron (3 AM)	0
15	Decay management	Scheduled	Weekly cron (Sunday 6 AM)	0

Summary by Type

TYPE	COUNT	TOKEN IMPACT
Passive (fire automatically)	7	Minimal to zero. Most are background operations that do not add to the prompt.
On demand (triggered by question or command)	3	Variable. Only fires when needed. Returns only relevant results.
Scheduled (cron, between sessions)	5	Zero. Pure Python/bash, no LLM tokens consumed.

Additionally, **remine** (re-indexing all files) can be triggered manually for a full palace rebuild, or runs incrementally as part of the nightly DMN schedule.

6. The /cortex Command Interface

All Cortex operations are accessible through slash commands within the agent session. These commands provide introspection, management, and control over the memory system.

COMMAND	DESCRIPTION
`/cortex`	Dashboard. Shows palace status: total memories, knowledge graph stats, last remine, health status.
`/cortex search <query>`	Semantic search across all wings. Returns top-k results with relevance scores.
`/cortex add <content>`	Manually store a memory in the appropriate wing.
`/cortex kg query <subject>`	Query the knowledge graph for all triples involving a subject.
`/cortex kg add`	Add a triple to the knowledge graph with optional temporal bounds.
`/cortex kg invalidate`	Set valid_to on an existing triple (fact no longer current).
`/cortex kg timeline <subject>`	Show chronological history of a subject across all predicates.
`/cortex status`	Detailed status of all wings, collections, and storage usage.
`/cortex remine`	Trigger a full re-index of all configured source directories.
`/cortex evolve`	Log an error, its root cause, and its fix into the evolution layer.
`/cortex evolve-review`	Weekly review of patterns. Shows counts, promotion candidates, and system health.
`/cortex diary`	Read or write to the agent diary (session summaries, reflections).
`/cortex backup`	Create a point-in-time backup of all palace data.
`/cortex rollback`	Restore palace data from the most recent backup.
`/cortex health`	Run integrity checks on ChromaDB and SQLite. Report any corruption.
`/cortex setup`	First-time setup wizard. Configures palace location, wings, and cron schedules.
`/cortex uninstall`	Full removal with user confirmation. Asks whether to delete palace data.

Dashboard Example

```
$ /cortex

  Cortex v1.0  |  Palace: ~/.mempalace/palace/
  ================================================

  Wings          Drawers    Last Remine
  -----          -------    -----------
  system         42         2026-04-10 03:00
  personal       18         2026-04-10 03:00
  projects       67         2026-04-10 03:00
  reference      31         2026-04-10 03:00

  Knowledge Graph
  ---------------
  Triples: 214  |  Active: 189  |  Expired: 25

  Evolution
  ---------
  Patterns tracked: 24  |  Ready for promotion: 2
  Last /evolve: 2026-04-09  (OWUI model visibility)

  Health: OK  |  Storage: 847 MB  |  DMN last run: 6h ago
```

7. Backup, Rollback, and Uninstall

Auto-Backup

Before any file modification (remine, consolidation, manual add), Cortex creates an automatic backup of the affected data. Backups are stored in the palace directory under `backups/` with ISO 8601 timestamps. A `manifest.json` file tracks all backups with their creation time, trigger event, and file checksums.

manifest.json

The manifest is the single source of truth for palace state. It records:

- Every file Cortex has created, modified, or deleted
- SHA-256 checksums for integrity verification
- Timestamps for each operation
- The Cortex version that performed the operation

/cortex rollback

Restores the palace to the most recent backup state. This is an atomic operation: either the full rollback succeeds or the palace remains unchanged. The rollback itself creates a backup of the current state first, so rollbacks are reversible.

/cortex uninstall

Full removal of Cortex from the system. The uninstall process follows an AUQ (Ask User to Confirm) pattern:

1. List all files and directories that will be removed.
2. Ask the user: "Delete palace data? This includes all memories, knowledge graph, and backups."
3. If confirmed, remove palace data directory.
4. Remove cron entries for DMN.
5. Remove MCP server registration.
6. Remove the Cortex Python package.
7. L1 markdown files (CLAUDE.md, etc.) are NOT deleted. They existed before Cortex and belong to the user.

Setup Safety

The `/cortex setup` command informs the user of every change it will make before making it. No files are created, no cron entries added, and no configurations modified until the user explicitly confirms. The setup wizard shows a preview of all planned changes and waits for approval.

8. Enterprise Mapping (KPMG / Azure)

Cortex was designed for a single-user, single-machine deployment. But the architecture maps cleanly to enterprise cloud services. The following table shows how each Cortex layer translates to Azure services for the KPMG AI Landing Zone project.

CORTEX LAYER	AZURE SERVICE	IMPLEMENTATION
L1: Identity	Azure OpenAI (system prompts) + Cosmos DB	System prompts stored in Cosmos DB per-agent. Loaded at session init via Azure OpenAI API. Agent identity configs versioned in Azure DevOps.
L2: Recall	Cosmos DB + Azure AI Search	Episodic memories in Cosmos DB with vector indexing. Azure AI Search provides hybrid search (vector + keyword). Per-user partitioning via partition keys.
L3: Knowledge	Azure AI Search + Blob Storage	Documents in Blob Storage. Azure AI Search indexes with chunking, embedding (Azure OpenAI ada-002), and hybrid retrieval. Skillsets for PDF/DOCX extraction.
L4: Evolution	Azure Functions + Cosmos DB	Error patterns stored in Cosmos DB. Azure Functions run the feedback loop: detect pattern, count occurrences, flag for promotion. Change feed triggers processing.
DMN	Azure Functions (Timer Triggers)	Timer-triggered functions replace cron. Consolidation, decay, and health checks run on schedule. Application Insights for monitoring.

Multi-User Support

The critical difference between personal Cortex and enterprise Cortex is multi-user partitioning. In the personal deployment, all memories belong to one user. In the enterprise deployment:

- **L1 (Identity):** Per-agent, not per-user. All users of a given agent share the same identity.
- **L2 (Recall):** Per-user partitioning. Each user's memories are isolated. User A cannot search User B's recall.
- **L3 (Knowledge):** Shared across users within a project. All team members search the same documentation index.
- **L4 (Evolution):** Shared within a team. One user's fix benefits all users on the same project.

Cost Model

The personal Cortex deployment has zero ongoing cost (local compute, local storage). The Azure deployment introduces costs for:

- Cosmos DB RU consumption (reads and writes to memory stores)
- Azure AI Search units (vector index hosting)
- Azure OpenAI tokens (embedding generation via ada-002)
- Azure Functions invocations (DMN scheduled tasks)
- Blob Storage (document corpus)

Estimated cost for a 10-user team with moderate activity: $200-400/month, dominated by Azure AI Search and Cosmos DB.

9. Open Source Roadmap

Current State: Wrapper over MemPalace

Cortex currently wraps the MemPalace project (github.com/milla-jovovich/mempalace), an open-source AI memory system by Milla Jovovich and Ben Sigman. MemPalace provides the ChromaDB vector store, knowledge graph, and MCP server foundation. Cortex adds the four-layer cognitive architecture, the evolution feedback loop, the DMN maintenance system, and the brain-region metaphor that structures everything.

This wrapper approach preserves upstream updates. When MemPalace releases improvements to its vector search or knowledge graph engine, Cortex benefits automatically.

Future: Fork as jdomian/cortex

The planned path to independence:

1. **Phase 1 (Current):** Cortex as a layer on top of MemPalace. All Cortex-specific logic in separate files. MemPalace installed as a dependency.
2. **Phase 2:** Fork MemPalace into `jdomian/cortex`. Rename all internal references. Preserve the core vector/graph engine but restructure around the four-layer architecture natively.
3. **Phase 3:** Publish to PyPI as `cortex-ai` (name verified available). Enable one-command installation and setup.

Installation Target

```
pip install cortex-ai
cortex init        # Initialize palace directory and configuration
cortex setup       # Interactive setup wizard (sources, wings, cron)
cortex status      # Verify installation
```

The goal is a five-minute path from zero to working cognitive architecture for any AI agent that supports MCP (Model Context Protocol) or equivalent tool-calling interfaces.

License

Cortex will be released under the same license as MemPalace (MIT), ensuring it remains freely usable in both personal and commercial contexts.

10. Comparison: Before and After

DIMENSION	BEFORE CORTEX	AFTER CORTEX
Lines loaded per session	2,965 lines (CLAUDE.md + all @ references)	~757 lines (L1 only)
Token reduction	~70,000 tokens/session	~18,000 tokens/session (74% reduction)
Cross-session recall	None. Every session starts from zero.	Full semantic search across all past context via L2.
Semantic search	Not available. User must remember which file contains the answer.	384-dimensional vector search. Query by meaning, not filename.
Knowledge graph	Not available. Temporal facts scattered across markdown files.	SQLite knowledge graph with temporal validity windows. "What was true on date X?"
Learning from errors	Manual. User must remember to update known-solutions.md. Same errors recur.	L4 evolution layer. Errors logged, patterns counted, fixes promoted to permanent rules after 3 occurrences.
Background maintenance	None. All maintenance is manual and in-session.	DMN runs nightly and weekly via cron. Zero token cost. Consolidation, decay, health checks.
Context window usage	~35% consumed by static context before first question.	~9% consumed by L1. Remaining 91% available for actual work.
Conversations per context window	~3 (200k context, 70k consumed by static load)	~8+ (200k context, 18k consumed by L1, rest retrieved on demand)
External dependencies	Markdown files only. Simple but limited.	Python + ChromaDB + SQLite. Minimal, fully local, no cloud.

The Key Insight

The before state is not "bad." It works. Thousands of AI agent users operate exactly this way: big markdown files, manual updates, no search. Cortex does not replace that approach. It structures it. The markdown files become L1. The manually-updated solution docs become L4. The reference material becomes L3. The decisions and facts that accumulate over months become L2. Everything that existed before still exists. It just has a place now, and the system knows how to find it.

> The 74% token reduction is not the point. The point is that the remaining 26% is the right 26%. L1 loads what matters for identity. L2-L4 provide everything else on demand, when it is actually needed, in the amount that is actually relevant. No more loading the entire infrastructure doc when the question is about a calendar event.