OpenClaw Memory Problems: Context Rot and the Forgetting Fix

February 14, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** OpenClaw's memory is markdown files stored in ~/.openclaw/ — a soul file, user identity, memory files, workspace config, and session history. All of these load on every API call, including heartbeats. After weeks of use, that's 75-100KB of context shipping with every request. Session history alone can hit 111KB (28,000 tokens). When context fills up, the model starts ignoring older instructions, contradicting itself, and hallucinating state. Fixes: create a 'new session' purge command (drops context from 111KB to 5-15KB), trim your context files to under 20KB total, edit memory files directly to remove drift, and pin critical instructions in the soul file where they load first.

📚 **More on this topic:** Why Your Chatbot Forgets Everything · Session-as-RAG Memory System · OpenClaw Token Optimization · Context Length Explained · OpenClaw Setup Guide · Planning Tool

You told your OpenClaw agent to always format reports in markdown tables. It did that for a week. Then it stopped. You told it your name, your timezone, your project stack. A few days later it asks your name again.

This is the most common OpenClaw complaint. The agent appears to develop amnesia mid-conversation, ignore instructions it followed yesterday, or contradict something it said ten messages ago. Users assume it’s a bug. It’s not. It’s how the memory system works, and once you understand the architecture, you can fix most of it in a few minutes.

How OpenClaw Memory Actually Works

OpenClaw doesn’t have a brain. It has a filing cabinet. Every piece of “memory” is a plain markdown file stored in `~/.openclaw/` that gets loaded into the prompt before every API call.

The Five Memory Layers

Layer	File Type	What It Stores	When It Loads
Soul file	Markdown	Agent personality, core operating instructions, behavioral rules	Every call
User identity	Markdown	Your name, preferences, communication style, project details	Every call
Memory files	Markdown	Accumulated context from previous sessions, facts the agent learned	Every call
Workspace config	Markdown	Task routing rules, model preferences, project-specific settings	Every call
Session history	.jsonl	Complete conversation log for the current session/ channel	Every call

Notice that last column. Every layer loads on every API call. Not just when you send a message. Heartbeats (every 30 minutes), tool calls, timer events, cron jobs. Every single interaction loads the full context stack.

What This Means in Practice

A fresh OpenClaw install starts at roughly 50KB of total context. After a few weeks of active use, that grows:

Time Using OpenClaw	Approx. Context Size	Tokens Per Call	Monthly Cost (Sonnet, idle)
Day 1	~50KB	~12,500	~$30
1 week	~70KB	~17,500	~$50
1 month	~100KB	~25,000	~$75
3 months	~150KB+	~37,500+	~$110+

The agent doesn't get smarter as memory grows. It gets slower, more expensive, and paradoxically worse at remembering what matters. The model's attention has to spread across more tokens, and the important instructions from your soul file compete with hundreds of kilobytes of accumulated noise.

The Critical Architecture Problem

Every major LLM processes tokens sequentially. The model pays the most attention to the beginning and end of the context window and less to the middle. Research confirms this: Stanford found that with just 4,000 tokens of retrieved context, LLM accuracy drops from 70-75% to 55-60%. The NoLiMa benchmark showed 11 out of 12 tested models dropped below 50% of their short-context performance at 32,000 tokens.

Your soul file loads first (good, it gets strong attention). Your session history loads last (good for recent messages). Everything in the middle — memory files, workspace config, old conversation turns — sits in the attention dead zone. The model literally pays less attention to it.

What Gets Lost and Why

1. Context Window Overflow

Every LLM has a hard limit on how many tokens it can process at once:

Model	Context Window	Usable After Memory Overhead
Claude Sonnet	1M tokens	~750-800K tokens
Claude Opus	1M tokens	~750-800K tokens
GPT-4o	128K tokens	~90-100K tokens
Qwen 3.5 32B (local, 32K ctx)	32K tokens	~22-26K tokens
Qwen 3 32B (local, 16K ctx)	16K tokens	~6-10K tokens
Llama 3.1 8B (local, 8K ctx)	8K tokens	~3-5K tokens

That “usable” column is what’s left after subtracting your soul file, identity, memory, workspace config, and tool definitions. Even with 1M token windows, recall quality on early-conversation details degrades as context fills — bigger windows don’t fix context rot, they just give you more room to experience it. On local models with small context windows, you might have 3-5K tokens for actual conversation. That’s maybe 10-15 exchanges before older messages get pushed out.

When context overflows, older messages get truncated. The model can’t see them. Instructions you gave 20 messages ago are gone.

2. Session History Bloat

If you connect OpenClaw through Slack or WhatsApp, it loads your entire messaging history for that channel on every API call. Not just the current conversation. Everything you've ever sent.

One token audit found 111 kilobytes of raw text (roughly 28,000 tokens) uploaded with every single prompt. This is old Slack messages from weeks ago, appended to every request, including heartbeats.

The longer you use a channel, the worse it gets:

Duration	Session History Size	Overhead Per Call
1 week	~20KB (~5,000 tokens)	Manageable
1 month	~60KB (~15,000 tokens)	Noticeable slowdown
3 months	~111KB+ (~28,000 tokens)	Severe. Rate limits, quality drops.

The web interface is less aggressive about history loading, which is why users sometimes find the agent works better on the web than through Slack. It's not the agent. It's the history payload.

3. Summarization Loses Details

When context gets long, some platforms summarize older conversation turns to save space. The summary preserves the general topic ("discussed Python deployment issues") but loses the specifics ("prefers uvicorn over gunicorn, uses port 8080, deploys to a Hetzner VPS with 4GB RAM").

Those specifics are exactly what you want the agent to remember. Summarization flattens them into something useless. You told the agent your entire deployment pipeline, and the summary recorded "user has a deployment pipeline."

4. Cross-Session Amnesia

Starting a new session in OpenClaw means starting from scratch for conversation context. The soul file, identity, and memory files carry over (they're persistent files). But the actual conversation — the back-and-forth where you explained your project, debugged an issue, made decisions. That lives in session history. New session = empty session history.

This is why the agent remembers your name (stored in identity file) but forgets that you spent an hour yesterday explaining why the database migration needs to happen before the API changes (stored only in session history, now cleared).

5. Memory File Drift

OpenClaw writes to its own memory files. When you tell it something, it might save a note in the memory file for future reference. The problem: the agent decides what's worth saving. It often stores the wrong things or writes them in ways that are useless later.

Common drift patterns:

- **Over-summarization:** You explained a nuanced preference. The agent stored "user prefers X." The nuance is gone.
- **Stale facts:** You changed your mind about something. The old preference is still in memory.
- **Conflicting entries:** The agent added a note contradicting an earlier one. Now both load, confusing the model.
- **Noise accumulation:** Trivial facts pile up ("user asked about weather on Feb 3") taking space from important ones.

Nobody audits these files. They grow silently until the accumulated noise makes the agent worse, not better.

6. Qwen 3.5's Thinking Tax

If you're running Qwen 3.5 locally as your agent's model, there's a context cost that isn't obvious: thinking tokens. Qwen 3.5 defaults to thinking mode, generating a `<think>...</think>` block before every response. On simple tasks — formatting a file, answering a direct question — the model can burn 300-600 tokens reasoning through something that doesn't need reasoning. On complex tasks, thinking blocks regularly hit 1,000+ tokens.

Those thinking tokens count against your context window. On a 16K context local setup, a few rounds of unnecessary overthinking eats the space you need for actual conversation. Users have reported the model "refusing to stop thinking" on straightforward prompts, spiraling through caveats and edge cases for questions that have one-line answers.

Qwen 3.5 doesn't support the `/nothink` soft switch that Qwen 3 had. You have two options: set `enable_thinking: false` in your API parameters to disable it entirely, or use a `ThinkingTokenBudgetProcessor` to cap thinking at a fixed token count (300 is usually enough for agent tasks). If you're using Ollama, set `PARAMETER num_predict` in a custom modelfile to limit total output length, which indirectly constrains thinking.

For agent work where you're already fighting for context space, disabling thinking on the model you use for heartbeats and simple tool calls is worth testing. Keep thinking enabled for the model handling complex reasoning tasks.

Context Rot: When Long Sessions Go Bad

Context rot is what happens when a conversation runs long enough that the model's output quality visibly degrades. It's not a sudden failure. It's a gradual slide. The agent starts strong and slowly gets worse.

What Context Rot Looks Like

Symptom	What's Happening
Agent contradicts something it said 30 messages ago	Earlier context has fallen out of the attention window
Agent ignores a preference you stated at the start	Instruction got pushed to the middle of context where attention is weakest
Agent repeats a question it already asked	Previous exchange got truncated or attention diluted
Agent hallucinates state ("as we discussed earlier…")	Model generating plausible-sounding references to context it can't actually see
Tool calls get less accurate	Tool schemas competing with conversation history for attention
Responses get slower	Context window near capacity, more tokens to process

The Math Behind It

Attention in transformers scales quadratically with sequence length. Processing 100K tokens doesn't take twice as long as 50K. It takes roughly four times the compute for the attention step. Near the context limit, every additional token makes the model slower and spreads attention thinner.

Practical rule: **stay within 80% of your model's context window.** Beyond that, quality drops noticeably. On a local model with 16K context, that means keeping total context (memory + conversation) under ~13K tokens.

Context Rot vs Genuine Bugs

Before blaming context rot, check:

- **Did you recently switch models?** Different models handle long context differently. Switching from Claude (200K) to a local 7B model (8K) will cause immediate "forgetting."

- **Is the session corrupted?** Orphaned tool calls cause cascading failures that look like memory loss. See our tool call troubleshooting guide.
- **Did the model change?** API providers update models. A model change can alter how well the agent follows persistent instructions.

Practical Fixes

Fix 1: Pin Critical Context in the Soul File

The soul file loads first in the prompt. It gets the strongest attention from the model. Anything you always want the agent to remember goes here.

Edit `~/.openclaw/` and find your agent's soul file. Add a section for permanent instructions:

```
## Always Remember
- My name is Sarah
- I'm in PST timezone
- Format all reports as markdown tables
- Never commit code without asking first
- My deployment target is a Hetzner VPS (4GB RAM, Ubuntu 22.04)
- I prefer uvicorn over gunicorn, port 8080
```

These instructions load at the top of every prompt. The model sees them first and gives them priority. Don't put transient project details here — only things that should persist across all tasks.

Target size: Keep the soul file under 5KB. Every byte competes with your actual conversation for context space.

Fix 2: Use Explicit Memory Commands

OpenClaw doesn't have built-in `/remember` or `/forget` commands. But you can create them by instructing the agent:

Tell your agent:

> "When I say 'remember this': save the following fact to your memory file with today's date. When I say 'forget this': remove the specified fact from your memory file."

Then use it:

> "Remember this: the staging database is at 10.0.1.50, credentials in 1Password vault 'DevOps'."

The agent writes it to the memory file. Next session, the fact loads with everything else. This is more reliable than hoping the agent decides to save something on its own.

Fix 3: The "New Session" Purge

Before any major task, purge old session history. Tell your agent:

> "When I say 'new session': save a summary of the current session to long-term memory, then clear all active session history. Future prompts should not load previous conversation context unless I explicitly ask to recall something."

The impact is dramatic:

Metric	Before Purge	After Purge
Context per prompt	50-111KB	5-15KB
Overhead tokens per call	12,000-28,000	2,000-4,000
Response quality	Degraded	Restored
Rate limit headroom	Nearly none	Plenty

When to purge:

- Start of each work day
- Before overnight batch tasks
- When switching between unrelated projects
- When the agent's responses get noticeably slower or less coherent
- After long troubleshooting sessions

Fix 4: Trim Context Files

Review everything in `~/.openclaw/`:

1. Open each markdown file in a text editor
2. Remove duplicate instructions
3. Remove stale facts (old project details, outdated preferences)
4. Consolidate overlapping files
5. Cut biographical details the agent doesn't need for tasks

Target: under 20KB total for all context files combined (soul + identity + memory + workspace). Every extra kilobyte ships with every API call, including heartbeats.

Fix 5: Memory File Hygiene

Your agent writes to its memory files, but you should edit them. Open `~/.openclaw/` and look at what's accumulated:

- Delete trivial entries ("user asked about weather on Tuesday")
- Fix or remove contradictory entries
- Update stale facts
- Consolidate related entries into single clear statements
- Remove anything already captured in the soul file

Do this weekly. Five minutes of cleanup prevents kilobytes of accumulated noise from degrading your agent's performance.

Memory Configuration in openclaw.json

Heartbeat Settings

Heartbeats load full memory every 30 minutes by default. That's 48 times a day your agent reads every context file even if you haven't said a word. Configure them:

```
{
  "agents": {
    "defaults": {
      "heartbeat": {
        "every": "30m",
        "model": "ollama/llama3.1:8b"
      }
    }
  }
}
```

Routing heartbeats to a free local model via Ollama eliminates the cost problem entirely. See our token optimization guide for the full setup.

Sub-Agent Memory Isolation

Sub-agents inherit the parent's context by default. For memory-intensive tasks, isolate them:

```
{
  "agents": {
    "defaults": {
      "subagents": {
        "model": "anthropic/claude-3-5-haiku-latest",
        "maxConcurrent": 3
      }
    }
  }
}
```

Sub-agents with their own model run in separate context spaces. A research scout doesn't need your soul file, identity, or workspace config. It just needs its task instructions. This keeps sub-agent context lean and focused.

Model Context Window

For local models through Ollama, set the context window in your modelfile:

```
FROM qwen3.5:32b
PARAMETER num_ctx 32768
```

Bigger context = more memory capacity but more VRAM. On 24GB with Qwen 3.5 32B at Q4, 32768 context uses about 22-23GB. Qwen 3.5 natively supports up to 262K tokens, but on consumer VRAM you'll want to cap it. Going past 32K on 24GB will cause out-of-memory errors or heavy CPU offloading.

When Built-In Memory Isn't Enough

OpenClaw's file-based memory works for preferences, instructions, and short-term facts. It doesn't work for recalling specific conversations from three weeks ago or searching your history semantically.

OpenClaw v2026.3.7 (March 2026) added a ContextEngine plugin slot that lets plugins replace the built-in compaction logic. The most notable plugin is lossless-claw, which replaces the

default sliding-window truncation with a DAG-based summarization system. Every message gets stored in SQLite. Older chunks get summarized hierarchically. The agent gets search tools (`lcm_grep`, `lcm_describe`, `lcm_expand`) to drill back into compacted history when it needs details. Install with `openclaw plugins install @martian-engineering/lossless-claw`. If you've been losing context to compaction, this is the first thing to try.

For memory that persists across sessions and across tools, MCP memory servers have matured fast in early 2026.

MCP Memory Servers

Engram is the most complete option right now. It's a Go binary with SQLite + FTS5 full-text search, exposed as an MCP server. It works with OpenClaw, Claude Code, Cursor, or anything that speaks MCP. The agent gets tools like `mem_save`, `mem_search`, and `mem_session_summary` to store and retrieve memories across sessions. Engram scores 80% accuracy on the LOCOMO agent memory benchmark (Mem0 scores 66.9%) and uses 96.6% fewer tokens than full-context approaches. Memories are stored locally, embedded with a local model (all-MiniLM-L6-v2), and retrieved via semantic similarity. It includes memory decay (unused memories fade), contradiction detection, and a relationship graph linking related memories.

Other MCP memory servers worth watching: mcp-memory-keeper for simpler persistent context, and the growing list on PulseMCP's memory category. The MCP approach solves the fundamental problem with OpenClaw's built-in memory: instead of cramming everything into markdown files that load on every call, the agent queries for relevant memories only when it needs them.

Session-as-RAG

Treat your conversation history as a searchable document corpus. Each exchange gets embedded in a vector database (ChromaDB, Pinecone). When you start a new conversation, the system searches your past sessions for relevant context and injects it into the prompt.

This turns "I know we discussed that ChromaDB error last week" from a dead end into a real retrieval: the system finds the exact exchange, pulls the resolution, and includes it in the current prompt.

We built a full implementation guide: Session-as-RAG: Teaching Your Local AI to Actually Remember. It's about 80 lines of Python with Ollama + ChromaDB.

AnythingLLM

AnythingLLM provides workspaces with persistent document and conversation memory. You can upload reference documents that persist across sessions, and the system manages context automatically. It's a simpler path than building custom RAG if you want persistent memory without writing code.

MemGPT / Letta

MemGPT (now called Letta) uses hierarchical memory: working memory for the current task, recall memory for recent sessions, and archival memory for long-term storage. The model manages its own memory, deciding what to keep and what to archive. It's the most sophisticated approach but also the most complex to set up.

The Fundamental Tradeoff

More memory = more tokens per call. Every fact you inject from external memory consumes context space that could hold conversation. A RAG system that retrieves 10 relevant past exchanges adds 2,000-5,000 tokens to every prompt. That's fine on a 200K context window. It's a problem on a local model with 8K context.

Budget your context like VRAM: there's a fixed amount, and everything competes for it.

Bottom Line

OpenClaw's memory is a filing cabinet, not a brain. Markdown files load into the prompt, the model reads them, and then the computation is gone. There's no "learning." There's no "remembering." There's reading from files and then forgetting.

Most memory problems come from three sources:

1. **Session history bloat.** Old Slack/WhatsApp messages pile up, eating 28,000+ tokens per call. Fix with a "new session" purge.
2. **Uncurated memory files.** The agent writes noise to its memory. Nobody cleans it up. Fix by editing the files directly, weekly.
3. **Context rot in long sessions.** After 30+ exchanges, attention dilutes and the model starts ignoring earlier context. Fix by purging before major tasks and staying under 80% of context capacity.

The soul file is your most powerful tool. Anything the agent should always know goes there, at the top, where the model pays the most attention. Trim everything else. Purge regularly. Edit the memory files yourself.

Your agent's memory is only as good as the files it reads. Make those files clean and you fix most of the forgetting.

Related Guides

- The AI Memory Wall: Why Your Chatbot Forgets Everything — the six architectural reasons LLMs forget
- Session-as-RAG: Teaching Your Local AI to Actually Remember — build persistent memory with ChromaDB
- OpenClaw Token Optimization — cut costs from memory overhead
- Context Length Explained — how context windows work
- OpenClaw Model Routing — route heartbeats to free local models
- OpenClaw Setup Guide — installation and configuration
- Local AI Planning Tool — VRAM Calculator

Get notified when we publish new guides.

Subscribe — free, no spam
