AI

- QMD
- Summary
- OpenViking
- LanceDB Pro
- design languange

QMD

QMD — Local Hybrid Search for OpenClaw Agent Memory

What it is: QMD is a fully local, on-device search engine for markdown files built by Tobi Lütke (Shopify CEO). It replaces OpenClaw's broken built-in memory search with a three-stage hybrid pipeline: BM25 keyword matching + vector semantic search + LLM re-ranking. No API keys, no cloud calls, no network traffic after initial model download.

Why it matters: OpenClaw's default `memory_search` uses pure vector embeddings that routinely return semantically adjacent but factually wrong results. QMD fixes this by running three search methods in parallel and fusing them with Reciprocal Rank Fusion, then re-ranking the top candidates with a local LLM. Community reports claim it cuts token usage by 95%+ compared to context-stuffing approaches.

Install: `npm install -g @tobilu/qmd` (requires Node.js 22+ or Bun). Three GGUF models (~2.5GB total) auto-download from HuggingFace on first use.

How It Works

Every query goes through this pipeline:

1. **Query Expansion** — A local 1.7B LLM generates two alternative formulations of your query
2. **Parallel Search** — All three query variants (original weighted 2×) run through both BM25 full-text search AND vector cosine similarity search simultaneously
3. **Reciprocal Rank Fusion (RRF)** — Results merge with k=60, original query weighted 2×, top-rank bonus (+0.05 for #1, +0.02 for #2-3)
4. **LLM Re-ranking** — Top 30 candidates scored by a local reranker model (yes/no + logprob confidence)
5. **Position-Aware Blending** — Final ranking blends RRF and reranker scores: ranks 1-3 use 75% RRF / 25% reranker, ranks 4-10 use 60/40, ranks 11+ use 40/60

Documents are chunked into ~900-token pieces with 15% overlap, using smart boundary detection that prefers markdown headings. Embeddings and LLM responses are cached in SQLite (`~/.cache/qmd/index.sqlite`

) with content-hash keying, so moving/renaming files doesn't require re-embedding.

Local Models

QMD runs three GGUF models locally via `node-llama-cpp`. All models auto-download to `~/.cache/qmd/models/` on first use:

Purpose	Model	Size	Source
Embedding	embeddinggemma-300M (Google)	~300MB	HuggingFace
Re-ranking	Qwen3-Reranker-0.6B-Q8_0	~600MB	HuggingFace
Query Expansion	qmd-query-expansion-1.7B (Tobi's fine-tune)	~1.7GB	HuggingFace

On Apple Silicon, QMD auto-detects Metal GPU acceleration at startup. Total VRAM/memory footprint is ~2.5GB — negligible on a 128GB Mac Studio.

Swapping the Embedding Model

The embedding model can be overridden via environment variable:

```
# Use Qwen3-Embedding for better multilingual (CJK) support
export QMD_EMBED_MODEL="hf:Qwen/Qwen3-Embedding-0.6B-GGUF/Qwen3-Embedding-0.6B-Q8_0.gguf"

# After changing model, re-embed all collections:
qmd embed -f
```

Note: This still loads a local GGUF file — it doesn't point at a remote API. Vectors are not cross-compatible between models, so you must re-index after switching.

Can I Point Models to a Remote Machine?

Short answer: No — QMD is designed to be fully local.

QMD runs all inference through `node-llama-cpp` in-process. There is no built-in configuration to route embedding, reranking, or query expansion to a remote API endpoint. The project's tagline is literally "Tracking current sota approaches while being all local."

There is a `QMD_OPENAI_BASE_URL` environment variable referenced in some third-party integration guides (specifically the `ehc-io/qmd` fork), but this is **not part of Tobi's original QMD** and applies

to a different use case.

Workarounds if you need remote inference:

- The models are tiny (~2.5GB total). On a 128GB Mac Studio they're negligible — just run them locally alongside your main LLM
- If you absolutely need to offload: QMD exposes an MCP server (`qmd mcp`) with HTTP transport mode. You could run QMD on a remote machine and connect to it as an MCP service. But the models themselves still run local to wherever QMD is installed
- Fork and modify — QMD is MIT licensed and the embedding/reranking code is in `src/`. You could swap the `node-llama-cpp` calls for HTTP calls to a remote endpoint, but this is custom development

---

OpenClaw Integration

Three integration paths, from simplest to most flexible:

1. Built-in Memory Backend (Recommended)

Set `memory.backend = "qmd"` in `~/.openclaw/openclaw.json`:

```
{
 "memory": {
  "backend": "qmd",
  "citations": "auto",
  "qmd": {
   "includeDefaultMemory": true,
   "command": "qmd",
   "searchMode": "search",
   "update": {
    "interval": "5m",
    "debounceMs": 15000,
    "onBoot": true,
    "waitForBootSync": false
   },
   "limits": {
    "maxResults": 6,
    "timeoutMs": 4000
   },
   "scope": {
    "default": "deny",
```

```
      "rules": [
        { "action": "allow", "match": { "chatType": "direct" } }
      ]
    }
  }
}
}
```

OpenClaw automatically creates collections (e.g. `memory-root` for `memory/**/*.md`) and indexes on boot.

2. MCP Server

Run `qmd mcp` to expose QMD as an MCP tool. Supports stdio and HTTP transport. HTTP daemon mode keeps models warm in VRAM between queries, reducing latency from ~16s to ~10s.

3. openclaw-engram Plugin

A community plugin (github.com/joshuaswarren/openclaw-engram) that uses QMD as the backend for a persistent long-term memory system with LLM-powered extraction.

Search Modes

Command	Method	Speed	Best For
`qmd search`	BM25 keyword only	~50-200ms	You know the exact terms
`qmd vsearch`	Vector similarity only	~500-1000ms	Semantic matches without reranking
`qmd query`	Full hybrid pipeline	~3-10s	Highest quality results (recommended)

Key Features

- **Query documents** — Structured multi-line queries with typed lines (`lex:`, `vec:`, `hyde:`) combining keyword precision with semantic recall
- **Intent parameter** — Optional `--intent` flag disambiguates queries across the entire pipeline. "performance" means different things in different contexts
- **Quoted phrases and negation** — `"C++ performance" -sports -athlete` works in lex search
- **Content-hash keying** — Moving/renaming files doesn't require re-embedding
- **LLM response caching** — Query expansion and rerank scores cached in SQLite

- **Agentic output formats** — `--json`, `--files`, `--md`, `--full` for structured agent consumption
- **Collection contexts** — Hierarchical semantic metadata (e.g. "Personal notes and meeting logs") improves relevance
- **Separate indexes** — `qmd --index work search "quarterly reports"` keeps knowledge bases isolated

Storage

- **Index:** `~/.cache/qmd/index.sqlite` (SQLite with FTS5 + sqlite-vec)
- **Models:** `~/.cache/qmd/models/` (~2.5GB)
- **Config:** `~/.config/qmd/index.yml` (collection definitions, respects XDG_CONFIG_HOME)

Quick Reference

```
# Install
npm install -g @tobilu/qmd

# Create a collection
qmd collection add ~/.openclaw/workspace --name workspace --mask "**/*.md"

# Generate embeddings
qmd embed

# Search
qmd query "what did I decide about the camera setup"

# Status
qmd status

# Re-index (with git pull for remote repos)
qmd update --pull
```

Version: v1.1.6 (current as of March 2026)
License: MIT

GitHub: github.com/tobi/qmd
npm: `@tobilu/qmd`
Runtime: Node.js 22+ or Bun
Dependencies: node-llama-cpp, sqlite-vec, better-sqlite3

Summary

OpenClaw Memory Enhancement Stack

Four tools that address different layers of OpenClaw's memory problem. Each solves a different failure mode — they're complementary, not competing.

Tool	What It Does	Runs On	Status
QMD	Hybrid search (BM25 + vector + LLM rerank) over markdown files	Linux agent box (CPU) or Mac (Metal)	Stable — v1.1.6
memory-lancedb-pro	Vector memory plugin with decay, hybrid retrieval, scope isolation	Linux agent box, embeddings on Mac	Stable — npm package
OpenViking	Context database with virtual filesystem and tiered token loading	Linux agent box, VLM calls on Mac	⚠ Hold — LiteLLM supply chain attack
MetaClaw	Continual learning proxy — agent gets smarter over time	Linux agent box, forwards inference to Mac	Beta — v0.4.0, very new

---

QMD — Search That Actually Works

Created by: Tobi Lütke (Shopify CEO)

GitHub: github.com/tobi/qmd
License: MIT
Install: `npm install -g @tobilu/qmd`

What it solves: OpenClaw's built-in `memory_search` uses pure vector embeddings that routinely return wrong results. QMD replaces it with a three-stage hybrid pipeline.

How it works:

1. A local 1.7B LLM expands your query into two alternative formulations

2. All three variants run through BM25 keyword search AND vector cosine similarity in parallel
3. Results merge via Reciprocal Rank Fusion (original query weighted 2×)
4. Top 30 candidates scored by a local reranker model
5. Final ranking blends RRF and reranker scores with position-aware weighting

Local models (~2.5GB total, auto-downloaded):

- embeddinggemma-300M (embedding)
- Qwen3-Reranker-0.6B (reranking)
- qmd-query-expansion-1.7B (query expansion — Tobi's custom fine-tune)

Remote inference: Not supported natively. Models run via node-llama-cpp in-process. On 128GB Mac Studio they're negligible. On the Linux agent box they run fine on CPU (~10-15s per query vs ~3-5s on Metal).

OpenClaw integration: Set `memory.backend = "qmd"` in openclaw.json. OpenClaw creates collections and indexes automatically on boot.

Key commands:

```
qmd query "what did I decide about the camera setup"   # hybrid search (best quality)
qmd search "frigate RTSP"                              # keyword only (fastest)
qmd vsearch "home automation preferences"              # vector only
qmd status                                             # index info
qmd update --pull                                      # re-index (git pull first)
qmd embed -f                                           # force re-embed all
```

---

memory-lancedb-pro — Long-Term Memory With Decay

Created by: CortexReach (community)

GitHub: github.com/CortexReach/memory-lancedb-pro
License: MIT
Install: `npm i memory-lancedb-pro`

What it solves: OpenClaw's default memory has no decay (everything stays equally weighted forever), no hybrid search (vector only), no deduplication, and MEMORY.md dumps its entire contents into every session wasting tokens.

How it works:

- **Auto-capture:** Extracts preferences, facts, decisions, and entities from conversations automatically (up to 3 per turn)
- **Auto-recall:** Injects relevant memories into context before agent responds (up to 3 entries)
- **Smart extraction:** LLM-powered 6-category classification (Profile, Preferences, Entities, Events, Cases, Patterns) with L0/L1/L2 metadata
- **Hybrid retrieval:** Vector search + BM25 keyword search fused with RRF, then cross-encoder reranking
- **Weibull time decay:** Memories that aren't accessed gradually fade from active retrieval
- **Three-tier system:** Core → Working → Peripheral with automatic promotion/demotion
- **Multi-scope isolation:** global, agent:<id>, project:<id>, user:<id>, custom:<name>

Remote inference: Yes — embedding uses any OpenAI-compatible `/v1/embeddings` endpoint. Point `baseUrl` at Ollama/LM Studio on the Mac. Reranking uses Jina, SiliconFlow, or any compatible reranker API.

Key config (openclaw.json):

```
{
  "plugins": {
    "slots": { "memory": "memory-lancedb-pro" },
    "entries": {
      "memory-lancedb-pro": {
        "enabled": true,
        "config": {
          "embedding": {
            "model": "nomic-embed-text",
            "baseUrl": "http://<mac-headscale-ip>:11434/v1",
            "apiKey": "not-needed"
          },
          "autoCapture": true,
          "autoRecall": true,
          "smartExtraction": { "enabled": true },
          "retrieval": { "mode": "hybrid", "vectorWeight": 0.7, "bm25Weight": 0.3 }
        }
      }
    }
  }
}
```

Key commands:

```
openclaw memory-pro list --scope global --limit 20      # list memories
openclaw memory-pro search "query" --scope global       # search memories
openclaw memory-pro stats --scope global                # count, categories, age
openclaw memory-pro export --output memories.json       # backup
openclaw memory-pro import memories.json --dry-run      # import (test first)
openclaw memory-pro reembed                             # after changing embedding model
openclaw memory-pro delete <id>                         # delete specific memory
openclaw memory-pro delete-bulk --before "2026-01-01"   # bulk delete
openclaw memory-pro migrate check --source /path/to/old  # migrate from built-in plugin
```

After config changes: `openclaw config validate` then `openclaw gateway restart`.
After editing plugin .ts files: `rm -rf /tmp/jiti/` then restart (jiti caches stale compiled code).

OpenViking — Context Database With Virtual Filesystem

Created by: ByteDance / Volcengine Viking Team

GitHub: github.com/volcengine/OpenViking
License: Apache 2.0
Stars: ~17,900

⚠ **DO NOT INSTALL RIGHT NOW.** OpenViking has a dependency on `litellm>=1.0.0`. LiteLLM was hit by a supply chain attack on March 24, 2026 (TeamPCP backdoored versions 1.82.7-1.82.8). The entire litellm package is currently quarantined on PyPI — fresh installs of anything depending on it will fail. Wait for the quarantine to lift, then install with `"provider": "openai"` pointing at your local Ollama endpoint to bypass LiteLLM entirely.

What it solves: Replaces flat vector storage with a hierarchical virtual filesystem. Instead of dumping everything into embeddings and hoping retrieval works, OpenViking organises context into structured directories accessible via `viking://` URIs.

How it works:

- **Virtual filesystem:** Three root directories — `viking://resources/` (documents, repos), `viking://user/` (preferences, habits), `viking://agent/` (skills, task memories)
- **Unix-like navigation:** `ls`, `find`, `read`, `tree`, `grep` against agent context
- **L0/L1/L2 tiered loading:** L0 = ~100 tokens (abstract), L1 = ~2,000 tokens (overview), L2 = full content on demand. Claims 91-95% token cost reduction vs traditional RAG
- **Automatic memory self-iteration:** At session end, extracts 6 memory categories and updates the appropriate directories

Remote inference: Yes — configure `"provider": "openai"` with `"api_base"` pointing at the Mac's Ollama. This bypasses the LiteLLM dependency entirely for your use case.

Benchmark results with OpenClaw:

- Task completion: 52.08% (vs 35.65% native OpenClaw — 43% improvement)
- Input tokens: 4.3M (vs 24.6M native — 91% reduction)

When to install: After LiteLLM quarantine lifts. Clone repo, strip `litellm>=1.0.0` from pyproject.toml if needed, use `openai` provider only.

MetaClaw — The Agent That Learns From Its Mistakes

Created by: AIMING Lab, UNC Chapel Hill

GitHub: github.com/aiming-lab/MetaClaw
License: Apache 2.0

Paper: arXiv 2603.17187 (ranked #1 on HuggingFace Daily Papers, March 18 2026)
Stars: ~2,700

What it solves: The other three tools store and retrieve memories. MetaClaw is the only tool that actually makes the agent *smarter over time*. It learns from failure patterns and generates corrective behavioural skills.

How it works: MetaClaw sits as an OpenAI-compatible proxy between OpenClaw and your LLM. It intercepts every interaction.

- **Skill-driven fast adaptation (gradient-free):** Analyses failure trajectories via an LLM evolver and synthesises new behavioural skills that take effect immediately. No GPU needed, no downtime. If your agent repeatedly makes the same mistake, MetaClaw generates a corrective skill.
- **Opportunistic policy optimisation (gradient-based, optional):** Cloud LoRA fine-tuning via RL, triggered only during user-inactive windows. An Opportunistic Meta-Learning Scheduler monitors sleep hours, keyboard inactivity, and calendar to find training windows. Can be skipped entirely.

Results: Skill-driven adaptation improved accuracy by up to 32% relative. Full pipeline advanced Kimi-K2.5 from 21.4% to 40.6% accuracy (approaching GPT-5.2's 41.1%).

Remote inference: Yes — MetaClaw IS a proxy. Run it on the Linux box, point its upstream at the Mac's LM Studio/Ollama endpoint. All inference happens on the Mac. MetaClaw just intercepts and learns.

For your setup: Use skills-only mode (gradient-free). No GPU needed on the Linux box. Since v0.3.3, MetaClaw has a native OpenClaw plugin. Since v0.4.0 (March 25), it includes a "Contexture layer" for cross-session memory.

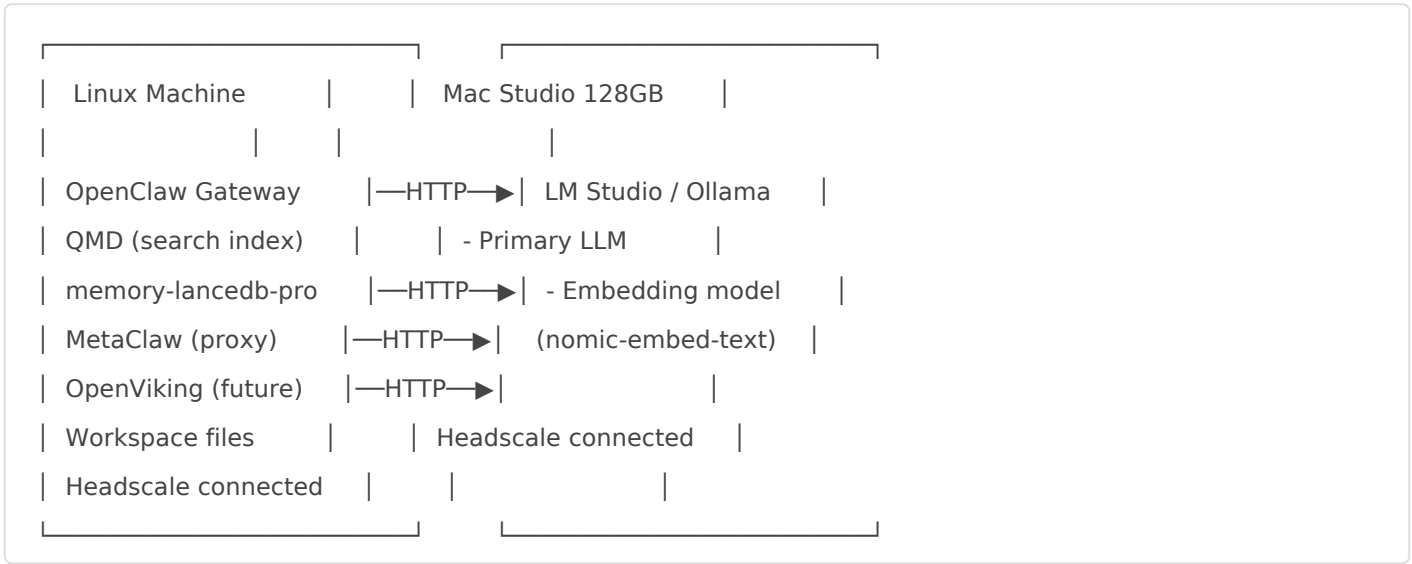
Caveat: Very new (16 days old, 7 releases). Academically rigorous but least battle-tested of the four tools. Worth running but expect rough edges.

Install Order

1. **QMD** — `npm install -g @tobilu/qmd`, set `memory.backend = "qmd"`. Immediate improvement to search/recall.
2. **memory-lancedb-pro** — `npm i memory-lancedb-pro`, configure embedding endpoint to Mac. Fixes memory bloat and adds decay.
3. **MetaClaw** — Install as OpenClaw plugin, configure as proxy. Adds learning over time.
4. **OpenViking** — Wait for LiteLLM situation to resolve. Then install with `openai` provider pointing at Mac.

Test each layer before adding the next. If something breaks, you know which component caused it.

Architecture



All heavy inference on the Mac. All files, indexing, gateway, and agent logic on the Linux box.

OpenViking

OpenViking — Context Database for AI Agents

What it is: OpenViking is an open-source context database from ByteDance's Volcengine team that replaces flat vector storage with a hierarchical virtual filesystem. Instead of dumping everything into embeddings and hoping retrieval works, it organises all agent context into structured directories accessible via `viking://` URIs — like a Unix filesystem for your agent's brain.

Why it matters: In benchmarks with OpenClaw, OpenViking improved task completion from 35.65% to 52.08% (43% improvement) while cutting input tokens from 24.6M to 4.3M (91% reduction). The tiered loading system means your agent only pulls in what it needs, when it needs it.

GitHub: github.com/volcengine/OpenViking
Stars: ~17,900
License: Apache 2.0
Language: Python (core) + Rust (CLI) + C++ (vector extensions) + Go (AGFS server)
Requires: Python 3.10+

How It Works

Virtual Filesystem

All agent context lives in three root directories:

- `viking://resources/` — documents, repos, web pages, project files
- `viking://user/` — preferences, habits, personal context
- `viking://agent/` — skills, task memories, instructions

You interact with context using Unix-like commands: `ls`, `find`, `read`, `tree`, `grep`. The agent navigates its own memory the same way you'd navigate a filesystem.

L0/L1/L2 Tiered Loading

This is the key innovation that solves token bloat. Instead of loading full documents into context, OpenViking processes everything into three tiers:

- **L0 (Abstract):** ~100 tokens — one-sentence summary for quick identification
- **L1 (Overview):** ~2,000 tokens — structured overview with key details
- **L2 (Full Content):** Complete document, loaded on demand only

The agent starts with L0 summaries, drills into L1 when something looks relevant, and only loads L2 when it genuinely needs the full content. Claims 91-95% token cost reduction vs traditional RAG approaches.

Automatic Memory Self-Iteration

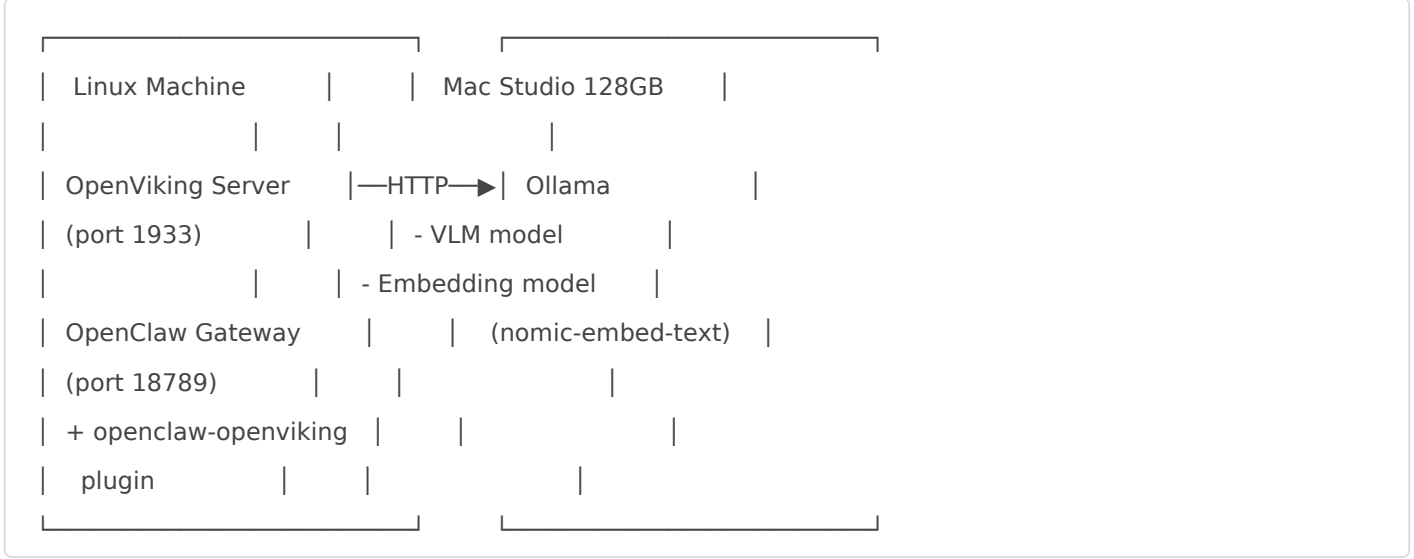
At session end, OpenViking automatically extracts six categories of memory and updates the appropriate directories:

1. **Profile** — factual information about the user
2. **Preferences** — how the user likes things done
3. **Entities** — people, projects, tools referenced
4. **Events** — decisions made, things that happened
5. **Cases** — problems solved, approaches taken
6. **Patterns** — recurring behaviours and workflows

This is significantly more structured than OpenClaw's default "hope the LLM remembers to write to MEMORY.md" approach.

Architecture For Our Setup

OpenViking runs on the Linux agent box. All LLM inference (VLM and embedding) routes to the Mac Studio over Headscale.



VLM Providers

OpenViking supports three VLM providers. For our local setup, use `openai` to bypass the LiteLLM dependency entirely:

Provider	Description	Use For Our Setup?
`volcengine`	ByteDance's Doubao models (cloud)	No — cloud only
`openai`	Any OpenAI-compatible API endpoint	**Yes — point at Ollama on Mac**
`litellm`	Multi-provider routing via LiteLLM	**No — compromised, avoid**

Installation

Prerequisites

```
# Install uv (Python package manager used by OpenViking)
curl -LsSf https://astral.sh/uv/install.sh | sh

# Install the Rust CLI
curl -fsSL https://raw.githubusercontent.com/volcengine/OpenViking/main/crates/ov_cli/install.sh | bash
```

Install OpenViking (with LiteLLM workaround)

```
# Clone the repo
git clone https://github.com/volcengine/OpenViking.git
cd OpenViking

# Create virtual environment
uv venv --python 3.11
source .venv/bin/activate

# IMPORTANT: Edit pyproject.toml to remove or comment out litellm dependency
# Find the line with litellm>=1.0.0 and remove it
nano pyproject.toml

# Install
uv pip install -e .
```

Install VikingBot (agent framework, optional)

```
uv pip install -e ".[bot]"
```

Install OpenClaw Plugin

```
# The official OpenClaw integration plugin
openclaw plugins install openclaw-openviking-plugin
```

Configuration

Create the config file at `~/.openviking/ov.conf`:

```
{
  "storage": {
    "workspace": "/home/conor/openviking_workspace"
  },
  "log": {
    "level": "INFO",
    "output": "stdout"
  },
  "embedding": {
    "dense": {
      "api_base": "http://100.64.0.9:11434/v1",
      "api_key": "not-needed",
      "provider": "openai",
      "dimension": 768,
      "model": "nomic-embed-text"
    },
    "max_concurrent": 10
  },
  "vlm": {
    "api_base": "http://100.64.0.9:11434/v1",
    "api_key": "not-needed",
    "provider": "openai",
    "model": "qwen3-coder-next",
    "max_concurrent": 100
  }
```

```
}
```

Notes:

- `100.64.0.9` is the Mac Studio's Headscale IP
- `provider: "openai"` works with any OpenAI-compatible endpoint including Ollama
- Embedding dimension must match the model — 768 for nomic-embed-text, 1024 for mxbai-embed-large
- The VLM model handles the intelligent context processing (summarisation, extraction, classification)

Running

```
# Start the OpenViking server
openviking-server

# Default address: 127.0.0.1:1933

# Start with VikingBot enabled (optional)
openviking-server --with-bot
```

Key Commands (ov CLI)

Adding Resources

```
# Add a GitHub repo
ov add-resource https://github.com/volcengine/OpenViking

# Add a local directory
ov add-resource /home/conor/.openclaw/workspace

# Wait for processing to complete (otherwise runs async)
ov add-resource /path/to/docs --wait
```

Browsing Context

```
# List root directories
ov ls viking://resources/

# Tree view (2 levels deep)
ov tree viking://resources/my-project -L 2

# Read a specific file at L1 (overview)
ov read viking://resources/my-project/README.md

# Check server status
ov status
```

Searching

```
# Semantic search across all context
ov find "what is the camera VLAN setup"

# Grep for exact terms within a scope
ov grep "RTSP" --uri viking://resources/home-automation
```

Interactive Chat (with VikingBot)

```
# Start interactive chat session
ov chat
```

OpenClaw Integration

Once the `openclaw-openviking-plugin` is installed, OpenViking becomes available as a context source for your agent. The plugin connects to the OpenViking server running on the same machine and provides the agent with access to the virtual filesystem commands.

The OpenViking server must be running before the OpenClaw gateway starts. Consider adding it as a systemd service:

```
# Example systemd service file
# /etc/systemd/user/openviking.service

[Unit]
Description=OpenViking Context Database
```

```
Before=openclaw.service

[Service]
ExecStart=/home/conor/OpenViking/.venv/bin/openviking-server
WorkingDirectory=/home/conor/OpenViking
Restart=always
RestartSec=5

[Install]
WantedBy=default.target
```

```
# Enable and start
systemctl --user enable openviking
systemctl --user start openviking
```

How OpenViking Differs From The Other Memory Tools

Feature	OpenClaw Default	memory-lancedb-pro	QMD	OpenViking
Storage model	Flat markdown files	Vector DB + BM25	SQLite + vector index	Virtual filesystem
Token efficiency	Loads everything	Top-N retrieval	Top-N retrieval	L0/L1/L2 tiered loading
Context organisation	None	Scopes (global/agent/project)	Collections	Hierarchical directories
Auto-categorisation	No	6 categories	No	6 categories + directory structure
Agent can browse	Read specific files	Search only	Search only	ls, tree, find, grep, read
Multi-resource support	Workspace only	Conversation memories	Markdown files	Git repos, URLs, docs, local dirs

OpenViking and memory-lancedb-pro are complementary, not competing. memory-lancedb-pro handles conversation-level memory (what you said, preferences extracted from chat). OpenViking handles resource-level context (your codebase, documentation, project files, knowledge bases). QMD provides the search layer across markdown files. Together they cover different retrieval needs.

Breaking Changes Warning

From the release notes: **after upgrading OpenViking, datasets/indexes generated by previous versions are not compatible.** You must rebuild:

```
# Stop the server
systemctl --user stop openviking

# Clear workspace (this deletes all indexed data — resources will need re-adding)
rm -rf /home/conor/openviking_workspace

# Restart
systemctl --user start openviking

# Re-add your resources
ov add-resource /path/to/your/stuff --wait
```

Key Files and Paths

- **Config:** `~/.openviking/ov.conf` (override with `OPENVIKING_CONFIG_FILE` env var)
- **Workspace:** Configured in ov.conf `storage.workspace` — stores all indexed data
- **CLI binary:** `ov` (Rust, installed via install script)
- **Server:** `openviking-server` (Python, from the pip install)
- **Default port:** 1933

Version: v0.2.1 (current as of March 2026)
Status: Alpha — performance and consistency not fully optimised

GitHub: github.com/volcengine/OpenViking

Docs: openviking.ai

LanceDB Pro

memory-lancedb-pro

Enhanced LanceDB memory plugin for OpenClaw — community reference guide

Overview

memory-lancedb-pro is a community-developed, production-grade long-term memory plugin for OpenClaw. It replaces the built-in `memory-lancedb` plugin with a significantly more capable retrieval pipeline, designed for agents that need persistent, high-quality memory across sessions without manual tagging or configuration overhead.

The core problem it solves: standard OpenClaw agents have no memory between sessions. Every conversation starts from zero. memory-lancedb-pro automatically captures what matters from each session and retrieves relevant context in future ones.

Primary upstream repo: `CortexReach/memory-lancedb-pro`. Several community forks exist (win4r, McBorisson, fryeggs, kvc0769) with varying additions such as Volcengine multimodal embeddings or unified Claude Code/Claude Desktop support.

> **OpenClaw 2026.3+ compatibility:** The CortexReach fork has been updated to use `before_prompt_build` hooks, replacing the deprecated `before_agent_start` hook. If you are on 2026.3.24 or later, use this fork. Run `openclaw doctor --fix` after upgrading.

Feature Comparison

Feature	Built-in memory-lancedb	memory-lancedb-pro
Vector search	✓	✓
BM25 full-text search	✗	✓
Hybrid fusion (Vector + BM25)	✗	✓ configurable weights
Cross-encoder reranking	✗	✓ Jina, SiliconFlow, Pinecone, etc.

Feature	Built-in memory-lancedb	memory-lancedb-pro
Recency / time decay scoring	✗	✓
MMR diversity filtering	✗	✓
Multi-scope isolation	✗	✓ global / agent / project / user
Smart LLM extraction	✗	✓ optional, uses any OpenAI-compatible LLM
Management CLI	✗	✓ list / search / stats / delete / export / import
Auto-capture on session end	✓ basic	✓ with deduplication, up to 3 per turn
Auto-recall before prompt	✓ basic	✓ adaptive — skips trivial/short queries
Noise filtering	✗	✓
Migration tool from built-in plugin	—	✓

Retrieval Pipeline

Queries pass through a multi-stage pipeline before results are injected into the agent prompt:

1. **Embed query** — using the configured OpenAI-compatible embedding provider
2. **Parallel search** — vector ANN search (cosine distance) + BM25 full-text search run simultaneously
3. **Hybrid fusion** — vector score used as base; BM25 hits receive a configurable weighted boost
4. **Rerank** — optional cross-encoder reranking via external API (60% cross-encoder score + 40% fused score)
5. **Lifecycle decay scoring** — recency boost, time decay, importance weight, length normalisation
6. **Filter** — hard minimum score, noise filter, MMR diversity deduplication
7. **Inject** — surviving memories injected as `<relevant-memories>` context block

If the reranker API fails, the pipeline degrades gracefully to cosine similarity reranking.

Installation

1. Clone into your OpenClaw workspace

```
cd ~/.openclaw/workspace
git clone https://github.com/CortexReach/memory-lancedb-pro.git plugins/memory-lancedb-pro
cd plugins/memory-lancedb-pro
npm install
```

> **Common mistake:** Cloning the repo somewhere other than your workspace and then using a relative path in `plugins.load.paths`. Relative paths are resolved from the workspace root. Use an absolute path if cloning elsewhere.

2. Disable the built-in memory plugin

Only one memory plugin can be active at a time. If you previously used `memory-lancedb`, disable it before enabling this plugin.

3. Add to openclaw.json

```
{
  "plugins": {
    "load": {
      "paths": ["plugins/memory-lancedb-pro"]
    },
    "entries": {
      "memory-lancedb-pro": {
        "enabled": true,
        "config": {
          "embedding": {
            "apiKey": "${JINA_API_KEY}",
            "model": "jina-embeddings-v5-text-small",
            "baseURL": "https://api.jina.ai/v1",
            "dimensions": 1024,
            "taskQuery": "retrieval.query",
            "taskPassage": "retrieval.passage",
            "normalized": true
          }
        }
      }
    },
    "slots": {
```

```
      "memory": "memory-lancedb-pro"
    }
  }
}
```

Config changes require a gateway restart. With config watch enabled (default), this happens automatically.

Key Configuration Options

Option	Default	Notes
`autoCapture`	true	Capture memories at session end
`autoRecall`	true	Inject memories before prompt build
`smartExtraction`	true	Use LLM to classify memories instead of regex
`extractMinMessages`	3	Minimum messages before extraction runs
`captureAssistant`	true	Set false to only capture user messages
`retrieval.mode`	hybrid	`vector`, `bm25`, or `hybrid`
`retrieval.vectorWeight`	0.7	Weight for vector scores in hybrid fusion
`retrieval.bm25Weight`	0.3	Weight for BM25 scores in hybrid fusion
`rerank.enabled`	false	Enable cross-encoder reranking
`rerank.candidatePoolSize`	12	Candidates passed to reranker
`rerank.minScore`	0.6	Soft minimum score post-rerank
`rerank.hardMinScore`	0.62	Hard cutoff — below this is always dropped
`sessionMemory.enabled`	true	Store session summaries on `/new`
`autoRecall.minPromptLength`	15 (EN) / 6 (CJK)	Skip recall for very short queries

Management CLI

The plugin ships with a CLI for direct memory management:

```
openclaw memory-pro list              # list stored memories
openclaw memory-pro search <query>    # semantic/keyword search
openclaw memory-pro stats             # storage stats
openclaw memory-pro delete <id>       # delete a specific memory
openclaw memory-pro export            # export all memories
openclaw memory-pro import <file>     # import memories
```

Agent Tool Definitions

When loaded, the plugin registers these tools for the agent to use directly:

- `memory_recall` — retrieve relevant memories for a query
- `memory_store` — explicitly store a memory
- `memory_forget` — delete a memory by ID or query
- `memory_update` — update an existing memory

Plus additional management tools exposed via the CLI commands above.

Multi-Scope Isolation

Memories can be scoped to control access between agents and users:

- global — shared across all agents
- agent:<id> — isolated to a specific agent
- project:<id> — shared within a project
- user:<id> — per-user isolation (useful for multi-user bots)
- custom:<name> — arbitrary named scope

Telegram Setup

If running OpenClaw with Telegram, the easiest way to configure the plugin is via the bot directly. Send the following to your main bot:

Help me connect this memory plugin with the most user-friendly configuration:

https://github.com/CortexReach/memory-lancedb-pro

Requirements:

1. Set it as the only active memory plugin
2. Use Jina for embedding and reranker
3. Use gpt-4o-mini for the smart-extraction LLM
... (continue with your preferences)

Important Notes

> **jiti cache:** After modifying any `.ts` file in the plugin, you must clear the jiti cache before restarting the gateway, or OpenClaw will load stale compiled code:
> `rm -rf /tmp/jiti/ && openclaw gateway restart`

> **Memory quality guidelines:** Never store raw conversation summaries, large blobs, or duplicates. Prefer structured, atomic facts with keywords. On any tool failure or repeated error, call `memory_recall` with relevant keywords before retrying — the fix may already be stored.

> **Spaced repetition:** Frequently recalled memories decay more slowly, similar to spaced-repetition learning systems.

Notable Community Forks

Fork	Notable additions
`CortexReach/memory-lancedb-pro`	Primary upstream. Updated for OpenClaw 2026.3+ hook architecture.
`win4r/memory-lancedb-pro`	Widely referenced in docs; standard feature set.
`fryeggs/memory-lancedb-pro`	Unified edition — extends to Claude Code, Codex CLI, and Claude Desktop via shared LanceDB backend.
`kvc0769/memory-lancedb-pro`	Adds Volcengine multimodal embedding support.
`McBorisson/memory-lancedb-pro`	Uses RRF fusion (vs. weighted boost in other forks); includes JSONL distillation pipeline.

Generated March 2026. Sources: CortexReach/memory-lancedb-pro, openclaw/openclaw docs, LanceDB blog.

design languange

● Design language summary

Vibe. Dark, minimal, slightly luminous. Information-dense but airy. Color is reserved — accent appears on actions and active states; semantic colors only for status. Most surfaces are low-contrast neutral; the eye is drawn by the one
accent.

Palette. Near-black background #0b0d12, two elevation tiers above it (#14171d panel, #1a1e26 panel-2). Borders are translucent white (7% default, 12% strong). Text is warm-white #e6eef6, muted #8a93a4, super-muted #5e6675. Accent is a
blue-violet #4f8cff with #6ea0ff for hover and a 15% soft variant for active backgrounds. A violet #8b5cf6 shows up only inside accent gradients (logos). Status: green #34d399, amber #fbbf24, red #f87171.

Type. Inter, system fallback. Base 14.5px / 1.5. Headings get tight tracking (-0.01em to -0.02em). Small labels are uppercase, ~0.78rem, 0.06–0.08em letter-spacing, in the muted tone. Numbers always use tabular figures. Antialiased.

Spacing & shape. Spacing scale of 4/8/12/16/24/32. Radii: 12px panels, 8px controls/inputs, 6px chips, 999px for bars. Two shadow tiers — subtle 0 2px 8px /35% for cards, lifted 0 8px 24px /45% for floating things. Active accents may
add a tinted glow (rgba(79,140,255,0.35)).

Layout. Sticky 240px sidebar + fluid main. On mobile the sidebar collapses to a horizontal top nav. Each page opens with a page-header (title + small meta on the right, divider underneath). Content lives inside card-panel blocks — never
floating directly on the bg.

Components.
- Stat card: tiny uppercase label with icon → big tabular number → muted sub-line. Border lightens on hover.
- Sidebar nav-item: icon + label, transparent base, accent-soft background when active. Destructive items shift to red on hover.
- Segmented control: pill group inside a panel-2 shell; active pill is solid accent with glow.
- Table: transparent rows, uppercase muted column headers, tabular nums right-aligned, 2% white row-hover.
- Bar (disk): 6px gradient fill, threshold-based color shift at 75% / 90%.
- Chip/folder pill: panel-2 with border, small inline icon, 6px radius.

Iconography. Bootstrap Icons (bi-*), inline with labels at slightly reduced opacity.

Motion. 0.15s ease for hover/active transitions; nothing splashy. Async sections fade to 50% opacity (is-loading); a single accent spinner for spinners.

Charts. Plotly with transparent bg, Inter font, 5% white gridlines, legend below the plot, custom dark hover label. Series colors come from a fixed 10-stop palette (blue → violet → green → amber → coral …) — never raw Plotly defaults.

Don'ts. No hard borders. No heavy drop shadows. No mixing accent colors for non-action UI. No raw Bootstrap-default buttons or tables. No solid color blocks for headers — keep them transparent over the panel.