# Architektur des Memory-Systems

Ein zentrales Memory-System läuft am besten als lokaler Dienst mit einer persistenter Datenbank (z.B. SQLite mit Vektorspeicher) und erlaubt allen KI-Clients – Claude-Code/-Flow genauso wie Codex CLI – Zugriff über eine API oder ein Protokoll. In modernen Open-Source-Lösungen (z.B. „LocalRecall“ oder dem MCP Memory Service) wird häufig ein **kombinierter Ansatz** verwendet: ein einfacher SQL-/Dateispeicher für Chat-Protokolle und parallel ein Vektor-Index (ChromaDB oder SQLite-vec) für semantische Erinnerungen[[1]](https://github.com/mudler/LocalRecall#:~:text=,driven%20applications%20requiring%20memory%20abstraction)[[2]](https://github.com/doobidoo/mcp-memory-service#:~:text=The%20MCP%20Memory%20Service%20supports,to%20suit%20different%20use%20cases). Diese Backends laufen lokal (keine Cloud-Abhängigkeit) und unterstützen WSL2/Linux. So kann etwa ein lokalen Speicher-Service (z.B. als Model Context Protocol-Server) eingerichtet werden, auf den alle KI-Tools per HTTP/JSON zugreifen – analog zum Beispiel „Memory Service MCP“[[3]](https://github.com/doobidoo/mcp-memory-service#:~:text=claude%20%2Fmemory,health)[[1]](https://github.com/mudler/LocalRecall#:~:text=,driven%20applications%20requiring%20memory%20abstraction). Wichtig ist auch eine **modulare Architektur**: Daten nach Themen oder Projekten („Collections“) getrennt halten (etwa per Tags oder separaten Datenbanken), um zielgerichtetes Abrufen zu erleichtern und Datenmischung zu vermeiden. Beispielhafte Kategorien könnten sein: code\_snippets, configurations, troubleshooting, project\_docs usw.[[4]](https://lobehub.com/mcp/vespo92-chromadblocal-mcp-server#:~:text=,automations%20Home%20Assistant%2C%20IoT%20scripts).

* **Speicher-Backend**: SQLite-vec (leichter Vektor-Speicher in SQLite) oder ChromaDB. SQLite-vec ist sehr ressourcenschonend, benötigt keine Netzwerkanbindung und eignet sich für Offlinespeicherung[[2]](https://github.com/doobidoo/mcp-memory-service#:~:text=The%20MCP%20Memory%20Service%20supports,to%20suit%20different%20use%20cases)[[1]](https://github.com/mudler/LocalRecall#:~:text=,driven%20applications%20requiring%20memory%20abstraction). ChromaDB ist ebenfalls Open Source und einfach zu benutzen, eignet sich vor allem für schnelle Prototypen[[5]](https://www.arsturn.com/blog/local-llm-persistent-memory-guide#:~:text=,great%20option%20for%20building%20LLM).
* **Datenmodell**: Jede gespeicherte Erinnerung enthält Text (Code, Notiz, Kommando-Ausgabe etc.), Metadaten (Tags wie Programmiersprache, Projekt, Typ) und den eingebetteten Vektor zur semantischen Suche[[5]](https://www.arsturn.com/blog/local-llm-persistent-memory-guide#:~:text=,great%20option%20for%20building%20LLM)[[4]](https://lobehub.com/mcp/vespo92-chromadblocal-mcp-server#:~:text=,automations%20Home%20Assistant%2C%20IoT%20scripts).
* **Multi-Client-Zugriff**: Ein lokaler Memory-Server (z.B. per Docker oder systemeigenem Dienst) kann universell angesprochen werden. Claude (über sein MCP-Protokoll) kann z.B. mit Befehlen wie /memory-store oder /memory-search direkt darauf schreiben und suchen[[3]](https://github.com/doobidoo/mcp-memory-service#:~:text=claude%20%2Fmemory,health). OpenAI-Modelle (Codex) nutzen dagegen Retrieval über Einbettungen: Hier liest ein Vorprozess oder LangChain-Agent die relevanten Daten aus der lokalen DB aus und fügt sie dann als Prompt-Kontext bei API-Aufrufen hinzu.
* **Versionierung und Backup**: Die Datenbank lässt sich via regelmäßigen Dumps oder integrierten Backup-Tools sichern. Viele Speicher-Systeme bieten Befehle wie create\_backup und optimize\_db[[6]](https://github.com/doobidoo/mcp-memory-service#:~:text=7.%20%60create_backup%60%20,Verify%20model%20status). Zur Versionierung kann man etwa Textdateien und Schema in Git halten oder Git-LFS für große Binärdateien nutzen. Regelmäßige Snapshots der Datenbank sichern gegen Datenverlust.

# Abruf-Strategien (Retrieval)

Der Kern des Systems ist ein **Retrieval-Augmented-Generation (RAG)**-Workflow: Bei einer Anfrage wird zuerst über Ähnlichkeitssuche in den lokalen Vektoren nach passenden Erinnerungen gesucht, bevor sie dem KI-Modell vorgelegt werden. Dabei helfen folgende Strategien:

* **Semantische Suche**: Eingehende Anfragen (oder Auszüge daraus) werden mit demselben Einbettungsmodell transformiert wie die gespeicherten Erinnerungen. Anschließend findet man per Kosinus- oder Distanz-Metrik die ähnlichsten Vektoren. So werden inhaltlich passende Erinnerungen ermittelt, nicht nur solche mit denselben Stichwörtern[[5]](https://www.arsturn.com/blog/local-llm-persistent-memory-guide#:~:text=,great%20option%20for%20building%20LLM).
* **Themen- und Projektauswahl**: Durch Tags oder geteilte „Collections“ lässt sich die Suche eingrenzen. Bei einer Frage zu *Python-Code* filtert man z.B. auf Erinnerungen mit dem Tag „Python“ oder in der Python-Collection. Dieses Vorausfiltern erhöht Relevanz und Effizienz (weil weniger Dokumente durchsucht werden). So kann man thematisch clusternd vorgehen – z.B. eigene Indizes für *UI-Probleme*, *Kommandozeilenausgaben*, *Architekturentscheidungen* etc. anlegen[[4]](https://lobehub.com/mcp/vespo92-chromadblocal-mcp-server#:~:text=,automations%20Home%20Assistant%2C%20IoT%20scripts).
* **Datenaufbereitung**: Lange Konversationen oder Code-Basen werden üblicherweise in kleine Abschnitte („Chunks“) zerlegt und einzeln indiziert. Ein Abschnitt sollte nicht zu lang sein (z.B. einige hundert Tokens), damit Suchanfragen präzise Teile zurückgeben können[[5]](https://www.arsturn.com/blog/local-llm-persistent-memory-guide#:~:text=,great%20option%20for%20building%20LLM).
* **Hybride Verfahren**: In manchen Fällen kombiniert man Vektor-Suche mit Stichwort-Filtern. Beispiel: Zunächst per Tag/Metadaten filtern (z.B. Projekt = X), dann auf diesen Teilreduktion semantisch suchen. Oder eine Volltext-/BM25-Suche parallel einsetzen, um bestimmte strukturierte Infos (z.B. Dateinamen oder exakte Fehlermeldungen) zu finden.
* **Retrieval-Optimierung**: Gute Systeme unterstützen auch zeitliche und genauere Abfragen – z.B. erinnert *Claude MCP* in natürlicher Sprache nach „letzter Woche“[[3]](https://github.com/doobidoo/mcp-memory-service#:~:text=claude%20%2Fmemory,health). Außerdem gibt es oft spezielle Such- und Lösch-APIs nach Tags oder Doppelungen, um das Wissen gezielt zu verwalten.

# Tokenkosten-Optimierung

Da bei API-Modellen (Claude, OpenAI) jeder gelieferte Kontext tokenbasiert abgerechnet wird, muss man darauf achten, **nur wirklich relevante Informationen zu senden**. Das heißt: Nicht das gesamte lokale Wissen geht an den Provider, sondern nur die Ergebnisse der Retrieval-Anfrage. Folgende Punkte sind wichtig:

* **Selektives Einbetten**: Nach der semantischen Suche zieht man nur die Top‑Ergebnisse (z.B. die 3–5 relevantesten Erinnerungen) heran, statt ganze Dokumente. So bleiben die Kontexttoken überschaubar. Ein einziger, großer Datenbank-Query kann sonst hunderte bis tausende Tokens umfassen und den Kontext rasch füllen[[7]](https://www.reddit.com/r/ClaudeAI/comments/1ltkm6c/use_sqlite_instead_of_markdown_files_to_give/#:~:text=Also%2C%20SQLite,lots%20of). Durch gezieltes Retrieval begrenzt man die Token-Anzahl.
* **Zusammenfassung**: Längere Textpassagen können vor dem Einfügen in den Prompt zusammengefasst werden – idealerweise vom LLM selbst oder einem kleineren Modell. Dadurch fasst man nur die wesentlichen Fakten in wenigen Sätzen zusammen, anstatt ganze Sitzungsprotokolle zu senden.
* **Kontext-Management**: Kurzfristige Gesprächskontexte (letzte Unterhaltungen) hält man ggf. im Agenten-Cache oder minimalistischen Speicher. Nur wirklich wichtige, langfristige „Erinnerungen“ werden in der Datenbank abgelegt. Im Prompt wird dann oft nur eine knappe Konversationserinnerung plus ausgewählte Langzeitgedächtnis-Fakten kombiniert.
* **Chunking und Overlap**: Wenn Code oder Dokumente chunkweise gespeichert sind, kann man bei einer konkreten Anfrage nur die relevanten Abschnitte heranziehen (z.B. eine bestimmte Funktion, anstatt das ganze Modul). Das reduziert unnötige Token.
* **Intelligente Retrieval-Module**: Manche Systeme setzen LLMs ein, die schon vor der großen API-Abfrage entscheiden, welche Speicher-Abfrage nötig ist – so rufen sie nur bei Bedarf inhaltliche Daten aus der Datenbank ab, anstatt pauschal alles mitzuschicken.

Insgesamt gilt: **Lokales Wissen wird nicht automatisch übertragen**, sondern nur bei Bedarf „gerufen“. Ein solides Design stellt sicher, dass pro API-Aufruf möglichst wenige, aber gezielt relevante Tokens übermittelt werden, um Kosten zu sparen[[7]](https://www.reddit.com/r/ClaudeAI/comments/1ltkm6c/use_sqlite_instead_of_markdown_files_to_give/#:~:text=Also%2C%20SQLite,lots%20of)[[3]](https://github.com/doobidoo/mcp-memory-service#:~:text=claude%20%2Fmemory,health).

# API-Kompatibilität

Das Memory-System muss mit allen eingesetzten Modellen und Tools zusammenarbeiten. Wichtige Aspekte sind:

* **Claude (Anthropic)**: Unterstützt das Model Context Protocol (MCP). Open-Source-Server wie *MCP Memory Service* bieten einen prägnanten Befehlssatz: So kann man z.B. mit claude /memory-store "…", /memory-search oder /memory-recall lokal Informationen speichern und abrufen[[3]](https://github.com/doobidoo/mcp-memory-service#:~:text=claude%20%2Fmemory,health). Diese Befehle landen auf dem lokalen Memory-Server (kein Cloud-Transfer), der die Abfragen an die SQLite/Chroma-Datenbank weiterleitet. Die Integration erfolgt über die üblichen Konfigurationsdateien (z.B. claude\_desktop\_config.json) und erfordert nur die IP/URL des lokalen Servers.
* **OpenAI (Codex)**: Hat kein eingebautes Persistenz-API, daher wird der Workflow typischer via Skript oder Agent realisiert. Vor einem Codex-Aufruf muss ein externer Schritt (z.B. in Python mit LangChain oder einem Shell-Skript) die Anfrage durch den Memory-Store jagen, relevante Erinnerungen abfragen (z.B. über eine REST-Schnittstelle) und dann das Ergebnis in den Prompt einbauen. Tools wie *LocalAI* oder *Haystack* können hier als Schnittstelle dienen. Wichtig ist, dass sämtliche Kommunikation (z.B. Embedding-Berechnung) lokal passiert. Selbst wenn man OpenAI-Embeddings nutzt, können diese durch einen lokalen Proxy (LocalAI) ersetzt werden, um das Wissen vor Ort zu halten.
* **Andere Clients**: Falls weitere Agenten oder IDE-Plugins (Cursor, VSCode-Extensions etc.) zum Einsatz kommen, bieten viele von ihnen ebenfalls MCP-Unterstützung oder einfache HTTP-APIs für Wissensabruf. Eine zentrale Speicher-API (z.B. als Docker-Service) kann universell angesprochen werden. Der Memory-Service-Beispielcode [36] zeigt, dass über das standardisierte MCP-Protokoll oder HTTP jeder Client auf denselben Speicher zugreifen kann, sodass Lernstoff plattformübergreifend verfügbar ist.
* **Offen vs. lokal**: Alle vorgeschlagenen Tools sind Open Source und plattformübergreifend. Docker-Container (für Chroma oder LocalRecall), Python-Pakete (für SQLite-vec) und Services (FastAPI/Go) laufen unter WSL2 oder nativen Linux-Umgebungen. Cloud-Abhängigkeiten entfallen vollständig.

# Organisation und Backup

Die optimale Struktur des Wissens hängt vom Anwendungsfall ab, aber einige Best Practices haben sich herauskristallisiert:

* **Strukturierung nach Kategorien**: Legen Sie feste Sammlungen oder Tags an (z.B. „Projekt A – Backend“, „Projekt A – UI“, „Allgemeine Hilfe“, „Dokumentation“). Trennen Sie etwa **Programmiersprachen**, Benutzeroberflächenthemen oder *Domänenwissen* in eigene Tag-Gruppen. Solche Kategorien (vgl. code\_snippets, troubleshooting, project\_docs, learning\_notes[[4]](https://lobehub.com/mcp/vespo92-chromadblocal-mcp-server#:~:text=,automations%20Home%20Assistant%2C%20IoT%20scripts)) erhöhen die Treffergenauigkeit beim Suchen.
* **Granularität**: Bewahren Sie kleinere, in sich abgeschlossene Einträge auf (Fehlerlösungen, Architekturentscheidungen, Codebeispiele) statt gigantischer Dokumente. Das erleichtert semantische Suche. Jedes Dokument sollte mit Metadaten versehen sein (z.B. „Sprache: Python“, „Projekt: X“, „Typ: UI-Bugfix“), damit man gezielt filtern kann.
* **Versionskontrolle**: Da bis zu 100 GB Daten anfallen können, empfiehlt es sich, das **Rohmaterial** (z.B. Quelltexte, Meeting-Notizen, Dokumentationen) in Git zu halten, während der Vektorindex als abgeleitetes, evtl. auslagerbares Objekt behandelt wird. Für Textressourcen kann man Git oder Git-LFS verwenden. Eine andere Option: Exportieren Sie regelmäßig die Datenbank (z.B. als JSON- oder CSV-Dump) und versionieren diese Snapshots[[6]](https://github.com/doobidoo/mcp-memory-service#:~:text=7.%20%60create_backup%60%20,Verify%20model%20status). Einige Speicherlösungen bieten integrierte Backups (create\_backup) und Optimierungen, die automatisiert werden können[[6]](https://github.com/doobidoo/mcp-memory-service#:~:text=7.%20%60create_backup%60%20,Verify%20model%20status).
* **Datensicherung**: Neben Git-Backups sollten Sie regelmäßige Sicherungen der Datenbank selbst erstellen. Viele Systeme (siehe oben) unterstützen optimize\_db und create\_backup[[6]](https://github.com/doobidoo/mcp-memory-service#:~:text=7.%20%60create_backup%60%20,Verify%20model%20status). Speichern Sie Backups idealerweise an einem sicheren Ort (externe Festplatte, NAS oder verschlüsseltes Cloud-Archiv, je nach Sicherheitsanforderung).
* **Wartung und Qualität**: Überprüfen Sie Inhalte gelegentlich auf Veraltung oder Fehler. Einige fortschrittliche Memory-Systeme bieten „Memory Consolidation“ oder intelligente Clustering-Tools, die Einträge komprimieren und Doppelungen entfernen (z.B. Dream-Inspired Consolidation[[8]](https://github.com/doobidoo/mcp-memory-service#:~:text=,term%20memory%20management)). Selbst wenn Sie keine KI-basierte Konsolidierung einsetzen, kann man manuelle Reviews einplanen: Etwa alte Einträge archivieren oder Tags anpassen, um die Suche präzise zu halten.

**Fazit:** Ein modernes, produktionsreifes Gedächtnissystem setzt auf eine lokale Kombination aus SQL/Dateispeicher und Vektor-Datenbank (wie SQLite-vec oder Chroma)[[1]](https://github.com/mudler/LocalRecall#:~:text=,driven%20applications%20requiring%20memory%20abstraction)[[5]](https://www.arsturn.com/blog/local-llm-persistent-memory-guide#:~:text=,great%20option%20for%20building%20LLM). Durch semantische Retrieval-Strategien, kluge Token-Filterung und konsistente Organisation (Tags/Sammlungen) lassen sich Kosten minimieren und die Treffergenauigkeit maximieren. Open-Source-Tools wie LocalRecall, verschiedene MCP-Server oder LangChain-Frameworks bieten die nötigen Bausteine. Mit Versionskontrolle der Quellelemente und regelmäßigen Backups (DB-Dumps) ist das System robust und erweiterbar – perfekt für den Einsatz unter WSL2/Linux ohne Cloud-Dependency[[1]](https://github.com/mudler/LocalRecall#:~:text=,driven%20applications%20requiring%20memory%20abstraction)[[6]](https://github.com/doobidoo/mcp-memory-service#:~:text=7.%20%60create_backup%60%20,Verify%20model%20status).

**Quellen:** Moderne Open-Source-Ansätze zeigen, dass lokale Memory-Server mit SQLite/Vektor-DB effektiv funktionieren[[1]](https://github.com/mudler/LocalRecall#:~:text=,driven%20applications%20requiring%20memory%20abstraction)[[2]](https://github.com/doobidoo/mcp-memory-service#:~:text=The%20MCP%20Memory%20Service%20supports,to%20suit%20different%20use%20cases), ebenso Empfehlungen zu Chroma/FAISS[[5]](https://www.arsturn.com/blog/local-llm-persistent-memory-guide#:~:text=,great%20option%20for%20building%20LLM) und Organisation nach Sammlungen[[4]](https://lobehub.com/mcp/vespo92-chromadblocal-mcp-server#:~:text=,automations%20Home%20Assistant%2C%20IoT%20scripts). Praxismeldungen weisen allerdings auf Tokenfresser bei unbedachter Nutzung hin[[7]](https://www.reddit.com/r/ClaudeAI/comments/1ltkm6c/use_sqlite_instead_of_markdown_files_to_give/#:~:text=Also%2C%20SQLite,lots%20of), weshalb selektiver Kontext und Clustering empfohlen werden. Die zitierten Projekte und Dokumentationen (MCP-Server, LocalRecall, u.a.) belegen diese Best Practices in der Praxis[[3]](https://github.com/doobidoo/mcp-memory-service#:~:text=claude%20%2Fmemory,health)[[6]](https://github.com/doobidoo/mcp-memory-service#:~:text=7.%20%60create_backup%60%20,Verify%20model%20status).

[[1]](https://github.com/mudler/LocalRecall#:~:text=,driven%20applications%20requiring%20memory%20abstraction) GitHub - mudler/LocalRecall: :brain: 100% Local Memory layer and Knowledge base for agents with WebUI

<https://github.com/mudler/LocalRecall>

[[2]](https://github.com/doobidoo/mcp-memory-service#:~:text=The%20MCP%20Memory%20Service%20supports,to%20suit%20different%20use%20cases) [[3]](https://github.com/doobidoo/mcp-memory-service#:~:text=claude%20%2Fmemory,health) [[6]](https://github.com/doobidoo/mcp-memory-service#:~:text=7.%20%60create_backup%60%20,Verify%20model%20status) [[8]](https://github.com/doobidoo/mcp-memory-service#:~:text=,term%20memory%20management) GitHub - doobidoo/mcp-memory-service: Universal MCP memory service with semantic search, multi-client support, and autonomous consolidation for Claude Desktop, VS Code, and 13+ AI applications

<https://github.com/doobidoo/mcp-memory-service>

[[4]](https://lobehub.com/mcp/vespo92-chromadblocal-mcp-server#:~:text=,automations%20Home%20Assistant%2C%20IoT%20scripts) ChromaDB MCP Server | MCP Servers · LobeHub

<https://lobehub.com/mcp/vespo92-chromadblocal-mcp-server>

[[5]](https://www.arsturn.com/blog/local-llm-persistent-memory-guide#:~:text=,great%20option%20for%20building%20LLM) How to Give Your Local LLM a Persistent Memory | Arsturn

<https://www.arsturn.com/blog/local-llm-persistent-memory-guide>

[[7]](https://www.reddit.com/r/ClaudeAI/comments/1ltkm6c/use_sqlite_instead_of_markdown_files_to_give/#:~:text=Also%2C%20SQLite,lots%20of) Use SQLite instead of markdown files to give Claude Code more enhanced memory. : r/ClaudeAI

<https://www.reddit.com/r/ClaudeAI/comments/1ltkm6c/use_sqlite_instead_of_markdown_files_to_give/>