BKK Atomium

KI-Assistent

Technische Dokumentation

Systemarchitektur · Komponenten · Datenflüsse

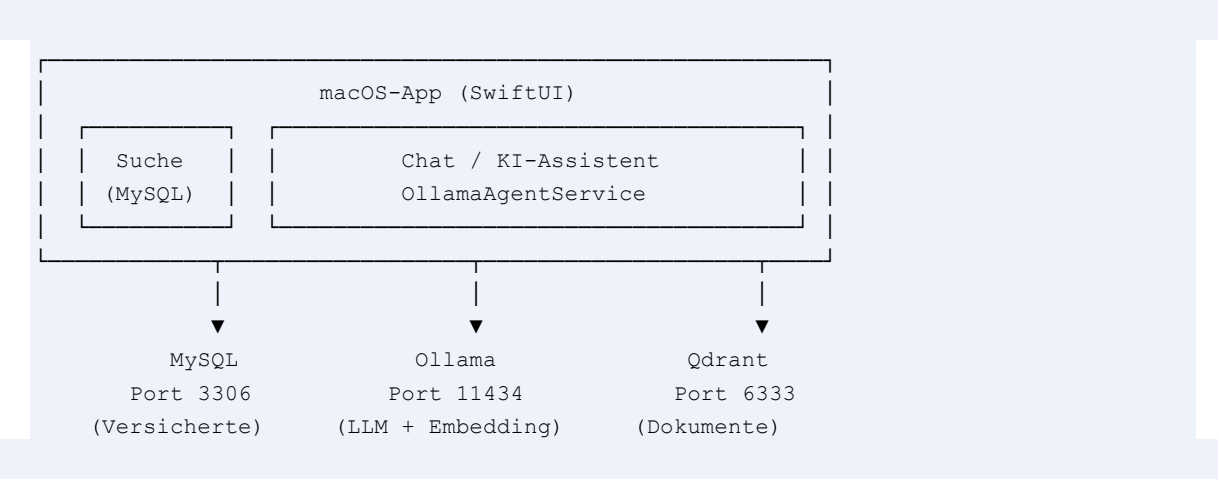
macOS-App · Claude Code MCP-Erweiterung

Zielgruppe: Entwickler und Architekten

Inhaltsverzeichnis

1. Systemüberblick

Ein vollständig **lokal laufender KI-Assistent** für eine gesetzliche Krankenversicherung. Alle Komponenten laufen auf demselben Mac — kein Daten-Upload in die Cloud, kein Internet-Zugriff während der Nutzung.



Dienst	Zweck	Port
MySQL	Versichertendaten (13 Tabellen, 1.000 Demo-Versicherte)	3306
Ollama + Gemma4	Lokales Sprachmodell (LLM) + natives Tool-Calling	11434
nomic-embed-text	Embedding-Modell für die semantische Suche	— (via Ollama)
Qdrant	Vektordatenbank für Gesetze, Richtlinien, FAQ und Versichertendokumente	6333

Die macOS-App in Aktion

Architektur-Überblick

[Abbildung: Architektur-Übersicht]

Programmstart — Systemstatus

Beim Start prüft die App automatisch die Erreichbarkeit aller Backends und zeigt einen farbcodierten Status.

[Abbildung: Programmstart und Systemübersicht]

Einstellungen

Alle Verbindungsparameter sind zur Laufzeit konfigurierbar — kein Rebuild erforderlich.

[Abbildung: Einstellungen]

Versichertensuche

Direkte MySQL-Abfrage nach Name, KVNR oder Geburtsdatum in Echtzeit.

[Abbildung: Suche]

Detailansicht

Alle Versichertendaten in einer tabellarischen Übersicht mit Tab-Navigation.

[Abbildung: Detailansicht]

KI — Stammdatenabfrage

Der Assistent erkennt die KVNR im Freitext und ruft die Daten automatisch ab.

[Abbildung: KI — Stammdaten]

KI — Gesetzesabfragen (RAG)

Fragen zu SGB V, SGB XI oder internen Richtlinien werden aus der Vektordatenbank beantwortet.

[Abbildung: KI — Gesetze]

KI — Kombinierte Abfragen

Komplexe Fragen kombinieren Datenbankabfrage und RAG-Suche in einem Gesprächsschritt.

[Abbildung: KI — Kombinierte Abfrage (1)]

[Abbildung: KI — Kombinierte Abfrage (2)]

Claude Code MCP-Erweiterung

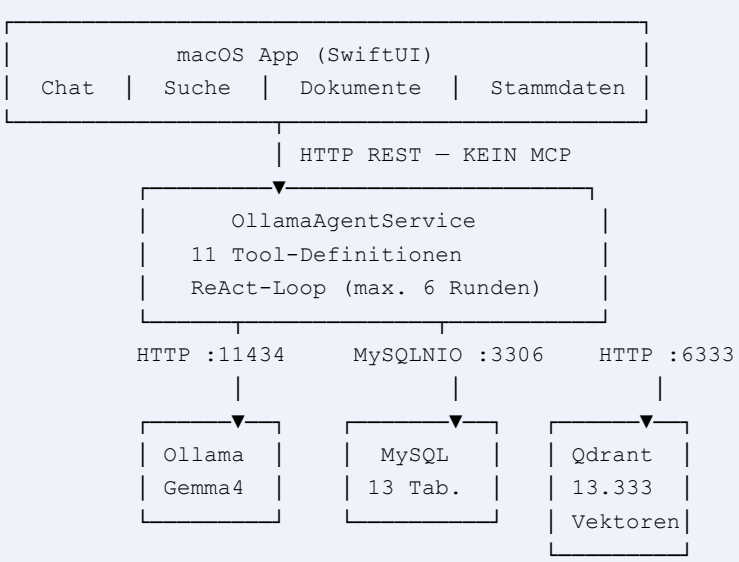
Dieselben Daten sind auch direkt aus Claude Code über MCP-Server abrufbar.

[Abbildung: Claude Code MCP-Erweiterung]

2. Architektur — zwei getrennte Systeme

Das Projekt enthält **zwei vollständig voneinander unabhängige Systeme** mit getrennten Code-Pfaden. Sie teilen dieselben Backends, nutzen aber unterschiedliche Kommunikationswege.

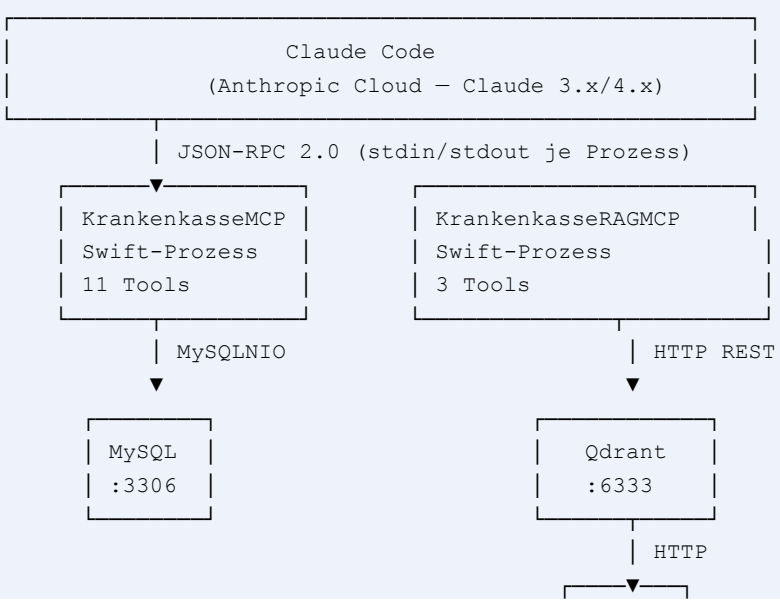
System 1: macOS-App (für Sachbearbeiter)

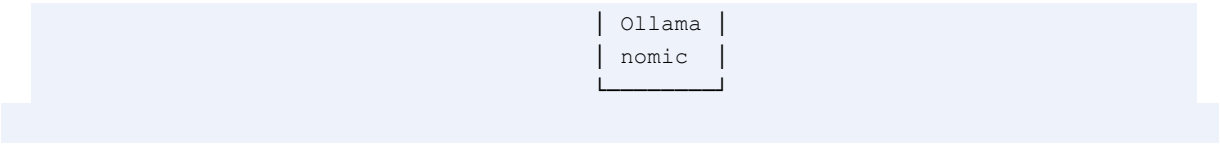


Kommunikation: Die App spricht direkt per HTTP mit Ollama, per MySQLNIO-Binary-Protocol mit MySQL und per HTTP-REST mit Qdrant. **Kein MCP-Protokoll, kein Zwischenprozess.**

Das Sprachmodell (Gemma4) antwortet bei Bedarf mit einem strukturierten `tool_calls`-Objekt statt mit Text. Der `OllamaAgentService` wertet dieses aus, führt den entsprechenden Datenbankaufruf oder die RAG-Suche aus und schickt das Ergebnis in der nächsten Runde zurück an Ollama.

System 2: Claude Code Terminal (für Entwickler und Analysten)





Kommunikation: Claude Code startet die MCP-Server als Kindprozesse und kommuniziert über **JSON-RPC 2.0 auf stdin/stdout**. Jede Zeile ist eine JSON-RPC-Nachricht. Die MCP-Server sind zustandslos — sie halten keine Gesprächshistorie.

Vergleich

Merkmal	macOS App	Claude Code
Endbenutzer	Sachbearbeiter	Entwickler / Analyst
KI-Modell	Gemma4 (lokal, Ollama)	Claude Sonnet/Opus (Anthropic Cloud)
Tool-Mechanismus	Natives `tool_calls` via Ollama HTTP API	MCP — JSON-RPC 2.0 via stdin/stdout
Datenbankzugriff	Direkt: MySQLNIO + HTTP	Über MCP-Serverprozesse
MCP genutzt	**NEIN**	**JA**
Offline-fähig	**JA** — vollständig lokal	Nein — Claude benötigt Internet

3. Komponenten

MySQL 8.x — Versichertendatenbank

Hersteller	Oracle Corporation
Website	mysql.com
Lizenz	GPL v2 (Community Edition)
Swift-Treiber	MySQLNIO (vapor.codes/mysql-nio)

Relationale Datenbank für alle strukturierten Versichertendaten. 13 Tabellen mit Foreign Keys, ENUM-Typen und `utf8mb4`. Alle Abfragen sind parametrisiert — SQL-Injection ist technisch ausgeschlossen. Der Swift-Treiber MySQLNIO ist asynchron und non-blocking (SwiftNIO-basiert) und integriert sich direkt in `async/await`.

Ollama — lokaler KI-Server

Hersteller	Ollama Inc.
Website	ollama.com
Quellcode	github.com/ollama/ollama
Lizenz	MIT
API	OpenAI-kompatibel (`/api/chat`, `/api/embeddings`)

Ollama lädt große Sprachmodelle lokal und stellt sie per HTTP bereit. Auf Apple-Silicon-Macs nutzt es das Metal-Backend für GPU-Beschleunigung. Das native Tool-Calling wird über das `tools`-Feld im Chat-Request übermittelt: Das Modell antwortet mit `tool_calls` (strukturiertes JSON) statt mit Text, wenn es externe Daten benötigt.

> **Installation:** Offizielle App von ollama.com verwenden — **nicht** `brew install ollama`. Die Homebrew-Version enthält in manchen Versionen ein fehlendes `llama-server`-Binary.

Gemma4 — Sprachmodell

Hersteller	Google DeepMind
Modell-Hub	huggingface.co/google/gemma-3
Lizenz	Gemma Terms of Use
Eingesetzt als	`gemma4:latest` (Q4_K_M-Quantisierung)

Kontextfenster	32.768 Token

Open-Weights-Sprachmodell das lokal auf dem Mac läuft. Unterstützt **natives Tool-Calling** über die Ollama-API — das Modell gibt strukturierte `tool_calls`-Objekte zurück, die ohne String-Parsing direkt verarbeitet werden. Trainings-Cutoff ca. Ende 2023; neuere Gesetze werden über RAG nachgeliefert.

---

nomic-embed-text — Embedding-Modell

Hersteller	Nomic AI Inc.
Website	nomic.ai
Modell-Hub	huggingface.co/nomic-ai/nomic-embed-text-v1
Lizenz	Apache 2.0
Größe	274 MB
Vektor-Dimensionen	768

Wandelt Text in 768-dimensionale Zahlenvektoren um die die semantische Bedeutung repräsentieren. Ähnliche Texte erzeugen ähnliche Vektoren — unabhängig von den exakten Wörtern. Wird für zwei Operationen eingesetzt: (1) beim Dokumentenimport jeden Textchunk einbetten, (2) bei jeder Suche die Anfrage einbetten und den ähnlichsten Vektor in Qdrant suchen. Das Modell erzeugt ein Embedding in 50–200 ms und belegt permanent ~274 MB RAM.

> **Kritisch:** Das Embedding-Modell kann nie gewechselt werden, ohne alle Vektoren in Qdrant neu zu berechnen (`./rag.sh import:reset`), da verschiedene Modelle inkompatible Vektordimensionen erzeugen.

Qdrant — Vektordatenbank

Hersteller	Qdrant Solutions GmbH (Berlin)
Website	qdrant.tech
Quellcode	github.com/qdrant/qdrant
Lizenz	Apache 2.0
Implementiert in	Rust
API	REST/JSON auf Port 6333

Spezialdatenbank für Vektoren mit HNSW-Index (Hierarchical Navigable Small World). Sucht in O(log n) den ähnlichsten Vektor über Cosine-Similarity. Payload-Filter ermöglichen es, die Suche auf eine bestimmte Sammlung oder KVNR einzuschränken — grundlegende Datenisolation zwischen Versicherten.

Aktueller Stand: 13.333 Vektoren in einer Collection `krankenkasse_wissen`.

Sammlung	Chunks
`gesetze` (SGB I/IV/V/IX/X/XI/XII)	13.224
`faq`	72
`richtlinien`	25
`versicherte` (KVNR-gefiltert)	12

Swift 6 + SwiftUI — App-Technologie

Hersteller	Apple Inc.
Website	swift.org / developer.apple.com/xcode/swiftui
Lizenz	Apache 2.0 (Swift)
Mindest-macOS	14.0 (Sonoma)

Swift 6.0 führt **Strict Concurrency Checking** ein: Der Compiler erzwingt threadsicheren Code und verhindert Race Conditions zur Compile-Zeit — wichtig für einen Agenten der gleichzeitig Datenbankabfragen, HTTP-Anfragen und UI-Updates ausführt. SwiftUI rendert deklarativ aus dem aktuellen Zustand; `@Observable` (iOS 17+/macOS 14+) ersetzt `@Published`-Properties mit einem präziseren Beobachtungsmodell.

MCP — Model Context Protocol

Entwickler	Anthropic (initiiert), offener Standard
Website	modelcontextprotocol.io
Spezifikation	github.com/modelcontextprotocol/specification
Lizenz	MIT
Transport (eingesetzt)	`stdio` — JSON-RPC 2.0 auf stdin/stdout

Offenes Protokoll für KI-Tool-Integration. Ein MCP-Server meldet beim Handshake seine Tools (`tools/list`); der Client (Claude Code) ruft sie bei Bedarf auf (`tools/call`). Das Protokoll ist modellunabhängig — dieselben MCP-Server funktionieren mit Claude, GPT-4o, Cursor und jedem anderen MCP-kompatiblen Client.

4. macOS-App — technischer Aufbau

Architektur: MVVM mit Service-Layer

```
┌──────────────────────────────────┐
│          Views (SwiftUI)         │  rendert nur den aktuellen Zustand
└──────────────┬───────────────────┘
               │ @Observable (beobachtet)
┌──────────────▼───────────────────┐
│     ViewModels (@MainActor)      │  hält UI-Zustand, delegiert Arbeit
└──────────────┬───────────────────┘
               │ async/await
┌──────────────▼───────────────────┐
│     OllamaAgentService           │  ReAct-Loop, Tool-Calling, RAG
│     DatabaseService              │  MySQLNIO-Verbindung
│     DocumentImportService        │  PDFKit + Chunking + Embedding
└──────────────────────────────────┘
```

56 Swift-Dateien in klarer Schichtenstruktur. Views enthalten keine Logik. ViewModels enthalten keine Datenbankabfragen. Services enthalten keine UI-Abhängigkeiten.

OllamaAgentService — das Kernstück

Der `OllamaAgentService` implementiert den **ReAct-Loop** (Reason + Act):

```
Benutzerfrage
      │
      ▼
Runde 1..6: POST /api/chat → Ollama (Gemma4)
      │
      ├── Antwort enthält tool_calls?
      │       │
      │       ▼ JA
      │   Werkzeug ausführen (MySQL oder Qdrant)
      │   Ergebnis als role:"tool" anhängen
      │   → nächste Runde
      │
      └── Antwort enthält content (Text)?
              │
              ▼
          Finale Antwort → ChatView

Falls alle 6 Runden ohne Textantwort:
  → Synthese-Runde ohne Werkzeuge erzwingt Zusammenfassung
```

Gesprächshistorie: Jeder `ask()`-Aufruf erhält System-Prompt + bisherige Historie + neue Frage als vollständigen Kontext. Die Historie ist auf 20 Einträge begrenzt — ältere Einträge werden entfernt.

Werkzeug-Dispatch: Anhand des `name`-Felds im `tool_calls`-Objekt wird die passende Datenbankabfrage oder RAG-Suche aufgerufen. Unbekannte Werkzeugnamen werden abgefangen; Datenbankfehler werden als Fehlertext an das Modell zurückgegeben statt den Loop abzubrechen.

Direkter Datenbankzugriff per MySQLNIO

Kein ORM, kein Object-Mapping — direkte SQL-Abfragen mit Parameter-Binding. Jede Abfrage hat ein dediziertes DAO (Data Access Object) das die SQL-Logik kapselt und das Ergebnis in ein Swift-Model umwandelt.

```
DatabaseService (Verbindung + query())
  └── PersonDAO          → SELECT aus person
  └── CareDAO            → JOIN pflege + pflegegrad + person
  └── ContributionDAO    → SELECT aus beitrag
  └── BenefitDAO         → JOIN leistung + leistungsart
  └── ... (9 DAOs gesamt)
```

Dokumentenimport nativ in Swift

Der Import von PDF- und Textdokumenten in die Wissensdatenbank erfolgt vollständig in Swift — **kein Python in der App**:

```
PDF/TXT/MD
  └── PDFKit (PDF) / String(contentsOf:) (Text)
        └── Text in Chunks aufteilen (500 Zeichen, 50 Überlappung)
              └── POST /api/embeddings → Ollama (nomic-embed-text)
                    └── 768-dimensionaler Vektor
                          └── PUT /collections/.../points → Qdrant
```

Das `rag.sh`-Skript für den CLI-Import nutzt weiterhin Python (`importer.py`).

Verbindungsparameter (UserDefaults-persistent)

Dienst	Standard
MySQL	`127.0.0.1:3306` / User: `kasse` / DB: `krankenkasse`
Ollama	`localhost:11434` / Modell: `gemma4:latest`
Ollama Embedding	`nomic-embed-text`
Qdrant	`localhost:6333`
Temperature	0.7 (Chat)
Max. Runden	6
Max. Kontexttokens	8.192
Timeout	120 Sekunden

5. Wissensdatenbank (RAG-Pipeline)

Dokumentfluss beim Import

```
rag-documents/
  richtlinien/   (TXT)
  faq/           (TXT)
  gesetze/       (TXT — via gesetze.sh von gesetze-im-internet.de)
  versicherte/   (TXT/PDF — je Verzeichnis eine KVNR)
        |
        ▼
  Python-Importer (rag-importer/importer.py)
  oder Swift DocumentImportService
        |
        |  Text in Chunks aufteilen
        |  Chunk-Größe: 500 Zeichen / Überlappung: 50 Zeichen
        |
        ▼
  POST /api/embeddings → nomic-embed-text
        |
        |  768-dimensionaler Float-Vektor
        |
        ▼
  PUT /collections/krankenkasse_wissen/points → Qdrant
  Payload: { text, quelle, sammlung, versicherungsnummer }
```

Suchstrategie — parallele kollektionssensitive Suche

Da 13.224 Gesetze-Chunks (98,4 %) die 97 FAQ- und Richtlinien-Chunks verdrängen würden, erfolgt die Suche in **drei parallelen Teilanfragen** mit eigenen Limits:

```
Anfrage-Embedding (Ollama)
        |
        ├── Qdrant: sammlung=faq         limit=4
        ├── Qdrant: sammlung=richtlinien limit=2
        └── Qdrant: sammlung=gesetze     limit=3
                |
                ▼
        Ergebnisse zusammenführen
        Deduplizieren (erste 80 Zeichen)
        Auf 6 Treffer begrenzen
        Als Kontext an Ollama übergeben
```

Gesetzestexte — automatischer Download

```
./gesetze.sh          # Alle Gesetze laden + importieren
./gesetze.sh sgb_11   # Nur SGB XI
```

Alle deutschen Bundesgesetze sind gemeinfrei (§ 5 UrhG) und werden von **gesetze-im-internet.de** (Bundesministerium der Justiz) als XML heruntergeladen, in Plain Text konvertiert und importiert.

Gesetz	Inhalt
SGB I	Allgemeiner Teil
SGB IV	Gemeinsame Vorschriften (Beiträge, Meldepflichten)
SGB V	Gesetzliche Krankenversicherung
SGB IX	Rehabilitation und Teilhabe
SGB X	Sozialverwaltungsverfahren, Datenschutz
SGB XI	Soziale Pflegeversicherung
SGB XII	Sozialhilfe

6. Claude Code Erweiterung via MCP

Aufbau: zwei eigenständige Swift-Prozesse

```
KrankenkasseMCP              KrankenkasseRAGMCP
──────────────────           ──────────────────
Package.swift                Package.swift
  └── MySQLNIO                 └── (keine ext. Deps)
  └── swift-log

main.swift                   main.swift
  Bootstrap:                   Bootstrap:
  DB-Verbindung                HTTP-Clients
  VersicherterService          RAGService
  ToolRegistry                 ToolRegistry
  MCPServer                    MCPServer
```

Beide Server sind **eigenständige ausführbare Programme** die von Claude Code bei Bedarf als Kindprozesse gestartet werden. Sie halten die Verbindung solange Claude Code läuft.

JSON-RPC 2.0 Nachrichtenfluss

```
Claude Code                          MCP-Server
     |                                    |
     |── initialize ──────────────────►   |
     |◄── {protocolVersion, capabilities} |
     |                                    |
     |── tools/list ──────────────────►   |
     |◄── [{name, description,            |
     |       inputSchema}, ...]           |
     |                                    |
     |── tools/call ──────────────────►   |
     |   {name: "get_versicherter",       |
     |    arguments: {versicherungsnummer |
     |                : "M000000014"}}    |
     |                                    |
     |           [MySQL-Abfrage]          |
     |                                    |
     |◄── {content: [{type: "text",       |
     |                text: "..."}]}      |
```

Jede Nachricht ist **eine JSON-Zeile** auf stdin/stdout. `fflush(stdout)` nach jeder Antwort ist notwendig, damit Claude Code die Antwort sofort empfängt.

KrankenkasseMCP — Stammdaten (11 Tools)

Tool	Datenbankabfrage
`get_versicherter`	`person`
`get_adressen`	`adresse`

`get_telefonnummern`	`telefonnummer`
`get_bankdaten`	`bankverbindung`
`get_leistungen`	`leistung` JOIN `leistungsart`
`get_leistungsarten`	`leistungsart`
`get_beitraege`	`beitrag`
`get_krankmeldungen`	`krankmeldung`
`get_pflegedaten`	`pflege` JOIN `pflegegrad`
`get_pflegegrade_referenz`	`pflegegrad` (alle Einträge)
`get_beschaeftigung`	`beschaeftigung` JOIN `arbeitgeber`

KrankenkasseRAGMCP — Wissensdatenbank (3 Tools)

Tool	Funktion
`search_knowledge_base`	Suche in allen Sammlungen (faq + richtlinien + gesetze)
`search_policy_documents`	Suche in Dokumenten eines bestimmten Versicherten (KVNR-Filter)
`list_knowledge_collections`	Listet verfügbare Sammlungen auf

Tool-Architektur: MCPTool-Protokoll

Beide Server folgen demselben Architekturmuster. Jedes Tool implementiert ein gemeinsames Protokoll:

```
MCPTool (Protokoll)
  ├── name          → "get_versicherter"
  ├── description   → Beschreibung für Claude (entscheidet Tool-Auswahl)
  ├── inputProperties → Parameter mit Typ und Beschreibung
  ├── requiredInputs  → Pflichtfelder
  └── execute(arguments) → String (immer Text zurück)

MCPServer
  └── liest tools/list  → iteriert ToolRegistry
  └── ruft tools/call   → sucht Tool per Name, ruft execute() auf

ToolRegistry
  └── hält alle Tool-Instanzen
  └── tool(named:) → O(n)-Suche
```

Neue Tools hinzufügen: neue Klasse anlegen die `MCPTool` implementiert, in `ToolRegistry` eintragen — `MCPServer` und `main.swift` bleiben unverändert.

Wie Claude Tools auswählt

Claude liest beim Start alle Tool-Definitionen via `tools/list`. Die `description` ist entscheidend für die automatische Tool-Auswahl. Konkrete Beispiele in der Beschreibung erhöhen die Treffsicherheit erheblich:

```
"Zeigt alle Stammdaten einer versicherten Person anhand der
 Versicherungsnummer (KVNR)"
→ Claude wählt dieses Tool bei "Wer ist M000000014?"

"Durchsucht die interne Wissensdatenbank nach Richtlinien, Gesetzen und FAQ.
 Verwenden für Fachfragen, z.B. 'Welche Regel gilt bei Pflegegrad 3?'"
→ Claude wählt dieses Tool bei "Was sagt § 15 SGB XI?"
```

Registrierung in Claude Code

```
claude mcp add -s user krankenkasse     ./mcp/.build/release/KrankenkasseMCP
claude mcp add -s user krankenkasse-rag ./rag-mcp/.build/release/KrankenkasseRAGMCP
```

7. Datenbankschema

13 Tabellen in MySQL, vollständig mit Foreign Keys und `utf8mb4`.

```
person (KVNR PK)
  ├── adresse            (1:n)
  ├── telefonnummer      (1:n)
  ├── bankverbindung     (1:n)
  ├── beschaeftigung     (1:n) → arbeitgeber
  ├── krankmeldung       (1:n)
  ├── pflege             (1:n) → pflegegrad
  └── leistung           (1:n) → leistungsart
        leistungsart     (Lookup, 20 Einträge)
        pflegegrad       (Lookup, 5 Einträge: PG1–PG5 mit Beträgen)
        beitragsart      (Lookup)
  └── beitrag            (1:n) → beitragsart
```

KVNR-Format: Ein Großbuchstabe + 9 Ziffern (z.B. `M000006069`). Alle 11 Agent-Werkzeuge filtern ausschließlich per KVNR — `ON DELETE CASCADE` sorgt für automatische DSGVO-konforme Löschkaskaden.

8. Inbetriebnahme

Voraussetzungen

Voraussetzung	Version
macOS	14.0+ (Sonoma)
Swift / Xcode	6.0+
Python	3.10+ (für CLI-Import)
Ollama	aktuell (ollama.com/download — offizielle App)
pip: requests, pypdf	aktuell

Erstinstallation

```
./rag.sh install
```

Führt automatisch aus: Voraussetzungen prüfen → MySQL aufsetzen (1.000 Demo-Versicherte) → Qdrant laden → Ollama-Modelle ziehen → beide MCP-Server bauen → MCP in Claude Code registrieren.

Täglicher Start

```
./rag.sh start     # MySQL + Qdrant (Ollama läuft als Menu-Bar-App automatisch)
./rag.sh status    # Alle Backends prüfen
```

Gesetzestexte und Dokumente importieren

```
./gesetze.sh               # Alle SGB-Gesetze von gesetze-im-internet.de laden
./rag.sh import            # Alle Dokumente aus rag-documents/ importieren
./rag.sh import:reset      # Qdrant leeren + Neuimport
```

9. Sicherheitshinweis (Prototyp)

> **Das System ist ein Prototyp für Testdaten.** Es fehlen Authentifizierung, TLS-Verschlüsselung und Audit-Log. Der Einsatz mit echten Versichertendaten ist ohne zusätzliche Maßnahmen rechtlich nicht zulässig (DSGVO Art. 35, § 75b SGB X).

Was fehlt	Auswirkung
Authentifizierung	Jeder der die App öffnet hat vollen Datenzugriff
Passwort in Keychain	DB-Passwort liegt als Klartext in UserDefaults
TLS für MySQL + Qdrant	Verbindungen sind im Netzwerk unverschlüsselt
Audit-Log	Kein Nachweis wer wann welche KVNR abgefragt hat
Datenschutz-Folgenabschätzung	Pflicht vor Einsatz mit Echtdaten (Art. 35 DSGVO)

Was bereits schützt: App Sandbox, Hardened Runtime, parameterisierte SQL-Abfragen, vollständig lokaler Betrieb (kein Cloud-Upload), Debug-Logs nur im DEBUG-Build.