BKK Atomium

GesetzesChat – BKK Atomium KI-Chatbot

Produktbeschreibung und Entwicklungsgeschichte

Lokaler KI-Chatbot für Sozialversicherungsrecht

macOS-App · Ollama · Qdrant · Retrieval-Augmented Generation

Entwickelt mit Claude Code als Governance-Werkzeug

Christian Drapatz

Juli 2026

Inhaltsverzeichnis

Warum dieses Projekt entstanden ist

Der Ausgangspunkt war nicht die Idee, noch einen Chatbot zu bauen. Der Ausgangspunkt war eine andere Frage: **Wie entwickelt ein Team verantwortungsvoll Software mit KI-Unterstützung, ohne dass am Ende ein uneinheitlicher, schwer wartbarer Code-Flickenteppich entsteht?** Setzt jeder Entwickler eine KI wie Claude Code nach eigenem Ermessen ein, bekommt er tendenziell einen eigenen Architekturstil zurück. Bei einer einzelnen Person mag das noch tragbar sein. **Im Team wird daraus schnell Chaos in mehreren Architektursprachen gleichzeitig**.

Der KI-Chatbot ist die praktische Antwort auf genau diese Frage. Eine vollständig funktionierende App als Beweis, dass sich KI-gestützte Entwicklung mit einem klaren, erzwungenen Regelwerk seriös betreiben lässt, statt sich auf lose Konventionen und gute Vorsätze zu verlassen. Bewusst wurden dabei keine fertigen, aus dem Internet verfügbaren Skill-Ressourcen eingesetzt. Das komplette Projekt wurde zunächst durchdacht und geplant, erst danach umgesetzt.

Abbildung: Informationen zum Projekt

Das Projekt: ein konfigurierbarer KI-Chatbot für Regelwerke

Der KI-Chatbot ist im Kern eine Demo-App unter macOS, die Mitarbeitende in einem chat-artigen Programm bei Fragen zu einem Regelwerk unterstützt, zum Beispiel "Fragen zu Leistungen, Pflege und Mitgliedschaft" in der Krankenversicherung. Man stellt eine Frage zu einem konkreten Thema und bekommt mithilfe der KI eine Antwort, die auf tatsächlich hinterlegtem Wissen beruht statt auf freier Erfindung.

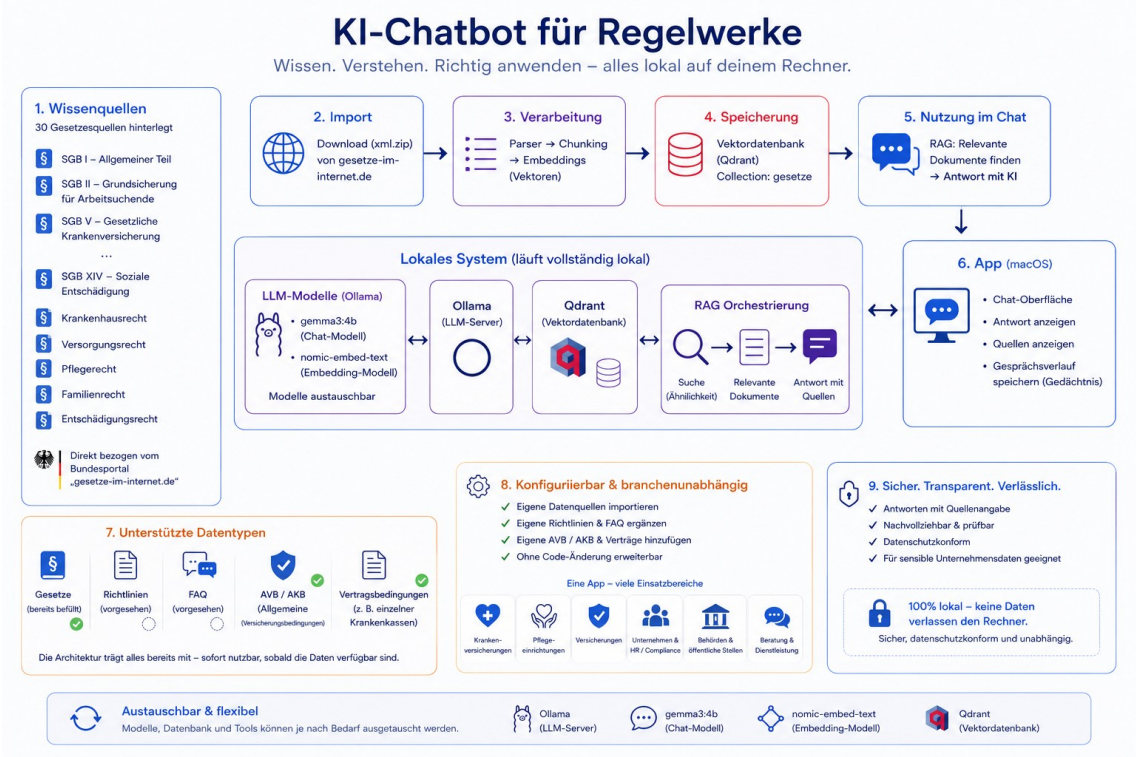

Abbildung: Gesetze als Datenbasis

Wichtig ist dabei die Konfigurierbarkeit. Die App ist nicht fest auf ein Gesetz oder eine Branche zugeschnitten, sondern lässt sich für viele Bereiche einsetzen. Als Datenquellen können Gesetze, Richtlinien und FAQ importiert werden, ebenso Allgemeine Versicherungsbedingungen (AVB/AKB). Je nach Einsatzgebiet lassen sich zusätzlich die Vertragsbedingungen einzelner Versicherungen oder Krankenkassen ergänzen. Aktuell sind 30 Gesetzesquellen hinterlegt, von den Sozialgesetzbüchern SGB I bis SGB XIV über Krankenhaus- und Versorgungsrecht bis zu Pflege-, Familien- und Entschädigungsrecht, alle direkt vom offiziellen Bundesportal gesetze-im-internet.de bezogen. Richtlinien und FAQ sind als Datentypen bereits vorgesehen, aber im aktuellen Stand noch leer — die Architektur trägt sie schon mit, ohne dass eine Zeile Code geändert werden müsste, sobald jemand sie befüllt.

Wie die App bedient wird

Die Bedienung ist bewusst einfach gehalten. Beim ersten Start werden die konfigurierten Gesetze, Richtlinien und FAQ-Einträge automatisch importiert und für die spätere Suche vorbereitet.

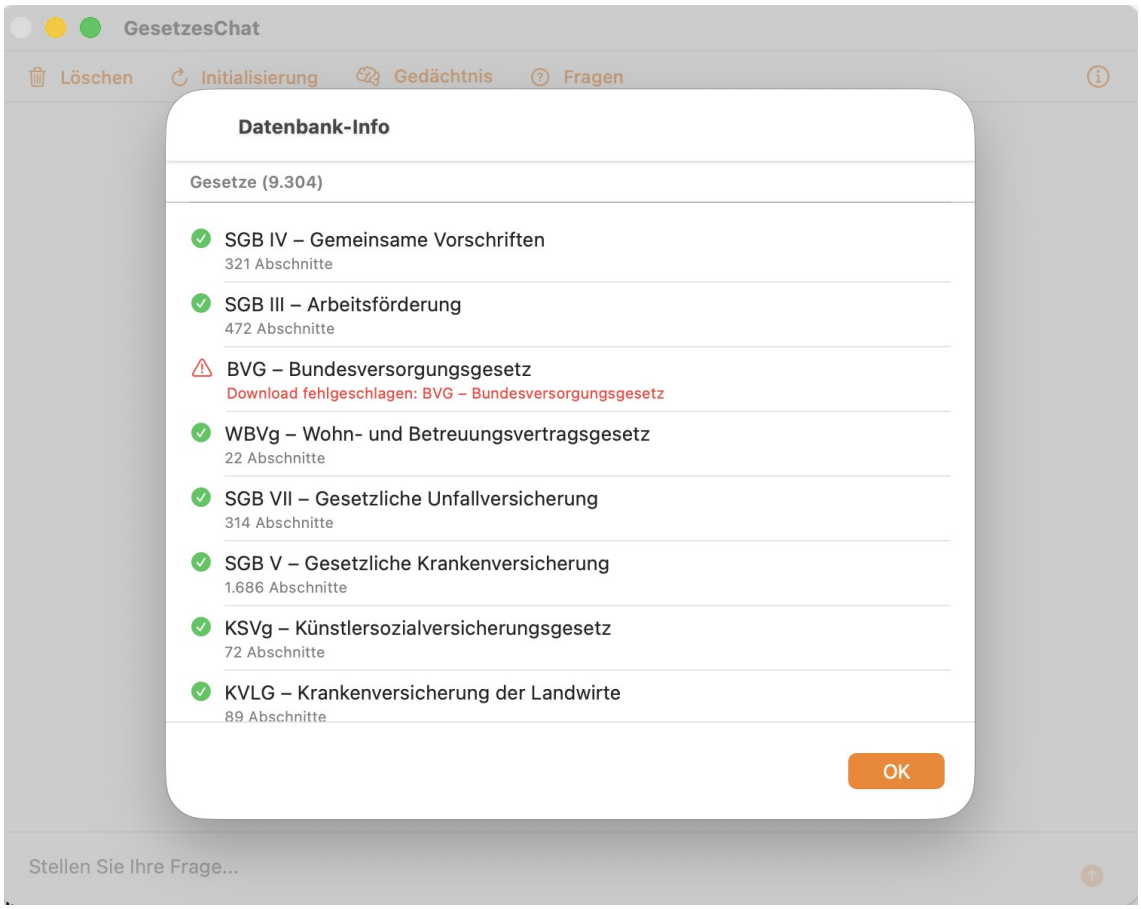

Abbildung: Import der Gesetzestexte beim ersten Start

Nach dem erfolgreichen Import kann man entweder eine der hinterlegten Beispielfragen auswählen oder eigene Fragen eingeben.

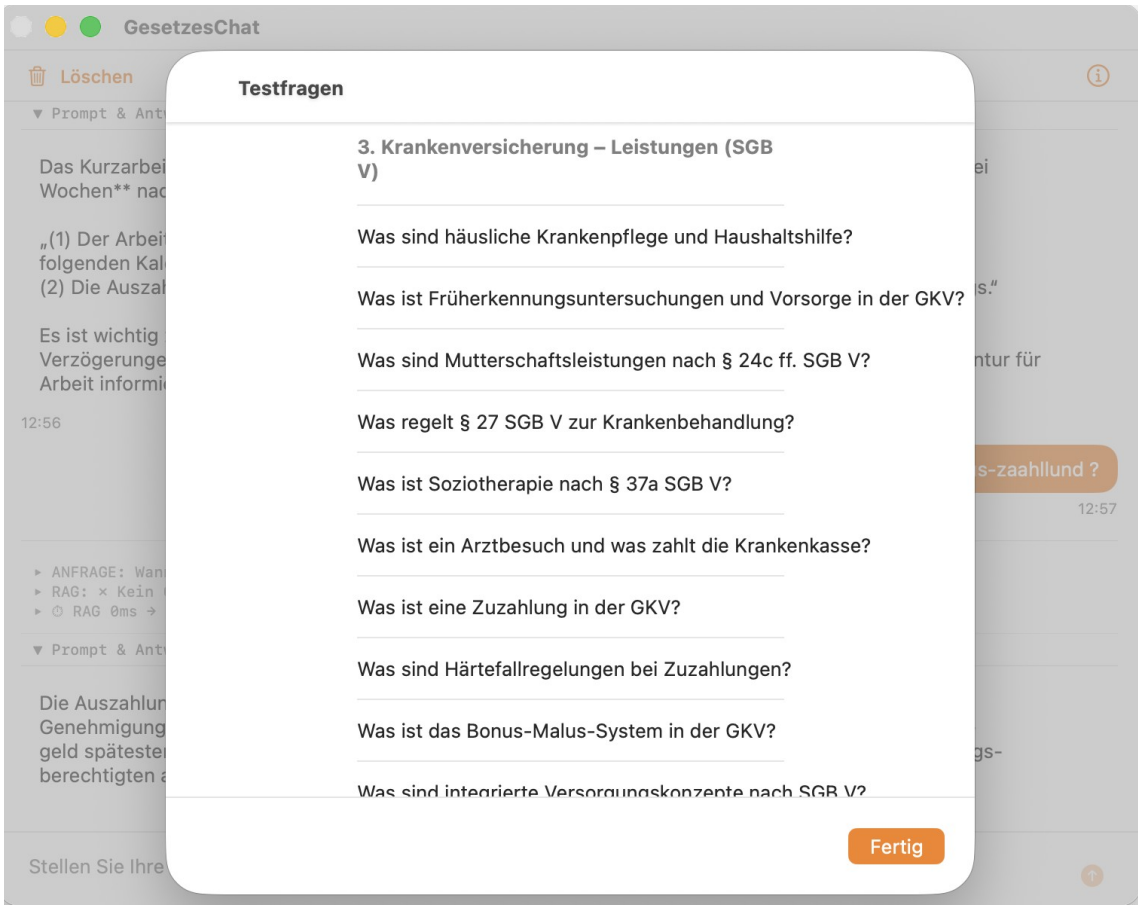

Abbildung: Beispielfragen zur Auswahl

Im unteren Bereich befindet sich ein Eingabefeld, über das Fragen und Rückfragen gestellt werden.

Damit auch Rückfragen richtig verstanden werden, besitzt die App ein Session-Gedächtnis: Die letzten Fragen und Antworten der laufenden Unterhaltung werden gespeichert und bei der nächsten Anfrage automatisch als zusätzlicher Kontext mitgegeben.

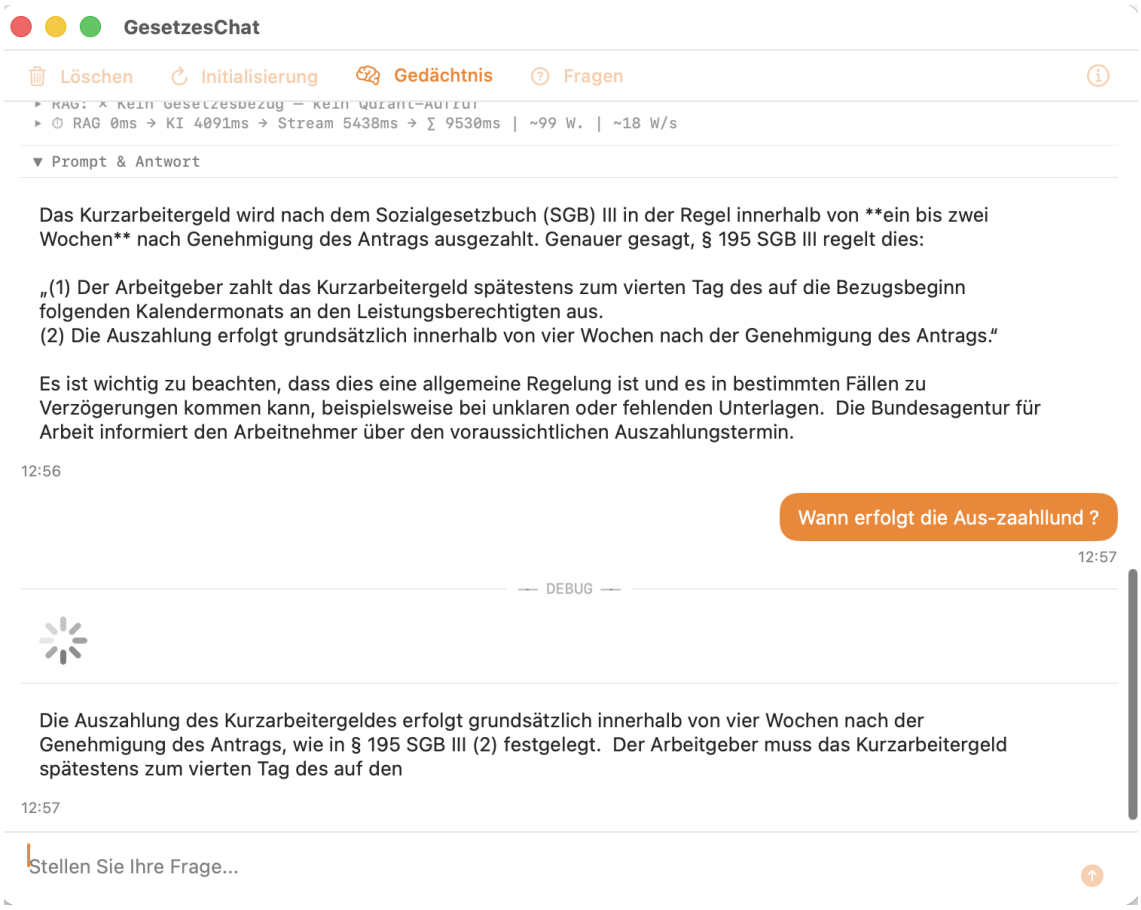

Abbildung: Rückfrage im laufenden Gespräch

So zeigt die App, wie sich mit einem lokalen RAG-System ein sicherer und erweiterbarer KI-Chatbot bauen lässt. Neue Gesetze, Richtlinien oder FAQ lassen sich jederzeit ergänzen, sodass sich die Lösung einfach an unterschiedliche Fachgebiete anpassen lässt.

Sicherheit und Datenschutz als Grundentscheidung, nicht als Nebeneffekt

Gerade bei einem sensiblen Thema wie Sozialrecht kombinieren Nutzer eine Rechtsfrage oft mit persönlichem Kontext. Deshalb läuft die KI vollständig lokal auf dem eigenen Rechner. Kein Cloud-Anbieter, kein Login, keine Nutzerdaten verlassen das Gerät. Sowohl das Sprachmodell über Ollama als auch die Vektordatenbank Qdrant sind ausschließlich über die lokale Adresse 127.0.0.1 erreichbar. Das ist ein bewusster, nachvollziehbarer Architekturentscheid und kein nachträglich aufgesetztes Datenschutz-Feature. Ein weiterer, oft unterschätzter Vorteil: keine laufenden API-Kosten und volle Funktionsfähigkeit auch ohne Internetverbindung, sobald die Wissensbasis einmal aufgebaut ist.

Abbildung: Weitere Informationen zum Projekt

Die eigentliche Geschichte: wie Claude Code das Projekt entwickelt hat

Der ungewöhnliche Teil an diesem Projekt ist weniger der Chatbot selbst als die Art, wie er entstanden ist. Im Repository liegt eine durchdachte Governance-Schicht: verbindliche Spezifikationsdokumente als einzige Wahrheitsquelle für Architektur, Glossar, Sicherheit, Fehlerbehandlung, Tests und Datenpipeline. Dazu spezialisierte, read-only Prüf-Subagenten für Architektur, Datenpipeline, UX und Sprachkonsistenz, sowie Skills, die nach jeder Implementierung automatisch genau diese Prüfungen anstoßen. Neue Begriffe müssen ins Glossar, neue Features zuerst in die Feature-Landkarte, jede Architekturentscheidung wird als Architecture Decision Record im Entwicklerjournal festgehalten.

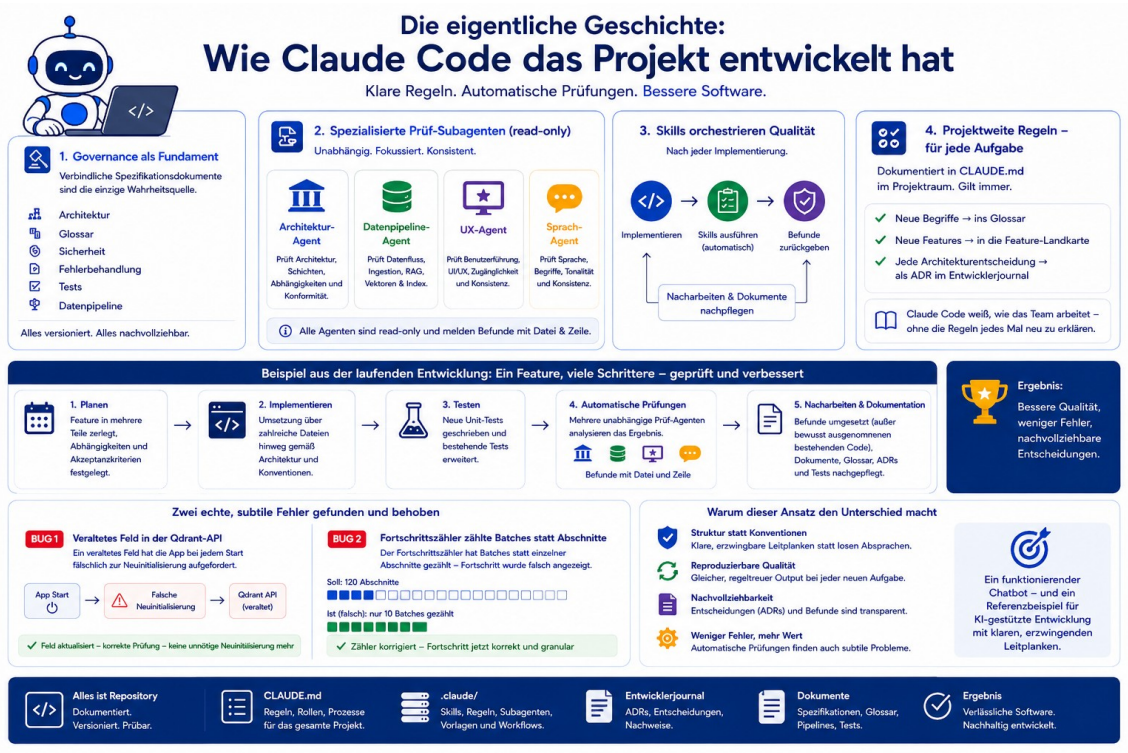

Abbildung: Claude Code als Governance-Schicht

Diese Struktur wurde nicht für eine einzelne Aufgabe angelegt. Sie existiert als feste Projekteinrichtung, dokumentiert in `CLAUDE.md` im Projektstamm, und gilt für jede Aufgabe in diesem Repository gleichermaßen, unabhängig vom Thema. Claude Code wird damit quasi angelernt, wie das Team programmieren möchte — und liefert bei jeder neuen Anfrage denselben, regeltreuen Output, ohne dass jemand die Regeln erneut erklären muss. Das Projekt ist damit gleichzeitig ein funktionierender Chatbot und ein Referenzbeispiel dafür, wie man KI-gestützte Entwicklung mit klaren, erzwungenen Leitplanken statt losen Konventionen betreibt.

Ein konkretes Beispiel aus der laufenden Entwicklung: Ein Feature mit mehreren Teilen wurde erst vollständig geplant, dann über zahlreiche Dateien hinweg implementiert und mit neuen Unit-Tests abgesichert. Anschließend liefen automatisch mehrere unabhängige Prüf-Agenten darüber und meldeten konkrete Befunde mit Datei und Zeile — ein Teil davon wurde bewusst nicht angefasst, weil er vorbestehenden Code betraf, der Rest wurde direkt behoben, inklusive Nachpflege der Spezifikationsdokumente. So wurden auch zwei echte, subtile Fehler gefunden: ein veraltetes Feld in der Qdrant-API, das die App bei jedem Start fälschlich zur Neuinitialisierung aufgefordert hat, und ein Fortschrittszähler, der Batches statt einzelner Abschnitte gezählt hat.

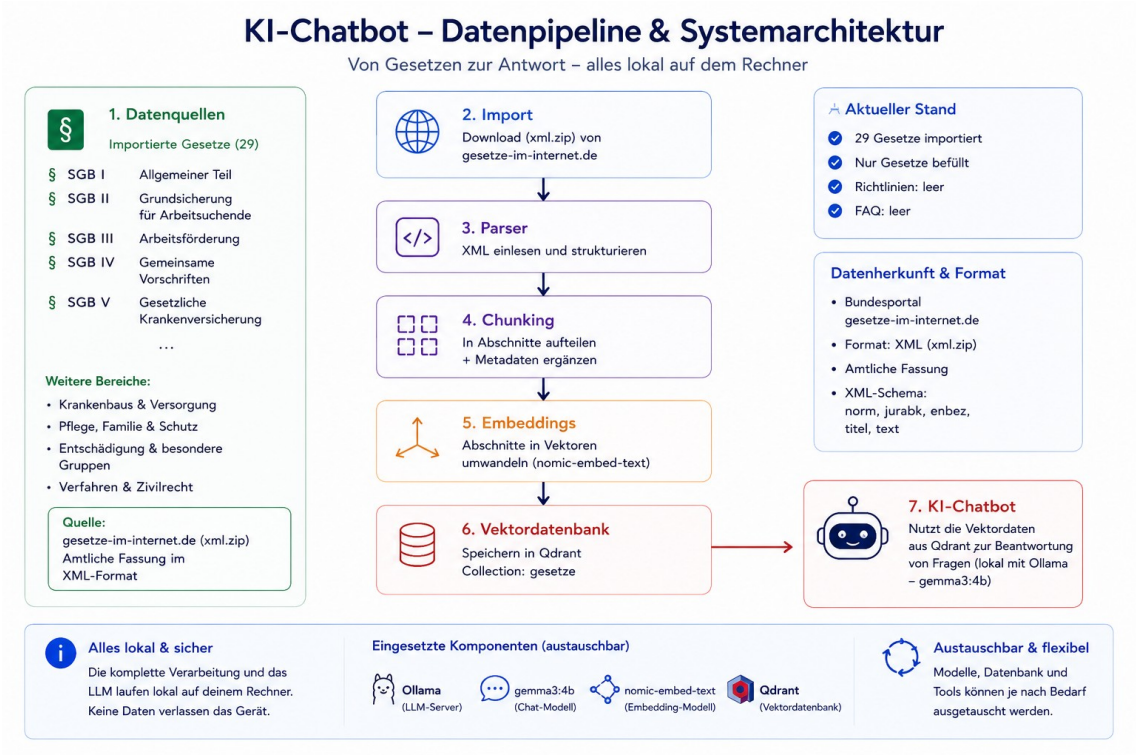

Abbildung: Importierte Gesetze

Architektur und Werkzeuge — kurz zusammengefasst

Architektur und technischer Ablauf

Die Architektur ist bewusst einfach gehalten und trennt die Verantwortlichkeiten klar voneinander. Die Benutzeroberfläche kümmert sich ausschließlich um die Darstellung und Benutzereingaben. Die Fachlogik entscheidet, was passieren soll. Der Datenbereich übernimmt den Zugriff auf externe Komponenten wie Ollama und Qdrant. Darüber liegt eine kleine Orchestrierungsschicht, die den gesamten RAG-Ablauf steuert. Alle Abhängigkeiten werden an einer zentralen Stelle erzeugt und über Dependency Injection bereitgestellt. Konfigurationswerte wie Modelle, Datenquellen oder Serveradressen stammen aus einer einzigen config.yaml. Dadurch lässt sich die Anwendung anpassen, ohne den Quellcode ändern zu müssen.

Technisch verwendet die App Swift und SwiftUI für die macOS-App, Ollama als lokalen Server für das Sprachmodell, Qdrant als lokale Vektordatenbank, XcodeGen zur Projektgenerierung und Swift Testing für Unit-Tests.

Der eigentliche Ablauf einer Anfrage beginnt in der Chat-Oberfläche. Die App nimmt die Frage des Benutzers entgegen und berücksichtigt den bisherigen Gesprächsverlauf als Kontext. Anschließend wird die Frage an die RAG-Komponente übergeben.

Abbildung: Architektur der App

RAG durchsucht nicht den vollständigen Gesetzestext, sondern eine zuvor vorbereitete Vektordatenbank. Dabei werden die semantisch passendsten Abschnitte aus Gesetzen, Richtlinien und FAQ gefunden und entsprechend ihrer Priorität gewichtet. Diese wenigen relevanten Textstellen bilden den Kontext für die eigentliche KI.

Erst jetzt kommt das Sprachmodell ins Spiel. Das Modell erhält nicht die komplette Wissensdatenbank, sondern lediglich die Benutzerfrage und die zuvor gefundenen Textabschnitte. Es analysiert den bereitgestellten Kontext, erkennt Zusammenhänge zwischen den Quellen und formuliert daraus eine

verständliche Antwort. Reicht der Kontext nicht aus, weist das Modell ausdrücklich darauf hin, anstatt Informationen zu erfinden.

Die fertige Antwort wird anschließend an die App zurückgegeben. Dort werden Quellenangaben ergänzt, die Antwort im Chat dargestellt und der Gesprächsverlauf gespeichert. Dieses Gesprächsgedächtnis ermöglicht es, dass spätere Fragen den bisherigen Dialog berücksichtigen können.

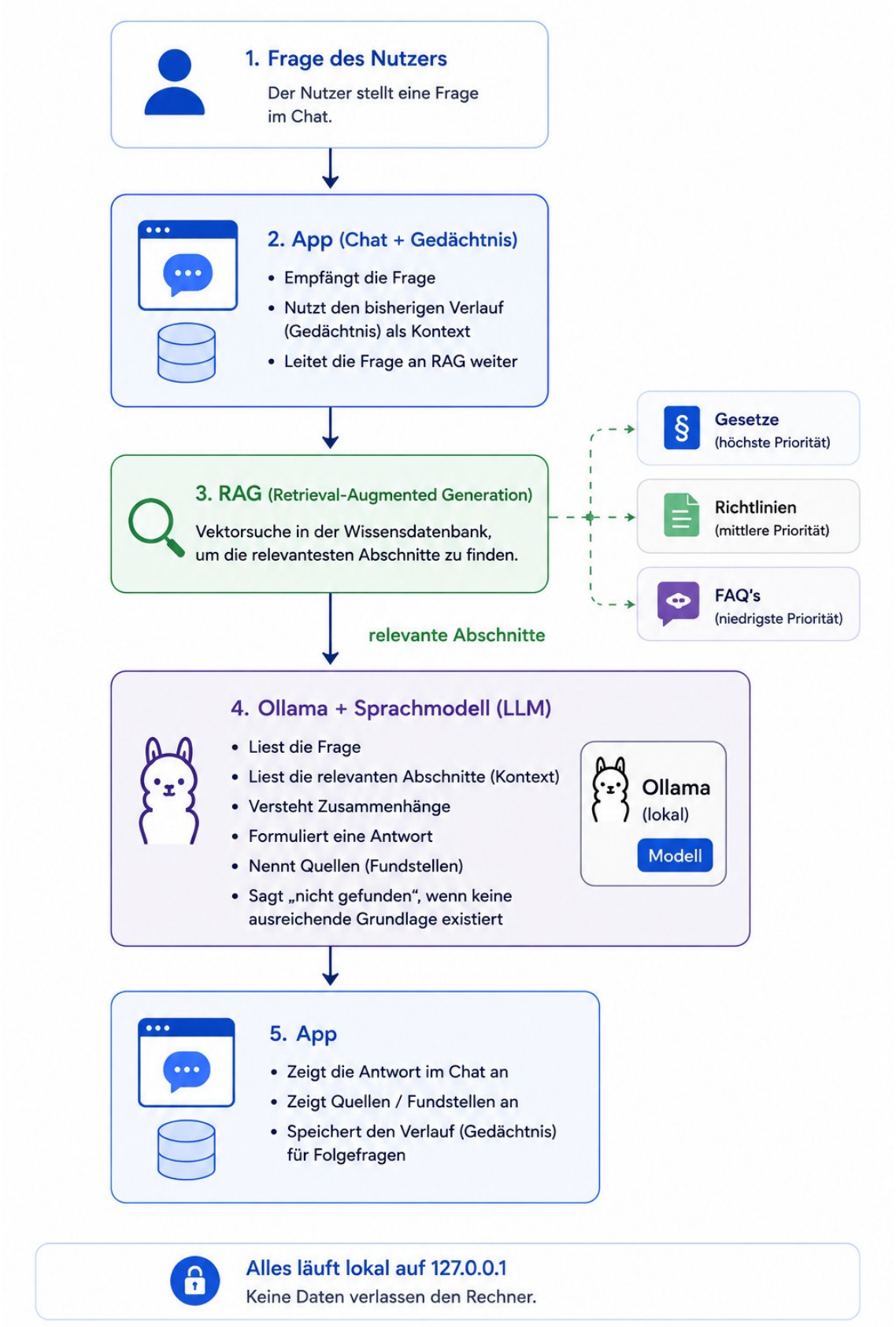

Abbildung: Entwicklungssystem

Der gesamte Datenfluss bleibt dabei lokal auf dem Rechner. Weder die Benutzerfrage noch die gefundenen Dokumente oder die erzeugte Antwort verlassen das Gerät.

Alle verwendeten Komponenten sind austauschbar. Statt Ollama könnte beispielsweise ein anderer lokaler Modellserver eingesetzt werden, ebenso ließe sich Qdrant gegen eine andere Vektordatenbank ersetzen. Der eigentliche Kern des Projekts ist jedoch weder das Sprachmodell noch die Datenbank, sondern das Governance-Regelwerk. Es definiert Architektur, Dokumentation, Qualitätsrichtlinien und Entwicklungsabläufe verbindlich und sorgt dafür, dass jede neue Funktion nach denselben Regeln geplant, umgesetzt und geprüft wird.

RAG in Kürze

Damit die KI nicht aus dem Gedächtnis heraus Paragraphen erfindet, arbeitet die App nach dem Prinzip **Retrieval-Augmented Generation** (RAG). Gesetzestexte werden einmalig heruntergeladen, geparst, in Abschnitte zerlegt und als mathematische Vektoren in Qdrant abgelegt. Stellt jemand eine Frage, wird auch sie in einen Vektor umgewandelt, die passendsten Abschnitte werden gefunden und gewichtet — Gesetze haben dabei bewusst Vorrang vor Richtlinien, Richtlinien vor FAQ, entsprechend der tatsächlichen juristischen Normenhierarchie. Erst dieser gefundene Kontext geht zusammen mit der Frage an das Sprachmodell, das angewiesen ist, ausschließlich auf dieser Grundlage zu antworten und offen zu sagen, wenn etwas nicht abgedeckt ist. Das ist die eigentliche Maßnahme gegen Halluzination.

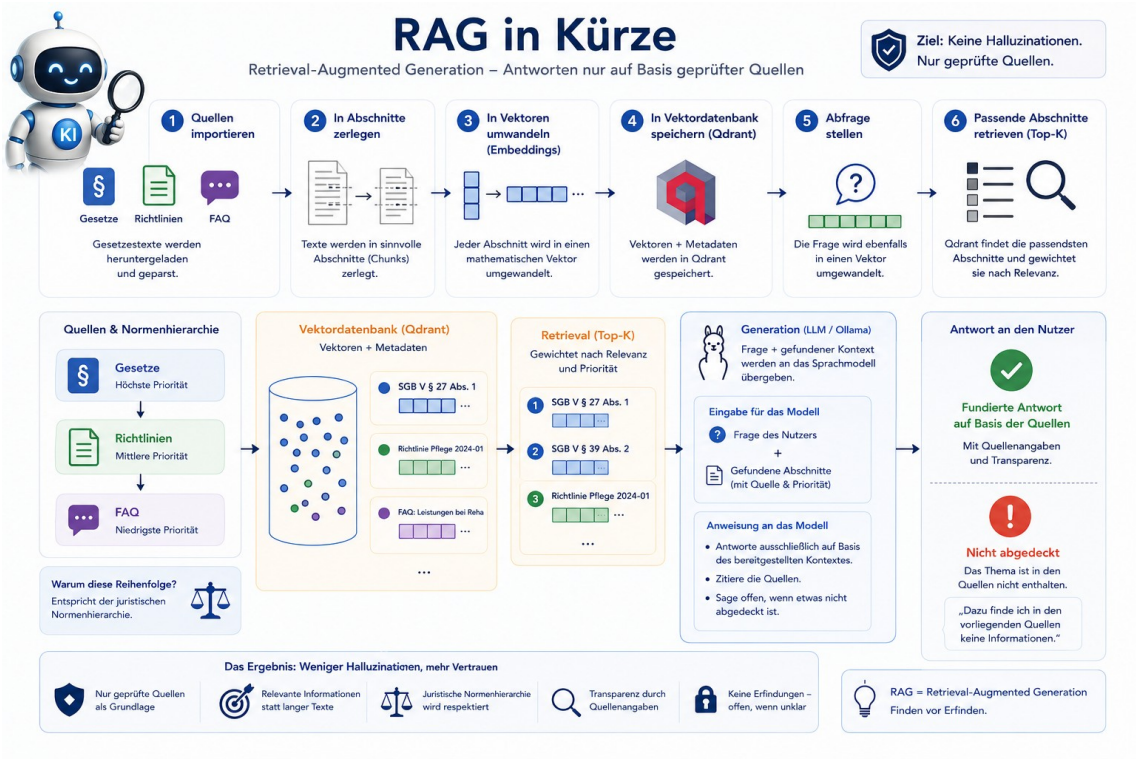

Abbildung: Die RAG-Pipeline

Weitere Tutorials verfügbar

Zu den verwendeten Technologien stehen auf dieser Website (https://learning.christiandrapatz.de/) bereits weitere Tutorials bereit. Dort findest du unter anderem Einführungen und Praxisbeispiele zu Ollama, OpenCode, Claude Code sowie Retrieval-Augmented Generation (RAG). Die Tutorials zeigen die Einrichtung, den praktischen Einsatz und das Zusammenspiel der einzelnen Komponenten anhand konkreter Beispiele.

Das Ziel

Das Ziel dieses Projekts war nie, den bestmöglichen Chatbot für Sozialversicherungsrecht zu bauen. Das Ziel war, eine App im Team zu programmieren, bei der klare Regeln gelten: eine festgelegte Architektur, verbindliche Pattern, eine eindeutige Vorgabe, wie Code dokumentiert wird — und das alles so, dass eine KI wie Claude Code diese Regeln bei jeder Sitzung zuverlässig anwendet, ohne dass sie jedes Mal neu erklärt werden müssen.

Der KI-Chatbot ist damit gleichzeitig zweierlei: eine funktionierende, lokale Lösung, die sich auf beliebige Regelwerke und Branchen übertragen lässt, und ein Referenzbeispiel dafür, wie KI-gestützte Teamentwicklung aussehen kann, wenn man dem Werkzeug vorher beibringt, wie das Team arbeiten möchte. Der Sourcecode bleibt dabei bewusst einfach und wartbar — nicht trotz, sondern gerade wegen des Regelwerks, das Beliebigkeit von vornherein ausschließt.