Lokale KI mit Ollama und Gemma 4

Sichere KI-Nutzung für Unternehmen mit Datenschutzanforderungen

Autor: Christian Drapatz

Mai 2026 · Ollama + Gemma 4 + OpenCode

Plattform: macOS · Apple Silicon

Version 1.0

Inhaltsverzeichnis

Disclaimer

Diese Anleitung wurde auf Basis öffentlich zugänglicher Quellen eigenständig erstellt und in eigenen Worten auf Deutsch formuliert. Als primäre Quellen dienten die offizielle Ollama-Dokumentation (ollama.com, MIT-Lizenz), die offizielle OpenCode-Dokumentation (opencode.ai, MIT-Lizenz) sowie eigene Tests und Community-Beiträge. Gemma 4 wird von Google unter den Gemma Terms of Use veröffentlicht (kein MIT). Weitere genannte Modelle (Llama, Mistral, Qwen, DeepSeek, Phi) unterliegen den jeweiligen Lizenzbedingungen ihrer Hersteller.

Technische Fakten wie API-Endpunkte, Konfigurationsoptionen und Befehle stammen aus den jeweiligen offiziellen Dokumentationen. Alle Texte wurden eigenständig strukturiert und formuliert – es erfolgte keine wörtliche Übernahme urheberrechtlich geschützter Formulierungen. Code-Beispiele orientieren sich an den offiziellen Beispielen der jeweiligen Projekte.

Die bereitgestellten Inhalte dienen ausschließlich der Wissensvermittlung. Es wird keine Gewähr für Vollständigkeit oder Aktualität übernommen. Alle genannten Marken, Produkte und Technologien gehören den jeweiligen Inhabern.

1. Warum lokale KI?

Viele Firmen möchten KI in ihren Entwicklungsprozess einbinden, stehen dabei aber vor einem grundsätzlichen Problem: Sensible Daten dürfen das eigene Netzwerk nicht verlassen.

Typische Beispiele:

- Quellcode mit Betriebsgeheimnissen
- Crash-Logs mit internen Stack Traces
- Interne Systemdokumentation
- Kundendaten oder personenbezogene Informationen
- Medizinische oder rechtliche Inhalte

Cloud-basierte KI-Dienste wie OpenAI, Anthropic oder Google senden Anfragen an externe Rechenzentren. Abhängig vom Anbieter und dessen Vertragsgestaltung können diese Rechenzentren außerhalb Deutschlands oder der EU liegen. Selbst bei Anbietern mit EU-Rechenzentren bleibt die Frage, wie Daten intern verarbeitet und gespeichert werden.

Lokale KI löst dieses Problem, indem das KI-Modell direkt auf dem eigenen Rechner oder im internen Firmennetzwerk läuft. Daten verlassen dabei die eigene Infrastruktur nicht.

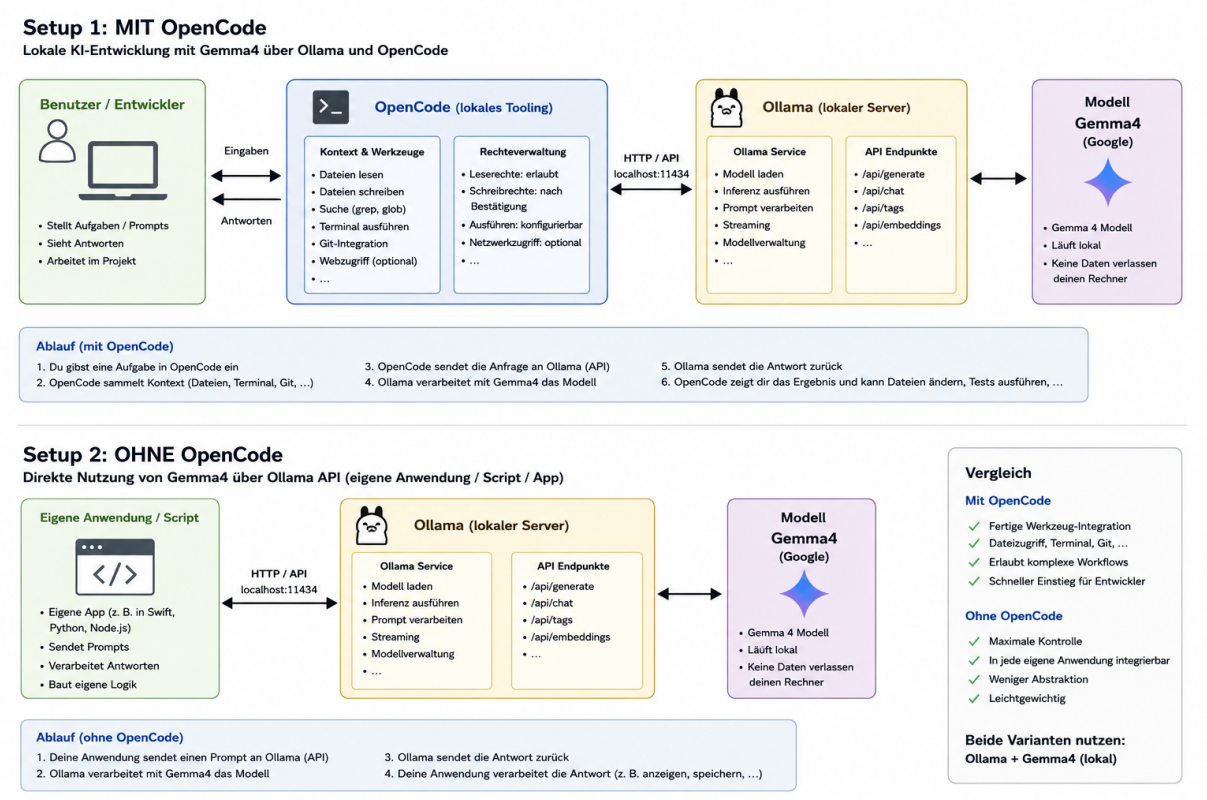
Das macht lokale KI besonders interessant für:

- Versicherungen, Banken und Finanzdienstleister
- Gesundheitswesen und Krankenkassen
- Behörden und öffentliche Einrichtungen
- Industrieunternehmen mit sensiblen Fertigungsdaten
- Softwarefirmen mit strenger IP-Schutzanforderung

Wichtiger Vorbehalt: Lokale KI ist kein Allheilmittel für Datenschutz. Sie verschiebt die Kontrolle auf die eigene Infrastruktur – setzt aber weiterhin sorgfältige Konfiguration, Zugriffsrechte und Netzwerksicherheit voraus.

2. Grundlegende Architektur

Bevor man konkrete Werkzeuge einrichtet, hilft ein Überblick über die Schichten des Systems.



Systemarchitektur: Ollama mit Gemma 4

Die Schichten im Überblick:

Schicht	Aufgabe
KI-Modell	Verarbeitet den Prompt und erzeugt eine Antwort
Ollama	Lädt, verwaltet und startet Modelle; stellt die HTTP-API bereit
Eigene Anwendung	Baut Prompts, sendet Anfragen, verarbeitet Antworten
OpenCode (optional)	Tool-System, das Dateien liest, ändert und Befehle ausführt

Wichtig: OpenCode und andere Tool-Systeme sind optional. Man kann Ollama direkt über HTTP ansprechen, ohne ein solches Tool-System zu verwenden.

3. Ollama

Was ist Ollama?

Ollama ist eine lokale Laufzeitumgebung für KI-Sprachmodelle. Es übernimmt das Laden, Verwalten und Ausführen von Modellen auf dem eigenen Rechner und stellt eine HTTP-Schnittstelle bereit, über die beliebige Anwendungen mit dem Modell kommunizieren können.

Ollama ist quelloffen und kostenlos. Es läuft als lokaler Dienst (Server) und ist für macOS, Linux und Windows verfügbar.

Was Ollama konkret tut:

- Modelle herunterladen und lokal speichern
- Modelle starten und Speicher verwalten
- Eine lokale HTTP-API auf Port 11434 bereitstellen
- Mehrere Modelle verwalten und bei Bedarf wechseln
- Eine OpenAI-kompatible API-Schnittstelle anbieten

Installation auf macOS

Voraussetzung: Homebrew

Homebrew muss installiert sein. Falls noch nicht vorhanden:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/homebrew/install/HEAD/install.sh)"
```

Hinweis zur Shell: macOS verwendet seit Catalina (10.15) standardmäßig **zsh**. Wer noch **bash** nutzt, verwendet statt `~/.zprofile` → `~/.bash_profile` und statt `~/.zshrc` → `~/.bashrc`. Welche Shell aktiv ist, zeigt:

```
echo $SHELL
```

Wichtig auf Apple Silicon (M1/M2/M3/M4): Nach der Homebrew-Installation muss der PATH gesetzt werden. Der Installer gibt diesen Befehl am Ende der Installation aus – falls er fehlt, manuell eintragen:

zsh:

```
echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> ~/.zprofile
source ~/.zprofile
```

bash:

```
echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> ~/.bash_profile
source ~/.bash_profile
```

Alternativ ein neues Terminalfenster öffnen – das lädt die Konfigurationsdatei ebenfalls automatisch.

Danach prüfen:

```
brew --version
```

Ollama installieren:

```
brew install ollama
```

Version prüfen:

```
ollama --version
```

Ollama als Hintergrunddienst starten (empfohlen):

Damit Ollama automatisch beim Login startet und immer im Hintergrund läuft, ohne ein Terminalfenster zu blockieren:

```
brew services start ollama
```

Status prüfen:

```
brew services list | grep ollama
```

Dienst stoppen:

```
brew services stop ollama
```

Alternativ: Manuell im Vordergrund starten:

```
ollama serve
```

Der Server läuft dann im Vordergrund und gibt Logs aus. Alternativ kann Ollama auch als macOS-App installiert werden, die im Hintergrund läuft. Wenn Ollama als Dienst über `brew services` läuft, ist `ollama serve` nicht notwendig.

Server-Erreichbarkeit prüfen:

```
curl http://localhost:11434
```

Bei laufendem Server antwortet Ollama mit:

```
Ollama is running
```

Modelle installieren

Der Server muss laufen, bevor Modelle installiert oder genutzt werden können.

Gemma 4 installieren:

```
ollama pull gemma4
```

Für eine spezifische Modellgröße (sofern verfügbar):

```
ollama pull gemma4:27b
```

Installierte Modelle anzeigen:

```
ollama list
```

Beispielausgabe:

```
NAME             ID              SIZE      MODIFIED
gemma4:latest    c6eb396dbd59    9.6 GB    About a minute ago
```

Modell interaktiv testen:

```
ollama run gemma4
```

Damit öffnet sich ein interaktives Gespräch mit dem Modell direkt im Terminal. Mit `/bye` beendet man die Session.

Modell entfernen:

```
ollama rm gemma4
```

Portbelegung prüfen

```
lsof -i :11434
```

oder:

```
netstat -an | grep 11434
```

Konfiguration und Verzeichnisse

Ollama speichert Modelle standardmäßig in:

```
~/.ollama/models
```

Über Umgebungsvariablen lässt sich das Verhalten anpassen:

Variable	Bedeutung
`OLLAMA_HOST`	Bindet den Server an eine bestimmte Adresse (Standard: `127.0.0.1:11434`)
`OLLAMA_MODELS`	Alternativer Speicherort für Modelle
`OLLAMA_NUM_PARALLEL`	Anzahl paralleler Anfragen

Umgebungsvariablen dauerhaft setzen:

Sollen Variablen bei jedem Terminal-Start automatisch gesetzt sein, in die Shell-Konfigurationsdatei eintragen:

```
# Ollama-Konfiguration
export OLLAMA_HOST=127.0.0.1:11434
export OLLAMA_MODELS=~/.ollama/models
```

Datei anwenden:

zsh:

```
source ~/.zshrc
```

bash:

```
source ~/.bashrc
```

B4ispiel: Server auf alle Netzwerkinterfaces binden (Vorsicht: nur in gesicherten Netzwerken):

```
OLLAMA_HOST=0.0.0.0:11434 ollama serve
```

Sicherheitsaspekte

Lokaler Betrieb: Standardmäßig lauscht Ollama nur auf `127.0.0.1`. Das bedeutet, nur Prozesse auf demselben Rechner können zugreifen. Das ist die sichere Grundkonfiguration.

Firewall: Wenn Ollama auf einem Server im Firmennetzwerk läuft, muss Port 11434 durch Firewall-Regeln auf autorisierte Clients beschränkt werden.

Modellquellen: Ollama bezieht Modelle aus seinem offiziellen Repository (`ollama.com/library`). Modelle aus unbekannten Quellen sollten nicht verwendet werden – ähnlich wie bei Softwarepaketen aus fremden Repositories.

Prompt-Inhalte: Auch bei lokalem Betrieb gilt: Keine Zugangsdaten, private Schlüssel oder unnötige personenbezogene Daten in Prompts aufnehmen. Das ist weniger eine Frage der Netzwerkübertragung als der sauberen Systementwicklung.

Lokale KI ersetzt keine Sicherheitsarchitektur. Rechteverwaltung, Netzwerksicherheit, Zugriffsprotokollierung und Backups sind weiterhin eigenverantwortlich zu organisieren.

Alternativen zu Ollama

Neben Ollama gibt es weitere lokale KI-Lösungen:

- **LM Studio** – Einfache GUI, gut für Einsteiger, aber etwas schwergewichtiger.
- **LocalAI** – Flexible OpenAI-kompatible API, aber komplexer in der Einrichtung.
- **Jan** – Moderne Desktop-App mit lokaler KI und API-Unterstützung.
- **Text Generation WebUI** – Sehr flexibel, aber eher für erfahrene Nutzer gedacht.
- **Open WebUI** – Keine Alternative zu Ollama, sondern eine grafische Oberfläche, die typischerweise *auf* Ollama läuft.

Ich habe mich zunächst für Ollama entschieden, weil es leichtgewichtig ist, schnell eingerichtet werden kann und eine einfache HTTP-Schnittstelle bietet. Genau diese API ist wichtig, damit eigene Software direkt mit lokalen KI-Modellen kommunizieren kann — komplett lokal und ohne Cloud-Anbindung.

4. Gemma 4

Was ist Gemma 4?

Gemma ist eine Familie lokaler KI-Sprachmodelle von Google. Die Modelle wurden mit dem Ziel entwickelt, effizient auf Consumer-Hardware zu laufen und gleichzeitig für typische Entwickleraufgaben brauchbare Ergebnisse zu liefern.

Gemma-Modelle sind als Open-Weight-Modelle verfügbar, das heißt: die Gewichte (die eigentlichen Modelldaten) können heruntergeladen und lokal betrieben werden.

Hinweis: Nicht zu verwechseln mit Googles Cloud-Modellen (Gemini). Gemma ist die lokale Variante für den Eigenbetrieb.

Wofür eignet sich Gemma 4?

Gut geeignet:

- Swift-, Python- oder JavaScript-Code analysieren
- Fehler im Code suchen und erklären
- Dokumentationen und Kommentare erzeugen
- Übersetzungen (Code-Kommentare, Dokumentationen)
- Refactoring-Vorschläge erstellen
- Unit-Test-Entwürfe generieren
- Crash-Logs oder Stack Traces einordnen
- Texte zusammenfassen

Weniger geeignet:

- Sehr große Multi-Projekt-Analysen (eingeschränktes Kontextfenster)
- Tiefes Verständnis komplexer, verteilter Architekturen
- Autonome, mehrstufige Agenten-Workflows
- Aufgaben, die aktuelle Informationen aus dem Web erfordern

Grenzen und Risiken

Halluzinationen: Wie alle KI-Sprachmodelle kann Gemma 4 Aussagen erzeugen, die plausibel klingen, aber sachlich falsch sind. Das ist besonders kritisch bei:

- Sicherheitsrelevanten Empfehlungen
- API-Schnittstellen und deren genauen Signaturen
- Datenbanklogik und Transaktionsverhalten
- Regulatorischen oder rechtlichen Fragen

Kein Ersatz für Fachurteile: Die KI-Ausgabe ist ein Entwurf oder ein Vorschlag – kein geprüftes Ergebnis. Security-Reviews, Architekturentscheidungen und Datenschutzprüfungen müssen weiterhin von Menschen vorgenommen werden.

Modellgrößen

Gemma-Modelle erscheinen in verschiedenen Größenklassen, typischerweise gemessen in Milliarden Parametern (B = Billion):

Größe	Beispiel	Eigenschaften
Klein (2B–4B)	gemma:2b	Schnell, wenig RAM, einfachere Aufgaben
Mittel (7B–9B)	gemma:9b	Ausgewogen, gute Alltagstauglichkeit
Groß (27B)	gemma:27b	Bessere Qualität, hoher Speicherbedarf

Größere Modelle liefern in der Regel bessere Ergebnisse, benötigen aber mehr Arbeitsspeicher und sind langsamer.

Hardware-Anforderungen

Die folgende Tabelle ist eine grobe Orientierung. Die tatsächlichen Anforderungen hängen von Kontextgröße und parallelen Anfragen ab.

Modellgröße	Empfohlener RAM
Kleine Modelle (bis 4B)	16 GB
Mittlere Modelle (7B–9B)	32 GB
Große Modelle (27B)	64 GB oder mehr

Apple Silicon: Macs mit M-Prozessoren eignen sich besonders gut für lokale KI, weil RAM und GPU-Speicher geteilt werden (Unified Memory). Das bedeutet: Ein Mac mit 64 GB RAM kann ein 27B-Modell vollständig im Speicher halten, ohne zwischen RAM und VRAM wechseln zu müssen. Das ist effizienter als viele PC-Konfigurationen mit separater GPU.

Getestete Prozessoren (als grobe Referenz aus der Praxis):

- M1 Pro / M2 Pro / M3 Pro / M4 Pro
- M1 Max / M2 Max / M3 Max / M4 Max
- M2 Ultra / M3 Ultra

Relative Leistung

Direkte Vergleiche zwischen lokalen und Cloud-Modellen sind schwierig, weil Benchmarks stark aufgabenabhängig sind. Als grobe, nicht-offizielle Orientierung für Entwicklungsaufgaben:

Modellgröße	Relative Qualität (Schätzung)
Kleine Gemma-Modelle (2B–4B)	35–50 % eines modernen Cloud-Modells
Mittlere Gemma-Modelle (7B–9B)	50–70 %
Große Gemma-Modelle (27B)	70–85 %

Diese Einschätzung gilt für typische Entwickleraufgaben wie Code-Analyse, Dokumentation und einfaches Refactoring. Bei sehr komplexen Aufgaben fällt die Lücke größer aus.

5. Weitere Modelle in Ollama

Ollama unterstützt neben Gemma viele weitere Modelle. Eine aktuelle Liste findet sich unter `ollama.com/library`.

Häufig verwendete Alternativen:

Modell	Herkunft	Besonderheiten
Llama 3 / Llama 3.1	Meta	Weit verbreitet, gut dokumentiert
Qwen	Alibaba	Stärken bei mehrsprachigen Aufgaben
Mistral / Mixtral	Mistral AI	Effizient, gute Instruction-Following-Qualität
DeepSeek	DeepSeek	Stärken bei Code-Aufgaben
Phi-3 / Phi-4	Microsoft	Kleine, effiziente Modelle

Tool-Calling

Ein wichtiges Merkmal für Tool-Systeme wie OpenCode: Nicht jedes Modell unterstützt Tool-Calling zuverlässig. Tool-Calling bedeutet, dass das Modell strukturiert Werkzeugaufrufe (Dateilesen, Befehlsausführung) zurückgeben kann.

Ohne funktionierendes Tool-Calling arbeiten automatisierte Systeme unzuverlässig oder gar nicht. Vor dem Einsatz in einem Tool-System sollte geprüft werden, ob das gewählte Modell Tool-Calling unterstützt.

6. Die HTTP-API

Die HTTP-API ist die eigentliche Grundlage von Ollama. Alle weiteren Werkzeuge – einschließlich OpenCode – kommunizieren intern über diese Schnittstelle.

Wer die API versteht, versteht das System.

Basis-URL

```
http://localhost:11434
```

Alternativ:

```
http://127.0.0.1:11434
```

Ollama lauscht standardmäßig nur auf `localhost`, das heißt nur lokale Prozesse haben Zugriff.

Endpunkte im Überblick

Endpunkt	Methode	Zweck
`/api/generate`	POST	Einfache Text-Generierung (Prompt → Antwort)
`/api/chat`	POST	Chat-Konversation mit Nachrichten-Array
`/api/tags`	GET	Installierte Modelle auflisten
`/api/show`	POST	Informationen zu einem Modell abrufen
`/api/pull`	POST	Modell herunterladen
`/api/delete`	DELETE	Modell entfernen
`/v1/chat/completions`	POST	OpenAI-kompatibler Endpunkt

Offizielle Dokumentation der Endpunkte:

Endpunkt	Dokumentation
Native Ollama-API (`/api/generate`, `/api/chat`, `/api/tags`, …)	`https://github.com/ollama/ollama/blob/main/docs/api.md`
OpenAI-kompatibler Endpunkt (`/v1/chat/completions`, …)	`https://github.com/ollama/ollama/blob/main/docs/openai.md`

Endpunkt: /api/generate

Der einfachste Einstieg. Sendet einen Prompt und empfängt eine Antwort.

Anfrage:

```
curl http://localhost:11434/api/generate \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gemma4",
    "prompt": "Was ist das Repository-Pattern in Swift?",
    "stream": false
  }'
```

Antwort (stark gekürzt):

```
{
  "model": "gemma4",
  "created_at": "2025-01-01T10:00:00.000Z",
  "response": "Das Repository-Pattern ist ein...",
  "done": true,
  "total_duration": 8540000000,
  "prompt_eval_count": 18,
  "eval_count": 120
}
```

Wichtige Felder in der Antwort:

Feld	Bedeutung
`response`	Die eigentliche Antwort des Modells
`done`	`true` wenn die Generierung abgeschlossen ist
`total_duration`	Gesamtzeit in Nanosekunden
`eval_count`	Anzahl generierter Tokens

Streaming

Mit `"stream": true` liefert Ollama die Antwort als Datenstrom (NDJSON: newline-delimited JSON). Jede Zeile ist ein JSON-Objekt mit einem Teil der Antwort.

```
curl http://localhost:11434/api/generate \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gemma4",
    "prompt": "Erkläre async/await in Swift",
    "stream": true
  }'
```

Jede Zeile der Antwort enthält einen Token:

```
{"model":"gemma4","response":"In","done":false}
{"model":"gemma4","response":" Swift","done":false}
{"model":"gemma4","response"," ist","done":false}
...
{"model":"gemma4","response":"","done":true,"total_duration":...}
```

Wann Streaming verwenden:

- In interaktiven UIs, die Text schrittweise anzeigen sollen
- Bei langen Antworten, um die Wartezeit zu überbrücken

Wann `stream: false` verwenden:

- Bei programmatischer Verarbeitung der Antwort
- Wenn die vollständige Antwort als JSON benötigt wird
- In einfachen Skripten und CLI-Tools

Endpunkt: /api/chat

Für Konversationen mit mehreren Nachrichten. Das Modell erhält den bisherigen Gesprächsverlauf.

Anfrage:

```
curl http://localhost:11434/api/chat \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gemma4",
    "messages": [
      {
        "role": "system",
        "content": "Du bist ein erfahrener Swift-Entwickler. Antworte präzise und
ohne Einleitung."
      },
      {
        "role": "user",
        "content": "Welche Probleme kann Force Unwrapping in Swift verursachen?"
```

```
      }
    ],
    "stream": false
  }'
```

Antwort:

```
{
  "model": "gemma4",
  "message": {
    "role": "assistant",
    "content": "Force Unwrapping mit ! kann zu einem..."
  },
  "done": true
}
```

Rollen:

Rolle	Bedeutung
`system`	Anweisung an das Modell (Verhalten, Kontext, Einschränkungen)
`user`	Nachricht des Nutzers
`assistant`	Frühere Antworten des Modells (für Gesprächsverlauf)

Endpunkt: /v1/chat/completions (OpenAI-kompatibel)

Ollama bietet eine OpenAI-kompatible API-Schnittstelle. Das ist relevant für Werkzeuge und Bibliotheken, die ursprünglich für OpenAI entwickelt wurden – sie funktionieren oft unverändert mit Ollama.

OpenCode nutzt intern diesen Endpunkt.

Anfrage:

```
curl http://localhost:11434/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer ollama" \
  -d '{
    "model": "gemma4",
    "messages": [
      {"role": "system", "content": "You are a Swift expert."},
      {"role": "user", "content": "Explain @MainActor in one paragraph."}
    ],
    "stream": false
  }'
```

Hinweis zum Authorization-Header: Bei Ollama wird der Token nicht geprüft. Der Header muss aber vorhanden sein, wenn Bibliotheken ihn erwarten. Jeder Wert funktioniert (`Bearer ollama`, `Bearer anything`).

Zusätzliche Optionen

Der `options`-Block erlaubt die Feinsteuerung des Modells:

```
{
  "model": "gemma4",
  "prompt": "...",
  "stream": false,
  "options": {
    "temperature": 0.2,
    "num_ctx": 8192,
    "num_predict": 2048,
    "top_p": 0.9
  }
}
```

Option	Bedeutung	Empfehlung
`temperature`	Kreativität der Antworten (0 = deterministisch, 1 = kreativ)	0.1–0.3 für Code-Aufgaben
`num_ctx`	Kontextfenstergröße in Tokens	Je nach Modell bis 8192 oder mehr
`num_predict`	Maximale Anzahl generierter Tokens	Je nach Aufgabe anpassen
`top_p`	Nucleus-Sampling-Parameter	Meist Standard belassen

Kontextgröße: Wenn große Dateien oder viel Kontext übergeben werden soll, muss `num_ctx` entsprechend groß gesetzt werden. Der Standardwert in Ollama ist oft niedriger als das Modellmaximum.

Installierte Modelle abfragen

```
curl http://localhost:11434/api/tags
```

Antwort:

```
{
  "models": [
    {
      "name": "gemma4:latest",
      "model": "gemma4:latest",
      "size": 9607000000,
      "modified_at": "2025-01-01T10:00:00.000Z"
    }
```

```
  ]
}
```

Typische Fehlerquellen bei der API

Ollama läuft nicht:

```
curl: (7) Failed to connect to localhost port 11434: Connection refused
```

Lösung: `ollama serve` ausführen.

Falscher Modellname:

```
{"error":"model \"gemma_4\" not found, try pulling it first"}
```

Lösung: `ollama list` prüfen, dann den genauen Namen aus der Liste verwenden.

Timeout bei großen Prompts:

Bei sehr langen Anfragen kann der Standard-Timeout von HTTP-Clients unterschritten werden. Die Lösung ist, den Timeout in der Anwendung auf mehrere Minuten zu setzen (z. B. 5–10 Minuten für komplexe Aufgaben).

Keine Antwort trotz laufendem Server:

Der Server kann ausgelastet sein, wenn mehrere große Modelle parallel laufen. Große Modelle nicht mehrfach gleichzeitig starten.

7. Eigene Anwendungen entwickeln

Grundprinzip

Eigene Anwendungen sprechen Ollama direkt über HTTP an. Es wird keine besondere Bibliothek benötigt – eine einfache HTTP-Anfrage mit JSON-Body reicht aus.

Das bedeutet auch: **Man benötigt weder OpenCode noch ein anderes Tool-System**, wenn man eine eigene Integration entwickelt. Die Anwendung ist selbst dafür verantwortlich:

- welche Dateien gelesen werden
- welcher Kontext für den Prompt aufgebaut wird
- wie die Antwort verarbeitet und dargestellt wird
- welche Dateien geändert werden dürfen

Beispiel in Swift

Das folgende Beispiel zeigt eine einfache Funktion, die einen Prompt an Ollama sendet und die Antwort zurückgibt.

```
import Foundation

// MARK: - Datenmodelle

struct OllamaGenerateRequest: Encodable {
    let model: String
    let prompt: String
    let stream: Bool
    let options: OllamaOptions?
}

struct OllamaOptions: Encodable {
    let temperature: Double
    let numCtx: Int

    enum CodingKeys: String, CodingKey {
        case temperature
        case numCtx = "num_ctx"
    }
}

struct OllamaGenerateResponse: Decodable {
    let model: String
    let response: String
    let done: Bool
}

// MARK: - Ollama-Client

struct OllamaClient {
    let baseURL: URL
    let model: String
    private let session: URLSession
```

```
    init(
        baseURL: URL = URL(string: "http://localhost:11434")!,
        model: String = "gemma4"
    ) {
        self.baseURL = baseURL
        self.model = model
        let config = URLSessionConfiguration.default
        config.timeoutIntervalForRequest = 300  // 5 Minuten
        self.session = URLSession(configuration: config)
    }

    func generate(prompt: String) async throws -> String {
        let endpoint = baseURL.appendingPathComponent("api/generate")
        let body = OllamaGenerateRequest(
            model: model,
            prompt: prompt,
            stream: false,
            options: OllamaOptions(temperature: 0.2, numCtx: 8192)
        )

        var request = URLRequest(url: endpoint)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.httpBody = try JSONEncoder().encode(body)

        let (data, response) = try await session.data(for: request)

        guard let http = response as? HTTPURLResponse,
              (200...299).contains(http.statusCode) else {
            throw URLError(.badServerResponse)
        }

        let decoded = try JSONDecoder().decode(OllamaGenerateResponse.self, from:
data)
        return decoded.response
    }
}

// MARK: - Verwendung

Task {
    let client = OllamaClient()
    do {
        let result = try await client.generate(
            prompt: "Analysiere den folgenden Swift-Code auf potenzielle
Fehler:\n\n```swift\n// ... code hier ...\n```"
        )
        print(result)
    } catch {
        print("Fehler: \(error.localizedDescription)")
    }
}
```

Chat-Konversation in Swift (OpenAI-kompatibler Endpunkt)

```
struct ChatMessage: Codable {
    let role: String
    let content: String
}

struct ChatRequest: Encodable {
    let model: String
    let messages: [ChatMessage]
    let stream: Bool
}

struct ChatResponse: Decodable {
    struct Choice: Decodable {
        let message: ChatMessage
    }
    let choices: [Choice]
}

func sendChatRequest(messages: [ChatMessage]) async throws -> String {
    let url = URL(string: "http://localhost:11434/v1/chat/completions")!
    let body = ChatRequest(model: "gemma4", messages: messages, stream: false)

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    request.setValue("Bearer ollama", forHTTPHeaderField: "Authorization")
    request.httpBody = try JSONEncoder().encode(body)
    request.timeoutInterval = 300

    let (data, _) = try await URLSession.shared.data(for: request)
    let decoded = try JSONDecoder().decode(ChatResponse.self, from: data)
    return decoded.choices.first?.message.content ?? ""
}
```

Hinweise zur Integration

Timeout: KI-Antworten können mehrere Minuten dauern, besonders bei großen Modellen und langen Prompts. Den HTTP-Timeout entsprechend konfigurieren.

Kontextgröße: Große Dateien überschreiten schnell das Kontextfenster. Dateien ggf. kürzen oder nur relevante Abschnitte übergeben.

Parallelität: Ollama verarbeitet Anfragen sequenziell (eine nach der anderen). Mehrere gleichzeitige Anfragen werden in einer Warteschlange bearbeitet, was die Gesamtlatenz erhöht.

Fehlerbehandlung: HTTP-Statuscodes und JSON-Fehlermeldungen von Ollama immer prüfen, insbesondere wenn Modelle nicht geladen sind oder der Server überlastet ist.

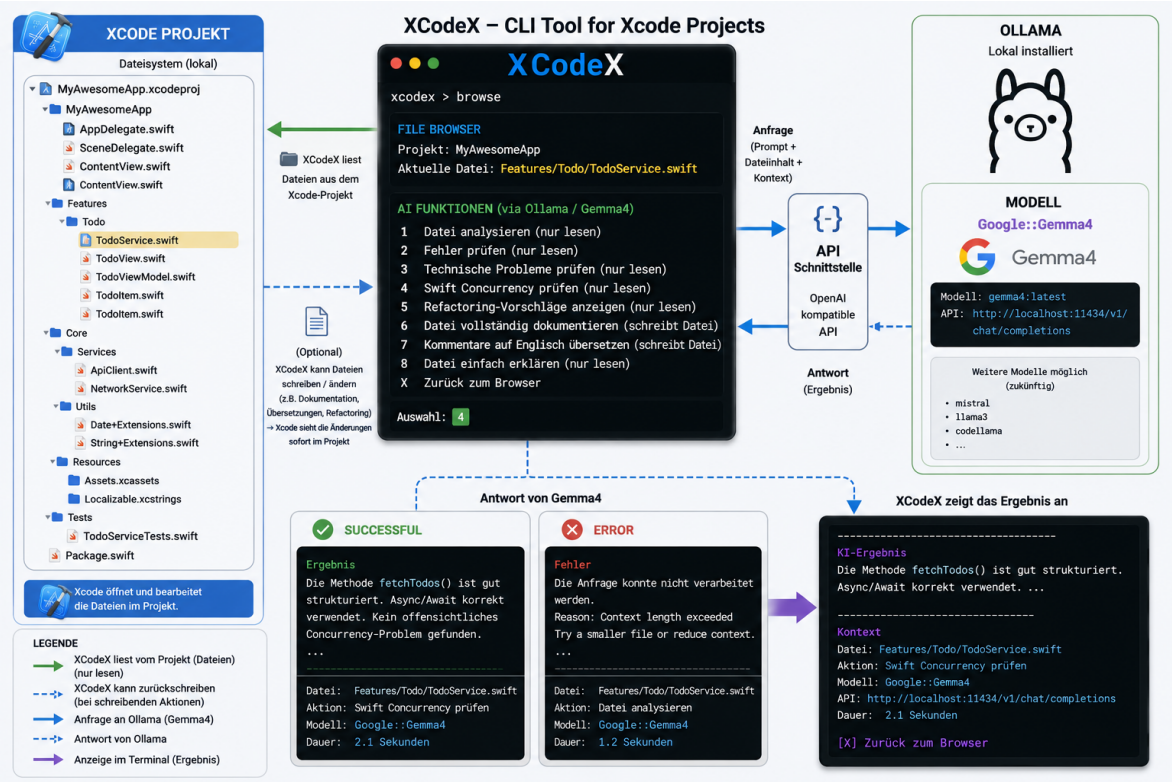
Praxisbeispiel: XcodeX CLI

Ein konkretes Beispiel für eigene KI-Integration ist **XcodeX CLI** – ein Skript, das den Build-, Test- und Deploy-Prozess unter Xcode optimiert. Es ermöglicht:

- Apps bauen und auf mehrere Testgeräte mit unterschiedlichen Betriebssystemen verteilen
- Automatisierte Unit-Tests mit verschiedenen Konfigurationen ausführen
- Unabhängig von Xcode arbeiten, da in einem separaten DerivedData gebaut wird

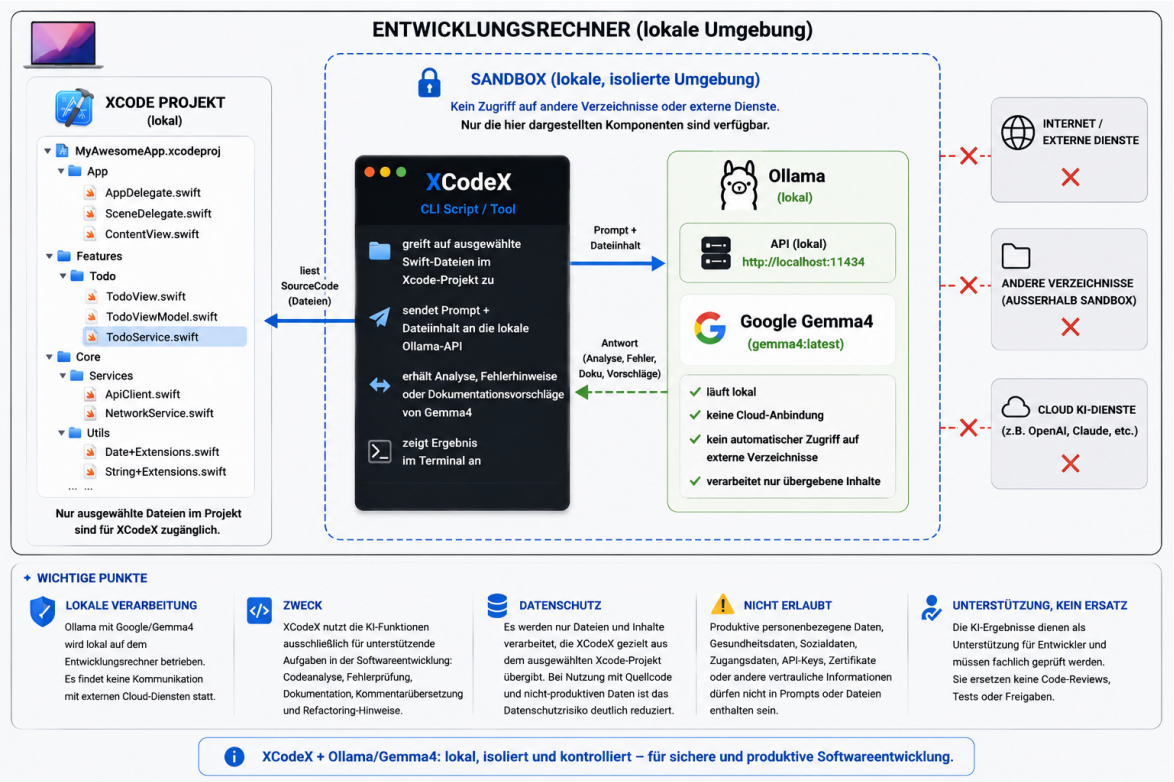
Ideal für iOS-, iPadOS- und macOS-Entwickler. Weitere Informationen: https://xcodexcli.com/

Im nächsten Release wird eine lokale KI eingebunden, um den Entwickler direkt im Workflow zu unterstützen.



XcodeX mit Ollama und Gemma 4

Besonders interessant für Firmen mit Datenschutzanforderungen: Da die KI-Integration auf Ollama basiert, verlassen keine Daten die eigene Infrastruktur — die Verarbeitung findet vollständig lokal statt.



XcodeX mit Ollama und Gemma 4

8. OpenCode

Was ist OpenCode?

OpenCode ist ein lokales Tool-System für KI-gestützte Softwareentwicklung. Die Idee orientiert sich an Werkzeugen wie Claude Code: Eine KI analysiert Code, schlägt Änderungen vor und kann Dateien direkt bearbeiten.

Im Unterschied zu Cloud-basierten Lösungen kann OpenCode mit lokalen Modellen über Ollama betrieben werden.

Modell vs. Tool-System

Das ist ein grundlegender Unterschied, den man verstehen sollte:

Begriff	Bedeutung
KI-Modell (z. B. Gemma 4)	Verarbeitet Text, erzeugt Antworten, hat keinen direkten Systemzugriff
Tool-System (z. B. OpenCode)	Führt lokale Aktionen aus: Dateien lesen/ändern, Befehle ausführen
Laufzeit (z. B. Ollama)	Startet und betreibt das Modell, stellt die API bereit

Das Modell selbst hat keinen Dateisystemzugriff. Das Tool-System sammelt zuerst die relevanten Informationen, baut daraus einen Prompt, und gibt die Modellantwort wieder in lokale Aktionen um.

Der vereinfachte Ablauf:

```
Aufgabe eingeben
    → OpenCode sucht relevante Dateien
    → Dateiinhalte werden gelesen
    → Prompt wird aufgebaut
    → Prompt geht an Ollama / Gemma 4
    → Antwort kommt zurück
    → OpenCode schlägt Änderungen vor oder fragt nach Bestätigung
    → Änderungen werden auf das Dateisystem geschrieben
```

Wie kommuniziert OpenCode mit Ollama?

OpenCode nutzt intern den OpenAI-kompatiblen Endpunkt von Ollama:

```
http://localhost:11434/v1
```

Das bedeutet: OpenCode spricht Ollama so an, als wäre es ein OpenAI-API-Server. Ollama übersetzt diese Anfragen intern für das jeweilige Modell.

Installation

Variante 1 – Installationsskript:

```
curl -fsSL https://opencode.ai/install | bash
```

Das Skript installiert OpenCode typischerweise nach `~/.local/bin`. Je nach Version kann der Pfad aber abweichen. Nach der Installation prüfen, ob der Befehl gefunden wird:

```
which opencode
```

Falls `which opencode` nichts zurückgibt, den tatsächlichen Installationspfad suchen:

```
find ~ -name "opencode" -type f 2>/dev/null
```

Den gefundenen Verzeichnispfad (ohne den Dateinamen) in die Shell-Konfigurationsdatei eintragen – hier am Beispiel des üblichen Pfads:

zsh:

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

bash:

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bash_profile
source ~/.bash_profile
```

Alternativ ein neues Terminalfenster öffnen. Danach nochmals prüfen:

```
which opencode
opencode --version
```

Variante 2 – npm:

Voraussetzung: Node.js und npm müssen installiert sein. Falls nicht vorhanden:

```
brew install node
```

Versionen prüfen:

```
node --version
npm --version
```

OpenCode installieren:

```
npm install -g opencode-ai
```

npm-Pakete werden in das globale npm-Bin-Verzeichnis installiert. Bei Installation über Homebrew ist dieses automatisch im PATH. Bei Installation über nvm muss nvm in der Shell-Konfigurationsdatei eingebunden sein:

zsh (`~/.zshrc`):

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
```

bash (`~/.bash_profile`):

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
```

Hinweis: Installationsmethoden und verfügbare Versionen ändern sich. Die aktuelle Installationsanleitung sollte direkt von der offiziellen OpenCode-Dokumentation bezogen werden. Skripte aus dem Internet vor der Ausführung prüfen.

Konfiguration (opencode.json)

Die Konfigurationsdatei liegt unter:

```
~/.config/opencode/opencode.json
```

Verzeichnis erstellen und Datei anlegen:

```
mkdir -p ~/.config/opencode
nano ~/.config/opencode/opencode.json
```

Beispielkonfiguration für Ollama mit Gemma 4:

```
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      },
      "models": {
        "gemma4": {
          "name": "Gemma 4"
        }
      }
    }
  },
  "model": "ollama/gemma4",
  "permission": {
    "bash": "allow",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
    "edit": "ask",
    "write": "ask",
    "task": "deny",
    "webfetch": "deny"
  }
}
```

Konfiguration: Felder erklärt

`provider`: Definiert, welche KI-Anbieter verwendet werden sollen. Im Beispiel wird Ollama als lokaler Provider konfiguriert.

`npm`: OpenCode verwendet intern die Vercel AI SDK-Bibliothek. Dieser Wert gibt an, welches npm-Paket für diesen Provider verwendet wird.

`baseURL`: Die Adresse, unter der Ollama erreichbar ist. Muss auf den OpenAI-kompatiblen Endpunkt zeigen (`/v1`).

`models`: Welche Modelle dieses Providers verfügbar sind. Der Key (`gemma4`) muss dem Ollama-Modellnamen entsprechen.

`model`: Das standardmäßig verwendete Modell im Format `provider/modell`.

Rechteverwaltung

Die `permission`-Sektion ist sicherheitskritisch. Sie legt fest, welche Aktionen OpenCode durchführen darf.

Wert	Bedeutung
`"allow"`	Wird automatisch ausgeführt, ohne Nachfrage
`"ask"`	OpenCode fragt den Nutzer vor jeder Aktion
`"deny"`	Wird grundsätzlich nicht erlaubt

Empfehlung für produktive Nutzung:

```
"permission": {
  "bash": "ask",
  "read": "allow",
  "glob": "allow",
  "grep": "allow",
  "edit": "ask",
  "write": "ask",
  "task": "deny",
  "webfetch": "deny"
}
```

- `bash: ask` statt `allow` reduziert das Risiko unerwünschter Befehlsausführung
- `webfetch: deny` verhindert, dass Daten über Netzwerkanfragen das System verlassen
- `task: deny` deaktiviert autonome Agenten-Aufgaben

Die Rechte sollten so restriktiv wie möglich gesetzt werden, besonders wenn OpenCode auf einem Rechner mit Zugang zu internen Systemen läuft.

OpenCode starten

```
cd /pfad/zu/deinem/projekt
opencode
```

Modell in der laufenden Session wechseln:

```
/model
```

Wichtige Verzeichnisse

Pfad	Inhalt
`~/.config/opencode/opencode.json`	Hauptkonfiguration
`~/.ollama/`	Ollama-Daten und Logs
`~/.ollama/models/`	Gespeicherte Modelle
`~/.local/bin/opencode`	OpenCode-Binary (Installationsskript)
`~/.zprofile`	Homebrew PATH – zsh (Apple Silicon)
`~/.bash_profile`	Homebrew PATH – bash (Apple Silicon)
`~/.zshrc`	PATH für `~/.local/bin`, Ollama-Umgebungsvariablen, nvm – zsh
`~/.bashrc` / `~/.bash_profile`	PATH für `~/.local/bin`, Ollama-Umgebungsvariablen, nvm – bash

Wann OpenCode, wann eigene API-Lösung?

OpenCode ist sinnvoll, wenn:

- Schneller Einstieg ohne eigene Entwicklung gewünscht ist
- Interaktives Arbeiten im Terminal bevorzugt wird
- Dateien lesen und ändern Teil des Workflows ist
- Kein eigenes Tool-System entwickelt werden soll

Eigene API-Integration ist sinnvoller, wenn:

- Spezifische Workflows oder Datenquellen angebunden werden sollen
- Vollständige Kontrolle über Prompt-Aufbau und Kontext erforderlich ist
- Integration in bestehende Anwendungen, CI/CD-Systeme oder interne Tools benötigt wird
- Datenschutzanforderungen exaktes Logging und Kontrolle über alle gesendeten Daten erfordern
- Das Ergebnis der KI-Analyse programmatisch weiterverarbeitet werden soll

Kernaussage: OpenCode ist optional. Ollama und die HTTP-API sind das Fundament. Wer eigene Anwendungen baut, braucht OpenCode nicht.

9. Claude Code im Vergleich

Dieser Abschnitt erklärt kurz, was Cloud-basierte Werkzeuge wie Claude Code zusätzlich bieten – nicht als Empfehlung, sondern als sachlicher Vergleich.

Was Claude Code ist

Claude Code ist ein KI-Entwicklungswerkzeug von Anthropic, das aus mehreren Komponenten besteht:

- Das KI-Modell (Claude, läuft in der Anthropic-Cloud)
- Ein Tool-System (Dateizugriff, Terminal, Git-Integration)
- Kontextverwaltung (automatische Projektanalyse)
- Agentenlogik (mehrschrittige Aufgaben autonom durchführen)

Vergleichstabelle

Merkmal	Ollama + eigene Integration	OpenCode + Ollama	Claude Code
Datenlage	Vollständig lokal	Vollständig lokal	Cloud (Anthropic)
Modellqualität	Abhängig vom Modell	Abhängig vom Modell	Hoch (Cloud-Modell)
Tool-System	Selbst entwickeln	Vorhanden	Vorhanden
Kontextverwaltung	Selbst entwickeln	Vorhanden	Automatisch
Datenschutz	Vollständig kontrollierbar	Vollständig kontrollierbar	Anbieterabhängig
Entwicklungsaufwand	Hoch	Mittel	Gering
Laufende Kosten	Hardware	Hardware	Pro Token / Abo
Modellstärke	50–85 % (Schätzung)	50–85 % (Schätzung)	Referenz

Was bei lokalen Systemen selbst entwickelt werden muss

Wer Ollama direkt nutzt und OpenCode nicht einsetzt, übernimmt selbst folgende Aufgaben:

- Relevante Dateien im Projekt suchen und filtern
- Dateiinhalte lesen und in den Prompt einbauen
- Kontextfenster-Limits berücksichtigen (zu lange Dateien kürzen)
- Prompt-Struktur definieren
- Antworten parsen und darstellen
- Dateien schreiben und ändern
- Fehlerbehandlung für API-Aufrufe
- Timeout-Management

Das ist kein prinzipielles Problem, erfordert aber Entwicklungsaufwand.

10. Fehlerdiagnose

Ollama antwortet nicht

Symptom:

```
curl: (7) Failed to connect to localhost port 11434: Connection refused
```

Lösung: Ollama-Server starten:

```
ollama serve
```

Prüfen ob der Prozess läuft:

```
lsof -i :11434
```

Falscher Modellname

Symptom:

```
{"error":"model \"gemma_4\" not found, try pulling it first"}
```

Lösung: Installierte Modelle prüfen:

```
ollama list
```

Exakten Namen aus der Ausgabe in den API-Aufruf übernehmen.

Modell nicht installiert

Lösung:

```
llama pull gemma4
```

Zu wenig RAM

Symptome:

- Extrem langsame Antworten
- System reagiert kaum noch
- Abstürze oder eingefrorene Antworten

Lösung:

- Kleineres Modell wählen
- Andere Anwendungen schließen
- `ollama rm` für nicht benötigte Modelle aufrufen

RAM-Auslastung beobachten:

```
 macOS
top -o MEM
```

OpenCode findet Ollama nicht

Prüfen: `baseURL` in der Konfiguration:

```
"options": {
  "baseURL": "http://localhost:11434/v1"
}
```

Testen ob der Endpunkt antwortet:

```
curl http://localhost:11434/v1/models \
  -H "Authorization: Bearer ollama"
```

Timeout bei langen Anfragen

Ursache: Die Standard-Timeout-Einstellung des HTTP-Clients ist zu kurz für große Modelle oder lange Prompts.

Lösung: Timeout auf mindestens 5 Minuten setzen. Bei sehr großen Kontexten ggf. 10 Minuten.

Antwort ist leer oder abgebrochen

Mögliche Ursachen:

- `num_predict` zu niedrig gesetzt (Antwort wird vorzeitig beendet)
- Kontextfenster überschritten (`num_ctx` erhöhen)
- Modell wurde während der Generierung gestoppt

Temperaturentwicklung auf MacBooks

Intensive KI-Berechnungen können MacBooks stark aufheizen. Symptome sind hohe Lüfterdrehzahl und thermisches Drosseln (Throttling), das die Geschwindigkeit reduziert.

Empfehlung: Für längere Sessions den Mac an einer Oberfläche mit guter Belüftung betreiben. Große Modelle nicht gleichzeitig mehrfach starten.

11. Empfehlungen aus der Praxis

Einstieg

12. Ollama installieren und mit `ollama serve` starten
13. Ein kleines Modell (z. B. `gemma4` in der Standard-Größe) herunterladen
14. Erst mit `ollama run gemma4` im Terminal testen
15. Dann einfache `curl`-Aufrufe ausprobieren
16. Erst danach eigene Integration oder OpenCode einrichten

Modellauswahl

- Für einfache Aufgaben und schnelles Testen: kleines Modell wählen
- Für produktive Nutzung: mittleres Modell als Kompromiss aus Qualität und Geschwindigkeit
- Große Modelle nur wenn die Hardware ausreichend RAM hat
- Verschiedene Modelle für verschiedene Aufgabentypen testen

Sicherheit in Firmenumgebungen

- Ollama nur auf `localhost` betreiben, solange kein Netzwerkbetrieb benötigt wird
- Wenn Netzwerkbetrieb: Firewall-Regeln definieren, Zugriff auf autorisierte Rechner beschränken
- Nur Modelle aus vertrauenswürdigen Quellen (offizielle Ollama-Registry) einsetzen
- Keine Zugangsdaten, API-Keys oder private Schlüssel in Prompts aufnehmen
- Für sensible Umgebungen: internes Testsystem aufsetzen, bevor produktiver Betrieb startet
- OpenCode-Berechtigungen so restriktiv wie möglich konfigurieren

Prompt-Qualität

- Kurze, präzise Prompts liefern oft bessere Ergebnisse als lange, vage Beschreibungen
- System-Prompt für Kontext und Einschränkungen nutzen
- Ausgabeformat im Prompt definieren (z. B. „Antworte nur mit Swift-Code, keine Erklärungen")
- KI-Ausgaben immer prüfen, nie blind übernehmen

Erwartungsmanagement

Lokale KI mit Gemma 4 ist ein nützliches Werkzeug für Routineaufgaben wie Code-Dokumentation, einfache Fehlersuche und Übersetzungen. Für komplexe architektonische Entscheidungen, tiefe Sicherheitsanalysen oder sehr große Codebasen sind aktuelle lokale Modelle noch deutlich schwächer als Cloud-Lösungen.

Der Hauptvorteil lokaler KI liegt nicht in der Modellqualität, sondern in der Datensouveränität: Die Daten bleiben auf der eigenen Infrastruktur.

Letzte Aktualisierung: Mai 2026

Alle Benchmark-Angaben und relativen Leistungsschätzungen sind nicht-offizielle Orientierungswerte aus der Praxis. Modellnamen, Versionsnummern und API-Details können sich ändern. Immer die aktuelle Dokumentation der jeweiligen Projekte prüfen.