Lokale KI für Apple-Entwickler mit Ollama, OpenCode und Coding-Modellen

Die komplette Anleitung

Von den ersten Schritten bis zum Expertenwissen

Autor: Christian Drapatz

Stand: Mai 2026 · Ollama + OpenCode

Plattform: macOS · Apple Silicon

Version 1.0

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/docs, MIT-Lizenz) sowie eigene Erfahrungen, Community-Beiträge und Tests. Technische Fakten wie API-Endpunkte, Konfigurationsoptionen und Befehle stammen aus diesen Dokumentationen und wurden als solche eingearbeitet. 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.*

Inhaltsverzeichnis

1 Einleitung – Worum geht es?

Viele Entwickler arbeiten heute mit Claude Code, ChatGPT, Gemini CLI, Cursor oder GitHub Copilot. Diese Werkzeuge sind stark, arbeiten aber meistens cloudbasiert. Quellcode, Prompts und Projektkontext verlassen dabei häufig den eigenen Rechner.

Ollama geht einen anderen Weg: Es startet KI-Modelle lokal auf deinem Mac und stellt sie über eine lokale API bereit. Ollama ist dabei der lokale Modell-Runner, aber kein Coding-Agent. Es kennt dein Projekt nicht automatisch und ändert keine Dateien von selbst.

OpenCode ist die Agent-Schicht darüber. Es läuft im Terminal im Projektverzeichnis und kann, je nach Konfiguration und Modell, Dateien lesen, Befehle ausführen, Code analysieren und Änderungen vorbereiten.

Kurz gesagt:

```
Ollama = lokaler KI-Motor
```

```
OpenCode = Coding-Agent im Terminal
```

Die Kombination sieht so aus:

```
Apple Silicon Mac
    ↓
Ollama (lokaler Modell-Runner)
    ↓
Lokales Modell (z. B. Qwen, Llama oder DeepSeek)
    ↓
OpenCode (Terminal-Agent)
    ↓
Dein Xcode-/Swift-/macOS-Projekt
```

1.1 Was du aus dieser Anleitung mitnimmst

- Ollama installieren und lokale Modelle starten
- OpenCode als Coding-Agenten nutzen
- Modelle für Apple-Entwicklung kennenlernen und selbst testen
- Sicher und kontrolliert mit lokalen Agenten arbeiten
- Lokale KI sinnvoll mit Cloud-Tools kombinieren

Die Anleitung ist so aufgebaut, dass sie am Anfang für Einsteiger gedacht ist. Je weiter du liest, desto tiefer wird das Wissen. Du kannst direkt in ein Kapitel einsteigen – das Inhaltsverzeichnis zeigt, wo Grundwissen endet und Expertenwissen beginnt.

2 Schnellstart in 10 Minuten

Dieser Abschnitt führt dich in unter zehn Minuten von null zu einem laufenden lokalen Coding-Agenten auf dem Mac. Tiefere Erklärungen zu jedem Schritt folgen in den nächsten Kapiteln.

Schritt 1 – Ollama installieren (1 Minute)

```
brew install ollama
```

Prüfen, ob es funktioniert:

```
ollama --version
```

Schritt 2 – Modell laden (5–8 Minuten, abhängig von Verbindung)

Für den Schnellstart das 7B-Coder-Modell – es lädt schnell und reicht für erste Tests:

```
ollama pull qwen2.5-coder:7b
```

Für ernsthafte Arbeit mit OpenCode wird später ein Modell mit zuverlässigem Tool-Calling empfohlen, zum Beispiel:

```
ollama pull qwen2.5:32b
```

Mehr zu dieser Modellwahl in Kapitel 8 und 14.

Schritt 3 – OpenCode installieren

```
npm install -g opencode-ai

# oder

brew install sst/tap/opencode

# oder

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

Schritt 4 – Git-Branch anlegen (10 Sekunden)

Bevor der Agent irgendeine Datei anfasst:

```
git checkout -b ai/erster-test
```

Das ist kein optionaler Schritt. KI-Agenten können viele Dateien gleichzeitig ändern. Ein Branch erlaubt dir, alles mit `git diff` zu prüfen und bei Bedarf mit `git checkout main` komplett zurückzusetzen.

Schritt 5 – OpenCode starten und Projekt initialisieren

```
cd /Pfad/zu/deinem/Projekt

opencode
```

In der OpenCode-Oberfläche:

```
/init
```

Das erzeugt eine `AGENTS.md` in deinem Projektordner mit einer Zusammenfassung der Projektstruktur und einer sinnvollen Startkonfiguration. Du kannst sie danach anpassen.

Schritt 6 – Erste Frage stellen

Starte immer mit Lesen, nie mit Ändern:

```
Erkläre mir die Architektur dieses Projekts. Ändere keine Dateien.
```

Dann prüfen, dann erst Änderungen erlauben.

Was passiert jetzt im Hintergrund?

```
OpenCode liest deine Projektdateien

    ↓

Das lokale Modell analysiert den Code

    ↓

Du siehst in Echtzeit, welche Dateien gelesen werden
```

```
    ↓
Die Antwort kommt lokal – kein Code verlässt deinen Mac
```

Tipp: Die genauen Erklärungen zu jedem Schritt findest du ab Kapitel 3.

3 Was ist Ollama?

Ollama ist ein lokaler Modell-Runner für macOS, Linux und Windows. Du installierst es auf dem Mac, lädst ein Modell herunter und kannst es direkt starten.

3.1 Typische Ollama-Befehle

Die wichtigsten Befehle im Überblick:

```
# Modell herunterladen
ollama pull qwen2.5:32b

# Modell starten (interaktiver Chat)
ollama run qwen2.5:32b

# Alle lokalen Modelle anzeigen
ollama list

# Server starten (für API-Zugriff durch andere Tools)
ollama serve

# Modell löschen
ollama rm qwen2.5-coder:7b
```

3.2 Die lokale API

Ollama stellt einen lokalen HTTP-Server bereit:

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

Dieser Server bietet zwei API-Stile:

- **Native Ollama-API** unter `/api/...` – z. B. `/api/generate`, `/api/chat`, `/api/tags`.
- **OpenAI-kompatible API** unter `/v1/...` – z. B. `/v1/chat/completions`, `/v1/models`.

Die OpenAI-kompatible Variante ist inzwischen der gemeinsame Nenner vieler Tools. Was genau dahintersteckt, erklärt Kapitel 20.

Wichtig: Das Modell läuft vollständig auf deiner Maschine. Bei rein lokaler Nutzung bleiben Code und Prompts auf dem eigenen Rechner.

3.3 Unterstützte Modell-Familien

Familie	Beispiele	Stärke

Qwen	qwen2.5, qwen2.5-coder, qwen3	Code, Reasoning
DeepSeek	deepseek-r1, deepseek-coder	Analyse, Code
Llama	llama3.3, llama3.2	Allgemein
Mistral	mistral, mixtral	Allgemein
Gemma	gemma3, codegemma	leicht, schnell
Phi	phi4, phi3.5	klein, effizient

Die vollständige Modell-Bibliothek findest du unter ollama.com/library.

4 Was ist OpenCode?

OpenCode ist ein Terminal-Agent für Coding-Aufgaben. Er öffnet keine grafische Oberfläche, sondern läuft direkt im Terminal und arbeitet in deinem Projektordner.

Der Unterschied zu einem normalen Chat-Tool:

Normaler Chat	**OpenCode-Agent**
gibt Antworten	liest und ändert Dateien
kein Datei-Zugriff	arbeitet direkt im Projektordner
kein Shell-Zugriff	kann Befehle ausführen (git, swift, xcode…)
sitzungsbezogen	kennt Projektstruktur und Git-Status

4.1 Provider-Freiheit

OpenCode unterstützt eine große Anzahl an Providern (laut Dokumentation 75+). Dieselbe Oberfläche lässt sich mit Ollama, OpenAI, Anthropic, Gemini oder anderen Anbietern nutzen – auch parallel und mit Wechsel zur Laufzeit.

4.2 Wofür sich OpenCode gut eignet

OpenCode ist besonders gut für:

```
Projekte analysieren

Architektur erklären

Code refactoren

Tests schreiben

Fehler suchen

Dokumentation erstellen
```

Für sehr große, mehrstufige Agent-Aufgaben über viele Dateien sind spezialisierte Tools wie Claude Code aktuell oft komfortabler. Mehr dazu in Kapitel 23.

5 Installation Schritt für Schritt

5.1 Ollama installieren

Mit Homebrew – der empfohlene Weg auf dem Mac:

```
brew install ollama
```

Danach den Server manuell starten (nur nötig, wenn Ollama nicht als Desktop-App installiert ist):

```
ollama serve
```

Hinweis: Wenn du Ollama als macOS-Desktop-App installierst, startet der Server automatisch im Hintergrund. `ollama serve` brauchst du dann nicht – ein zweiter Start würde mit `address already in use` scheitern.

Prüfen, ob der Server läuft:

```
curl http://localhost:11434

# Antwort: Ollama is running
```

5.2 Was macht `ollama serve` genau?

`ollama serve` startet einen HTTP-Server auf Port 11434. Tools wie OpenCode, Continue, Cline, Open WebUI oder eigene Apps sprechen mit diesem Server.

```
Tool (z. B. OpenCode)

    ↓  HTTP-Anfrage an localhost:11434

Ollama-Server

    ↓  lädt Modell in den Unified Memory

Modell (z. B. qwen2.5:32b)

    ↓  generiert Antwort

Ollama-Server

    ↓  HTTP-Antwort

Tool
```

Beim Start passiert:

```
1. ollama serve läuft als Prozess im Hintergrund

2. Port 11434 ist offen und bereit für Anfragen

3. Andere Prozesse auf dem Mac können jetzt anfragen
```

5.3 Modelle laden

Woher bekommt man Ollama-Modelle?

Ollama-Modelle werden über die Ollama Model Library bereitgestellt. Dort findet man unterschiedliche Modellfamilien wie Qwen, Llama, DeepSeek, Mistral, Gemma oder Phi.

Die Modelle werden über das Terminal heruntergeladen und anschließend lokal auf dem Mac ausgeführt. Welche Modelle bereits installiert sind, lässt sich jederzeit mit folgendem Befehl prüfen:

```
ollama list
```

Wichtig: Ein Modell, das in Ollama gut funktioniert, ist nicht automatisch auch für OpenCode geeignet. Einige Modelle liefern direkt in Ollama gute Chat-, Analyse- oder Übersetzungsergebnisse, unterstützen aber keine Tool-Aufrufe.

Wenn OpenCode bei einem Modell meldet:

```
does not support tools
```

kann dieses Modell in OpenCode nicht sinnvoll für Dateioperationen, Bash-Befehle oder agentische Workflows verwendet werden.

Kann man auch andere Modelle anbinden?

Ja. Ollama kann viele verschiedene Modelle ausführen, sofern sie in der Ollama Model Library verfügbar sind oder in einem passenden Format eingebunden werden können.

Für OpenCode reicht es aber nicht, dass ein Modell grundsätzlich in Ollama startet. Entscheidend ist, ob das Modell Tool-Calling zuverlässig unterstützt. OpenCode arbeitet nicht nur mit normalen Textantworten, sondern nutzt Werkzeuge wie bash, read, grep, glob, edit oder write.

Das Modell muss solche Tool-Aufrufe korrekt erzeugen und an OpenCode zurückgeben können. Ein Modell kann also in Ollama problemlos funktionieren, aber in OpenCode trotzdem ungeeignet sein.

Deshalb sollte jedes Modell im eigenen Setup getestet werden, besonders wenn OpenCode Dateien lesen, Befehle ausführen oder Code ändern soll.

Für den produktiven Einsatz mit OpenCode empfiehlt sich ein Modell mit möglichst zuverlässigem Tool-Calling. Normale Chat- oder Reasoning-Modelle können für Erklärungen und Analysen hilfreich sein, sind aber nicht immer für agentische Datei- und Terminaloperationen geeignet.

```
ollama pull qwen2.5:32b
```

Für einfaches Code-Lesen, schnelle Tests oder schwächere Hardware:

```
ollama pull qwen2.5-coder:14b   # empfohlen
ollama pull qwen2.5-coder:7b    # kompakte Alternative
```

Modelle werden einmalig heruntergeladen und bleiben lokal gespeichert – alle weiteren Anfragen starten sofort.

Speicherort auf dem Mac: Ollama speichert alle Modelle im Benutzerverzeichnis:

```
~/.ollama/models/
├── blobs/      # Modelldaten (große Dateien)
└── manifests/  # Zuordnung von Namen und Versionen
```

Modelle nicht manuell löschen. Stattdessen die Ollama-CLI verwenden:

```
ollama list       # installierte Modelle anzeigen
ollama rm <name>  # Modell entfernen
```

5.4 OpenCode installieren und starten

Installation:

```
npm install -g opencode-ai

# oder

brew install sst/tap/opencode
```

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

Starten im Projektverzeichnis:

```
cd /Pfad/zu/deinem/Projekt
opencode
```

OpenCode erkennt einen lokalen Ollama-Server nicht automatisch als Provider. Damit das funktioniert, ist die Konfiguration aus Abschnitt 5.6 nötig.

5.5 Welche anderen Tools können Ollama nutzen?

Der lokale Server `localhost:11434` ist über `/v1` mit der OpenAI-API-Struktur kompatibel. Folgende Tools funktionieren direkt:

Continue (VS Code / JetBrains)

Code-Completion und Chat direkt im Editor. Konfiguration in `~/.continue/config.json`:

```
{
  "models": [
    {
      "title": "Qwen2.5-Coder",
      "provider": "ollama",
      "model": "qwen2.5-coder:14b"
    }
  ]
}
```

Cline (VS Code)

VS-Code-Agent mit Datei- und Shell-Zugriff. In den Cline-Einstellungen:

```
Base URL: http://localhost:11434/v1
API Key:  ollama  (beliebiger String)
Model:    qwen2.5-coder:14b
```

Open WebUI

Lokales Web-Interface, das wie ChatGPT aussieht. Installation mit Docker:

```
docker run -d \
  -p 3000:8080 \
  --add-host=host.docker.internal:host-gateway \
  -v open-webui:/app/backend/data \
  ghcr.io/open-webui/open-webui:main
```

Danach unter `http://localhost:3000` verfügbar. Alle Ollama-Modelle nutzbar, kein Terminal nötig.

Eigene Swift-App

Ollama direkt aus Swift ansprechen, kein SDK nötig:

```
var request = URLRequest(url: URL(string: "http://localhost:11434/api/generate")!)
request.httpMethod = "POST"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")

let body: [String: Any] = [
    "model": "qwen2.5:32b",
    "prompt": "Erkläre Swift Actors in zwei Sätzen.",
    "stream": false
]
request.httpBody = try? JSONSerialization.data(withJSONObject: body)
let (data, _) = try await URLSession.shared.data(for: request)
```

Hinweis: Alle Integrationen sprechen mit demselben lokalen Server. Der Code verlässt zu keiner Zeit den eigenen Rechner.

5.6 Wichtige Konfigurationsdateien von OpenCode

OpenCode legt im Betrieb mehrere Dateien an, die Zustand, Authentifizierung und Konfiguration speichern. Diese Dateien sind wichtig für Diagnose, Anpassung und Verständnis des Systems.

Übersicht

Datei	Speicherort	Zweck	Pflicht?
`opencode.json`	`~/.config/opencode/opencode.json`	Hauptkonfiguration (Provider, Modell, Permissions)	Nein, aber empfohlen
`model.json`	`~/.local/state/opencode/model.json`	Zuletzt verwendete und bevorzugte Modelle	Nein, automatisch
`auth.json`	`~/.local/share/opencode/auth.json`	Authentifizierungstoken für Provider	Nein, automatisch

`~/.config/opencode/opencode.json` – Hauptkonfiguration

Die wichtigste Datei. Hier wird definiert, welcher Provider und welches Modell standardmäßig verwendet werden, welche Berechtigungen OpenCode hat und wie das Tool-Verhalten gesteuert wird.

Muss sie vorhanden sein? Nein. Ohne die Datei startet OpenCode mit Standardeinstellungen, ohne lokalen Ollama-Provider. Für Ollama-Setups ist sie zwingend notwendig.

Kann man sie ändern? Ja, jederzeit. Änderungen gelten beim nächsten Start von OpenCode.

Format: JSON. Optionales JSON-Schema für Autocomplete verfügbar (siehe unten).

Wann muss die Datei geändert werden?

Die opencode.json muss nicht bei jedem Start angepasst werden. Sie wird nur geändert, wenn sich die grundsätzliche Konfiguration von OpenCode ändern soll.

Typische Fälle:

Situation: Neues Ollama-Modell soll verfügbar sein → Eintrag unter provider.ollama.models ergänzen

Situation: Standardmodell soll geändert werden → Feld model anpassen

Situation: Modellname in Ollama weicht von Config ab → Namen in models korrigieren

Situation: Ollama-Endpunkt ändert sich → baseURL unter options anpassen

Situation: Tool-Permissions anpassen → permission-Sektion bearbeiten

Situation: Schreibzugriffe ändern → edit / write in permission setzen

Situation: Subagents oder Webzugriff aktivieren/deaktivieren → task / webfetch in permission setzen

Situation: Anderen Provider verwenden → provider-Sektion entsprechend erweitern

Beispiel: Nach einem ollama pull qwen3:8b muss das neue Modell in der Config ergänzt werden, damit es in OpenCode über /models auswählbar ist.

Nach Änderungen an der opencode.json OpenCode neu starten. Ollama selbst muss dabei normalerweise nicht neu gestartet werden.

Minimale Konfiguration für Ollama

```
{
  "providers": {
    "ollama": {
      "url": "http://localhost:11434"
    }
  },
  "model": "ollama/qwen2.5-coder:14b"
}
```

Vollständiges Beispiel mit mehreren Modellen und Permissions

```
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      },
      "models": {
        "qwen3:8b": {
          "name": "Qwen 3 8B"
        },
        "qwen2.5-coder:14b": {
```

```
          "name": "Qwen 2.5 Coder 14B"
        },
        "qwen2.5:32b": {
          "name": "Qwen 2.5 32B"
        }
      }
    }
  },
  "model": "ollama/qwen2.5:32b",
  "permission": {
    "bash": "allow",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
    "edit": "ask",
    "write": "ask",
    "task": "deny",
    "webfetch": "deny"
  }
}
```

Die `permission`-Sektion steuert, was OpenCode ohne Nachfrage darf. `bash`, `read`, `glob` und `grep` auf `allow` setzen ergibt flüssiges Arbeiten. `edit` und `write` auf `ask` lassen, um ungewollte Dateiänderungen zu verhindern.

Weitere Informationen: opencode.ai/docs/de/config/

Konfigurationsreihenfolge und Priorität

OpenCode liest Konfigurationen in dieser Reihenfolge (höhere Priorität überschreibt niedrigere):

```
1. Remote   → Organisations-Standards via .well-known/opencode
2. Global   → ~/.config/opencode/opencode.json
3. Projekt  → opencode.json im Projektordner
4. Umgebung → OPENCODE_CONFIG-Umgebungsvariable
```

Konfigurationen werden zusammengeführt, nicht ersetzt. Projektspezifische Einstellungen überschreiben nur die Felder, die du angibst.

Die wichtigsten Konfigurationsfelder

```
{
  "model": "ollama/qwen2.5:32b",
  "theme": "dark",
```

```
  "autoupdate": true,
  "permission": {
    "*": "ask"
  },
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      }
    }
  }
}
```

Feld	**Bedeutung**
`model`	Standard-Modell im Format `provider_id/model_id`
`theme`	Farbschema der TUI-Oberfläche
`autoupdate`	automatische Updates aktivieren/deaktivieren
`permission`	Permissions-Regeln (siehe Kapitel 16)
`provider`	Provider-URLs und Optionen
`system`	globaler System-Prompt für alle Sessions
`mcp`	MCP-Server-Konfiguration (siehe Kapitel 21)

Variable Substitution

In der Konfiguration kannst du auf Umgebungsvariablen und Dateien verweisen:

```
{
  "provider": {
    "openai": {
      "apiKey": "{env:OPENAI_API_KEY}"
    }
  },
  "system": "{file:./prompts/system-prompt.md}"
}
```

Praktisch, wenn API-Keys nicht im Klartext in der Konfiguration stehen sollen.

Modell per Kommandozeile überschreiben

Das Modell lässt sich auch direkt beim Start übergeben:

```
opencode --model ollama/qwen2.5:32b
```

```
# oder kurz:
opencode -m ollama/qwen2.5:32b
```

Editor-Unterstützung

Das Schema für Autocomplete und Validierung ist verfügbar unter:

```
https://opencode.ai/config.json
```

In VS Code in der Konfigurationsdatei einbinden:

```
{
  "$schema": "https://opencode.ai/config.json"
}
```

`~/.local/state/opencode/model.json` – Modell-Verlauf

Wird von OpenCode automatisch angelegt und aktualisiert. Speichert, welche Modelle zuletzt verwendet wurden (`recent`) und welche als Favorit markiert sind (`favorite`).

Muss sie vorhanden sein? Nein. OpenCode legt sie beim ersten Modellwechsel automatisch an.

Kann man sie ändern? Ja. Du kannst z. B. die `recent`-Liste kürzen oder die Reihenfolge anpassen. Die Datei wird beim nächsten Modellwechsel überschrieben.

Format: JSON.

```
{
  "recent": [
    { "providerID": "ollama", "modelID": "qwen2.5:32b" },
    { "providerID": "ollama", "modelID": "qwen2.5-coder:14b" }
  ],
  "favorite": [],
  "variant": {
    "ollama/qwen2.5:32b": "default"
  }
}
```

`~/.local/share/opencode/auth.json` – Authentifizierung

Speichert API-Keys und Authentifizierungstoken für konfigurierte Provider. Für Ollama enthält sie einen symbolischen Key (`"ollama"`), da Ollama lokal keine echte Authentifizierung benötigt.

Muss sie vorhanden sein? Nein. OpenCode legt sie automatisch an, wenn ein Provider verbunden wird.

Kann man sie ändern? Ja, aber mit Vorsicht – falsche Einträge können dazu führen, dass Provider nicht erkannt werden.

Format: JSON.

```
{
```

```
  "ollama": {
    "type": "api",
    "key": "ollama"
  }
}
```

Für Cloud-Provider wie Anthropic oder OpenAI stehen hier die echten API-Keys. Die Datei sollte **nicht ins Git-Repository** aufgenommen werden.

Weitere Informationen: opencode.ai/docs/de/providers/

6 Erster Start – was dich erwartet

6.1 Die TUI-Oberfläche

OpenCode zeigt eine **TUI** (Terminal User Interface) – eine interaktive Oberfläche direkt im Terminal. Kein Fenster, keine App, aber eine strukturierte Ansicht mit Eingabefeld, Gesprächsverlauf und Statuszeile.

Beim ersten Start siehst du typischerweise:

```
┌─ OpenCode ──────────────────────────────────────────┐
│  Provider: ollama · Model: qwen2.5:32b              │
│  Working directory: /Users/.../MeinProjekt          │
│                                                     │
│  > _                                                │
└─────────────────────────────────────────────────────┘
```

Du siehst, welcher Provider aktiv ist, welches Modell geladen ist und in welchem Ordner du arbeitest.

6.2 Was OpenCode beim Start tut

OpenCode analysiert beim Start die Struktur deines Projektordners. Es kennt also schon einige Dateien, bevor du die erste Frage stellst.

Wenn OpenCode eine Aufgabe ausführt, siehst du die Aktionen in Echtzeit:

```
Reading:  Sources/App/SearchService.swift
Reading:  Sources/App/NetworkLayer.swift
Running:  grep -r "SearchServiceImpl" .
Writing:  Sources/App/SearchService.swift
```

Das ist der wichtige Unterschied zu normalem Chat: Du siehst, was der Agent tut, nicht nur die fertige Antwort.

6.3 Sicher starten: Git ist Pflicht

Bevor OpenCode Dateien ändern darf, immer einen Branch anlegen:

```
git checkout -b ai/opencode-test
```

Git-Grundkenntnisse sind Voraussetzung. Du solltest verstehen, was `git status`, `git diff`, `git checkout` und `git reset` tun, sonst verlierst du schnell den Überblick.

Wer lieber eine grafische Oberfläche nutzt:

Client	Plattform	Besonderheit
Fork	Mac und Windows	schnell, übersichtlich, kostenlos nutzbar
GitHub Desktop	Mac und Windows	einfach, direkt mit GitHub verbunden
Tower	Mac und Windows	professionell, viele Features, kostenpflichtig
SourceTree	Mac und Windows	kostenlos, gut für Atlassian-Nutzer

6.4 Empfohlener Ablauf für die erste Session

```
1. git checkout -b ai/test

2. opencode

3. /init  (Projekt analysieren lassen)

4. "Erkläre mir dieses Projekt. Ändere keine Dateien."

5. git diff  (nach jeder Änderung prüfen)
```

7 Warum ist das für Apple-Entwickler spannend?

Für Swift- und Xcode-Entwicklung gibt es klare Einsatzfälle:

```
Code erklären

Refactoring vorbereiten

SwiftUI Views analysieren

Unit Tests schreiben

Architektur prüfen

Fehler suchen

CLI-Skripte verbessern

Dokumentation schreiben

Localizable-Dateien prüfen

Commit-Diffs erklären
```

7.1 Besonders interessant für Firmen und Kundenprojekte

Bei Firmenprojekten, Gesundheitsapps, internen Tools oder Kundencode ist lokale KI besonders interessant. Du kannst Code analysieren und bearbeiten lassen, ohne ihn an einen Cloud-Dienst zu schicken.

Hinweis: Das ersetzt Claude Code oder Codex nicht in jedem Szenario. Für viele tägliche Aufgaben reicht es aber überraschend weit.

7.2 Apple Silicon ist gut geeignet

Apple Silicon nutzt eine einheitliche Speicherarchitektur (Unified Memory). CPU, GPU und Neural Engine teilen denselben RAM. Das macht lokale Modelle auf dem Mac in der Praxis effizienter als auf vergleichbarer PC-Hardware mit separatem GPU-VRAM.

8 Lokale Modelle – Orientierung und Erfahrungswerte

> *Die folgenden Modelle sind als Orientierung zu verstehen. Welches Modell im eigenen Setup am besten funktioniert, muss jeder Benutzer selbst testen. Die tatsächliche Qualität hängt stark von Hardware, Modellgröße, Quantisierung, Projektumfang, Prompt-Stil und der Tool-Calling-Unterstützung des Modells ab. Da die Hardware der Leser nicht bekannt ist, können hier nur Erfahrungswerte und Hinweise gegeben werden.*

In den folgenden Abschnitten beschreibe ich drei Modelle, die sich im eigenen Setup als nützlich erwiesen haben. Die Aussagen sind ausdrücklich Einschätzungen aus eigenen Tests, keine allgemeinen Garantien. Ergebnisse können auf anderer Hardware oder mit anderen Versionen von Ollama und OpenCode abweichen.

8.1 Qwen 2.5 32B – aus eigener Erfahrung stark beim Tool-Calling

`qwen2.5:32b` ist das allgemeine Qwen-2.5-Modell in der 32B-Größe. In meinen eigenen Tests mit OpenCode und Ollama hat es sich als das zuverlässigste Modell für **Tool-Calling** gezeigt – also dafür, wann das Modell Befehle ausführt, Dateien liest und Änderungen vornimmt, statt nur JSON-Text auszugeben.

```
ollama pull qwen2.5:32b
```

Mögliche Aufgaben, die in eigenen Tests gut funktionierten:

```
Projektanalyse (Modell liest und versteht Dateien selbstständig)

Fehlersuche über mehrere Dateien

Swift-Code verstehen und erklären

Architekturüberblick erstellen

Befehle ausführen (git, find, grep)
```

Einordnung:

Eigenschaft	**Wert**
Größe	32B (~19 GB auf Disk)
RAM-Bedarf	empfohlen ab 32 GB (besser 48 GB+)
Tool-Calling	in eigenen Tests stabil und korrekt
Geschwindigkeit	mittel bis langsamer (je nach Hardware)
Stärke	Coding, Analyse, Reasoning

> **Hinweis zu den Coder-Varianten:** *In meinen Tests hatten die Coder-Varianten (`qwen2.5-coder:7b`, `qwen2.5-coder:14b`) Probleme mit Tool-Calling über Ollama in OpenCode. OpenCode bekam die Tool-Call-Ausgabe teilweise als reinen Text statt als strukturierten `tool_calls`-API-Aufruf. Das Modell beschrieb Befehle, statt sie auszuführen. `qwen2.5:32b` (ohne `-coder`) zeigte dieses Verhalten in meinen Tests nicht. Diese Beobachtung gilt nicht zwangsläufig universell – siehe auch Abschnitt 8.2.*

8.2 Qwen2.5-Coder – stark für Coding, in eigenen Tests aber nicht zuverlässig beim Tool-Calling

Qwen2.5-Coder ist eines der starken offenen Coding-Modelle für lokale Nutzung. Es gibt sechs Größen: `0.5b`, `1.5b`, `3b`, `7b`, `14b` und `32b`.

```
ollama pull qwen2.5-coder:14b
```

In mehrfachen eigenen Tests zeigte sich:

- **Code-Antworten:** Qwen2.5-Coder kann sehr gute Code-Antworten liefern. Für reines Code-Lesen, -Erklären und Vorschläge ohne Agent-Aktionen ist es eine sinnvolle Wahl.
- **Tool-Calling über Ollama in OpenCode:** funktioniert in eigenen Tests **nicht zuverlässig**. Tool-Aufrufe wurden teilweise nur als JSON ausgegeben oder nicht korrekt ausgeführt. Das ist eine eigene Beobachtung – auf anderer Hardware, mit anderen Modelfile-Varianten oder neueren Ollama-/OpenCode-Versionen kann das Ergebnis abweichen.

Mögliche Aufgaben für die Coder-Varianten – ohne Agent-Aktionen:

```
Swift-Code erklären

SwiftUI Views analysieren

Code-Vorschläge zur manuellen Übernahme

DTOs und Services entwerfen

Fehler im Code beschreiben
```

Einordnung nach Größe:

Größe	Geschwindigkeit	Hinweis
7B	schnell	gut für kleine, klare Aufgaben
14B	mittel	brauchbarer Einstieg für Code-Aufgaben
32B	langsamer	stärker, braucht mehr RAM

Konsequenz: Wer OpenCode + Ollama mit echtem Tool-Calling nutzen möchte, fährt aktuell – nach eigener Erfahrung – mit `qwen2.5:32b` ruhiger als mit den Coder-Varianten. Wer Qwen2.5-Coder als reines Code-Modell ohne Agent-Aktionen einsetzen möchte, kann das parallel tun.

9 Hardware-Empfehlung für Apple Silicon

RAM ist die wichtigste Ressource für lokale Modelle auf dem Mac. Die folgenden Werte sind grobe Richtwerte – die tatsächliche Nutzbarkeit hängt zusätzlich von Modellgröße, Quantisierung und parallel laufenden Apps ab.

Mac-RAM	Sinnvolle Modelle
8 GB	kleine Modelle (3B/7B), nur eingeschränkt
16 GB	Qwen2.5-Coder 7B, kleinere DeepSeek-Varianten
24 GB	Qwen2.5-Coder 14B sinnvoll nutzbar
32 GB	Qwen2.5-Coder 14B, DeepSeek R1 14B gut nutzbar
48–64 GB	Qwen 2.5 32B, Qwen2.5-Coder 32B, DeepSeek R1 32B
96–128 GB	große 70B-Modelle möglich, aber langsamer

9.1 Empfehlung für die meisten Apple-Entwickler

Mit 32 GB RAM ein guter Startpunkt:

```
ollama pull qwen2.5-coder:14b
```

Mit Mac Studio oder MacBook Pro (48 GB+):

```
ollama pull qwen2.5:32b
```

Tipp: Wer ernsthaft mit lokalen Modellen arbeiten möchte, sollte beim nächsten Mac-Kauf mindestens 32 GB RAM einplanen.

10 Tastenkürzel in OpenCode

OpenCode nutzt ein **Leader-Key-System**: Die meisten Aktionen werden mit einer Tastenfolge ausgelöst, nicht mit einer einzelnen Taste. Das vermeidet Konflikte mit dem Terminal.

10.1 Der Leader Key

```
Standard: ctrl+x
```

Beispiel: Neue Session anlegen = `ctrl+x` → dann `n`

10.2 Die wichtigsten Tastenkürzel

Taste / Kombination	Was passiert
`Tab`	Plan Mode aktivieren / wechseln
`Shift+Tab`	Agent wechseln
`ctrl+x n`	Neue Session starten
`ctrl+x l`	Session-Liste anzeigen
`Bild↑ / Bild↓`	Im Gesprächsverlauf scrollen
`ctrl+a`	Zum Anfang der Eingabezeile
`ctrl+e`	Zum Ende der Eingabezeile
`ctrl+c`	Aktuelle Aktion abbrechen

10.3 Plan Mode vs. Build Mode

OpenCode kennt zwei Modi:

```
Plan Mode  (Tab)   → Agent erstellt nur einen Plan, ändert keine Dateien

Build Mode         → Agent führt Änderungen aus
```

Empfehlung: Im Plan Mode starten, den Plan lesen, dann in den Build Mode wechseln. Das gibt Kontrolle über das, was als Nächstes passiert.

10.4 Tastenkürzel anpassen

Tastenkürzel können in `~/.config/opencode/tui.json` angepasst werden. Einzelne Shortcuts lassen sich deaktivieren:

```
{
```

```
  "keybinds": {
    "session:new": "none"
  }
}
```

11 Projektregeln und Instruktionsdateien

KI-Agenten können sehr mächtig sein – ohne klare Vorgaben tun sie aber manchmal Dinge, die du nicht willst. Projektregeln sind die Lösung: Dateien, die dem Agenten beim Start sagen, wie er sich in deinem Projekt verhalten soll.

11.1 Was ist CLAUDE.md – und was hat das mit OpenCode zu tun?

`CLAUDE.md` ist eine Instruktionsdatei aus dem Ökosystem von **Claude Code** (dem KI-Coding-Agenten von Anthropic). Die Idee: Du legst eine Textdatei im Projektordner ab, und der Agent liest sie beim Start als Regelsatz.

OpenCode hat dieses Konzept übernommen und erweitert. Bei OpenCode heißt die primäre Datei **`AGENTS.md`**.

Tool-Varianten:

Dateiname	Tool
`AGENTS.md`	OpenCode (primäre Datei)
`CLAUDE.md`	Claude Code (Anthropic)
`RULES.md`	allgemeine Konvention

Hinweis: OpenCode liest auch `CLAUDE.md` und den globalen Claude-Fallback unter `~/.claude/CLAUDE.md`. Wer Claude Code kennt, kann dieselbe Datei für beide Tools nutzen.

11.2 Wo liegen diese Dateien – globale vs. projektspezifische Regeln

Es gibt zwei Ebenen:

```
~/                                    ← Home-Verzeichnis
├── .claude/
│   └── CLAUDE.md                     ← Claude-Code-Fallback (auch von
OpenCode gelesen)
├── .config/
│   └── opencode/
│       ├── opencode.json             ← globale OpenCode-Konfiguration
│       └── AGENTS.md                 ← globale Regeln für alle Projekte
└── GIT-Home/
    └── MeinProjekt/
        ├── AGENTS.md                 ← projektspezifische Regeln
        ├── opencode.json             ← projektspezifische Konfiguration
```

```
└── Sources/
```

11.3 In welcher Reihenfolge sucht OpenCode nach Regeldateien?

OpenCode sucht beim Start in dieser Reihenfolge:

```
1. Aktuelles Verzeichnis → Elternverzeichnis → weiter aufwärts  (AGENTS.md, CLAUDE.md)
2. Globale Datei: ~/.config/opencode/AGENTS.md
3. Claude-Code-Fallback: ~/.claude/CLAUDE.md
```

OpenCode sucht also nicht nur im Startverzeichnis, sondern auch in übergeordneten Ordnern. Wenn du `AGENTS.md` im übergeordneten Ordner ablegst, gilt sie für alle Projekte darunter.

Wichtig: Die erste gefundene Datei pro Ebene gewinnt – Dateien werden nicht automatisch zusammengeführt.

11.4 Automatische Erzeugung mit `/init`

Du musst `AGENTS.md` nicht von Hand schreiben. Der Befehl `/init` analysiert dein Projekt und erzeugt eine passende Datei:

```
/init
```

OpenCode scannt dann:

- Projektstruktur und Ordner
- verwendete Frameworks und Libraries
- erkannte Architekturmuster
- vorhandene Tests und CI-Konfiguration

Das Ergebnis ist eine fertige `AGENTS.md` in deinem Projektordner, die du anschließend anpassen kannst.

11.5 Wie sage ich OpenCode explizit, welches Regelwerk gelten soll?

Weg 1 – Automatisch über Datei im Projektordner

Die einfachste Methode: `AGENTS.md` im Projektordner ablegen. OpenCode liest sie beim Start automatisch.

Weg 2 – Global über AGENTS.md im Config-Ordner

Regeln, die für alle Projekte gelten sollen:

```
~/.config/opencode/AGENTS.md
```

Weg 3 – System-Prompt in opencode.json

In `~/.config/opencode/opencode.json` oder im Projekt-`opencode.json`:

```
{
  "system": "Kommentare immer auf Deutsch. Keine force unwraps. Frage vor Dateiänderungen."
}
```

Weg 4 – Direkt im Prompt zu Beginn der Session

Falls keine Datei vorhanden ist, lassen sich Regeln direkt mitgeben:

```
Beachte für diese Session: Keine force unwraps, Kommentare auf Deutsch,
```

```
zeige mir Änderungen vor der Ausführung.
```

Das gilt nur für die aktuelle Session. Nach `/clear` ist es vergessen.

11.6 Beispiel-AGENTS.md für ein Swift-Projekt

```
# Projektregeln für MeinProjekt

## Architektur
- MVVM-Muster beibehalten
- Netzwerklogik gehört in Services, nicht in ViewModels
- Keine Logik direkt in SwiftUI Views

## Code-Stil
- Keine force unwraps (`!`)
- Keine impliziten force casts (`as!`)
- Code-Bezeichner auf Englisch
- Kommentare und DocC auf Deutsch

## Sicherheitsregeln
- SwiftLint muss grün bleiben
- Keine neuen Dependencies ohne Rückfrage
- Keine Änderungen an CI-Skripten
- Tests nie löschen

## Arbeitsweise
- Zeige mir zuerst alle Dateien, die du ändern würdest
- Warte auf meine Bestätigung vor Änderungen
- Bei Unsicherheit: nachfragen statt raten
```

11.7 Empfohlene Inhalte

```
1. Architekturvorgaben (MVVM, Clean Architecture, …)
2. Code-Stil (Kommentare, Sprache, Naming)
3. Verbote (force unwrap, bestimmte Patterns)
4. Shell-Einschränkungen (was darf ausgeführt werden)
5. Rückfragepflichten (wann muss nachgefragt werden)
6. Lint- und Test-Anforderungen
```

Tipp: Mit `/init` starten, die erzeugte Datei lesen, die Regeln anpassen. Eine wachsende `AGENTS.md` ist ein gutes Zeichen, dass der Agent produktiv genutzt wird.

12 Gedächtnis und Kontext

OpenCode hat kein echtes Langzeitgedächtnis. Es gibt aber mehrere Ebenen, die zusammen wie Gedächtnis wirken.

12.1 Session-Kontext

Die aktuelle Unterhaltung bleibt während der Session erhalten. In eigenen Tests verlieren lokale 14B-Modelle Kontext schneller als Cloud-Modelle.

12.2 Repository-Kontext

OpenCode liest aktiv Dateien, Ordner und Git-Historie. Das wirkt wie Gedächtnis, ist aber eigentlich laufende Projektanalyse.

12.3 Instruktionsdateien als dauerhaftes Gedächtnis

Das ist die wichtigste Form. Dateien wie `AGENTS.md` geben Regeln, Architektur und Stil vor und sind bei jedem Start wieder verfügbar. Mehr dazu in Kapitel 11.

12.4 Erfahrung mit lokalen Modellen

Nach eigener Erfahrung vergessen lokale Modelle schneller. Cloud-Modelle wie Claude Opus oder GPT-4 behalten Architektur, Regeln und Zusammenhänge länger.

Gedächtnistyp	**Lokal (14B)**	**Claude Opus**
Session-Kontext	mittel	sehr gut
Architektur-Regeln	mit Datei gut	sehr gut
Lange Dateiketten	eingeschränkt	deutlich besser
Agent-Schritte	5–10 stabil	20+ stabil

Diese Werte sind Erfahrungseinschätzungen, keine Benchmarks.

13 Interne Befehle und Slash-Kommandos

OpenCode hat eingebaute Befehle, die du jederzeit in der TUI eingeben kannst. Sie steuern das Verhalten des Agents, ohne eine neue Frage zu stellen.

13.1 Slash-Befehle im Überblick

Befehl	**Was passiert**
`/init`	Projekt analysieren und AGENTS.md automatisch erzeugen
`/help`	alle verfügbaren Befehle anzeigen
`/model`	Modell wechseln (z. B. `/model qwen2.5:32b`)
`/models`	Liste aller konfigurierten Modelle anzeigen
`/clear`	Gesprächsverlauf zurücksetzen

`/undo`	letzte Dateiänderungen rückgängig machen
`/redo`	rückgängig gemachte Änderungen wiederherstellen
`/share`	Link zur aktuellen Unterhaltung erzeugen
`/connect`	neuen Provider verbinden
`/summarize`	Zusammenfassung des bisherigen Kontexts erzeugen

13.2 Modell wechseln

Mit `/model` lässt sich das Modell mitten in einer Session tauschen, etwa von einem schnellen 7B auf ein stärkeres Modell, wenn eine Aufgabe komplexer wird:

```
/model qwen2.5:32b
```

13.3 Session zurücksetzen

Wenn der Kontext zu groß wird oder das Modell Anweisungen vergisst:

```
/clear
```

Das löscht den Gesprächsverlauf, aber nicht die Projektdateien oder die `AGENTS.md`.

13.4 Änderungen rückgängig machen

Wenn OpenCode etwas geändert hat, das du nicht wolltest:

```
/undo
```

Das rollt die letzten Dateiänderungen zurück. Mehrfaches `/undo` geht weiter zurück.

13.5 Dateien und Shell-Output im Prompt referenzieren

Du kannst Dateien direkt im Prompt per `@`-Syntax einbinden – OpenCode sucht dann per Fuzzy-Search:

```
Erkläre mir @SearchService und wie er mit @NetworkLayer zusammenhängt.
```

Shell-Ausgaben lassen sich direkt einbetten:

```
Der Befehl `!swift test` schlägt fehl. Was ist falsch?
```

OpenCode führt den Befehl aus und gibt die Ausgabe als Kontext mit.

13.6 Hilfe anzeigen

`/help` zeigt alle aktuell verfügbaren Befehle der installierten Version:

```
/help
```

Hinweis: Die Befehle entwickeln sich mit jeder Version weiter. Im Zweifel `/help` eingeben.

14 Praxistest: Vollständige Einrichtung

Dieser Abschnitt dokumentiert eine vollständige Einrichtung und einen Praxistest mit Ollama und OpenCode auf macOS mit Apple Silicon. Die Ergebnisse ergänzen die theoretischen Kapitel mit realen Beobachtungen – insbesondere zu Tool-Calling-Verhalten, Konfiguration und Prompt-Strategie. Der Abschnitt enthält bewusst praktische Wiederholungen aus früheren Kapiteln, damit er als eigenständige Anleitung lesbar bleibt.

>

Getestet mit: Ollama 0.22.0, qwen2.5:32b, Apple M4 Max (28 GB RAM), OpenCode aktuell – Mai 2026

Schritt 1 – Ollama installieren

```
brew install ollama
```

```
ollama --version
```

Schritt 2 – Das richtige Modell wählen

Die Modellwahl ist die wichtigste Entscheidung. Nicht jedes Modell unterstützt Tool-Calling in OpenCode + Ollama gleich zuverlässig. Tool-Calling ist die Voraussetzung dafür, dass OpenCode Befehle wirklich ausführt, statt nur JSON-Text auszugeben.

Empfehlung aus eigenen Tests: `qwen2.5:32b`

```
ollama pull qwen2.5:32b
```

```
ollama list
```

Eigene Testergebnisse im Vergleich

Diese Tabelle fasst eigene Beobachtungen zusammen. Auf anderer Hardware oder mit neueren Versionen kann das Verhalten abweichen.

Modell	Tool-Calling (eigene Tests)	Eignung	Anmerkung
`qwen2.5:32b`	korrekt	gut für Code	empfohlen, ~19 GB
`llama3.2:latest`	korrekt	eingeschränkt	nur 3B – zu schwach für viele Code-Aufgaben
`qwen2.5-coder:14b`	nicht zuverlässig	nicht für Agent-Modus	Tool-Calls erschienen teils als Text
`hhao/qwen2.5-coder-tools:14b`	nicht zuverlässig	nicht für Agent-Modus	Community-Variante, gleiches Problem

Mögliche Erklärung: *Das Ollama-Modelfile mancher Modelle scheint kein passend konfiguriertes Tool-Calling-Template zu enthalten. Ollama kann die Tool-Call-Ausgabe dann nicht in das* `tool_calls`*-API-Feld übersetzen, sondern liefert sie als Text aus. Diese Einschätzung beruht auf eigenen Tests, kein offizielles Statement.*

Tool-Calling eines Modells manuell prüfen

```
curl http://localhost:11434/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen2.5:32b",
    "messages": [{"role":"user","content":"list files"}],
    "tools": [{"type":"function","function":{"name":"bash","description":"run shell","parameters":{"type":"object","properties":{"command":{"type":"string"}},"required":["command"]}}}]
  }'
```

- Antwort enthält `"finish_reason": "tool_calls"` → Modell unterstützt Tool-Calling im Test
- Antwort enthält `"finish_reason": "stop"` mit JSON im `content`-Feld → Modell liefert Tool-Aufruf als Text statt strukturiert

Schritt 3 – OpenCode installieren

```
npm install -g opencode-ai

# oder

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

# oder

brew install sst/tap/opencode
```

Schritt 4 – Konfiguration anlegen

OpenCode erkennt Ollama nicht automatisch als Provider. Die Konfiguration muss manuell angelegt werden:

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

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

Inhalt (im eigenen Setup funktionierende Konfiguration):

```
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      },
      "models": {
        "qwen2.5:32b": {
          "name": "Qwen 2.5 32B"
        }
      }
    }
  },
  "model": "ollama/qwen2.5:32b",
  "permission": {
    "bash": "allow",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
```

```
    "edit": "ask",
    "write": "ask",
    "task": "deny",
    "webfetch": "deny"
  }
}
```

`bash`, `read`, `glob` und `grep` auf `allow` setzen, sonst fragt OpenCode bei jedem Dateizugriff nach. `edit` und `write` auf `ask` lassen für Kontrolle über Dateiänderungen.

Schritt 5 – OpenCode starten

Ollama muss im Hintergrund laufen. Die macOS-App startet den Daemon automatisch (Icon in der Menüleiste). Falls nötig:

```
# Prüfen, ob Ollama läuft (Port 11434):
pgrep -fl ollama || open -a Ollama


# Ins Projektverzeichnis und OpenCode starten:
cd ~/MeinProjekt
opencode
```

> **Hinweis:** `ollama serve` *ist* **nicht** *nötig, wenn die Ollama-App installiert ist. Ein zweiter Start würde mit* `address already in use` *scheitern.* `ollama serve` *nur ohne Desktop-App verwenden (reine CLI-Installation).*

Schritt 6 – Prompts richtig formulieren

Mit lokalen Modellen muss man den Agenten in eigener Erfahrung stärker führen als mit Cloud-Tools wie Claude Code. Das Modell sollte explizit angewiesen werden, welche Schritte es ausführen soll. Allgemeine Prompts wie „Analysiere dieses Projekt“ funktionieren nicht immer zuverlässig.

Projektanalyse:

```
Antworte auf Deutsch.


Analysiere dieses Swift-Projekt vorsichtig und schrittweise.


Wichtig:
- Ändere keine Dateien.
- Erstelle keine Dateien.
- Nutze keine Subagents.
- Nutze kein webfetch.
- Lies zuerst die Projektstruktur.
- Erfinde keine Abhängigkeiten oder Architektur-Schichten.
```

Prüfe zuerst:

- pwd

- ls -la

- find . -maxdepth 4 -type f \(-name "*.swift" -o -name "Package.swift" -o -name "*.pbxproj" \) | sort

Danach lies die wichtigsten Dateien und erstelle eine Analyse:

1. Projektart

2. Erkennbare Schichten

3. Abhängigkeiten

4. Problematische Stellen

5. Empfohlene nächste Schritte

Fehlersuche in einer Datei:

Antworte auf Deutsch.

Ändere keine Dateien.

Suche im Projekt nach einer Datei, die "DependencyContainer" im Namen enthält:

```
find . -name "*DependencyContainer*.swift"
```

Lies danach die gefundene Datei ein.

Prüfe auf offensichtliche Fehler, fehlende Imports, falsche Initialisierung oder Syntax-Probleme.

Analysiere nur den tatsächlich gelesenen Inhalt.

Syntaxprüfung aller Swift-Dateien:

Antworte auf Deutsch.

Ändere keine Dateien.

Führe aus:

```
find . -name "*.swift" | sort
```

Lies jede Datei einzeln ein.

Prüfe auf Syntax-Fehler (falsche Schlüsselwörter, fehlende Klammern, falsche Deklarationen).

Melde nur echte Fehler aus dem gelesenen Inhalt.

Troubleshooting aus der Praxis

Problem: OpenCode zeigt JSON statt Ergebnisse

```
{
  "name": "bash",
  "arguments": { "command": "ls" }
}
```

Mögliche Ursache: Das Modell unterstützt Tool-Calling in der vorliegenden Ollama-Variante nicht zuverlässig.

Lösung: Modell wechseln (z. B. auf `qwen2.5:32b`), `opencode.json` aktualisieren, OpenCode neu starten.

Problem: OpenCode erklärt Befehle, führt sie aber nicht aus

Symptom: OpenCode schreibt „Führe `cat Datei.swift` aus und füge den Inhalt hier ein", statt die Datei selbst zu lesen.

Mögliche Ursache: Das Modell verhält sich wie ein Chatbot und nutzt Tools nicht selbstständig.

Lösung: Prompt expliziter formulieren:

```
Nutze bash direkt.
Führe selbst aus: find . -name "DependencyContainer.swift"
Lies danach die gefundene Datei selbst ein.
Warte nicht auf meine Eingabe.
```

Problem: Ollama reagiert nicht

```
lsof -i :11434                          # prüfen, ob Port belegt
ollama serve                            # Ollama manuell starten (nur ohne Desktop-App)
curl http://localhost:11434/api/tags    # Verbindung testen
```

OpenCode + Ollama vs. Claude Code – Praxisvergleich

Die folgenden Aussagen sind sachliche Erfahrungswerte. Beide Tools haben ihre Berechtigung – sie spielen einfach in unterschiedlichen Situationen ihre Stärken aus.

Kriterium	OpenCode + Ollama	Claude Code
Tool-Calling	Stark abhängig vom Modell-Template	Nativ, in Tests sehr zuverlässig
Prompt-Führung	Profitiert von expliziten Schritt-Anweisungen	Erfasst Kontext meist automatisch
Selbstständigkeit	Führt Aktionen oft erst auf Anweisung aus	Sucht und liest Dateien selbst
Datenschutz	Vollständig lokal möglich	Cloud-basiert
Kosten	Nach Hardware-Investition kostenfrei	Nutzungsbasierte API-Kosten
Modellqualität	Begrenzt durch lokale Rechenleistung	Zugriff auf große Cloud-Modelle
Konsistenz	Schwankt je nach Modell und Prompt	In Tests gleichbleibend hoch

Fazit: OpenCode mit Ollama ist gut geeignet, wenn Datenschutz oder Offline-Betrieb im Vordergrund stehen. Für komplexe, mehrstufige Aufgaben – Architekturanalysen, Refactorings über viele Dateien, Debugging-Ketten – ist Claude Code aktuell oft komfortabler, weil es Kontext selbst erschließt und mit deutlich leistungsfähigeren Modellen arbeitet.

15 Eingebaute Tools im Überblick

OpenCode bringt eine Reihe eingebauter Tools mit, die das Modell während einer Session nutzen kann. Du musst sie nicht manuell aufrufen – das Modell entscheidet selbst, welche Tools für eine Aufgabe nötig sind.

Tool	Was es tut
`bash`	Shell-Befehle ausführen (git, swift, xcodebuild, …)
`edit`	bestehende Dateien bearbeiten (exakte String-Ersetzung)
`write`	neue Dateien erstellen oder vollständig überschreiben
`read`	Dateiinhalte lesen
`grep`	Dateien mit regulären Ausdrücken durchsuchen
`glob`	Dateien per Musterabgleich finden (z. B. `**/*.swift`)
`lsp`	Code-Intelligence via LSP (Definitionen, Referenzen)
`patch`	Patch-Dateien anwenden
`todowrite`	Todo-Listen während der Session verwalten
`webfetch`	Webseiten abrufen und als Kontext nutzen
`websearch`	Web durchsuchen
`question`	dich während einer Aufgabe um Klärung bitten

Alle Tools lassen sich über das Permissions-System (Kapitel 16) kontrollieren. Einzelne Tools können erlaubt, auf Nachfrage gestellt oder komplett gesperrt werden.

Wichtiger Hinweis bei lokalen Modellen: Damit das Modell diese Tools selbstständig aufruft, ist Tool-Calling-Unterstützung notwendig. In eigenen Tests funktioniert das nicht mit allen Ollama-Modellen gleich zuverlässig (siehe Kapitel 8 und 14).

16 Das Permissions-System

OpenCode hat ein feingranulares System, das steuert, ob Aktionen automatisch ausgeführt werden, ob sie Bestätigung erfordern oder komplett gesperrt sind.

16.1 Die drei Aktionen

Aktion	Bedeutung
`"allow"`	Aktion wird ohne Nachfrage ausgeführt
`"ask"`	OpenCode fragt dich vor der Ausführung

`"deny"`	Aktion ist komplett gesperrt

16.2 Was kann kontrolliert werden?

Kategorie	**Bedeutung**
`read`	Dateien lesen
`edit`	Dateien bearbeiten
`write`	Dateien erstellen/überschreiben
`bash`	Shell-Befehle ausführen
`glob`	Dateien per Muster suchen
`grep`	Dateien durchsuchen
`webfetch`	Webseiten abrufen
`websearch`	Web durchsuchen
`task`	Subagenten starten
`external_directory`	Zugriff auf Ordner außerhalb des Projekts
`doom_loop`	identische Wiederholungsschleifen blockieren

16.3 Konfiguration in opencode.json

Globale Grundregel setzen und einzelne Tools überschreiben:

```
{
  "permission": {
    "*": "ask",
    "read": "allow",
    "bash": "allow",
    "edit": "ask"
  }
}
```

Feingranulare Kontrolle mit Wildcard-Mustern – besonders wichtig für Shell-Befehle:

```
{
  "permission": {
    "bash": {
      "*": "ask",
      "git status": "allow",
      "git diff": "allow",
      "swift test": "allow",
      "rm *": "deny",
      "git reset *": "ask"
```

```
    }
  }
}
```

16.4 Empfehlung für Swift/Xcode-Projekte

```
{
  "permission": {
    "*": "ask",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
    "bash": {
      "*": "ask",
      "git status": "allow",
      "git diff": "allow",
      "swift build": "allow",
      "swift test": "allow",
      "xcodebuild test *": "allow",
      "rm *": "deny",
      "git reset --hard": "deny"
    }
  }
}
```

Tipp: Mit `"*": "ask"` (alles nachfragen) starten, beobachten, was OpenCode tut, und die `allow`-Liste schrittweise erweitern. So lernst du das Verhalten kennen, bevor du Vertrauen aufbaust.

Standardverhalten: `.env`-Dateien sind standardmäßig gesperrt. `external_directory` und `doom_loop` fragen standardmäßig nach.

17 OpenCode als Agent – was er wirklich kann

Viele Entwickler installieren OpenCode und behandeln ihn wie einen normalen Chat. Das ist ein Missverständnis.

OpenCode ist:

- ein **Terminal-Agent**
- ein **Datei-Agent**
- ein **Shell-Agent**
- ein **Workflow-Agent**

17.1 Was wirklich passiert

Wenn du schreibst:

```
Refactore mein Projekt
```

kann OpenCode intern:

- 40 Dateien ändern
- Tests starten
- Bash-Skripte ausführen
- Git-Kommandos starten
- Dateien verschieben

Achtung: Immer vorher `git checkout -b ai/test`. Danach `git diff` prüfen, bevor du etwas übernimmst.

17.2 Verhalten mit lokalen Ollama-Modellen

Ein wichtiger Praxispunkt: OpenCode mit lokalen Ollama-Modellen verhält sich nicht immer wie ein Cloud-Tool wie Claude Code. Aus eigener Erfahrung:

- Allgemeine Prompts wie „Analysiere dieses Projekt“ funktionieren nicht in jedem Setup zuverlässig.
- Lokale Modelle nutzen Tools nicht immer selbstständig. Manche Modelle reagieren wie ein Chatbot und beschreiben den Schritt, statt ihn auszuführen.
- Tool-Calls können bei einigen Ollama-Modellen als JSON im Antworttext erscheinen, statt korrekt als strukturierter Tool-Aufruf an OpenCode übergeben zu werden.

In der Praxis hilft es, OpenCode explizit anzuweisen:

```
Lies zuerst die Projektstruktur (z. B. mit ls und find).

Suche danach gezielt die relevanten Dateien.

Lies diese Dateien wirklich ein.

Werte die echte Terminalausgabe aus.

Bewerte erst danach.

Führe Bash-Befehle selbst aus, warte nicht auf meine Ausgaben.
```

Diese Beobachtung ist eine Praxiserkenntnis aus eigenen Tests, nicht zwangsläufig universell. Mit zukünftigen Versionen von Ollama, OpenCode oder verbesserten Modelfiles kann sich das Bild ändern.

17.3 Der beste Arbeitsablauf

```
Schlecht: "Refactore meine komplette App."


Besser:

1. "Analysiere zuerst die Architektur."

2. "Schlage Refactoring-Schritte vor."

3. "Ändere nur die Search-Schicht."
```

Kleine, klare Aufgaben liefern bessere Ergebnisse als große, vage Anweisungen. Das gilt besonders bei lokalen Modellen.

18 Zugriff auf das Dateisystem

OpenCode arbeitet direkt im aktuellen Projektordner. Je nach Konfiguration kann es Dateien lesen, ändern, erzeugen, löschen und Git verwenden.

18.1 Wichtig bei macOS

OpenCode hat dieselben Rechte wie dein Benutzer. Wenn dein Benutzer Zugriff auf SSH-Keys, Zertifikate, Firmen-Repositories oder Keychain-Dateien hat, kann OpenCode theoretisch damit arbeiten.

Das bedeutet nicht automatisch Gefahr, aber man sollte es wissen und mit dem Permissions-System (Kapitel 16) steuern.

18.2 Empfehlung für Firmenprojekte

```
eigener macOS-Benutzer für KI-Arbeit

Sandbox-Projekte ohne Produktionszugriff

Test-Repositories

keine unnötigen Schlüssel im KI-Arbeitsordner
```

19 Zugriff auf Shell und Bash

OpenCode kann Befehle direkt im Terminal ausführen:

```
git · find · grep · swift · xcodebuild · npm · brew · rm · mv
```

Das macht es mächtig. Wenn du schreibst „Starte die Tests und behebe die Fehler", passiert intern zum Beispiel:

```
xcodebuild test -scheme MeinScheme ...

grep -r "FehlermeldungX" .

swift test
```

19.1 Was immer kontrolliert werden sollte

Diese Befehle immer prüfen, bevor sie bestätigt werden:

```
rm -rf

mv

chmod

git reset --hard

git clean -fd
```

Achtung: Niemals destruktive Kommandos blind bestätigen. Das Permissions-System in Kapitel 16 hilft, diese gezielt zu sperren.

20 Die OpenAI-API-Struktur erklärt

Dieses Kapitel richtet sich an Entwickler, die eigene Apps oder Skripte gegen Ollama bauen wollen.

20.1 Was ist die OpenAI-API-Struktur?

Die OpenAI-API-Struktur ist ein HTTP-basiertes Protokoll, das OpenAI für seine Modelle definiert hat und das inzwischen zum De-facto-Standard der Branche geworden ist. Viele Anbieter und lokale Tools implementieren dieselben Endpunkte.

20.2 Native Ollama-API vs. OpenAI-kompatible API

Ollama bietet beide Varianten parallel an:

```
Native Ollama-API:
  POST /api/generate
  POST /api/chat
  GET  /api/tags


OpenAI-kompatible API:
  POST /v1/chat/completions
  POST /v1/completions
  GET  /v1/models
```

Die OpenAI-kompatible Variante ist meist die richtige Wahl, wenn ein Tool oder eine Bibliothek bereits für OpenAI gebaut wurde. Die native Ollama-API bietet zusätzliche, Ollama-spezifische Optionen.

Bei Ollama lokal:

```
http://localhost:11434/v1/chat/completions
http://localhost:11434/v1/models
http://localhost:11434/api/tags
```

20.3 Das Request-Format (OpenAI-kompatibel)

Eine Anfrage ist ein JSON-Objekt mit `model` und `messages`:

```
{
  "model": "qwen2.5:32b",
  "messages": [
    { "role": "system", "content": "Du bist ein Swift-Experte." },
    { "role": "user",   "content": "Erkläre Swift Optionals." }
  ],
  "temperature": 0.7
}
```

Role	Bedeutung
`system`	Systemanweisung – Charakter und Regeln des Modells
`user`	deine Eingabe
`assistant`	Antwort des Modells (für Gesprächsverlauf)

20.4 Das Response-Format

```
{
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "Ein Optional in Swift ist..."
      }
    }
  ],
  "usage": {
    "prompt_tokens": 42,
    "completion_tokens": 120
  }
}
```

20.5 Warum hat sich dieses Format durchgesetzt?

OpenAI war früh am Markt, und viele Tools wurden gegen diese API entwickelt. Als Anthropic, Google, Mistral und andere kamen, wurde die OpenAI-Struktur die gemeinsame Basis. Es ist ein Industriestandard durch Marktdominanz, kein offener Standard durch Komitee.

20.6 Der praktische Vorteil

Jede Bibliothek, die für OpenAI geschrieben wurde, funktioniert oft mit Ollama – mit einer einzigen Änderung:

```
# OpenAI (Cloud)
client = OpenAI(base_url="https://api.openai.com/v1", api_key="sk-...")


# Ollama (lokal) – gleiche Bibliothek, andere URL
client = OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")
```

Das ist auch der Grund, warum Continue, Cline und andere Tools Ollama ohne eigenen SDK-Support nutzen können.

21 MCP Server – externe Tools einbinden

MCP (**Model Context Protocol**) ist ein Standard, der es KI-Agenten erlaubt, externe Tools zu nutzen. OpenCode unterstützt MCP-Server – damit lassen sich dem Agenten Werkzeuge geben, die über die eingebauten Tools hinausgehen.

21.1 Was sind MCP Server?

MCP-Server sind externe Prozesse, die OpenCode als Tools zur Verfügung stellen. Nach der Konfiguration nutzt das Modell sie automatisch neben den eingebauten Tools.

Praktische Beispiele:

- **Sentry** – Fehler und Issues direkt im Agent-Kontext
- **Context7** – aktuelle Dokumentation aus dem Web abrufen
- **GitHub** – Issues und PRs lesen und erstellen

21.2 Konfiguration in opencode.json

Lokaler MCP-Server (läuft auf deinem Mac):

```
{
  "mcp": {
    "mein-tool": {
      "type": "local",
      "command": ["npx", "-y", "mein-mcp-paket"]
    }
  }
}
```

Remote-Server:

```
{
  "mcp": {
    "mein-remote-tool": {
      "type": "remote",
      "url": "https://mcp-server.example.com"
    }
  }
}
```

21.3 Authentifizierung

Für Server mit OAuth-Authentifizierung:

```
opencode mcp auth mein-tool
```

OpenCode öffnet dann den OAuth-Flow automatisch.

21.4 Wichtiger Hinweis für Ollama-Nutzer

Wenn Tool-Aufrufe über MCP nicht zuverlässig funktionieren, kann es helfen, das Kontextfenster zu vergrößern:

```
ollama run qwen2.5:32b --num_ctx 8192
```

Lokale Modelle benötigen für komplexes Tool-Calling manchmal mehr Kontext.

22 Custom Commands – eigene Befehle erstellen

Du kannst eigene Slash-Befehle für wiederkehrende Aufgaben definieren. Das ist besonders nützlich für Swift-Projektroutinen wie Tests starten, Lint prüfen oder Dokumentation erzeugen.

22.1 Wo liegen Custom Commands?

```
MeinProjekt/
└── .opencode/
    └── commands/
        ├── test.md
        ├── lint.md
        └── docc.md
```

22.2 Format einer Command-Datei

```
---
description: Tests starten und Fehler erklären
---

Starte die Tests mit `swift test`. Zeige alle Fehler und erkläre,
was in jedem einzelnen Fall falsch ist. Ändere noch keine Dateien.
```

Nach dem Start ist `/test` als Befehl verfügbar.

22.3 Mit Argumenten

```
---
description: DocC für eine Klasse generieren
---

Erzeuge deutsche DocC-Kommentare für die Klasse $1. Code-Bezeichner
bleiben Englisch. Prüfe vorher den aktuellen Zustand mit @$1.
```

Aufruf: `/docc SearchService`

22.4 Shell-Output einbetten

```
---
description: SwiftLint prüfen und fixen
---

SwiftLint meldet folgende Fehler:
!swiftlint lint --reporter json
```

```
Behebe alle auto-fixbaren Probleme. Zeige mir die übrigen Fehler.
```

23 OpenCode vs. Claude Code – Vergleich

Claude Code ist Modell + Agent + Workflow + Tuning in einem geschlossenen System. OpenCode ist ein offenes Agent-Framework – die Qualität hängt stark vom gewählten Modell ab. Beides hat seine Berechtigung; der folgende Vergleich ist sachlich gemeint.

Bereich	**Claude Code**	**OpenCode**
Einrichtung	sehr bequem	technischer
Modellqualität	sehr hoch	abhängig vom gewählten Modell
Lokale Modelle	nicht der Hauptweg	sehr gut geeignet
Datenschutz	meist Cloud	lokal möglich
Provider-Freiheit	begrenzt	sehr hoch (75+ Provider)
Kosten	Abo/API	lokal kostenlos (außer Hardware)
Agent-Stabilität	meist stabiler	schwankt je nach Modell
Tool-Calling	sehr ausgereift	modellabhängig
Permissions	integriert	konfigurierbar (Kapitel 16)
MCP-Support	ja	ja

23.1 Einschätzung

```
Claude Code              → komfortabel und automatisch bei großen, mehrstufigen
Aufgaben

OpenCode + Ollama        → stark bei lokaler Kontrolle, Kosten und Datenschutz

OpenCode + Cloud-API     → bringt Provider-Freiheit zusammen mit starken Modellen
```

Tipp: OpenCode und Claude Code schließen sich nicht aus. Viele Entwickler nutzen OpenCode mit Ollama für Alltagsaufgaben und greifen für große Refactorings zu Claude Code.

24 Vergleich zu Codex, Gemini CLI und Grok

Ein kurzer Überblick über weitere bekannte Terminal-Agenten und Cloud-Tools.

24.1 OpenAI Codex

Cloudbasierter Software-Engineering-Agent. Kann Features schreiben, Bugs fixen und Pull Requests vorschlagen. Aufgaben laufen in isolierten Cloud-Sandboxes mit Repository-Kontext.

24.2 Gemini CLI

Open-Source-Agent im Terminal. Macht Gemini direkt im Terminal nutzbar und arbeitet über einen Reason-and-Act-Loop mit Tools und MCP-Servern.

24.3 Grok

xAI Grok 4 ist ein Modell für agentisches Reasoning, Wissensarbeit und Tool-Nutzung. Primär über API und Web-Interface verfügbar.

24.4 Gesamtübersicht

Werkzeug	**Typ**	**Stärke**
Claude Code	Premium Coding-Agent	sehr stark bei großen Projekten
OpenAI Codex	Cloud Coding-Agent	Aufgaben, PRs, Bugfixes
Gemini CLI	Terminal-Agent	stark mit Google-Ökosystem und MCP
OpenCode	offener Terminal-Agent	Provider-Freiheit, Ollama, lokale Modelle
Cline/Roo Code	IDE-Agent	gute VS-Code-Integration
Continue	IDE-Erweiterung	Chat, Completion, lokale Modelle

25 Leistungsübersicht bekannter Modelle

Die folgenden Werte sind grobe Praxis-Einordnungen aus eigenen Erfahrungen, **keine offiziellen Benchmarks***. Sie können je nach Hardware, Aufgabe, Prompt und Modellversion deutlich abweichen.*

Modell	**Typ**	**Praxiseinschätzung Coding**
Claude Opus + Claude Code	Premium Agent	95–100%
GPT / Codex	Premium Agent	93–98%
Gemini Pro / Gemini CLI	Premium Agent	90–96%
Grok 4.x	Premium Modell/API	88–94%
Claude Sonnet	Premium Alltag	86–93%
DeepSeek R1 32B/70B	Open Reasoning	80–90%
Qwen 2.5 32B	Open Allround	80–88%
Qwen2.5-Coder 32B	Open Coding	80–88%
Qwen2.5-Coder 14B	Open Coding	74–82%
DeepSeek-Coder	Open Coding	72–82%
Qwen2.5-Coder 7B	Open Coding	68–76%
Gemma / Phi	kleine Modelle	62–74%
Code Llama	älter, solide	58–70%

Hinweis: Ein 80%-Modell ist nicht schlecht. Bei einfachen und mittleren Aufgaben reicht es oft gut. Bei großen Refactorings und langen Agent-Ketten ist der Abstand zu Cloud-Modellen spürbar.

26 Welche Modelle sollte ein Apple-Entwickler nehmen?

Die Antwort hängt von Hardware, Aufgabe und Arbeitsweise ab. Die folgenden Empfehlungen sind Erfahrungswerte – im eigenen Setup unbedingt selbst testen.

26.1 Lokale Startkombination

Für viele Entwickler mit 32 GB RAM – ein Modell für Code, eines für Analyse:

```
ollama pull qwen2.5-coder:14b
```

Wer mit OpenCode echtes Tool-Calling nutzen möchte, sollte zusätzlich oder alternativ ein Modell mit zuverlässigem Tool-Calling laden:

```
ollama pull qwen2.5:32b
```

26.2 Für stärkere Macs (48 GB+ RAM)

Wer einen Mac Studio oder MacBook Pro mit 48 GB+ RAM hat, kann die 32B-Varianten nutzen – stärker, Ergebnisse näher an Cloud-Modellen:

```
ollama pull qwen2.5:32b
```

```
ollama pull qwen2.5-coder:32b
```

26.3 Für kleinere Macs (16 GB RAM)

Das 7B-Modell – schnell, klein, für viele alltägliche Aufgaben oft ausreichend:

```
ollama pull qwen2.5-coder:7b
```

26.4 Entscheidungshilfe

Situation	**Mögliche Wahl**
Tägliche Coding-Aufgaben (lokal)	Qwen2.5-Coder 14B oder 32B
Coding mit Agent-Aktionen / Tool-Calls	Qwen 2.5 32B (eigene Tests)
Architektur und Analyse	DeepSeek R1 14B / 32B
Schnelle Fragen, wenig RAM	Qwen2.5-Coder 7B
Große Refactorings	Claude Code oder Codex
Sensible Firmenprojekte	Ollama lokal

27 Stärken und Schwächen lokaler Modelle

Lokale Modelle haben klare Vor- und Nachteile gegenüber Cloud-Diensten. Beide kennen hilft bei der richtigen Aufgabenauswahl.

27.1 Stärken

```
lokal und privat
```

```
keine laufenden API-Kosten
```

```
keine Ratenlimits
```

```
keine Serverausfälle durch Anbieter
```

```
gut für Alltagscode

gut für Erklärungen und Tests

gut für Offline-Arbeit
```

27.2 Schwächen

```
schwächer bei sehr großen Codebases

langsamer bei großen Modellen

mehr Konfiguration nötig

weniger zuverlässig bei langen Agent-Ketten

Tool-Calling stark vom Modell abhängig

kleine Modelle halluzinieren schneller

begrenztes Kontextfenster

mehr eigene Hardware nötig
```

27.3 Wichtigster Punkt

Lokale Modelle sind nicht automatisch schlechter. Sie profitieren aber von:

- besserer Aufgabenstellung
- kleineren Arbeitspaketen
- klaren Regeldateien (`AGENTS.md`)
- konfigurierten Permissions zur Kontrolle

28 Hybrid-Workflow: lokal und Cloud kombiniert

Viele Apple-Entwickler nutzen heute eine Kombination – lokale Modelle für den Alltag, Cloud für schwere Aufgaben:

Aufgabe	**Mögliches Tool**
Lokales Daily Coding	OpenCode + Qwen
Architekturanalyse	DeepSeek R1
Große Refactorings	Claude Code
Schnelle Code-Completion	lokale Modelle
Sensible Projekte	Ollama lokal
Massive Agent-Tasks	Claude / GPT / Codex
Offline / Flugzeug	Ollama lokal
Test-Erzeugung	Qwen + OpenCode
Dokumentation	lokale Modelle

28.1 Gedanke dahinter

```
Lokale KI = Standardwerkzeug für viele Alltagsaufgaben
```

```
Cloud-KI  = Reserve für schwere und komplexe Aufgaben
```

Das spart Kosten und gibt Kontrolle, ohne bei schwierigen Aufgaben auf Qualität zu verzichten.

29 Fehlerbehebung

Dieser Abschnitt hilft bei den häufigsten Problemen beim Einstieg.

29.1 Ollama antwortet nicht

Prüfen, ob der Server läuft:

```
curl http://localhost:11434

# Erwartete Antwort: Ollama is running
```

Falls nicht:

```
ollama serve
```

(Nur ohne Desktop-App nötig.)

29.2 OpenCode startet nicht

Log-Dateien finden:

```
macOS/Linux: ~/.local/share/opencode/log/
```

OpenCode neu starten und auf Fehlermeldungen achten:

```
opencode --log-level debug
```

29.3 Tool-Aufrufe funktionieren nicht (besonders mit Ollama)

Lokale Modelle brauchen manchmal ein größeres Kontextfenster für Tool-Calling. In der Ollama-Konfiguration oder beim Start:

```
ollama run qwen2.5:32b --num_ctx 8192
```

Wenn das nicht hilft, das Modell wechseln (siehe Kapitel 8 und 14).

29.4 Modell läuft sehr langsam

Das Modell passt möglicherweise nicht in den RAM – Ollama lagert dann Teile auf die SSD aus (Swap). Lösung: kleineres Modell verwenden.

```
ollama list  # Speicherbedarf prüfen
```

Als Faustregel: Das Modell braucht ca. 1 GB pro Milliarde Parameter.

29.5 OpenCode hat unerwünschte Änderungen gemacht

Änderungen rückgängig machen:

```
/undo
```

Oder über Git komplett zurücksetzen:

```
git checkout .
```

29.6 Wo bekomme ich Hilfe?

```
GitHub Issues:  github.com/sst/opencode

Discord:        Einladungslink auf opencode.ai

Ollama Discord: discord.gg/ollama
```

Logs aus `~/.local/share/opencode/log/` bei Bug-Reports mitschicken.

30 Fazit

Ollama ist für Entwickler inzwischen mehr als ein Spielzeug. In Verbindung mit OpenCode bekommst du einen lokalen Coding-Agenten, der für viele Alltagsaufgaben reicht. Qwen2.5-Coder ist eine starke Wahl für reine Code-Aufgaben. Für den Agent-Modus mit Tool-Calling hat sich in eigenen Tests Qwen 2.5 32B als zuverlässiger erwiesen. DeepSeek R1 ist eine sinnvolle Ergänzung für Analyse und technische Entscheidungen.

Claude Code, Codex und Gemini CLI bleiben bei großen, schwierigen Aufgaben oft komfortabler. Für lokale Arbeit, Datenschutz und wiederkehrende Entwickleraufgaben ist OpenCode mit Ollama eine sehr gute Ergänzung.

30.1 Praktische Empfehlung

```
Qwen 2.5 32B          → Standardmodell für Agent-Modus mit Tool-Calling

Qwen2.5-Coder 14B/32B → Code-Modell für Erklärungen und Vorschläge

DeepSeek R1 14B       → Analyse und Architektur

OpenCode              → lokaler Agent

AGENTS.md             → Projektregeln definieren

Permissions           → Sicherheit konfigurieren

Claude Code           → für besonders schwere Aufgaben
```

30.2 Realität 2026

Lokale KI ist ernstzunehmend. Die Agent-Systeme sind noch jung – manchmal beeindruckend, manchmal sperrig. Die besten Ergebnisse entstehen mit:

```
kleinen, klaren Aufgaben

sauberem Git-Zustand

guter AGENTS.md

konfigurierten Permissions

kontrollierten Änderungen
```

31 Weiterführende Ressourcen

31.1 Offizielle Dokumentation (Quellen dieser Anleitung)

Die nachfolgenden Ressourcen dienten als Hauptquellen für diese Anleitung. Inhalte wurden daraus sinngemäß übernommen, eigenständig strukturiert und ins Deutsche übertragen.

Ressource	URL	Lizenz	Inhalt
OpenCode Docs (DE)	https://opencode.ai/docs/de	MIT	vollständige Dokumentation auf Deutsch
OpenCode GitHub	https://github.com/sst/opencode	MIT	Quellcode, Issues, Changelog
Ollama Docs	https://docs.ollama.com	MIT	Ollama-API, Befehle, Konfiguration
Ollama GitHub	https://github.com/ollama/ollama	MIT	Quellcode, Modell-Formate
Ollama Bibliothek	https://ollama.com/library	—	alle verfügbaren Modelle mit Beschreibung

31.2 Wichtige Unterseiten der OpenCode-Dokumentation

Seite	Wann relevant
opencode.ai/docs/de/rules/	AGENTS.md und Regelwerk
opencode.ai/docs/de/permissions/	Permissions-Konfiguration
opencode.ai/docs/de/config/	opencode.json vollständig
opencode.ai/docs/de/mcp-servers/	externe Tools einbinden
opencode.ai/docs/de/providers/	alle Provider konfigurieren
opencode.ai/docs/de/commands/	Custom Commands erstellen
opencode.ai/docs/de/troubleshooting/	Fehlerbehebung
opencode.ai/docs/de/keybinds/	Tastenkürzel anpassen

31.3 Community

```
GitHub:   github.com/sst/opencode

Discord:  Link auf opencode.ai
```

32 Glossar

Agent: Ein KI-System, das nicht nur antwortet, sondern Dateien lesen, ändern und Befehle ausführen kann.

AGENTS.md: Instruktionsdatei für OpenCode. Enthält Projektregeln, Architekturvorgaben und Verhaltensregeln für den Agent. Entspricht `CLAUDE.md` im Claude-Code-Ökosystem.

Apple Silicon: ARM-basierte Chip-Familie von Apple (M1–M4). Unified Memory macht lokale KI-Modelle in der Praxis effizienter als auf klassischer PC-Hardware.

auth.json: Datei in `~/.local/share/opencode/`, die API-Keys und Authentifizierungstoken für Provider speichert.

Context Window: Die maximale Textmenge, die ein Modell auf einmal verarbeiten kann. Bei lokalen Modellen oft kleiner als bei Cloud-Modellen.

CLAUDE.md: Instruktionsdatei aus dem Claude-Code-Ökosystem. Wird auch von OpenCode gelesen.

Custom Commands: Eigene Slash-Befehle für wiederkehrende Aufgaben, definiert als Markdown-Dateien in `.opencode/commands/`.

DeepSeek R1: Open-Reasoning-Modell, in eigenen Tests stark für Analyse, Architektur und technische Entscheidungen.

Halluzination: Wenn ein KI-Modell plausibel klingende, aber falsche Informationen erzeugt – besonders bei kleinen Modellen ein Risiko.

Homebrew: Paketverwaltung für macOS. Mit `brew install ollama` wird Ollama installiert.

Leader Key: Taste in OpenCode (Standard: `ctrl+x`), nach der weitere Tastenkürzel folgen. Vermeidet Konflikte mit Terminal-Shortcuts.

LLM: Large Language Model. Große Sprachmodelle wie GPT-4, Claude Opus oder Qwen 2.5.

MCP: Model Context Protocol. Standard für externe Tools in KI-Agenten.

Modellgröße: Gemessen in Milliarden Parametern (7B, 14B, 32B). Mehr Parameter = meist stärker, aber mehr RAM nötig.

num_ctx: Kontextfenster-Größe in Ollama. Erhöhen kann bei komplexem Tool-Calling helfen.

Ollama: Lokaler Modell-Runner. Startet und verwaltet LLMs auf der eigenen Hardware.

OpenCode: Offener Terminal-Agent für Coding-Aufgaben. Unterstützt 75+ Provider.

OpenAI-API-Struktur: HTTP-Protokoll, das OpenAI definiert hat und das zum De-facto-Standard der KI-Branche geworden ist. Ollama implementiert die Endpunkte unter `/v1/...`.

Permissions: Konfiguration in `opencode.json`, die steuert, welche Aktionen erlaubt, nachgefragt oder gesperrt sind.

Plan Mode: Modus in OpenCode (Tab-Taste), in dem der Agent nur plant und keine Dateien ändert.

Provider: Anbieter von KI-Modellen (Anthropic, OpenAI, Google, Ollama, …).

Qwen 2.5 / Qwen2.5-Coder: Modelle aus der Qwen-Familie von Alibaba Cloud. Qwen 2.5 ist ein Allround-Modell, Qwen2.5-Coder ist auf Code spezialisiert.

Quantisierung: Technik zur Verkleinerung von Modellen. Ermöglicht den Betrieb großer Modelle auf weniger RAM.

Reasoning-Modell: Modell, das vor der Antwort intern Denkschritte durchführt. DeepSeek R1 ist ein Beispiel.

Session-Kontext: Gesprächsverlauf der aktuellen Sitzung. Nach `/clear` gelöscht.

Tool-Calling: Mechanismus, mit dem ein Modell strukturierte Tool-Aufrufe an den Agenten übergibt (`tool_calls`-Feld in der Antwort) – Voraussetzung dafür, dass OpenCode Befehle wirklich ausführt.

TUI: Terminal User Interface. Interaktive Oberfläche direkt im Terminal – kein grafisches Fenster.

Unified Memory: Apple-Architektur, bei der CPU, GPU und Neural Engine denselben RAM teilen. Kein separater GPU-VRAM nötig – ideal für lokale KI.

xcodebuild: Apple-CLI zum Bauen und Testen von Xcode-Projekten aus dem Terminal.

Stand: Mai 2026 — Quellen: opencode.ai/docs/de (MIT) · ollama.com (MIT) · github.com/sst/opencode · github.com/ollama/ollama