OpenCode + Ollama + Gemma4 – Anleitung

Lokales KI-Setup für macOS: OpenCode (Open-Source-CLI-Agent) mit dem lokalen Modell Gemma4 betreiben. Datenschutzkonform, offline-fähig, ohne laufende API-Kosten — und ohne Proxy zwischen Tool und Modell.

Autor: Christian Drapatz

Stand: Mai 2026 · Claude Code + Ollama + Gemma4

Plattform: macOS · Apple Silicon

Version 1.0

Inhaltsverzeichnis


```
1. Konzept und Komponenten
2. Voraussetzungen
3. Installation
4. Konfiguration
5. Betrieb
6. Slash-Befehle in OpenCode
7. Praxisverhalten und Stolpersteine
8. Troubleshooting
9. Referenz
A. Anhang: Skripte zum Selbst-Anlegen
```

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.

Quickstart (5 Befehle)

Für erfahrene Leser. Ausführliche Erklärung folgt in den Kapiteln 1–5.

```
chmod +x *.sh

./install-opencode.sh     # einmalig: alles installieren
./opencode-up.sh          # Ollama starten
./opencode-start.sh       # OpenCode mit Vorab-Checks starten
./opencode-status.sh      # Status prüfen
./opencode-down.sh        # Ollama beenden
```

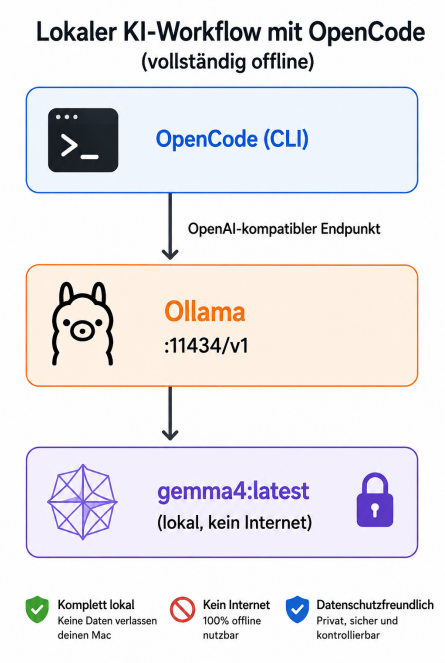
---

1. Konzept und Komponenten

Worum geht's in diesem Kapitel? Bevor wir installieren, ein

Überblick: Drei Komponenten arbeiten zusammen — eine weniger als beim Claude-Code-Setup, weil OpenCode direkt mit Ollama spricht (kein Proxy).

Du erfährst, was jede einzelne ist, wer dahintersteht und warum sie im Stack gebraucht wird. Keine Vorkenntnisse nötig.



1.1 OpenCode

Was? Ein Open-Source-CLI-Agent für Code- und Projektarbeit — vergleichbar mit Claude Code, aber quelloffen und provider-agnostisch. OpenCode ist **kein** Sprachmodell, sondern das Orchestrierungs-Tool: liest Dateien, ruft Shell-Befehle auf, führt Edits aus. Das Modell (Gemma4, qwen2.5, optional auch Anthropic/OpenAI in der Cloud) liefert die Intelligenz darunter.

Von wem? SST / Anomaly Innovations — Open Source (Repository `sst/opencode` auf GitHub).

Homepage: https://opencode.ai

Installation: Eigener Installer (`curl -fsSL https://opencode.ai/install | bash`) oder via npm (`@opencode-ai/opencode`).

Was ist anders gegenüber Claude Code?

- **Open Source** — auditierbar, kein Vendor-Lock-in
- **Provider-agnostisch** — Ollama, Anthropic, OpenAI, Gemini u. a. über eine einzige Config-Datei
- **Direkter Ollama-Support** über den OpenAI-kompatiblen Endpunkt `/v1` — kein Proxy nötig
- **Feines Permission-System** pro Tool (`bash`, `read`, `edit`, `write`, `webfetch` …)
- **Slash-Befehle** (`/init`, `/model`, `/clear`, …) — siehe Kapitel 6

Vorkenntnisse: Grundkenntnisse Terminal/Shell.

1.2 Ollama

Was? Eine lokale Laufzeitumgebung für KI-Sprachmodelle — das "Docker für LLMs". Lauscht auf `http://localhost:11434` und stellt zwei APIs bereit: das eigene Ollama-Format **und** einen OpenAI-kompatiblen Endpunkt unter `/v1`. Genau diesen `/v1`-Endpunkt verwendet OpenCode.

Von wem? Ollama Inc. — Open Source (Repository `ollama/ollama` auf GitHub).

Homepage: https://ollama.com

Warum überhaupt nutzen? Ohne Ollama müsstest du Sprachmodelle in PyTorch oder MLX selbst laden und verwalten. Ollama ist der einfachste Weg, ein lokales Modell wie Gemma4, Llama oder Qwen zu betreiben.

Apple-Silicon-Vorteil: Nutzt Metal-GPU automatisch — kein zusätzliches Setup nötig.

1.3 Gemma4

Was? Ein Open-Source-Sprachmodell. Die "4" bezeichnet die Modellgeneration (Nachfolger von Gemma 1–3). Verfügbar in mehreren

Größen, die sich in Parameter-Anzahl und Speicherbedarf unterscheiden.

Von wem? Google DeepMind.

Homepage / Modell-Katalog: https://ollama.com/library/gemma

Lizenz: Gemma Terms of Use — erlaubt kommerziellen Einsatz.

Warum dieses Modell? Guter Kompromiss zwischen Qualität, Geschwindigkeit und Speicherbedarf für 16-GB-Macs. Verfügbare Größen:

```
gemma4:2b       ~2 GB     sehr schnell, begrenzte Qualität
gemma4:latest   ~6–8 GB   guter Kompromiss (empfohlen ab 16 GB RAM)
gemma4:27b      ~20+ GB   hohe Qualität, langsamer
```

Der Tag `latest` zeigt auf die Standardgröße — keine feste Version.

Wichtig für OpenCode: Der Erfolg hängt stark vom **Tool-Calling-Verhalten des Modells in Ollama** ab. Wenn Gemma4 in der Praxis Aufrufe nur als JSON-Text statt als API-Aufrufe ausgibt, ist das Modelfile-Template in Ollama dafür nicht korrekt verdrahtet. Siehe Kapitel 7 für Diagnose und Fallback (`qwen2.5:32b`).

1.4 Kurzvergleich zu Claude Code

Aspekt	Claude Code	OpenCode
Lizenz	proprietär (Anthropic)	Open Source
Lokale Anbindung	über LiteLLM-Proxy	direkt zu Ollama
Komponenten lokal	4	3
Permission-Modell	grob	fein pro Tool
Tool-Calling lokal	robust (über Proxy)	modellabhängig
Reife	hoch	jünger
Lokale Modellführung	selbstständig	präzise Prompts

---

2. Voraussetzungen

Worum geht's in diesem Kapitel? Wir prüfen, ob dein Mac für das Setup geeignet ist: Hardware, macOS-Version, freier Speicher und die nötigen Tools (Xcode CLT, Homebrew, curl). Mit dem Schnellcheck unten in 2–5 Minuten erledigt. Node.js ist **nicht** zwingend nötig — der OpenCode-Installer bringt eine eigene Laufzeit mit.

Vorkenntnisse: Du kannst Befehle im Terminal eingeben. Wenn etwas fehlt, steht der passende Installationsbefehl daneben.

2.1 Hardware und macOS

Anforderung	Minimum	Empfohlen
Mac	Apple Silicon (M1+) oder Intel	Apple Silicon
RAM	16 GB	32 GB

```
Freier Speicher    10 GB                          20 GB+
macOS              12 (Monterey)                  aktuell
```

Schnellcheck:

```
sw_vers
sysctl -n hw.memsize | awk '{printf "%.0f GB RAM\n", $1/1024/1024/1024}'
uname -m                          # arm64 = Apple Silicon
df -h ~ | tail -1                 # freier Speicher im Home
```

2.2 Tools

```
Tool                       Prüfen              Installieren
-------------------------------------------------------------------
Xcode Command Line Tools   xcode-select -p     xcode-select --install
Homebrew                   brew --version      (siehe unten)
curl                       curl --version      liegt macOS bei
```

Homebrew installieren (falls fehlend):

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

Node.js wird **nicht zwingend** benötigt — der OpenCode-Installer (`curl -fsSL https://opencode.ai/install | bash`) bringt eine passende Laufzeit mit. Wer es lieber über npm installiert, braucht trotzdem Node:

```
brew install node      # nur falls npm-Installation gewünscht
```

---

3. Installation

Worum geht's in diesem Kapitel? Hier wird der vollständige Stack installiert: Ollama, OpenCode und das Modell Gemma4. Zwei Wege stehen zur Auswahl — ein Skript für alles auf einmal oder Schritt für Schritt.

Was passiert? Etwa 6–8 GB werden heruntergeladen (das Modell). Plane Bandbreite und Zeit ein: je nach Internetanbindung 10–30 Minuten.

Vorkenntnisse: Kapitel 2 abgeschlossen (Homebrew vorhanden, genügend RAM und Speicher).

3.1 Schnellinstallation

Das Skript `install-opencode.sh` führt alle Schritte aus, ist idempotent (mehrfach ausführbar) und lädt `gemma4:latest` herunter.

```
chmod +x install-opencode.sh
./install-opencode.sh
```

3.2 Manuelle Installation

Falls du Schritt für Schritt vorgehen willst:

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

```
# Ollama
brew install ollama
ollama --version

# OpenCode (offizieller Installer)
curl -fsSL https://opencode.ai/install | bash

# alternativ über npm
# npm install -g opencode-ai

# PATH neu laden, damit `opencode` gefunden wird
source ~/.zshrc

# Ollama starten und Gemma4 laden (ca. 6–8 GB)
brew services start ollama
ollama pull gemma4:latest

# Kurzer Funktionstest, mit /bye beenden
ollama run gemma4:latest "Hallo"
```

---

4. Konfiguration

Worum geht's in diesem Kapitel? Du legst die zentrale OpenCode-Konfiguration `~/.config/opencode/opencode.json` an. Sie sagt OpenCode, welcher Provider (Ollama), welcher Endpunkt (`/v1`) und welches Modell (`gemma4:latest`) verwendet wird. Außerdem definierst du das Permission-Modell pro Tool — z. B. `webfetch: "deny"` für strikten Offline-Betrieb.

Vorkenntnisse: Du kannst eine Textdatei mit `nano` öffnen und JSON-Syntax grob lesen. Mehr nicht.

4.1 opencode.json

OpenCode liest seine Konfiguration aus `~/.config/opencode/opencode.json`. Diese Datei legt fest, welcher Provider, welcher Endpunkt und welches Modell verwendet werden.

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

Inhalt:

```
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
```

```
      },
      "models": {
        "gemma4:latest": {
          "name": "Gemma 4"
        }
      }
    }
  },
  "model": "ollama/gemma4:latest",
  "permission": {
    "bash": "allow",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
    "edit": "ask",
    "write": "ask",
    "task": "deny",
    "webfetch": "deny"
  }
}
```

4.2 Permission-System

Jedes Tool, das OpenCode aufrufen kann, lässt sich separat konfigurieren:

```
"allow"   Tool darf ohne Rückfrage genutzt werden
"ask"     OpenCode fragt vor jeder Nutzung
"deny"    Tool ist gesperrt
```

Sinnvolle Defaults für lokale Code-Arbeit:

```
bash, read, glob, grep   → "allow"   (Standard-Recherche)
edit, write              → "ask"     (Dateiänderungen kontrolliert)
task, webfetch           → "deny"    (kein Subagent, kein Internet)
```

4.3 Modellwahl

Mit OpenCode hängt das Ergebnis stark vom Tool-Calling-Verhalten des Modells ab. Empfehlungen:

```
Modell             RAM       Tool-Calling   Code-Qualität
-------------------------------------------------------------------
gemma4:latest      ~6–8 GB   variabel       gut
qwen2.5:32b        ~20 GB    zuverlässig    sehr gut für Code
llama3.2:latest    ~2 GB     zuverlässig    begrenzt (3B)
```

Wenn Gemma4 in der Praxis Tool-Calls nur als JSON-Text ausgibt statt sie auszuführen (siehe Kapitel 7), liegt das nicht an OpenCode, sondern am Modelfile-Template in Ollama. In dem Fall auf `qwen2.5:32b` ausweichen:

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

Dann in `opencode.json` den Block `models` und `model` entsprechend anpassen.

4.4 Kontextfenster

Ollama wählt das Kontextfenster anhand des vorhandenen VRAM automatisch (typisch 32 768 Token bei Apple Silicon mit ausreichend RAM). Manuell setzen lässt es sich über die Umgebungsvariable beim Start:

```
OLLAMA_CONTEXT_LENGTH=16384 ollama serve
```

Im normalen `brew services`-Betrieb genügt der Default.

---

5. Betrieb

Worum geht's in diesem Kapitel? Den fertigen Stack starten, stoppen und überwachen. Erst ab hier nutzt du dein lokales KI-System aktiv.

Was hast du jetzt?

Nach Kapitel 2–4 sieht dein System so aus:

- **OpenCode** ist installiert und über den Befehl `opencode` aufrufbar.
- **Ollama** läuft im Hintergrund als launchd-Dienst (überlebt den Reboot) und stellt unter Port 11434 sowohl die native Ollama-API als auch den OpenAI-kompatiblen `/v1`-Endpunkt bereit.
- **Gemma4** liegt als ca. 6–8 GB großes Modell im Ollama-Cache und kann geladen werden.
- **`~/.config/opencode/opencode.json`** sagt OpenCode, welcher Provider (Ollama), welcher Endpunkt und welches Modell verwendet werden — und welche Tools erlaubt, eingeschränkt oder gesperrt sind.

Was kannst du damit machen?

- Code lesen, schreiben, refactoren, dokumentieren — **vollständig offline**, ohne dass auch nur ein Token den Mac verlässt.
- Per Slash-Befehl `/init` einen Projekt-Kontext (`AGENTS.md`) anlegen, der künftige Sessions kürzer und treffsicherer macht.
- Per Slash-Befehl `/model` zwischen mehreren lokalen Modellen wechseln, ohne OpenCode neu zu starten.
- Sensible Projekte (NDA, regulierte Branchen, firmeninternes IP) bleiben auf deinem Mac.
- Strikte Offline-Garantie über `webfetch: "deny"` in der Config.
- Keine API-Kosten, keine Rate Limits.

Was solltest du nicht erwarten?

Lokale Modelle verhalten sich anders als die Cloud. Vor allem das **Tool-Calling** kann fragil sein — wenn Gemma4 Aufrufe nur als JSON-Text statt als API-Aufrufe ausgibt, geht keine Aktion durch.

Kapitel 7 beschreibt Diagnose und Fallback (`qwen2.5:32b`). Außerdem müssen Prompts deutlich präziser formuliert werden als bei Claude Code mit Cloud — auch dazu mehr in Kapitel 7.

5.1 Stack starten und stoppen

Mit den mitgelieferten Skripten:

```
./opencode-up.sh          # Ollama (brew service) starten
./opencode-down.sh        # Ollama beenden
```

Manuell:

```
brew services start ollama
brew services stop ollama
```

5.2 OpenCode starten

Empfohlen über Wrapper-Skript (prüft vorab, ob Ollama läuft und `opencode.json` existiert):

```
./opencode-start.sh
```

Manuell:

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

OpenCode wird im aktuellen Verzeichnis gestartet — das wird zum Projekt-Arbeitsverzeichnis.

5.3 Status prüfen

```
./opencode-status.sh
```

Zeigt: Ollama läuft?, welche Modelle sind geladen?, ist `opencode.json` vorhanden?, ist `opencode` im PATH?

---

6. Slash-Befehle in OpenCode

Worum geht's in diesem Kapitel? OpenCode kennt — anders als Claude Code — eingebaute Slash-Befehle, die du **innerhalb** einer laufenden Session eingibst. Sie steuern Modellwahl, Kontext, Verlauf und Projekt-Setup.

Vorkenntnisse: OpenCode läuft (Kapitel 5). Den ersten Befehl gibst du nach dem Start mit `./opencode-start.sh` direkt im Chat ein.

Innerhalb der OpenCode-Session:

```
/init         Projekt analysieren und AGENTS.md erzeugen
/help         Alle Befehle anzeigen
/model        Modell wechseln
/models       Liste aller konfigurierten Modelle
/clear        Gesprächsverlauf zurücksetzen
/undo         Letzte Dateiänderungen rückgängig machen
```

```
/redo        Rückgängig gemachte Änderungen wiederherstellen
/share       Link zur aktuellen Unterhaltung erzeugen
/connect     Neuen Provider verbinden
/summarize   Kontext zusammenfassen
```

`/init` ist sinnvoll als erster Befehl in einem neuen Projekt — OpenCode legt eine `AGENTS.md` an, die als Projekt-Kontext für künftige Sessions dient.

---

7. Praxisverhalten und Stolpersteine

Worum geht's in diesem Kapitel? Ehrliche Einordnung: Was funktioniert lokal gut, was nicht? Vor allem die größte Stolperfalle — fragiles Tool-Calling — wird hier diagnostiziert. Mit konkretem `curl`-Diagnose-Befehl und dokumentiertem Fallback (`qwen2.5:32b`).
Vorkenntnisse: Keine. Lesenswert auch vor der Installation, wenn du abschätzen willst, ob OpenCode + lokales Modell für deinen Anwendungsfall taugt.

7.1 Was zu erwarten ist

OpenCode mit lokalen Modellen verhält sich **nicht** wie Claude Code mit Anthropic Cloud. Die zwei wichtigsten Unterschiede:

- **Lokale Modelle müssen geführt werden.** Claude Code entscheidet selbständig, welche Dateien gelesen werden müssen. Lokale Modelle verlangen oft explizite Schritt-für-Schritt-Anweisungen.
- **Tool-Calling kann fragil sein.** Manche Modelle geben Tool-Aufrufe als JSON-Text aus statt als API-Aufruf. OpenCode kann diesen Text dann nicht ausführen.

7.2 Symptom: JSON statt Ausführung

Wenn OpenCode auf eine simple Anweisung nur antwortet mit:

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

dann ist das Tool-Calling des aktuellen Modells in Ollama nicht korrekt verdrahtet. Der Aufruf landet als reiner Text-Output statt als API-Tool-Call.
Sofortmaßnahme:
1. `curl` testen, ob Ollama selbst korrekte Tool-Calls liefert
2. Modell wechseln auf eines mit verifiziertem Tool-Calling-Template

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

Diagnose-Befehl:

```
curl http://localhost:11434/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gemma4:latest",
    "messages": [{"role":"user","content":"Liste das Verzeichnis"}],
    "tools": [{
      "type": "function",
      "function": {
        "name": "bash",
        "description": "Run a shell command",
        "parameters": {
          "type": "object",
          "properties": {"command":{"type":"string"}}
        }
      }
    }]
  }'
```

Wenn die Antwort einen `tool_calls`-Array enthält, ist das Modell geeignet. Wenn nicht (nur `content`-Text mit JSON darin), nicht.

7.3 Prompts für lokale Modelle

Mit OpenCode + lokalem Modell sollten Prompts deutlich präziser sein als mit Claude Code. Bewährtes Schema:

```
Antworte auf Deutsch.

Ändere keine Dateien.

Suche im aktuellen Projekt nach Datei X.
Nutze dafür lokale Bash-Befehle.

Führe zuerst aus:
    find . -name "*X*"

Lies danach die gefundene Datei.
Analysiere nur den tatsächlich gelesenen Inhalt.
Erfinde keine Datei und keine Fehler.
```

7.4 Empfehlungen pro Aufgabe

Aufgabe	OpenCode + Gemma4	OpenCode + qwen2.5:32b
Einzelne Funktion erklären	gut	sehr gut
Code dokumentieren	gut	sehr gut
Tool-Calling (Dateien lesen etc.)	variabel	zuverlässig
Kleines Refactoring (1 Datei)	begrenzt	gut
Refactoring über viele Dateien	schwach	begrenzt
Architekturplanung	schwach	begrenzt

Selbstkorrektur bei Fehlern	schwach	begrenzt
Offline-Betrieb	ja	ja
DSGVO-konform	ja	ja
Kostenlos	ja	ja

7.5 Vorteile gegenüber Claude Code

- **Open Source** — auditierbar, kein Vendor-Lock-in
- **Direkt zu Ollama** — keine Proxy-Komponente, weniger Konfiguration
- **Feines Permission-Modell** — z. B. `webfetch: "deny"` für strikte Offline-Garantie
- **Provider-agnostisch** — derselbe Stack funktioniert auch mit Cloud-Providern, falls nötig

---

8. Troubleshooting

Worum geht's in diesem Kapitel? Häufige Fehlermeldungen und ihre Lösung — tabellarisch. Hier erst hineinschauen, wenn beim Betrieb etwas klemmt. Am Ende des Kapitels: Wo finde ich Logs und Service-Status?

Vorkenntnisse: Kapitel 5 (Betrieb) durchgegangen.

Symptom	Ursache / Lösung
`opencode: command not found`	Installer-PATH fehlt → `source ~/.zshrc` oder Terminal neu öffnen
`ollama: command not found`	brew-PATH nicht aktiv (Apple Silicon) → in `~/.zprofile` ergänzen: `eval "$(/opt/homebrew/bin/brew shellenv)"`
`Error: model 'gemma4:latest' not found`	Modell nicht geladen → `ollama pull gemma4:latest`
OpenCode startet, aber wählt falsches Modell oder bricht ab	`opencode.json` fehlt oder leer → `~/.config/opencode/opencode.json` prüfen (siehe Kap. 4)
OpenCode antwortet nur mit JSON `{"name":"bash",…}`	Modell führt Tool-Calls nicht aus → Diagnose in Kap. 7.2 → ggf. Modell auf `qwen2.5:32b` wechseln
Port 11434 belegt	Zweite Ollama-Instanz läuft → `brew services restart ollama`
OpenCode "vergisst" das Projekt	Kein `/init` ausgeführt → in der Session `/init` eingeben

```
Sehr langsame Antworten              Modell zu groß für RAM
                                     → kleineres Modell, z. B.
                                       gemma4:2b oder llama3.2:latest
```

Logs und Service-Status

```
brew services info ollama        # Ollama-Service-Status
ollama ps                        # geladene Modelle und RAM-Verbrauch
ls ~/.local/state/opencode/      # OpenCode-State (Modellauswahl etc.)
```

---

9. Referenz

Worum geht's in diesem Kapitel? Nachschlagewerk — Dateien, Ports, Skripte, Update- und Deinstallationsbefehle. Nicht zum Durchlesen, zum Nachschlagen.

9.1 Dateien

```
Datei                                Zweck
-------------------------------------------------------------------
~/.config/opencode/opencode.json     OpenCode-Konfiguration
~/.local/state/opencode/model.json   zuletzt gewähltes Modell
~/.local/share/opencode/auth.json    Auth-Daten (für Cloud-Provider)
<Projekt>/AGENTS.md                  Projekt-Kontext (via /init)
```

9.2 Ports

```
Port    Dienst
-------------------------
11434   Ollama HTTP-API (inkl. /v1 für OpenAI-kompatibel)
```

9.3 Skripte

```
Skript                Aufgabe
-------------------------------------------------------------------
install-opencode.sh   Einmalige Installation des kompletten Stacks
opencode-up.sh        Ollama starten
opencode-down.sh      Ollama beenden
opencode-start.sh     OpenCode mit Vorab-Checks starten
opencode-status.sh    Aktiven Status anzeigen
```

9.4 Update

```
brew upgrade ollama
ollama pull gemma4:latest
curl -fsSL https://opencode.ai/install | bash        # OpenCode neu installieren
```

9.5 Deinstallation

```
./opencode-down.sh
brew uninstall ollama
rm -rf ~/.config/opencode
```

```
rm -rf ~/.local/state/opencode
rm -rf ~/.local/share/opencode
# OpenCode-Binary: Pfad mit `which opencode` ermitteln und löschen
```

Zusammenfassung

```
OpenCode (CLI)  →  Ollama :11434/v1  →  gemma4:latest
```

Kein Proxy, keine API-Schlüssel, keine Cloud. Nur drei Komponenten, verbunden über den OpenAI-kompatiblen Endpunkt von Ollama. Die größte Stolperfalle ist das Tool-Calling-Verhalten des Modells — Kapitel 7 beschreibt das Vorgehen, wenn Gemma4 dabei nicht zuverlässig ist.

Anhang A: Skripte zum Selbst-Anlegen

Worum geht's in diesem Anhang? Alle fünf Shell-Skripte aus der Anleitung im Volltext — damit du sie ohne separate Lieferung selbst anlegen kannst. Eine Datei pro Skript, Inhalt einfügen, ausführbar machen, fertig.

Vorkenntnisse: Du kannst eine Datei in `nano` anlegen und `chmod +x` ausführen.

Die Anleitung verweist auf fünf Shell-Skripte. Sie sind hier vollständig abgedruckt, damit keine separaten Dateien ausgeliefert werden müssen.

Vorgehen:

1. Ein Verzeichnis anlegen, z. B. `~/opencode-stack`
2. Pro Skript eine Datei mit dem angegebenen Namen anlegen
3. Inhalt einfügen, speichern
4. Einmalig ausführbar machen:

```
cd ~/opencode-stack
chmod +x *.sh
```

A.1 install-opencode.sh

Einmalige Installation des kompletten Stacks. Idempotent — kann mehrfach ausgeführt werden, ohne bereits installierte Komponenten erneut zu laden.

```
#!/bin/sh
# Einmalige Installation des OpenCode-Stacks (Ollama + OpenCode + Gemma4)
# Idempotent — kann mehrfach ausgeführt werden
# Kompatibel mit zsh und bash

set -e

echo "==> Voraussetzungen prüfen"
```

```
if ! command -v brew >/dev/null 2>&1; then
  echo "FEHLER: Homebrew fehlt. Siehe Anleitung Kap. 2."
  exit 1
fi

echo "==> Ollama installieren"
if ! brew list ollama >/dev/null 2>&1; then
  brew install ollama
fi

echo "==> OpenCode installieren"
if ! command -v opencode >/dev/null 2>&1; then
  curl -fsSL https://opencode.ai/install | bash
fi

echo "==> Ollama starten (brew service)"
brew services start ollama >/dev/null 2>&1 || true
for _ in 1 2 3 4 5 6 7 8 9 10; do
  curl -s --max-time 2 http://localhost:11434 >/dev/null 2>&1 && break
  sleep 1
done

echo "==> Gemma4 herunterladen (ca. 6–8 GB, einmalig)"
if ! ollama list 2>/dev/null | grep -q "gemma4:latest"; then
  ollama pull gemma4:latest
fi

echo "==> ~/.config/opencode/opencode.json anlegen (falls fehlend)"
CONFIG_DIR="$HOME/.config/opencode"
CONFIG_FILE="$CONFIG_DIR/opencode.json"
mkdir -p "$CONFIG_DIR"
if [ ! -f "$CONFIG_FILE" ]; then
  cat > "$CONFIG_FILE" <<'EOF'
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      },
      "models": {
        "gemma4:latest": {
          "name": "Gemma 4"
        }
      }
    }
  },
  "model": "ollama/gemma4:latest",
```

```
  "permission": {
    "bash": "allow",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
    "edit": "ask",
    "write": "ask",
    "task": "deny",
    "webfetch": "deny"
  }
}
EOF
  echo "   angelegt: $CONFIG_FILE"
else
  echo "   existiert bereits, unverändert"
fi

echo ""
echo "Fertig. Nächste Schritte:"
echo "  ./opencode-up.sh         # Ollama starten"
echo "  ./opencode-start.sh      # OpenCode starten"
```

A.2 opencode-up.sh

Startet Ollama als launchd-Service.

```
#!/bin/sh
# Startet Ollama (brew service) für OpenCode
# Kompatibel mit zsh und bash

brew services start ollama >/dev/null 2>&1
echo "Ollama: brew service gestartet"

# Auf Bereitschaft warten
for _ in 1 2 3 4 5 6 7 8 9 10; do
  curl -s --max-time 2 http://localhost:11434 >/dev/null 2>&1 && break
  sleep 1
done

if curl -s --max-time 2 http://localhost:11434 >/dev/null 2>&1; then
  echo "Ollama: erreichbar auf http://localhost:11434"
else
  echo "Ollama: noch nicht erreichbar — bitte erneut prüfen mit ./opencode-
status.sh"
fi

echo ""
echo "OpenCode starten mit:"
echo "  ./opencode-start.sh"
```

A.3 opencode-down.sh

Stoppt den Ollama-Service.

```
#!/bin/sh
# Beendet Ollama (brew service)
# Kompatibel mit zsh und bash

if brew services stop ollama >/dev/null 2>&1; then
  echo "Ollama: brew service gestoppt"
else
  echo "Ollama: war nicht aktiv"
fi
```

A.4 opencode-start.sh

Startet OpenCode im aktuellen Arbeitsverzeichnis. Prüft vorab, ob Ollama läuft, ob `opencode.json` vorhanden ist und ob das `opencode`-Binary im PATH liegt.

```
#!/bin/sh
# Startet OpenCode mit Vorab-Checks
# Kompatibel mit zsh und bash

OLLAMA_URL="http://localhost:11434"
CONFIG="$HOME/.config/opencode/opencode.json"

# opencode im PATH?
if ! command -v opencode >/dev/null 2>&1; then
  echo "FEHLER: opencode ist nicht im PATH."
  echo "Installation: ./install-opencode.sh"
  echo "Falls installiert: 'source ~/.zshrc' und Terminal neu öffnen."
  exit 1
fi

# Ollama erreichbar?
if ! curl -s --max-time 2 "$OLLAMA_URL" >/dev/null 2>&1; then
  echo "FEHLER: Ollama läuft nicht ($OLLAMA_URL)"
  echo "Starten mit: ./opencode-up.sh"
  exit 1
fi

# opencode.json vorhanden?
if [ ! -f "$CONFIG" ]; then
  echo "FEHLER: $CONFIG fehlt."
  echo "Bitte zuerst ./install-opencode.sh ausführen."
  exit 1
fi

echo "-----------------------------------------------"
echo " OpenCode → Lokale KI (Gemma4)"
echo " Modell:      gemma4:latest via Ollama"
```

```
echo " Datenschutz: vollständig lokal, kein Internet nötig"
echo " Kosten:      keine"
echo "-----------------------------------------------"
echo ""

opencode "$@"
```

A.5 opencode-status.sh

Zeigt den aktuellen Status von Ollama, geladene Modelle und prüft, ob OpenCode und seine Konfiguration vorhanden sind.

```
#!/bin/sh
# Zeigt den Status des OpenCode-Stacks
# Kompatibel mit zsh und bash

OLLAMA_URL="http://localhost:11434"
CONFIG="$HOME/.config/opencode/opencode.json"

echo "================================"
echo " OpenCode — Stack-Status"
echo "================================"
echo ""

# opencode Binary
if command -v opencode >/dev/null 2>&1; then
  VERSION="$(opencode --version 2>/dev/null | head -1)"
  echo " OpenCode:    installiert ($VERSION)"
else
  echo " OpenCode:    NICHT installiert"
  echo "              Installation: ./install-opencode.sh"
fi

# opencode.json
if [ -f "$CONFIG" ]; then
  MODEL="$(grep -E '"model"\s*:\s*"' "$CONFIG" | head -1 | sed
's/.*"model"[^"]*"\([^"]*\)".*/\1/')"
  echo " Config:      $CONFIG"
  [ -n "$MODEL" ] && echo " Modell:      $MODEL"
else
  echo " Config:      FEHLT ($CONFIG)"
fi

echo ""
echo "------------------------------"
echo " Dienste"
echo "------------------------------"
echo ""

# Ollama
if curl -s --max-time 2 "$OLLAMA_URL" >/dev/null 2>&1; then
```

```
  echo " Ollama:      läuft ($OLLAMA_URL)"
  MODELS=$(ollama list 2>/dev/null | tail -n +2 | awk '{print $1}' | tr '\n' ' ')
  [ -n "$MODELS" ] && echo " Modelle:     $MODELS"
else
  echo " Ollama:      läuft nicht"
  echo "              Start: ./opencode-up.sh"
fi

echo ""
echo "=============================="
echo " Aktionen"
echo "=============================="
echo ""
echo " Stack starten: ./opencode-up.sh"
echo " OpenCode:      ./opencode-start.sh"
echo " Stack stoppen: ./opencode-down.sh"
echo ""
```