Claude Code + Ollama + Gemma4 – Anleitung

Lokales KI-Setup für macOS: Claude Code mit dem lokalen Modell Gemma4 betreiben statt mit der Anthropic Cloud. Datenschutzkonform, offline-fähig, ohne laufende API-Kosten.

Autor: Christian Drapatz

Stand: Mai 2026 · Claude Code + Ollama + Gemma4

Plattform: macOS · Apple Silicon

Version 1.0

Inhaltsverzeichnis

Disclaimer

Diese Anleitung wurde auf Basis öffentlich zugänglicher Quellen eigenständig erstellt und in eigenen Worten auf Deutsch formuliert. Als primäre Quellen dienten die offizielle Ollama-Dokumentation (ollama.com, MIT-Lizenz), öffentlich verfügbare Informationen und Dokumentationen zu Claude Code 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 der genannten Projekte und Werkzeuge. Alle Texte wurden eigenständig strukturiert und formuliert – es erfolgte keine wörtliche Übernahme urheberrechtlich geschützter Formulierungen. Code-Beispiele orientieren sich an öffentlich dokumentierten Beispielen und eigenen Tests. 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 1–6.

```
chmod +x *.sh

./install-localai.sh      # einmalig: alles installieren
./localai-up.sh           # Stack starten (Ollama + LiteLLM)
./claude-local.sh         # Claude Code mit Gemma4
./claude-status.sh        # Status prüfen
./localai-down.sh         # Stack beenden
```

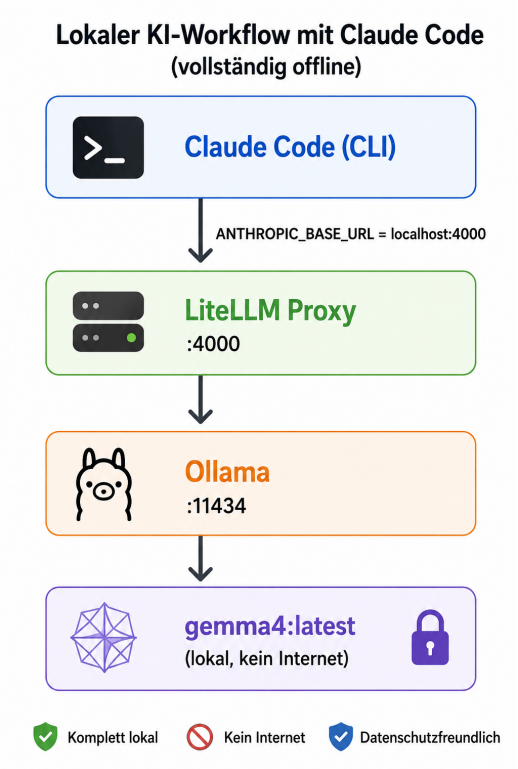
---

1. Konzept und Komponenten

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

Vier Komponenten arbeiten zusammen. Du erfährst, was jede einzelne ist, wer dahintersteht und warum sie im Stack gebraucht wird. Keine

Vorkenntnisse nötig — wer schon weiß, was Claude Code, Ollama, Gemma4 und LiteLLM sind, kann zu Kapitel 2 springen.



1.1 Claude Code

Was? Ein KI-gestützter Coding-Agent für die Kommandozeile. Wichtig: Claude Code ist **kein** Sprachmodell, sondern ein Orchestrierungs-Tool. Es liest und schreibt Dateien, führt Shell- und Git-Befehle aus, plant Schritte und prüft seine Ergebnisse selbst. Das Sprachmodell (Claude in der Cloud oder Gemma4 lokal) liefert nur die "Intelligenz" darunter.

Von wem? Anthropic — derselbe Hersteller wie das Sprachmodell Claude.

Homepage: https://claude.com/claude-code

Installation: npm-Paket `@anthropic-ai/claude-code` (kein Open-Source-Repository).

Warum überhaupt nutzen? Claude Code ersetzt den manuellen Workflow "Code kopieren → in ChatGPT/Claude einfügen → Antwort zurückkopieren" durch einen Agent, der direkt im Terminal arbeitet und Dateien eigenständig manipuliert.

Vorkenntnisse: Grundkenntnisse Terminal/Shell. Node.js wird vom Installer mit installiert, falls noch nicht vorhanden.

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`.

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.

1.4 LiteLLM

Was? Ein API-Übersetzer als lokaler HTTP-Proxy. Claude Code spricht das Anthropic-API-Format. Ollama spricht das OpenAI-kompatible Format. LiteLLM nimmt Anfragen im Anthropic-Format entgegen, übersetzt sie ins OpenAI-Format und reicht sie an Ollama weiter — und übersetzt die Antwort zurück.

Von wem? BerriAI — Open Source (Repository `BerriAI/litellm`).

Homepage: https://www.litellm.ai

Warum hier nötig? Claude Code kann nicht direkt mit Ollama sprechen. LiteLLM ist der notwendige Adapter dazwischen. (Beim parallelen OpenCode-Setup entfällt das, weil OpenCode direkt mit Ollama spricht.)

Einbau: Läuft als lokaler Hintergrund-Prozess auf Port 4000.

---

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, Node.js). Mit dem Schnellcheck unten in 2–5 Minuten erledigt.

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)
Node.js / npm              node --version      brew install node
Python 3 / pip3            python3 --version   liegt macOS bei
```

Homebrew installieren (falls fehlend):

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

2.3 Anthropic API Key (optional)

Nur für den Cloud-Betrieb nötig. Kostenlos unter https://console.anthropic.com → "API Keys" → "Create Key". Der Key beginnt mit `sk-ant-…`. Für den rein lokalen Betrieb mit Gemma4 wird kein Key benötigt.

---

3. Installation

Worum geht's in diesem Kapitel? Hier wird der vollständige Stack installiert: Claude Code, Ollama, LiteLLM 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-localai.sh` führt alle Schritte aus, ist idempotent (mehrfach ausführbar) und lädt `gemma4:latest` herunter.

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

3.2 Manuelle Installation

Falls du Schritt für Schritt vorgehen willst:

```
# Claude Code
npm install -g @anthropic-ai/claude-code
claude --version

# Ollama
brew install ollama
ollama --version

# Python 3.13 (LiteLLM benötigt max. Python 3.13, siehe Kap. 8)
brew install python@3.13
```

```
# LiteLLM via pipx mit Python 3.13 (isolierte Python-Installation)
brew install pipx
pipx ensurepath
pipx install --python "$(brew --prefix python@3.13)/bin/python3.13"
'litellm[proxy]'
litellm --version

# 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 eine YAML-Datei an, die LiteLLM mitteilt, wie Anfragen von Claude Code auf Gemma4 in Ollama umgeleitet werden. Optional kannst du das Kontextfenster anpassen.
Was passiert hier konkret? Claude Code fragt unter bestimmten Modellnamen (z. B. `claude-3-5-sonnet-20241022`) an. Diese Namen existieren bei LiteLLM, der die Anfragen dann an Gemma4 umleitet.
Vorkenntnisse: Du kannst eine Textdatei mit `nano` öffnen. Mehr nicht.

4.1 litellm_config.yaml

Liegt im Home-Verzeichnis und legt fest, welches Modell unter welchem Namen erreichbar ist.

```
nano ~/litellm_config.yaml
```

Inhalt:

```
litellm_settings:
  drop_params: true

model_list:
  # Aktuelle Claude Code Modellnamen (v2.1+)
  # ollama_chat/ nutzt /api/chat statt /api/generate → Tool Use aktiv
  - model_name: claude-haiku-4-5-20251001
    litellm_params:
      model: ollama_chat/gemma4:latest
      api_base: http://localhost:11434
      num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

  - model_name: claude-sonnet-4-6
    litellm_params:
      model: ollama_chat/gemma4:latest
```

```
    api_base: http://localhost:11434
    num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

- model_name: claude-opus-4-7
  litellm_params:
    model: ollama_chat/gemma4:latest
    api_base: http://localhost:11434
    num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

# Ältere Modellnamen (Fallback)
- model_name: claude-3-5-sonnet-20241022
  litellm_params:
    model: ollama_chat/gemma4:latest
    api_base: http://localhost:11434
    num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

- model_name: claude-3-opus-20240229
  litellm_params:
    model: ollama_chat/gemma4:latest
    api_base: http://localhost:11434
    num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext
```

`drop_params: true` sorgt dafür, dass Claude-Code-spezifische Parameter (z. B. `context_management`), die Ollama nicht kennt, stillschweigend ignoriert werden statt einen Fehler zu liefern.

Warum `ollama_chat/` statt `ollama/`? LiteLLM kennt zwei Ollama-Provider: `ollama/` nutzt den alten `/api/generate`-Endpunkt ohne Tool-Use-Unterstützung. `ollama_chat/` nutzt `/api/chat` und aktiviert das Tool-Use-Protokoll — Claude Code benötigt das zwingend, da es fast alle Aktionen (Bash, Read, Edit …) als Tool Calls ausführt.

Warum so viele Namen? Claude Code v2+ verwendet intern mehrere Modellnamen gleichzeitig — je nach Aufgabe unterschiedliche (Hauptmodell, Hintergrundanalyse, Token-Zählung). Das UI zeigt z. B. „Sonnet 4.6", weil Claude Code diesen Namen anfragt. LiteLLM leitet alle diese Namen auf Gemma4 weiter — Claude Code selbst bemerkt die Umleitung nicht.

Fehlt ein Name in der Config, antwortet LiteLLM mit `400 Bad Request` (siehe Troubleshooting Kap. 8).

4.2 Kontextfenster und Geschwindigkeit

`num_ctx` ist der wichtigste Geschwindigkeitshebel. Ollama setzt für Gemma4 standardmäßig 32 768 Token — das macht Antworten sehr langsam. Die Config in Kap. 4.1 setzt 8 192, was für die meisten Claude Code Aufgaben ausreicht und deutlich schneller ist.

```
num_ctx    Geschwindigkeit    Geeignet für
-------------------------------------------------------------
```

```
4096        am schnellsten      einfache Fragen, 1 Datei
8192        schnell             normale Aufgaben (Empfehlung)
16384       mittel              größere Dateien / mehr Kontext
32768       sehr langsam        große Codebasen (Standard Ollama)
```

Wert anpassen in `~/litellm_config.yaml`:

```
litellm_params:
  model: ollama_chat/gemma4:latest
  api_base: http://localhost:11434
  num_ctx: 8192        # hier anpassen
```

Nach jeder Änderung Stack neu starten:

```
./localai-down.sh && ./localai-up.sh
```

Hinweis: Größeres Kontextfenster braucht deutlich mehr GPU-Speicher und Rechenzeit. Bei 16 GB RAM ist 8 192–16 384 ein realistischer Bereich.

---

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:

- **Claude Code** ist installiert und über den Befehl `claude` aufrufbar.
- **Ollama** läuft im Hintergrund als launchd-Dienst (überlebt den Reboot).
- **Gemma4** liegt als ca. 6–8 GB großes Modell im Ollama-Cache und kann geladen werden.
- **LiteLLM** läuft als Hintergrundprozess auf Port 4000 und vermittelt zwischen Claude Code und Ollama.
- **`~/litellm_config.yaml`** sagt LiteLLM, welche "Anthropic-Modelle" auf Gemma4 umgeleitet werden.

Was kannst du damit machen?

- Code lesen, schreiben, refactoren, dokumentieren — **vollständig offline**, ohne dass auch nur ein Token den Mac verlässt.
- Mit einer einzigen Umgebungsvariable (`ANTHROPIC_BASE_URL`) zwischen Cloud (Claude) und lokal (Gemma4) umschalten — Kapitel 6.
- Sensible Projekte (NDA, regulierte Branchen, firmeninternes IP) bleiben auf deinem Mac.
- Keine API-Kosten, keine Rate Limits, keine Internetabhängigkeit.

Was solltest du nicht erwarten?

Gemma4 ist deutlich schwächer als das Cloud-Claude. Komplexe Refactorings

über viele Dateien, große Architekturentscheidungen oder Aufgaben mit sehr großem Kontext bleiben Cloud-Domäne. Eine ehrliche Einordnung pro Aufgabentyp folgt in Kapitel 7.

5.1 Lokalen Stack starten und stoppen

Mit den mitgelieferten Skripten:

```
./localai-up.sh        # Ollama (brew service) + LiteLLM im Hintergrund
./localai-down.sh      # beides beenden
```

Manuell:

```
brew services start ollama
litellm --config ~/litellm_config.yaml --port 4000
```

`brew services start` läuft als launchd-Dienst und überlebt einen Reboot. Wer Ollama nur temporär will, nutzt stattdessen `ollama serve` in einem eigenen Terminal-Tab.

5.2 Cloud-Betrieb starten

```
./claude-cloud.sh
```

Manuell:

```
unset ANTHROPIC_BASE_URL
export ANTHROPIC_API_KEY="sk-ant-..."
claude
```

5.3 Status prüfen

```
./claude-status.sh
```

Zeigt aktiven Modus, ob Ollama und LiteLLM laufen und welche Modelle geladen sind.

---

6. Umschalten zwischen Cloud und Lokal

Worum geht's in diesem Kapitel? Eine einzige Umgebungsvariable entscheidet, ob Claude Code mit der Anthropic Cloud oder mit deinem lokalen Gemma4 spricht. Du lernst die richtige Shell-Konfigurationsdatei (`~/.zshrc` für zsh, `~/.bash_profile` für bash) und richtest Komfortfunktionen `use-claude` / `use-local` / `ai-status` ein.

Vorkenntnisse: Du weißt, welche Shell auf deinem Mac läuft (Standard seit Catalina: zsh). Falls unklar: `echo $SHELL` ausführen.

Das Umschalten erfolgt über die Umgebungsvariable `ANTHROPIC_BASE_URL`:

```
gesetzt        →  Claude Code spricht den lokalen LiteLLM-Proxy an
nicht gesetzt  →  Claude Code spricht die Anthropic Cloud an
```

6.1 Aktive Shell prüfen

```
echo $SHELL
```

Zuordnung:

```
/bin/zsh   →  ~/.zshrc           (Standard seit macOS Catalina)
/bin/bash  →  ~/.bash_profile    (ältere macOS-Versionen)
```

Auf macOS wird `~/.bashrc` von Login-Shells nicht automatisch geladen.

Für bash bitte `~/.bash_profile` verwenden.

Hinweis zu `~/.zprofile`: Wird einmal pro Login geladen, ideal für PATH und langlebige ENV-Variablen. `~/.zshrc` wird bei jedem neuen Terminal geladen, ideal für Funktionen und Aliase. Für dieses Setup reicht eine Datei (`~/.zshrc` bzw. `~/.bash_profile`).

6.2 Shell-Funktionen einrichten

Datei öffnen:

```
nano ~/.zshrc            # zsh
nano ~/.bash_profile     # bash
```

Am Ende ergänzen:

```
# Claude (Anthropic Cloud)
use-claude() {
  unset ANTHROPIC_BASE_URL
  export ANTHROPIC_API_KEY="sk-ant-DEIN-KEY"
  echo "Aktiv: Anthropic Cloud (Claude)"
}

# Lokale KI (Gemma4 via Ollama + LiteLLM)
use-local() {
  export ANTHROPIC_BASE_URL="http://localhost:4000"
  export ANTHROPIC_API_KEY="local"
  echo "Aktiv: Lokale KI (Gemma4)"
}

# Status anzeigen
ai-status() {
  if [ -z "$ANTHROPIC_BASE_URL" ]; then
    echo "Aktiv: Anthropic Cloud"
  else
    echo "Aktiv: Lokale KI → $ANTHROPIC_BASE_URL"
  fi
}
```

Datei neu laden:

```
source ~/.zshrc          # oder ~/.bash_profile
```

Ab jetzt stehen `use-claude`, `use-local` und `ai-status` in jedem Terminal zur Verfügung.

6.3 Skript-Variante

Wer keine Shell-Funktionen will, nutzt direkt die Skripte:

```
./claude-cloud.sh        # Claude Code mit Cloud starten
./claude-local.sh        # Claude Code mit lokaler KI starten
./claude-status.sh       # aktiven Modus anzeigen
```

---

7. Praxisverhalten und Grenzen

Worum geht's in diesem Kapitel? Eine ehrliche Einordnung: Was funktioniert lokal gut, was nicht? Tabellen mit Empfehlungen pro Aufgabentyp, damit du weißt, wann sich der Cloud-Wechsel lohnt.
Vorkenntnisse: Keine. Lesenswert auch vor der Installation, falls du unsicher bist, ob sich der Aufwand lohnt.

7.1 Was sich gegenüber Claude (Cloud) ändert

Das Agent-Framework von Claude Code bleibt gleich — Shell, Tool Calling, Git, Dateioperationen funktionieren unverändert. Was sich ändert, ist das Sprachmodell:

- Antwortqualität: gut für einfache Aufgaben, schwächer bei Komplexität
- Kontext: 8k–32k Token lokal vs. 200k+ in der Cloud
- Geschwindigkeit: ca. 15–40 Token/s lokal, ca. 50–100+ in der Cloud
- Selbstkorrektur: Gemma4 wiederholt Fehler eher als sie zu erkennen

7.2 Empfehlungen pro Aufgabe

Aufgabe	Gemma4 (lokal)	Claude (Cloud)
Einzelne Funktion erklären	gut	sehr gut
Einfache Bugs in einer Datei	gut	sehr gut
Code dokumentieren	gut	sehr gut
Kleines Refactoring (1 Datei)	gut	sehr gut
Refactoring über viele Dateien	begrenzt	sehr gut
Architekturplanung	schwach	sehr gut
Große Codebasen analysieren	nein	ja
App Store Connect API	nein	ja
Swift Concurrency (komplex)	begrenzt	sehr gut
Selbstkorrektur bei Compile-Errors	schwach	sehr gut
Offline-Betrieb	ja	nein
DSGVO-konform	ja	eingeschränkt
Kostenlos	ja	nein

7.3 Vorteile des lokalen Betriebs

- Datenschutz: kein Token verlässt den Mac. Relevant bei NDA, regulierten Branchen (Gesundheit, Finanzen, Recht), firmeninternem Code.
- Keine API-Kosten nach einmaliger Hardware-/Strom-Investition.
- Offline-fähig: Flugzeug, gesicherte Netze, Firewall-Umgebungen.
- Keine Rate Limits.
- Volle Kontrolle: Modellwahl, Kontextfenster, Updates entscheidest du.

7.4 Praktischer Hybridansatz

Lokal für sensible Daten, kleine Aufgaben und Offline-Arbeit. Cloud für komplexe Refactorings, Architekturentscheidungen und App-Store-Connect. Das Umschalten dauert eine Sekunde (`use-local` / `use-claude`).

7.5 Persönliche Bewertung und Modellwahl

Testen und selbst urteilen

Ob dieses Setup für dich geeignet ist, musst du selbst entscheiden. Zumindest läuft die KI vollständig lokal — datenschutzkonform, kostenlos, offline. Ob die Qualität für deine konkreten Aufgaben ausreicht, zeigt nur der eigene Test. Starte mit einfachen Aufgaben (Kap. 8.1) und steigere die Komplexität schrittweise.

Gemma4 ist nur ein Startpunkt

Gemma4 ist in dieser Anleitung das Standardmodell — nicht weil es das beste ist, sondern weil es gut dokumentiert, breit verfügbar und für Apple Silicon optimiert ist. Für manche Aufgaben und Arbeitsstile sind andere Modelle besser geeignet. Das ist normal und Teil des Setups.

Anderes Modell verwenden — drei Schritte

1. Modell herunterladen:

```
ollama pull <modellname>
```

Beispiele aus dem Ollama-Katalog (https://ollama.com/library):

```
qwen2.5-coder:7b    Coding-spezialisiert, gutes Tool Use
llama3.1:8b         Starkes Allgemeinmodell
mistral:7b          Schnell, präzise Antworten
phi4:14b            Microsofts Modell, stark bei Code
```

2. In `~/litellm_config.yaml` den Modellnamen überall ersetzen:

```
model: ollama_chat/gemma4:latest
→
model: ollama_chat/<modellname>
```

3. Stack neu starten:

```
./localai-down.sh && ./localai-up.sh
```

Wichtig: Tool Use prüfen

Claude Code funktioniert nur mit Modellen, die Tool Use unterstützen. Vor dem Wechsel prüfen:

```
ollama show <modellname> | grep -A5 Capabilities
```

In der Ausgabe muss `tools` erscheinen. Fehlt es, gibt das Modell Tool-Calls als Rohtext aus und Claude Code funktioniert nicht korrekt (siehe Kap. 8.3).

Prefix `ollama_chat/` immer verwenden — nicht `ollama/`. Nur `ollama_chat/` aktiviert den Chat-Endpunkt mit Tool-Use-Protokoll.

8. Troubleshooting

Worum geht's in diesem Kapitel? Zuerst: ein praktischer Test, der zeigt, dass alles funktioniert. Danach häufige Fehlermeldungen und ihre Lösung — tabellarisch. Am Ende: Wo finde ich Logs und Service-Status?
Vorkenntnisse: Kapitel 5 (Betrieb) durchgegangen.

8.1 Offline-Test — So prüfst du, ob alles funktioniert

Dieser Test beweist, dass Claude Code wirklich lokal läuft und kein Internet braucht.

Schritt 1 — Internet trennen

WLAN ausschalten oder Ethernet-Kabel ziehen. Der Mac muss vollständig offline sein.

Schritt 2 — Stack starten

```
./localai-up.sh
```

Beide Dienste (Ollama + LiteLLM) starten ohne Internetverbindung, weil Gemma4 bereits lokal gespeichert ist.

Schritt 3 — Claude Code im lokalen Modus starten

```
./claude-local.sh
```

Der Banner erscheint und bestätigt den lokalen Modus.

Schritt 4 — Testanfragen stellen

Einfacher Allgemeintest:

```
Schreibe einen Geburtstagsgruß auf Spanisch.
```

Coding-Test:

```
Schreibe Beispielcode in Swift für Push Notifications.
Zeige nur den relevanten Code, keine Erklärungen.
```

Wenn Gemma4 antwortet, läuft alles korrekt — vollständig ohne Internet.

Schritt 5 — Internet wieder einschalten

Danach WLAN wieder aktivieren. Der lokale Stack läuft weiter und muss nicht neu gestartet werden.

8.2 Hinweise zur Nutzung — was sich gegenüber Cloud Claude unterscheidet

Sprache

Claude Code hat eine eingestellte Sprache (Standard: Englisch). Gemma4 antwortet in der Sprache, in der die Frage gestellt wird — aber nicht immer zuverlässig. Am sichersten: die gewünschte Sprache explizit nennen.

```
Antworte auf Deutsch. Schreibe einen Geburtstagsgruß auf Spanisch.
```

Wer Claude Code dauerhaft auf Deutsch einstellen will:

```
/config
```

Dort unter "Language" auf Deutsch wechseln. Trotzdem gilt: lokal immer

die Sprache im Prompt mitangeben, da Gemma4 schwächer darin ist als Cloud-Claude.

Präzise Prompts

Gemma4 braucht genauere Anweisungen als Cloud-Claude. Vage Fragen liefern vage Antworten. Konkrete Vorgaben helfen:

```
Schlecht:  "Zeig mir Swift Code"
Besser:    "Schreibe eine Swift-Funktion, die eine lokale Push
            Notification nach 5 Sekunden auslöst. Nur den Code,
            keine Erklärung, kein Kommentar."
```

Gute Prompt-Struktur für Gemma4:

```
1. Sprache: "Antworte auf Deutsch."
2. Aufgabe: Was genau soll erstellt werden?
3. Format:  Wie soll die Antwort aussehen? (Code, Liste, Satz …)
4. Umfang:  Kurz / ausführlich / nur das Wesentliche?
```

8.3 Fehlermeldungen und Lösungen

```
Symptom                              Ursache / Lösung
-------------------------------------------------------------------
claude: command not found            npm-PATH fehlt
                                     → Terminal neu öffnen oder
                                       source ~/.zshrc

ollama: command not found            brew-PATH nicht aktiv
                                     (Apple Silicon)
                                     → in ~/.zprofile ergänzen:
                                       eval "$(/opt/homebrew/bin/brew shellenv)"

Error: model 'gemma4:latest'         Modell nicht geladen
not found                            → ollama pull gemma4:latest

Port 11434 belegt                    Zweite Ollama-Instanz läuft
                                     → brew services restart ollama

Port 4000 belegt                     LiteLLM läuft bereits
                                     → lsof -i :4000, dann
                                       ./localai-down.sh

LiteLLM antwortet 401                ANTHROPIC_API_KEY leer
                                     → export ANTHROPIC_API_KEY=local

Sehr langsame Antworten              Ollama setzt num_ctx standard-
                                     mäßig auf 32 768 — das ist für
                                     lokale Modelle viel zu groß
                                     → in ~/litellm_config.yaml
                                       num_ctx: 8192 setzen
                                       (siehe Kap. 4.2)
```

	→ ./localai-down.sh && ./localai-up.sh
Claude Code nutzt trotz use-local die Cloud	ANTHROPIC_BASE_URL im aktuellen Tab nicht gesetzt → echo $ANTHROPIC_BASE_URL prüfen, ggf. use-local erneut
pip: error failed-wheel-build / orjson baut nicht / "Python 3.14 is newer than PyO3's maximum"	macOS-Standard-Python ist 3.14+, orjson/PyO3 unterstützt max. 3.13 → brew install python@3.13 → pipx install --python \ "$(brew --prefix python@3.13)/bin/python3.13" \ 'litellm[proxy]' install-localai.sh macht das automatisch (ab dieser Version)
FEHLER: LiteLLM Proxy läuft nicht — obwohl localai-up.sh meldet "LiteLLM: gestartet"	Der /health-Endpoint macht echte Modellaufrufe bei Ollama und antwortet langsamer als 2 s → claude-local.sh prüft jetzt /health/liveliness (sofortige Antwort, kein Modelltest) mit 5 s Timeout
400 Bad Request im LiteLLM-Log / ProxyModelNotFoundError / "Invalid model name passed in model=claude-haiku-4-5-20251001"	Claude Code v2+ sendet moderne Modellnamen (claude-sonnet-4-6, claude-haiku-4-5-20251001 …), die nicht in der Config stehen → ~/litellm_config.yaml ergänzen (alle aktuellen Namen eintragen, siehe Kap. 4.1) → ./localai-down.sh && ./localai-up.sh
API Error: UnsupportedParamsError / "ollama does not support parameters: ['context_management']"	Claude Code v2+ schickt Parameter wie context_management mit, die Ollama nicht kennt → in ~/litellm_config.yaml am Anfang ergänzen: litellm_settings: drop_params: true → ./localai-down.sh && ./localai-up.sh (litellm_config.yaml in Kap. 4.1 enthält diese Einstellung bereits)
Modell gibt Tool-Calls als Raw-JSON aus statt sie	ollama/ nutzt /api/generate ohne Tool-Use-Unterstützung

```
auszuführen, z. B.:                    → in ~/litellm_config.yaml alle
{"tool_calls": [{"function":             model: ollama/… ersetzen durch
"Bash", ...}]}                           model: ollama_chat/…
                                       → ./localai-down.sh &&
                                         ./localai-up.sh
                                       ollama_chat/ nutzt /api/chat und
                                       aktiviert das Tool-Use-Protokoll
                                       (siehe Kap. 4.1)
```

8.4 Logs und Service-Status

```
tail -f /tmp/litellm.log          # LiteLLM (wenn via localai-up.sh)
brew services info ollama         # Ollama-Service-Status
ollama ps                         # geladene Modelle und RAM-Verbrauch
```

---

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
-------------------------------------------------------------------
~/litellm_config.yaml           LiteLLM-Konfiguration
~/.zshrc / ~/.bash_profile      Shell-Funktionen use-claude/use-local
~/.claude/settings.json         Claude Code intern (selten manuell)
/tmp/litellm.log                LiteLLM-Logs (via localai-up.sh)
/tmp/litellm.pid                LiteLLM-PID (via localai-up.sh)
```

9.2 Ports

```
Port    Dienst
-------------------------
11434   Ollama HTTP-API
4000    LiteLLM Proxy
```

9.3 Skripte

```
Skript                Aufgabe
-------------------------------------------------------------------
install-localai.sh    Einmalige Installation des kompletten Stacks
localai-up.sh         Ollama + LiteLLM starten
localai-down.sh       Ollama + LiteLLM beenden
claude-cloud.sh       Claude Code mit Anthropic Cloud starten
claude-local.sh       Claude Code mit Gemma4 (lokal) starten
claude-status.sh      Aktiven Modus + Service-Status anzeigen
```

9.4 Update

```
brew upgrade ollama
ollama pull gemma4:latest
pipx upgrade litellm
npm update -g @anthropic-ai/claude-code
```

9.5 Deinstallation

```
./localai-down.sh
brew uninstall ollama
pipx uninstall litellm
npm uninstall -g @anthropic-ai/claude-code
rm -f ~/litellm_config.yaml
```

Zusammenfassung

```
ANTHROPIC_BASE_URL nicht gesetzt   →  Anthropic Cloud  →  Claude
ANTHROPIC_BASE_URL=localhost:4000  →  LiteLLM Proxy    →  Ollama  →  Gemma4
```

Umschalten heißt: eine Umgebungsvariable setzen oder löschen. Claude Code selbst muss nicht neu installiert oder umkonfiguriert werden.

Anhang A: Skripte zum Selbst-Anlegen

Worum geht's in diesem Anhang? Alle sechs 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 sechs Shell-Skripte. Sie sind hier vollständig abgedruckt, damit keine separaten Dateien ausgeliefert werden müssen.

Vorgehen:

1. Ein Verzeichnis anlegen, z. B. `~/localai`
2. Pro Skript eine Datei mit dem angegebenen Namen anlegen (z. B. mit `nano ~/localai/install-localai.sh`)
3. Inhalt einfügen, speichern
4. Einmalig ausführbar machen:

```
cd ~/localai
chmod +x *.sh
```

A.1 install-localai.sh

Einmalige Installation des kompletten Stacks. Idempotent — kann mehrfach

ausgeführt werden, ohne bereits installierte Komponenten erneut zu laden.

```
#!/bin/sh
# Einmalige Installation des lokalen KI-Stacks (Ollama + LiteLLM + 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

if ! command -v npm >/dev/null 2>&1; then
  echo "==> Node.js installieren"
  brew install node
fi

echo "==> Claude Code installieren"
if ! command -v claude >/dev/null 2>&1; then
  npm install -g @anthropic-ai/claude-code
fi

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

echo "==> Python 3.13 installieren (LiteLLM benötigt max. Python 3.13)"
if ! brew list python@3.13 >/dev/null 2>&1; then
  brew install python@3.13
fi
PYTHON313="$(brew --prefix python@3.13)/bin/python3.13"

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

echo "==> LiteLLM installieren"
if ! pipx list 2>/dev/null | grep -q litellm; then
  pipx install --python "$PYTHON313" 'litellm[proxy]'
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 "==> ~/litellm_config.yaml anlegen (falls fehlend)"
if [ ! -f "$HOME/litellm_config.yaml" ]; then
  cat > "$HOME/litellm_config.yaml" <<'EOF'
litellm_settings:
  drop_params: true

model_list:
  # Aktuelle Claude Code Modellnamen (v2.1+)
  - model_name: claude-haiku-4-5-20251001
    litellm_params:
      model: ollama_chat/gemma4:latest
      api_base: http://localhost:11434
      num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

  - model_name: claude-sonnet-4-6
    litellm_params:
      model: ollama_chat/gemma4:latest
      api_base: http://localhost:11434
      num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

  - model_name: claude-opus-4-7
    litellm_params:
      model: ollama_chat/gemma4:latest
      api_base: http://localhost:11434
      num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

  # Ältere Modellnamen (Fallback)
  - model_name: claude-3-5-sonnet-20241022
    litellm_params:
      model: ollama_chat/gemma4:latest
      api_base: http://localhost:11434
      num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext

  - model_name: claude-3-opus-20240229
    litellm_params:
      model: ollama_chat/gemma4:latest
      api_base: http://localhost:11434
      num_ctx: 8192      # 8192 = guter Kompromiss Geschwindigkeit/Kontext
EOF
  echo "   angelegt"
else
```

```
  echo "   existiert bereits, unverändert"
fi

echo ""
echo "Fertig. Nächste Schritte:"
echo "  ./localai-up.sh        # Stack starten"
echo "  ./claude-local.sh      # Claude Code mit Gemma4"
```

A.2 localai-up.sh

Startet Ollama (als launchd-Service via `brew services`) und LiteLLM (im Hintergrund mit PID-Datei und Log).

```
#!/bin/sh
# Startet den lokalen KI-Stack: Ollama (brew service) + LiteLLM (Hintergrund)
# Kompatibel mit zsh und bash

CONFIG="$HOME/litellm_config.yaml"
PID_FILE="/tmp/litellm.pid"
LOG_FILE="/tmp/litellm.log"

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

# Ollama als launchd-Service starten
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

# LiteLLM: prüfen ob bereits aktiv
if [ -f "$PID_FILE" ] && kill -0 "$(cat "$PID_FILE")" 2>/dev/null; then
  echo "LiteLLM: läuft bereits (PID $(cat "$PID_FILE"))"
else
  nohup litellm --config "$CONFIG" --port 4000 > "$LOG_FILE" 2>&1 &
  echo $! > "$PID_FILE"
  echo "LiteLLM: gestartet (PID $(cat "$PID_FILE"))"
  echo "         Log: $LOG_FILE"
fi

echo ""
echo "Stack läuft. Claude Code starten mit:"
echo "  ./claude-local.sh"
```

A.3 localai-down.sh

Beendet LiteLLM und stoppt den Ollama-Service.

```
#!/bin/sh
# Beendet den lokalen KI-Stack
# Kompatibel mit zsh und bash

PID_FILE="/tmp/litellm.pid"

# LiteLLM beenden
if [ -f "$PID_FILE" ]; then
  PID=$(cat "$PID_FILE")
  if kill -0 "$PID" 2>/dev/null; then
    kill "$PID"
    echo "LiteLLM: PID $PID beendet"
  fi
  rm -f "$PID_FILE"
else
  # Fallback: alle litellm-Prozesse mit unserer Config
  if pkill -f "litellm --config" 2>/dev/null; then
    echo "LiteLLM: beendet (Fallback)"
  fi
fi

# Ollama stoppen
if brew services stop ollama >/dev/null 2>&1; then
  echo "Ollama:  brew service gestoppt"
fi

echo ""
echo "Lokaler Stack gestoppt."
```

A.4 claude-cloud.sh

Startet Claude Code im Cloud-Modus (Anthropic API). Erwartet, dass `ANTHROPIC_API_KEY` gesetzt ist.

```
#!/bin/sh
# Claude Code mit Anthropic Cloud (Claude 3.5 / 4)
# Kompatibel mit zsh und bash

# Sicherstellen, dass kein lokaler Proxy aktiv ist
unset ANTHROPIC_BASE_URL

# Anthropic API Key prüfen
if [ -z "$ANTHROPIC_API_KEY" ]; then
  echo "FEHLER: ANTHROPIC_API_KEY ist nicht gesetzt."
  echo "Setzen mit: export ANTHROPIC_API_KEY=\"sk-ant-...\""
  exit 1
fi

echo "-------------------------------------------"
```

```
echo " Claude Code → Anthropic Cloud"
echo " Modell:      Claude 3.5 / 4 (Anthropic)"
echo " Datenschutz: Cloud — Daten verlassen den Mac"
echo "-----------------------------------------------"
echo ""

# Claude Code starten, Argumente durchreichen
claude "$@"
```

A.5 claude-local.sh

Startet Claude Code im Lokal-Modus (Gemma4 via Ollama + LiteLLM). Prüft vorab, ob beide Dienste erreichbar sind, und gibt eine klare Fehlermeldung mit Lösungsweg aus, wenn nicht.

```
#!/bin/sh
# Claude Code mit lokaler KI (Gemma4 via Ollama + LiteLLM Proxy)
# Kompatibel mit zsh und bash

OLLAMA_URL="http://localhost:11434"
LITELLM_URL="http://localhost:4000"

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

# LiteLLM prüfen (/health/liveliness antwortet sofort ohne Modell-Test)
if ! curl -s --max-time 5 "$LITELLM_URL/health/liveliness" >/dev/null 2>&1; then
  echo "FEHLER: LiteLLM Proxy läuft nicht ($LITELLM_URL)"
  echo "Starten mit: ./localai-up.sh"
  exit 1
fi

# Claude Code auf lokalen Proxy zeigen
export ANTHROPIC_BASE_URL="$LITELLM_URL"
export ANTHROPIC_API_KEY="local"

echo ""
echo "╔═══════════════════════════════════════════════╗"
echo "║   LOKALER MODUS — Gemma4 via Ollama           ║"
echo "╠═══════════════════════════════════════════════╣"
echo "║  Datenschutz: vollständig lokal               ║"
echo "║  Kosten:      keine                           ║"
echo "╠═══════════════════════════════════════════════╣"
echo "║  ⚠️  Claude Code zeigt 'Sonnet 4.6' an —      ║"
echo "║     das ist nur der interne Routing-Name.     ║"
echo "║     Tatsächlich läuft: Gemma4 (lokal).        ║"
echo "╚═══════════════════════════════════════════════╝"
```

```
echo ""

claude "$@"
```

A.6 claude-status.sh

Zeigt den aktiven Modus, geladene Modelle und den Status beider Dienste.

```
#!/bin/sh
# Zeigt den aktuellen Status der Claude Code KI-Konfiguration
# Kompatibel mit zsh und bash

OLLAMA_URL="http://localhost:11434"
LITELLM_URL="http://localhost:4000"
PID_FILE="/tmp/litellm.pid"

echo "=============================="
echo " Claude Code — KI-Status"
echo "=============================="
echo ""

# Aktive Konfiguration
if [ -z "$ANTHROPIC_BASE_URL" ]; then
  echo " Aktive KI:   Anthropic Cloud (Claude)"
  echo " Modell:      Claude 3.5 / 4"
  echo " Datenschutz: Cloud — Daten verlassen den Mac"
  if [ -z "$ANTHROPIC_API_KEY" ]; then
    echo " API Key:     NICHT gesetzt"
  else
    KEY_PREVIEW="$(printf '%s' "$ANTHROPIC_API_KEY" | cut -c1-10)..."
    echo " API Key:     $KEY_PREVIEW"
  fi
else
  echo " Aktive KI:   Lokale KI (Gemma4 via Ollama)"
  echo " Base URL:    $ANTHROPIC_BASE_URL"
  echo " Datenschutz: vollständig lokal"
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: ./localai-up.sh"
```

```
fi

echo ""

# LiteLLM
if curl -s --max-time 2 "$LITELLM_URL/health" >/dev/null 2>&1; then
  PID_INFO=""
  [ -f "$PID_FILE" ] && PID_INFO=" (PID $(cat "$PID_FILE"))"
  echo " LiteLLM:     läuft ($LITELLM_URL)$PID_INFO"
else
  echo " LiteLLM:     läuft nicht"
  echo "              Start: ./localai-up.sh"
fi

echo ""
echo "================================"
echo " Backend wechseln"
echo "================================"
echo ""
echo " Cloud: ./claude-cloud.sh"
echo " Lokal: ./claude-local.sh"
echo ""
```