Lokale KI in Xcode 27 — Ollama & Coding-LLMs für Entwickler

Zielgruppe: iOS-, macOS- und Swift-Entwickler, die in Xcode 27 mit einem lokalen, kostenlosen und datenschutzfreundlichen KI-Modell arbeiten möchten — ohne Cloud-Anbindung, ohne API-Kosten, ohne dass Code das Gerät verlässt.

Stand: Xcode 27 Beta 3 (macOS 27 „Golden Gate"), Juli 2026

Voraussetzung: Apple Silicon Mac (M1 oder neuer) — Xcode 27 unterstützt keine Intel-Macs mehr

Inhaltsverzeichnis

1. Warum lokale KI in Xcode?

Xcode 27 bietet KI-Unterstützung in zwei Formen: den **Coding Assistant** (reaktiv — man fragt, er antwortet) und **Coding Agents** (autonom — man gibt ein Ziel vor, der Agent plant und arbeitet mehrere Schritte selbstständig ab). Beide lassen sich an verschiedene Modell-Anbieter anschließen: Apples On-Device-Modell, Private Cloud Compute (PCC), Cloud-Anbieter wie Claude, OpenAI oder Gemini — und eben auch **selbst gehostete, lokale Modelle** über Ollama.

1.1 Chat, Agent und MCP

Chat (Coding Assistant): die reaktive Oberfläche im Editor. Man stellt eine Frage oder markiert Code und bekommt eine Antwort; von selbst passiert nichts. Gut für „Explain this code", Fehlerdiagnose, kurze Vervollständigungen. Jede Änderung muss man selbst bestätigen. Einrichtung mit Ollama: Kapitel 5.

Agent (Coding Agent): die autonome Variante. Man beschreibt ein Ziel (z. B. per `/plan`), der Agent durchsucht die Codebasis, schreibt Code über mehrere Dateien, führt Builds und Tests aus und iteriert, bis das Ziel erreicht ist. Ergebnisse erscheinen als Diffs, Pläne und Preview-Snapshots, die man am Ende prüft. Für lokale Modelle gilt: nur Modelle mit zuverlässigem Tool-Calling (Kapitel 4.1) eignen sich als Agent-Backend.

MCP (Model Context Protocol): kein dritter Modus, sondern eine Erweiterungsschicht für beide. MCP definiert, wie Chat oder Agent auf externe Werkzeuge und Datenquellen zugreifen dürfen — etwa ein zweites, lokales Ollama-Modell als spezialisiertes Werkzeug (Kapitel 7). Man richtet MCP-Server einmal ein (`.xcode/mcp-config.json`) und gibt sie explizit frei.

Baustein	Interaktion	Typischer Einsatz	Kapitel
Chat	Reaktiv, ein Frage-Antwort-Schritt	Erklärungen, Diagnose, kleine Vervollständigungen	5
Agent	Autonom, mehrstufig	Feature-Implementierung, Multi-File-Refactoring, Bugfixing	5.1, 9
MCP	Erweitert Chat/Agent um Werkzeuge	Externe Tools, spezialisierte Zweit-Modelle	7

1.2 Agent im Vergleich: Xcode-Agent, Claude Code, Codex CLI, Gemini

Xcode implementiert Agenten nicht als eigenes System, sondern bindet die Agenten von Anthropic, OpenAI und (seit Xcode 27) Google direkt ein.

Seit Xcode 26.3 (Februar 2026) gibt es dafür `mcpbridge`, ein von Apple ausgeliefertes Kommandozeilen-Werkzeug, das MCP-Anfragen in Xcodes internes XPC-Protokoll übersetzt. Extern laufende Agenten wie Claude Code oder Cursor können sich darüber mit einer laufenden Xcode-Instanz verbinden und bekommen strukturierten Zugriff auf Diagnosen, Symbolinformationen, SwiftUI-Previews und den Swift-REPL — bestätigt über mehrere unabhängige Quellen.

> ⚠ **Prüfhinweis:** Ob der im Editor eingebaute „Claude Agent"-Panel selbst intern über exakt denselben `mcpbridge`-Mechanismus läuft, oder ob das eine separate Integration über das Claude Agent SDK ist

> und `mcpbridge` nur den umgekehrten Weg (externe Terminal-Agenten → Xcode) abdeckt, liess sich nicht eindeutig klären. Beide Mechanismen existieren nachweislich; wie sie intern zusammenhängen, ist nicht abschliessend belegt. Fachlich unstrittig ist: Der „Claude Agent" in Xcode nutzt laut Anthropic dieselbe Agent-SDK-Basis wie das eigenständige Claude-Code-CLI, inklusive Subagenten und Hintergrund-Tasks.

Claude Code (Anthropic): läuft nativ im Terminal, liest Dateien, führt Bash-Befehle aus, startet Tests, ändert Code. Durchsucht die Codebasis automatisch, koordiniert Änderungen über mehrere Dateien. Stärken: Codequalität, tiefes Debugging, architektonische Änderungen. Kontextfenster: bis zu 1 Million Token (seit März 2026 für Opus/Sonnet-Modelle allgemein verfügbar, bestätigt).

Codex CLI (OpenAI): cloudbasierter Agent (isolierte Sandbox, öffnet am Ende einen Pull Request), lokale CLI, sowie eine IDE-Erweiterung — Letztere nutzt Xcode. Stärken: Geschwindigkeit, Terminal-Aufgaben, Token-Effizienz. Kontextfenster: 272K Token effektiv nutzbar. ⚠ **Prüfhinweis:** Die häufig kolportierte Zahl „bis 1,05 Mio Token im Long-Context-Modus" liess sich für Codex CLI nicht bestätigen — ein grösseres Kontextfenster (bis 1 Mio Token) existiert für das zugrundeliegende GPT-5.5-API-Modell, nicht nachweislich als Codex-CLI-Option.

Gemini (Google): seit Xcode 27 dritter offiziell benannter Agenten-Anbieter, technisch über denselben `mcpbridge`/ACP-Weg eingebunden — bestätigt über mehrere unabhängige Quellen (u. a. 9to5Mac). Anbindung über das Foundation-Models-`LanguageModel`-Protokoll, Auth über Gemini-API-Key oder Enterprise-Agent-Platform.

Direkter Vergleich:

Kriterium	Claude Code / „Claude Agent"	Codex CLI / „Codex"	Gemini
Grundphilosophie	Entwickler bleibt im Loop, Schritt für Schritt	Autonome Delegation, lokal oder Cloud-Sandbox	Multi-Modell-Alternative, Firebase-SDK-Anbindung
Stärke	Codequalität, Architektur, Repo-weite Refactorings	Geschwindigkeit, Terminal-Aufgaben	Analyse, Architektur
Kontextfenster	bis 1 Mio. Token (bestätigt)	272K effektiv (bestätigt), grössere Zahl unbestätigt	nicht recherchiert

> Benchmark-Angaben sind Richtwerte aus Community-Quellen (Stand Frühsommer 2026) und ändern sich mit jedem Modell-Update.

Alternativen ausserhalb von Xcode:

Werkzeug	Charakter	Besonderheit
OpenCode	Open Source, terminalbasiert (Kapitel 8)	75+ Modell-Provider (bestätigt), u. a. Ollama
Cline	Open Source, VS-Code-Erweiterung	Breite Provider-Auswahl inkl. Ollama
Aider	Open Source, terminalbasiert	Git-zentrierter Workflow, Kapitel 8.6
Cursor / Windsurf	Kommerziell, eigenständige IDE	Schnelle Autovervollständigung bzw. eigener Multi-File-Agent
GitHub Copilot	Kommerziell, IDE-Erweiterung	Einziger Anbieter mit brauchbarer Gratis-Stufe

Und wo bleibt Ollama? Weder Claude Code, Codex CLI noch der Gemini-Agent lassen sich auf ein rein lokales Modell umstellen — alle drei sind fest an ihre jeweiligen Cloud-Modelle gebunden. Wer den Agenten-Ansatz vollständig lokal fahren möchte, nutzt **OpenCode** (Kapitel 8) oder Alternativen wie **Cline**/**Aider**.

Gründe für den lokalen Weg: Datenschutz (Code verlässt das Gerät nie), keine API-Kosten, Offline-Fähigkeit, freie Modellwahl, Enterprise-/MDM-Kompatibilität dort, wo Cloud-KI kategorisch gesperrt ist.

Der Preis dafür: geringeres Kontextfenster, spürbar niedrigere Qualität bei komplexen, mehrdateiübergreifenden Aufgaben, direkte Abhängigkeit von der eigenen Hardware. Kapitel 10 bis 12 quantifizieren diesen Trade-off.

Der Grundweg funktioniert bereits mit dem regulär veröffentlichten Xcode 26 (Kapitel 5.1) — nichts davon ist an die Xcode-27-Beta gebunden.

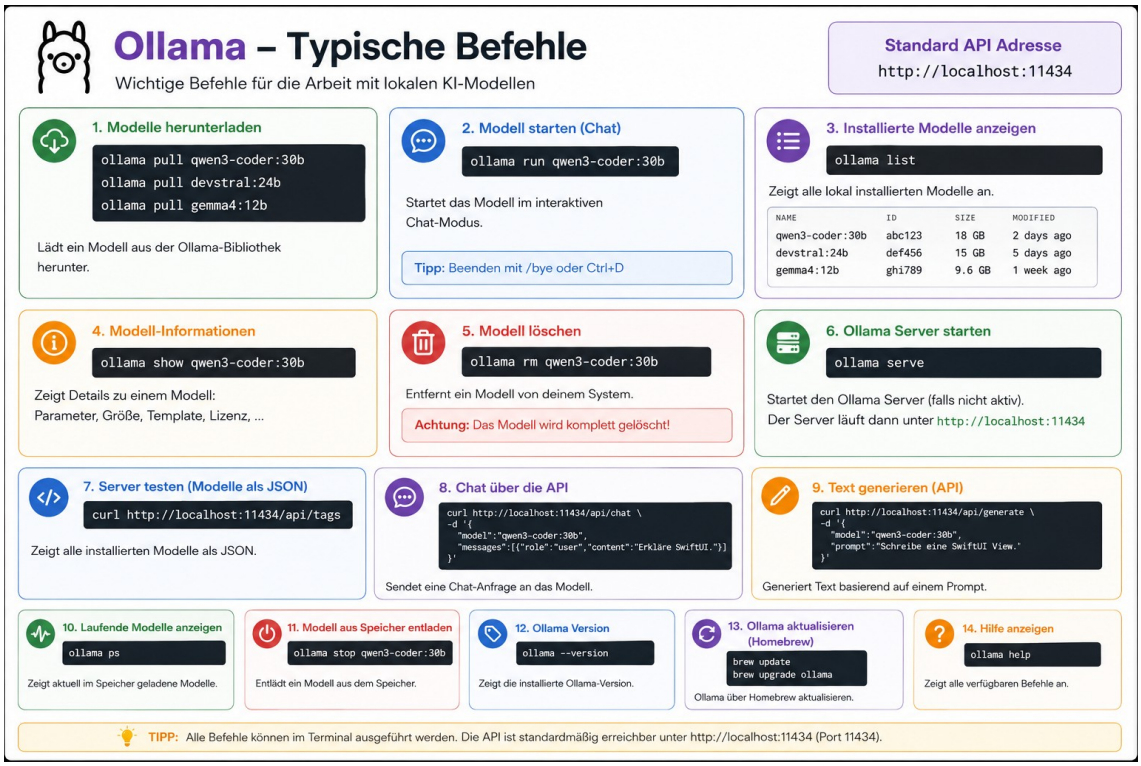
2. Voraussetzungen

Anforderung	Minimum	Empfohlen
Hardware	Apple Silicon M1, 16 GB RAM	M3/M4 Pro oder Max, 32–64 GB RAM
macOS	macOS 27 (Golden Gate)	Aktuellste Beta/Release
Xcode	Xcode 27 Beta 2 oder neuer (Xcode 26 genügt für den Grundweg, Kapitel 5.1)	Aktuellste Beta
Freier Speicher	20 GB (für ein 7B–14B-Modell)	100 GB+ (mehrere Modelle inkl. 30B-Klasse)
Ollama	Version 0.x aktuell	Immer per `ollama --version` prüfen

> **Wichtig:** Das unified memory von Apple Silicon wird zwischen System, Xcode, Simulator/Device Hub und dem lokalen Modell geteilt. Ein 32B-Modell in Q4-Quantisierung belegt real 20–24 GB RAM — auf einem 16-GB-Mac bleibt kaum Luft für Xcode selbst.

3. Schritt 1 — Ollama installieren

Ollama ist ein kostenloses Open-Source-Werkzeug, das LLMs lokal betreibt — ohne Cloud, ohne Account. Es lädt fertige Modelle aus einer öffentlichen Bibliothek, nutzt GPU-beschleunigte Inferenz über Metal und stellt einen lokalen Server (`http://localhost:11434`) mit REST-API bereit. Über diesen Server stellen Terminal, Skripte, Xcode oder OpenCode Anfragen — wie bei einem Cloud-Anbieter, nur bleibt der Datenverkehr auf dem Gerät.



Ollama-Befehle im Überblick

Installation:

```
# per Homebrew (empfohlen für CLI-Workflow)
brew install ollama

# oder Installer-App von ollama.com/download
# (bringt zusätzlich eine Menüleisten-App, die den Dienst automatisch startet)
```

Dienst starten und prüfen:

```
ollama serve
curl http://localhost:11434
# Erwartete Antwort: "Ollama is running"

# dauerhaft im Hintergrund als LaunchAgent
brew services start ollama
brew services restart ollama
brew services stop ollama
```

Prozesskontrolle:

```
pgrep -fl ollama          # laufende Prozesse mit PID
pkill ollama              # harter Reset bei hängendem Dienst
lsof -i :11434            # Port-Belegung prüfen
ollama stop <modell>      # einzelnes Modell aus dem RAM entladen
ollama --version
brew upgrade ollama
```

4. Schritt 2 — Ein Coding-Modell laden

4.1 Modellwahl nach Hardware-Klasse

Richtwerte für Q4-Quantisierung (Stand Frühsommer 2026), schwanken je nach Kontextfenster-Grösse.

RAM-Klasse	Modell	Parameter	RAM-Bedarf (Q4)	Charakteristik
8–16 GB	`qwen3-coder:7b`	7B	~6 GB	Solide Autovervollständigung
16 GB	`qwen3-coder-next` (MoE)	80B (3B aktiv)	~8–10 GB	Starke Leistung trotz kleinem aktivem Footprint (Ollama-Bibliothek bestätigt)
16 GB	`gpt-oss:20b`	20B	~14–16 GB	Guter Allrounder
16–24 GB	`deepseek-coder-v2:16b`	16B (MoE)	~10 GB	Starkes Preis-Leistungs-Verhältnis
16–24 GB	`devstral:24b`	24B	~14–16 GB	Für agentische Multi-File-Workflows trainiert
24 GB+	`codestral:22b`	22B	~13–14 GB	Fill-in-Middle-Spezialist
24–32 GB	`qwen3-coder:14b`	14B	~10 GB	Guter Default-Pick
32 GB+	`qwen3-coder:30b`	30B (MoE)	~20 GB	Bis 256K Kontext, stark bei Multi-File
48 GB+	`qwen2.5-coder:32b`	32B	~24 GB	Reifer Vorgänger
64 GB+	`llama3.3:70b`	70B	~40 GB	Bester lokaler Allrounder, kein reines Coding-Modell
96 GB+	`deepseek-coder-v3` ⚠	236B (21B aktiv)	100 GB+	⚠ Modellname/Benchmark nicht eindeutig verifiziert, siehe Kapitel 11

> **Faustregel:** 16-GB-MacBook → `qwen3-coder:7b` oder `qwen3-coder-next`; 32-GB-Mac → `qwen3-coder:14b`; 64-GB-Mac Studio/MacBook Pro → `qwen3-coder:30b`. Tag-Namen vor dem Pull in der Ollama-Modellbibliothek prüfen.

Praktische Befehle:

```
ollama pull qwen3-coder:14b     # herunterladen
ollama run qwen3-coder:14b      # interaktiv testen
ollama list                     # verfügbare Modelle
```

```
ollama ps                       # geladene Modelle + RAM-Verbrauch
ollama show qwen3-coder:14b     # Details (Kontext, Quantisierung, Lizenz)
ollama rm qwen3-coder:14b       # entfernen
```

4.2 Quantisierung kurz erklärt

Ollama lädt standardmässig **Q4_K_M** (4-Bit-Gewichte, gemischte Präzision) — guter Kompromiss zwischen Qualität und Speicherbedarf.

```
ollama pull qwen3-coder:14b-q8_0   # höhere Qualität, ~doppelter RAM-Bedarf
ollama pull qwen3-coder:14b-q4_K_M # Standard-Kompromiss
```

Als Richtwert: Q8 liegt qualitativ nah am unquantisierten Original, Q4 verliert meist nur 1–3 Prozentpunkte auf Coding-Benchmarks bei etwa halbem Speicherbedarf — für Xcode ist Q4 in der Regel die richtige Wahl.

5. Schritt 3 — Ollama in Xcode einrichten

Xcode besitzt seit Version 26 einen Mechanismus, um selbst gehostete Modelle als gleichwertigen Provider neben Apples On-Device-Modell, PCC, Claude, OpenAI und Gemini einzuhängen. Der Weg ist in Xcode 26 und 27 praktisch identisch — bestätigt über mehrere unabhängige Quellen für Xcode 26.

5.1 Xcode 26

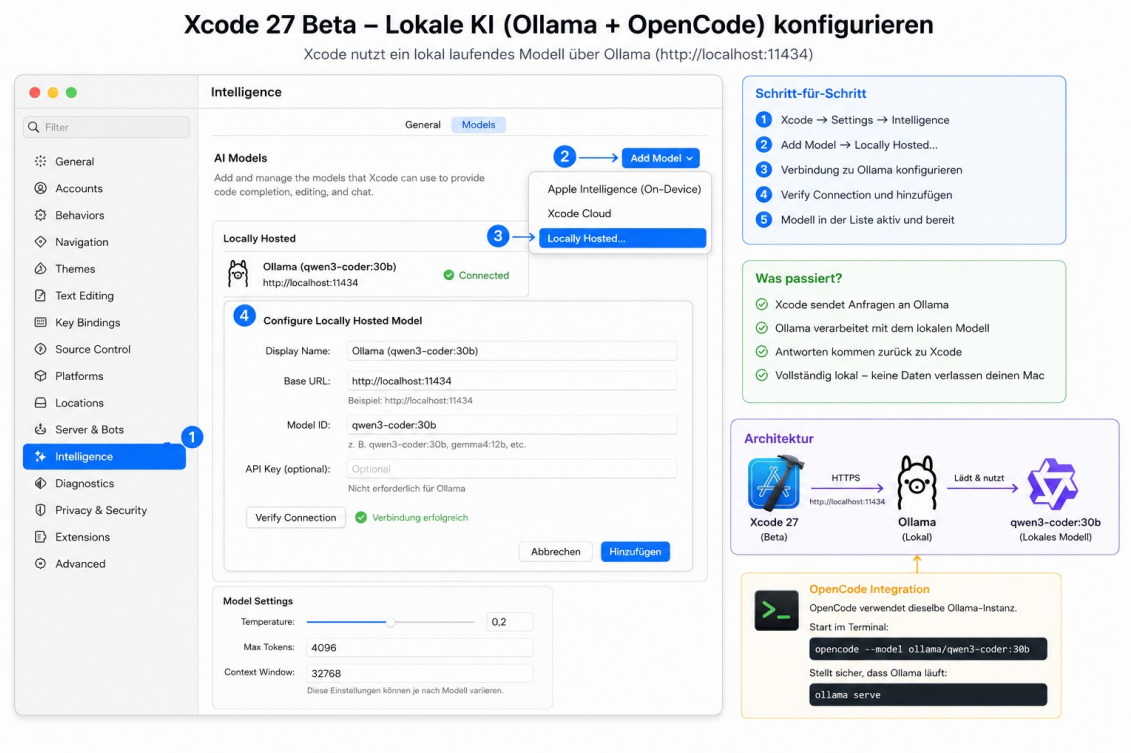
1. Ollama-Dienst starten, mindestens ein Modell geladen (Kapitel 4)
2. Xcode 26 → **Settings → Intelligence**
3. **„Add a Model Provider" → „Locally Hosted"**
4. **Port:** `11434` (Ollama-Standard), **Description:** freier Anzeigename
5. Modell aus der automatisch befüllten Liste auswählen
6. Coding Assistant (⌘+0) öffnen, neuen Provider als aktives Modell wählen

Als Standard-Provider ist ChatGPT vorkonfiguriert, Claude lässt sich direkt einhängen. Ollama kommt als zusätzlicher, gleichberechtigter Provider hinzu.

Meilenstein Xcode 26.3 (Februar 2026, bestätigt): Aus dem bis dahin reaktiven Coding Assistant wurde erstmals ein autonomer Coding Agent — Claude und Codex können seither hochrangige Ziele selbstständig umsetzen, inklusive Build/Test-Ausführung und Preview-Screenshots. Diese Agentenfähigkeit lässt sich auch mit einem lokalen Ollama-Modell kombinieren, sofern es zuverlässiges Tool-Calling unterstützt (Kapitel 4.1).

5.2 Xcode 27

1. Ollama-Dienst starten, Modell per `ollama pull` geladen
2. Xcode 27 → **Settings → Intelligence → „Add Provider" → „Locally Hosted Model"**
3. **Port:** `11434`, **Host:** `localhost` (bei Remote-Server anpassen), **Description:** frei wählbar
4. Modell im Dropdown auswählen — erscheint es nicht sofort, Xcode vollständig neu starten
5. Im Coding-Assistant-Panel (seit Beta 2 im Editor-Bereich) als aktives Modell wählen



Ollama als lokalen Modell-Provider in Xcode 27 einrichten

> ⚠ **Prüfhinweis:** Dass dieser GUI-Weg in Xcode 27 unverändert vom Xcode-26-Weg ist, liess sich nicht abschliessend bestätigen. Eine Quelle beschreibt für Xcode 27 stattdessen einen Zugriff auf lokale Modelle über den neuen ACP-/Plugin-Weg. Beide Beschreibungen können gleichzeitig zutreffen (zwei Wege zum selben Ziel) — vor Einrichtung lohnt ein Blick in die tatsächlich installierte Beta.

Mehrere „Locally Hosted Model"-Einträge lassen sich parallel anlegen — z. B. ein schnelles 7B-Modell für Autovervollständigung und ein 30B-Modell für tiefere Analysen.

5.3 Xcode 26 vs. Xcode 27 — was sich ändert

Merkmal	Xcode 26 (inkl. 26.3)	Xcode 27
Lokaler Modell-Provider	Vorhanden	Unverändert vorhanden (siehe Prüfhinweis oben)
Architektur	Provider-Liste pro Modell	Einheitliches Foundation-Models-Framework, `LanguageModel`-Protokoll für alle Quellen
Coding Assistant	Eigenes Panel	Editor-Bereich, Konversations-Transkript
Agenten-Ergebnisse	Text, teils Datei-Änderungen	Diffs, Pläne, Preview-Snapshots, Markdown-Canvas (bestätigt)
Agenten-Anbieter	Anthropic, OpenAI	Zusätzlich Gemini (Google) — bestätigt
Agent Skills	keine	Apple-eigene Skills, u. a. für Lokalisierung, UIKit-Modernisierung, Accessibility (bestätigt, exakte Skill-Liste variiert je nach Quelle)

Erweiterbarkeit	Kein Plugin-System	Plugin-System (Skills, MCP-Server, ACP-Agenten)
Performance-Messung	Keine dedizierten Werkzeuge	Foundation Models Instrument in Instruments (TTFT, TPS) — bestätigt
`/plan`-Kommando	Nicht vorhanden	Vorhanden — bestätigt
Hardware	Intel + Apple Silicon	Nur noch Apple Silicon — bestätigt

Wer heute schon mit Xcode 26 und Ollama arbeitet, muss beim Umstieg die Provider-Konfiguration nicht neu einrichten.

6. Schritt 4 (fortgeschritten) — Programmatische Einbindung

Für Automatisierung, Tests oder eigene Tooling-Skripte lässt sich Ollama direkt über das **Foundation Models Framework** anbinden (`LanguageModel`-Protokoll). Ollama bietet dafür einen OpenAI-kompatiblen Endpoint unter `/v1`.

```
import FoundationModels
import Foundation

/// Minimaler LanguageModel-Wrapper für einen lokal laufenden Ollama-Server.
/// Nutzt Ollamas OpenAI-kompatiblen Endpoint (`/v1/chat/completions`).
struct OllamaLanguageModel: LanguageModel {
    let modelName: String
    let baseURL: URL = URL(string: "http://localhost:11434/v1")!

    func respond(to prompt: String) async throws -> LanguageModelResponse {
        var request = URLRequest(url: baseURL.appending(path:
"chat/completions"))
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.httpBody = try JSONEncoder().encode(
            OllamaChatRequest(model: modelName, messages: [.init(role: "user",
content: prompt)])
        )
        let (data, _) = try await URLSession.shared.data(for: request)
        let decoded = try JSONDecoder().decode(OllamaChatResponse.self, from:
data)
        return LanguageModelResponse(content:
decoded.choices.first?.message.content ?? "")
    }
}

let localModel = OllamaLanguageModel(modelName: "qwen3-coder:14b")
let session = LanguageModelSession(model: localModel)
let response = try await session.respond(to: "Erkläre diesen Swift-Concurrency-
Code")
print(response.content)
```

6.1 Unterschied zu Kapitel 5

Kapitel 5 bindet Ollama als Provider für den Coding Assistant ein — das hilft beim Schreiben von Code für das eigene Projekt. Kapitel 6 baut ein eigenes Werkzeug, das selbst ein Sprachmodell **aufruft** — damit bekommt die eigene App oder der eigene Build-Prozess eine KI-Fähigkeit, nicht Xcode. Weil derselbe `LanguageModel`-Typ auch für On-Device, PCC und Cloud gilt, bleibt der übrige App-Code unabhängig davon, welches Modell am Ende antwortet — man entwickelt kostenlos gegen ein lokales Modell und tauscht vor dem Release nur die eine Zeile `LanguageModelSession(model:)` aus.

Einsatzszenario	Praktischer Nutzen

Austauschbare KI-Funktion in der eigenen App (z. B. „Notiz zusammenfassen")	Kostenloses, unbegrenztes Testen während der Entwicklung; vor Release Austausch gegen `SystemLanguageModel()` oder Cloud-Backend, Rest der App bleibt unverändert
CI-Skript für automatisierte Codeanalyse (fehlende Doku, hartcodierte Strings, Secrets)	Läuft offline und kostenlos in CI, ohne API-Key-Verwaltung — sinnvoll bei jedem Commit

7. Schritt 5 (fortgeschritten) — Ollama als MCP-Werkzeug

Statt Ollama als direkten Modell-Provider einzuhängen, lässt es sich auch als **MCP-Server** bereitstellen — dann steuert weiterhin ein anderes Modell (Apples On-Device-Modell, PCC, oder ein Cloud-Agent) den Agenten-Workflow, kann aber gezielt Anfragen an das lokale Modell delegieren.

```
// .xcode/mcp-config.json
{
  "servers": [
    {
      "name": "ollama-local-analysis",
      "command": "npx",
      "args": ["-y", "mcp-server-ollama"],
      "env": {
        "OLLAMA_HOST": "http://localhost:11434",
        "OLLAMA_MODEL": "qwen3-coder:14b"
      }
    }
  ]
}
```

Nach dem Einrichten erscheint der Server unter **Settings → Intelligence → Plugins** zur Freigabe — Xcode verlangt explizite Nutzerbestätigung, granulare Berechtigungen und führt ein Audit-Log aller Zugriffe.

7.1 Unterschied zu Kapitel 6

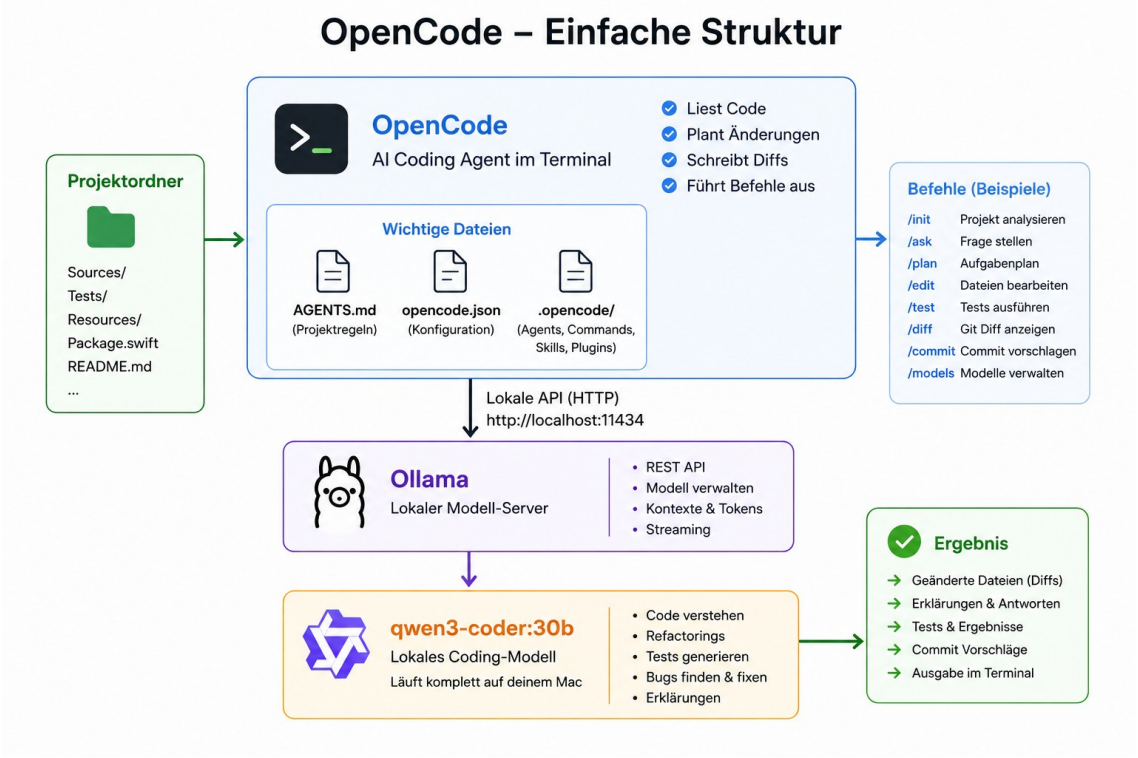
Kapitel 6 ruft Ollama direkt aus eigenem Code auf — man entscheidet im Code, wann das lokale Modell arbeitet. Kapitel 7 dreht das um: Ein **stärkeres** Modell bleibt der „Kopf" des Workflows und delegiert nur bestimmte, sensible Teilaufgaben lokal.

Einsatzszenario	Praktischer Nutzen
Compliance-pflichtige Einzeldatei in einer sonst cloud-gestützten App (z. B. Healthcare-Berechnung)	Hauptagent arbeitet in der Cloud, delegiert nur diese eine Datei an den lokalen MCP-Server — Vertragspflicht bleibt gewahrt
Lokaler Secret-Scanner als Vor-Commit-Prüfung	Dateiinhalt geht für diese Prüfung nie an einen Cloud-Anbieter

8. Alternative: OpenCode und Aider

8.1 OpenCode

OpenCode (`opencode.ai`) ist ein quelloffener, terminalbasierter KI-Coding-Agent — vergleichbar mit Claude Code, aber anbieterunabhängig: über 75 Modell-Provider werden unterstützt (bestätigt), darunter Ollama, GitHub-Copilot-Abos und klassische API-Keys. OpenCode liest den Projektordner, bearbeitet Dateien direkt auf der Festplatte, führt Shell-Befehle aus — komplett unabhängig von Xcode. Xcode bemerkt Dateiänderungen automatisch und lädt sie neu.



OpenCode im Überblick

Warum neben Xcodes eigenem Agenten interessant: freie Modellwahl unabhängig von der Xcode-Provider-Konfiguration, skriptbar für CI/Batch-Aufgaben, funktioniert auch für Nicht-Xcode-Teile eines Projekts (Server, Build-Skripte), volle Terminal-Historie.

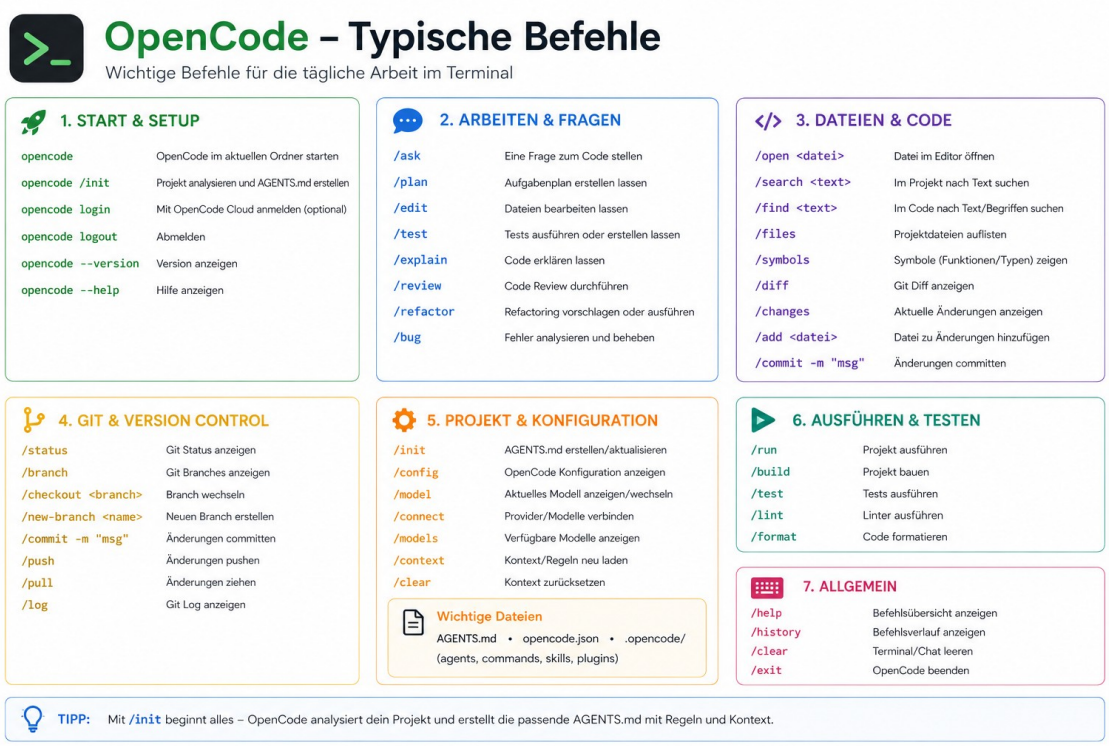
Installation und Einrichtung:

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

# Schnellstart mit Ollama (bestätigt, ollama.com/blog/launch) — koppelt beides
automatisch
ollama launch opencode
# ab Ollama v0.15+ mit direkter Modellwahl:
# ollama launch opencode --model qwen3-coder
```

Ollama setzt den Kontext standardmässig auf nur 4.096 Token, OpenCode braucht für agentische Aufgaben mindestens 16.000, empfohlen 64.000+:

```
ollama run qwen3-coder:14b
>>> /set parameter num_ctx 32768
>>> /save qwen3-coder-32k

# oder dauerhaft per Modelfile
cat <<'EOF' > Modelfile
FROM qwen3-coder:14b
PARAMETER num_ctx 32768
EOF
ollama create qwen3-coder-32k -f Modelfile
```

```
cd /pfad/zum/xcode-projekt
opencode
```



OpenCode-Befehle im Überblick

Braucht man ein Zwischentool? Für den Standardweg nein — Ollama liefert bereits einen OpenAI-kompatiblen Endpoint, den OpenCode direkt anspricht. Sinnvoll wird ein Proxy nur in Sonderfällen: mehrere Backends (Ollama + Cloud) hinter einer Adresse bündeln (LiteLLM), unzuverlässiges Tool-Calling normalisieren, oder ein zentraler Team-Server mit Auth/Rate-Limiting im Firmennetz.

Ausblick — OpenCode direkt im Editor über ACP: OpenCode unterstützt zusätzlich das **Agent Client Protocol** (`opencode acp`, ein offener, von Zed initiierter Standard) und lässt sich dadurch bei Zed, JetBrains-IDEs und Neovim direkt als natives Agenten-Panel einbinden. Xcode 27 hat ebenfalls ein „Agent Client Protocol" — aber laut Apples eigener Beschreibung als **Orchestrierungsschicht innerhalb von Xcode** (ein Agent delegiert Datei-Operationen an einen anderen, Kontext bleibt synchron), nicht als offener Standard für externe Editoren. ⚠ **Prüfhinweis:** Es handelt sich vermutlich um zwei gleichnamige,

aber unterschiedliche Protokolle — eine offizielle Auflistung von Xcode als ACP-Host für OpenCode existiert nicht. Der Terminal-Begleiter-Workflow oben bleibt der praxiserprobte Weg.

8.2 Aider als Alternative

Aider (`aider.chat`) ist ebenfalls quelloffen und terminalbasiert, arbeitet aber bewusst **git-zentriert** und minimalistisch statt auf Provider-Vielfalt ausgelegt.

```
python -m pip install aider-install
aider-install
export OLLAMA_API_BASE=http://127.0.0.1:11434
```

Ollama setzt das Kontextfenster standardmässig auf nur 2.048 Token und verwirft alles darüber **stillschweigend** — ein häufiger, schwer diagnostizierbarer Fehler bei mehrdateiübergreifenden Änderungen. Server mit erhöhtem Kontext starten:

```
OLLAMA_CONTEXT_LENGTH=32768 ollama serve
cd /pfad/zum/xcode-projekt
aider --model ollama_chat/qwen3-coder:14b   # ollama_chat/-Präfix bevorzugen,
nicht ollama/
```

Dauerhaft fest in `.aider.model.settings.yml`:

```
- name: ollama_chat/qwen3-coder:14b
  extra_params:
    num_ctx: 32768
```

Workflow: Aider startet gezielt mit einzelnen Dateien (`aider SpotRepository.swift`) oder Dateien werden per `/add` ergänzt — zu viele Dateien auf einmal überfordern kleinere lokale Modelle. Jede Änderung wird automatisch als eigener Git-Commit abgelegt, per `/undo` gezielt rückgängig machbar.

Watch-Mode — Auslösung direkt aus Xcode heraus, ohne Terminal-Fokuswechsel: `aider --watch-files` im Hintergrund, dann im Editor:

```
// Validierung ergänzen: Name darf nicht leer sein. AI!
```

Beim Speichern erkennt Aider den Kommentar und setzt ihn um; `AI?` stellt nur eine Frage. Funktioniert mit jedem Editor, der Dateien auf die Festplatte schreibt.

Kriterium	OpenCode	Aider
Philosophie	Breiter Agent, 75+ Provider, hohe Autonomie	Minimalistisch, git-zentriert
Sicherheitsnetz	Diff-Review vor Übernahme	Automatischer Commit pro Änderung, granulares `/undo`
Editor-Trigger ohne Terminal	Nicht vorgesehen	Watch-Mode über `AI!`-Kommentare
Ollama-Kontext-Standard	4.096 Token	2.048 Token, **stillschweigend** abgeschnitten

ACP-Editor-Integration	Ja (Zed/JetBrains/Neovim)	Nein, rein dateibasiert
Am besten für	Grössere, mehrdateiübergreifende Durchläufe	Gezielte Einzeldatei-Änderungen mit maximaler Nachvollziehbarkeit

Für Xcode-Alltag eher ein Ergänzungswerkzeug als ein Ersatz: Aider für gezielte Bugfixes an genau einer Funktion und risikoarmes Experimentieren (jeder Schritt einzeln zurückrollbar), OpenCode für grosse, mehrdateiübergreifende agentische Refactorings.

9. Im Alltag arbeiten

Aufgabe	Empfohlener Weg mit lokalem Modell
Inline-Autovervollständigung	Kleines, schnelles Modell (7B–14B) — Latenz wichtiger als Qualität
„Explain this code"	Beliebiges 7B+-Modell reicht meist
Einzelne Funktion refaktorieren	14B–30B-Modell, Kontext auf die betroffene Datei begrenzen
Bug in einer Datei finden	Reasoning-starkes Modell (DeepSeek-R1-Klasse) — sichtbares Chain-of-Thought hilft
`/plan`-Kommando für neues Feature	Eher Cloud/PCC, sofern Datenschutz es zulässt — lokale Modelle liefern spürbar schwächere Pläne (Kapitel 11)
Mehrdateien-Refactoring über den ganzen Agenten-Workflow	Nur mit 30B-Klasse oder grösser, Ergebnis immer per Diff-Review gegenprüfen — alternativ OpenCode (Kapitel 8)

10. Performance

Ein wichtiger Unterschied zu Apples On-Device-Modell: Dieses läuft auf der **Neural Engine**, optimiert auf minimale Latenz und Energieeffizienz. Ollama (wie LM Studio, llama.cpp) nutzt die **GPU über Metal** und den Unified-Memory-Bus — leistungsfähiger bei grösseren Modellen, aber energiehungriger, mit spürbarem Aufheizen bei längeren Sessions auf MacBooks.

Tokens/Sekunde nach Hardware-Klasse (Richtwerte, keine Herstellerangaben):

Modellgrösse	M1/M2 (16 GB)	M2/M3 Pro/Max (32–48 GB)	M4 Max (64–128 GB)	M-Ultra (128 GB+)
7B (Q4)	20–35 Tok/s	35–55 Tok/s	50–70 Tok/s	60–80 Tok/s
14B (Q4)	10–18 Tok/s	20–35 Tok/s	35–50 Tok/s	45–60 Tok/s
24B–32B (Q4)	kaum praktikabel	10–15 Tok/s	20–35 Tok/s	30–45 Tok/s
70B (Q4)	nicht praktikabel	8–12 Tok/s (48 GB+)	25–35 Tok/s	35–50 Tok/s
MoE (z. B. Qwen3-Coder-Next)	30–45 Tok/s	45–60 Tok/s	55–75 Tok/s	65–85 Tok/s

Reale Werte schwanken mit Kontextlänge, Hintergrundlast (Xcode-Indexierung, Simulator) und Thermik im Akkubetrieb.

RAM-Bedarf nach Modellgrösse/Quantisierung (Richtwerte):

Parameter	Q4_K_M	Q8_0	Zusätzlich pro 32K Kontext
7B	~5,5 GB	~9 GB	+1–2 GB
14B	~10 GB	~16 GB	+2–3 GB
22B–24B	~14 GB	~24 GB	+3–4 GB
30B–32B	~20 GB	~34 GB	+4–6 GB
70B	~40 GB	~70 GB	+6–8 GB

Kontextfenster: Apples On-Device-Modell arbeitet mit 8.192 Token, PCC mit 32.768 Token (bestätigt), Qwen3-Coder:30b theoretisch bis 256K — praktisch limitiert das verfügbare RAM den nutzbaren Kontext früher, da der KV-Cache linear mitwächst. Für Xcode heisst das: lieber kleineres Kontextfenster mit höherer Modellqualität als ein riesiges Fenster mit einem zu kleinen Modell.

Messen mit dem Foundation Models Instrument (bestätigt): Xcode 27 bringt in Instruments ein eigenes Template, das TTFT (Time-to-First-Token) und Tokens/Sekunde direkt aufzeichnet — unabhängig vom Provider.

```
Xcode → Product → Profile → Foundation Models
→ Coding-Assistant-Anfrage auslösen
→ TTFT, Tokens/Sek und Reasoning-Token-Anteil im Track ablesen
```

Der **Organizer** liefert ergänzend die bekannten Crash- und Energie-Berichte aus TestFlight/App Store — auch für Agent-generierten Code, unabhängig vom Provider.

11. Modellvergleich: Coding, Analyse, Bugfix, Refactoring

Alle Werte sind Richtwerte aus öffentlichen Community-Ranglisten (SWE-bench Verified / HumanEval, Stand Frühsommer 2026) und zeigen die relative Positionierung, keine exakten Prozentzahlen. ⚠ Die folgenden Zahlen wurden nicht gegen Primärquellen (offizielle Leaderboards) geprüft.

Modell	Programmierung	Bugfix	Refactoring (Multi-File)	Besonderheit
Qwen3-Coder 7B	gut (~70 %)	mittel	schwach	bestes Preis-/Speicher-Verhältnis für 8 GB
Qwen3-Coder-Next (MoE)	sehr gut (~65–70 %)	gut	mittel	3B aktive Parameter, läuft stark auf 16 GB
Qwen3-Coder 14B	sehr gut (~82 %)	gut	mittel	solider Allrounder-Default
Codestral 22B	gut (~78 %), Fill-in-Middle	mittel	schwach	weniger für „Chat"-Aufgaben
DeepSeek-R1-Klasse (14B)	mittel (~40–44 %)	sehr gut	mittel	sichtbares Denken hilft bei Fehlersuche
Devstral 24B	gut (~47 % SWE-bench Verified)	gut	gut	für agentische Multi-File-Workflows trainiert
Qwen2.5-Coder 32B	sehr gut (~49–50 %)	gut	mittel–gut	reifer, breit getesteter Klassiker
Qwen3-Coder 30B (MoE)	sehr gut (~87 %)	sehr gut	gut	bester Kompromiss Konsumenten-Hardware/Multi-File
Llama 3.3 70B	mittel (~48 %)	mittel	mittel	eher Generalist als Coding-Spezialist
„DeepSeek-Coder-V3" ⚠	~91 % (Quelle unklar)	sehr gut	sehr gut	⚠ Modellname nicht eindeutig — evtl. Verwechslung mit DeepSeek-V3 (236B) bzw. DeepSeek-Coder-V2, dessen offizielle Werte (~90 % HumanEval) ähnlich, aber nicht identisch sind
Referenz: Claude (Cloud)	sehr stark	sehr stark	⚠ „stark überlegen" — die zitierte Zahl ~82 % vs. ~28–35 % lokaler 70B-Modelle liess sich nicht verifizieren, als	grösster Qualitätsabstand bei komplexen, mehrdateiübergreifenden Aufgaben

			illustrative Schätzung behandeln	

Kernaussagen: Bei einfachen, klar umrissenen Aufgaben erreichen lokale 14B–30B-Modelle einen Grossteil der Cloud-Qualität — für den Alltag oft ausreichend. Bei Bugfixing liefern Reasoning-Modelle (DeepSeek-R1-Klasse) durch sichtbares Chain-of-Thought oft treffsicherere Ergebnisse. Bei mehrdateiübergreifendem Refactoring bleibt die Lücke zu Cloud-Modellen am grössten — lokale Modelle unter 30B nur mit engem Review einsetzen. **Devstral** und **Qwen3-Coder 30B** gelten aktuell als die praxistauglichsten Kandidaten für autonome Agent-Workflows auf Konsumenten-Hardware.

12. Vergleich: On-Device vs. Ollama vs. PCC vs. Cloud

Kriterium	On-Device	Ollama (lokal)	PCC	Cloud (Claude/OpenAI/Gemini)
Datenschutz	Maximal — nie Netzwerk	Maximal — nie Netzwerk	Sehr hoch — On-Device-Garantien in der Cloud	Standard — Anbieter-AGB gilt
Kosten	Kostenlos	Kostenlos (nur Strom)	Kostenlos für Small-Business-Programm, sonst kontingentiert	API-Kosten pro Token
Kontextfenster	8.192 Token	modellabhängig, bis 256K theoretisch	32.768 Token	meist 200K+ Token
Modellqualität	Für Apple-SDKs abgestimmt, sonst begrenzt	Frei wählbar, 7B bis 236B	Höher als On-Device	Höchste verfügbare Qualität
Geschwindigkeit	Sehr schnell (Neural Engine)	Stark hardwareabhängig	Netzwerkabhängig, optimiert	Netzwerkabhängig
Offline-fähig	Ja	Ja	Nein	Nein
Enterprise/MDM	Immer erlaubt	Meist erlaubt (kein Netzwerkausgang)	Konfigurierbar	Oft per Policy gesperrt

Faustregel: On-Device für triviale, latenzkritische Vervollständigung; Ollama für alles, was privat bleiben muss und mehr Qualität als On-Device braucht; PCC als Mittelweg; Cloud-Modelle für die anspruchsvollsten Aufgaben, wenn Datenschutzrichtlinien es erlauben.

13. Vergleichssysteme: Ollama vs. LM Studio vs. llama.cpp vs. MLX

System	Typ	Xcode-Integration	Stärken	Schwächen
Ollama	Server + CLI, REST-API	Nativ als „Locally Hosted Model"-Provider, Basis für OpenCode	Grösste Modellbibliothek, einfachstes Setup	Wenig Feintuning über CLI hinaus
LM Studio	GUI-App + lokaler Server	Über OpenAI-kompatiblen Endpoint anbindbar	Komfortable GUI, MLX-Unterstützung	Keine offizielle CLI-Automatisierung
llama.cpp (roh)	C++-Inferenz-Engine	Nur über selbst gebauten Wrapper	Maximale Kontrolle, kleinster Overhead	Kompilieren, Setup in Eigenregie
MLX (Apple)	Array-Framework + `mlx-lm`	Kein offizieller Xcode-Provider, nur programmatisch (Kapitel 6)	Für Apple-Silicon optimiert, oft beste Perf/Watt	Kleineres Ökosystem, mehr Handarbeit

Empfehlung: Für den Einstieg ist **Ollama** die pragmatischste Wahl. **LM Studio** lohnt sich bei Präferenz für grafische Modellverwaltung oder MLX-Modelle. **MLX direkt** für Performance-Feinschliff auf einem bestimmten Mac. **llama.cpp roh** primär für eigene Tool-Entwicklung.

14. Pro und Contra der lokalen KI-Strategie

Pro: Vollständige Datenkontrolle, keine laufenden Kosten, volle Offline-Fähigkeit, freie Modellwahl (`ollama pull`/`ollama rm`), erfüllt strenge Enterprise-/Compliance-Vorgaben, reproduzierbare Ergebnisse ohne stille Anbieter-Updates, zwei gleichwertige Zugangswege (nativ in Xcode oder terminalbasiert über OpenCode), funktioniert bereits mit dem regulär veröffentlichten Xcode 26.

Contra: deutlich geringere Qualität bei komplexen, mehrdateiübergreifenden Aufgaben, hoher RAM-Bedarf, zusätzlicher Energieverbrauch/Wärme bei längeren Sessions, Wartungsaufwand für Modelle/Version/Konfiguration, kein Support-Kanal wie bei Cloud-Anbietern, kleineres Kontextfenster in der Praxis sobald RAM-Grenzen greifen.

15. Troubleshooting

Problem	Ursache	Lösung
Modell erscheint nicht im Xcode-Dropdown	Xcode cacht Provider-Liste beim Start	Xcode vollständig beenden und neu starten
Verbindung zu `localhost:11434` schlägt fehl	Ollama-Dienst läuft nicht	`ollama serve` bzw. `brew services start ollama`, mit `curl` prüfen
Sehr langsame Antworten	Modell zu gross für RAM, System swapt	Kleineres/stärker quantisiertes Modell, `ollama ps` zur RAM-Kontrolle
Mac wird heiss, Lüfter laut	GPU-Last durch Inferenz auf Akku	Bei längeren Sessions am Netzteil arbeiten
Antwort da, aber Qualität schlecht	Zu aggressive Quantisierung/zu kleines Modell	Q8-Variante oder grösseres Modell testen, ggf. auf PCC/Cloud auslagern
Mehrere Modelle blockieren sich im RAM	Ollama hält zuletzt genutzte Modelle im Speicher	`ollama ps` prüfen, mit `ollama stop <modell>` entladen
Speicherplatz läuft voll	Mehrere grosse Modelle gleichzeitig geladen	`ollama list` prüfen, mit `ollama rm` löschen
OpenCode bricht bei Datei-/Bash-Aktionen ab	Ollama-Standardkontext (4.096 Token) zu klein	Kontext auf mindestens 16K, besser 32K–64K erhöhen (Kapitel 8.1)
Neues Modell erscheint nicht im OpenCode-Picker	Nur in `opencode.json` eingetragen, nicht wirklich gepullt	Erst `ollama pull <modell>`, dann OpenCode neu starten

16. Empfehlung & Zusammenfassung

Für den Alltag empfiehlt sich ein hybrider Ansatz: ein kleines, schnelles lokales Modell (7B–14B-Klasse, z. B. `qwen3-coder:14b`) als Standard-Provider für Autovervollständigung, Erklärungen und einfache Bugfixes. Für anspruchsvolle Aufgaben wie mehrdateiübergreifendes Refactoring oder komplexe Architekturplanung (`/plan`) lohnt sich der bewusste Wechsel auf PCC oder ein Cloud-Modell, sofern Datenschutzvorgaben es zulassen. Wer strikt lokal bleiben muss, investiert in möglichst viel RAM (64 GB+) und setzt auf MoE-Modelle wie `qwen3-coder:30b`.

Die Einrichtung ist mit Ollama in unter zehn Minuten erledigt: installieren, ein Modell laden, in **Xcode → Settings → Intelligence → Add Provider → Locally Hosted Model** mit Port 11434 verbinden — bereits mit dem heute verfügbaren Xcode 26 (Kapitel 5.1). Für terminalgestützte, mehrdateiübergreifende Aufgaben ergänzt **OpenCode** (Kapitel 8) denselben lokalen Ollama-Server um einen zweiten Zugangsweg.

17. Die komplette Kette auf einen Blick

Die komplette Kette: Was kann ich mit lokaler KI machen?

```
Apple-Silicon-Mac (unified memory)
        │
        ▼
   Ollama-Server  (Modell-Engine, lokal, Port 11434)
        │
        ├── GUI-Provider in Xcode 26/27 (Kapitel 5)
        ├── Foundation-Models-Framework, programmatisch (Kapitel 6)
        ├── MCP-Werkzeug innerhalb eines Agenten-Workflows (Kapitel 7)
        └── OpenCode / Aider, eigenständiger Terminal-Agent (Kapitel 8)
                │
                ▼
   Aufgabe: Vervollständigung · Erklärung · Bugfix · Refactoring ·
Automatisierung
                │
                ▼
   Ergebnis im Xcode-Projekt (Diff, Preview, Testlauf, Datei-Änderung)
```

Alle vier Zugangswege greifen auf denselben Ollama-Server zurück — die Wahl richtet sich nach der Aufgabe. Unabhängig vom gewählten Weg landet das Ergebnis immer an derselben Stelle: im Xcode-Projekt auf der Festplatte, egal ob über Xcode selbst als Diff/Preview/Testlauf sichtbar oder über OpenCode/Aider als direkte Dateiänderung, die Xcode automatisch übernimmt.

18. Apple-Fahrplan: bestätigt vs. erwartet

Apple veröffentlicht keine offizielle Produkt-Roadmap für kommende Xcode-Versionen — alles unten Stehende stützt sich entweder auf bereits ausgelieferte Beta-Versionen (Fakten) oder auf den bekannten, wiederkehrenden Beta-Rhythmus früherer Jahre (Muster, keine Ankündigung). Es werden bewusst keine Gerüchte oder Leaks zu Funktionen wiedergegeben, die Apple nicht selbst bestätigt hat.

Was bereits bestätigt und ausgeliefert ist (Stand Beta 3, Juli 2026):

- Beta 1 (09.06.2026, WWDC26): Coding Agents (Claude, Codex, seit Beta 1 bereits inkl. Gemini als Multi-Provider), Device Hub, einheitliches Foundation-Models-Framework, Plugin-System, MCP nativ, `/plan`-Kommando, Markdown-Canvas, Apple-eigene Agent Skills (u. a. Lokalisierung, UIKit-Modernisierung), Foundation Models Instrument, Apple Silicon-only
- Beta 2 (22.06.2026): TestFlight-Unterstützung für mit Beta 2 gebaute Apps, weitere Detailkorrekturen an den MCP-Werkzeugen
- Beta 3 (06.07.2026): dritte reguläre Beta im gewohnten rund zweiwöchigen Rhythmus, laut Apples Release Notes primär Fehlerkorrekturen und API-Verfeinerungen

Was nach bisherigem Muster noch zu erwarten ist (kein Leak, sondern der übliche Ablauf früherer Beta-Zyklen): In der Regel folgen bis zum Herbst weitere Betas, eine Release Candidate und die finale Version parallel zum jährlichen iOS-Release im September. Erfahrungsgemäss werden in diesem Zeitraum vor allem Stabilität, Performance und Detailverhalten der bereits angekündigten Funktionen nachgeschärft — grosse, bisher unangekündigte Funktionen sind zu diesem späten Zeitpunkt im Zyklus untypisch, aber nicht ausgeschlossen.

Offene Fragen, die sich derzeit nicht zuverlässig beantworten lassen: ob sich die genaue GUI-Führung für lokale Modelle („Locally Hosted Model" vs. ACP-Weg) bis zur finalen Version noch ändert; wie stabil die Skill-Liste der Apple-eigenen Agenten bleibt (verschiedene Quellen nennen unterschiedliche Skill-Namen, was für eine noch nicht endgültig fixierte Beta-Funktion spricht); ob `mcpbridge` und die native Editor-Integration des Claude-Agenten dauerhaft getrennte Mechanismen bleiben oder zusammengeführt werden.

Für belastbare Aussagen zur finalen Xcode-27-Version empfiehlt sich vor einer Grundsatzentscheidung immer ein Blick in die aktuell installierte Beta sowie in Apples offizielle Release Notes.

19. Quellenverzeichnis

- Xcode 27 Beta Release Notes — Apple Developer Documentation
- Apple Newsroom — Apple aids app development with new intelligence frameworks and advanced tools (Juni 2026)
- Xcode 26.3 unlocks the power of agentic coding — Apple Newsroom (Februar 2026)
- Apple's Xcode now supports the Claude Agent SDK — Anthropic
- Setting up coding intelligence — Apple Developer Documentation
- Use Custom Models in the New Xcode 26 Intelligence — Wendy Liga
- Ollama OpenAI Compatibility — offizielle Dokumentation
- OpenCode — Ollama-Integration, offizielle Dokumentation
- OpenCode — ACP-Unterstützung, offizielle Dokumentation
- Aider — Ollama-Integration, offizielle Dokumentation
- Aider — Watch-Mode, offizielle Dokumentation
- Ergänzend: interne Referenzdokumentation `Tutorial-iOS27/XCode/xcode/xcode27.md` (streng quellenbasiert nach Apple-Vorgaben, deckt Beta 1 ab)

Ergänzende Quellen aus dem Faktencheck (Juli 2026):

- Xcode 27 expands agentic coding toolset with Gemini integration — 9to5Mac
- Apple rolls out the third iOS 27, macOS 27 developer betas — AppleInsider
- Xcode 27 and Codex CLI: Connecting Apple's MCP Bridge — Codex Knowledge Base
- Xcode 27 Agentic Coding: MCP, On-Device AI, Agent Skills Explained — byteiota
- Claude's 1 Million Context Window — Übersicht 2026
- Increase effective context window 272000 → Codex Issue #9429 — GitHub
- OpenCode Review: Go CLI Terminal Coding Agent With 75+ Models — OpenAIToolsHub
- qwen3-coder-next — Ollama-Modellbibliothek

> **Hinweis:** Community-Ranglisten und Blog-Benchmarks (Kapitel 10–11) sind keine offiziellen Herstellerangaben und ändern sich mit jeder neuen Modellgeneration. Vor einer Grundsatzentscheidung empfiehlt sich immer ein eigener Test mit dem Foundation Models Instrument (Kapitel 10) auf der tatsächlich genutzten Hardware.