LLM Hosting in 2026: Local, Self-Hosted und Cloud-Infrastruktur Vergleich

LLM Hosting — Übersicht und Entscheidungshilfe

Large Language Models laufen längst nicht mehr nur in der Cloud. 2026 ist die Frage nicht mehr "Kann ich ein LLM selbst betreiben?" — sondern:

> Welche Hosting-Strategie passt zu meinem Workload, meinem Budget und meinen Anforderungen an Kontrolle und Datenschutz?

Diese Seite ist meine persönliche Referenz. Ich dokumentiere hier, was ich tatsächlich betreibe und getestet habe — keine vollständige Marktübersicht, sondern ehrliche Erfahrungswerte aus zwei verschiedenen Kontexten: dem Arbeitsumfeld (Kubernetes-Cluster, Team-Betrieb) und dem Home-Lab (Workstation, Einzelnutzer).

Was LLM Hosting eigentlich bedeutet

LLM Hosting beschreibt, wie und wo Modelle für Inferenz betrieben werden. Diese Entscheidung hat direkte Auswirkungen auf:

- Latenz und Time-to-First-Token
- Durchsatz und Parallelität
- Kosten pro Request

- Datenschutz und Datenhoheit
- Infrastrukturkomplexität
- Operativen Aufwand

LLM Hosting ist keine reine Installationsaufgabe — es ist eine Infrastrukturentscheidung.

Entscheidungsmatrix

Ansatz	Geeignet für	Hardware	Produktionstauglich	Kontrolle
llama.cpp	GGUF-Modelle, CLI + Server, Offline	CPU / GPU	Ja (llama-server)	Sehr hoch
vLLM	Hochdurchsatz-Produktion	Dedizierter GPU-Server	Ja	Hoch
Jan.ai	Lokale Experimente, Einzelnutzer	Consumer GPU / CPU	Begrenzt	Hoch
LM Studio	Lokale Experimente, Parameter-Tuning	Consumer GPU / CPU	Begrenzt	Hoch
LiteLLM	Proxy-Layer, Multi-Backend, API-Unified	Keiner (nur Proxy)	Ja	Hoch
Open WebUI	Team-Frontend für lokale Modelle	Keiner (nur Frontend)	Ja	Hoch

Nur noch selten in Benutzung

Ansatz	Geeignet für	Hardware	Produktionstauglich	Kontrolle

Ollama	Schneller Einstieg, minimale Konfiguration	Consumer GPU / CPU	Eingeschränkt	Hoch
Cloud-Anbieter	Zero-Ops, Skalierung ohne Hardware	Keiner (remote)	Ja	Gering

---

Lokales LLM Hosting

Lokales Hosting bringt:

- Vollständige Kontrolle über Modelle und Parameter
- Keine tokenbasierte Abrechnung
- Vorhersehbare Latenz
- Datenschutz ohne Kompromisse

Die Kehrseite: Hardware-Grenzen, Wartungsaufwand und manuelle Skalierung.

---

llama.cpp

llama.cpp ist mein primäres Inference-Backend — sowohl auf der Arbeit als auch zu Hause. Es ist eine schlanke C/C++-Engine für GGUF-Modelle mit sehr feingranularer Kontrolle über Speicher, Threads, Kontextgröße und Quantisierung.

Ich nutze llama.cpp wenn:

- Ich maximale Kontrolle über Inference-Parameter brauche
- Das Deployment offline oder ohne Python-Stack laufen muss
- `llama-server` als OpenAI-kompatibler Endpunkt gefragt ist
- Ich GGUF-Modelle direkt aus dem Hub laden will

Arbeit: llama.cpp-Instanzen laufen als Kubernetes-Pods, je ein Modell pro Quadro-RTX-6000-Ada-GPU (24 GB VRAM), hinter LiteLLM als einheitlichem API-Proxy.

Zuhause: llama.cpp direkt auf der RTX 5070 (12 GB VRAM), meistens als Backend für Jan.ai oder OpenCode.

- llama.cpp RPC Multi-GPU Setup Guide (Windows 11 & Linux Distributed Cluster)
- Agentic LLM Inference Tuning Reference für Qwen 3.6 und Gemma 4
- *llama-server Konfiguration und Slot-Management* — in Vorbereitung
- *llama.cpp unter Kubernetes: Deployment-Muster und Fallstricke* — in Vorbereitung

---

LiteLLM

LiteLLM ist kein Inference-Backend — sondern der Proxy-Layer davor. Ein einheitlicher OpenAI-kompatibler Endpunkt, der Requests an verschiedene Backends weiterleitet: llama.cpp-Instanzen, vLLM, externe Cloud-APIs. LiteLLM habe ich hauptsächlich im Arbeitsumfeld im Einsatz. Private nur eine Instanz für Tests und Updates.

Ich nutze LiteLLM wenn:

- Mehrere Modelle und Backends hinter einer einzigen API-URL liegen sollen
- Load-Balancing zwischen mehreren llama.cpp-Instanzen gefragt ist
- Nutzungsstatistiken und API-Key-Management zentral verwaltet werden sollen
- Clients (Open WebUI, Coding-Tools, Skripte) nichts vom Backend-Wechsel mitbekommen sollen
- *LiteLLM als Proxy vor llama.cpp: Setup und Konfiguration* — in Vorbereitung
- *API-Usage-Tracking mit LiteLLM und Open WebUI* — in Vorbereitung

---

vLLM

vLLM ist auf Hochdurchsatz-Inferenz ausgelegt — PagedAttention, kontinuierliches Batching, tensor-paralleles Sharding. Ich habe vLLM installiert und getestet, betreibe es aber nicht produktiv, weil llama.cpp + LiteLLM für mein Workload-Profil ausreichend ist.

Interessant wird vLLM wenn:

- Viele parallele Requests gleichzeitig bedient werden müssen

- Durchsatz wichtiger ist als einfache Konfiguration
- Hugging-Face-Modelle (Safetensors) direkt geladen werden sollen
- Lokale Installation und Betrieb von LLMs mit vLLM
- llama.cpp — Installation, Betrieb und Konfiguration
- *vLLM vs. llama.cpp: Wann lohnt sich der Wechsel?* — in Vorbereitung

---

Jan.ai

Jan.ai ist mein primäres lokales Frontend zu Hause — eine plattformübergreifende App mit eingebautem llama.cpp-Backend, experimenteller MLX-Unterstütung, übersichtlicher Modellverwaltung mit Zugriff auf Huggingface und direktem Zugang zu Inference-Parametern. Gut geeignet für Experimente, lokale Assistenten mit MCP-Server Unterstützung, Rollenspiel-Prompts (nicht so gut wie SillyTavern) und Modellvergleiche im interaktiven Betrieb.

Ich nutze Jan.ai wenn:

- Ich schnell verschiedene Modelle ausprobieren will ohne CLI
- Parameter wie Temperature, Top-P oder Repeat-Penalty direkt anpassen will
- Eine lokale GUI ohne Cloud-Abhängigkeit gefragt ist
- *Jan.ai: Setup, Modellverwaltung und Parameter-Tuning* — in Vorbereitung
- *Jan.ai vs. LM Studio: Vergleich aus der Praxis* — in Vorbereitung

---

LM Studio

LM Studio ist eine Alternative zu Jan.ai mit ähnlichem Ansatz: lokale GUI, eingebettetes llama.cpp-Backend, OpenAI-kompatibler lokaler Server. Etwas stärker auf Entwickler-Workflows ausgerichtet, mit besserem Prompt-Template-Editor.

Ich nutze LM Studio wenn:

- Der eingebaute lokale Server als API-Endpunkt für andere Tools gefragt ist

- Ich Prompt-Templates für verschiedene Modellformate pflegen will
- Ich Modelle aus Hugging Face direkt in der App suchen und laden will

Weiterführende Links

- *LM Studio als lokaler API-Server für OpenCode und andere Clients* — in Vorbereitung

---

Open WebUI

Open WebUI ist das Team-Frontend im Arbeitsumfeld — eine ChatGPT-ähnliche Web-Oberfläche, die hinter LiteLLM auf die llama.cpp-Instanzen zugreift. Kollegen brauchen nichts zu installieren; sie öffnen den Browser.

Ich nutze Open WebUI wenn:

- Mehrere Nutzer Zugang zu den lokalen Modellen brauchen
- Ein benutzerfreundliches Frontend ohne technische Hürden gefragt ist
- RAG, Bildupload und Gesprächsverwaltung zentral verwaltet werden sollen

Weiterführende Links

- Open WebUI — Übersicht und Setup

OpenCode

OpenCode ist mein KI-gestütztes Coding-Werkzeug — ein CLI-Agent, der llama.cpp als Backend nutzt und direkt in den Entwicklungsworkflow integriert ist. Kein Frontend, kein Chat — sondern Codeanalyse, Refactoring und Aufgabenausführung im Terminal.

- The AI Coding Loop: Verification-Driven Development with AI Agents
- OpenCode LLM Benchmarks: IndexNow & Migration Mapping
- *OpenCode mit lokalem llama.cpp-Backend: Konfiguration und Modellwahl* — in Vorbereitung

---

Ollama

Ollama habe ich getestet, betreibe ich aber nicht aktiv. Es ist der einfachste Einstieg in lokale LLMs — gute CLI, automatisches Modell-Pulling, minimale Konfiguration. Die Einschränkungen beim Scheduling und die begrenzte Kontrolle über Inference-Parameter haben mich zu llama.cpp geführt.

Ollama macht Sinn wenn:

- Der schnellste Weg zu einem laufenden Modell gefragt ist
- Kein Interesse an Parameter-Tuning besteht
- Ein einzelner Nutzer lokal experimentiert

Weiterführende Links

- *Warum ich von Ollama zu llama.cpp gewechselt habe* — in Vorbereitung

---

Cloud LLM Hosting

Cloud-Anbieter abstrahieren die Hardware vollständig. Ich nutze Cloud-APIs punktuell — vor allem für Modelle, die lokal nicht sinnvoll betreibbar sind (frontier-Modelle wie Claude oder GPT-4-class).

Vorteile:

- Sofortige Skalierbarkeit
- Kein Hardware-Investment
- Keine Wartung

Nachteile:

- Laufende Kosten pro Token
- Daten verlassen die eigene Infrastruktur
- Vendor-Abhängigkeit

Weiterführende Links

- *Cloud-APIs sinnvoll einsetzen: wann lokales Hosting nicht reicht* — in Vorbereitung

---

Frontends & Oberflächen

Das Hosting-Backend ist nur ein Teil des Systems. Welches Frontend Nutzer sehen, beeinflusst die Akzeptanz erheblich.

Frontend	Kontext	Typ
Open WebUI	Arbeit, Team	Web-App
Jan.ai	Zuhause, Einzelnutzer	Desktop-App
LM Studio	Zuhause, Entwicklung	Desktop-App
OpenCode	Zuhause, Terminal	CLI-Agent

Kosten und Kontrollvergleich

Faktor	Lokal	Cloud
Anschaffungskosten	Hardware	Keine
Laufende Kosten	Strom	Token-Abrechnung
Datenschutz	Vollständig	Eingeschränkt
Skalierung	Manuell	Automatisch
Wartung	Selbst	Anbieter
Modellkontrolle	Vollständig	Eingeschränkt

Wann welcher Ansatz?

llama.cpp wenn maximale Kontrolle und GGUF-Modelle gefragt sind — sowohl für Einzelnutzer als auch im Kubernetes-Cluster.

LiteLLM sobald mehrere Backends oder mehrere Nutzer hinter einer einheitlichen API zusammengefasst werden sollen.

vLLM wenn llama.cpp unter Parallellast zum Engpass wird und Durchsatz Priorität hat.

Jan.ai oder LM Studio für lokale Experimente mit GUI, ohne CLI-Aufwand.

OpenCode wenn KI direkt in den Coding-Workflow integriert werden soll.

Open WebUI wenn Kollegen ohne technisches Setup Zugang zu lokalen Modellen brauchen.

Cloud wenn frontier-Modelle gefragt sind oder lokale Hardware nicht ausreicht.

Häufige Fragen

Was ist der einfachste Einstieg in lokale LLMs? Ollama — minimale Konfiguration, sofort lauffähig. Wer mehr Kontrolle braucht, wechselt zu llama.cpp.

Ist Self-Hosting günstiger als Cloud-APIs? Bei hohem, gleichmäßigem Volumen oft ja — sobald die Hardware amortisiert ist. Bei sporadischer Nutzung rechnet sich Cloud meistens besser.

Kann man LLMs ohne GPU betreiben? Ja, aber die Inferenzgeschwindigkeit ist erheblich eingeschränkt. Für produktiven Betrieb ist eine dedizierte GPU praktisch Pflicht.

Welches Frontend empfehle ich für ein Team? Open WebUI — es braucht keine lokale Installation auf den Client-Rechnern und lässt sich gut hinter einen Reverse-Proxy stellen.

Zur Performance und Optimierung der Inference-Backends: LLM Performance Hub

---

Revision #7

Created 2026-07-03 09:03:58 UTC by Carsten

Updated 2026-07-08 08:31:58 UTC by Carsten