llama.cpp — Installation, Betrieb und Konfiguration

llama.cpp ist mein primäres Inference-Backend — sowohl im Arbeitscluster als auch zu Hause. Es ist eine schlanke C/C++-Engine für GGUF-Modelle mit sehr direktem Zugang zu allen relevanten Parametern: Kontextgröße, GPU-Layer-Verteilung, Quantisierung, Threading. Kein Python-Stack, kein Framework-Overhead.

Der Einstieg ist etwas steiler als bei Ollama oder Jan.ai, aber die Kontrolle ist es wert sobald man weiß was man tut.

Wann llama.cpp, wann etwas anderes?

Situation	Empfehlung
GGUF-Modell, maximale Kontrolle	llama.cpp
Schneller Einstieg ohne CLI	Jan.ai oder LM Studio
Hugging-Face-Modell (Safetensors)	vLLM
Hoher Durchsatz, viele parallele Nutzer	vLLM
Team-Frontend ohne lokale Installation	Open WebUI + LiteLLM

Modelle beschaffen

llama.cpp arbeitet ausschließlich mit GGUF-Dateien. Die beste Anlaufstelle ist Hugging Face — dort gibt es für fast jedes bekannte Modell fertige GGUF-Versionen, meistens von bartowski oder direkt vom Modellhersteller.

Quantisierungsstufen im Überblick:

Quantisierung	VRAM-Bedarf (7B)	Qualität	Empfehlung
Q8_0	~8 GB	Sehr gut	Wenn VRAM reicht

Quantisierung	VRAM-Bedarf (7B)	Qualität	Empfehlung
Q6_K	~6 GB	Gut	Guter Kompromiss
Q4_K_M	~4.5 GB	Akzeptabel	Standard für knappes VRAM
Q4_K_S	~4 GB	Etwas schlechter	Nur wenn nötig
Q3_K_M	~3.5 GB	Spürbare Verluste	Notlösung

Für die RTX 5070 (12 GB) passen 7B-Modelle in Q8_0 bequem, 13B-Modelle in Q4_K_M noch gut. Für die Quadro RTX 6000 Ada (24 GB) sind 30B-Modelle in Q4_K_M oder kleinere Modelle in höherer Quantisierung realistisch.

Modell direkt von Hugging Face laden:

```
# Mit huggingface-cli (pip install huggingface_hub)
huggingface-cli download bartowski/Qwen3-8B-GGUF \
  Qwen3-8B-Q6_K.gguf \
  --local-dir ./models
```

Installation

Linux — aus dem Quellcode bauen

Empfohlen für maximale Kompatibilität und aktuelle Features. Auf einem i5 mit 8 parallelen Jobs ca. 6 Minuten Buildzeit.

Abhängigkeiten:

```
sudo apt install build-essential cmake git curl libcurl4-openssl-dev
```

Bauen mit CUDA-Support:

```
git clone https://github.com/ggml-org/llama.cpp
cd llama.cpp
cmake -B build -DGGML_CUDA=ON
cmake --build build --config Release -j 8
```

Die fertigen Binaries liegen in `build/bin/`.

Windows — winget

Der einfachste Weg für den Windows-Master-Node:

```
winget install llama.cpp
```

Wer eine spezifische Version oder CUDA-optimierte Builds braucht, findet diese als fertige Releases auf der llama.cpp GitHub-Seite.

---

llama-cli — Interaktiver Betrieb

`llama-cli` ist das Kommandozeilen-Interface für direkte Interaktion mit einem Modell. Gut für schnelle Tests, Prompt-Experimente und das Debuggen von Parametern.

Basis-Aufruf:

```
./build/bin/llama-cli \
  -m ./models/Qwen3-8B-Q6_K.gguf \
  --interactive \
  -ngl 99 \
  --ctx-size 8192
```

Mit System-Prompt:

```
./build/bin/llama-cli \
  -m ./models/Qwen3-8B-Q6_K.gguf \
  --interactive \
  -ngl 99 \
  --ctx-size 8192 \
  --system-prompt "Du bist ein hilfreicher Assistent."
```

Einzelne Ausgabe ohne interaktiven Modus (gut für Scripting):

```
./build/bin/llama-cli \
  -m ./models/Qwen3-8B-Q6_K.gguf \
  -ngl 99 \
  --ctx-size 4096 \
  -p "Erkläre den Unterschied zwischen TCP und UDP in drei Sätzen."
```

llama-server — OpenAI-kompatibler API-Endpunkt

`llama-server` ist das produktive Backend — startet einen HTTP-Server mit OpenAI-kompatibler API. Damit lassen sich LiteLLM, Open WebUI, OpenCode, Jan.ai und alle anderen OpenAI-kompatiblen Clients direkt anbinden.

Starten:

```
./build/bin/llama-server \
  -m ./models/Qwen3-8B-Q6_K.gguf \
  --host 0.0.0.0 \
  --port 8080 \
  -ngl 99 \
  --ctx-size 8192
```

API testen:

```
# Verfügbare Modelle
curl http://localhost:8080/v1/models

# Chat-Request
curl http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen3-8b",
    "messages": [
      {"role": "user", "content": "Was ist llama.cpp?"}
    ]
  }'
```

Wichtige Parameter

GPU und Speicher

Parameter	Beschreibung	Empfehlung
`-ngl` / `--n-gpu-layers`	Layer auf der GPU	`99` — alles auf GPU
`--ctx-size`	Kontextfenstergröße	8192 als Ausgangswert
`--flash-attn`	Flash Attention aktivieren	Immer

Parameter	Beschreibung	Empfehlung
`--mlock`	Modell im RAM fixieren	Empfohlen

Threading und Parallelität (llama-server)

Parameter	Beschreibung	Empfehlung
`--parallel`	Parallele Request-Slots	2–4 für Einzelnutzer
`--threads`	CPU-Threads für Prefill	Physische Kerne
`--threads-batch`	Threads für Batch-Verarbeitung	Wie `--threads`

Ausgabesteuerung

Parameter	Beschreibung
`--temp`	Temperature (0 = deterministisch)
`--top-p`	Nucleus Sampling
`--repeat-penalty`	Wiederholungsstrafe
`--seed`	Reproduzierbare Ausgaben

Ein typischer Produktions-Start für den Arbeitscluster (Quadro RTX 6000 Ada, 24 GB):

```
./build/bin/llama-server \
  -m ./models/Qwen3-Coder-30B-Q4_K_M.gguf \
  --host 0.0.0.0 \
  --port 8080 \
  -ngl 99 \
  --ctx-size 16384 \
  --flash-attn \
  --mlock \
  --parallel 4 \
  --threads 8
```

Für die RTX 5070 zu Hause (12 GB) mit einem 8B-Modell:

```
./build/bin/llama-server \
  -m ./models/Qwen3-8B-Q6_K.gguf \
  --host 127.0.0.1 \
```

```
  --port 8080 \
  -ngl 99 \
  --ctx-size 8192 \
  --flash-attn \
  --mlock \
  --parallel 2
```

Als systemd-Service betreiben

Für dauerhaften Betrieb auf einem Linux-Server:

```
[Unit]
Description=llama.cpp Inference Server
After=network.target

[Service]
Type=simple
User=DEIN_USER
WorkingDirectory=/home/DEIN_USER/llama.cpp
ExecStart=/home/DEIN_USER/llama.cpp/build/bin/llama-server \
  -m /home/DEIN_USER/models/Qwen3-8B-Q6_K.gguf \
  --host 0.0.0.0 \
  --port 8080 \
  -ngl 99 \
  --ctx-size 8192 \
  --flash-attn \
  --mlock \
  --parallel 4
Restart=on-failure
RestartSec=10

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload
sudo systemctl enable llamacpp
sudo systemctl start llamacpp
sudo systemctl status llamacpp
```

Integration mit LiteLLM

Im Arbeitscluster läuft llama-server hinter LiteLLM als einheitlichem API-Proxy. LiteLLM-Konfiguration für einen llama.cpp-Endpunkt:

```
model_list:
  - model_name: qwen3-coder-30b
    litellm_params:
      model: openai/qwen3-coder-30b
      api_base: http://localhost:8080
      api_key: none
```

Damit ist das Modell für alle LiteLLM-Clients unter `qwen3-coder-30b` erreichbar, ohne dass sie direkt mit llama-server kommunizieren müssen.

Fehlerbehebung

Modell startet, aber Inferenz ist sehr langsam Prüfen ob `-ngl 99` gesetzt ist und ob `nvidia-smi` während der Inferenz VRAM-Auslastung zeigt. Wenn der VRAM bei 0 bleibt, wurde llama.cpp ohne CUDA gebaut.

CUDA-Build schlägt fehl `nvcc --version` prüfen — wenn kein NVCC gefunden wird, fehlt das CUDA Toolkit:

```
sudo apt install nvidia-cuda-toolkit
```

Out of Memory beim Start Entweder `--ctx-size` reduzieren oder ein stärker quantisiertes Modell verwenden. Mit `--verbose` starten um die genaue VRAM-Kalkulation zu sehen.

Langsamer Prefill, schneller Decode Normales Verhalten — Prefill ist compute-bound, Decode ist memory-bound. Flash Attention (`--flash-attn`) verbessert den Prefill messbar.

Weiterführend

- llama.cpp RPC Cluster — verteilte Inferenz im Heimnetz
- Agentic LLM Inference Tuning für Qwen 3.6 und Gemma 4
- LLM Performance Hub

Revision #1

Created 2026-07-05 21:53:42 UTC by Carsten

Updated 2026-07-05 21:53:55 UTC by Carsten