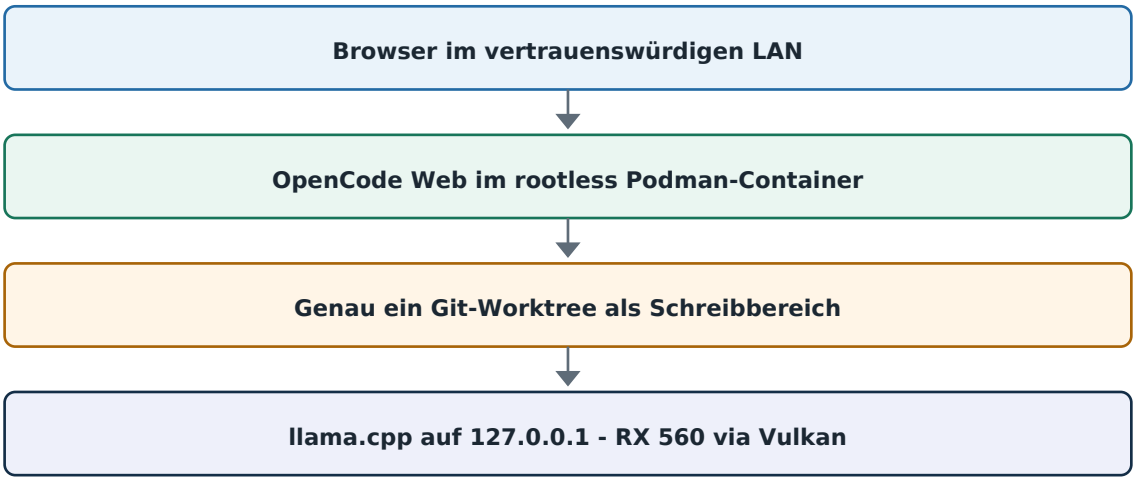
Lokaler Coding-Assistent

Ubuntu 26.04 LTS, Radeon RX 560 (4 GB), 32 GB RAM

Vollständige Schritt-für-Schritt-Anleitung für llama.cpp, OpenCode, OpenCode Web im LAN, Git-Worktrees und einen optional gehärteten Ausführungscontainer.



Zielbild

Ein einzelner lokaler Coding-Agent bearbeitet jeweils genau einen Task-Branch. Die Modell-API ist nicht im LAN sichtbar. OpenCode Web ist passwortgeschützt und wird nur für das lokale Subnetz freigegeben. Code-Completion, Telegram, Reverse Proxy und Open WebUI sind in dieser Ausbaustufe bewusst nicht enthalten.

Komponente	Aufgabe	Standard
llama.cpp	Lokale Inferenz mit Vulkan und CPU-Fallback	127.0.0.1:8080
Qwen2.5-Coder 7B	Coding-Modell, GGUF Q4_K_M	16k Kontext
OpenCode	Agent, Werkzeuge, Sitzungen und Web-UI	LAN-Port 4096
Git worktree	Ein separater Arbeitsbaum pro Task-Branch	ein Mount je Agent
Podman rootless	Worktree schreibbar, Git-Metadaten nur lesbar	empfohlen

Stand: 29. August 2026. Zielsystem: Ubuntu 26.04 LTS (Resolute Raccoon), x86-64.

Inhaltsverzeichnis

1. Ergebnis, Grenzen und empfohlener Weg

Ja, das Vorhaben ist auf der vorhandenen Hardware möglich. Die Radeon RX 560 beschleunigt einen Teil des Modells über Vulkan; der restliche Modellzustand liegt im Arbeitsspeicher und wird von der CPU verarbeitet. Für agentische Aufgaben ist die Antwortgeschwindigkeit daher deutlich niedriger als bei aktuellen High-End-GPUs. Der praktische Startpunkt ist ein 7B-Coding-Modell in 4-Bit-Quantisierung.

1.1 Was diese Anleitung am Ende bereitstellt

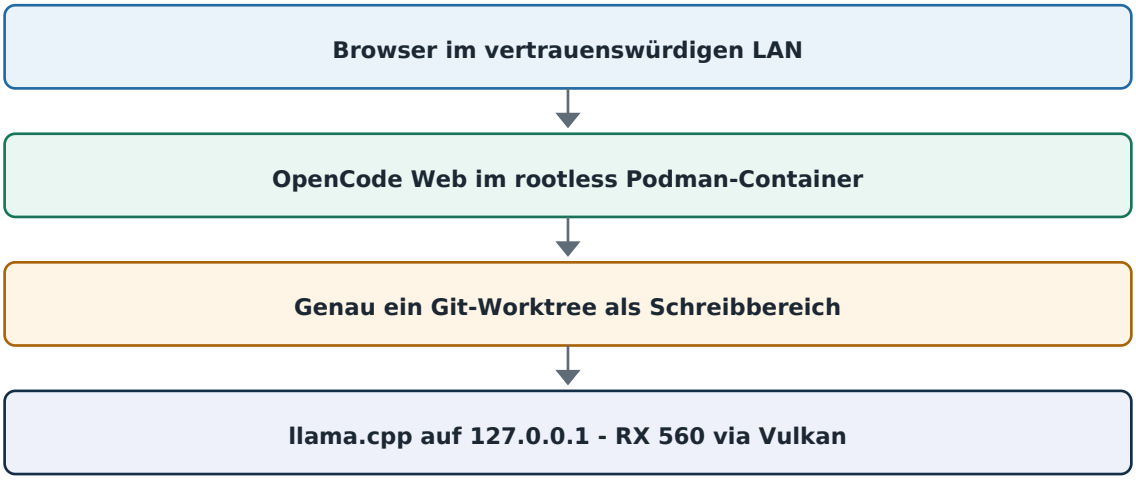
- Einen lokal gebauten llama.cpp-Server mit Vulkan-Backend und OpenAI-kompatibler API.
- Ein lokales Qwen2.5-Coder-7B-Instruct-GGUF Q4_K_M mit 16.384 Token Kontext als konservativem Startwert.
- OpenCode im Terminal für Funktionstests und OpenCode Web als LAN-Oberfläche.
- Ein Git-Worktree je Task, damit Änderungen verschiedener Aufgaben nicht vermischt werden.
- Einen empfohlenen rootless-Podman-Betrieb, in dem der Agent nur den gewählten Worktree schreiben kann.
- Start-, Test-, Update-, Backup- und Fehlerdiagnosebefehle.

1.2 Bewusste Grenzen

- Keine Inline-Code-Completion und keine JetBrains-Integration in dieser Ausbaustufe.
- Kein Telegram, kein Reverse Proxy, kein Zugriff aus dem Internet.
- Open WebUI wird nur als späterer optionaler Ausbau beschrieben, nicht installiert.
- Ein 7B-Modell kann kleine bis mittlere Änderungen selbstständig erledigen, ist aber bei großen Refactorings, subtilen Architekturentscheidungen und langen Tool-Ketten fehleranfälliger als starke Cloud-Modelle.
- OpenCode-Berechtigungen sind Leitplanken, keine Isolation. Die eigentliche Dateisystemgrenze entsteht erst durch den Container und den einzelnen Worktree-Mount.

Empfohlene Reihenfolge	Zuerst GPU, Modellserver und OpenCode lokal testen. Danach Git-Worktree anlegen. Erst dann OpenCode Web im Container für das LAN starten. So bleibt jede Fehlerquelle einzeln prüfbar.

2. Architektur und Sicherheitsmodell



2.1 Daten- und Befehlsfluss

1. Der Browser verbindet sich über HTTP Basic Auth mit OpenCode Web auf Port 4096.
2. OpenCode liest den gemounteten Worktree, erstellt einen Plan und sendet Modellanfragen an die lokale llama.cpp-API.
3. llama.cpp verarbeitet Prompts lokal. Der Port 8080 bleibt auf 127.0.0.1 gebunden und ist nicht aus dem LAN erreichbar.
4. OpenCode fordert je nach Regel eine Bestätigung an oder führt erlaubte Aktionen aus. Im Container sind nur der Task-Worktree und ausgewählte Cache-Verzeichnisse beschreibbar.
5. Du prüfst Diff und Tests, bevor du commitest, mergst oder den Worktree entfernst.

2.2 Bedrohungsmodell

Risiko	Gegenmaßnahme	Verbleibendes Risiko
Fehlerhafter Shell-Befehl	rootless Container, read-only Basis, cap-drop, einzelner Projekt-RW-Mount	Dateien im gemounteten Worktree können beschädigt werden
Zugriff auf private Dateien	kein Mount von Home, SSH, Browserprofil oder Docker-Socket	Dateien im Worktree sind vollständig sichtbar
Unbefugter LAN-Zugriff	starkes Passwort und Firewall-Regel auf vertrauenswürdiges Subnetz	Basic Auth läuft ohne TLS; nicht für fremde Netze
Unbeabsichtigtes Pushen	kein SSH-Agent/Token-Mount und git push in OpenCode deny	bereits im Repository gespeicherte Zugangsdaten wären sichtbar
Prompt Injection im Repository	manuelle Freigaben, deny-Regeln, Diff-Review	Modell kann trotzdem schädliche Vorschläge machen

Nicht ins Internet stellen

Ohne TLS und vorgeschaltete Authentisierung ist diese Konfiguration ausschließlich für ein vertrauenswürdiges Heim- oder Entwicklungs-LAN gedacht. Keine Portweiterleitung im Router einrichten.

3. Hardware-Erwartungen und Modellwahl

3.1 Was die RX 560 leisten kann

Die RX 560 gehört zur Polaris-Generation und besitzt nur 4 GB VRAM. Vulkan über Mesa/RADV ist hier der pragmatische Weg; ROCm ist für dieses ältere Modell unnötig kompliziert und je nach Release nicht offiziell unterstützt. Da Q4_K_M des 7B-Modells ungefähr 4 bis 5 GB Modellgewicht benötigt, passt nicht alles gleichzeitig in den VRAM. llama.cpp lagert deshalb einen Teil der Transformer-Schichten auf die GPU aus und lässt den Rest auf der CPU.

Variante	Eignung	Startparameter
Qwen2.5-Coder 3B Q4_K_M	schneller, weniger zuverlässig bei Agentenketten	8k bis 16k Kontext, 20+ GPU-Layer testen
Qwen2.5-Coder 7B Q4_K_M	empfohlener Kompromiss	16k Kontext, 12 GPU-Layer
Qwen2.5-Coder 14B Q4_K_M	bessere Qualität, deutlich langsamer	8k bis 16k Kontext, 4 bis 10 GPU-Layer
CPU-only	funktioniert immer, aber langsam	--n-gpu-layers 0

3.2 Realistische Arbeitsweise

- Gib kleine, überprüfbare Tasks mit klaren Akzeptanzkriterien statt sehr großer Umbauten.
- Lasse zunächst analysieren, dann implementieren und anschließend gezielte Tests ausführen.
- Führe nur einen aktiven Agenten gleichzeitig aus. Der Modellserver ist auf eine parallele Sequenz eingestellt.
- Rechne bei großen Repositories mit längeren Prompt-Verarbeitungszeiten. Beschränke den Task auf relevante Module.

Kontext ist nicht kostenlos

32k Kontext ist laut Modellkarte möglich, erhöht aber RAM/VRAM-Bedarf und Latenz. 16k ist für diese Hardware der bessere Start. Bei Speicherproblemen zuerst auf 8192 reduzieren.

4. Platzhalter und Vorab-Check

4.1 In Befehlen verwendete Platzhalter

Platzhalter	Beispiel	Bedeutung
<LAN-IP>	192.168.178.50	statische oder reservierte IPv4-Adresse des Ubuntu-Rechners
<LAN-CIDR>	192.168.178.0/24	vertrauenswürdiges lokales Subnetz
<REPO>	/home/alex/code/shop	Pfad zum vorhandenen Haupt-Repository
<WORKTREE>	/home/alex/worktrees/shop/task-001	separater Arbeitsbaum für den Task
<STARTBRANCH>	main	Branch, von dem der Task abzweigt
<TASKBRANCH>	ai/task-001	neuer Branch für die Aufgabe

4.2 System erfassen

```
cat /etc/os-release
uname -a
uname -m
free -h
df -h /
lscpu | sed -n '1,25p'
lspci -nnk | grep -EA3 'VGA|3D|Display'
ip -4 -brief address
ip route
```

Erwartet werden Ubuntu 26.04.x LTS, x86_64, ungefähr 32 GB RAM und die Radeon RX 560 mit dem Kernel-Treiber amdgpu. Wenn das System über SSH eingerichtet wird, vor Firewall-Änderungen unbedingt die bestehende SSH-Verbindung und den SSH-Port berücksichtigen.

4.3 Sicherung und sauberer Git-Zustand

```
cd <REPO>
git status --short
git branch --show-current
git remote -v
git worktree list
```

Vor Änderungen Nicht commitete eigene Änderungen zuerst committen, stashen oder in einem separaten Worktree belassen. Die Anleitung löscht keine bestehenden Änderungen.

5. Ubuntu 26.04 vorbereiten

5.1 Aktualisieren

```
sudo apt update
sudo apt full-upgrade -y
sudo reboot
```

Nach dem Neustart erneut anmelden und mit `cat /etc/os-release` kontrollieren, dass weiterhin Ubuntu 26.04 verwendet wird.

5.2 Pakete installieren

```
sudo apt install -y \
  build-essential cmake ninja-build pkg-config ccache \
  git git-lfs curl ca-certificates jq ripgrep openssl \
  libvulkan-dev glslc spirv-headers vulkan-tools mesa-vulkan-drivers \
  libopenblas-dev libcurl4-openssl-dev libssl-dev \
  podman uidmap slirp4netns fuse-overlayfs ufw
```

Falls ein Paketname abweicht

```
apt-cache policy glslc spirv-headers mesa-vulkan-drivers
apt search '^glslc$|spirv-headers|vulkan-tools'
sudo apt install -y shaderc  # nur falls glslc nicht als eigenes Paket existiert
```

Die offizielle llama.cpp-Bauanleitung nennt für Debian/Ubuntu mindestens `libvulkan-dev glslc spirv-headers`. OpenBLAS beschleunigt vor allem die Prompt-Verarbeitung der CPU-Anteile; es verändert die Token-Generierung selbst kaum.

5.3 Git LFS initialisieren

```
git lfs install
git --version
git lfs version
cmake --version
ninja --version
podman --version
```

6. Radeon und Vulkan prüfen

6.1 Kernel-Treiber und Gerätezugriff

```
lspci -nnk | grep -EA3 'VGA|3D|Display'
ls -l /dev/dri
id
getent group render
getent group video
```

Beim RX-560-Eintrag sollte `Kernel driver in use: amdgpu` erscheinen. Unter `/dev/dri` wird üblicherweise mindestens ein `renderD*`-Gerät angezeigt.

Falls der Benutzer nicht in render/video ist

```
sudo usermod -aG render,video "$USER"
# Danach vollständig abmelden und wieder anmelden.
id
```

6.2 Vulkan-Stack validieren

```
vulkaninfo --summary
vulkaninfo 2>/dev/null | grep -E 'deviceName|driverName|driverInfo|apiVersion' | head -20
```

Abbruchkriterium	Wenn als Gerät nur llvmpipe erscheint, arbeitet Vulkan rein auf der CPU. In diesem Fall nicht weiter optimieren, sondern zuerst Mesa/amdgpu reparieren. Bei einer Radeon sollte RADV beziehungsweise AMD RADV sichtbar sein.

6.3 Häufige Vulkan-Abweichungen

Symptom	Maßnahme
vulkaninfo: command not found	vulkan-tools erneut installieren
Permission denied auf /dev/dri/renderD*	render/video-Gruppen prüfen und neu anmelden
Nur llvmpipe	mesa-vulkan-drivers prüfen; keine fremden Vulkan-ICD-Variablen setzen
Mehrere Vulkan-Geräte	später llama-server --list-devices prüfen und bei Bedarf --device verwenden
Desktop friert bei hoher Last	GPU-Layer reduzieren, Kontext verkleinern, Cache begrenzen

7. llama.cpp mit Vulkan bauen

7.1 Quellcode beziehen

```
mkdir -p "$HOME/src"
cd "$HOME/src"
git clone https://github.com/ggml-org/llama.cpp.git
cd llama.cpp
git status --short
git rev-parse --short HEAD
```

Für maximale Reproduzierbarkeit kann später ein getesteter Commit oder Release-Tag ausgecheckt werden. Für die erste Installation ist der aktuelle Hauptzweig sinnvoll, weil Vulkan und Tool-Calling regelmäßig verbessert werden.

7.2 Release-Build mit Vulkan und OpenBLAS

```
cd "$HOME/src/llama.cpp"
cmake -S . -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=Release \
  -DGGML_VULKAN=ON \
  -DGGML_BLAS=ON \
  -DGGML_BLAS_VENDOR=OpenBLAS \
  -DLLAMA_CURL=ON \
  -DGGML_NATIVE=ON

cmake --build build --config Release -j "$(nproc)" \
  --target llama-server llama-cli llama-bench
```

7.3 Build prüfen

```
./build/bin/llama-server --version
./build/bin/llama-server --list-devices
ldd ./build/bin/llama-server | grep -E 'vulkan|openblas|curl'
```

In der Geräteliste muss die Radeon erscheinen. Wenn der Build Vulkan nicht erkennt, den Build-Ordner neu konfigurieren, nachdem die Vulkan-Pakete installiert wurden:

```
cd "$HOME/src/llama.cpp"
rm -r build
cmake -S . -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=Release \
  -DGGML_VULKAN=ON \
  -DGGML_BLAS=ON \
  -DGGML_BLAS_VENDOR=OpenBLAS \
  -DLLAMA_CURL=ON
cmake --build build -j "$(nproc)" --target llama-server llama-cli llama-bench
```

Zur Löschzeile	Der Befehl entfernt ausschließlich den generierten Build-Ordner im zuvor ausdrücklich gesetzten llama.cpp-Verzeichnis. Niemals einen variablen oder ungeprüften Pfad an rm -r übergeben.

8. Modell laden und einzeln testen

8.1 Empfohlenes Modell

Verwendet wird das offizielle GGUF-Repository `Qwen/Qwen2.5-Coder-7B-Instruct-GGUF` mit der Quantisierung `Q4_K_M`. Die Modellkarte nennt 7,61 Milliarden Parameter, 28 Schichten, 32.768 Token nativen GGUF-Kontext und Apache-2.0-Lizenz.

8.2 Erster Download und CLI-Test

```
cd "$HOME/src/llama.cpp"
./build/bin/llama-cli \
  -hf Qwen/Qwen2.5-Coder-7B-Instruct-GGUF:Q4_K_M \
  --n-gpu-layers 12 \
  --ctx-size 4096 \
  --predict 160 \
  --temp 0.2 \
  -p 'Schreibe eine kurze Python-Funktion is_prime(n) mit drei Tests.'
```

Beim ersten Start lädt llama.cpp die GGUF-Datei in seinen Cache. Der Download ist mehrere Gigabyte groß. Im Startprotokoll muss ein Vulkan-Gerät erscheinen und es sollten GPU-Layer ausgelagert werden.

8.3 CPU-Gegenprobe

```
./build/bin/llama-cli \
  -hf Qwen/Qwen2.5-Coder-7B-Instruct-GGUF:Q4_K_M \
  --n-gpu-layers 0 \
  --ctx-size 2048 \
  --predict 64 \
  -p 'Antworte nur mit: CPU-Test erfolgreich.'
```

Wenn CPU-only funktioniert, Vulkan aber nicht, sind Modell und Binärdatei grundsätzlich in Ordnung. Die Fehlersuche kann dann auf Treiber, VRAM und GPU-Layer beschränkt werden.

9. RX-560-Tuning ohne Rätselraten

9.1 Startwerte

Option	Start	Warum
--n-gpu-layers	12	konservativer Teil-Offload bei 4 GB VRAM
--ctx-size	16384	ausreichend für viele Agentenaufgaben, noch beherrschbarer Cache
--parallel	1	ein Agent; vermeidet geteilten KV-Cache
--cache-type-k/v	q8_0	reduziert KV-Speicher gegenüber f16
--flash-attn	auto	Backend entscheidet; bei Fehlern auf off setzen
--cache-ram	2048	begrenzt zusätzlichen Prompt-Cache

9.2 Kurzer Benchmark

```
cd "$HOME/src/llama.cpp"
./build/bin/llama-bench \
  -hf Qwen/Qwen2.5-Coder-7B-Instruct-GGUF:Q4_K_M \
  --n-gpu-layers 12 \
  -p 512 -n 64
```

Notiere Prompt-Verarbeitung (pp) und Token-Generierung (tg). Teste danach 8, 12 und 16 GPU-Layer einzeln. Behalte den höchsten stabilen Wert, der den Desktop nicht in den Swap drückt und keine Vulkan- oder Speicherfehler erzeugt.

9.3 Reihenfolge bei Problemen

1. GPU-Layer von 12 auf 8, dann 4 reduzieren.
2. Kontext von 16384 auf 8192 reduzieren.
3. Flash Attention explizit mit `--flash-attn off` deaktivieren.
4. KV-Cache testweise wieder auf f16 setzen, falls q8_0 beim Backend Probleme macht.
5. Als sichere Rückfallebene `--n-gpu-layers 0` verwenden.

Optionaler RADV-Leistungstest

In llama.cpp-Community-Benchmarks wird für RADV gelegentlich `RADV_PERFTEST=nogttspill` empfohlen. Das ist kein notwendiger Standard. Nur verwenden, wenn ein A/B-Benchmark auf dem eigenen Rechner eine stabile Verbesserung zeigt:

```
RADV_PERFTEST=nogttspill ./build/bin/llama-bench \
  -hf Qwen/Qwen2.5-Coder-7B-Instruct-GGUF:Q4_K_M \
  --n-gpu-layers 12 -p 512 -n 64
```

10. llama.cpp als lokalen Dienst betreiben

10.1 Server zunächst manuell starten

```
cd "$HOME/src/llama.cpp"
./build/bin/llama-server \
  -hf Qwen/Qwen2.5-Coder-7B-Instruct-GGUF:Q4_K_M \
  --alias qwen2.5-coder-7b-q4km \
  --host 127.0.0.1 \
  --port 8080 \
  --ctx-size 16384 \
  --parallel 1 \
  --n-gpu-layers 12 \
  --cache-type-k q8_0 \
  --cache-type-v q8_0 \
  --flash-attn auto \
  --jinja \
  --temp 0.2 \
  --top-p 0.9 \
  --cache-ram 2048 \
  --ctx-checkpoints 4
```

Warum --jinja? Der eingebettete Chat-Template wird damit auch für Funktions- und Tool-Aufrufe genutzt. Agentische Clients benötigen zuverlässige strukturierte Tool-Calls.

10.2 Health-, Modell- und Chat-Test

```
curl -fsS http://127.0.0.1:8080/health
curl -fsS http://127.0.0.1:8080/v1/models | jq

curl -fsS http://127.0.0.1:8080/v1/chat/completions \
  -H 'Content-Type: application/json' \
  -d '{
    "model": "qwen2.5-coder-7b-q4km",
    "messages": [
      {"role": "user", "content": "Antworte exakt mit: API OK"}
    ],
    "temperature": 0.1,
    "max_tokens": 32
  }' | jq
```

10.3 Tool-Call-Test

```
curl -fsS http://127.0.0.1:8080/v1/chat/completions \
  -H 'Content-Type: application/json' \
  -d '{
    "model": "qwen2.5-coder-7b-q4km",
    "messages": [
      {"role": "user", "content": "Nutze get_project_status."}
    ],
    "tools": [{
      "type": "function",
      "function": {
        "name": "get_project_status",
        "description": "Liefert den Projektstatus",
        "parameters": {"type": "object", "properties": {}}
      }
    }],
    "tool_choice": "auto",
    "max_tokens": 256
  }' | jq '.choices[0].message'
```

Erwartet wird ein Feld `tool_calls` mit dem Funktionsnamen. Falls nur Prosa ausgegeben wird, zuerst aktuellen llama.cpp-Stand, `--jinja` und das richtige Instruct-Modell prüfen.

11. llama.cpp automatisch mit systemd starten

11.1 Benutzer-Service anlegen

```
mkdir -p "$HOME/.config/systemd/user"
nano "$HOME/.config/systemd/user/llama-server.service"
```

Dateiinhalt:

```
[Unit]
Description=Lokaler llama.cpp Coding-Modellserver
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
WorkingDirectory=%h/src/llama.cpp
ExecStart=%h/src/llama.cpp/build/bin/llama-server \
  -hf Qwen/Qwen2.5-Coder-7B-Instruct-GGUF:Q4_K_M \
  --alias qwen2.5-coder-7b-q4km \
  --host 127.0.0.1 --port 8080 \
  --ctx-size 16384 --parallel 1 \
  --n-gpu-layers 12 \
  --cache-type-k q8_0 --cache-type-v q8_0 \
  --flash-attn auto --jinja \
  --temp 0.2 --top-p 0.9 \
  --cache-ram 2048 --ctx-checkpoints 4
Restart=on-failure
RestartSec=5
Nice=5
MemoryHigh=16G
MemoryMax=22G

[Install]
WantedBy=default.target
```

11.2 Aktivieren und prüfen

```
systemctl --user daemon-reload
systemctl --user enable --now llama-server.service
systemctl --user status llama-server.service --no-pager
journalctl --user -u llama-server.service -n 100 --no-pager
curl -fsS http://127.0.0.1:8080/health
```

Nach Abmeldung weiterlaufen lassen

```
sudo loginctl enable-linger "$USER"
loginctl show-user "$USER" -p Linger
```

Linger ist nur nötig, wenn der Dienst auch ohne aktive Desktop-Sitzung laufen soll. Auf einem reinen Arbeitsplatzrechner kann darauf verzichtet werden.

12. OpenCode installieren und konfigurieren

12.1 Host-Installation für Funktionstests

```
curl -fsSL https://opencode.ai/install | bash
exec "$SHELL" -l
command -v opencode
opencode --version
```

Alternative über npm

```
sudo apt install -y nodejs npm
sudo npm install -g opencode-ai
opencode --version
```

Nur eine Installationsmethode verwenden. Der offizielle Installer legt das Programm typischerweise unter `~/.opencode/bin` ab und ergänzt den PATH. Bei einem Shell-Neustart wird die Änderung wirksam.

12.2 Globale Konfiguration

```
mkdir -p "$HOME/.config/opencode"
nano "$HOME/.config/opencode/opencode.jsonc"
```

Dateiinhalt:

```
{
  "$schema": "https://opencode.ai/config.json",
  "model": "llama.cpp/qwen2.5-coder-7b-q4km",
  "small_model": "llama.cpp/qwen2.5-coder-7b-q4km",
  "enabled_providers": ["llama.cpp"],
  "share": "disabled",
  "autoupdate": "notify",
  "snapshot": true,
  "compaction": {"auto": true, "prune": true, "reserved": 4096},
  "provider": {
    "llama.cpp": {
      "npm": "@ai-sdk/openai-compatible",
      "name": "llama.cpp lokal",
      "options": {
        "baseURL": "http://127.0.0.1:8080/v1",
        "timeout": 600000,
        "chunkTimeout": 120000
      },
      "models": {
        "qwen2.5-coder-7b-q4km": {
          "name": "Qwen2.5-Coder 7B Q4_K_M lokal",
          "limit": {"context": 16384, "output": 4096}
        }
      }
    }
  },
  "permission": {
    "*": "ask",
    "external_directory": "deny",
    "read": {
      "*": "allow",
      "*.env": "deny",
      "*.env.*": "deny",
      "**/.ssh/**": "deny"
    },
    "edit": "ask",
    "bash": {
      "*": "ask",
      "pwd": "allow",
      "ls *": "allow",
      "git status*": "allow",
      "git diff*": "allow",
      "git log*": "allow",
      "git branch --show-current": "allow",
      "sudo *": "deny",
      "su *": "deny",
      "rm -rf *": "deny",
      "git clean*": "deny",
      "git reset --hard*": "deny",
      "git push*": "deny",
      "docker *": "deny",
      "podman *": "deny",
      "ssh *": "deny",
      "scp *": "deny"
    }
  }
}
```

Regelreihenfolge	Bei granularen OpenCode-Regeln gewinnt die letzte passende Regel. Deshalb steht die breite ask-Regel zuerst und die konkreten allow/deny-Ausnahmen folgen danach.

12.3 Konfiguration plausibilisieren

```
jq empty "$HOME/.config/opencode/opencode.jsonc" \
  || echo 'JSONC kann Kommentare enthalten; ohne Kommentare muss jq erfolgreich sein.'
curl -fsS http://127.0.0.1:8080/v1/models | jq -r '.data[].id'
```

13. OpenCode im Terminal testen

13.1 Wegwerf-Repository anlegen

```
TEST_REPO="$HOME/opencode-smoketest"
mkdir -p "$TEST_REPO"
cd "$TEST_REPO"
git init
git config user.name "OpenCode Test"
git config user.email "opencode-test@localhost"
printf 'def add(a, b):\n    return a + b\n' > calc.py
git add calc.py
git commit -m 'Initialer Smoke-Test'
opencode
```

In OpenCode zuerst den Plan-Agenten wählen und eingeben:

```
Analysiere calc.py. Plane zwei sinnvolle pytest-Tests.
Nimm noch keine Änderungen vor.
```

Danach zum Build-Agenten wechseln:

```
Implementiere die geplanten Tests. Ändere calc.py nur, wenn ein Test
einen echten Fehler zeigt. Führe die Tests aus und fasse das Ergebnis zusammen.
```

13.2 Erwartete Prüfpunkte

- Im llama.cpp-Journal erscheinen Modellanfragen ohne 4xx/5xx-Fehler.
- OpenCode kann Dateien lesen und fragt vor Änderungen und Shell-Befehlen nach Erlaubnis.
- Der Agent erzeugt einen Diff im Test-Repository und fasst ausgeführte Tests zusammen.
- Ein abgelehnter Befehl wird nicht ausgeführt.

13.3 Logs bei Fehlern

```
opencode --print-logs --log-level DEBUG
ls -lt "$HOME/.local/share/opencode/log" | head
journalctl --user -u llama-server.service -n 200 --no-pager
```

14. Projektregeln mit AGENTS.md

14.1 Datei erzeugen

OpenCode kann im Projekt über `/init` eine AGENTS.md erstellen. Bei einem kleinen lokalen Modell ist eine kurze manuell geprüfte Datei meist zuverlässiger. Beispiel:

```
# AGENTS.md

## Aufgabe und Grenzen
- Arbeite ausschließlich im aktuellen Git-Worktree.
- Verändere keine Dateien außerhalb des Repositories.
- Kein git push, kein sudo, keine Zugangsdaten lesen oder ausgeben.
- Vor größeren Änderungen zuerst einen kurzen Plan erstellen.

## Projektbefehle
- Build: <HIER_BUILD_BEFEHL_EINTRAGEN>
- Tests: <HIER_TEST_BEFEHL_EINTRAGEN>
- Lint: <HIER_LINT_BEFEHL_EINTRAGEN>

## Qualitätsregeln
- Bestehenden Stil beibehalten.
- Nur für den Task notwendige Dateien ändern.
- Neue Logik mit gezielten Tests abdecken.
- Am Ende geänderte Dateien, Tests und offene Risiken nennen.
- Nicht committen, bis der Benutzer den Diff geprüft hat.
```

14.2 Datei versionieren

```
cd <REPO>
git add AGENTS.md
git commit -m 'Dokumentiere Regeln für Coding-Agenten'
```

Keine Geheimnisse AGENTS.md wird vollständig an das Modell gesendet. Keine Tokens, Passwörter, internen URLs oder vertraulichen Daten darin ablegen.

15. Ein Git-Worktree pro Task

15.1 Neuen Task-Branch mit Worktree anlegen

```
cd <REPO>
git status --short
git fetch --all --prune

mkdir -p "$HOME/opencode-worktrees/<PROJEKTNAME>"
git worktree add \
  -b <TASKBRANCH> \
  "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001" \
  <STARTBRANCH>

git worktree list
git -C "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001" \
  branch --show-current
```

Wenn der Task-Branch bereits existiert

```
cd <REPO>
git worktree add \
  "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001" \
  <TASKBRANCH>
```

15.2 Warum nicht nur git checkout?

Ein Worktree ist ein eigener Ordner mit eigenem ausgechecktem Branch, teilt aber die Git-Objektdatenbank mit dem Haupt-Repository. Dadurch bleiben IDE-Arbeit, manuelle Änderungen und Agententasks räumlich getrennt. Für die Containergrenze ist dieser Ordner zugleich der einzige beschreibbare Projekt-Mount.

15.3 Regeln für mehrere Aufgaben

- Jeder Task erhält einen eindeutigen Branch und Ordner.
- Je aktiver OpenCode-Webinstanz genau einen Worktree mounten.
- Auf dieser Hardware nur eine Instanz gleichzeitig arbeiten lassen.
- Ports bei parallelen reinen Review-Sitzungen eindeutig vergeben, zum Beispiel 4096, 4097, 4098.

15.4 Besonderheit der Worktree-Gitdaten

Die Datei `.git` eines Worktrees verweist auf Metadaten im Haupt-Repository. Nur der Worktree-Mount reicht deshalb für `git status` und `git diff` im Container nicht aus. Kapitel 18 bindet das von Git ermittelte gemeinsame Metadatenverzeichnis zusätzlich read-only am identischen absoluten Pfad ein. Commits erfolgen weiterhin außerhalb des Containers.

Alternative mit noch klarerer Grenze

Statt eines Worktrees kann pro Task ein eigenständiger lokaler Clone verwendet werden. Dann liegt .git vollständig im Task-Ordner, benötigt aber zusätzlichen Plattenplatz:

```
git clone --local --no-hardlinks <REPO> \
  "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001"
cd "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001"
git switch -c <TASKBRANCH> <STARTBRANCH>
```

16. OpenCode Web zuerst nur lokal testen

16.1 Passwort erzeugen

```
mkdir -p "$HOME/.config/opencode"
chmod 700 "$HOME/.config/opencode"
SERVER_PASSWORD="$(openssl rand -base64 24)"
umask 077
printf 'OPENCODE_SERVER_USERNAME=opencode\nOPENCODE_SERVER_PASSWORD=%s\n' \
  "$SERVER_PASSWORD" > "$HOME/.config/opencode/server.env"
printf 'OpenCode-Webpasswort: %s\n' "$SERVER_PASSWORD"
unset SERVER_PASSWORD
chmod 600 "$HOME/.config/opencode/server.env"
```

Das angezeigte Passwort in einem Passwortmanager speichern. Die Datei nicht in Git aufnehmen.

16.2 Nur auf localhost starten

```
cd "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001"
set -a
. "$HOME/.config/opencode/server.env"
set +a
opencode web --hostname 127.0.0.1 --port 4096
```

Im Browser des Ubuntu-Rechners `http://127.0.0.1:4096` öffnen. Benutzername ist standardmäßig opencode. Ohne Passwort darf die Seite nicht zugänglich sein.

16.3 TUI an dieselbe Sitzung anbinden

```
set -a
. "$HOME/.config/opencode/server.env"
set +a
opencode attach http://127.0.0.1:4096
```

Noch keine Sandbox Dieser lokale Test läuft direkt auf dem Host. OpenCode-Berechtigungen fragen nach, aber ein freigegebener Befehl besitzt deine normalen Benutzerrechte. Für den LAN-Dauerbetrieb folgt deshalb der Containerweg.

17. Empfohlener LAN-Betrieb in rootless Podman

17.1 Warum rootless Podman

Rootless Podman startet Container ohne root-Daemon. Container-root wird in einen unprivilegierten Benutzer-Namespace abgebildet. Zusammen mit einem read-only Basisdateisystem, entfernten Linux-Capabilities und genau einem schreibbaren Worktree reduziert das den möglichen Schaden deutlich.

17.2 Podman prüfen

```
podman info --format '{{.Host.Security.Rootless}}'
podman run --rm docker.io/library/alpine:latest id
```

Die erste Ausgabe soll `true` sein. Falls Rootless-Mappings fehlen:

```
grep "^$USER:" /etc/subuid /etc/subgid
sudo usermod --add-subuids 100000-165535 "$USER"
sudo usermod --add-subgids 100000-165535 "$USER"
podman system migrate
```

17.3 Ubuntu-26-Agentenimage erstellen

```
mkdir -p "$HOME/opencode-container"
cd "$HOME/opencode-container"
nano Containerfile
```

Minimaler, allgemein nutzbarer Dateiinhalt:

```
FROM ubuntu:26.04

ARG DEBIAN_FRONTEND=noninteractive
ARG OPENCODE_VERSION=latest

RUN apt-get update && apt-get install -y --no-install-recommends \
    ca-certificates curl git git-lfs jq ripgrep \
    build-essential cmake ninja-build pkg-config \
    python3 python3-venv python3-pip \
    nodejs npm openjdk-21-jdk-headless \
    && rm -rf /var/lib/apt/lists/*

RUN npm install -g "opencode-ai@${OPENCODE_VERSION}" \
    && opencode --version \
    && git lfs install --system

WORKDIR /workspace
ENTRYPOINT ["opencode"]
```

```
cd "$HOME/opencode-container"
podman build --pull=always \
  --build-arg OPENCODE_VERSION=latest \
  -t localhost/opencode-agent:ubuntu26 .

podman run --rm localhost/opencode-agent:ubuntu26 --version
```

Leichtere Alternative

Wenn das Projekt keine Compiler, Java, Node oder Python im Agentencontainer benötigt, kann das offizielle OpenCode-Image verwendet werden. Es ist kleiner, enthält aber möglicherweise nicht alle projektspezifischen Werkzeuge:

```
podman pull ghcr.io/anomalyco/opencode:latest
podman run --rm ghcr.io/anomalyco/opencode:latest --version
```

18. Einen Task-Worktree im Container freigeben

18.1 Persistente, getrennte OpenCode-Verzeichnisse

```
mkdir -p \
  "$HOME/.local/share/opencode-container" \
  "$HOME/.cache/opencode-container"
chmod 700 \
  "$HOME/.local/share/opencode-container" \
  "$HOME/.cache/opencode-container"
```

18.2 Pfad validieren

```
WORKTREE_PATH="$(realpath "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001")"
test -d "$WORKTREE_PATH" || { echo 'Worktree fehlt'; exit 1; }
git -C "$WORKTREE_PATH" rev-parse --is-inside-work-tree
git -C "$WORKTREE_PATH" status --short
GIT_COMMON_DIR="$(git -C "$WORKTREE_PATH" \
  rev-parse --path-format=absolute --git-common-dir)"
test -d "$GIT_COMMON_DIR" || { echo 'Git-Metadaten fehlen'; exit 1; }
printf 'Freigegebener Worktree: %s\n' "$WORKTREE_PATH"
printf 'Git-Metadaten read-only: %s\n' "$GIT_COMMON_DIR"
```

Mounts vor Start lesen	Der Worktree ist der einzige schreibbare Projekt-Mount. Das gemeinsame Git-Verzeichnis wird nur lesbar eingebunden. Niemals das gesamte Home-Verzeichnis, den Projekt-Elternordner, /, /var/run/docker.sock, ~/.ssh oder einen SSH-Agenten mounten.

18.3 OpenCode Web im Container starten

```
podman run --rm -it \
  --name opencode-task-001 \
  --network host \
  --cap-drop all \
  --security-opt no-new-privileges \
  --pids-limit 512 \
  --memory 20g \
  --read-only \
  --tmpfs /tmp:rw,nosuid,nodev,size=2g \
  --mount type=bind,src="$WORKTREE_PATH",dst=/workspace,rw \
  --mount type=bind,src="$GIT_COMMON_DIR",dst="$GIT_COMMON_DIR",ro \
  --mount type=bind,src="$HOME/.config/opencode",dst=/root/.config/opencode,ro \
  --mount type=bind,src="$HOME/.local/share/opencode-container",dst=/root/.local/share/opencode,rw \
  --mount type=bind,src="$HOME/.cache/opencode-container",dst=/root/.cache/opencode,rw \
  --env-file "$HOME/.config/opencode/server.env" \
  --env GIT_OPTIONAL_LOCKS=0 \
  --workdir /workspace \
  localhost/opencode-agent:ubuntu26 \
  web --hostname 0.0.0.0 --port 4096
```

`--network host` wird hier bewusst verwendet: Dadurch erreicht der Container den nur auf 127.0.0.1 gebundenen Modellserver. Die Dateisystem- und Privilegienisolation bleibt bestehen, die Netzwerkisolation jedoch nicht. Der Agent kann grundsätzlich ausgehende Netzwerkverbindungen aufbauen; Downloads müssen daher weiterhin per Berechtigungsregel bestätigt werden. `GIT_OPTIONAL_LOCKS=0` verhindert optionale Index-Schreibversuche bei reinen Statusabfragen auf den read-only Git-Metadaten.

18.4 Container prüfen

```
podman ps
podman inspect opencode-task-001 \
  --format '{{json .Mounts}}' | jq
ss -ltnp | grep -E ':4096|:8080'
curl -I http://127.0.0.1:4096
```

19. Firewall und Zugriff aus dem LAN

19.1 LAN-Daten bestimmen

```
ip -4 -brief address
ip route
hostname -I
```

In den folgenden Befehlen <LAN-CIDR> und <LAN-IP> ersetzen. Beispiel: 192.168.178.0/24 und 192.168.178.50.

19.2 UFW vorsichtig konfigurieren

```
sudo ufw status verbose

# Nur falls der Rechner per SSH verwaltet wird:
sudo ufw allow OpenSSH

sudo ufw allow from <LAN-CIDR> to any port 4096 proto tcp \
  comment 'OpenCode Web LAN'
sudo ufw deny 4096/tcp

# Nur aktivieren, wenn bestehende Dienste berücksichtigt wurden:
sudo ufw enable
sudo ufw status numbered
```

19.3 Von einem zweiten LAN-Gerät testen

```
curl -I http://<LAN-IP>:4096
# Danach im Browser öffnen:
# http://<LAN-IP>:4096
```

Der Browser muss nach Benutzername und Passwort fragen. Der Modellport darf aus dem LAN nicht antworten:

```
curl --connect-timeout 3 http://<LAN-IP>:8080/health \
  && echo 'FEHLER: Modellport ist im LAN erreichbar' \
  || echo 'OK: Modellport nicht erreichbar'
```

Basic Auth ohne TLS Auf einem vertrauenswürdigen LAN ist dies für den ersten Ausbau vertretbar. Das Passwort und die Sitzung sind über unverschlüsseltes HTTP jedoch nicht gegen Mitschneiden in einem fremden oder kompromittierten Netz geschützt.

20. Agentischen Task sauber durchführen

20.1 Auftragsschablone

```
Arbeite ausschließlich im aktuellen Worktree und auf dem vorhandenen
Task-Branch. Lies zuerst AGENTS.md und die für den Task relevanten Dateien.

Ziel:
<KONKRETES ZIEL>

Akzeptanzkriterien:
1. <KRITERIUM 1>
2. <KRITERIUM 2>
3. Relevante Tests sind grün.

Vorgehen:
- Erstelle zuerst einen kurzen Plan und nenne Annahmen.
- Ändere nur notwendige Dateien.
- Führe gezielte Tests, Lint und Build aus.
- Kein git push und kein Zugriff auf Geheimnisse.
- Am Ende: Dateien, Tests, offene Risiken und nächste Schritte auflisten.
- Nicht committen, bis ich den Diff geprüft habe.
```

20.2 Arbeitsablauf

1. Mit Plan starten und falsche Annahmen korrigieren.
2. Berechtigungen einzeln prüfen. Bei Paketinstallationen und Netzwerkzugriff besonders skeptisch sein.
3. Nach dem ersten kleinen Änderungssatz `git diff --stat` und gezielte Tests verlangen.
4. Vor Abschluss vollständigen Diff außerhalb des Agenten prüfen.
5. Erst danach manuell committen oder einen ausdrücklich erlaubten Commit-Befehl bestätigen.

20.3 Review außerhalb des Containers

```
git -C "$WORKTREE_PATH" status --short
git -C "$WORKTREE_PATH" diff --stat
git -C "$WORKTREE_PATH" diff
git -C "$WORKTREE_PATH" diff --check
git -C "$WORKTREE_PATH" log --oneline --decorate -5
```

20.4 Commit und Integration

```
cd "$WORKTREE_PATH"
git add -p
git diff --cached
git commit -m '<AUSSAGEKRÄFTIGE COMMIT-NACHRICHT>'

cd <REPO>
git switch <STARTBRANCH>
git merge --no-ff <TASKBRANCH>
```

Vor dem Merge die projektspezifische Testsuite erneut im vertrauenswürdigen Haupt-Entwicklungsumfeld ausführen. Alternativ den Branch normal zu einem Remote pushen und über einen Pull Request reviewen; der Agent selbst erhält dafür weiterhin keine Zugangsdaten.

21. Sprach- und Projektabweichungen im Container

21.1 Java und Gradle/Maven

Das Beispielimage enthält OpenJDK 21. Wenn das Projekt eine andere JDK-Version benötigt, den Paketnamen im Containerfile anpassen. Abhängigkeiten in getrennten Volumes cachen:

```
podman volume create opencode-gradle
podman volume create opencode-m2

# Zusätzliche Optionen für podman run:
--mount type=volume,src=opencode-gradle,dst=/root/.gradle,rw \
--mount type=volume,src=opencode-m2,dst=/root/.m2,rw
```

21.2 Node.js

Für ein Projekt mit festgelegter Node-Version besser ein passendes Basisimage oder NodeSource im Containerfile verwenden. npm-Cache separat halten:

```
podman volume create opencode-npm
# Zusätzliche Option:
--mount type=volume,src=opencode-npm,dst=/root/.npm,rw
```

21.3 Python

Virtuelle Umgebungen innerhalb des Worktrees sind sichtbar und werden mit dem Worktree gelöscht. Besser reproduzierbar ist ein projektbezogenes `.venv` im Worktree:

```
python3 -m venv .venv
. .venv/bin/activate
python -m pip install -U pip
python -m pip install -r requirements.txt
```

21.4 Docker-basierte Tests

Docker-Socket nicht mounten	Ein Mount von /var/run/docker.sock gäbe dem Agenten praktisch Host-Root-Rechte. Für Integrationstests entweder einen getrennten CI-Runner nutzen oder einen bewusst isolierten, kurzlebigen Testhost bereitstellen.

22. Betrieb, Updates und Backups

22.1 Start und Stopp

```
# Modellserver
systemctl --user start llama-server.service
systemctl --user stop llama-server.service
systemctl --user status llama-server.service --no-pager

# OpenCode-Container
podman ps
podman stop -t 20 opencode-task-001
podman logs --tail 100 opencode-task-001
```

22.2 llama.cpp aktualisieren

```
systemctl --user stop llama-server.service
cd "$HOME/src/llama.cpp"
git fetch --all --tags
git pull --ff-only
cmake -S . -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=Release \
  -DGGML_VULKAN=ON \
  -DGGML_BLAS=ON \
  -DGGML_BLAS_VENDOR=OpenBLAS \
  -DLLAMA_CURL=ON
cmake --build build -j "$(nproc)" \
  --target llama-server llama-cli llama-bench
systemctl --user start llama-server.service
curl -fsS http://127.0.0.1:8080/health
```

22.3 OpenCode aktualisieren

```
# Host-Installation
opencode upgrade
opencode --version

# Containerimage
cd "$HOME/opencode-container"
podman build --pull=always \
  --build-arg OPENCODE_VERSION=latest \
  -t localhost/opencode-agent:ubuntu26 .
```

Nach Updates immer zuerst Smoke-Test, Tool-Call-Test und einen Wegwerf-Worktree verwenden. Autoupdate ist in der Konfiguration auf Benachrichtigung gesetzt, damit produktive Sitzungen nicht unbemerkt ihr Verhalten ändern.

22.4 Was sichern?

- Projekt-Branches und Commits über das normale Git-Backup beziehungsweise Remote.
- AGENTS.md und projektspezifische OpenCode-Befehle im Repository.
- Globale Konfiguration `~/.config/opencode`, aber Passwortdatei getrennt und sicher.
- OpenCode-Sitzungen unter `~/.local/share/opencode-container`, wenn ihre Historie wichtig ist.
- Das Modell muss nicht zwingend gesichert werden; es kann erneut geladen werden. Ein lokales Backup spart jedoch Downloadzeit.

23. Task abschließen und Worktree entfernen

23.1 Vor dem Entfernen

```
podman stop -t 20 opencode-task-001 2>/dev/null || true
git -C "$WORKTREE_PATH" status --short
git -C "$WORKTREE_PATH" log --oneline --decorate -5
git -C <REPO> branch --contains <TASKBRANCH>
```

Worktree erst entfernen, wenn alle gewünschten Änderungen committed, gesichert oder verworfen wurden und der Branch nicht mehr aktiv bearbeitet wird.

23.2 Sauber entfernen

```
cd <REPO>
git worktree remove "$WORKTREE_PATH"
git worktree prune
git worktree list
```

Branch optional löschen

```
git branch -d <TASKBRANCH>
```

Kein -D als Standard Die sichere Variante -d verweigert das Löschen nicht gemergter Branches. -D nur nach ausdrücklicher Prüfung verwenden, weil dadurch ungemergte Arbeit leichter verloren geht.

24. Fehlerdiagnose

24.1 Entscheidungsbaum

Fehler	Prüfung	Typische Lösung
llama-server startet nicht	journalctl, --version, Modellcache	Dienst manuell mit identischen Optionen starten
Vulkan OOM	Serverlog und GPU-Layer	12 auf 8/4 senken; Kontext 16k auf 8k
Nur CPU trotz Vulkan-Build	--list-devices und Startlog	RADV/Mesa prüfen; korrektes Gerät wählen
OpenCode sieht kein Modell	GET /v1/models und Modell-ID	Alias und provider/models-Schlüssel angleichen
Tool-Calls werden Prosa	Tool-Call-curl-Test	--jinja, aktuelles llama.cpp, Instruct-GGUF
OpenCode läuft in Timeout	chunkTimeout, lange Promptphase	Kontext verkleinern; Timeout erhöhen; Repo eingrenzen
LAN-Seite nicht erreichbar	ss, podman ps, UFW, LAN-IP	Portbindung und Subnetzregel korrigieren
Dateien gehören falschem Nutzer	ls -ln im Worktree	rootless Mapping prüfen; kein rootful podman

24.2 Kompakte Diagnosesammlung

```
vulkaninfo --summary
"$HOME/src/llama.cpp/build/bin/llama-server" --list-devices
systemctl --user status llama-server.service --no-pager
journalctl --user -u llama-server.service -n 200 --no-pager
curl -v http://127.0.0.1:8080/health
curl -s http://127.0.0.1:8080/v1/models | jq
opencode --version
podman info --format '{{.Host.Security.Rootless}}'
podman ps --all
podman logs --tail 200 opencode-task-001
ss -ltnp | grep -E ':4096|:8080'
sudo ufw status numbered
git -C "$WORKTREE_PATH" status --short
```

24.3 OpenCode-Providercache

OpenCode lädt Providerpakete dynamisch und cached sie. Bei API- oder Paketfehlern kann laut Troubleshooting-Dokumentation der Cache geleert werden. Zuerst den Container stoppen; danach nur das explizite Cache-Verzeichnis der Containerinstallation entfernen oder umbenennen:

```
podman stop -t 20 opencode-task-001 2>/dev/null || true
mv "$HOME/.cache/opencode-container" \
   "$HOME/.cache/opencode-container.backup-$(date +%Y%m%d-%H%M%S)"
mkdir -m 700 "$HOME/.cache/opencode-container"
```

25. Optionale Verbesserungen

25.1 OpenCode Web beim Login starten

Für einen festen Worktree kann später ein systemd-Benutzerdienst erstellt werden. Für wechselnde Task-Branches ist der manuelle Start sicherer, weil der freigegebene Pfad vor jedem Start sichtbar geprüft wird. Automatisierung erst einrichten, wenn ein dauerhaftes, klar benanntes Worktree-Verzeichnis feststeht.

25.2 Größeres Modell

Wenn die Geschwindigkeit akzeptabel ist, kann Qwen2.5-Coder 14B Q4_K_M getestet werden. Modell-ID, Alias und OpenCode-Konfiguration müssen gemeinsam geändert werden. Kontext zunächst auf 8192 setzen und GPU-Layer konservativ bei 4 bis 8 starten. 32 GB RAM reichen grundsätzlich, aber Prompt-Verarbeitung und Generierung werden deutlich langsamer.

25.3 Kleineres Modell

Bei zu hoher Latenz Qwen2.5-Coder 3B Instruct Q4_K_M verwenden. Die Bedienung bleibt gleich; nur `-hf`, Alias und OpenCode-Modell-ID ändern. Das 3B-Modell eignet sich besser für kleine Reparaturen, Tests und Dokumentation als für komplexe autonome Refactorings.

25.4 Open WebUI als zweiter Ausbau

Open WebUI wird in dieser Anleitung nicht installiert. Als späterer Ausbau sollte es nicht einfach direkt an llama.cpp gehängt und als Coding-Agent bezeichnet werden: Das wäre nur Modellchat. Für agentische Repository-Aufgaben wäre zusätzlich eine kontrollierte Workspace-/Computer-Schicht nötig. Bis dahin bleibt OpenCode Web die einzige Weboberfläche und vermeidet doppelte Sitzungs-, Rechte- und Modellkonfiguration.

26. Abnahmetest

Die Installation ist erst abgeschlossen, wenn alle Punkte erfüllt sind:

Nr.	Prüfung	Sollzustand
1	Ubuntu-Version	26.04.x LTS
2	vulkaninfo	RX 560 / RADV, nicht llvmpipe
3	llama-server --list-devices	Vulkan-Gerät sichtbar
4	GET /health	lokal erfolgreich
5	GET /v1/models	Alias qwen2.5-coder-7b-q4km
6	Tool-Call-Test	strukturierter tool_calls-Eintrag
7	OpenCode Terminal	liest, plant, fragt vor Änderungen
8	Worktree	eigener Task-Branch im eigenen Ordner
9	Podman	rootless=true; nur Worktree als Projekt-Mount
10	OpenCode Web	LAN erreichbar und passwortgeschützt
11	Modellport	aus dem LAN nicht erreichbar
12	Agent-Smoke-Test	Diff, Tests und Zusammenfassung vorhanden
13	Review	Diff außerhalb des Containers geprüft

Fertig

Wenn diese Liste vollständig grün ist, steht eine lokale, branch-isolierte OpenCode-Agentenumgebung mit LAN-Weboberfläche bereit. Die verbleibende Hauptgrenze ist die Modellqualität und Geschwindigkeit der vorhandenen Hardware, nicht die technische Integration.

27. Befehlsreferenz

27.1 Täglicher Start

```
systemctl --user start llama-server.service
curl -fsS http://127.0.0.1:8080/health

WORKTREE_PATH="$(realpath "$HOME/opencode-worktrees/<PROJEKTNAME>/task-001")"
GIT_COMMON_DIR="$(git -C "$WORKTREE_PATH" \
  rev-parse --path-format=absolute --git-common-dir)"
git -C "$WORKTREE_PATH" status --short

podman run --rm -it \
  --name opencode-task-001 \
  --network host --cap-drop all \
  --security-opt no-new-privileges \
  --pids-limit 512 --memory 20g --read-only \
  --tmpfs /tmp:rw,nosuid,nodev,size=2g \
  --mount type=bind,src="$WORKTREE_PATH",dst=/workspace,rw \
  --mount type=bind,src="$GIT_COMMON_DIR",dst="$GIT_COMMON_DIR",ro \
  --mount type=bind,src="$HOME/.config/opencode",dst=/root/.config/opencode,ro \
  --mount type=bind,src="$HOME/.local/share/opencode-container",dst=/root/.local/share/opencode,rw \
  --mount type=bind,src="$HOME/.cache/opencode-container",dst=/root/.cache/opencode,rw \
  --env-file "$HOME/.config/opencode/server.env" \
  --env GIT_OPTIONAL_LOCKS=0 \
  -w /workspace localhost/opencode-agent:ubuntu26 \
  web --hostname 0.0.0.0 --port 4096
```

27.2 Täglicher Abschluss

```
podman stop -t 20 opencode-task-001 2>/dev/null || true
git -C "$WORKTREE_PATH" status --short
git -C "$WORKTREE_PATH" diff --stat
systemctl --user stop llama-server.service
```

27.3 Zustände prüfen

```
systemctl --user is-active llama-server.service
curl -fsS http://127.0.0.1:8080/health
podman ps
git worktree list
sudo ufw status numbered
```

28. Quellen und Versionshinweise

Alle Webquellen wurden am 29. August 2026 geprüft. Die Befehle orientieren sich an den jeweils offiziellen Dokumentationen. Versionsabhängige Optionen vor einem späteren Upgrade mit `--help` und den Release Notes gegenprüfen.

Thema	Quelle
Ubuntu 26.04 LTS	https://documentation.ubuntu.com/release-notes/26.04/
OpenCode Installation	https://opencode.ai/docs/
OpenCode Web und Auth	https://opencode.ai/docs/web/
OpenCode Server	https://opencode.ai/docs/server/
OpenCode llama.cpp Provider	https://opencode.ai/docs/providers/
OpenCode Konfiguration	https://opencode.ai/docs/config/
OpenCode Berechtigungen	https://opencode.ai/docs/permissions/
OpenCode AGENTS.md	https://opencode.ai/docs/rules/
OpenCode Troubleshooting	https://opencode.ai/docs/troubleshooting/
llama.cpp Build	https://github.com/ggml-org/llama.cpp/blob/master/docs/build.md
llama.cpp Serveroptionen	https://github.com/ggml-org/llama.cpp/blob/master/tools/server/README.md
Qwen2.5-Coder 7B GGUF	https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct-GGUF
Podman Installation	https://podman.io/docs/installation
Podman run	https://docs.podman.io/en/latest/markdown/podman-run.1.html

28.1 Wesentliche Versionsannahmen

- Ubuntu 26.04 LTS ist seit April 2026 veröffentlicht; diese Anleitung verwendet Podman aus den Ubuntu-Paketquellen.
- OpenCode Web bindet standardmäßig nur an 127.0.0.1; für LAN-Zugriff wird 0.0.0.0 verwendet und OPENCODE_SERVER_PASSWORD gesetzt.
- OpenCode verwendet für lokale OpenAI-kompatible APIs den Provider `@ai-sdk/openai-compatible`.
- llama.cpp unterstützt Vulkan, OpenBLAS, Modellalias, quantisierten KV-Cache und OpenAI-kompatible Tool-Calls.
- Konfigurationsschlüssel können sich bei späteren OpenCode-Versionen ändern. Bei Warnungen zuerst die aktuelle offizielle Config- und Permissions-Seite prüfen.