Fastest Local LLM Setup: Ollama vs vLLM vs llama.cpp Real Benchmarks

February 3, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Ollama for easy local use — it's llama.cpp with a friendly wrapper, handles model management, and just works. llama.cpp directly for maximum control, CPU inference, or when you need features Ollama hasn't exposed yet. vLLM for production serving or when you need to handle multiple users — it's 23% faster than llama.cpp at 16 concurrent requests and scales to hundreds of users with PagedAttention. For multi-GPU setups, skip llama.cpp/Ollama entirely and use vLLM or ExLlamaV2. The common mistake is using Ollama for production APIs — it works, but vLLM will handle 4x the load on the same hardware.

📚 **More on this topic:** Ollama Troubleshooting Guide · Run Your First Local LLM · VRAM Requirements · Open WebUI Setup · Planning Tool

Three tools dominate local LLM inference: llama.cpp, Ollama, and vLLM. They solve different problems, and picking the wrong one either wastes your hardware or makes your life harder than it needs to be.

Here's the short version: Ollama is llama.cpp with training wheels. vLLM is for serving models to multiple users. llama.cpp is the engine under Ollama's hood that you can also drive directly.

This guide covers when each one makes sense, with actual benchmarks instead of vibes.

What Each Tool Actually Is

llama.cpp

The original. Georgi Gerganov wrote it to run LLaMA on a MacBook, and it's become the de facto standard for running quantized models on consumer hardware. Now maintained under ggml-org (builds hit b8200+ as of March 2026). Written in C++, it handles GGUF model files and runs on basically anything — CPUs, NVIDIA GPUs, AMD GPUs, Apple Silicon, WebGPU, even phones. March 2026 was a big month: MCP client support merged (tool calling via Model Context Protocol directly in llama-server), an autoparser that handles structured output for new models automatically, and multiple speed improvements for Qwen 3.5 and linear attention architectures.

Key insight: llama.cpp is an inference engine, not a user-facing application. You interact with it via command line or build applications on top of it.

Ollama

A friendly wrapper around llama.cpp (v0.17.7 as of March 2026). It handles model downloads, versioning, and provides a simple CLI and REST API. When you run `ollama run llama3`, it's using llama.cpp underneath to actually do the inference. Recent additions include dynamic context scaling (auto-fits context to your VRAM), `ollama launch` for integrating with Claude Code and OpenClaw, and cloud model support for offloading to remote hardware.

Key insight: Ollama adds convenience, not performance. The inference speed is llama.cpp speed with a small overhead from the Go-based server layer.

vLLM

A production inference engine built for serving (v0.17.0 as of March 7, 2026 — 699 commits from 272 contributors). Uses PagedAttention to efficiently handle multiple concurrent requests, tensor parallelism for multi-GPU setups, and continuous batching to maximize throughput. Written in Python. AMD ROCm is now a first-class platform alongside NVIDIA CUDA, with pre-built Docker images and 93% CI test pass rate. v0.17.0 upgrades to PyTorch 2.10, adds FlashAttention 4, matures Model Runner V2 with pipeline parallel and decode context parallel, adds a `--performance-mode` flag for simplified tuning, and brings Anthropic API compatibility. v0.16 added async scheduling with pipeline parallelism (30.8% throughput improvement), speculative decoding with structured outputs, and GGUF support on ROCm.

Key insight: vLLM is designed for "many users, one server" — the opposite of the desktop use case Ollama targets.

Head-to-Head Comparison

Feature	llama.cpp	Ollama	vLLM
Primary use	Engine/library	Desktop local AI	Production serving
Setup difficulty	Medium	Easy	Medium-Hard
Model format	GGUF	GGUF (via llama.cpp)	HuggingFace, GPTQ, AWQ, GGUF
Single-user speed	Fastest	~Same as llama.cpp	Slightly slower

Feature	llama.cpp	Ollama	vLLM
Multi-user throughput	Poor	Poor	Excellent
Multi-GPU support	Improved (RPC, parallel loading)	Limited	Excellent
CPU inference	Yes	Yes	Limited (ARM optimized in v0.16)
AMD ROCm	Yes	Yes	First-class (v0.16+)
Memory efficiency	Good	Good (dynamic context scaling)	Best (PagedAttention)
MCP / Tool calling	Native (llama-server)	Via external tools	Via external tools
Structured output	Autoparser (March 2026)	Via llama.cpp	JSON mode
API	HTTP server optional	REST API built-in	OpenAI-compatible
Model management	Manual (or llama-swap)	Automatic	Manual

Performance Benchmarks

Real numbers from a Qwen 2.5 3B on RTX 4090:

Single User (You Alone)

Scenario	llama.cpp	vLLM	Winner
2K prompt + 256 gen	90.0s	94.4s	llama.cpp (4.7% faster)
30K prompt + 256 gen	231.6s	245.8s	llama.cpp (5.8% faster)

For single-user desktop use, llama.cpp (and therefore Ollama) is slightly faster. The difference is small — under 6% — but it's real.

Multiple Users (Concurrent Requests)

Scenario	llama.cpp	vLLM	Winner
16 concurrent, 2K prompt	265.5s	215.3s	vLLM (23% faster)

Scenario	llama.cpp	vLLM	Winner
16 concurrent, 24K prompt	3640.7s	3285.3s	vLLM (11% faster)

Once you have multiple users hitting the same model, vLLM pulls ahead significantly. At 16 concurrent requests, it's 23% faster. At higher concurrency, the gap widens further.

The PagedAttention Advantage

vLLM's secret weapon is PagedAttention, which eliminates 60-80% of the memory waste from KV cache fragmentation. In practice:

- Standard implementation on 24GB VRAM: ~32 concurrent sequences
- vLLM with PagedAttention: **~128 concurrent sequences**

Same hardware, 4x the capacity. That's why production deployments use vLLM.

Ollama: When to Use It

Use Ollama when:

- You're running models for yourself on your own machine
- You want model management handled automatically
- You're building local apps that need an LLM API
- You don't want to think about quantization formats or compile flags
- You're using Open WebUI or similar frontends

Don't use Ollama when:

- You're serving multiple users simultaneously
- You need multi-GPU tensor parallelism
- You need maximum control over inference parameters
- You're building a production API

Ollama's Real Value

Ollama's killer feature isn't performance — it's convenience. Compare the workflows:

With Ollama:

```
ollama run qwen3.5:9b
```

With llama.cpp directly:

```
# Find and download GGUF file manually
# Figure out the right quantization
./llama-cli -m ./models/qwen3.5-9b-q4_k_m.gguf \
  -c 8192 -n 256 --temp 0.7 -p "Your prompt here"
```

Ollama handles model discovery, downloads, versioning, and provides a consistent interface. For desktop use, that convenience is worth the tiny overhead.

Ollama Limitations

- **No tensor parallelism:** Multi-GPU support exists but doesn't split models across GPUs efficiently
- **Limited quantization control:** You get what's in the Ollama library, can't easily use custom quants
- **Server overhead:** The Go layer adds some latency, noticeable at very high request rates
- **Batching:** Handles concurrent requests poorly compared to vLLM
- **Request serialization bug:** As of March 2026, concurrent requests to different loaded models can queue for 50+ seconds even when both models are in VRAM

---

llama.cpp: When to Use It Directly

Use llama.cpp when:

- You need CPU inference or heavy CPU offloading
- You want maximum control over quantization and inference settings
- You're building something that needs to embed inference directly
- You're on unusual hardware (ARM, older GPUs, etc.)
- You need features Ollama hasn't exposed yet

Don't use llama.cpp when:

- You just want to chat with a model (use Ollama)
- You need high-concurrency serving (use vLLM)
- You're doing multi-GPU inference (use vLLM or ExLlamaV2)

llama.cpp's Unique Strengths

MCP tool calling (March 2026) — llama-server now has a built-in MCP client with full support for tools, resources, and prompts. Run `llama-server` with the `--webui-mcp-proxy` flag and you get an agentic loop directly in the web UI — connect any MCP server, and the model can call tools, browse resources, and use prompt templates. This is a major feature gap closer vs Ollama, which still requires external tools for MCP. The implementation includes a CORS proxy on the backend side, server capability detection, and streaming stats for tool call chains.

Autoparser for structured output (March 2026) — A complete rewrite of the parser architecture landed on March 6. Instead of maintaining separate parsers for each model's tool-calling format, the autoparser uses differential analysis to automatically detect template markers — reasoning blocks, content blocks, tool call syntax — from any chat template. New models get structured output support out of the box without needing custom parser code. Only two models (Ministral and GPT-OSS) still need dedicated parsers. This matters for agentic use cases where tool calling previously broke with many model templates.

CPU offloading — llama.cpp is the only tool that gracefully handles models too large for your VRAM by offloading layers to system RAM. Yes, it's slow (~1 tok/s for huge models), but it works. vLLM can't do this at all.

Hardware compatibility — Runs on NVIDIA, AMD, Intel, Apple Silicon, WebGPU, OpenCL, and even pure CPU. If you have unusual hardware, llama.cpp probably supports it. Recent builds added CUDA async copy for faster GPU transfers and reduced synchronization overhead between tokens.

Qwen 3.5 speed improvements — A compute graph rework in recent builds cut unnecessary tensor copies and improved kernel selection for Qwen 3.5 and similar architectures. On dual RTX 6000 Ada GPUs running Qwen3 Coder Next 80B Q8_0, token generation jumped from ~88 tok/s to over 118 tok/s — a 30%+ improvement. CUDA graphs are now enabled for Gated DeltaNet architectures. Separately, optimizations for Kimi Linear's delta attention architecture improved prompt processing by ~30% for linear attention models, which benefits Qwen 3.5's hybrid linear attention layers too.

RPC distributed inference — llama.cpp now supports splitting model layers across remote machines via its RPC server. You can offload layers to a second box's GPU over the network. It's

not as polished as vLLM's tensor parallelism, but it works for setups where you have spare GPUs on different machines. Parallel model loading across GPU contexts also landed in March 2026, speeding up multi-GPU startup.

Quantization ecosystem — The GGUF format and llama.cpp's quantization tools are the standard. K-quants (Q4_K_M, Q5_K_M) and I-quants give you fine-grained control over the size/quality tradeoff:

Quant	Bits/Weight	Perplexity Impact	Use Case
Q4_K_M	~4.5 bpw	+0.05 ppl	Default recommendation
Q5_K_M	~5.3 bpw	+0.01 ppl	Quality sweet spot
Q3_K_M	~3.7 bpw	+0.66 ppl	VRAM constrained
Q6_K	~6.0 bpw	+0.004 ppl	Near-lossless
IQ2_XS	~2.3 bpw	Higher	Extreme compression

Speculative decoding — Use a small draft model to speed up generation. Reports of ~12 tok/s vs 8-9 tok/s baseline with good draft model matches. Qwen 3.5 0.8B makes a strong draft model for Qwen 3.5 27B. vLLM also supports speculative decoding now (v0.16+, including with structured outputs), so this is no longer a llama.cpp exclusive — but llama.cpp's implementation is more mature for GGUF models.

Basic llama.cpp Usage

```
# Run inference
./llama-cli -m model.gguf -p "Your prompt" -n 256

# Start a server
./llama-server -m model.gguf -c 4096 --host 0.0.0.0 --port 8080

# With GPU layers (offload 35 layers to GPU)
./llama-cli -m model.gguf -ngl 35 -p "Your prompt"

# Enable flash attention
./llama-cli -m model.gguf -fa -p "Your prompt"
```

vLLM: When to Use It

Use vLLM when:

- You're serving a model to multiple users
- You need an OpenAI-compatible API
- You have multi-GPU setups and want tensor parallelism
- Throughput and cost efficiency matter more than setup simplicity
- You're running a production inference service

Don't use vLLM when:

- You're the only user (Ollama is simpler)
- You need CPU inference or heavy CPU offloading
- You're on consumer AMD GPUs (ROCm support targets MI300/MI350 datacenter cards, not RDNA)
- You want the simplest possible setup

vLLM's Production Numbers

The throughput improvements are dramatic:

- **14-24x higher throughput** vs standard HuggingFace Transformers
- **3-10x improvement** from continuous batching alone
- **30.8% additional throughput** from async scheduling + pipeline parallelism (v0.16)
- **FlashAttention 4** integration for next-gen attention performance (v0.17.0)
- `--performance-mode {balanced, interactivity, throughput}` flag for simplified tuning (v0.17.0)
- Stripe reduced inference costs by **73%** using vLLM

Concrete example: Serving Mixtral-8x7B on 2x A100 at 100 requests/second:

- vLLM P50 latency: 180ms
- Standard serving P50 latency: 650ms

At scale, vLLM isn't optional — it's required to make the math work.

What's new in v0.17.0 (March 7, 2026)

The v0.17.0 release is substantial — 699 commits from 272 contributors. The headline features:

- **PyTorch 2.10 upgrade** — breaking change for environment dependencies, so plan your upgrade
- **FlashAttention 4** — next-gen attention backend
- **Model Runner V2 maturation** — pipeline parallel, decode context parallel, Eagle3 speculative decoding with CUDA graphs
- **Full Qwen 3.5 support** — GDN (Gated Delta Networks), FP8 quantization, MTP speculative decoding, reasoning parser
- **`--performance-mode` flag** — choose `balanced`, `interactivity`, or `throughput` instead of tuning individual knobs
- **Anthropic API compatibility** — thinking blocks, `count_tokens` API, `tool_choice=none`
- **Weight offloading V2** — prefetching hides onloading latency, selective CPU offloading without doubling pinned memory
- **Elastic expert parallelism** — dynamic GPU scaling for MoE models
- **Quantized LoRA adapters** — load QLoRA adapters directly

Known issue: CUDA 12.9+ can trigger a `CUBLAS_STATUS_INVALID_VALUE` error from a library mismatch. Fix by unsetting `LD_LIBRARY_PATH` or pinning the CUDA wheel version.

vLLM Setup

```
# Install (NVIDIA)
pip install vllm

# Install (AMD ROCm — or use the pre-built Docker image)
# docker pull vllm/vllm-openai:latest-rocm

# Start server
vllm serve meta-llama/Llama-3.1-8B-Instruct

# With tensor parallelism (multi-GPU)
vllm serve meta-llama/Llama-3.1-70B-Instruct --tensor-parallel-size 4

# With performance mode (new in v0.17.0)
vllm serve model-name --performance-mode throughput

# With quantization
vllm serve model-name --quantization awq
```

```
# Speculative decoding (v0.16+ — works with structured outputs)
vllm serve Qwen/Qwen3.5-27B --speculative-model Qwen/Qwen3.5-0.8B
```

The API is OpenAI-compatible, so existing code that calls OpenAI can point at your vLLM server instead:

```
from openai import OpenAI

client = OpenAI(base_url="http://localhost:8000/v1", api_key="none")
response = client.chat.completions.create(
    model="meta-llama/Llama-3.1-8B-Instruct",
    messages=[{"role": "user", "content": "Hello!"}]
)
```

vLLM Hardware Requirements

vLLM is GPU-hungry. NVIDIA CUDA and AMD ROCm (MI300/MI350 datacenter cards) are both first-class platforms as of v0.16. Consumer AMD GPUs (RDNA) are not well supported.

Model Size	Minimum VRAM	Recommended
7B	16 GB	24 GB
13B	24 GB	40 GB
30B+	40 GB+	80 GB+

For consumer GPUs, an RTX 3090 or 4090 can run 7-13B models in vLLM. Larger models need professional cards or multi-GPU setups.

Multi-GPU: Critical Guidance

Ollama still can't do real multi-GPU tensor parallelism. If you need to split a model efficiently across 2+ GPUs, use vLLM or ExLlamaV2.

llama.cpp's multi-GPU story has improved — parallel model loading across GPU contexts landed in March 2026, and RPC-based distributed inference lets you offload layers to remote GPUs. It's

usable for hobbyist multi-GPU setups but still not in the same league as vLLM's tensor parallelism for throughput.

For multi-GPU setups:

- **vLLM** — Best for FP16/BF16 models, excellent tensor parallelism, now with AMD ROCm support
- **ExLlamaV2** — Best for quantized models (EXL2 format), good multi-GPU support
- **llama.cpp** — Workable via layer splitting or RPC, but not optimized for throughput

If you have 2+ datacenter GPUs and want maximum throughput, vLLM. If you have 2 consumer GPUs and want to load a bigger model, llama.cpp's layer splitting works.

Other Tools Worth Knowing

ik_llama.cpp

A CPU-optimized fork of llama.cpp by Iwan Kawrakow that dramatically outperforms mainline on CPU inference. On a Ryzen 7950X running LLaMA-3.1-8B, prompt processing is 3-5x faster across most quantization types (IQ3_S hits 5.18x). Token generation improves 1.5-2x. On Apple M2 Max, the gains are even larger — IQ3_S prompt processing is 7x faster than mainline. The project's own benchmark summary: "On the Ryzen-7950X the slowest quantization type in ik_llama.cpp is faster than the fastest type in llama.cpp for prompt processing."

Use when: You're running CPU-only inference or heavy CPU offloading and want maximum speed. Worth the extra build step.

Skip when: You're GPU-bound. The fork's advantages are CPU-specific.

llama-swap

A lightweight Go proxy (2.7K stars) that hot-swaps between llama-server instances based on which model an API request targets. Zero dependencies, single binary. Think of it as model management for llama.cpp without Ollama's opinions — you get OpenAI and Anthropic API compatibility, TTL-based auto-unloading, model groups for parallel loading, and a web UI. Originally built for llama-server but works with any OpenAI-compatible backend including vLLM and SGLang.

Use when: You want Ollama's model-switching convenience with llama.cpp's raw control. Power users managing multiple models on constrained hardware.

Skip when: Ollama's defaults work for you. llama-swap is the "I know what I'm doing" option.

ExLlamaV2

The speed king for quantized inference on NVIDIA GPUs. Uses EXL2 format (similar to GPTQ but better quality). Benchmark: 120-150 tok/s on RTX 4090 for 13B models vs 80-100 tok/s for llama.cpp.

Use when: You need maximum speed for quantized models on NVIDIA, especially multi-GPU.

Skip when: You need CPU offloading or non-NVIDIA hardware.

kobold.cpp

A fork of llama.cpp focused on creative writing and roleplay. Adds features like soft prompts, memory management, and UI tailored for story generation.

Use when: Fiction writing, roleplay, story continuation.

text-generation-inference (TGI)

HuggingFace's production inference server. Similar to vLLM in goals, different implementation. Good HuggingFace integration.

Use when: You're already deep in the HuggingFace ecosystem.

Decision Flowchart

Are you the only user?

- Yes → **Ollama** (or llama.cpp if you need more control)
- No → Continue

Do you have multiple GPUs you want to use together?

- Yes → **vLLM** (or ExLlamaV2 for quantized models)
- No → Continue

Are you serving >5 concurrent users?

- Yes → **vLLM**
- No → **Ollama** is probably fine

Do you need CPU inference or offloading?

- Yes → **llama.cpp** (only real option)
- No → See above

Are you building a production API?

- Yes → **vLLM**
- No → **Ollama**

Using Multiple Tools Together

You don't have to pick just one. A common setup:

1. **Ollama for development** — Quick testing, trying new models, local experimentation
2. **vLLM for production** — Serving the final model to users

Or for power users:

1. **Ollama for daily use** — Chat, quick queries, Open WebUI
2. **llama.cpp directly** — When you need specific quantization or settings Ollama doesn't expose

Ollama → llama.cpp Migration

If you've been using Ollama and want to try llama.cpp directly, your models are already downloaded. Find them at:

- macOS: `~/.ollama/models/`
- Linux: `~/.ollama/models/` or `/usr/share/ollama/.ollama/models/`
- Windows: `C:\Users\<user>\.ollama\models\`

The actual GGUF files are in `blobs/` with hash names. You can use them directly with llama.cpp.

Qwen 3.5 and Gated DeltaNet: Runtime Support

Qwen 3.5 uses a hybrid architecture called Gated DeltaNet that mixes standard attention with linear attention layers. This is a new architecture, not a new model on an old one, and it requires explicit runtime support. Not every tool handled it at launch.

Current support (March 2026):

Runtime	Qwen 3.5 Support	Notes
Ollama	v0.17.4+	Native Gated DeltaNet support. v0.17.5+ needed for GPU/CPU split stability. v0.17.7 for full tool calling + thinking levels.
llama.cpp	Supported	Works via GGUF. Unsloth quants available on HuggingFace. Build from recent source (b8200+) for best results.
vLLM	v0.17.0+	Full GDN support, FP8 quantization, MTP speculative decoding, reasoning parser. Recommended for multi-GPU production.
SGLang	Supported	Recommended by the Qwen team for production serving. Supports multi-token prediction (MTP), which can speed up generation.

The MoE + linear attention hybrid in Qwen 3.5 changes the performance math compared to standard transformers. The 35B-A3B variant (35 billion total parameters, 3 billion active per token) runs at roughly 112 tok/s on an RTX 3090 through Ollama. That's faster than most dense 7B models on the same card, because inference speed depends on active parameters, not total parameters. The KV cache also stays smaller at long context lengths thanks to the linear attention layers — the 9B model can hold 32K+ context on an 8GB card where a standard transformer 9B would choke.

If you're running Qwen 3.5 in production with multiple users, SGLang is worth evaluating alongside vLLM. The Qwen team specifically recommends it, and multi-token prediction can give a meaningful throughput bump on models that support it.

Bottom Line

The choice is simpler than it looks:

Ollama — You're running models locally for yourself. It just works. Start here.

llama.cpp — You need control Ollama doesn't give you, or you need CPU inference/offloading. Power user territory.

vLLM — You're serving models to other people, or you have multi-GPU setups. Production territory.

The mistake to avoid: using Ollama for production APIs when vLLM would handle 4x the load. Ollama is fantastic for what it's designed for — desktop local AI. It's not designed for serving hundreds of concurrent users, and it shows in benchmarks.

For most readers of this site — hobbyists running models on their own hardware — Ollama is the right answer. You'll know when you've outgrown it.

```
# Start here
ollama run qwen3.5:9b

# Graduate to this when you need to
vllm serve Qwen/Qwen3.5-9B-Instruct
```

Updated March 8, 2026 for Ollama v0.17.7, vLLM v0.17.0 (FlashAttention 4, PyTorch 2.10), and llama.cpp with MCP tool calling and autoparser.
