LM Studio vs llama.cpp: Why Your Model Runs Slower in the GUI

March 5, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** LM Studio runs llama.cpp internally, but users consistently report 30-50% slower inference compared to running llama.cpp compiled from source. On an RTX Pro 6000, one benchmark showed 86 tok/s in LM Studio vs 180 tok/s in llama.cpp with the same model. The main causes: LM Studio bundles an older llama.cpp version that misses recent optimizations, conservative default settings (larger context windows, different batch sizes), and minor GUI/IPC overhead. For Qwen 3.5 35B-A3B specifically, LM Studio has additional compatibility issues that cause broken output. You can close the gap by adjusting LM Studio's settings, but you can't fully eliminate it. If raw speed matters, compile llama.cpp from source. If convenience matters more, LM Studio's speed is fine for interactive chat.

📚 **More on this topic:** llama.cpp vs Ollama vs vLLM · Why Is My LLM So Slow? · VRAM Requirements Guide · LM Studio Tips

---

You download Qwen 3.5 35B-A3B in LM Studio, run it, get 40 tok/s. Not bad. Then you compile llama.cpp from source, load the same GGUF, and get 90 tok/s. Same hardware, same model, same quantization. What happened?

This confuses people because LM Studio literally uses llama.cpp as its inference engine. Same code, different speed. The reasons are mundane, but they're fixable once you know what to look for.

The Gap Is Real

These numbers come from various community benchmarks. Your mileage varies with hardware and model, but the pattern is consistent:

Hardware	Model	LM Studio	llama.cpp (source)	Gap
RTX Pro 6000	GPT-OSS 120B Q4	~86 tok/s	~180 tok/s	52% slower
RTX 3090	Qwen 3.5 35B-A3B Q4	~60 tok/s	~111 tok/s	46% slower

Hardware	Model	LM Studio	llama.cpp (source)	Gap
M3 Ultra	Gemma 3 27B	~22 tok/s	~33 tok/s	33% slower
RTX 4070 Ti	Qwen 2.5 7B Q4	~45 tok/s	~65 tok/s	31% slower

Sources: HN thread on LM Studio/llama.cpp perf, Arsturn speed test, llama.cpp GPU benchmarks.

The gap ranges from 30% to 50% depending on hardware and model. Bigger on high-end GPUs where the overhead becomes a larger fraction of total inference time, smaller on budget hardware where the model itself is the bottleneck.

Why LM Studio Is Slower (Three Reasons)

LM Studio is llama.cpp. But it's not your llama.cpp. That distinction matters.

1. Bundled runtime version lag

This is the biggest factor. LM Studio ships with a specific llama.cpp build baked into the application. The llama.cpp project pushes performance-relevant commits weekly, sometimes daily. Flash Attention improvements, KV cache rework, architecture-specific fixes for models like Qwen 3.5's MoE routing. The pace is relentless.

When you compile llama.cpp from source, you get all of that. When you use LM Studio, you get whatever build the LM Studio team bundled in their last release. That lag can be anywhere from a few days to several weeks. On a rapidly evolving model like Qwen 3.5 35B-A3B, where early llama.cpp builds had MoE routing issues that were fixed in later commits, the version gap hits especially hard.

This same problem affects Ollama. Both tools wrap llama.cpp but neither tracks HEAD in real time.

2. Default settings are conservative

LM Studio picks safe defaults that work across a wide range of hardware. When you compile llama.cpp and run it yourself, you tune for your specific GPU:

Setting	LM Studio default	Optimized llama.cpp
Context length	4096-8192	Set to what you need (2048 for chat)

Setting	LM Studio default	Optimized llama.cpp
Batch size	Auto (conservative)	Tuned per GPU
Flash Attention	Sometimes off	Enabled
Thread count	Auto-detected	Pinned to performance cores
GPU layers	Auto	All layers offloaded manually
Mmap	Enabled	Disabled on fast SSDs (can be faster)

Context length alone explains a chunk of the gap. A 4096-context window uses 2x the KV cache memory of a 2048 window. On VRAM-constrained cards, that pushes you closer to the edge where performance degrades. LM Studio doesn't know you only need short conversations, so it allocates more than you might need.

3. UI and IPC overhead

The smallest factor, but it's real. LM Studio runs an Electron-based UI that communicates with the llama.cpp backend through inter-process communication. Each token generation involves:

1. The llama.cpp process generates a token
2. It sends the token through IPC to the UI process
3. The UI process renders it in the chat window
4. Repeat

This adds microseconds per token. At 100+ tok/s, those microseconds add up to a few percentage points. Not the main cause, but it's there.

On Apple Silicon specifically, there's an additional wrinkle: Metal shader compilation can behave differently between a standalone llama.cpp binary and one embedded in an Electron app's process space. I've seen reports of GPU utilization differences in Activity Monitor between the two.

The Qwen 3.5 Problem (It's Worse Than Usual)

If you're running Qwen 3.5 models in LM Studio or Ollama right now, the gap might be worse than the usual 30-50%. Both tools have documented compatibility issues with Qwen 3.5's architecture:

- Chain-of-thought loops that never terminate

- Broken tool-call formatting
- Garbage output on tasks that work fine in llama.cpp directly

The root cause: Qwen 3.5 relies on the `presence_penalty` parameter to end its thinking sequences, and neither Ollama nor LM Studio handle this correctly yet. The llama.cpp server does, because it exposes the full parameter set.

If Qwen 3.5 35B-A3B is giving you problems in LM Studio, try llama.cpp's server directly before blaming the model. The model is good. The wrapper isn't ready for it yet.

Where the Gap Matters Most

The gap doesn't hit every setup equally:

Your hardware	Gap matters?	Why
8GB VRAM (RTX 4060, 3060)	Less	You're already VRAM-limited. The bottleneck is memory, not software overhead
12-16GB VRAM (RTX 4070 Ti, 4080)	Moderate	You have enough VRAM that software efficiency starts mattering
24GB VRAM (RTX 3090, 4090)	More	Fast hardware amplifies the overhead percentage
48GB+ (dual GPU, Mac Ultra)	Most	You're leaving serious performance on the table
Apple Silicon (M1-M4)	Depends	Metal path differences between standalone and embedded can be significant

The general rule: the faster your hardware, the more the wrapper overhead costs you in absolute terms. On an RTX 4060 running a 7B model at 20 tok/s, a 30% gap means you lose 6 tok/s. Noticeable but livable. On an RTX 4090 running the same model at 100 tok/s, that same 30% gap is 30 tok/s. That's the difference between snappy and instant.

How to Benchmark Your Own Setup

Don't trust anyone else's numbers for your hardware. Measure it yourself. It takes 10 minutes.

Benchmark in LM Studio

Load your model, open the chat, and ask it to write a 500-word essay on any topic. Watch the status bar at the bottom, which shows tok/s in real time. Note the generation speed once it stabilizes (ignore the first few tokens, which include prompt processing).

Better method: use LM Studio's built-in server mode and hit the API with curl:

```
time curl -s http://localhost:1234/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{"model": "qwen3.5-35b-a3b", "messages": [{"role": "user", "content": "Write a 500-word es
  | jq '.usage'
```

Benchmark in llama.cpp

Build from source (see our llama.cpp guide for details), then use the built-in benchmark:

```
./build/bin/llama-bench \
  -m your-model.gguf \
  -t $(nproc) \
  -ngl 99 \
  -fa 1
```

This gives you prompt processing (pp) and token generation (tg) speeds separately, which is more useful than a single tok/s number. The `-fa 1` enables Flash Attention.

For an apples-to-apples comparison, also run the server and hit it with the same curl command:

```
./build/bin/llama-server \
  -m your-model.gguf \
  -t $(nproc) \
  -ngl 99 \
  -fa \
  -c 2048 \
  --host 127.0.0.1 \
  --port 8080
```

```
time curl -s http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{"model": "test", "messages": [{"role": "user", "content": "Write a 500-word essay about c
  | jq '.usage'
```

What to compare

- **Token generation speed** (tg tok/s): This is what you feel during chat
- **Prompt processing speed** (pp tok/s): Matters for long conversations and RAG
- **VRAM usage**: Check with `nvidia-smi` or Activity Monitor. LM Studio sometimes allocates more

Run each test 3 times and average the results. First runs are always slower due to model loading and shader compilation.

Closing the Gap in LM Studio

You don't have to leave LM Studio to claw back some speed. The biggest win is context length: go to Settings > Model defaults > Context Length and drop it to 2048 or 4096 unless you actually need long conversations. A 4096-context window uses 2x the KV cache memory of 2048, and on tight VRAM that's the difference between full GPU offloading and partial CPU spillover.

Next, check that Flash Attention is enabled in the model settings panel. Some LM Studio versions ship with it off for compatibility reasons. Turning it on costs nothing and helps on every supported GPU.

Verify GPU offloading too. LM Studio's auto-detection occasionally leaves a few layers on CPU, which bottlenecks the entire generation pipeline. Force all layers to GPU if your VRAM can handle it.

And update LM Studio itself. Right after a new release, the bundled llama.cpp build is close to HEAD. Two months later, the gap can be massive. Keep it current.

These tweaks won't fully close the gap, but I've seen them shrink it from 50% to around 15-20%.

When LM Studio Wins Anyway

Raw tok/s isn't everything. LM Studio does things that make the speed tax worth paying.

Model management alone justifies it for a lot of people. Browse, download, switch between models, all without touching HuggingFace or juggling GGUF files. Chat history persists across sessions with branching and search. llama.cpp gives you a blank terminal every time you start it.

Then there's the settings UI: temperature, top-p, repeat penalty, system prompts, all adjustable mid-conversation without restarting anything. Side-by-side model comparison for picking between quantization levels. One-click API server that gives you an OpenAI-compatible endpoint.

For interactive chat where you're reading and thinking between messages, the difference between 40 tok/s and 90 tok/s is invisible. You're not reading at 90 tokens per second. The gap matters for batch processing, long generations, and automated pipelines. For conversation, it doesn't.

My Take

I use both. LM Studio when I'm trying a new model or just want to chat without thinking about flags. llama.cpp compiled from source when I'm running Qwen 3.5 35B-A3B for coding tasks and I want every token as fast as the GPU can push it, or when LM Studio hasn't caught up to a new model architecture yet.

The 30-50% gap is real, but it's the price of not thinking about command-line flags. If your workflow is "load model, chat, close," LM Studio is fine and trying to optimize past it is procrastination disguised as productivity. If your workflow involves sustained generation or automated pipelines, compile llama.cpp and don't look back.

For the full comparison of inference tools including Ollama and vLLM, see our llama.cpp vs Ollama vs vLLM guide. If your model is running slower than expected regardless of tool, check our slow LLM troubleshooting guide. And for figuring out whether your model fits in VRAM in the first place, see the VRAM requirements guide.

Free guides for running AI locally