Fix your 10-minute Ollama responses on M4

Your Mac Mini M4 with 32GB RAM is perfectly capable of running Qwen3-14B — the hardware isn't the problem. **The ~10 minute response times stem from a performance death spiral: thinking mode generates thousands of hidden tokens per response, rapidly filling your 30K context window, triggering 26 compactions that each force re-evaluation of ~15K tokens.** This is fixable with a few configuration changes that should bring response times from minutes down to seconds.

The good news: Qwen3-14B at Q4_K_M quantization (9.3 GB) only uses ~19–21 GB total at 30K context, well within your 32GB. The bad news: Ollama's default context for your RAM tier is likely 32,768 tokens, GitHub and the combination of thinking mode + large context + compaction cascades is what's killing you. Here's exactly how to fix it, ranked by impact.

The compaction death spiral is your primary bottleneck

Ollama uses llama.cpp's "context shifting" mechanism — this is not summarization, it's brute-force truncation. When total tokens hit the `num_ctx` limit, Ollama discards the oldest half of context and **re-evaluates the remaining ~15K tokens from scratch**. With 26 compactions, your conversation triggered this re-evaluation cycle 26 times. Each re-evaluation of 15K tokens on a 14B model at 120 GB/s bandwidth takes significant time — potentially 30–60+ seconds per compaction. Multiply that by 26, and you've found your 10 minutes.

The root cause is almost certainly Qwen3's thinking mode. Each response generates an internal `<think>...</think>` block Hugging Face that can run **2,000–10,000+ tokens** before producing the visible answer. Qwen Those thinking tokens consume context just like regular tokens. A single thinking response at 5K tokens eats ~15% of a 33K context window. After a few exchanges, the window fills, triggers a compaction, the model re-evaluates, generates another long thinking response, fills the window again — a cascading loop.

Reducing `num_ctx` to 8,192 and disabling thinking mode are the two highest-impact fixes. Together, they eliminate the compaction cascade entirely for normal conversations.

Optimal settings for your M4 32GB hardware

The M4 base chip has **120 GB/s memory bandwidth** Apple +2 — this is the hard ceiling for token generation speed, not compute cores or RAM capacity. For Qwen3-14B at Q4_K_M (9.3 GB), the theoretical maximum is ~12.9 tokens/sec, with real-world performance around **8–12 tok/s**. That's perfectly usable for interactive chat. Here's the complete optimization stack, ordered by impact:

Step 1 — Set environment variables (biggest wins):

```
launchctl setenv OLLAMA_FLASH_ATTENTION 1
launchctl setenv OLLAMA_KV_CACHE_TYPE "q8_0"
launchctl setenv OLLAMA_NUM_PARALLEL 1
launchctl setenv OLLAMA_MAX_LOADED_MODELS 1
launchctl setenv OLLAMA_KEEP_ALIVE "30m"
```

Restart Ollama after setting these. Flash attention reduces memory usage and improves speed with zero quality loss. KV cache quantization at Q8_0 **halves your KV cache memory** Localllm +2 (from ~4.6 GB to ~2.3 GB at 30K context) with negligible quality impact Localllm (~0.004 perplexity increase). Flash attention must be enabled for KV cache quantization to work. smcleod smcleod.net Setting `NUM_PARALLEL=1` prevents Ollama from allocating multiple KV cache slots, which would multiply memory usage. Medium

Step 2 — Create an optimized Modelfile:

```
FROM qwen3:14b
PARAMETER num_ctx 8192
PARAMETER num_predict 4096
PARAMETER temperature 0.7
PARAMETER top_p 0.8
PARAMETER top_k 20
PARAMETER repeat_penalty 1.05
SYSTEM You are a helpful assistant. /no_think
```

Then run `ollama create qwen3-fast -f Modelfile`. The `num_predict 4096` caps output length, preventing runaway generation. The `/no_think` in the system prompt disables chain-of-thought by default. Qwen The sampling parameters follow Qwen's official recommendations. Unsloth AI

Step 3 — Verify GPU offloading:

```
ollama ps
```

You should see `100% GPU` in the PROCESSOR column. If it shows a CPU/GPU split, the model is partially offloaded and you need to reduce `num_ctx` or close background applications. Also check the logs: `grep -E "flash_attn|type_k|offloaded" ~/.ollama/logs/server.log | tail -20` should show `flash_attn = 1`, `type_k = 'q8_0'`, and `offloaded 41/41 layers to GPU`.

Context window sizing: why 8K beats 30K for your use case

KV cache memory scales linearly with context length. Localllm localllm Here's what your M4 32GB actually looks like at different context sizes with Qwen3-14B Q4_K_M:

Context	KV cache (FP16)	KV cache (Q8_0)	Total with OS	Headroom
4,096	0.8 GB	0.4 GB	~15 GB	~17 GB ✅
8,192	1.7 GB	0.85 GB	~16 GB	~16 GB ✅
16,384	3.3 GB	1.65 GB	~18 GB	~14 GB ✅
30,000	4.6 GB	2.3 GB	~20 GB	~12 GB ⚠️

All sizes technically fit in 32GB, but the performance implications differ dramatically. At 30K context, each compaction re-evaluates ~15K tokens. At 8K, a compaction only re-evaluates ~4K tokens — **nearly 4× faster recovery**. More importantly, with thinking disabled and `num_predict` capped at 4,096, you're unlikely to trigger compactions at all with an 8K window during normal multi-turn conversations.

For agent and chatbot use cases, **manage conversation length at the application layer** rather than relying on a massive context window. Implement conversation summarization, keep a sliding window of recent messages, and start fresh conversations rather than letting a single thread grow indefinitely. A fast 8K context is far more useful than a sluggish 30K one. Insiderllm

An important note: Ollama auto-detects your M4 32GB as having 24–48 GiB VRAM and may default `num_ctx` to **32,768** GitHub — which is almost certainly what happened here. Always set this explicitly.

Qwen3-14B quantization fits fine — Q4_K_M is the sweet spot

The model itself is not too large. Here's the breakdown for different quantization levels of Qwen3-14B:

Quantization	Model size	Speed (M4 base)	Quality	Verdict
Q4_K_M	**9.3 GB**	**8–12 tok/s**	~95% of FP16	**Best balance — recommended**
Q5_K_M	10.5 GB	7–10 tok/s	~97% of FP16	Marginal quality gain, 15–20% slower
Q8_0	~14.8 GB	5–7 tok/s	~99% of FP16	Near-perfect quality but ~40% slower
FP16	~29.6 GB	N/A	Baseline	Does not fit after OS overhead

Q4_K_M is the clear winner for your setup. It retains ~95% of full-precision quality, runs at the fastest speed tier, and leaves abundant RAM headroom. Medium Localllm The quality difference between Q4_K_M and Q5_K_M is minimal for chat and agent tasks — the perplexity gap is just 0.02 points. Reserve Q8_0 only if you're doing precision-critical work and accept nearly half the speed.

For creative writing, code generation, and general chat, Q4_K_M performs "remarkably well" per multiple benchmark analyses. For complex multi-step reasoning, the difference becomes slightly more noticeable, One Dollar VPS but switching to thinking mode on-demand compensates far more than upgrading quantization.

Smaller models that outperform your current setup

If response speed is your priority for agent/chatbot work, several alternatives deliver faster, high-quality responses on your M4:

Qwen3-30B-A3B (Mixture of Experts) may be the single best model for your hardware. Despite having 30B total parameters, only **3B are active per token**, DataCamp +2 yielding roughly 20–30 tok/s on the M4 base — faster than the dense 14B while delivering higher quality. The Q4 version at ~18–20 GB fits comfortably in 32GB. It supports hybrid think/no-think mode Hugging Face Qwen and strong tool-calling. Ollama This is the "have your cake and eat it too" option.

Qwen3:8B is the speed champion for agent work. Docker's rigorous tool-calling evaluation (3,570 tests across 21 models) Docker ranked it at **F1: 0.919–0.933** — only

3–5% below the 14B variant's 0.971. docker At 25–40 tok/s on M4, it's 2–3× faster than the 14B. Qwen's own benchmarks show Qwen3-8B performs at the level of the previous generation's 14B model (Qwen2.5-14B). github Qwen

Other solid picks include **Llama 3.1:8B** (28–32 tok/s, Like2Byte F1: 0.835 for tool calling, extremely well-tested), **Qwen2.5:7B** (32–35+ tok/s, the fastest quality option), Like2Byte and **Mistral 7B** (30–35 tok/s, solid for simple agent routing). For ultra-fast simple tasks, **Llama 3.2:3B** hits 50–70 tok/s. Medium localllm

Consider a tiered approach: use Qwen3:8B with `/no_think` for routine agent interactions (fast), and switch to Qwen3-30B-A3B or 14B with thinking enabled for complex reasoning tasks.

Apple Silicon GPU acceleration and memory monitoring

Metal GPU acceleration is **built into Ollama on macOS and active by default** — no configuration needed. LocalAimaster Do not run Ollama inside Docker on macOS, as Docker cannot access the Metal GPU and you'll get CPU-only performance (5–6× slower). Viktor Chalyi

To confirm everything is working correctly, run `ollama ps` and verify the PROCESSOR column shows `100% GPU`. If you see a CPU/GPU split like `48%/52% CPU/GPU`, the model is being partially offloaded due to memory pressure Medium — reduce `num_ctx`, close background apps, or use a smaller model. You can also check GPU utilization in real-time with `sudo powermetrics --samplers gpu_power -i 1000`.

Memory pressure is your canary. Open Activity Monitor → Memory tab and watch the pressure graph during inference. Green means healthy. Yellow means macOS is compressing memory pages — still functional but watch closely. Red means active swapping to SSD, which drops throughput Arsturn from 8–12 tok/s to potentially 2–5 tok/s despite the M4's fast NVMe (~7 GB/s, still 17× slower than the 120 GB/s unified memory bandwidth). Keep the "Swap Used" counter near zero during inference.

The rule of thumb for Apple Silicon: keep total model footprint (weights + KV cache) at **≤60–70% of unified memory** for long-context sessions. Insiderllm Like2Byte On 32GB, that's ~19–22 GB — which Qwen3-14B Q4_K_M at 8K context (~16 GB total including OS) satisfies easily, but at 30K context (~20 GB) you're approaching the boundary.

One additional consideration: if maximum speed matters, **MLX-based inference (via LM Studio) runs 20–30% faster than Ollama** on Apple Silicon Insiderllm due to native Metal optimization. Insiderllm The tradeoff is that Ollama has a better API server and model management ecosystem. For agent use cases where you need the Ollama API, stick with Ollama and apply the optimizations above.

Conclusion

Your 10-minute response times are not a hardware limitation — they're a configuration issue with a clear fix. The three changes that will have the most dramatic impact, in order:

1. **Reduce context to 8,192 tokens** (`PARAMETER num_ctx 8192`) — eliminates the compaction death spiral that's the primary cause of your slowdowns
2. **Disable thinking mode** (`think: false` or `/no_think`) — cuts total generated tokens by 2–10× per response, preventing rapid context filling
3. **Enable flash attention and Q8_0 KV cache** — free memory savings that provide headroom and slight speed gains

With these three changes alone, expect response times to drop from ~10 minutes to **5–15 seconds** for typical queries. For even faster responses, consider Qwen3:8B (2–3× faster, only 3–5% quality loss on tool-calling) or the Qwen3-30B-A3B MoE model (faster than 14B dense, higher quality). The M4's 120 GB/s bandwidth sets a hard ceiling of ~12 tok/s for 14B models, but that's entirely adequate for interactive chat Oobabooga when you're not burning time on 26 compaction cycles.