Memory Leak in Long Conversations: Causes and Fixes

February 18, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Usually not a memory leak — it's the KV cache growing by design. Every token in conversation history adds to VRAM usage. A 7B model at 4K context uses ~0.5 GB for the cache; at 32K it uses ~4 GB. Fix: set an explicit context limit (num_ctx in Ollama, --ctx-size in llama.cpp), restart conversations periodically, and use keep_alive to unload idle models. Actual leaks are rare — if VRAM doesn't drop after unloading, restart Ollama or the Docker container.

📚 **More on this topic:** Context Length Explained · VRAM Requirements · Beyond Transformers: 5 Architectures · Planning Tool

You're running a local model. First response is fast. Tenth response is slower. By the twentieth, VRAM is maxed and the model crashes or the system freezes. Something is eating memory with every turn.

Here's what's actually happening — and it's probably not what you think.

It's Usually Not a Leak

The most common cause of climbing VRAM isn't a bug. It's the **KV cache** — and it's working exactly as designed.

Every transformer-based LLM stores attention states for every token it has seen in the current conversation. This is the key-value (KV) cache. It lets the model reference earlier tokens without recomputing attention from scratch. Without it, generating the 100th token would require re-processing all 99 previous tokens.

The KV cache grows linearly with conversation length. More messages = more tokens = more VRAM.

Context Used	KV Cache (7B model)	KV Cache (14B model)	KV Cache (70B model)
1K tokens	~125 MB	~250 MB	~1.2 GB

Context Used	KV Cache (7B model)	KV Cache (14B model)	KV Cache (70B model)
4K tokens	~500 MB	~1 GB	~5 GB
8K tokens	~1 GB	~2 GB	~10 GB
16K tokens	~2 GB	~4 GB	~20 GB
32K tokens	~4 GB	~8 GB	~40 GB

A 14B Q4_K_M model takes ~9 GB for weights. At 4K context, the KV cache adds ~1 GB — total ~10 GB. At 16K context, the cache adds ~4 GB — total ~13 GB. Same model, same quantization, 3 GB difference just from conversation length.

This is why VRAM usage climbs during a conversation even though nothing is "leaking." The model is remembering more, and remembering costs memory.

Actual Memory Leaks (Rare but Real)

Sometimes VRAM genuinely doesn't come back when it should. These are real leaks:

Ollama Not Releasing VRAM

After unloading a model (`ollama stop model` or switching models), VRAM should drop. If it doesn't:

```
# Check what Ollama has loaded
curl http://localhost:11434/api/ps

# Force unload all models
curl http://localhost:11434/api/generate -d '{"model": "modelname", "keep_alive": 0}'

# Nuclear option: restart Ollama
sudo systemctl restart ollama
```

This happens occasionally after loading and unloading many models in sequence. Restarting Ollama clears it every time.

Python Scripts Not Releasing

If you're using `llama-cpp-python`, `transformers`, or similar libraries in a script, failing to delete the model object can hold VRAM:

```
# Bad: model stays in VRAM after you're done
model = Llama(model_path="model.gguf")
# ... use model ...
# script keeps running, VRAM stays allocated

# Better: explicitly free
del model
import gc
gc.collect()

# For CUDA: also clear the cache
import torch
torch.cuda.empty_cache()
```

In Jupyter notebooks, this is especially common — cells hold references to models indefinitely.

Docker Container VRAM

Docker containers with GPU access can hold VRAM after the process inside exits, especially if the container is still running. The fix:

```
# Stop the container
docker stop container_name

# If VRAM still held:
docker rm container_name

# Verify with nvidia-smi
nvidia-smi
```

If VRAM remains allocated after removing the container, restart the Docker daemon: `sudo systemctl restart docker`.

Diagnosing the Pattern

Watch VRAM over time to determine if you have normal KV cache growth or an actual leak.

Normal behavior: VRAM increases during a conversation, then drops to baseline when the model unloads or the conversation resets. Each new conversation starts at the same VRAM level.

Actual leak: VRAM increases and never returns to baseline, even after unloading models. Each new conversation starts higher than the last.

Monitoring Tools

```
# Watch VRAM every second
nvidia-smi -l 1

# Better: visual GPU monitor (install: sudo apt install nvtop)
nvtop

# RAM monitoring
htop

# Ollama-specific: see loaded models and VRAM usage
curl -s http://localhost:11434/api/ps | python3 -m json.tool
```

The `nvidia-smi` output shows `Memory-Usage` per process. Track the Ollama or llama.cpp process specifically — other GPU processes (desktop compositor, video player) also use VRAM.

Fixes

Set an Explicit Context Limit

Don't let context grow unbounded. Cap it to what your VRAM can handle.

```
# Ollama: set in Modelfile
PARAMETER num_ctx 8192

# Ollama: set at runtime
```

```
curl http://localhost:11434/api/generate -d '{
  "model": "qwen3:8b",
  "prompt": "Hello",
  "options": { "num_ctx": 8192 }
}'

# llama.cpp server
llama-server -m model.gguf --ctx-size 8192
```

When conversation exceeds the limit, oldest messages get truncated. The model forgets earlier context, but it doesn't crash. This is the tradeoff: bounded memory vs complete history.

Unload Idle Models

Ollama keeps models loaded in VRAM by default (5 minutes idle timeout). For tight VRAM budgets, reduce this:

```
# Unload after 60 seconds of inactivity
curl http://localhost:11434/api/generate -d '{
  "model": "qwen3:8b",
  "keep_alive": "1m"
}'

# Unload immediately after response
curl http://localhost:11434/api/generate -d '{
  "model": "qwen3:8b",
  "keep_alive": 0
}'
```

Or set the default globally: `OLLAMA_KEEP_ALIVE=1m` as an environment variable.

Periodic Conversation Reset

For long-running apps (chatbots, agents, automation), implement a sliding window:

```
MAX_TURNS = 20

def trim_conversation(messages):
    system = [m for m in messages if m["role"] == "system"]
    history = [m for m in messages if m["role"] != "system"]
```

```
    # Keep system prompt + last N turns
    return system + history[-(MAX_TURNS * 2):]
```

This keeps recent context and the system prompt while dropping older messages. The KV cache stays bounded because the model only processes the trimmed history.

For more sophisticated approaches, summarize dropped messages and prepend the summary. This preserves key information without the memory cost.

Flash Attention

If your engine supports it, flash attention reduces KV cache memory by 2-4x:

```
# llama.cpp
llama-server -m model.gguf --ctx-size 16384 --flash-attn

# Ollama: enabled by default on supported hardware
```

This doesn't change behavior — just makes the same context window cost less VRAM.

The Transformer Tax

The KV cache problem is inherent to transformer architecture. Attention is O(n) in memory for the cache (and O(n²) in compute for full attention). Every model based on the transformer — Llama, Qwen, Mistral, Gemma, Phi — pays this tax.

Alternative architectures avoid it entirely:

- **RWKV** — constant memory regardless of sequence length. No KV cache. Runs 8B models on 4GB VRAM at any context length.
- **Mamba / Mamba 2** — linear-time state space model. Memory doesn't grow with conversation length.
- **Jamba** — hybrid transformer+Mamba. Uses KV cache for some layers but much less than pure transformer.

These aren't replacements for transformers yet — quality and ecosystem lag behind. But if your primary constraint is running long conversations on limited memory, they're worth investigating.

Bottom Line

If VRAM climbs during a conversation, it's almost certainly the KV cache growing — not a leak. Set an explicit context limit, implement conversation trimming for long-running apps, and unload idle models.

True memory leaks (VRAM not returning after model unload) are fixed by restarting Ollama, clearing Python references, or restarting Docker. They're rare and always fixable with a process restart.

The fundamental tradeoff: longer conversations cost more memory. You can make the model forget older context (truncation), make it remember more efficiently (flash attention), or switch to an architecture that doesn't have this constraint at all (RWKV, Mamba). Pick the tradeoff that fits your use case and VRAM budget.