

Tune R9700s for jobs, not tokens

Nick didn't write this. It's an AI compilation of our tuning run: a day of measurements on one of our quad-R9700 servers in October 2026, with every arm, result and trace kept in a ledger. Where it says "I" or "we", that's Nick's setup, told from that ledger and his notes.

Use this guide at your own risk. It records what we did with our own hardware and software in October 2026. It isn't professional advice, and it comes with no warranty of any kind. Changing a serving stack can crash it, slow it down or give you wrong answers, and patched software behaves differently from what its authors ship and support. Test every change on your own work before you trust it, and keep a way back to what you had. Software, images and models change, so check them yourself. You are responsible for your own setup; Nick Reese and his companies accept no liability for any damage or loss from using this guide. No manufacturer or project sponsored, reviewed or endorsed it, and product names belong to their owners.

By Nick Reese. These machines keep USAArms.com up to date.

The tuning guide for the video. One server with four 32 GB Radeon AI PRO R9700 cards, one model the whole time (Qwen3.8-27B in FP8), and a hundred real jobs from USA Arms. Stock vLLM got 475 right answers an hour. What we run now gets 4,241, almost nine times more, and more of them are right. This is every step of that climb, with the exact flags, why each one helped, and how to test it on your own work. It also lists what we tried that didn't help, so you can skip it.

The ladder at a glance

Every number is right answers an hour on the same hundred jobs, with the whole server working: two engines of two cards each, sixteen jobs in flight between them.

Step	What changes	Right answers an hour	Right of 100
0. Stock vLLM 0.30	Starts on a 7.x kernel with one variable	475	76
1. Grammar patch	Tool-call strings can hold a newline or a quote	1,385	100
2. vLLM-Radiance 1.0.387	A community fork built for these cards	2,584	98
3. Eight-token draft	Speculative decoding, judged on our jobs	4,241	98
4. FP8 cache, kept	Twice the context of a 16-bit cache	4,241	98
5. One hardware queue	No more half speed after long documents	4,677	98

Step 5 was measured on its own run, and two runs of the same setup differ by about 5%, so read 4,677 as "no cost", not as a gain. On long documents the climb was bigger: stock vLLM did 62 of our 27,000 to 54,000-token jobs an hour, and what we run does 502 to 517.

Two ways to use this guide

- **Hand it to your AI.** Download the text version (Markdown), give it to your coding agent along with your own cards, model and jobs, and have it walk your engine up the same ladder one step at a time, measuring each step on your work before keeping it.
- **Read it.** Each step says what to change, why, how to check it, and what it did for us.

Measure the right thing first

The whole guide rests on this. Tokens a second for one user is the number most benchmarks print, and it pointed us the wrong way more than once: one engine across all four cards was 31% faster for one user and did a third less real work.

1. **Use your own jobs.** We froze a hundred real jobs from USA Arms: data extraction with tool calls, about 5,000 tokens of prompt each, every one with a known right answer. A second suite has 40 long jobs at 27,000 to 54,000 tokens.
2. **Check every answer.** Our harness, the software around the model, hands each job to the engine and checks the answer. Only right answers count, so every number here is right answers an hour.
3. **Run at your real load.** We keep sixteen jobs in flight across the two engines, as production does. One-at-a-time speed and many-at-once throughput rank setups differently.
4. **Judge a change on its second run.** The first run after you change any vLLM setting is about 40% slower, because vLLM compiles kernels for the new configuration on first use and caches them (2,752 an hour with an empty cache against 4,241 and 4,468 with it filled). Two filled-cache runs of the same setup differ by about 5%, so a smaller gap between two setups is noise.
5. **Restart before a speed test.** Until step 5 below, a batch of long prompts could leave an engine at half speed, so speed measured right after long jobs isn't comparable.

For speed alone we used `vllm bench serve` from a separate stock vLLM container: MT-Bench with one, four and eight streams, and four 32,000-token prompts at once.

Our setup

- **Server:** AMD EPYC 7452 (one NUMA node), four ASRock Radeon AI PRO R9700 Creator 32 GB cards, each on PCIe Gen4 x16 to the CPU, power capped at 210 W a card. Clocks, power profiles and firmware were never tuning levers in this run.
- **Software:** Ubuntu with a 7.x kernel, Docker, and the vLLM images named below.
- **Model:** Qwen/Qwen3.8-27B-FP8, revision `017b9c7af6b5689d5dd426a76e0bc077eb5ca20a`. It is a hybrid model, which is why `--mamba-cache-mode=align` appears below.
- **Layout:** two engines of two cards each (tensor parallel 2), one on cards 0 and 1 and one on cards 2 and 3. Which two cards you pair didn't matter: one stream ran 51 to 53 tokens a second on all four pairs we tried, including two cards on the same CPU root complex.

Step 0: Start stock vLLM on two cards

Change: run `vllm/vllm-openai-rocm:v0.30.0` (we pinned `sha256:2e7da1ad1c66836802072588adea75f9f4991da5f9545b4318e91d422c22ce6a`) with `HSA_ENABLE_IPC_MODE_LEGACY=0`.

Why: on a 7.x kernel, stock vLLM won't start two cards per engine. RCCL dies at `p2p.cc:256` with `hipIpcGetMemHandle invalid argument`. `HSA_ENABLE_IPC_MODE_LEGACY=0` fixes it; `NCCL_PROTO=Simple` alone does not (we set both).

Check: both engines reach ready and answer a request.

What it did for us: 475 right answers an hour, 76 of 100 right. Its log shows why it's slow on these cards: no tuned FP8 block-quantized kernel configuration for the R9700, no ROCm custom paged attention (it falls back to Triton), and no speculative decoding.

Step 1: Let tool-call strings hold a newline or a quote

Skip this step only if you never use tool calls or JSON-schema output.

Why: vLLM's grammar engine, xgrammar, compiles a JSON-schema string that has `minLength` or `maxLength` into the pattern `[^"\\r\n]{min,max}`, with no escape alternative. Inside such a string the model can't write a newline, a quote or a backslash, so a quote of text that contains one ends early or fails. On our jobs that cost 24 answers in 100 and two thirds of the throughput, because the failing jobs also ran long. We found it in xgrammar 0.2.3 and it was still in 0.2.8 and upstream at the time of writing; vLLM-Radiance 1.0.387 ships the same file.

Check: send a tool schema with a length-bounded string and ask for a value with a line break or a " in it. If the string stops at the first line break, you have the bug.

Change: patch vLLM's `vllm/v1/structured_output/backend_xgrammar.py` so it drops string length bounds from the schema before xgrammar compiles it. The bounds stay in the request and in whatever validates the answer afterwards; only the decode-time grammar stops enforcing them. The core of our patch:

```
def _escape_safe_schema(value):
    """Drop JSON-schema string length bounds before xgrammar compiles them."""
    if isinstance(value, list):
        for item in value:
            _escape_safe_schema(item)
    elif isinstance(value, dict):
        for key in ("minLength", "maxLength"):
            bound = value.get(key)
            if isinstance(bound, int) and not isinstance(bound, bool):
                del value[key]
        for key, item in value.items():
            if key in ("properties", "patternProperties", "$defs", "definitions"):
                if isinstance(item, dict): # name -> schema maps; names are not keywords
                    for schema in item.values():
                        _escape_safe_schema(schema)
            elif key not in ("const", "enum", "default", "examples"): # literal JSON values
                _escape_safe_schema(item)
    return value

def _escape_safe_spec(grammar_spec: str) -> str:
    return json.dumps(_escape_safe_schema(json.loads(grammar_spec)))
```

Apply `_escape_safe_spec` to every JSON-schema grammar spec in that file before it is compiled, including the schemas inside structural tags, and log a line at import so you can see the patch is active. Copy the original file out of the exact image you run, patch that copy, and mount it read-only over the original. A new image needs its own copy of the file, patched again.

What it did for us: 1,385 right answers an hour, 100 of 100 right. The median job went from 63.3 to 27.7 seconds.

Step 2: Switch to vLLM-Radiance

Change: run vLLM-Radiance 1.0.387, a community fork of vLLM built for these cards (we pinned `magiccodingman/vllm-radiance@sha256:c0e8f375de5a461e6aff0761593c85d57f32165b616f073fc3b9971eb702545f`), with the grammar patch from step 1 mounted over its copy of the file. Its settings are environment variables; ours are in Every flag we run.

Why: it pairs vLLM 0.30 with hand-written RDNA4 kernels for these cards (libr4d), including its own attention and all-reduce, which stock vLLM is missing on the R9700.

What it did for us: 2,584 right answers an hour without speculative decoding, 98 of 100 right.

Step 3: Speculative decoding with an eight-token draft

Change: `RADIANCE_SPECULATIVE_CONFIG={"method":"mtp","num_speculative_tokens":8}`. The model's own multi-token-prediction head drafts eight tokens a step and the model checks them in one pass.

Why: on our jobs the draft is almost always right. It gets 95 to 96% of drafted tokens accepted on our jobs, about 8.4 tokens a step, and 83% on live production traffic (7.6 a step). On MT-Bench chat it manages only about 2.8 tokens a step, so a chat benchmark would have talked us out of it. Judge speculative decoding on your own jobs.

Check: the engine's `/metrics` counters `vllm:spec_decode_num_accepted_tokens_total`, `vllm:spec_decode_num_drafts_total` and `vllm:spec_decode_num_draft_tokens_total` before and after a run of your jobs give tokens a step and the share accepted.

What it did for us: 4,241 right answers an hour, 98 of 100 right. This is what we run in production. One stream writes 52.7 tokens a second on a fresh two-card engine.

A draft of four tokens and a draft of twelve both did less work on our traffic: 548 and 385 conversation turns an hour against 608 to 638 with eight.

Step 4: Keep the FP8 KV cache

Change: `--kv-cache-dtype=fp8` (none, if you already have it).

Why: on vLLM-Radiance it matched a 16-bit cache on accuracy (98 against 99 of 100, and 39 of 40 long jobs each, failing the same one), holds twice the tokens (583,000 against 291,000 per two-card engine) and is slightly faster (52.7 against 51.2 tokens a second for one stream). The one extra right answer cost half the headroom, so we kept FP8.

Step 5: One hardware queue, or long documents leave you at half speed

Symptom: after a batch of long prompts, an engine decodes at about half speed and stays there until it restarts. One stream fell from 52.7 to 29.0 tokens a second, and eight streams from 285 to 198. It happened in 7 of 8 runs. The cards weren't hot, out of memory or throttled; they ran higher clocks at the same power, because they were waiting.

Why: each decode step, vLLM copies the sampled tokens to the host on a side stream that waits for the main one. When ROCm puts that waiting copy on hardware queue 3 while the model runs on queue 1, every kernel on that GPU starts about 17 microseconds later (the median gap between kernels went from 4.2 to 21.7 microseconds), and the other GPU of the pair waits for it in the all-reduce. Which queue the copy lands on drifts under load, so after a long batch it was a coin flip weighted about seven to one towards slow.

Change: `GPU_MAX_HW_QUEUES=1` in the engine's environment. Every HIP stream then shares one hardware queue.

Check: run your long jobs, then a one-stream speed test (`vllm bench serve` at one stream), and compare it with the same test on a freshly started engine. Do it more than once: the slowdown is a chance event, so one clean run proves nothing.

What it did for us: no slowdown in 10 engine runs against 7 of 8 without it, the same speed fresh, and no cost on short jobs (4,677 right answers an hour, 98 of 100 right).

What didn't work for us

Each of these was one change from what we run, measured the same way.

- **One engine across all four cards (tensor parallel 4).** One stream was 31% faster (69.0 against 52.7 tokens a second) and long prompts read 18% faster, but it did a third less real work: 2,826 right answers an hour against 4,241, and 316 long jobs an hour against 502 to 531. Split four cards into two engines of two unless you serve one user at a time.
- **Four single-card engines.** On 4-bit weights each card holds only 40,000 tokens of context next to the model, which can't take our long jobs. FP8 weights leave no room for context on one 32 GB card at all.
- **Fast draft (RADIANCE_FAST_DRAFT=1).** It writes each token faster (73.1 against 89 milliseconds), but on our traffic it got 368 turns an hour against 608 to 638, because conversations waited longer and lost their cached prefixes (52% cached against 71%).
- **A shorter or longer draft.** Four tokens: 548 turns an hour. Twelve: 385.
- **Bigger prefill batches (--max-num-batched-tokens=4096).** 409 turns an hour; every conversation writing at the time slowed by about 60% a token.
- **More GPU memory (--gpu-memory-utilization=0.91).** Inside the run-to-run noise.
- **A newer vLLM-Radiance (1.0.394).** No faster on our traffic, though it was the only setup to get 100 of 100 on the hundred jobs.

The turns-an-hour figures come from a replay of production's traffic shape: eight growing conversations built from our long jobs, from about 20,000 to 154,000 tokens a turn, split over both engines and sent one turn at a time.

The next rung: 4-bit weights

AMD's 4-bit build, amd/Qwen3.8-27B-Quark-AWQ-MXFP4, with one hardware queue, did 5,278 right answers an hour on our hundred jobs at 98 of 100, and 836 turns an hour on our traffic against production's 608 to 638. On a hundred jobs no setup had been tuned on, though, it got 91 right against production's 95. For jobs that must be right, we stay on FP8 for now. Our next video, "Right, or close enough", is about that trade-off. The 4-bit run used the same settings as production plus:

```
model: amd/Qwen3.8-27B-Quark-AWQ-MXFP4 (revision 5233554c5fa56afda40150556b95573c2d7d29c0)
RADIANCE_MXFP4=1
RADIANCE_MXFP4_W4A8=1
RADIANCE_MXFP4_WPERM=1
RADIANCE_MXFP4_DECODE_NT=1
RADIANCE_QUARK_BF16_MTP=1
RADIANCE_FAST_DRAFT=1
GPU_MAX_HW_QUEUES=1
```

Every flag we run

One engine of what we run, on cards 0 and 1, with step 5's variable added. The second engine is the same with HIP_VISIBLE_DEVICES=2,3 and its own port. Replace the three paths with your own model folder, cache folder and patched file.

```

docker run -d --name engine-a --restart=no \
  --device=/dev/kfd --device=/dev/dri --ipc=host \
  --security-opt=seccomp=unconfined \
  --volume /path/to/Qwen3.8-27B-FP8:/models/model:ro \
  --volume /path/to/cache/engine-a:/cache \
  --volume /path/to/backend_xgrammar.py:/opt/vllm/lib/python3.12/site-
packages/vllm/v1/structured_output/backend_xgrammar.py:ro \
  --publish 18000:8000 \
  --env HIP_VISIBLE_DEVICES=0,1 \
  --env GPU_MAX_HW_QUEUES=1 \
  --env 'RADIANCE_SPECULATIVE_CONFIG={"method":"mtp","num_speculative_tokens":8}' \
  --env 'RADIANCE_COMPILATION_CONFIG={"max_cudagraph_capture_size":64}' \
  --env RADIANCE_WEIGHT_QUANTIZATION=auto \
  --env RADIANCE_USE_R4D=1 --env RADIANCE_USE_R4D_GDN=1 --env RADIANCE_USE_R4D_AR=1 \
  --env RADIANCE_MXFP4=0 --env RADIANCE_MXFP4_W4A8=0 --env RADIANCE_MXFP4_W4A8_MIN_M=0 \
  --env RADIANCE_MXFP4_DECODE_MAX_M=64 --env RADIANCE_MXFP4_TN4_MIN_M=2048 \
  --env RADIANCE_MXFP4_WPERM=0 --env RADIANCE_MXFP4_DECODE_NT=0 \
  --env RADIANCE_FAST_DRAFT=0 --env RADIANCE_NORMQUANT_FUSION=0 --env RADIANCE_FP8_STREAM=0 \
  --env RADIANCE_QUARK_BF16_MTP=0 --env RADIANCE_GDN_MERGE_INPROJ=0 --env RADIANCE_RUN_BWTEST=0 \
  --env VLLM_ROCM_USE_AITER=1 --env VLLM_ROCM_USE_AITER_UNIFIED_ATTENTION=1 \
  --env VLLM_ROCM_USE_AITER_LINEAR=0 --env VLLM_ROCM_USE_AITER_MOE=0 --env VLLM_ROCM_USE_AITER_MHA=0 \
  --env VLLM_ROCM_USE_AITER_MLA=0 --env VLLM_ROCM_USE_AITER_FP8BMM=0 --env VLLM_ROCM_USE_AITER_FP4BMM=0 \
  --env VLLM_ROCM_USE_AITER_RMSNORM=0 \
  --env NCCL_PROTO=Simple \
  --env TORCHINDUCTOR_COMPILE_THREADS=4 \
  --env VLLM_CACHE_ROOT=/cache/vllm --env TORCHINDUCTOR_CACHE_DIR=/cache/inductor \
  --env TRITON_CACHE_DIR=/cache/triton --env AITER_JIT_DIR=/cache/aiter \
  magiccodingman/vllm-radiance@sha256:c0e8f375de5a461e6aff0761593c85d57f32165b616f073fc3b9971eb702545f \
  /models/model \
  --tensor-parallel-size=2 --port=8000 \
  --served-model-name=qwen/qwen3.8-27b \
  --max-model-len=262144 \
  --max-num-seqs=8 \
  --max-num-batched-tokens=2048 \
  --gpu-memory-utilization=0.85 \
  --kv-cache-dtype=fp8 \
  --attention-backend=R4D \
  --enable-prefix-caching \
  --mamba-cache-mode=align \
  --no-async-scheduling \
  --enable-auto-tool-choice \
  --tool-call-parser=qwen3_coder \
  --reasoning-parser=qwen3 \
  --generation-config=auto \
  --limit-mm-per-prompt='{"image":0,"video":0}'

```

Our own engines also pass `--host` so the server listens on every address inside the container; check what your image's endpoint needs. Keep the cache folder between restarts: it holds the compiled kernels from step 4 of "Measure the right thing first", and an empty one makes the first run slow again.

A note for your AI: `--max-num-seqs=8` per engine is what gives sixteen jobs in flight across two engines. `--max-model-len=262144` is the model's full context; lower it if your jobs are shorter and you want more of them in the cache at once.