A full cache throws 47 streams out, and all 400 still finish

400 requests arrive at once, each with a 4,000-token prompt and a 2,000-token answer. The server lets in 261, which fill its cache. As their answers grow it throws 47 of them out, in two waves, and puts them back in the line. Nothing is refused and nothing fails: the price is waiting.

Qwen3.8-27B · one AMD Instinct MI300X · vLLM 0.27.1 **Run** 20261125T201346Z-series **Changed** nothing: baseline settings, 1,024 stream slots
Load 400 requests at once: 4,000 tokens in, 2,000 out

I sent one server 400 requests at once, each with a prompt of 4,000 tokens and an answer of 2,000. It let in 261, which filled its cache, and made the other 139 wait. Then the answers grew and no longer fitted: the server threw 47 running streams out of the cache, in two waves, and put them back in the line. In this plan nothing was refused and nothing failed. All 400 answers arrived, the last after 10 min 30 s.

400
of 400 requests ended well: none refused, none failed, none cut

47
streams thrown out of the full cache and put back in the line

171 s
longest silence inside one answer: a stream that was thrown out

119 s
median wait for a first token; 477 s at the longest

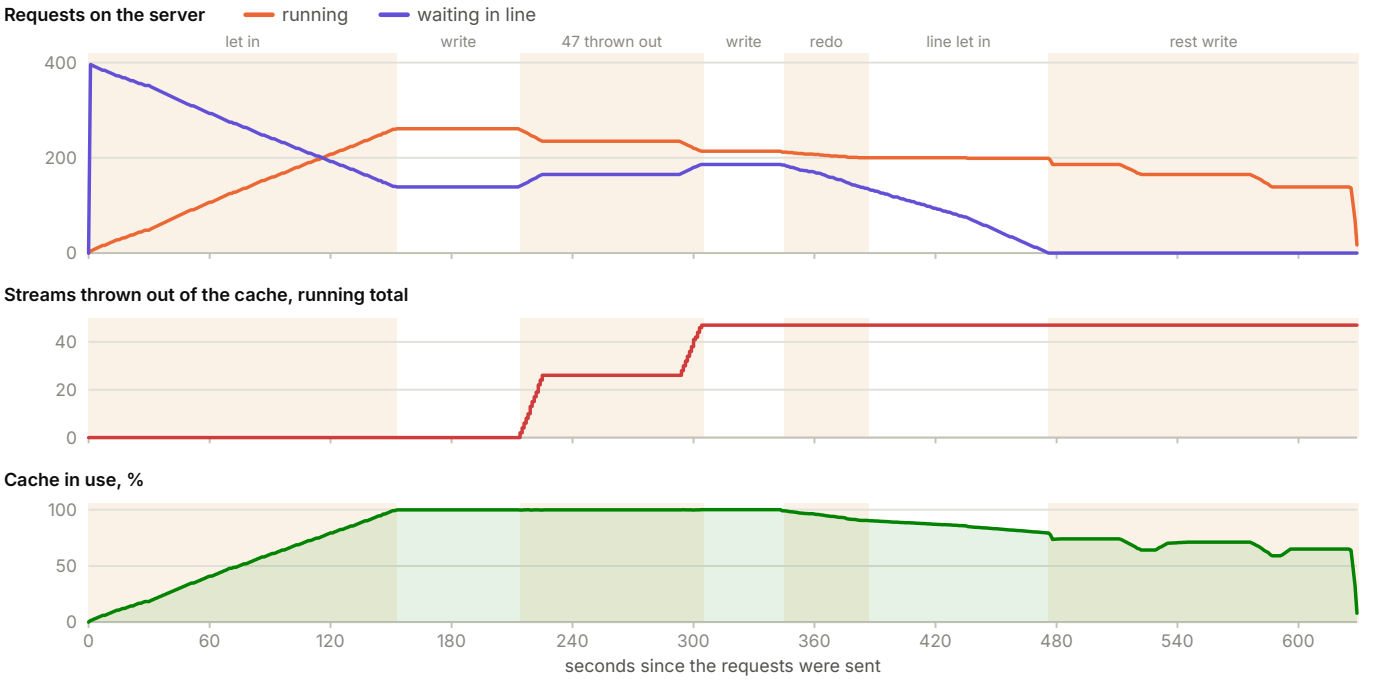


Figure 1 The whole plan, second by second. Top: requests running on the server and waiting in its line. Middle: streams thrown out of the cache, counted up. Bottom: the cache. Shaded bands are the phases of the plan, found in the record: prompts let in, answers written, streams thrown out, the redo, the line let in, the rest written.
Source: plan-kv-growth/timeline.json (vLLM's gauges and its preemption counter, sampled every second)

MEASURED
The cache gauge read 99.79% as soon as 261 streams were in, and 100% once 214 were left. No request had ended by then: the 47 are streams, not repeats. The server's reason for every waiting request was "capacity".

MEASURED
The request log shows the same 47: that many answers have a silence of 42 to 171 s inside them. No other answer has one longer than 12.9 s.

The machine, and the part that overflows

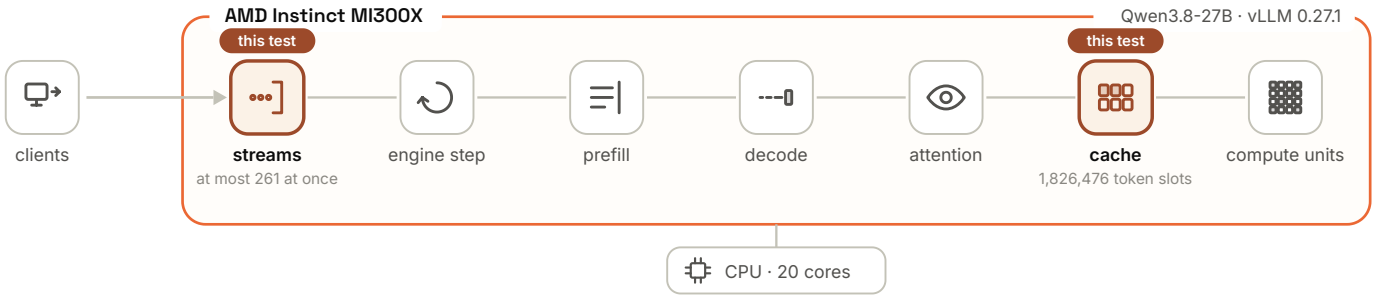


Figure 2 The card as every document in this series draws it: seven parts wired left to right, the clients outside, the host CPU below. This test is about the cache: what the server does when the streams it has let in outgrow it.
Source: FACTS.json; summary.json server.kv_cache_tokens, plans.plan-kv-growth.stages[stage=grow-400].running_max

One burst, and answers that grow

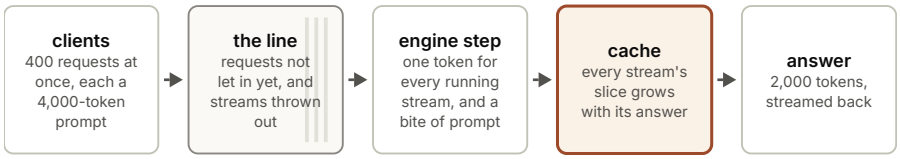


Figure 3 One burst: every request is sent in the same instant, and nothing arrives after it. A stream's slice of the cache grows while its answer is written; a stream thrown out of the cache goes back into the line.
Source: loadgen.py (a "once" stage); plans/kv-growth.json

1. The plan sends 400 requests in one instant and lasts until all have ended. Nothing else arrives.
2. Every prompt is 4,000 random token ids, so no two share a prefix; every answer is forced to 2,000 tokens. A stream grows from 4,000 to 6,000 tokens while it is answered.
3. The plan counted on all 400 fitting at first: by tokens alone they are 87.6% of the cache's 1,826,476 token slots. Only 261 fitted, because a running stream takes more than its tokens. The cache overfilled all the same, which is what the plan was for.
4. Beside it a sampler reads the server's counters, the card and the CPU once a second.

Settings

Model	Qwen3.8-27B: 27.8 billion parameters, BF16 weights, 262,144 tokens of context
Card	one AMD Instinct MI300X: 304 compute units, 191.7 GiB of memory
Server	vLLM 0.27.1+rocm723, ROCm 7.2.3, PyTorch 2.11.0+gitd0c8b1f
Machine	a cloud GPU virtual machine: 20 virtual cores, Intel Xeon Platinum 8568Y+
This run	20261125T201346Z-series, started 2026-11-25T20:13:59Z, server ready after 94 s

KEEP IN MIND

The load is made on the same machine as the server and shares its CPU cores. The generator runs inside the server's container and talks to it on the loopback address: no request leaves the machine.

Words used

Preempted
a running stream thrown out of a full cache and put back in line; its work is redone later.

Cache
(KV cache) the GPU memory where the server keeps what the model computed for every token of every open stream, so it need not be recomputed.

Capacity
(waiting reason) the server's label for a request it cannot schedule now, because the cache or the step's token budget is used up. Reported in `vllm:num_requests_waiting_by_reason`.

Prefill
the server reading the prompt. Every prompt token passes through the model once before the first answer token.

Decode
the server writing the answer, one token per stream per engine step.

Engine step
one pass of the model over a batch: one new token for every decoding stream, plus a bite of prefill.

Watermark
a share of the cache kept free when admitting requests (`--watermark`), to avoid preemption.

TFLOPS
trillions of arithmetic operations per second. Here always vLLM's estimate from the model's shape (`--enable-mfu-metrics`), not a hardware counter. It counts attention in all 64 layers where this model has it in 16: 1.7% high for short prompts, over half too high from 128,000 tokens of context.

Letting requests in and writing answers do not go together

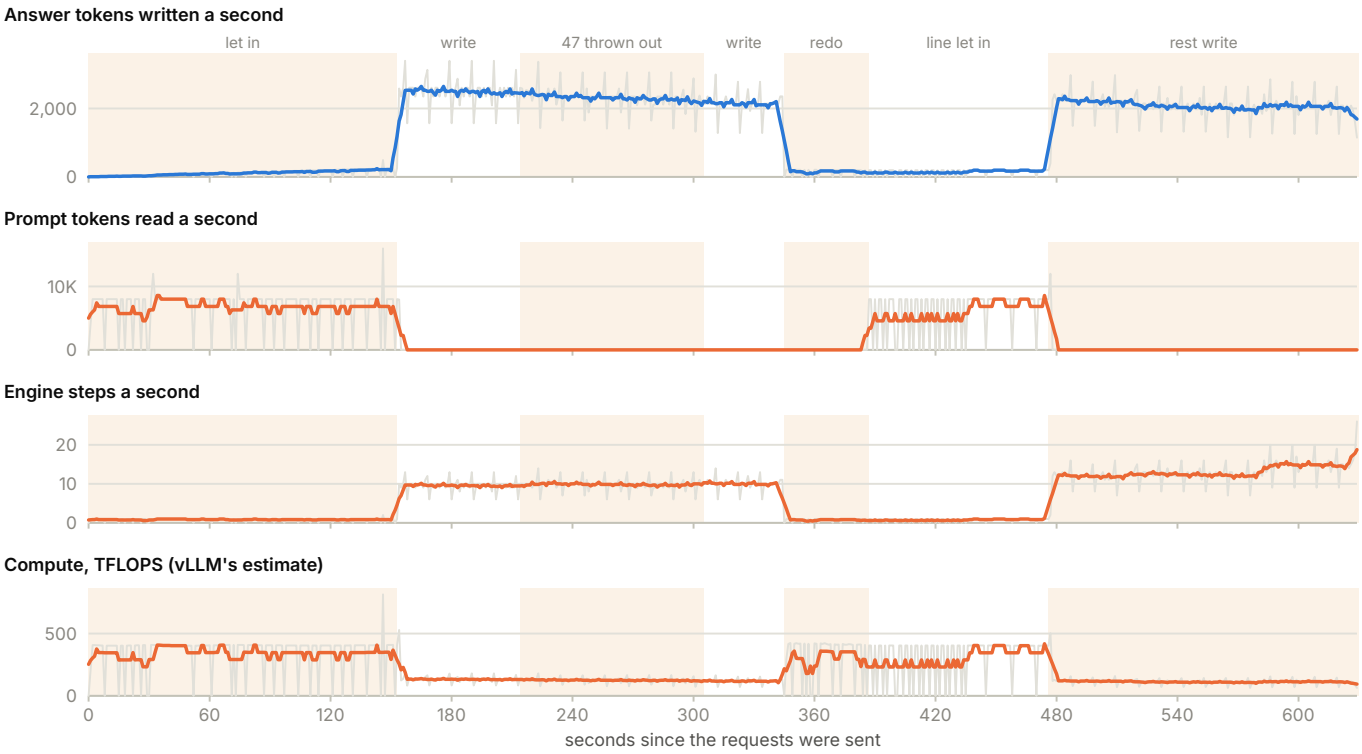


Figure 4 What the engine did, second by second: each second thin and grey, a 7-second centred mean in colour. While prompts are read it makes about one step a second and writes almost nothing; between those phases it makes about ten steps a second.
Source: plan-kv-growth/timeline.json (vLLM's token, step and estimate counters)

The exact record, phase by phase

WHAT HAPPENS	SECONDS	STREAMS RUNNING	THROWN OUT SO FAR	PROMPT TOKENS/S	ANSWER TOKENS/S	ENGINE STEPS/S	TFLOPS, EST.	WATTS
Prompts are read; all that fit are let in	0 to 153	0 to 260	0	6,718	108	0.8	338	661
261 streams write	153 to 214	261	0	261	2,455	9.4	143	750
Answers outgrow the cache: 47 streams thrown out, in 2 waves	214 to 305	259 to 214	47	0	2,318	9.8	126	750
The streams that are left write; the cache is full	305 to 345	214	47	0	2,156	10.1	119	750
First answers end; no new prompt is read yet	345 to 387	212 to 200	47	0	146	0.7	294	633
The requests that never got in are let in	387 to 476	200	47	6,022	149	0.8	304	674
The rest write	476 to 630	199 to 17	47	130	2,058	12.9	117	749

Phases are found in the per-second record: the first second with the most streams running, the first and the last preemption, the first answer ended, the first prompt tokens read after that, and the line empty. Rates are means of the per-second values of the server's counters; TFLOPS is vLLM's estimate (about 6% too high for prompts and answers of this length), watts are AMD's exporter (a new reading every 16 s).

MEASURED

While it lets requests in, the engine makes about one step a second: two 4,000-token prompts fill a step's budget of 8,192 tokens, and each stream already running gets one token in it. With 261 streams in and no prompt left to read it makes about ten, and writes about 2,495 answer tokens a second.

MEASURED

The same happens again when the first answers end, after 345 s. The room that comes free lets the line in, steps fall back to one a second, and the streams that were never thrown out need until 478 s for their last tokens.

A growing answer needs another block of cache, and none is free

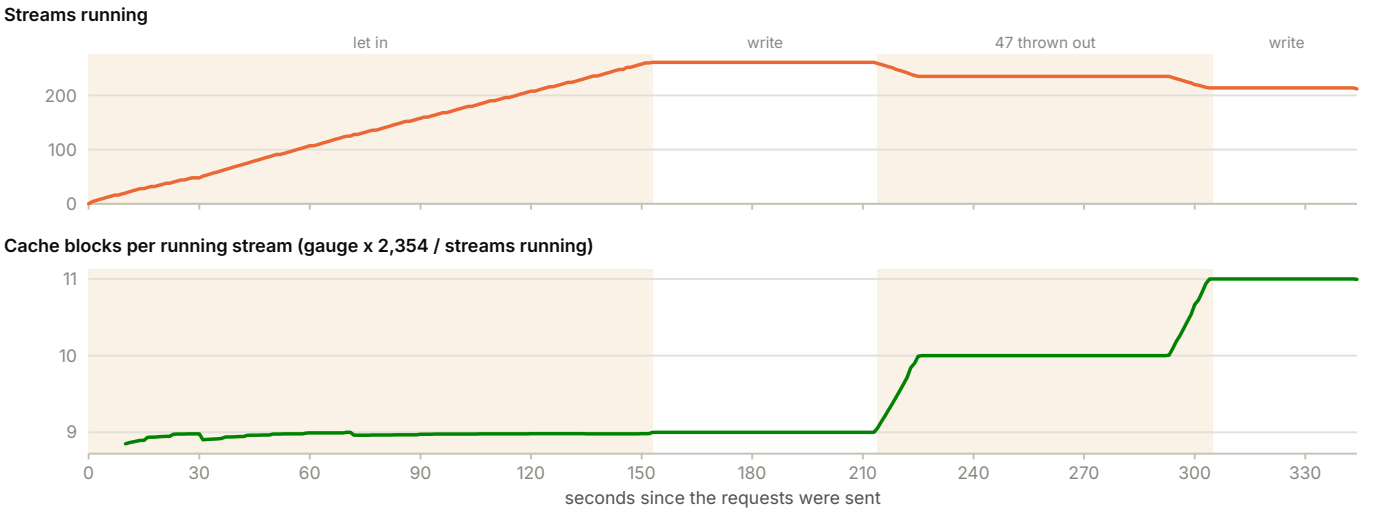


Figure 5 Until the first answer ends. Top: streams running. Bottom: the cache in use divided by the streams running, in blocks. It stands at 9, then 10, then 11: every running stream takes one more block at the same time, and to make room the server throws streams out.
Source: plan-kv-growth/samples.jsonl (vllm:kv_cache_usage_perc, vllm:num_requests_running); blocks computed, see the text



Figure 6 The three standstills, in blocks of cache: streams running times blocks each, against all 2,354.
Source: plan-kv-growth/samples.jsonl; computed

SECONDS	STREAMS RUNNING	CACHE GAUGE, %	BLOCKS IN USE (GAUGE X 2,354)	BLOCKS PER STREAM	BLOCKS FREE
153 to 213	261	99.788	2,349	9.000	5
226 to 292	235	99.830	2,350	10.000	4
304 to 343	214	100.000	2,354	11.000	0

COMPUTED

Cut the cache into **2,354 equal blocks**, and every value its gauge took in this plan is a whole number of them. Counted so, 261 streams hold 9 blocks each, 235 hold 10 and 214 hold 11: the gauge's 99.79%, 99.83% and 100%. A block is 776 of the cache's 1,826,476 token slots.

READING, NOT MEASUREMENT

My reading: the cache is handed out in equal blocks. A running stream holds 3 whatever its length, and one more for every 776 tokens, begun or full: 9 for a 4,000-token prompt, 10 from 4,655 tokens, 11 from 5,431. vLLM's help text speaks of "KV cache blocks"; their number and what a stream takes are my arithmetic on the gauge, not read in its source.

What the block count asks for, and what the server did

SECONDS	STREAMS BEFORE	BLOCKS EACH: FROM, TO	BLOCKS FREE	FREED BY ONE STREAM THROWN OUT	MUST GO, COMPUTED	THROWN OUT, MEASURED	STREAMS AFTER
214 to 225	261	9 to 10	5	9	26	26	235
294 to 304	235	10 to 11	4	10	21	21	214

COMPUTED

When every stream needs one more block and too few are free, a stream thrown out frees its own blocks and needs none. The fewest that must go to serve the rest come to 26 and 21: what the server's counter shows, wave by wave.

MEASURED

The streams thrown out were the last ones let in: their first tokens came 127 to 154 s after the burst, those of the streams that stayed 1 to 127 s.

The same count, in token slots: 3 blocks are 2,328 slots, and a stream's last block is partly empty. Derived from the cache size alone, at four moments of this run, a stream's footprint beside its tokens came out at roughly 2,500 to 2,900 slots; at the last standstill here it is 2,535.

Three kinds of request, three kinds of wait

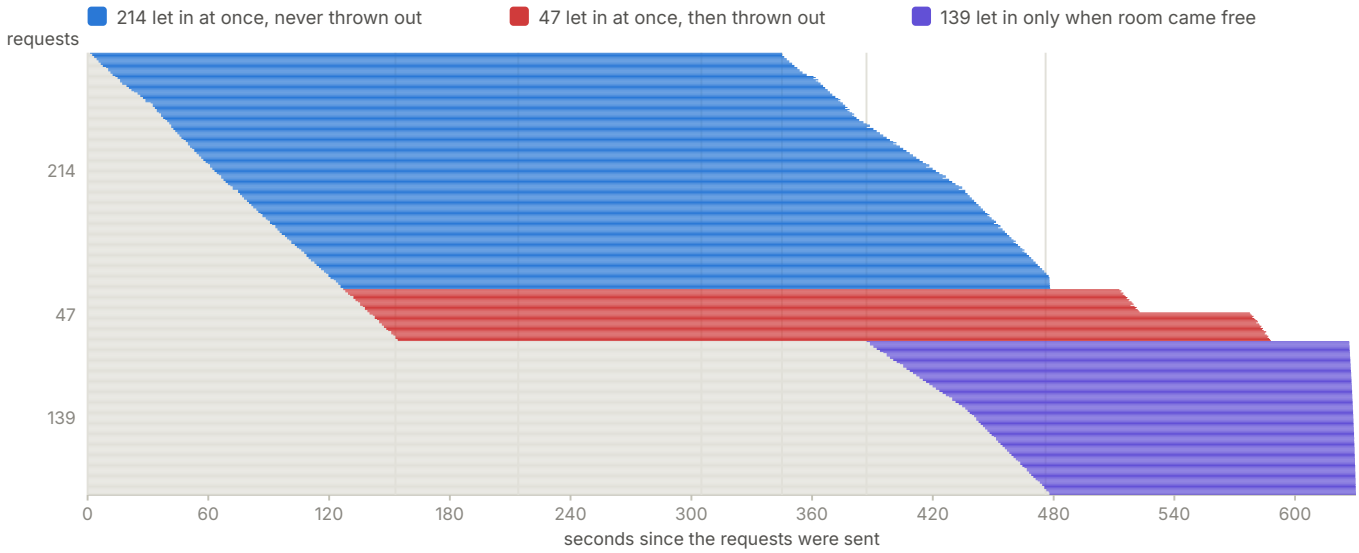


Figure 7 Every request as one thin line, from its first token to the end of its answer (grey before it: waiting for the first token). Sorted by what happened to it, then by first token. The streams thrown out are the last ones that were let in; they ended a minute or two after the others of their burst.
Source: plan-kv-growth/requests.jsonl (t_sent, t_first, t_end, gap_max); phase lines from timeline.json

WHAT HAPPENED TO THE REQUEST	REQUESTS	FIRST TOKEN AFTER, S	LONGEST SILENCE INSIDE THE ANSWER, S	ANSWER ENDED AFTER, S
Let in at once, never thrown out	214	1 to 127	2.7 to 12.9	345 to 478
Let in at once, then thrown out	47	127 to 154	42.1 to 171.0	513 to 588
Let in only when room came free	139	387 to 478	0.2 to 6.6	627 to 630

MEASURED

A stream that was thrown out fell silent for 42 to **171 s** and then went on: its answer is complete, 2,000 tokens like every other. The first wave waited longest, since room came free only when the first answers ended.

MEASURED

The 139 requests that were never let in at first got their first token after 387 to 477 s, behind the streams that had been thrown out, and ended last, after 630 s.

MEASURED

When the first answers ended the server read no new prompt for 42 s. In that time 64 answers ended, the line shrank by 52, and the engine made less than one step a second.

READING, NOT MEASUREMENT

My reading: in those seconds the server computed the thrown-out streams again, prompt and the part of the answer already written. That is the second price of throwing a stream out: its work is redone, and while it is, every running stream gets about one token a second instead of ten.

Against the same plan with room kept free

THE SAME SERVER AND PLAN, ONE SETTING APART	THIS RUN	10% OF THE CACHE KEPT FREE (`--WATERMARK 0.1`)
streams let in at first	261	235
streams thrown out	47	21
longest silence inside one answer, s	171.0	72.2
until all 400 had ended, s	629.7	603.6

Run `20261125T230017Z-series-watermark`: the same settings, the same default attention kernel, the same cache of 1,826,476 token slots, the same plan. One run each. That run has its own document.

What the card was doing

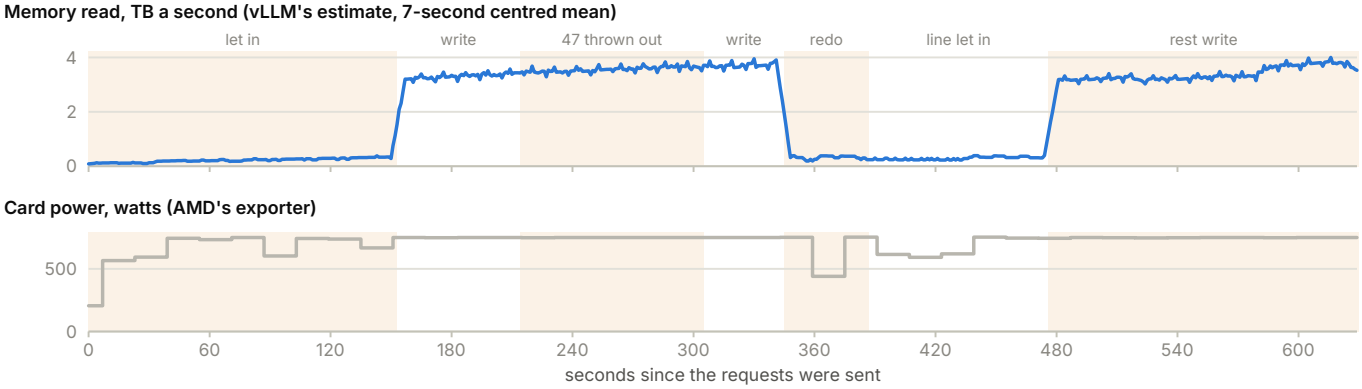


Figure 8 Underneath the same plan. Memory read is vLLM's estimate from the model's shape, not a hardware counter; power is AMD's device exporter, which refreshes every 11 to 16 s. Writing answers for some 200 long streams is the phase that moves memory; reading prompts is the phase that computes (the compute estimate is on page 3).
Source: plan-kv-growth/timeline.json

What I take from it

● A cache pushed past full costs time, not requests: all 400 of this plan ended well, 47 of them after being thrown out and let back in.

● The server throws out the streams it let in last.

◌ It lets them back in before anything that never got in: the times fit, though I did not see the line itself.

○ The cache gauge moves in whole blocks, 2,354 to the cache, and the two waves are the counts that the block arithmetic asks for.

◑ vLLM's estimate of memory read is 3.7 TB a second while 214 long streams write, 69.4% of AMD's stated peak for the card's memory.

◌ Letting a request in is expensive for everyone already running: each time the server reads prompts, the writing all but stops.

Limits

- One plan, played once. Every stream has the same length here, so all of them need their next block within seconds of each other; a mix of lengths would spread that out.
- Prompts are random tokens: nothing repeats, and answers are not readable text.
- The blocks are my reading of the gauge. What a block holds inside the server I did not look up.

KEEP IN MIND

Over the whole run the server did turn away two requests, on purpose: the context ladder's two prompts made too long for the model. This plan has none.

● measured: read from a run file · ○ computed: arithmetic on measured values · ◑ estimate: vLLM's, from the model's shape · ◌ my reading: interpretation, not measurement

Every number in this document was read from these files at build time (sha256, first 12 characters):

`FACTS.json` 160f5ca0d495 · `GLOSSARY.md` 98323be796af · `plans/kv-growth.json` 89c49daf377d · `runs/20261125T201346Z-series/plan-kv-growth/client.jsonl` a3a0ed7fb3ae · `runs/20261125T201346Z-series/plan-kv-growth/requests.jsonl` 38cbadbcd5d8 · `runs/20261125T201346Z-series/plan-kv-growth/samples.jsonl` 015baec3f8a7 · `runs/20261125T201346Z-series/plan-kv-growth/timeline.json` 630fd28b7c90 · `runs/20261125T201346Z-series/summary.json` aab3940eb12e · `runs/20261125T230017Z-series-watermark/summary.json` 55f891019fda

97 values bound by path, 8 figures drawn from data. built 2026-11-26 02:37Z, explainer-pdf-system v2.



CREATED AND DIRECTED BY

Peter Styk

HPC & AI Platform Engineer

GPU inference, data and performance

linkedin.com/in/peterstyk · github.com/styk-tv

Questions about a run? Connect on LinkedIn.

LinkedIn