QWEN3.8-27B · OVERLOAD STAGED LOAD, OPEN LOOP

Overload costs time, never a request

Requests arrive at 2, 4, 6, 8, then 12 a second, whether the server keeps up or not. It keeps up to 4, falls behind at 6, fills its cache at 8 and holds a line at 12. Every one of the 1,399 gets its answer, and the backlog is gone 81 seconds after arrivals stop.

Qwen3.8-27B · one AMD Instinct MI300X · vLLM 0.27.1 **Run** 20261125T201346Z-series **Changed** nothing: baseline settings, 1,024 stream slots
Load open loop: 1,024 tokens in, 256 out, 2 to 12 arrivals a second

I sent one server requests on a timer instead of on its answers: 2 a second, then 4, 6, 8 and 12, for 45 seconds each, every prompt 1,024 tokens and every answer 256. It kept up to 4 a second, fell behind at 6, filled its cache at 8 and held a line at 12. It refused nobody. When arrivals stopped, 813 requests were open; 81 seconds later every one of the 1,399 had its answer.

1,399
of 1,399 requests ended well: none refused, none failed, none cut

81 s
to empty a backlog of 813 open requests once arrivals stopped

36.4 s
median wait for a first token at 12 arrivals a second; 0.31 s at 4

11.1 times
as long for a whole answer at 12 a second as at 2, on average

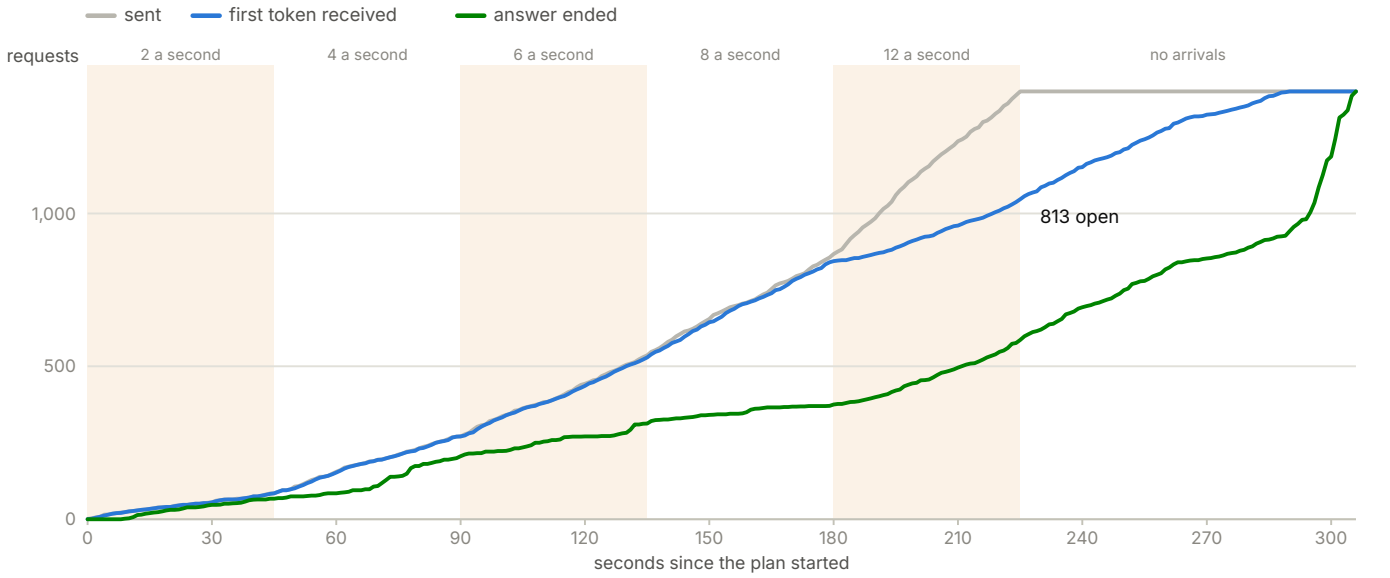

Figure 1 Running totals over the plan: requests sent (grey), requests that had their first token (blue), requests whose answer had ended (green). The height between grey and green is the number of requests open at that moment; the distance to the right, from grey to green, is how long the requests sent then took. All three lines end at 1,399. Shaded bands are the stages, labelled with the arrivals asked for.
Source: plan-overload-open/requests.jsonl (t_sent, t_first, t_end of every request); stage times from client.jsonl

2 a second
keeps up: 23 open at most, first token after 0.19 s

4 a second
keeps up: 91 open at most, first token after 0.31 s

6 a second
falls behind: open requests climb to 223 and do not level

8 a second
the cache fills: 470 streams running, 99.8% in use

12 a second
a line forms: 329 waiting, first token after 36.4 s

MEASURED
In this plan nothing was refused and nothing failed. All 1,399 requests got HTTP 200 and their full 256 tokens. The load generator cut none: it allowed 240 s for the line to empty, and 81 were enough.

MEASURED
The price was time, in two places. From 6 arrivals a second the answers themselves slow down: green falls away from blue. Only at 12 does a line form in front of the first token: blue falls away from grey.

The machine, and who sets the pace

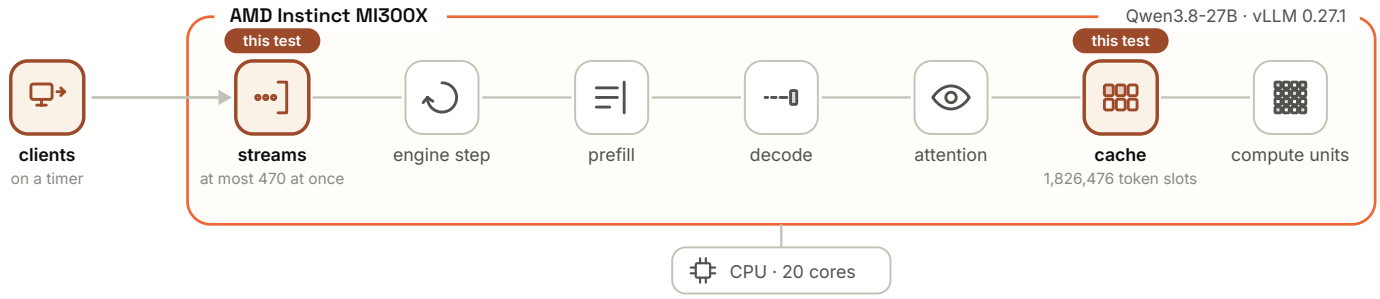

Figure 2 The card as every document in this series draws it: seven parts wired left to right, the clients outside, the host CPU below. In this test the clients set the pace, not the server. Lit: where requests arrive, where they run, and where their state is kept.
Source: FACTS.json; plans/overload-open.json; summary.json server.kv_cache_tokens, plans.plan-overload-open.stages[].running_max

An open loop

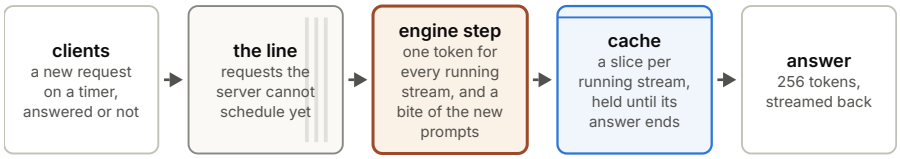

Figure 3 An open loop: a new request is sent on a timer, whether or not the earlier ones have been answered. What the server cannot schedule yet waits in its own line, and that line has no limit.
Source: loadgen.py (rate stages); plans/overload-open.json

1. The plan plays five stages back to back, 45 seconds each: 2, 4, 6, 8 and 12 new requests a second. Then no arrivals, until every open request has ended.
2. The gaps between requests are random around the stage's rate, so a stage sends about its rate, not exactly: 532 requests at 12 a second, where 540 is the mean.
3. Every prompt is 1,024 random token ids, so no two prompts share a prefix; every answer is forced to 256 tokens.
4. Beside it a sampler reads the server's counters, the card and the CPU once a second.

Closed loop (the staircase)

A fixed number of requests is kept open; a new one starts only when one ends. The server sets the pace and is never sent more than it finishes.

Open loop (this test)

Arrivals keep their own pace. If the server is slower, open requests pile up: first as running streams, then as a line.

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.

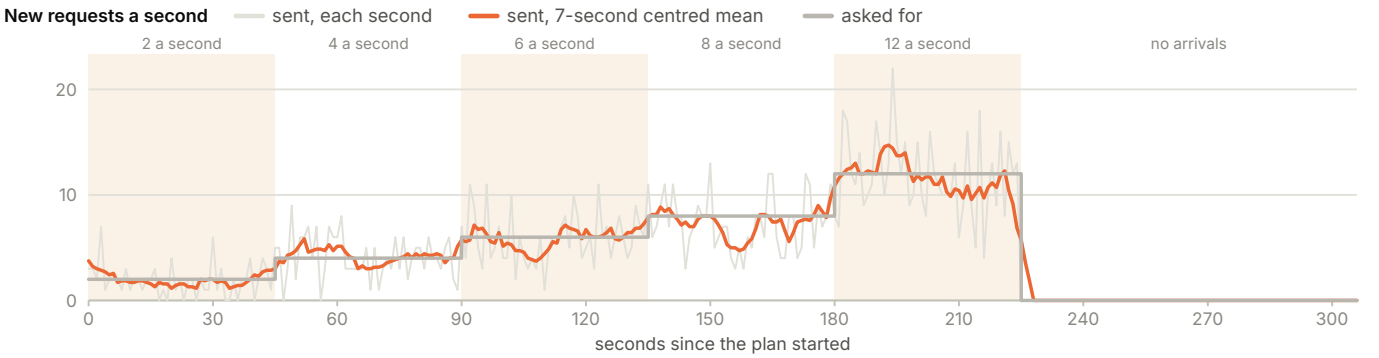

Figure 4 What was asked for and what was sent. The gaps are random, so each second differs (thin grey); the 7-second centred mean follows the steps of the plan.
Source: plan-overload-open/timeline.json (target, sent)

It keeps up to 4 a second, then requests pile up

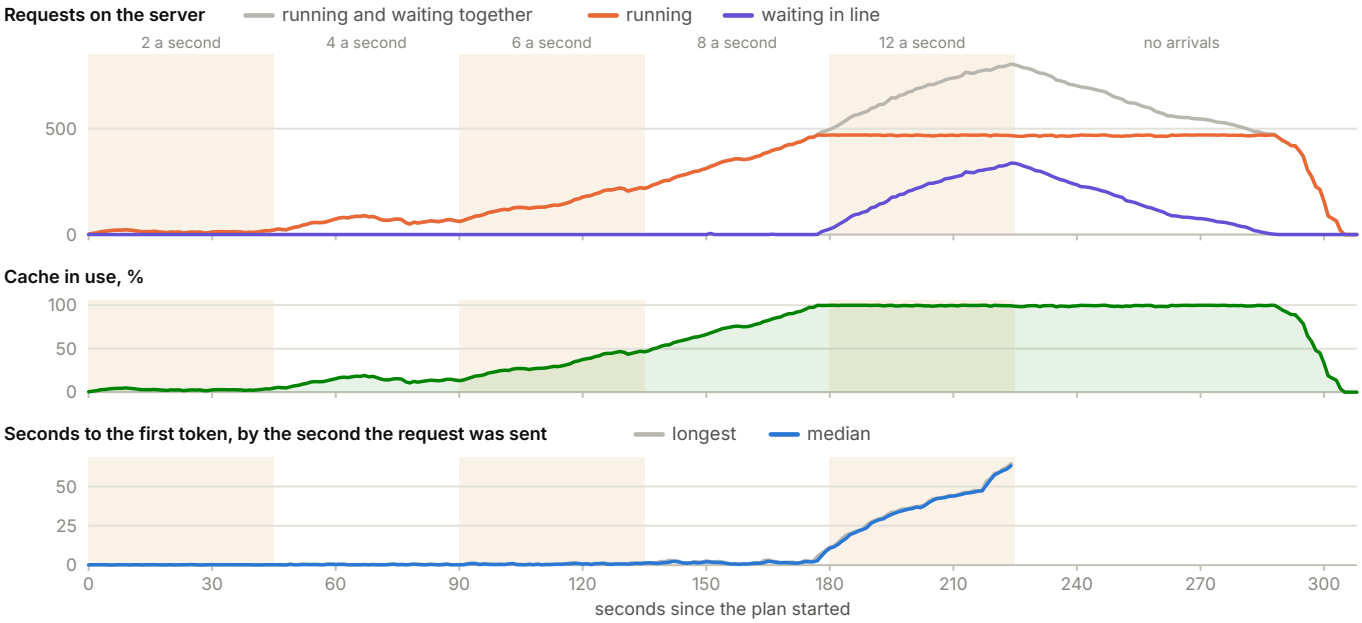

Figure 5 The plan second by second. Top: requests running on the server, waiting in its line, and the two together. Middle: the cache. Bottom: how long the requests sent in each second then waited for a first token, median and longest.
Source: plan-overload-open/timeline.json (vLLM's gauges, and requests.jsonl by the second sent)

MEASURED

At 4 a second the running streams peaked at 90 and were back at 65 when the stage ended. At 6 they climbed from 62 to 221 and never levelled: the server had fallen behind, with nothing waiting yet and the cache at most 46.5% full.

MEASURED

At 8 the cache filled: its gauge reached 99.8% with 470 streams running, and went no higher. From there a request is let in only when another ends, and the server's reason for every waiting request is "capacity". At 12 the line grew through the whole stage, to **337 waiting** in its last second.

The exact record: the line

ARRIVALS ASKED FOR	SENT	ENDED	OPEN AT MOST	RUNNING AT MOST	WAITING AT MOST	CACHE AT MOST, %	FIRST TOKEN, MEDIAN S	99TH PERCENTILE S	IN LINE, SERVER'S TIMER, S
2 a second	85	67	23	23	0	4.9	0.19	0.35	0.0
4 a second	186	139	91	90	0	19.1	0.31	0.73	0.0
6 a second	265	107	223	219	0	46.5	0.64	1.42	0.0
8 a second	331	63	483	470	14	99.8	1.60	9.25	0.0
12 a second	532	210	813	470	329	99.8	36.38	63.14	0.0
no arrivals	0	813	813	470	336	99.8	-	-	23.6

The exact record: the engine

ARRIVALS ASKED FOR	PROMPT TOKENS/S	ANSWER TOKENS/S	BOTH	ENGINE STEPS/S	TOKENS PER STEP	MS PER STEP	RUNNING, MEAN	ANSWER TOKENS/S PER RUNNING STREAM	TFLOPS, EST.
2 a second	1,953	436	2,389	33.58	71	30	13	33.31	109
4 a second	4,309	930	5,239	15.65	335	64	61	15.19	239
6 a second	5,906	918	6,824	6.67	1,022	150	146	6.31	311
8 a second	7,166	751	7,918	2.16	3,662	463	347	2.17	360
12 a second	4,548	1,788	6,336	3.81	1,661	262	469	3.81	291
no arrivals	4,390	1,745	6,135	5.62	1,091	178	423	4.12	282

One row per stage, from `summary.json`. Ended: requests whose answer ended in the stage, whenever they were sent. First-token times are for requests sent in it; the server's timer is vLLM's own queue time, averaged over the requests that ended in it. "Both", "ms per step" and "per running stream" are computed; TFLOPS is vLLM's estimate. A row spans 44 s of a 45 s stage, so its last second is missing: the row for 12 a second has 329 waiting at most, the per-second record 337. "Open" is the generator's count; it differs from the server's running plus waiting by up to 51 for a second or two, while requests are on their way in or out.

The server keeps reading new prompts, and pays for it in answers

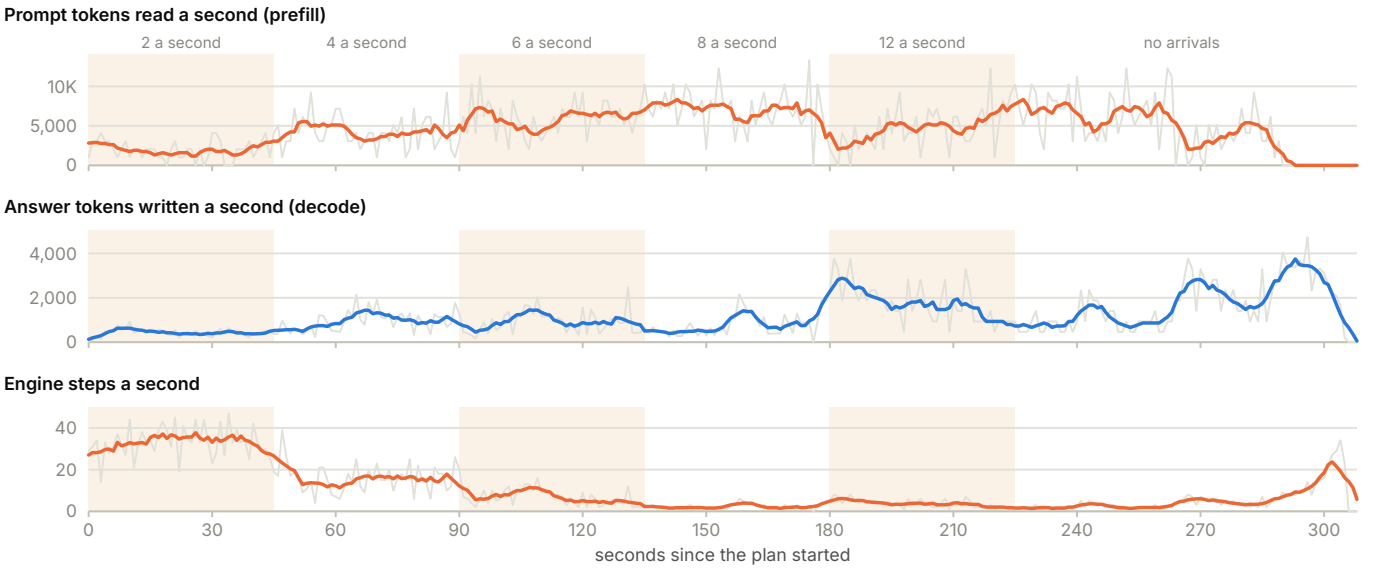

Figure 6 What the engine did, second by second: each second thin and grey, a 7-second centred mean in colour. Prompt tokens read climb with the arrivals; answer tokens written do not, until the line forms at 12 a second. Steps become fewer.
Source: plan-overload-open/timeline.json (vLLM's token and step counters)

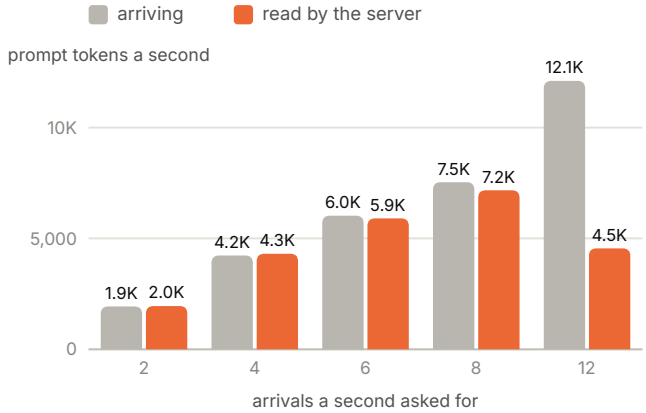

Figure 7 Prompts: tokens arriving a second and tokens the server read, by stage. It reads them as they come, up to 8 a second.
Source: summary.json stages[].sent, server_prompt_tok_s; arriving = sent x prompt length / stage seconds, computed

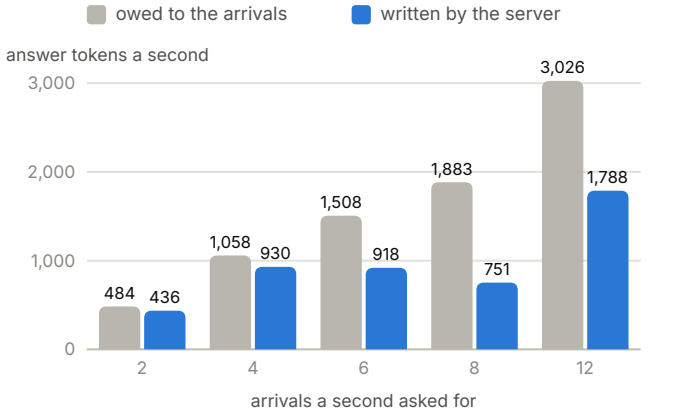

Figure 8 Answers: tokens owed to the requests that arrived and tokens the server wrote. The shortfall is here.
Source: summary.json stages[].sent, server_output_tok_s; owed = sent x answer length / stage seconds, computed

COMPUTED

Every stream gets one token per engine step, and the record on the page before shows it: answer tokens a second per running stream equal the engine's steps a second (33.3 and 33.6 at 2 a second, 2.17 and 2.16 at 8). So an answer of 256 tokens needs 256 steps, and at **2.16 steps a second** that is 119 s.

MEASURED

The steps get long because they carry the new prompts: 71 tokens a step at 2 a second, 3,662 at 8. Eight arrivals bring 8,192 prompt tokens a second on average, which is what one step may hold (8,192 tokens, the server's default).

READING, NOT MEASUREMENT

◌ The line is what saves the answers. Once the cache is full the server cannot let every arrival in, so less of each step goes to new prompts (4,548 prompt tokens a second at 12 against 7,166 at 8), steps come faster (3.81 a second) and output more than doubles, from 751 to 1,788 answer tokens a second. The numbers are measured; that one causes the other is my reading.

COMPUTED

The line forms when the cache is full, and it is full at 470 streams: 1,826,476 token slots / 470 = 3,886 slots for each running stream, 2,606 more than the 1,280 tokens a stream can hold. On the same server the staircase's shorter streams took 3,106 each. ◌ My reading: a stream keeps a fixed amount of state besides its tokens. Arithmetic on the cache size, not read from vLLM's source.

Later and slower, never refused

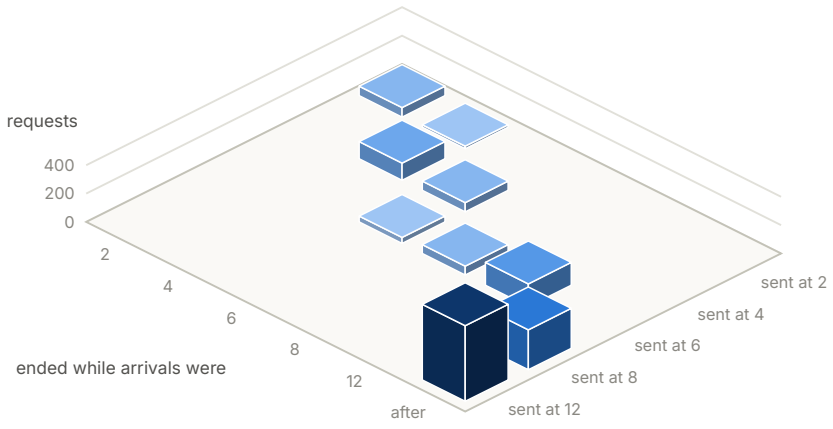

Figure 9 Requests by the stage in which they were sent (back to front) and the stage in which their answer ended (left to right). A view only: the exact values are in the matrix beside it.
Source: plan-overload-open/requests.jsonl (stage, t_end); stage times from client.jsonl

	ended while arrivals were 2	4	6	8	12	after
sent at 2	67	18	0	0	0	0
sent at 4	0	121	65	0	0	0
sent at 6	0	0	42	63	160	0
sent at 8	0	0	0	0	50	281
sent at 12	0	0	0	0	0	532

darker = more: requests

Figure 10 The same values, exactly. "After" is the time with no arrivals.
Source: plan-overload-open/requests.jsonl; client.jsonl

Read along a row. Requests sent at 2 and at 4 a second mostly ended in their own stage. Of those sent at 6, most ended two stages later. Not one request sent at 8 or at 12 a second ended in the stage it was sent in. The column sums are the "ended" column of the record.

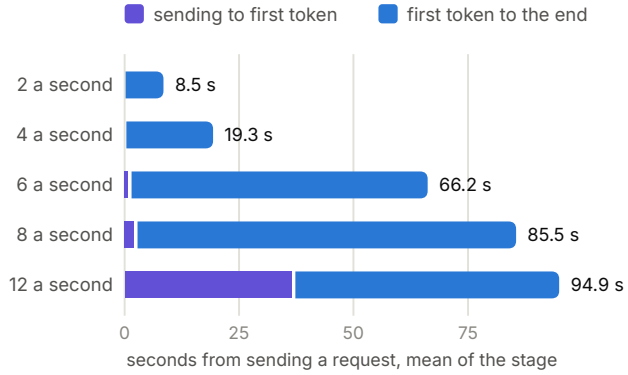

Figure 11 Where a request's time went, by the stage in which it was sent. Until the line forms, almost all of it is the answer being written slowly.
Source: plan-overload-open/requests.jsonl (t_sent, t_first, t_end), means per stage

COMPUTED

A whole answer took 8.5 s on average for requests sent at 2 a second and 94.9 s for those sent at 12. No request took longer than 104.5 s.

MEASURED

The first token came after 0.19 s at the median at 2 a second, 1.6 s at 8 and **36.4 s** at 12, when 362 requests waited more than half a minute for it.

The exact record, by the stage in which a request was sent

SENT AT	REQUESTS	FIRST TOKEN: MEDIAN S	LONGEST S	MEAN S	FIRST TOKEN TO THE END: MEAN S	WHOLE ANSWER: MEAN S	LONGEST S	LONGEST SILENCE IN AN ANSWER, S
2 a second	85	0.19	0.35	0.21	8.3	8.5	13.7	0.36
4 a second	186	0.31	0.79	0.36	19.0	19.3	30.5	1.49
6 a second	265	0.64	1.56	0.66	65.5	66.2	83.7	4.14
8 a second	331	1.60	9.69	1.96	83.6	85.5	94.8	5.61
12 a second	532	36.38	64.55	36.45	58.5	94.9	104.5	5.66

First-token times and the longest silence are from `summary.json`; the means are computed from `requests.jsonl`. A silence is the longest gap between two pieces of text inside one answer, the largest among the stage's requests.

What the card was doing

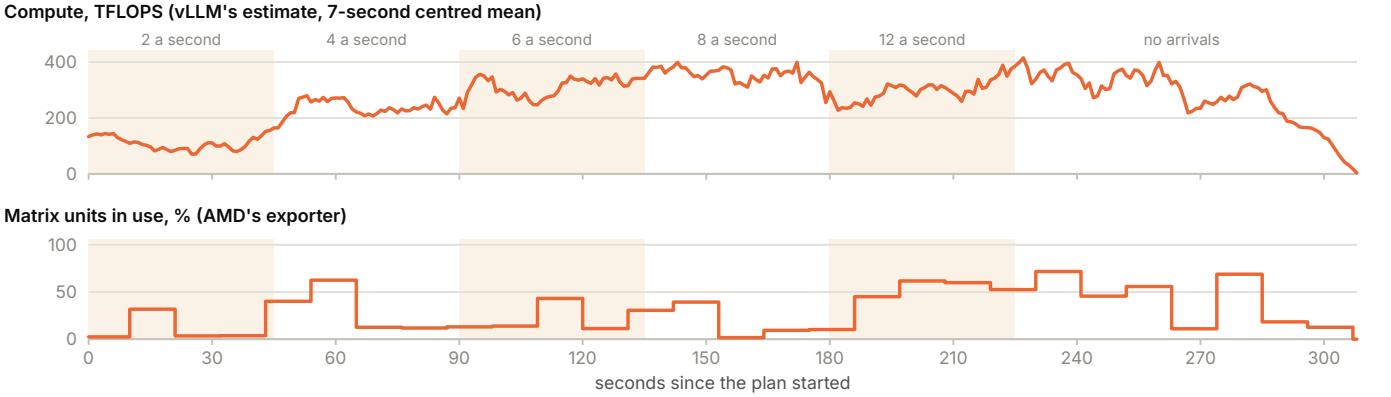

Figure 12 Underneath the same plan. Compute is vLLM's estimate from the model's shape, not a hardware counter. Matrix units in use is a reading of AMD's device exporter, which refreshes every 11 to 16 s. The card drew 747 to 751 W as stage means, and the engine process used 2.0 of the 20 CPU cores throughout: two flat lines, not drawn.
Source: plan-overload-open/timeline.json; summary.json stages[].watts, engine_cores

What I take from it

● All 1,399 requests of this plan ended well, with arrivals up to 12 a second. None was refused, none failed, none was cut.

● The backlog of 813 open requests was gone 81 s after arrivals stopped.

○ The most the server handled in a stage was 7,918 prompt and answer tokens a second, at 8 a second. The arrivals at 6 a second already brought 7,538.

◑ vLLM's estimate of the compute is highest, 360 TFLOPS or 27.6% of AMD's stated peak, in the stage with the least output: the card was busy reading prompts.

◌ An overloaded server makes its clients wait; it does not turn them away. The line shields the streams that are already running.

Limits

- One plan, played once, with one seed for the random gaps. Each stage starts where the last one left off, so a stage is not a steady state: at 6 a second the server was still falling behind when the stage ended.
- Prompts are random tokens: nothing repeats, and answers are not readable text.
- No other finished run has played this plan, so nothing here is compared with another server.
- The CPU was 10.6 to 11.2% busy in every stage, of 20 cores.

KEEP IN MIND

I found no setting in this vLLM's help text for a longest line, for refusing work under overload, or for a request timeout.

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

Words used

Open loop
requests arrive at a fixed rate whether or not the server keeps up. The line can grow.

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

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.

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`.

Every number in this document was read from these files at build time (sha256, first 12 characters):
`FACTS.json` 160f5ca0d495 · `GLOSSARY.md` 98323be796af · `plans/overload-open.json` 204fa20d344e · `runs/20261125T201346Z-series/plan-overload-open/client.jsonl` c20eca48ac28 · `runs/20261125T201346Z-series/plan-overload-open/requests.jsonl` 33e1c43237ae · `runs/20261125T201346Z-series/plan-overload-open/timeline.json` 3709ccddee19 · `runs/20261125T201346Z-series/summary.json` aab3940eb12e

107 values bound by path, 12 figures drawn from data. built 2026-11-26 02:38Z, 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