Serving de LLMs con vLLM

Para levantar un LLM con vLLM con la documentación abierta, saber cuántas conversaciones caben en tu GPU y diagnosticar por qué el primer token tarda, las respuestas se cortan o el servidor no arranca por memoria. En el C10, un modelo de ≈ 70B con pesos FP8 corre en 2 H100 por réplica: vLLM lee el prompt completo de una vez (prefill), guarda lo leído en el KV cache y escribe la respuesta token por token (decode), mezclando en cada paso las conversaciones de muchos usuarios. Cada sección trae la sintaxis como plantilla con variables, las piezas, un ejemplo real del curso, una figura, un truco para recordarla, la trampa típica y un laboratorio opcional. En la versión web, al final hay un diagnosticador de síntomas, una calculadora de VRAM y KV cache y un quiz de errores.

Cómo recordarla · Trampa típica · *variable* la cambias tú · Ejemplos en verde

EN LAS FIGURAS: pesos · KV cache · prefill o espera en la cola · decode o tokens generados · ok, libre o dentro del SLO · OOM, preemption, fuera del SLO o rechazado · overhead, activaciones, reservado u ocioso

0 Números de bolsillo: tokens, bytes por parámetro, KV cache, TTFT y TPOT, puertos y endpoints

¿Qué números de un LLM servido con vLLM conviene saberse de memoria para dimensionar una réplica y leer sus métricas en segundos?

SINTAXIS CON VARIABLES

`curl -s localhost:`*puerto*`/v1/models`

El primer chequeo: el servidor contesta y sirve el nombre que esperas, el de `--served-model-name`. Puerto de bolsillo: `8000`, el default de `vllm serve` y el del C10.

`curl -s localhost:`*puerto*`/tokenize -H 'content-type: application/json' -d '{"model": "`*modelo*`", "messages": [{"role": "user", "content": "`*texto*`"}]}'`

Cuenta tokens con el **tokenizer** (el que parte el texto en tokens) y el **chat template** (la plantilla que envuelve tus mensajes con las marcas del modelo) del modelo servido. Devuelve `count`, `max_model_len` y la lista de `tokens`.

`GPU KV cache size:` *tokens* **`tokens, Maximum concurrency for`** *max_model_len* **`tokens per request:`** *x* **`x`**

La línea del log de arranque con tu capacidad real: cuántos tokens de **KV cache** (lo que el modelo ya leyó de cada conversación, guardado en la GPU) caben, y cuántas secuencias del largo máximo a la vez. Con los números del C10, la cuenta da 208,736 tokens y 25.48x; vLLM no corrió en la referencia, así que el número real sale del log.

`curl -s localhost:`*puerto*`/metrics | grep -E '^vllm:(num_requests_running|num_requests_waiting|kv_cache_usage_perc)'`

Los tres **gauges** (números que suben y bajan) de capacidad: requests en el batch, requests en la cola y fracción del KV cache ocupada (1 = 100 %). La utilización de la GPU no sirve para esto: el **decode** (generar la respuesta token por token) marca ≈ 100 % con 5 secuencias o con 90.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
puerto	dónde escucha vllm serve	8000: el default y el del C10
modelo	el nombre que pide el cliente: el de --served-model-name	chat-70b-fp8-2026-09-28
texto	lo que quieres contar en tokens	Hola, ¿dónde está mi pedido?
tokens, *max_model_len*, *x*	lo que imprime vLLM al arrancar: tokens de KV, el contexto máximo y cuántas secuencias de ese largo caben	208,736 · 8,192 · 25.48 con la cuenta del C10
L	capas del modelo (num_hidden_layers en el config.json)	80
H_{kv}	KV heads: cuántos pares de K y V tiene cada capa (num_key_value_heads)	8, de 64 heads de atención (GQA)
d_{head}	tamaño de cada head (head_dim; si el config no lo trae, hidden_size ÷ num_attention_heads)	128 = 8,192 ÷ 64
b_{kv}, b_w	bytes por número: en el KV cache y en los pesos	2 (BF16, --kv-cache-dtype auto) y 1 (pesos FP8)
P	parámetros del modelo	70.6 B (miles de millones)
N_{gpu}, M_{gpu}	GPUs por réplica y memoria de cada una	2 × 80 GB (TP=2)
u	--gpu-memory-utilization: la fracción de la memoria de cada GPU que vLLM se permite usar	0.90 en el C10; 0.92 si no la escribes (desde v0.20)

VARIABLE	QUÉ ES	EJEMPLO
overhead	lo que no es pesos ni KV: activaciones, CUDA graphs y buffers de comunicación	≈ 5 GB en la réplica (estimación del C10)
T_{seq}, N_{out}	tokens de una secuencia (prompt + salida) y tokens de salida	2,250 = 2,000 + 250; N_out = 250
TTFT, TPOT	tiempo hasta el primer token; tiempo promedio por token después del primero	≈ 0.5 s y ≈ 40 ms (los SLOs, las metas en p95: 1 s y 50 ms)
C, λ, W	secuencias vivas a la vez, llegadas por segundo y cuánto dura cada request (ley de Little)	630 = 60 × 10.5

FÓRMULAS

$$\text{KV}_{\text{token}} = 2 \times L \times H_{kv} \times d_{head} \times b_{kv} = 2 \times 80 \times 8 \times 128 \times 2\ \text{B} = 327{,}680\ \text{B} = 320\ \text{KiB}$$

$$\text{pesos} = P \times b_w = 70.6\ \text{B} \times 1\ \text{B} \approx 70.6\ \text{GB} \qquad (\text{BF16: } 141.2\ \text{GB; INT4: } \approx 35.3\ \text{GB})$$

$$\text{KV}_{\text{libre}} = N_{gpu} \times M_{gpu} \times u - \text{pesos} - \text{overhead} = 2 \times 80 \times 0.90 - 70.6 - 5 \approx 68.4\ \text{GB}$$

$$N_{seq} = \left\lfloor \frac{\text{KV}_{\text{libre}}}{T_{seq} \times \text{KV}_{\text{token}}} \right\rfloor = \left\lfloor \frac{68.4\ \text{GB}}{2{,}250 \times 327{,}680\ \text{B}} \right\rfloor = \lfloor 92.8 \rfloor = 92$$

$$\text{E2E} = \text{TTFT} + (N_{out} - 1) \times \text{TPOT} \approx 0.5\ \text{s} + 249 \times 40\ \text{ms} \approx 10.5\ \text{s}$$

$$C = \lambda \times W = 60\ \text{rps} \times 10.5\ \text{s} = 630 \Rightarrow 630 \div 16 \approx 39.4\ \text{secuencias por réplica} < 92$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
1 token ≈ ¾ de palabra en inglés	La unidad en que el modelo lee y escribe: un pedazo de palabra. La regla de ¾ es la de OpenAI; con el tokenizer de Llama 3.1 se midieron ≈ 1.23 tokens por palabra en inglés y ≈ 1.61 en español, en noticias.	2,000 tokens de prompt ≈ 1,240 palabras en español
2 · 1 · 0.5 bytes por parámetro	BF16 (16 bits, como se entrenó), FP8 o INT8 (8 bits) e INT4 (4 bits). Los pesos pesan parámetros × bytes.	70.6 B: 141.2 · 70.6 · ≈ 35.3 GB; el C10 sirve FP8
KV por token = 2 × capas × KV heads × head_dim × bytes	Lo que ocupa en la GPU guardar lo que el modelo ya leyó de un token: una K y una V por capa y por KV head. No depende de los parámetros.	320 KiB en BF16 y 160 KiB en FP8; Llama 3.1 8B: 128 KiB
GQA (grouped-query attention)	Varios heads de atención comparten un mismo par de K y V: menos KV heads y un KV cache más chico. Se lee del config.json.	64 heads y 8 KV heads: 8× menos KV que con un par por head
--gpu-memory-utilization 0.92	La fracción de la memoria TOTAL de cada GPU que vLLM se permite usar entre pesos, activaciones y KV. Es el default desde v0.20; hasta v0.19 era 0.9.	el C10 fija 0.90: 2 × 80 GB × 0.90 = 144 GB
Bloque de 16 tokens	PagedAttention reparte el KV cache en bloques de 16 tokens (--block-size): cada secuencia desperdicia, como mucho, lo que le falta a su último bloque.	2,250 tokens = 141 bloques
GPU KV cache size	La línea del arranque con la capacidad que midió vLLM con tu modelo, tu versión y tus GPUs. Manda sobre cualquier cuenta.	la cuenta del C10 da 208,736 tokens y 25.48x para 8,192
TTFT · TPOT · ITL	TTFT (time to first token): lo que se espera hasta el primer token. TPOT (time per output token): el promedio entre tokens después del primero. ITL (inter-token latency): el hueco entre dos tokens seguidos.	SLOs del C10: TTFT p95 ≤ 1 s en el gateway y TPOT p95 ≤ 50 ms
E2E = TTFT + (N − 1) × TPOT	El request completo: el primer token y después un paso de decode por cada uno de los demás.	0.5 + 249 × 0.040 ≈ 10.5 s
Throughput · goodput	Throughput: tokens de salida, o requests, por segundo de la réplica. Goodput: requests por segundo que cumplen todos los SLOs, uno por uno. Se planea con el goodput.	≈ 6 rps por réplica (72 en vuelo, en el barrido sintético); planeado al 70 %: 4.2
8000 · /v1/chat/completions · /v1/completions · /v1/models	El puerto por defecto y la API compatible con OpenAI que expone vllm serve.	el gateway llama a http://vllm-70b.c10.svc.cluster.local:8000/v1
/health · /metrics · /tokenize	/health: responde 200 si el motor está vivo. /metrics: las métricas vllm: para Prometheus. /tokenize: cuenta tokens como los cuenta el modelo.	las tres probes del C10 llaman a /health

PIEZA	QUÉ HACE	EJEMPLO
vllm:num_requests_running · vllm:num_requests_waiting · vllm:kv_cache_usage_perc	Requests en el batch, requests en la cola y fracción del KV cache ocupada (1 = 100 %).	**KVCacheSaturation: KV > 95 % y requests esperando, 15 min**
vllm:time_to_first_token_seconds · vllm:inter_token_latency_seconds	Los histogramas de TTFT y de ITL que mide el propio vLLM, del lado del servidor.	**el TTFT del SLO se mide en el gateway, que además ve la red y los proxies**
H100 SXM: 80 GB · 3.35 TB/s · ≈ 1,979 TFLOPS en FP8	La GPU del C10. Su memoria decide cuántas secuencias caben; su ancho de banda, cuánto tarda cada token; sus TFLOPS (billones de operaciones por segundo), el prefill. La hoja de NVIDIA dice 3,958 TFLOPS «with sparsity» (con la mitad de los pesos en cero, algo que casi ningún modelo usa): el denso es la mitad.	**una réplica = 2 H100 unidas por NVLink (el enlace rápido entre GPUs); 8 por p5.48xlarge, ≈ US$55/h**
US$2.50 y US$10 por millón de tokens	Precios de lista ilustrativos del C10 para el proveedor, de entrada y de salida: un token de salida cuesta 4×.	**un request de soporte (2,000 + 250): US$0.0075**

FLAG	DEFAULT (V0.30.0 Y V0.32.0)	EN EL C10	POR QUÉ IMPORTA
--port	8000	8000	el Service y el gateway apuntan ahí
--gpu-memory-utilization	0.92 (0.9 hasta v0.19)	0.90	explícito: la capacidad no cambia con un upgrade
--max-model-len	el del config.json del modelo (131,072 en Llama 3.3)	8192	prompt + salida; acepta auto (el máximo que cabe) y sufijos: 8k = 8,000 y 8K = 8,192
--max-num-seqs	con vllm serve: 1,024 en GPUs de ≥ 70 GiB que no son A100 (H100, H200, B200); 256 en las demás	128	techo de secuencias por paso: el KV del C10 da para 92 de 2,250 tokens
--max-num-batched-tokens	con vllm serve: 8,192 en la H100; 16,384 desde 160 GiB por GPU; 2,048 en la A100 y en las chicas	8192	tokens por paso: más mejora el TTFT; menos, el ITL
--block-size	16	sin cambiar	una secuencia de 2,250 tokens ocupa 141 bloques
--kv-cache-dtype	auto: el dtype del modelo (BF16)	auto	fp8 duplica las secuencias; entra solo si pasa el golden set
--enable-prefix-caching · --enable-chunked-prefill	activados si el modelo los soporta	explícitos	el plan del C10 cuenta con los dos
--tensor-parallel-size	1	2	70.6 GB de pesos FP8 más el KV no caben cómodos en 80 GB
--scheduling-policy	fcfs	priority	la documentación dice que con fcfs un priority ≠ 0 da error, pero el código de v0.30.0 y v0.32.0 no lo valida (la cola fcfs lo ignora): no cuentes con el error, pon la flag. Con priority, las preemptions caen primero sobre el batch
--watermark	0.0 (apagado)	sin cambiar	deja bloques libres al admitir, para no tener que desalojar después
--max-num-active-seqs	no existe en v0.30; desde v0.31, None	—	tope de secuencias corriendo sin achicar max_num_seqs
max_tokens del request	sin él, en chat: todo lo que queda de la ventana (en /v1/completions, 16)	el gateway pone el tope del tenant (1,024 en soporte)	una respuesta sin tope ocupa su KV hasta llenar la ventana
prompt + max_tokens > max_model_len	v0.30: 400; desde v0.32: entra y la salida se recorta a lo que queda	el gateway lo frena antes, con un 400 propio: prompt estimado ≤ 6,000 y max_tokens ≤ 1,024 en soporte	el upgrade cambia el comportamiento sin avisar (sección 2)

Para ver lo que usó tu servidor: el log de arranque imprime la configuración y la línea `GPU KV cache size`, y `/metrics` expone `vllm:cache_config_info`, con la configuración del cache en sus labels.

Defaults de vLLM v0.30.0 (el pin del C10, 22 de septiembre de 2026) y v0.32.0 (la versión actual, 18 de octubre) contra lo que fija el C10. Leídos en el código de las dos versiones (vllm/config, vllm/engine/arg_utils.py y vllm/renderers/params.py) y en la referencia de vllm serve. Son iguales en las dos salvo donde se dice.

EJEMPLO REAL · LA CUENTA DE VRAM DEL C10, A MANO

TEXTO reference/c10/capacity/vram_calc.py (líneas 3–12, el docstring)

```
La cuenta del caso, a mano, con la arquitectura de Llama 3.x 70B (la de #m04) y los números del caso:

  KV por token  = 2 (K y V) × 80 capas × 8 KV heads × 128 dims × 2 B (BF16)  = 327,680 B = 320 KiB
                  (en FP8: 1 B por elemento → 160 KiB)
  pesos en FP8  = 70.6 B params × 1 B                                        ≈ 70.6 GB
```

```
una réplica   = 2 × H100 (80 GB cada una) con tensor parallelism 2 (TP=2)
lugar para KV = 0.90 × 160 GB − 70.6 GB − ≈ 5 GB (runtime, activaciones, CUDA graphs; est.) ≈ 68.4 GB
una secuencia = 2,000 tokens de prompt + 250 de salida = 2,250 tokens × 320 KiB ≈ 0.737 GB
secuencias    = 68.4 ÷ 0.737 ≈ 92.8 → 92 enteras (el caso dice "≈ 93": es el cociente redondeado)
                con KV en FP8, el doble: ≈ 185
```

- La primera línea es la fórmula del KV cache con el config de Llama 3.x 70B: **80 capas**, **8 KV heads** y **128 dims**. Los 2 B por número son BF16, lo que da `--kv-cache-dtype auto` con este modelo; en FP8, la mitad.
- «70.6 B params × 1 B»: un checkpoint FP8 guarda un byte por peso. Uno real suele dejar en BF16 las dos tablas del vocabulario, embeddings (la de entrada) y lm_head (la de salida): ≈ 2.1 B parámetros, y entonces pesa ≈ 72.7 GB, no 70.6. Mide los safetensors (los archivos de pesos) que vas a servir.
- «0.90 × 160 GB»: el 0.90 es el `--gpu-memory-utilization` que fija el manifest. Los ≈ 5 GB de overhead son una estimación; vLLM los mide al arrancar con un paso de prueba y los resta antes de armar el KV cache.
- «92 enteras»: la cuenta supone todas las secuencias en su largo final. El «≈ 93» del curso es el cociente redondeado (92.77).
- Lo que el docstring aclara después: «80 GB» es el nominal. `nvidia-smi` muestra 81,559 MiB (≈ 85.5 GB) en una H100 SXM, y la fracción se aplica sobre eso: en la práctica, ≈ 78 GB para KV y 106 secuencias. El número que vale es el del log de arranque.

CÓMO RECORDARLA

Puerto 8000, bloques de 16 y `--gpu-memory-utilization` 0.92 si no la fijas. KV por token = 2 × capas × KV heads × head_dim × bytes: **320 KiB** en el 70B del C10. Una réplica de 2 × H100: 144 GB usables − 70.6 de pesos − ≈ 5 de overhead = **≈ 68 GB de KV ≈ 92 conversaciones de 2,250 tokens**. Y dos SLOs: **TTFT ≤ 1 s, TPOT ≤ 50 ms**.

TRAMPA

Confundir parámetros con bytes, y la cuenta con la capacidad. «70B» son 70.6 mil millones de parámetros: 141 GB en BF16 y 70.6 en FP8, más ≈ 2 GB por las tablas del vocabulario que un checkpoint FP8 suele dejar en BF16. Y la cuenta de servilleta no es tu capacidad: la que vale es `GPU KV cache size: N tokens, Maximum concurrency for 8,192 tokens per request: Kx` en el log. Además, `--gpu-memory-utilization` se aplica sobre la memoria que reporta la GPU (81,559 MiB ≈ 85.5 GB en una H100 «de 80 GB»), así que vLLM suele dar más KV que la cuenta nominal: ≈ 78 GB en vez de 68.

En el curso: M14 · KV cache: la memoria que decide la concurrencia · M4 · Cálculo de VRAM paso a paso · C10 · Estimaciones back-of-the-envelope · Números · Fórmulas de bolsillo · Hoja · Formulario de estimaciones · memoria de GPU

La cadena: de un prompt a un stream de tokens

¿Qué le pasa a un request de chat, en orden, desde que llega a vLLM hasta que la última palabra sale por SSE, y qué parte de la latencia pone cada eslabón?

SINTAXIS CON VARIABLES

`curl -sN` *url* `/v1/chat/completions -H 'content-type: application/json' -d '{"model": "` *modelo* `", "messages": [{"role": "user", "content": "` *texto* `"}], "stream": true, "max_tokens":` *n* `}'`

El request entero. `-N` le quita el buffer a curl: ves cada evento **SSE** (Server-Sent Events: líneas `data: {…}` que el servidor manda sin cerrar la conexión) apenas llega, y al final `data: [DONE]`.

`curl -s` *url* `/tokenize -H 'content-type: application/json' -d '{"model": "` *modelo* `", "messages": [{"role": "user", "content": "` *texto* `"}]}'`

El primer eslabón, solo: chat template y tokenizer. `count` es lo que entra de verdad, con las marcas de la plantilla y el encabezado del asistente.

`Engine 000: Avg prompt throughput:` *p* `tokens/s, Avg generation throughput:` *g* `tokens/s, Running:` *r* `reqs, Waiting:` *w* `reqs, GPU KV cache usage:` *k* `%, Prefix cache hit rate:` *h* `%`

La línea que vLLM escribe en su log cada 10 s (`VLLM_LOG_STATS_INTERVAL`; en DEBUG si está ocioso): la cadena entera en una línea. Si en esos 10 s hubo preemptions, agrega `Preemptions: N` antes de `GPU KV cache usage`.

`curl -s` *url* `/metrics | grep -E '^vllm:request_(queue|prefill|decode)_time_seconds_(sum|count)'`

Dónde se fue el tiempo, por fase: **cola** (waiting), **prefill** y **decode**. `_sum` ÷ `_count` da el promedio; para percentiles, `histogram_quantile` sobre los `_bucket`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
url	la base del servidor, vista desde donde corres curl	`localhost:8000 con kubectl port-forward a un pod`
modelo	el --served-model-name	`chat-70b-fp8-2026-09-28`
texto	el mensaje del usuario	`¿Dónde está mi pedido?`
n	tope de tokens de salida del request	`512 en el ejemplo del README; el gateway corta en 1,024`
p, *g*, *r*, *w*, *k*, *h*	tokens por segundo de prompt y de salida, requests corriendo y esperando, KV usado y aciertos del prefix cache	`una línea cada 10 s`
t_{cola}	lo que espera en waiting hasta entrar al batch	`≈ 0 con lugar libre; lo primero que crece en sobrecarga`
t_{prefill}	lo que tarda en leer los tokens del prompt que no estaban en caché	`≈ 1,000 de 2,000 con el system prompt en caché`
T_{prompt}, T_{cache}	tokens del prompt y tokens que ya estaban en el prefix cache	`2,000 y ≈ 1,000`
t_{red}	red, gateway y proxies entre vLLM y quien mide	`lo que separa el TTFT del servidor del que mide el gateway`
N_{out}	tokens de salida del request	`250 en promedio en el C10`

FÓRMULAS

$$\text{TTFT} \approx t_{\text{cola}} + t_{\text{prefill}}(T_{\text{prompt}} - T_{\text{caché}}) + t_{\text{red}}$$

$$\text{TPOT} = \frac{\text{E2E} - \text{TTFT}}{N_{out} - 1} \qquad \text{(la de vllm bench serve; con } N_{out} = 1 \text{ no se calcula)}$$

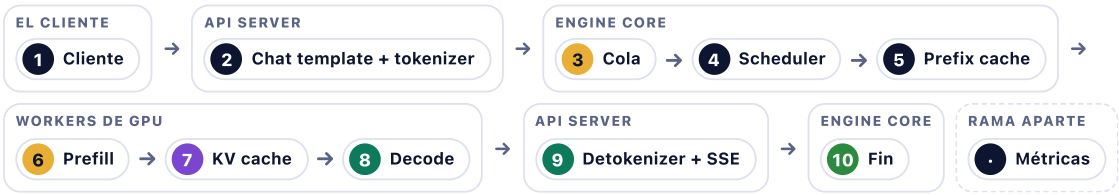
$$\text{E2E} = \text{TTFT} + (N_{out} - 1) \times \text{TPOT} \approx 0.5\,\text{s} + 249 \times 40\,\text{ms} \approx 10.5\,\text{s}$$

$$\text{TTFT}_{\text{cliente}} = \text{E2E} \qquad \text{si un proxy junta toda la respuesta antes de reenviarla}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`POST /v1/chat/completions`	Entra el request: messages, max_tokens y stream: true. En el C10, vLLM no autentica (con --api-key solo pediría una llave compartida, o varias, sin tenants ni cuotas): eso lo hace el gateway.	`el gateway agrega priority 0 y stream_options.include_usage`
`API server`	El proceso que atiende HTTP: aplica el chat template, tokeniza, valida el largo y, al final, convierte los tokens en texto y manda el stream. No toca la GPU.	`1 de los 4 procesos de una réplica TP=2: 1 API server, 1 engine core y 2 workers`
`Chat template`	Une los messages en un solo texto con las marcas del modelo: inicio, rol y fin de turno.	`<|start_header_id|>user<|end_header_id|> … <|eot_id|> en Llama 3.3`
`Tokenizer`	Parte el texto en tokens del vocabulario del modelo y los convierte en números.	`≈ 2,000 tokens de prompt`
`Validación de largo`	prompt + max_tokens contra --max-model-len. En v0.30, si se pasa, un 400 antes de la cola; desde v0.32 solo cuenta el prompt.	`8,192 en el C10`
`Engine core: scheduler`	El proceso que decide, en cada paso, qué secuencias corren y cuántos tokens procesa cada una: primero los decodes y, con el presupuesto que sobra, trozos de prefill.	`--max-num-batched-tokens 8192 y hasta 128 secuencias`

PIEZA	QUÉ HACE	EJEMPLO
Cola (waiting)	Donde espera un request si no hay lugar en el batch o bloques de KV libres.	`vllm:num_requests_waiting` y `vllm:request_queue_time_seconds`
Prefix cache	Si el prompt empieza con bloques que ya están en la GPU, los reusa: el prefill calcula solo lo nuevo.	`≈ 1,000 tokens de system prompt por producto`
Prefill	Lee todos los tokens del prompt de una vez y escribe su KV. Termina con el primer token: marca el TTFT.	`piso ≈ 71 ms para 2,000 tokens (≈ 36 con la mitad en caché); el README estima ≈ 0.3–1 s con la cola`
KV cache	Guarda la K y la V de cada token leído o generado, en bloques de 16; crece un token por paso.	`2,250 tokens × 320 KiB ≈ 0.74 GB al final`
Workers, uno por GPU	Corren el modelo. Con TP=2, cada GPU tiene la mitad de cada capa y la mitad de los KV heads, y las dos se sincronizan por NVLink en cada capa.	`2 H100 del mismo p5.48xlarge`
Decode	Un paso por token: lee los pesos y el KV de todo el batch y saca un token por secuencia. Marca el TPOT.	`≈ 40 ms por paso en la estimación del C10`
Sampling y parada	Elige el token siguiente (temperature, top_p) y para con el fin de turno del modelo, con un stop o con max_tokens.	`finish_reason stop; length si llegó a max_tokens`
Detokenizer y SSE	Convierte los tokens nuevos en texto y los manda como un evento data: por paso; al final, data: [DONE].	`el gateway reenvía cada evento apenas llega`
Cancelación	Si el cliente cierra la conexión, vLLM aborta la secuencia y libera sus bloques.	`el gateway la cierra y la cuenta como 499`



1 Cliente — EL CLIENTE

HTTP

```
POST /v1/chat/completions
```

stream: true y max_tokens 512. Antes, el gateway ya autenticó, cobró la cuota y admitió el request.

2 Chat template + tokenizer — API SERVER

HTTP `POST /tokenize`

Los messages pasan a un texto con marcas y a ≈ 2,000 números. 2,000 + 512 ≤ 8,192: entra.

3 Cola — ENGINE CORE

PROMQL

```
vllm:num_requests_waiting
```

Espera si no hay lugar ni bloques libres. En sobrecarga, este eslabón se come el TTFT antes que cualquier otro.

4 Scheduler — ENGINE CORE

FLAG

```
--max-num-batched-tokens 8192
```

En cada paso, primero los decodes de los que ya corren; con lo que sobra del presupuesto, trozos de prefill.

5 Prefix cache — ENGINE CORE

PROMQL

```
vllm:prefix_cache_hits_total
```

Los ≈ 1,000 tokens del system prompt ya están en bloques: no se recalculan.

6 Prefill — WORKERS DE GPU

PROMQL

```
vllm:request_prefill_time_seconds
```

Lee el resto del prompt de una vez, escribe su KV y saca el primer token. Es cómputo: marca el TTFT.

7 KV cache — WORKERS DE GPU

PROMQL

```
vllm:kv_cache_usage_perc
```

Bloques de 16 tokens; al final, 2,250 × 320 KiB ≈ 0.74 GB para esta conversación.

8 Decode WORKERS DE GPU

PROMQL

```
vllm:inter_token_latency_seconds
```

Un token por paso para cada secuencia del batch, cada ≈ 40 ms. Es memoria: marca el TPOT.

9 Detokenizer + SSE API SERVER

EVENTO SSE

```
data: {"choices": [{"delta": {"content": "…"}}]}
```

En cada paso, el texto nuevo sale como un evento. Al final: un evento con finish_reason, el de usage si se pidió y data: [DONE].

10 Fin ENGINE CORE

JSON

```
finish_reason: "stop"
```

El modelo cerró su turno. Sus bloques se liberan; los del prefijo pueden quedar en el prefix cache para el próximo request.

RAMA APARTE **Métricas**

PROMQL

```
vllm:e2e_request_latency_seconds
```

Cada fase quedó en un histograma de /metrics; Prometheus lo lee cada 15 s con el PodMonitor del C10.

HTTP el request HTTP que llega · PROMQL una consulta: va en Prometheus o en Grafana · FLAG una flag de vllm serve

EVENTO SSE un evento del stream (Server-Sent Events): lo manda vLLM, no lo escribes · JSON un campo del JSON de la respuesta

Un request de soporte del C10 dentro de una réplica de vLLM v0.30, con los números del README (estimaciones: vLLM no corrió en la referencia). El chat template, el tokenizer, el detokenizer y el SSE corren en el proceso del API server; la cola, el scheduler y el prefix cache, en el engine core; el prefill y el decode, en los dos workers de GPU. Si el cliente se va a mitad, vLLM aborta la secuencia y libera su KV.

EJEMPLO REAL · UN REQUEST DE SOPORTE DEL C10, DE PUNTA A PUNTA

TEXTO reference/c10/README.md (líneas 62–79, «Un request de punta a punta»)

```
t = 0        soporte manda POST /v1/chat/completions {model: "chat-default", stream: true, max_tokens: 512}
             con su API key al llm-gateway
+ < 1 ms     key → sha256 → tenant support-assistant (interactive, 5.4 M tokens/min, tope 1,024, sin vendor)
             validación: alias permitido, n = 1, max_tokens ≤ 1,024, prompt estimado ≤ 6,000
             cuota: cobra ≈ 2,000 (prompt estimado) + 512 del token bucket de esta réplica (5.4 M ÷ 4 por minuto)
+ ≤ 0.5 s   admisión: un lugar interactivo de los 288 de esta réplica, o cola (≤ 64, ≤ 0.5 s), o 503 + Retry-After
             → upstream vllm-70b (Service de los pods estables) con model = chat-70b-fp8-2026-09-28,
               priority = 0, stream_options.include_usage = true; conexión TCP nueva (sin keep-alive hacia vLLM)
+ ≈ 0.3–1 s vLLM: prefix cache del system prompt, prefill troceado del resto, primer token
             el gateway espera el primer evento con CONTENIDO (≤ 10 s): hasta acá una falla todavía es un 502/504
             limpio, incluido un stream que termina sin texto (así llega un abort antes del primer token, como en
             un drain: el rol, el usage y [DONE], sin chunk de abort)
             TTFT = desde que el request llegó al gateway hasta el primer evento con contenido
cada ≈ 40 ms un evento SSE por token, reenviado apenas llega (TPOT p95 ≤ 50 ms)
fin          [DONE]; el chunk de usage que pidió el gateway no se reenvía (el cliente no lo pidió)
             cierre: libera el lugar, concilia la cuota con usage (devuelve lo cobrado de más), métricas y una línea
             de log sin prompt ni respuesta
si el cliente se va: se cierra la conexión con vLLM (vLLM aborta y libera el KV), status 499 en las métricas
```

- Las primeras líneas pasan en el gateway (sección 11): la llave, la cuota en tokens y la admisión. vLLM recibe el request ya con `priority = 0`, el nombre de la revisión en `model` y `stream_options.include_usage = true`.
- «prefix cache del system prompt, prefill troceado del resto, primer token»: los eslabones del prefix cache y del prefill. El «≈ 0.3–1 s» es una estimación del caso: suma la cola de vLLM y el prefill de los ≈ 1,000 tokens que no estaban en caché.
- «TTFT = desde que el request llegó al gateway hasta el primer evento con contenido»: el C10 mide donde lo ve el producto. El primer evento de vLLM puede traer solo el rol, sin texto. `vllm bench serve`, en cambio, marca el TTFT en el primer evento con `choices`, tenga texto o no.
- «cada ≈ 40 ms un evento SSE por token»: el decode. Con el SLO de TPOT p95 ≤ 50 ms son ≈ 20 tokens por segundo por usuario, más rápido de lo que se lee.
- «si el cliente se va»: el gateway cierra la conexión con vLLM, vLLM aborta la secuencia y libera su KV; en las métricas del gateway queda un `499`.

CÓMO RECORDARLA

Entra texto, se vuelve números, espera su turno, se lee de una vez (prefill), se escribe de a un token (decode) y vuelve a ser texto, evento por evento. El TTFT es la cola más el prefill; el TPOT es un paso de decode; el KV cache une las dos fases. Si algo tarda, recorre la cadena con sus métricas: `num_requests_waiting` → `request_queue_time_seconds` → `request_prefill_time_seconds` → `inter_token_latency_seconds`.

TRAMPA

Mirar el TTFT del servidor y creer que es el que ve la persona. Entre vLLM y el usuario hay un gateway, un load balancer y a veces un proxy: si alguno junta la respuesta antes de reenviarla (buffering), el primer token llega pegado al último. Un TTFT de 0.5 s se vuelve uno de ≈ 10 s, y `vllm:time_to_first_token_seconds` sigue en verde. Por eso el C10 mide el TTFT en el gateway, hasta el primer evento con contenido. El lab de esta sección lo muestra con un proxy de juguete.

En el curso: M14 · Prefill y decode: TTFT, TPOT, throughput y goodput · C10 · Serving · C10 · Archivos de referencia · M7 · Servidores de LLMs: vLLM, SGLang, TGI, TensorRT-LLM · Hoja · Del dato a producción · el patrón LLM

1 Por qué servir un LLM es distinto y qué hace vLLM

¿Qué cambia cuando el modelo escribe su respuesta token por token, y qué piezas trae vLLM para que una GPU atienda muchas conversaciones a la vez?

SINTAXIS CON VARIABLES

`vllm serve` *modelo* **`--tensor-parallel-size`** *tp*

Un comando levanta el servidor compatible con OpenAI en el puerto 8000: carga los pesos, mide la memoria, arma el **KV cache** (la memoria de lo que el modelo ya leyó) y empieza a aceptar requests. Los flags están en la sección 4.

```
curl -sN localhost:8000/v1/chat/completions -H 'content-type: application/json' -d '{"model": "nombre", "messages": [{"role": "user", "content": "texto"}], "stream": true}'
```

Con `stream: true` se ve que la respuesta es un loop: un evento por token. Sin streaming, esperas al último token para ver el primero.

```
curl -s localhost:8000/metrics | grep -E '^vllm:num_requests_(running|waiting)'
```

El **continuous batching** (el batch se rearma en cada paso) a la vista: cuántas conversaciones comparten la GPU en este paso (`running`) y cuántas esperan lugar (`waiting`).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo	ruta local o repo de Hugging Face del modelo	`/models/chat-70b-fp8-2026-09-28 en el C10`
tp	cuántas GPUs reparten cada capa (tensor parallelism)	`2`
nombre	el nombre servido	`chat-70b-fp8-2026-09-28`
texto	el mensaje	`Resume esta política en tres puntos`
P, T_{prompt}	parámetros del modelo y tokens del prompt	`70.6 B y 2,000`
b_w	bytes por peso	`1 (FP8)`
S	secuencias en el batch	`1, 40 o 92`
t_{paso}	lo que tarda un paso de decode: leer los pesos y el KV de todo el batch	`≥ 14.9 ms con 40 secuencias (el lab)`
I_{decode}	intensidad aritmética: operaciones por cada byte que se trae de la memoria	`≈ 2 FLOPs/B con una secuencia, contra 591 que pide la H100 para estar ocupada`
N_i	largo de la respuesta i de un batch estático	`de 50 a 1,000 tokens en el ejemplo de M7`

FÓRMULAS

$$\text{FLOPs}_{\text{prefill}} \approx 2 \times P \times T_{\text{prompt}} = 2 \times 70.6\ \text{B} \times 2{,}000 \approx 2.8 \times 10^{14} \Rightarrow t \gtrsim \frac{2.8 \times 10^{14}}{2 \times 1{,}979\ \text{TFLOPS}} \approx 71\ \text{ms}$$

$$t_{\text{paso}} \gtrsim \frac{\text{pesos+KV del batch}}{N_{gpu} \times \text{ancho de banda}} = \frac{70.6\ \text{GB} + 0.74\ \text{GB}}{2 \times 3.35\ \text{TB/s}} \approx 10.6\ \text{ms} \qquad \text{(una secuencia)}$$

$$\text{throughput} \lesssim \frac{S}{t_{\text{paso}}} = \frac{40}{14.94\ \text{ms}} \approx 2{,}678\ \text{tokens/s} \qquad \text{(y por usuario, } \leq 67\text{)}$$

$$I_{\text{decode}} \approx \frac{2 \times P \times S}{P \times b_w} = \frac{2S}{b_w} \ll \frac{1{,}979\ \text{TFLOPS}}{3.35\ \text{TB/s}} \approx 591\ \text{FLOPs/B}$$

$$\text{útil}_{\text{estático}} = \frac{\sum_i N_i}{S \times \max_i N_i} = \frac{2{,}400}{8 \times 1{,}000} = 30\ \%$$

$$\text{reservando 8,192 por secuencia: } \left\lfloor \frac{13{,}046}{512} \right\rfloor = 25 \qquad \text{en bloques de 16: } \left\lfloor \frac{13{,}046}{141} \right\rfloor = 92$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Autoregresivo	El modelo escribe un token por pasada, y cada token nuevo depende de todos los anteriores: la respuesta es un loop, no una llamada.	`250 tokens de salida = 1 prefill + 249 pasos de decode`
Prefill	Leer todo el prompt de una vez: cada peso se usa para los 2,000 tokens, así que manda el cómputo (compute-bound).	`piso ≈ 71 ms en 2 × H100 con FP8; ≈ 143 ms en BF16`
Decode	Escribir de a un token: cada paso relee todos los pesos para sacar un token por secuencia, así que manda el ancho de banda de la memoria (memory-bound).	`piso ≈ 10.6 ms por paso con TP=2; el C10 estima ≈ 40 ms`
KV cache	Lo que el modelo ya leyó de cada conversación (las keys y values de cada capa), guardado en la GPU para no recalcularlo en cada token nuevo.	`320 KiB por token: ≈ 0.74 GB por conversación de 2,250 tokens`
Static batching	Un batch fijo: no termina hasta su respuesta más larga, y los lugares de las que ya terminaron quedan vacíos.	`30 % de lugares útiles en el ejemplo de M7`
Continuous batching	El batch se rearma en cada paso: una secuencia que termina deja su lugar y otra entra en el mismo paso. La idea viene de Orca (OSDI 2022).	`hasta 128 secuencias por réplica en el C10 (--max-num-seqs)`

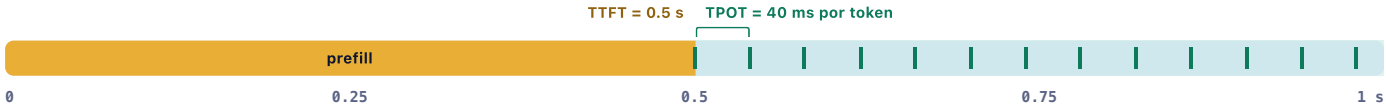
PIEZA	QUÉ HACE	EJEMPLO
PagedAttention	Guarda el KV cache en bloques chicos, como las páginas de la memoria virtual, en vez de reservar por secuencia el largo máximo. Su paper (SOSP 2023) reporta 2–4× más throughput a igual latencia.	`bloques de 16 tokens: 92 secuencias en vez de 25`
Chunked prefill	Parte un prompt largo en trozos que comparten cada paso con los decodes de los demás: un prompt de 8k no congela al resto.	`activado por defecto en V1`
Prefix caching	Reusa los bloques de KV de un comienzo de prompt que ya calculó.	`el system prompt de ≈ 1,000 tokens de cada producto`
Servidor compatible con OpenAI	vllm serve expone la API de OpenAI: el cliente oficial funciona cambiando la base_url.	`base_url del gateway: http://vllm-70b.c10.svc.cluster.local:8000/v1`
Motor V1	La arquitectura actual de vLLM, la única desde v0.11: un proceso para HTTP y tokens, un engine core con el scheduler y un worker por GPU.	`TP=2: 4 procesos`
SGLang	La alternativa más cercana. Su RadixAttention guarda los prefijos en un árbol y los reusa: rinde cuando muchos prompts comparten comienzo.	`candidata a medir contra vLLM con la mezcla del C10`
TensorRT-LLM	El de NVIDIA: kernels propios, FP8 y FP4, y trtllm-serve compatible con OpenAI. Más atado a la GPU, el driver y el contenedor.	`el C10 no lo elige para empezar`
TGI	El de Hugging Face. Está en modo mantenimiento y su repositorio está archivado (el último cambio es del 21 de marzo de 2026); su README recomienda vLLM o SGLang.	`no para un sistema nuevo`
llama.cpp · MLX	Para correr modelos en local, en CPU o en Apple Silicon; el README de TGI los recomienda para eso.	`tu laptop, no la flota del C10`
NVIDIA Dynamo · llm-d	No son motores: orquestan varios (prefill y decode en GPUs distintas, ruteo según el KV cache de cada réplica).	`una palanca para después en el C10`

EL REQUEST ENTERO, A ESCALA (DE 0 A 15 S)



- **0.5 s** prefill: 2,000 tokens a la vez, más la cola · cómputo
- **9.96 s** decode: 249 pasos de ≈ 40 ms, un token cada uno · memoria
- TTFT ≈ 0.5 s: la primera palabra
- E2E ≈ 10.5 s: la última
- **1 s** el SLO del TTFT (p95) · **13.45 s** el peor caso dentro de los dos SLOs: 1 s + 249 × 50 ms

EL PRIMER SEGUNDO: EL PREFILL Y LOS PRIMEROS 13 TOKENS, CADA 40 MS (A ESCALA, DE 0 A 1 S)



piso teórico del prefill: ≈ 71 ms (el real: 500 ms, con la cola)

piso teórico de un paso de decode: ≈ 10.6 ms (el real: 40 ms)

prefill: lee el prompt (cómputo) · decode: un token por paso (memoria) · TTFT y E2E de este request · el SLO

Con los dos SLOs en su límite p95, el peor caso dentro del SLO es 1.0 + 249 × 0.050 ≈ 13.5 s. Con streaming no importa: la persona empieza a leer en el TTFT.

Un request de soporte del C10, a escala: 2,000 tokens de prompt y 250 de salida, con TTFT ≈ 0.5 s y ≈ 40 ms por token (estimaciones del caso). El prefill es un bloque de cómputo; el decode, 249 pasos chicos de memoria. Los pisos teóricos en 2 × H100: ≈ 71 ms de prefill y ≈ 10.6 ms por paso.

EJEMPLO REAL · UN REQUEST DE LLM DURA SEGUNDOS Y OCUPA SU LUGAR TODO ESE TIEMPO

PYTHON reference/c10/capacity/fleet_calc.py (líneas 136–143)

```python
def request_seconds(ttft_s: float, output_tokens: float, tpot_s: float) -> float:
    """Duración de un request de streaming ≈ TTFT + tokens de salida × TPOT."""
    return ttft_s + output_tokens * tpot_s


def littles_law_concurrency(rps: float, seconds_per_request: float) -> float:
    """L = λ × W (#m03.little): secuencias vivas a la vez = llegadas por segundo × segundos que dura cada una."""
    return rps * seconds_per_request
```

- Un request de LLM no es un forward: es un TTFT y después un paso por token. Con los números del caso, 0.5 + 250 × 0.040 = **10.5 s**. El ResNet-50 del C01 usa ≈ 5 ms de GPU por imagen.
- La función cuenta un paso de más: con la definición de `vllm bench serve` son 249 pasos después del primero (10.46 s). La diferencia no cambia ninguna decisión.
- `littles_law_concurrency` es la ley de Little: 60 rps × 10.5 s = **630 secuencias vivas** en la flota, ≈ 39.4 por réplica. Cada una ocupa su lugar en el batch y su KV durante esos 10 segundos.
- Por eso un servidor de LLMs vive del continuous batching: con secuencias que entran y salen en cada paso, las ≈ 40 de una réplica comparten cada lectura de los pesos. Y por eso la memoria del KV, no el cómputo, decide cuántas caben (sección 3).

CÓMO RECORDARLA

Prefill es leer la pregunta entera de una vez; decode es contestar palabra por palabra. Leer es cómputo (TTFT); contestar es memoria (TPOT). Y el continuous batching es **un bus que deja subir y bajar pasajeros en cada parada**, en vez de esperar a que todos lleguen al final del recorrido.

TRAMPA

Servir un LLM como una CNN: FastAPI con un `model.generate()` por batch. El batch es estático: no termina hasta la respuesta más larga, los lugares vacíos esperan (30 % útil en el ejemplo de M7), cada secuencia reserva su KV entero y no hay streaming si no lo escribes. Y medir la carga con `nvidia-smi`: un decode memory-bound marca ≈ 100 % de utilización con los tensor cores (las unidades que multiplican matrices) casi ociosos. La capacidad se lee en la cola y en el KV: `vllm:num_requests_waiting` y `vllm:kv_cache_usage_perc`.

En el curso: M7 · Servidores de LLMs: vLLM, SGLang, TGI, TensorRT-LLM · M14 · Qué se pregunta de LLMs y el mapa del sistema · M4 · Compute-bound vs memory-bound: la intuición del roofline · M14 · Batching, caching, speculative decoding y paralelismo · Hoja · De PyTorch a Triton · qué es servir un modelo

2 Tokens y contexto: tokenizer, chat template, max_model_len, costo por token, español contra inglés

¿Qué cuenta vLLM como token, qué entra en la ventana de contexto, qué pasa cuando un request no cabe y por qué el mismo texto cuesta más en español que en inglés?

SINTAXIS CON VARIABLES

`curl -s localhost:8000/tokenize -H 'content-type: application/json' -d '{"model": "` *modelo* **`", "messages": [{"role": "system", "content": "`** *system* **`"}, {"role": "user", "content": "`** *texto* **`"}], "return_token_strs": true}'**

Cuenta como el modelo servido: su **tokenizer** (el que parte el texto en tokens) y su **chat template** (la plantilla que une los mensajes con las marcas del modelo). `count` incluye las marcas de la plantilla y el encabezado del asistente (`add_generation_prompt` , `true` por defecto); `token_strs` muestra cada pedazo.

`curl -s localhost:8000/tokenize -H 'content-type: application/json' -d '{"model": "` *modelo* **`", "prompt": "`** *texto* **`"}'**

Solo el texto, sin plantilla: la diferencia entre los dos `count` es lo que agrega el chat template. Con `prompt` , `add_special_tokens` es `true` por defecto: suma el token de inicio.

`"max_completion_tokens":` *n*

El tope de salida de un request; `max_tokens` sigue funcionando, pero está deprecado a favor de este. Si no lo mandas, en chat vLLM puede generar hasta llenar la ventana: `max_model_len` – prompt; en `/v1/completions` , el default es 16.

`vllm serve` *modelo* **`--max-model-len`** *ctx*

La **ventana de contexto**: cuántos tokens caben entre prompt y respuesta. Acepta `8192` , `8K` (= 8,192) u `8k` (= 8,000), y `auto` : el máximo que entra en la memoria.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo	el nombre servido	`chat-70b-fp8-2026-09-28`
system , *texto*	el system prompt y el mensaje del usuario	`Eres el asistente de soporte del marketplace. · ¿Dónde está mi pedido?`
n	tope de tokens de salida	`1,024 en soporte, 600 en listings y 400 en catálogo`
ctx	la ventana de contexto	`8192`
T_{prompt}, T_{in}, T_{out}	tokens del prompt (los de entrada) y de la salida	`2,000 y 250 en promedio`
p_{in}, p_{out}	precio por millón de tokens de entrada y de salida	`US$2.50 y US$10 (lista ilustrativa del C10)`
f	fertilidad: tokens por palabra de un tokenizer en un idioma	`Llama 3.1: 1.23 en inglés y 1.61 en español`
$\hat{T}_{\text{gateway}}$	lo que estima el gateway del C10 antes de tokenizar, a propósito pesimista	`66 tokens para un chat de 174 caracteres (el lab)`

FÓRMULAS

$$T_{\text{prompt}} + \text{max_tokens} \le \text{max_model_len}: \quad 2{,}000 + 1{,}024 \le 8{,}192 \qquad \text{(v0.30 responde 400 si no)}$$

$$\text{max_tokens}_{\text{efectivo}} = \min(\text{max_model_len} - T_{\text{prompt}},\ \text{max_tokens}) \qquad \text{(v0.32: } 7{,}500 + 1{,}024 \to 692\text{)}$$

$$\text{costo} = \frac{T_{in} \times p_{in} + T_{out} \times p_{out}}{10^6} = \frac{2{,}000 \times 2.50 + 250 \times 10}{10^6} = \text{US\$0.0075}$$

$$T \approx \text{palabras} \times f: \quad 200 \times 1.61 \approx 322(\text{español}) \quad \text{contra} \quad 200 \times 1.23 \approx 246(\text{inglés})$$

$$\hat{T}_{\text{gateway}} = \left\lceil \frac{\text{caracteres}}{3} \right\rceil + 4 \times \text{mensajes}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`Token`	Un pedazo de texto del vocabulario del modelo (una palabra corta, parte de una palabra o un signo), con su número.	`Llama 3: 128,256 entradas en el vocabulario`
`Tokenizer`	Parte el texto en tokens y los convierte en números, y al revés. Cada familia de modelos tiene el suyo: el mismo texto da cantidades distintas en cada una.	`el de Llama 3.1: ≈ 1.23 tokens por palabra en inglés y ≈ 1.61 en español`
`Tokens especiales`	Marcas que no son texto: inicio, encabezado de rol y fin de turno. Cada una es un solo token.	`<|begin_of_text|> · <|start_header_id|> · <|end_header_id|> · <|eot_id|>`
`Chat template`	La plantilla Jinja (un lenguaje de plantillas) del tokenizer_config.json que convierte la lista de messages en un solo texto con esas marcas. vLLM usa la del modelo y --chat-template la reemplaza; sin plantilla, los requests de chat fallan.	`la de Llama 3.3 agrega siempre un bloque system con «Cutting Knowledge Date: December 2023» y «Today Date: 26 Jul 2024»`
`add_generation_prompt`	Agrega al final el encabezado del asistente, para que el modelo empiece a contestar ahí. /tokenize lo pone por defecto con messages.	`<|start_header_id|>assistant<|end_header_id|> y un salto doble`
`max_model_len`	La ventana: cuántos tokens caben entre prompt y respuesta. Si no la fijas, vale la del config del modelo.	`8,192 en el C10; 131,072 en el config de Llama 3.3`
`max_tokens · max_completion_tokens`	El tope de tokens de salida de un request. Sin él, en chat, vLLM puede generar hasta llenar la ventana; en /v1/completions el default es 16.	`el gateway lo fija en 1,024 (soporte), 600 (listings) y 400 (catálogo)`

PIEZA	QUÉ HACE	EJEMPLO
Prompt + salida > ventana	En v0.30, un 400 que empieza con «This model's maximum context length is». Desde v0.32, si el prompt cabe por sí solo, entra y la salida se recorta a lo que queda.	`7,500 + 1,024: un 400 en v0.30; como mucho 692 tokens en v0.32`
usage	Lo que la respuesta dice que gastó: prompt_tokens, completion_tokens y total_tokens. En streaming llega al final si lo pides con stream_options.include_usage.	`el gateway concilia la cuota con esto`
finish_reason	Por qué terminó: stop (el modelo cerró su turno o encontró un stop), length (llegó a max_tokens o a la ventana), tool_calls (pidió una herramienta) o abort (vLLM canceló la secuencia).	`length en soporte = una respuesta cortada: mide cuántas hay`
Prompt contra salida	Los tokens de entrada se leen juntos en el prefill; cada token de salida es un paso de decode y agranda el KV.	`≈ 8 de entrada por cada 1 de salida en el interactivo del C10`
Costo por token	Las APIs cobran aparte la entrada y la salida, y la salida suele costar varias veces más.	`US$2.50 y US$10 por millón: un request de soporte cuesta US$0.0075, un tercio por la salida`
Español contra inglés	El mismo tokenizer parte el español en más pedazos por palabra: más tokens, más costo, más KV y más pasos de decode.	`200 palabras: ≈ 246 tokens en inglés y ≈ 322 en español (Llama 3.1)`
Estimar sin tokenizer	Dividir los caracteres: rápido y aproximado. Sirve para cobrar una cuota por adelantado, no para facturar.	`el gateway del C10: caracteres ÷ 3 + 4 por mensaje, y concilia con usage`
/tokenize	El endpoint de vLLM que cuenta con el tokenizer y el chat template del modelo servido.	`devuelve count, max_model_len y tokens`

TOKENS POR PALABRA, A ESCALA DE 0 A 2 · TEXTO FORMAL: NOTICIAS DE XL-SUM EN CADA IDIOMA



200 palabras de respuesta con el tokenizer de Llama 3.1: **246** tokens en inglés contra **322** en español (+31 %)



Fuente: Nayeem et al., «Beyond Fertility: Analyzing STRR as a Metric for Multilingual Tokenization Evaluation», 2025, tabla 1 (arxiv.org/abs/2510.09947)

EL CHAT TEMPLATE DE LLAMA 3.3 70B INSTRUCT: LO QUE DE VERDAD LEE EL MODELO

`<|begin_of_text|>` `<|start_header_id|>` **system** `<|end_header_id|>` ↵↵

Cutting Knowledge Date: December 2023↵*Today Date: 26 Jul 2024*↵↵ Eres el asistente de soporte del marketplace. `<|eot_id|>` `<|start_header_id|>`

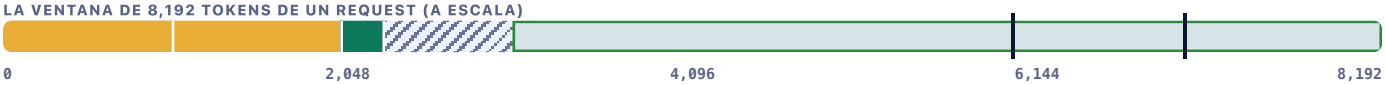
user `<|end_header_id|>` ↵↵ ¿Dónde está mi pedido? `<|eot_id|>` `<|start_header_id|>` **assistant** `<|end_header_id|>` ↵↵

`<|…|>` marca especial: 1 token (aquí, 9) **rol** el rol del turno *texto* lo agrega la plantilla texto tus mensajes ↵ salto de línea

Cada marca especial es un solo token: 9 en este prompt. El texto, tuyo o de la plantilla, lo cuenta el tokenizer: POST /tokenize da el total. Si no pasas date_string en chat_template_kwargs, la plantilla dice que hoy es 26 Jul 2024.

Fuente: el chat_template del tokenizer_config.json (copia pública: huggingface.co/unsloth/Llama-3.3-70B-Instruct; el repo de Meta pide aceptar la licencia)

LA VENTANA DE 8,192 TOKENS DE UN REQUEST (A ESCALA)



- **1,000** system prompt del producto (prefijo compartido)
- **250** respuesta promedio
- **5,168** libre hasta 8,192
- **7,024** 6,000 + 1,024: lo máximo que pide un request de soporte
- **1,000** historial y mensaje del usuario
- **774** resto del tope de salida de soporte (1,024)
- **6,000** max_prompt_tokens de soporte en el gateway

inglés · español · prompt (lo lee el prefill) · salida (la escribe el decode) · reservado para la salida · libre

Arriba, cuántos tokens por palabra da cada tokenizer en inglés y en español (medido por Nayeem et al., 2025, en noticias): con el de Llama 3.1, la familia del modelo del C10, el español usa ≈ 31 % más. Al centro, el chat template de Llama 3.3 aplicado a dos mensajes cortos: 9 tokens especiales antes de que el modelo escriba nada. Abajo, la ventana de 8,192 tokens de un request de soporte del C10.

EJEMPLO REAL · CÓMO ESTIMA LOS TOKENS EL GATEWAY DEL C10 ANTES DE TOKENIZAR

PYTHON reference/c10/gateway/quota.py (líneas 32–48)

```python
# Caracteres por token para ESTIMAR sin tokenizer. ≈ 4 es la regla para inglés con tokenizers BPE; en
# español suele haber más tokens por carácter. 3.0 sobrestima a propósito (el error se devuelve al conciliar).
# Mídelo con tu tokenizer sobre una muestra de tus prompts: es una estimación, no un dato del modelo.
CHARS_PER_TOKEN = 3.0
TOKENS_PER_MESSAGE = 4  # overhead del chat template por mensaje (rol, separadores): estimación


def estimate_prompt_tokens(messages: list[dict]) -> int:
    """Tokens de entrada estimados a partir de los mensajes (texto y partes de texto; ignora imágenes)."""
    chars = 0
    for m in messages:
        content = m.get("content")
        if isinstance(content, str):
            chars += len(content)
        elif isinstance(content, list):  # formato multi-parte: [{"type": "text", "text": "..."}, ...]
            chars += sum(len(p.get("text", "")) for p in content if isinstance(p, dict))
    return math.ceil(chars / CHARS_PER_TOKEN) + TOKENS_PER_MESSAGE * len(messages)
```

- `CHARS_PER_TOKEN = 3.0`: la regla de ≈ 4 caracteres por token es para inglés, y en español salen más tokens. 3.0 sobrestima a propósito: el gateway cobra de más al entrar y devuelve la diferencia al terminar, con el `usage` real.
- `TOKENS_PER_MESSAGE = 4` es lo que agrega la plantilla por mensaje, estimado. En Llama 3.3, cada mensaje lleva tres tokens especiales (`<|start_header_id|>`, `<|end_header_id|>` y `<|eot_id|>`) más el rol y un salto de línea doble, y la plantilla agrega un bloque system con dos líneas de fecha aunque no mandes ninguno. El número exacto lo da `/tokenize`.
- Por qué estimar: la cuota se cobra al entrar (prompt estimado + `max_tokens`) y el costo real se conoce al final. Con ≈ 2,000 de prompt y `max_tokens` 1,024, el gateway cobra 3,024 y, si se generaron 250, devuelve 774 (el docstring de `quota.py`).
- El gateway también pone los topes de la ventana: `max_prompt_tokens` 6,000 y `max_tokens_cap` 1,024 en soporte (`tenants.yaml`), porque 6,000 + 1,024 < 8,192. Así, mientras la estimación no se quede corta, ningún request de soporte llega a vLLM pidiendo más de lo que cabe; si se queda corta, vLLM responde el 400 y el gateway lo reenvía.

CÓMO RECORDARLA

Cuenta lo que cuenta el tokenizer del modelo, después del chat template. Prompt + salida ≤ `max_model_len`. La salida es la parte cara (4× en la lista del C10) y la lenta (un paso por token). Y en español, con el tokenizer de Llama 3.1, ≈ 1.3× más tokens por palabra que en inglés.

TRAMPA

Dejar `max_tokens` sin fijar. En chat, vLLM toma entonces todo lo que queda de la ventana (8,192 – prompt): una respuesta que no para ocupa su lugar y su KV hasta llenarla, y la cuota no sabe cuánto cobrar. El gateway del C10 lo fija en el tope del tenant. Y la trampa del upgrade: en v0.30, el pin del C10, un prompt de chat de 7,500 tokens con `max_tokens` 1,024 recibe un 400 («This model's maximum context length is 8192 tokens. However, you requested 1024 output tokens and your prompt contains at least 7169 input tokens, for a total of at least 8193 tokens. …»). Dice «at least 7169» y no 7,500 porque vLLM deja de tokenizar un token después del límite. Desde v0.32 el mismo request entra, y la respuesta se corta en 692 tokens como mucho, con `finish_reason: "length"` si llega. Si un cliente dependía del 400, el upgrade lo cambia sin avisar: valida en el gateway.

En el curso: M14 · Costo por token y cómo bajarlo · C10 · Estimaciones back-of-the-envelope · Números · Bytes, dtypes y tamaños típicos · C10 · Serving

3 Memoria: pesos + KV cache, la fórmula con GQA, gpu_memory_utilization, bloques, max_num_seqs y preemption

¿En qué se va la memoria de la GPU, cuántas conversaciones caben a la vez en una réplica y qué hace vLLM cuando se le acaban los bloques de KV?

SINTAXIS CON VARIABLES

`vllm serve` *modelo* `--tensor-parallel-size` *tp* `--gpu-memory-utilization` *u* `--max-model-len` *ctx* `--max-num-seqs` *s* `--kv-cache-dtype` *dtype*

Los cinco flags de la cuenta de memoria. *u* es la fracción de la memoria TOTAL de cada GPU que vLLM se permite usar (0.92 si no la escribes); *dtype* es `auto` (el del modelo) o `fp8`.

`Available KV cache memory:` *gib* `GiB`

Lo que imprime después de cargar los pesos y de perfilar un paso de prueba: lo que quedó para KV, por GPU y en GiB. Ya descontó pesos, activaciones y **CUDA graphs** (los pasos de la GPU grabados para repetirlos rápido).

`GPU KV cache size:` *tokens* `tokens, Maximum concurrency for` *ctx* `tokens per request:` *x* `x`

La capacidad real de la réplica: los tokens de KV (los mismos en las dos GPUs con TP) y cuántas secuencias del largo máximo caben a la vez. Manda sobre cualquier cuenta.

`curl -s localhost:8000/metrics | grep -E '^vllm:(kv_cache_usage_perc|num_preemptions_total|num_requests_waiting)'`

Las tres señales de que el KV se llenó: uso cerca de 1, **preemptions** (secuencias que vLLM saca del batch para hacer lugar) que suben y requests esperando. Con cola, lo primero que se rompe es el TTFT.

`--kv-cache-memory-bytes` *bytes*

Fija el KV por GPU en bytes y se salta la medición de memoria: ignora `--gpu-memory-utilization`. Acepta los sufijos de `--max-model-len`: la minúscula es decimal y la mayúscula binaria (`30g` = 30 × 10⁹ bytes, `30G` = 30 × 2³⁰).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo	ruta o repo del modelo	`/models/chat-70b-fp8-2026-09-28`
tp	GPUs que reparten cada capa	`2`
u	fracción de la memoria de cada GPU para vLLM	`0.90`
ctx	contexto máximo por secuencia	`8192`
s	techo de secuencias por paso	`128`
dtype	tipo de número del KV cache	`auto (BF16) o fp8`
gib, *tokens*, *x*	lo que imprime vLLM: GiB libres por GPU, tokens de KV y concurrencia para {ctx}	`con la cuenta del C10: ≈ 31.9 GiB, 208,736 y 25.48`
bytes	tamaño fijo del KV por GPU	`no lo usa el C10`
M_{GPU}, u	memoria total de una GPU (la que reporta el dispositivo) y la fracción	`80 GB nominales, 81,559 MiB reportados; 0.90`
L, H, H_{kv}, g	capas, heads de atención, KV heads y cuántos heads comparten cada KV head	`80, 64, 8 y 8`
d_{head}, b_{kv}	tamaño de cada head y bytes por número del KV	`128 y 2`
N_{bloques}, T_{seq}	bloques de 16 tokens que caben y tokens por secuencia	`13,046 y 2,250`
N_{vivas}, T_{pref}, N_{pref}	secuencias vivas, tokens del prefijo compartido y prefijos distintos	`40, 1,000 y 3 (el mejor caso del C10)`

FÓRMULAS

$$M_{\text{vLLM}} = \lceil M_{\text{GPU}} \times u \rceil \text{ por GPU} \qquad \text{KV}_{\text{libre}} = M_{\text{vLLM}} - \text{pesos} - \text{activaciones} - \text{CUDA graphs}$$

$$\text{KV}_{\text{token}} = 2 \times L \times H_{kv} \times d_{head} \times b_{kv}, \qquad H_{kv} = \frac{H}{g} = \frac{64}{8} = 8 \Rightarrow 320\ \text{KiB en vez de 2.5 MiB}$$

$$N_{\text{bloques}} = \left\lfloor \frac{\text{KV}_{\text{libre}}/\text{TP}}{16 \times \text{KV}_{\text{token}}/\text{TP}} \right\rfloor = \left\lfloor \frac{34.2\ \text{GB}}{16 \times 163{,}840\ \text{B}} \right\rfloor = 13{,}046 \Rightarrow 208{,}736\ \text{tokens}$$

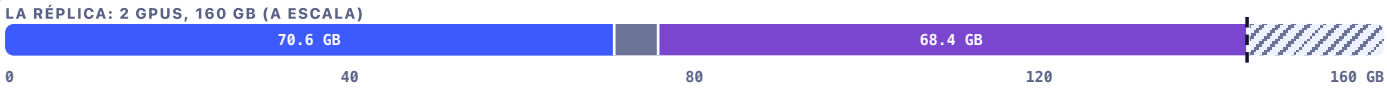
$$N_{seq} = \left\lfloor \frac{N_{\text{bloques}}}{\lceil T_{seq}/16 \rceil} \right\rfloor = \left\lfloor \frac{13{,}046}{141} \right\rfloor = 92 \qquad \text{concurrencia} = \frac{13{,}046}{\lceil 8{,}192/16 \rceil} = 25.48$$

$$\text{KV}_{\text{vivo}} = N_{vivas} \times (T_{seq} - T_{pref}) + N_{pref} \times T_{pref} = 40 \times 1{,}250 + 3 \times 1{,}000 = 53{,}000\ \text{tokens (en vez de 90,000)}$$

LAS PIEZAS

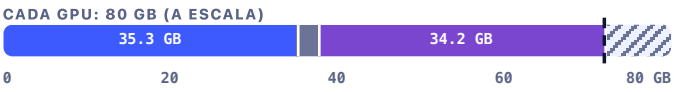
PIEZA	QUÉ HACE	EJEMPLO
Pesos	Los números del modelo: parámetros × bytes. Con TP=2 se reparten entre las GPUs, no se copian.	`70.6 GB en FP8: 35.3 por GPU`
Activaciones y CUDA graphs	Lo que ocupa calcular un paso (resultados intermedios) y los CUDA graphs (los pasos de la GPU grabados una vez, para repetirlos sin el costo de lanzarlos). vLLM los mide al arrancar con un paso de prueba.	`≈ 5 GB en la réplica (estimación del C10)`
KV cache	Todo lo que queda dentro del presupuesto: se convierte en bloques para K y V.	`68.4 GB por réplica`
Margen (1 − u)	Lo que vLLM no toca: picos, fragmentación, otros procesos.	`10 % con 0.90: 16 GB en la réplica`
GQA	Ocho heads de atención comparten cada par de K y V: el KV es 8× más chico que con un par por head.	`320 KiB por token en vez de 2.5 MiB`

PIEZA	QUÉ HACE	EJEMPLO
`--gpu-memory-utilization`	La fracción de la memoria TOTAL de cada GPU (la que reporta el dispositivo) que vLLM se permite usar. Si al arrancar hay menos libre, falla con «Free memory on device … is less than desired GPU memory utilization».	`0.90 explícito en el C10`
`--kv-cache-memory-bytes`	Fija el KV por GPU en bytes y se salta la medición de memoria: ignora gpu_memory_utilization.	`no lo usa el C10`
`Bloque (--block-size 16)`	La unidad en que se reparte el KV: 16 tokens de una secuencia. Una secuencia pide bloques a medida que crece.	`2,250 tokens = 141 bloques`
`--max-model-len`	El largo máximo de una secuencia. vLLM se niega a arrancar si no cabe ni una.	`8,192: una sola secuencia de 8k ocupa 2.7 GB`
`--max-num-seqs`	Cuántas secuencias corren en el mismo paso, como máximo. No reserva KV cache: es un techo (sí dimensiona algunos buffers y los CUDA graphs).	`128 en el C10; 1,024 por defecto en la H100`
`Preemption`	Si a una secuencia que corre le falta un bloque y no hay, vLLM saca a otra del batch: libera sus bloques y la devuelve a la cola (al frente, con fcfs). Al volver, recalcula su KV desde cero, salvo lo que siga en el prefix cache.	`con priority sale primero la de menor prioridad: el batch nocturno`
`vllm:num_preemptions_total`	El counter de preemptions. También aparece «Preemptions: N» en la línea periódica del log cuando hubo; el scheduler de V1 no escribe un warning al desalojar.	`LLMPreemptionsRising: > 0.05 por request terminado, 15 min, de día`
`vllm:kv_cache_usage_perc`	La fracción del KV ocupada (1 = 100 %).	`KVCacheSaturation: > 95 % y requests esperando`
`--kv-cache-dtype fp8`	Guarda K y V en 1 byte: la mitad de KV por token y el doble de secuencias.	`185 en vez de 92; el C10 lo deja en auto hasta pasar el golden set`
`--watermark`	La fracción de bloques que el scheduler deja libre al admitir requests, para no tener que sacar a nadie después. Por defecto 0.0: apagado.	`existe en v0.30 y en v0.32; el C10 no lo usa`
`Prefix caching y memoria`	Las secuencias con el mismo comienzo comparten esos bloques: no se duplican.	`40 × 1,250 + 3 × 1,000 = 53,000 tokens en vez de 90,000`

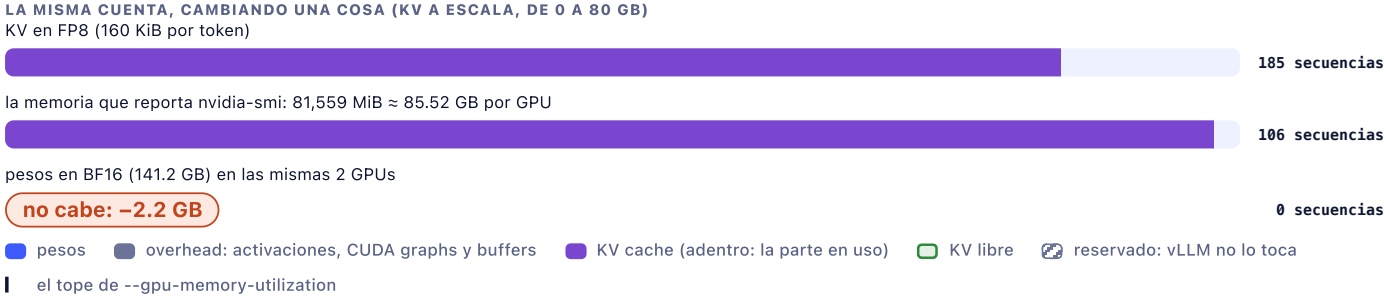


- **70.6 GB** pesos FP8
- **68.4 GB** KV cache
- **144 GB** el tope de vLLM: 0.9 × 160 GB
- **5 GB** activaciones, CUDA graphs y buffers (estimación)
- **16 GB** margen que vLLM no toca (10 %)





13,046 bloques de 16 tokens = 208,736 tokens de KV, los mismos en las 2 GPUs



Los números son los de `capacity/vram_calc.py` y sus tests. Con TP=2, cada GPU guarda la mitad de los KV heads de todas las capas: los 13,046 bloques son los mismos tokens en las dos GPUs.

Una réplica del C10 (2 × H100 de 80 GB, TP=2, --gpu-memory-utilization 0.90), a escala: 70.6 GB de pesos FP8, ≈ 5 GB de overhead (estimación), 68.4 GB de KV y 16 GB que vLLM no toca. Sobre el KV, lo que ocupan 40 conversaciones (la ley de Little en el pico), el máximo de 92 de 2,250 tokens y 25 secuencias de 8,192. Abajo, tres variantes de la misma cuenta.

EJEMPLO REAL · LUGAR PARA KV Y SECUENCIAS POR RÉPLICA, EN VRAM_CALC.PY

PYTHON reference/c10/capacity/vram_calc.py (líneas 110–152)

```python
def kv_room_per_replica(
    m: ModelConfig,
    gpu: GPU,
    tensor_parallel: int,
    gpu_memory_utilization: float,
    overhead_bytes: float,
) -> float:
    """Bytes que quedan para KV cache en UNA réplica (tensor_parallel GPUs que sirven juntas un modelo).

    gpu_memory_utilization: fracción de la VRAM que vLLM se permite usar (pesos + activaciones + KV). El
      default es 0.92 desde vLLM v0.20 (0.9 hasta v0.19): el caso fija 0.90 explícito en el manifest para
      que la capacidad no cambie con un upgrade.
    overhead_bytes: activaciones del forward, workspace, CUDA graphs y buffers de NCCL, sumados en la
      réplica. Es una estimación (≈ 5 GB en el caso); vLLM la mide al arrancar con un forward de prueba.
    """
    if tensor_parallel < 1:
        raise ValueError("tensor_parallel debe ser >= 1")
    if not 0 < gpu_memory_utilization <= 1:
        raise ValueError("gpu_memory_utilization debe estar en (0, 1]")
    if m.kv_heads % tensor_parallel != 0:
        # Con TP, cada GPU guarda los K/V de SUS KV heads. Si no se dividen exacto, el runtime replica heads
        # y la cuenta de abajo (KV total repartido sin duplicar) deja de valer: mejor fallar que mentir.
        raise ValueError(f"{m.kv_heads} KV heads no se dividen entre TP={tensor_parallel}")
    budget = tensor_parallel * gpu.vram_bytes * gpu_memory_utilization  # lo que vLLM puede usar en total
    room = budget - weight_bytes(m) - overhead_bytes  # los pesos se reparten entre las GPUs (TP), no se copian
    if room <= 0:
        raise ValueError(
            f"{m.name} no cabe: presupuesto {budget / GB:.1f} GB, pesos {weight_bytes(m) / GB:.1f} GB, "
            f"overhead {overhead_bytes / GB:.1f} GB"
        )
    return room


def max_concurrent_sequences(room_bytes: float, m: ModelConfig, tokens_per_sequence: int) -> int:
    """Cuántas secuencias de ese largo caben a la vez, todas en su largo máximo (peor caso, entero)."""
    return math.floor(room_bytes / kv_bytes_per_sequence(m, tokens_per_sequence))


def max_kv_tokens(room_bytes: float, m: ModelConfig) -> int:
    """Tokens totales de KV que caben. Al arrancar vLLM imprime un poco menos como "GPU KV cache size": cuenta
    bloques ENTEROS de 16 tokens por GPU (con TP=2 cada GPU guarda la mitad de los KV heads de cada token). En el
    caso: 208,740 aquí; 13,046 bloques × 16 = 208,736 en el log de vLLM."""
    return math.floor(room_bytes / kv_bytes_per_token(m))
```

- `budget = tensor_parallel × vram × utilización`: la memoria que vLLM se permite usar en la réplica. Los pesos se restan una sola vez porque TP los reparte entre las GPUs; con 2 × 80 GB y 0.90, 144 − 70.6 − 5 = **68.4 GB**.
- Si `kv_heads % tensor_parallel` no da 0, la cuenta falla a propósito: con TP cada GPU guarda los K y V de sus propios KV heads, y si no se dividen exacto el runtime replica heads y el KV ya no se reparte sin duplicar. Con 8 KV heads, TP puede ser 1, 2, 4 u 8.
- «no cabe»: con pesos BF16 en dos H100, 144 − 141.2 − 5 = −2.2 GB. vLLM fallaría al arrancar con un error de memoria; la cuenta lo dice antes de pedir las GPUs.
- `max_concurrent_sequences` usa el **peor caso**: todas las secuencias en su largo final. `max_kv_tokens` da 208,740 tokens; vLLM cuenta bloques enteros de 16 por GPU y diría 208,736.

CÓMO RECORDARLA

Pesos fijos, KV variable: lo que queda después de los pesos decide cuántas conversaciones atiendes. 144 GB usables − 70.6 − ≈ 5 = 68.4 GB = 13,046 bloques de 16 tokens = 92 conversaciones de 2,250. Cuando a una secuencia que corre le falta un bloque, vLLM saca a otra (preemption) y la recalcula después: trabajo tirado.

TRAMPA

Subir `--max-num-seqs` creyendo que compra capacidad. No reserva KV cache: es un techo. Si dejas entrar más secuencias de las que el KV aguanta en su largo final (128 × 141 bloques = 18,048 > 13,046), vLLM admite mientras haya bloques para los prompts (104 en el lab) y, cuando crecen, empieza a sacar y recalcular: el lab cuenta 130 preemptions y 273,100 tokens recalculados para el mismo throughput que con 92. El scheduler de V1 no escribe un warning al desalojar: lo ves en `vllm:num_preemptions_total` y en `Preemptions: N` de la línea periódica. La documentación todavía muestra el warning de V0 («Sequence group 0 is preempted by PreemptionMode.RECOMPUTE mode…»).

En el curso: M14 · KV cache: la memoria que decide la concurrencia · M4 · Cálculo de VRAM paso a paso · C10 · Estimaciones back-of-the-envelope · M4 · Bytes por dtype y cuánto pesa un modelo · Hoja · Formulario de estimaciones · memoria de GPU

4 Arrancar vLLM: vllm serve y sus flags, la imagen de Docker, el manifest del C10 en Kubernetes (GPUs, shm, probes, caché del modelo, secreto de HF)

¿Qué le pasas a vllm serve para que un 70B arranque en 2 H100, atienda solo lo que debe y se apague sin cortarle la respuesta a nadie?

SINTAXIS CON VARIABLES

```
vllm serve modelo \
  --served-model-name nombre \
  --tensor-parallel-size tp \
  --gpu-memory-utilization fracción \
  --max-model-len contexto \
  --max-num-seqs techo \
  --port 8000
```

El arranque mínimo que fijarías en producción. *modelo* es un repo de Hugging Face (`org/nombre`) o una carpeta local con `config.json`, el tokenizer y los pesos. Todo lo demás tiene default, pero los defaults cambian con la versión y con la GPU: fija lo que tu cuenta de capacidad supone. La lista completa de flags está en la página de `vllm serve` de docs.vllm.ai, que tiene una versión por release.

```
docker run --runtime nvidia --gpus all --ipc=host -p
8000:8000 \
  -v ~/.cache/huggingface:/root/.cache/huggingface \
  --env "HF_TOKEN=$HF_TOKEN" \
  vllm/vllm-openai:versión modelo flags
```

La imagen oficial ya trae `ENTRYPOINT ["vllm", "serve"]`: después del nombre de la imagen van solo el modelo y los flags. `--ipc=host` (o `--shm-size`) le da memoria compartida de sobra: los procesos de cada GPU y **NCCL** (la librería de NVIDIA que comunica GPUs) se pasan datos por `/dev/shm`, y los 64 MB que Docker da por defecto no alcanzan con TP (el modelo partido entre varias GPUs). El volumen guarda la caché de Hugging Face entre arranques; `HF_TOKEN` solo hace falta para repos **gated** (con una licencia que aceptar antes de bajar).

```
env:
  - name: HF_TOKEN
    valueFrom:
      secretKeyRef: {name: secreto, key: token}
volumeMounts:
  - {name: shm, mountPath: /dev/shm}
volumes:
  - {name: shm, emptyDir: {medium: Memory, sizeLimit:
tamaño}}
```

En Kubernetes, el token de Hugging Face sale de un Secret, nunca del YAML, y `/dev/shm` es un `emptyDir` en RAM que cuenta contra el límite de memoria del contenedor. El C10 no usa token: sus pesos vienen de su propio S3 y vLLM corre con `HF_HUB_OFFLINE=1`.

```
curl -s -o /dev/null -w '%{http_code}\n' host:8000/health
curl -s host:8000/v1/models
curl -s host:8000/version
```

Tres preguntas a una réplica: ¿el motor vive? (`/health` da 200, o 503 si el engine murió), ¿qué nombre sirve y con qué contexto? (`/v1/models`: `id`, `root` y `max_model_len`) y ¿qué versión corre? (`/version`). A un pod se llega con `kubectl port-forward pod/pod 8000:8000` y *host* = localhost.

```
kubectl logs pod -c vllm | grep 'KV cache size'
```

La capacidad real la imprime vLLM al arrancar, en una línea: `GPU KV cache size: N tokens, Maximum concurrency for M tokens per request: Kx`. Ese número manda; la cuenta de la sección 3 sirve para saber antes de comprar. Para el C10, la cuenta espera ≈ 208,736 tokens y 25.48× para 8,192.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo	repo de Hugging Face o carpeta local con config.json, tokenizer y pesos	`/models/chat-70b-fp8-2026-09-28`
nombre	el nombre que los clientes ponen en model	`chat-70b-fp8-2026-09-28`
tp	en cuántas GPUs se parte cada capa	`2`
fracción	parte de la memoria de cada GPU que vLLM puede usar	`0.90`
contexto	tokens máximos de prompt + respuesta por request	`8192`
techo	secuencias máximas en un paso	`128`
versión	tag de la imagen; en producción, con su digest (la huella sha256 de la imagen, que no cambia)	`v0.30.0@sha256:8a69ffad…`
flags	los mismos flags de vllm serve	`los del manifest, abajo`
secreto · *tamaño*	Secret con el token de Hugging Face · tope de /dev/shm	`el C10 no usa token · 16Gi`
host · *pod*	dónde escucha la réplica · el pod	`localhost con port-forward · un pod de vllm-70b`
s · *t*	segundos de drain (vaciado: terminar lo que está en vuelo sin aceptar trabajo nuevo) · plazo de gracia del pod	`60 · 90`
t_{preStop}	espera antes del SIGTERM para que el pod salga de los endpoints	`15 s`
margen	lo que queda para cerrar procesos y soltar las GPUs	`15 s`

```
--shutdown-timeout s                # 0 (default) =
abortar todo; > 0 = drenar hasta s segundos
terminationGracePeriodSeconds: t    # ≥ preStop + s +
margen
```

El apagado en orden: corre el `preStop` (un paso que Kubernetes corre antes de apagar), llega el SIGTERM (la señal de «termina»), vLLM aborta lo que llega y deja terminar lo que está en vuelo hasta `s` segundos; al vencer ese plazo mata su engine (lo que quedaba termina con un error, no con un abort). Si al vencer `t` el pod sigue vivo, SIGKILL (muerte inmediata).

FÓRMULAS

$$t_{\text{arranque máx}} = \text{failureThreshold} \times \text{periodSeconds} = 90 \times 10\ \text{s} = 900\ \text{s} = 15\ \text{min}$$

$$t_{\text{gracia}} \geq t_{\text{preStop}} + t_{\text{drain}} + \text{margen} \qquad 15 + 60 + 15 = 90\ \text{s}$$

$$t_{\text{drain}} \geq t_{\text{request más largo}} \approx 1{,}024 \times 50\ \text{ms} \approx 51\ \text{s} < 60\ \text{s}$$

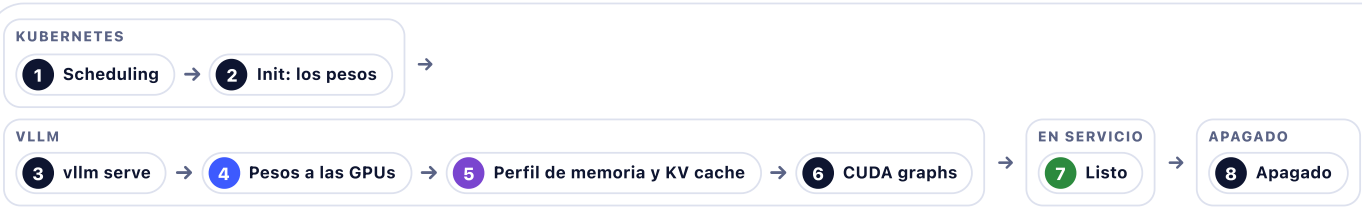
$$\text{nvidia.com/gpu} = \text{TP} \times \text{PP} = 2 \times 1 = 2$$

$$\text{memoria de vLLM} = 0.90 \times 80\ \text{GB} \times 2 = 144\ \text{GB} \qquad \text{(con el default 0.92: 147.2 GB)}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
vllm/vllm-openai:v0.30.0	La imagen que fija el C10 (22 de septiembre de 2026). Hoy la última es v0.32.0 (18 de octubre): defaults y mensajes cambian entre versiones, así que lee la página de vllm serve de la tuya.	**por digest: sha256:8a69ffad…**
vllm serve *modelo*	Levanta el servidor HTTP compatible con OpenAI, el engine (el motor que arma los batches y corre el modelo) y un proceso por GPU. El modelo va primero.	**/models/chat-70b-fp8-2026-09-28: la carpeta que dejó el init container**
--served-model-name	El nombre que vale en el campo model. El pod atiende solo ese: otro da 404 «The model `x` does not exist.»	**chat-70b-fp8-2026-09-28 (el alias chat-default del gateway apunta aquí)**
--load-format safetensors	Solo safetensors. El default auto prueba safetensors y, si no hay, carga los .bin de PyTorch, que son pickle: un formato que puede ejecutar código al abrirse. vLLM v0.30 los abre con weights_only=True, que lo limita; safetensors son solo datos.	**nunca un .bin**
cuantización	Sin flag: vLLM la lee del quantization_config del config.json. --quantization es para forzarla.	**checkpoint FP8: ≈ 70.6 GB de pesos**
--tensor-parallel-size	En cuántas GPUs se parte cada capa (sección 8). El pod pide exactamente esas GPUs.	**2: dos H100 del mismo nodo**
--gpu-memory-utilization	La fracción de cada GPU que vLLM toma para pesos, activaciones y KV cache. Default 0.92 desde v0.20 (0.9 hasta v0.19).	**0.90 explícito: 144 GB entre las dos**
--max-model-len	Tope de prompt + respuesta. Sin el flag sale del config del modelo (131,072 en Llama 3.3 70B): un solo request podría pedir 40 GiB de KV.	**8192**
--max-num-seqs	Techo de secuencias por paso. Con vllm serve, el default es 1,024 en GPUs de 70 GiB o más (salvo la A100) y 256 en las demás.	**128: el KV da para ≈ 92 de 2,250 tokens**
--max-num-batched-tokens	Tokens por paso, sumando decodes y trozos de prefill (sección 6). Con vllm serve en una H100, el default ya es 8,192.	**8192, explícito**
--kv-cache-dtype	Formato del KV cache. auto = el dtype del modelo; fp8 lo achica a la mitad (sección 9).	**auto: KV en BF16**
--enable-prefix-caching · --enable-chunked-prefill	Reusar prefijos y partir prompts largos. En v0.30 ya son default para modelos decoder; explícitos porque el plan cuenta con ellos.	**system prompts de ≈ 1,000 tokens**
--scheduling-policy priority	La cola se ordena por el priority de cada request (menor = antes; empate: el que llegó primero) y en una preemption (cuando falta KV y vLLM saca una secuencia del batch) sale primero el número más alto. Default fcfs: orden de llegada.	**0 interactivo, 10 batch**
--generation-config vllm	Ignora el generation_config.json del modelo y usa los defaults de vLLM. Con auto (default), una revisión nueva puede cambiar la temperatura por defecto sin que nadie lo note.	**los productos mandan su temperatura**
--shutdown-timeout	Con SIGTERM, cuántos segundos deja terminar lo que está en vuelo. Default 0: aborta todo de inmediato.	**60 (un request de 1,024 tokens ≈ 51 s)**
--port	Puerto de la API, /health y /metrics. Default 8000.	**8000, con nombre http en el pod**

PIEZA	QUÉ HACE	EJEMPLO
`sin --enable-log-requests`	Default False. Encendido, en INFO anota el id y los parámetros de cada request; en DEBUG, también el prompt.	el log por request vive en el gateway, sin contenido
`HF_HUB_OFFLINE=1 · VLLM_NO_USAGE_STATS=1`	Ninguna llamada a Hugging Face en runtime (si falta un archivo, error en vez de descarga) y sin telemetría de uso.	pesos solo desde el S3 propio
`HF_TOKEN`	Token de Hugging Face para bajar modelos gated. En Kubernetes, de un Secret (secretKeyRef).	el C10 no lo necesita
`fetch-weights (initContainer)`	Corre antes que vLLM: baja la revisión al NVMe y verifica cada archivo con sha256. Si algo no coincide, el pod no arranca.	aws s3 sync + sha256sum -c
`emptyDir de los pesos`	Disco efímero del nodo: con Karpenter (el que crea los nodos) y su instanceStorePolicy RAID0, son los NVMe locales de la p5 sumados en un solo disco. Cada pod baja su copia; una caché por nodo ahorraría descargas a cambio de más piezas.	sizeLimit: 200Gi, cuenta en ephemeral-storage
`nvidia.com/gpu`	GPUs enteras, iguales en requests y limits: no se comparten ni se sobreasignan.	2 = TP × PP
`/dev/shm`	Memoria compartida en RAM (emptyDir medium: Memory). La usan los procesos de cada GPU y NCCL; cuenta contra el límite de memoria.	sizeLimit: 16Gi (el default de un contenedor es 64 MiB)
`startupProbe · readiness · liveness`	Las tres a /health: 200 si el engine vive, 503 si murió. La de startup da tiempo de carga; readiness saca del Service; liveness reinicia.	90 × 10 s · 3 × 5 s · 6 × 10 s
`preStop + terminationGracePeriodSeconds`	Esperar a salir de los endpoints (la lista de pods a los que el Service manda tráfico) antes del SIGTERM; el plazo total cubre preStop + drain + margen.	sleep de 15 s · 90 s
`runAsUser: 2000`	La imagen corre como root por defecto, pero trae el usuario vllm (UID 2000, GID 0, home /home/vllm) para correr sin root.	HOME=/home/vllm



1 Scheduling KUBERNETES

YAML

```
nodeSelector p5.48xlarge · toleration nvidia.com/gpu · nvidia.com/gpu: 2
```

El pod espera un nodo con 2 GPUs libres. Sin GPUs libres queda Pending; un nodo nuevo tarda ≈ 2–5 min, si hay capacidad.

2 Init: los pesos KUBERNETES

TERMINAL

```
aws s3 sync … && sha256sum -c SHA256SUMS
```

≈ 70 GB de S3 al NVMe (≈ 1–3 min) y verificados (≈ 1 min de CPU). Un archivo cambiado o agregado: el pod no arranca.

3 vllm serve VLLM

TERMINAL

```
vllm serve /models/chat-70b-fp8-2026-09-28 --tensor-parallel-size 2 …
```

Arrancan el servidor HTTP, el engine y un worker por GPU. Lee el config.json y la cuantización FP8.

4 Pesos a las GPUs VLLM

VALOR ≈ 35.3 GB por GPU

Cada GPU carga su mitad de cada capa desde el NVMe local. Desde un disco de red, este paso se vuelve minutos.

5 Perfil de memoria y KV cache VLLM

LOG

```
GPU KV cache size: … tokens, Maximum concurrency for 8,192 tokens per request: …x
```

Un paso de prueba mide las activaciones (y compila con torch.compile si no hay caché); el KV cache recibe lo que queda de 0.90 × 80 GB por GPU. Si no alcanza ni para una secuencia de 8,192, no arranca.

6 CUDA graphs VLLM

Graba las secuencias de kernels de varios tamaños de batch para repetirlas sin overhead. Pasos 4 a 6: ≈ 1–3 min (est.).

7 Listo EN SERVICIO

HTTP `GET /health` → 200

Pasa la startupProbe (tenía hasta 15 min) y la readiness lo agrega al Service.

8 Apagado APAGADO

SECUENCIA

```
preStop sleep 15 s → SIGTERM → drain ≤ 60 s → salida
```

Sale de los endpoints, termina lo que está en vuelo y aborta lo que llega. Si a los 90 s sigue vivo, SIGKILL.

YAML campos del manifest (YAML) TERMINAL un comando: va en la terminal VALOR un número de la cuenta, no un comando

LOG una línea del log de vLLM: la buscas, no la escribes HTTP el request HTTP que llega SECUENCIA una secuencia de pasos, no un comando

La vida de un pod de vllm-70b, del scheduling al SIGKILL. Los tiempos son estimaciones del C10: vLLM no corrió (no hay GPU). La línea del KV cache la imprime vLLM al arrancar, y es la que manda.

EJEMPLO REAL · EL CONTENEDOR VLLM DEL C10: LA IMAGEN POR DIGEST, LOS ARGS DE VLLM SERVE Y SU ENTORNO

YAML reference/c10/k8s/vllm-70b.yaml (líneas 175–221)

```yaml
      - name: vllm
        # vLLM v0.30.0 (2026-09-22) por digest. ENTRYPOINT de la imagen: ["vllm", "serve"] → aquí van solo args.
        image: vllm/vllm-openai:v0.30.0@sha256:8a69ffad015f138d7170c4ddc429e230a3bc1c1719f67e14324749df200a4b90
        args:
          - /models/chat-70b-fp8-2026-09-28  # los pesos locales (checkpoint FP8: la cuantización se lee de su config)
          - --served-model-name
          - chat-70b-fp8-2026-09-28  # el nombre que ve el gateway: el alias chat-default apunta aquí
          - --load-format
          - safetensors  # solo safetensors: un .bin es un pickle. vLLM v0.30 lo abre con weights_only=True (no
                         # ejecuta código arbitrario), pero safetensors es solo tensores: no depende de esa guardia
          - --tensor-parallel-size
          - "2"  # 70.6 GB de pesos FP8 + KV no entran cómodos en 80 GB; en 2 GPUs quedan ≈ 68 GB para KV
          - --gpu-memory-utilization
          - "0.90"  # explícito: el default es 0.92 desde v0.20 (0.9 hasta v0.19); la capacidad no cambia con un upgrade
          - --max-model-len
          - "8192"  # contexto máximo del producto: más largo = menos secuencias a la vez (capacity/vram_calc.py)
          - --max-num-seqs
          - "128"  # techo de secuencias por réplica (el default en H100 es 1,024): el KV da ≈ 92 de 2,250 tokens
          - --kv-cache-dtype
          - auto  # KV en BF16. "fp8" duplica las secuencias; se activa solo si pasa el golden set
          - --enable-prefix-caching  # default en v0.30; explícito porque el plan cuenta con él (system prompts)
          - --enable-chunked-prefill  # default en v0.30: un prompt de 8k no frena el decode de los demás
          - --max-num-batched-tokens
          - "8192"  # tokens por paso (prefill troceado + decode): más = mejor TTFT y peor TPOT; se ajusta con el bench
          - --scheduling-policy
          - priority  # el gateway manda priority 0 (interactive) o 10 (batch). Sin este flag vLLM v0.30 NO falla
                      # (su documentación dice que sí): la cola FCFS ignora priority y el batch compite con el chat
          - --generation-config
          - vllm  # defaults de muestreo de vLLM, no del generation_config.json del modelo: una revisión nueva no
                  # cambia la temperatura por defecto en silencio (los productos mandan la suya)
          - --shutdown-timeout
          - "60"  # con SIGTERM, deja terminar lo que está en vuelo hasta 60 s (default 0 = abortar todo)
          - --port
          - "8000"
        # NO se agrega --enable-log-requests. En INFO (el nivel por defecto) no loguea prompts: solo el id y los
        # parámetros de muestreo de cada request (el prompt sale en DEBUG; la respuesta, con --enable-log-outputs).
        # Es una línea por request que no hace falta, y con un VLLM_LOGGING_LEVEL=DEBUG puesto para depurar
        # volcaría prompts con PII. El log por request vive en el gateway, sin contenido.
        # Tampoco --sse-keep-alive-interval (default 0): el gateway corta un stream tras 30 s sin bytes
        # (read_timeout_s); un ": keep-alive" reiniciaría ese reloj y un vLLM trabado nunca cortaría.
        env:
          - name: HOME
            value: /home/vllm  # caches de vLLM, torch.compile y Triton en un lugar escribible para UID 2000
          - name: HF_HUB_OFFLINE
            value: "1"  # nunca bajar nada de Hugging Face en runtime: los pesos vienen solo de nuestro S3
          - name: VLLM_NO_USAGE_STATS
            value: "1"  # sin telemetría de uso hacia afuera desde una VPC con PII
```

- `image:` fija `v0.30.0` por digest: es el del índice multi-arquitectura que Docker Hub publica para ese tag (22 de septiembre de 2026), y la variante amd64 pesa ≈ 8.7 GB comprimida. Como el ENTRYPOINT ya es `vllm serve`, `args` empieza por la ruta del modelo y no hay `command`.
- Los pesos los trae el init container `fetch-weights` (líneas 121–149): `aws s3 sync` al NVMe local; `echo "${MANIFEST_SHA256} SHA256SUMS" | sha256sum -c -` (el manifiesto de hashes es el que se revisó: su hash está fijado en el YAML); `sha256sum -c SHA256SUMS` (cada archivo) y un `comm -23` que rechaza archivos que no están en el manifiesto. Si algo falla, el pod no arranca: «mejor una réplica menos que un modelo corrupto respondiendo». El lab corre esos tres chequeos.
- GPUs y memoria (líneas 225–233): `nvidia.com/gpu: 2` en requests y limits (TP=2: dos H100 del mismo nodo), `memory: 256Gi` igual en los dos (pasarse es un OOMKilled claro: el kernel mata el proceso por memoria) y `cpu: "40"`, porque tokenizar, el scheduler y el servidor HTTP son CPU. `/dev/shm` es un `emptyDir` con `medium: Memory` y `sizeLimit: 16Gi` (líneas 281–284) que cuenta dentro de esos 256Gi.
- Las tres probes van a `/health` (líneas 238–261): la startupProbe da 90 × 10 s = 15 min para cargar ≈ 70 GB, perfilar la memoria y capturar CUDA graphs; la readiness (cada 5 s, 3 fallas) saca al pod del Service si el engine murió (503); la liveness (6 × 10 s) reinicia un engine colgado. Ninguna sabe si la réplica está llena: eso es la admisión del gateway.

- El apagado (líneas 82–96 y 262–267): `preStop` con `sleep: seconds: 15` (acción nativa, estable desde Kubernetes 1.34) para salir de los endpoints; con el SIGTERM, `--shutdown-timeout 60` pone a vLLM en drain: el servidor HTTP sigue abierto, lo que está en vuelo termina y lo nuevo se aborta enseguida. Sin streaming, eso es un 200 con `finish_reason: "abort"` y sin texto; con streaming, un 200 con el chunk del rol, el usage (si lo pediste) y `[DONE]`, sin ningún `finish_reason`, porque vLLM no manda chunks vacíos. El gateway del C10 trata los dos como falla (502 o el vendor). A los 60 s, vLLM mata el proceso del engine: lo que quede termina con un evento de error o con la conexión cortada, no con un abort. `terminationGracePeriodSeconds: 90` = 15 + 60 + 15.
- `HOME=/home/vllm` porque el pod corre como UID 2000 (el usuario `vllm` que trae la imagen, con GID 0): ahí escribe las cachés de torch.compile. `HF_HUB_OFFLINE=1`: ninguna llamada a Hugging Face en runtime. `VLLM_NO_USAGE_STATS=1`: nada de telemetría de uso hacia afuera.
- Las líneas 200–201 lo dicen: sin `--scheduling-policy priority`, vLLM no rechaza un `priority` distinto de 0, aunque la documentación del campo diga que sí. En el código de v0.30.0 (y de v0.32.0) no hay esa validación: con `fcfs`, la cola es una fila por orden de llegada que no lee el campo. Sin la flag no hay error ni prioridad: el batch compite con el chat.
- Las líneas 209–212 dejan apagado `--enable-log-requests`. Con el log en INFO (el default de `VLLM_LOGGING_LEVEL`) anotaría el id y los parámetros de cada request; el prompt aparece en DEBUG. No hace falta, porque el log por request vive en el gateway, y así subir el nivel de log para depurar no vuelca prompts. Las líneas 213–214 dejan apagado `--sse-keep-alive-interval`: un comentario keep-alive reiniciaría el read timeout con el que el gateway corta un stream trabado.

CÓMO RECORDARLA

modelo, nombre, GPUs, memoria y contexto; después, el apagado. Arrancar vLLM es como abrir un restaurante: llega la mercadería y se revisa contra la factura (init container y sha256), se ordena la cocina (pesos, perfil de memoria, KV cache, CUDA graphs) y recién ahí se abre la puerta (`/health` en 200). Para cerrar: se apaga el letrero (preStop), se termina de atender a quien ya está sentado (drain de 60 s) y se cierra con llave (SIGKILL a los 90 s).

TRAMPA

llamar `vllm` al Service de Kubernetes. El kubelet (el agente de Kubernetes en cada nodo) inyecta en cada pod nuevo del namespace variables con el nombre de cada Service, como `VLLM_PORT=tcp://10.0.12.7:8000`, y vLLM lee `VLLM_PORT` como suya: al final del arranque, ya con los pesos cargados, lee todas sus variables y se cae con `VLLM_PORT 'tcp://…' appears to be a URI`, con cualquier TP. La documentación de vLLM lo advierte. El Service del C10 se llama `vllm-70b`: sus variables (`VLLM_70B_PORT` y compañía) solo dejan el aviso `Unknown vLLM environment variable detected`. La otra: dejar defaults que cambian solos. `--gpu-memory-utilization` pasó de 0.9 a 0.92 en v0.20, y `--max-num-seqs` vale 1,024 con `vllm serve` en una H100 pero 256 en una A100: el mismo YAML se comporta distinto.

En el curso: C10 · Serving | C10 · Archivos de referencia | M6 · GPUs en Kubernetes | M6 · Cold starts de punta a punta | M5 · GPUs dentro del contenedor: NVIDIA Container Toolkit | M15 · Supply chain: imágenes, dependencias y pesos verificados | Hoja · Kubernetes · GPUs y scheduling | Hoja · Kubernetes · probes y apagado | Hoja · Docker · GPU en contenedores

5 La API compatible con OpenAI: chat completions, streaming SSE, parámetros de sampling, structured outputs, el cliente de openai

¿Cómo le hablas a vLLM: qué endpoints tiene, cómo llega un stream token por token y qué parámetros son de OpenAI y cuáles solo de vLLM?

SINTAXIS CON VARIABLES

```
curl -s host:8000/v1/chat/completions -H 'Content-Type: application/json' -d '{
  "model": "nombre",
  "messages": [{"role": "system", "content": "instrucciones"},
               {"role": "user", "content": "pregunta"}],
  "max_completion_tokens": n, "temperature": t
}'
```

Un request y una respuesta JSON: `choices[0].message.content`, `finish_reason` y `usage` (`prompt_tokens`, `completion_tokens`, `total_tokens`). `messages` es la conversación con roles; vLLM la convierte en texto con el **chat template** del modelo (la plantilla que marca dónde empieza y termina cada turno) antes de tokenizar. `/v1/completions` recibe un `prompt` crudo, sin template.

```
curl -sN host:8000/v1/chat/completions -H 'Content-Type: application/json' -d '{
  "model": "nombre", "messages": [{"role": "user", "content": "pregunta"}],
  "stream": true, "stream_options": {"include_usage": true}
}'
```

Streaming con Server-Sent Events (SSE): por cada evento, una línea `data: chunk` y una línea vacía. El primero trae solo el rol; después, un `delta.content` (el texto nuevo) por cada paso de decode, la vuelta del modelo que genera un token; el último chunk trae el `finish_reason` con el último texto (o con `content` vacío, si el modelo paró con su token de fin); con `include_usage`, un chunk más con `choices: []` y el `usage`; al final, `data: [DONE]`. `-N` le quita el buffer a curl para que veas cada evento al llegar.

```
client = OpenAI(base_url="http://host:8000/v1", api_key="llave")
client.chat.completions.create(model="nombre", messages=[…], stream=True,
    stream_options={"include_usage": True}, extra_body={"top_k": k, "priority": p})
```

El cliente oficial de `openai` sirve tal cual: cambias `base_url`. Lo que no existe en OpenAI va en `extra_body`. Por defecto el cliente reintenta 2 veces ante 408, 409, 429 y 5xx (respetando `Retry-After`) y espera hasta 600 s: fija `timeout` y `max_retries`.

```
"response_format": {"type": "json_schema",
                    "json_schema": {"name": "esquema", "schema": json_schema}}
"structured_outputs": {"choice": ["a", "b"]}     # o "regex", "json", "grammar"
```

Structured outputs (salidas con forma garantizada): el motor solo deja pasar tokens que mantienen la salida válida contra el schema. `response_format` es la forma de OpenAI; `structured_outputs` es la de vLLM, con una sola restricción por request. La librería que aplica la restricción (el backend) la elige vLLM: `--structured-outputs-config.backend`, default `auto`. Garantiza la forma, no que el contenido sea verdad.

```
curl -s host:8000/v1/models
curl -s -o /dev/null -w '%{http_code}\n' host:8000/health
```

`/v1/models` lista lo que sirve el pod: `id` (el `--served-model-name`), `root` (la ruta) y `max_model_len`. El mismo servidor expone también `/v1/completions`, `/v1/responses`, `/health`, `/metrics` y `/version`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
host	dónde escucha vLLM	`vllm-70b.c10.svc.cluster.local` (el Service), o `localhost` con port-forward
nombre	el --served-model-name	`chat-70b-fp8-2026-09-28` (el gateway traduce `chat-default`)
instrucciones · *pregunta*	el mensaje system y el de la persona	≈ 1,000 tokens fijos por producto + ≈ 1,000 propios
n	tope de tokens de la respuesta	1,024 para soporte (tenants.yaml)
t	temperatura: 0 = siempre el token más probable; más alta = más variedad	la manda cada producto
llave	la API key; sin --api-key en vLLM, cualquier texto	el gateway no le manda llave a vLLM: lo protege la NetworkPolicy
k · *p*	top_k (elegir solo entre los k tokens más probables) · priority	– · 0 interactivo, 10 batch
esquema · *json_schema*	nombre y JSON Schema de la salida	atributos del catálogo
a · *b*	las únicas salidas permitidas	"positivo" · "negativo"
n_{out}	tokens de la respuesta (completion_tokens)	≈ 250 en promedio en el C10

FÓRMULAS

$$\text{eventos de un stream} \approx 1_{\text{rol}} + n_{\text{out}} + 1_{\text{usage}} + 1_{[\text{DONE}]} \qquad \text{(--stream-interval 1, el default)}$$

$$\text{total_tokens} = \text{prompt_tokens} + \text{completion_tokens}$$

v0.30: prompt_tokens + max_tokens ≤ max_model_len (si no, 400) v0.32: max_tokens ← min(max_tokens, max_model_len − prompt_tokens)
C10, soporte: 6,000 + 1,024 = 7,024 ≤ 8,192 (max_prompt_tokens + max_tokens_cap)

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
POST /v1/chat/completions	Conversación con roles → chat template → tokens. Es la que usa el C10.	**casi siempre con stream: true**
POST /v1/completions	Texto crudo en prompt, sin chat template: tú pones las marcas de cada turno.	**benchmarks y modelos base**
GET /v1/models	Lo que sirve el pod: id, root y max_model_len.	**id = chat-70b-fp8-2026-09-28**
model	Tiene que ser el --served-model-name. Otro nombre: 404 «The model `x` does not exist.»	**un pod canario sirve otro nombre**
max_completion_tokens (max_tokens)	Tope de la respuesta; OpenAI marcó max_tokens como obsoleto (deprecated) y vLLM acepta los dos. Sin tope, vLLM genera hasta llenar el contexto.	**si no viene, el gateway pone el tope del tenant**
tope + prompt > max_model_len	En v0.30 el tope se reserva entero: si no entra, un 400 («This model's maximum context length is…»). Desde v0.32 entra y la salida se recorta a lo que queda.	**7,500 + 1,024 > 8,192: 400 en v0.30; en v0.32, hasta 692 tokens**
temperature · top_p · seed	De OpenAI: el azar del sampling (el sorteo de cada token; de 0 a 2), el núcleo de probabilidad (solo los tokens más probables hasta sumar top_p) y la semilla. Si faltan, valen los defaults del motor: con --generation-config vllm, temperature 1.0 y top_p 1.0.	**extracción: temperature 0**
top_k · min_p · repetition_penalty	Solo de vLLM (van en extra_body): los k más probables, un piso de probabilidad y un castigo por repetir.	**—**
stop · stop_token_ids	Cortar al ver un texto o un token: finish_reason = stop.	**—**
stream · stream_options	SSE; include_usage agrega el chunk final con usage (choices vacío). stream_options sin stream: error.	**el gateway lo fuerza para conciliar la cuota**
finish_reason	stop: terminó solo o por stop; length: llegó al tope (respuesta cortada); tool_calls; abort (de vLLM): lo cortó el servidor, como un pod en drain.	**length con texto cortado = tope chico**
usage	prompt_tokens, completion_tokens y total_tokens: lo que se cobra y lo que mide la cuota.	**≈ 2,000 + 250 en el C10**
response_format json_schema	La forma de OpenAI de pedir JSON contra un schema.	**atributos del catálogo**
structured_outputs	La de vLLM: choice, regex, json, json_object, grammar o structural_tag; una sola por request.	**{"choice": ["positivo", "negativo"]}**
guided_json y compañía	Se quitaron en v0.12: hoy se ignoran y la respuesta sale sin restricción (desde v0.30, con un warning en el log del servidor, una sola vez por proceso).	**tutoriales viejos**
priority · X-Vllm-Priority	Menor = antes. Solo hace algo con --scheduling-policy priority. El header pisa al campo del cuerpo.	**0 interactivo, 10 batch**
--api-key (VLLM_API_KEY)	Una o varias llaves compartidas (sirve cualquiera) que protegen /v1, /v2, /inference y /cohere; el resto queda abierto, incluso /invocations, que también genera texto. No es autenticación por tenant.	**el C10 usa NetworkPolicy + gateway**
user	vLLM lo ignora; en OpenAI lo están reemplazando safety_identifier y prompt_cache_key.	**el gateway del C10 lo usa para elegir canario**
cache_salt	Solo de vLLM: requests con la misma sal comparten prefijos en caché; otra sal, otra caché.	**una sal por tenant (pendiente en el C10)**

DESPUÉS DEL PREFILL: 1 t ≈ 0.30 s · el rol → UN EVENTO POR TOKEN: 2 mismo instante · 1.er token → 3 +40 ms por token → 4 el último token → EL CIERRE: 5 usage → 6 fin

1 t ≈ 0.30 s · el rol — DESPUÉS DEL PREFILL

EVENTO SSE

```
data: {"object": "chat.completion.chunk", "choices": [{"index": 0, "delta": {"role": "assistant", "content": ""}, "finish_reason": null}]}
```

Sale cuando el prefill (la lectura del prompt entero) ya produjo el primer token. Trae choices, pero no texto.

2 mismo instante · 1.er token UN EVENTO POR TOKEN

EVENTO SSE

```
data: {… "delta": {"content": "Tu"}, "finish_reason": null}]}
```

Aquí la persona empieza a leer: el TTFT que importa.

3 +40 ms por token UN EVENTO POR TOKEN

EVENTO SSE

```
data: {… "delta": {"content": " pedido"}, "finish_reason": null}]}
```

Un evento por paso de decode (--stream-interval 1, el default). El espacio viaja pegado al token.

4 el último token UN EVENTO POR TOKEN

EVENTO SSE

```
data: {… "delta": {"content": " mañana"}, "finish_reason": "length"}]}
```

El finish_reason viaja con el último texto. length: se cortó por max_completion_tokens (4).

5 usage EL CIERRE

EVENTO SSE

```
data: {… "choices": [], "usage": {"prompt_tokens": 4, "completion_tokens": 4, "total_tokens": 8}}
```

Solo con include_usage. choices viene vacío: el código que lee choices[0] revienta aquí.

6 fin EL CIERRE

EVENTO SSE `data: [DONE]`

No es JSON. Si la conexión se corta antes, no llegan ni el usage ni el [DONE].

EVENTO SSE un evento del stream (Server-Sent Events): lo manda vLLM, no lo escribes

Un stream de 4 tokens con include_usage, evento por evento: la salida del mock del lab, con el orden de vLLM v0.30.0 (sin id, created ni model). Tiempos del mock: 0.3 s hasta el primer token y 40 ms entre tokens.

EJEMPLO REAL · UN CLIENTE DE OPENAI CONTRA VLLM: STREAMING, USAGE Y PARÁMETROS PROPIOS DE VLLM

PYTHON ilustrativo, con los nombres del C10 (k8s/vllm-70b.yaml y gateway/tenants.yaml). Revisado contra el cliente openai-python v3.37.0 (OpenAI, chat.completions.create, stream_options, extra_body) y el protocolo de vLLM v0.30.0; no se ejecutó: el paquete openai no está instalado en esta máquina

```python
from openai import OpenAI

# Sin --api-key en vLLM, cualquier texto sirve de llave; el cliente no arranca sin una.
client = OpenAI(base_url="http://localhost:8000/v1", api_key="EMPTY",
                timeout=120.0)  # default: 600 s; en un stream es el silencio máximo entre chunks

stream = client.chat.completions.create(
    model="chat-70b-fp8-2026-09-28",           # el --served-model-name, no el alias del gateway
    messages=[{"role": "system", "content": "Eres el asistente de soporte del marketplace."},
              {"role": "user", "content": "¿Cuándo llega mi pedido 4512?"}],
    max_completion_tokens=512,
    temperature=0.3,
    stream=True,
    stream_options={"include_usage": True},    # un chunk final con usage y choices vacío
    extra_body={"top_k": 20, "priority": 0},    # solo de vLLM: viajan en el JSON tal cual
)
for chunk in stream:
    if chunk.choices:                           # el chunk de usage trae choices = []
        choice = chunk.choices[0]
        if choice.delta.content:
            print(choice.delta.content, end="", flush=True)
        if choice.finish_reason == "length":
            print("\n[cortada por max_completion_tokens]")
    if chunk.usage:
        print(f"\n{chunk.usage.prompt_tokens} + {chunk.usage.completion_tokens} tokens")
```

- `base_url` termina en `/v1`: el cliente le agrega `/chat/completions`. En el C10 los productos no llaman a vLLM: llaman al llm-gateway con `model="chat-default"`, y el gateway pone la revisión, el `priority` y el `include_usage` (`_self_body`, en gateway/llm_gateway.py, líneas 574–578). Directo a vLLM es para probar una réplica con port-forward.
- `extra_body` mezcla esas claves en el JSON y vLLM las lee como suyas (`top_k`, `priority`, `structured_outputs`, `cache_salt`). Un servidor que no las conoce puede rechazarlas: ahí el código deja de ser portable.
- `priority` solo cambia algo si vLLM corre con `--scheduling-policy priority`. La documentación del campo dice que, sin esa política, un valor distinto de 0 da error; en el código de v0.30.0 no encontré esa validación (la cola `fcfs` no lo lee). No cuentes con el error: pon la flag.
- El TTFT (el tiempo hasta el primer token) que siente la persona es el primer `delta.content` con texto: el primer chunk trae solo el rol (en vLLM sale junto con el primer token). Para medir con este cliente, toma la hora justo antes de `create` (sección 7).
- `finish_reason == "length"` es una respuesta cortada por el tope, no un error del servidor. `"abort"`, propio de vLLM, es un request que cortó el servidor (por ejemplo, un pod que se está apagando): el gateway del C10 lo trata como una falla del servidor de atrás (el upstream). Si lo corta antes del primer token, un stream no trae ese `finish_reason`: solo el rol, el usage y `[DONE]` (vLLM v0.30 no manda chunks vacíos). Por eso el gateway del C10 trata un `[DONE]` sin texto como falla (502 o el vendor), no como una respuesta vacía.

CÓMO RECORDARLA

es la API de OpenAI con otra base_url. Lo que existe en OpenAI va en el cuerpo tal cual; lo que es solo de vLLM (`top_k` , `priority` , `structured_outputs` , `cache_salt`) va en `extra_body` . Un stream es una fila de sobres `data: …` : el primero trae el rol, cada uno de los siguientes un pedazo de texto, el penúltimo (si lo pediste) la cuenta de tokens, y el último dice [DONE].

TRAMPA

copiar un tutorial viejo con `extra_body={"guided_json": schema}` . Los campos `guided_*` se quitaron en v0.12: vLLM contesta 200 con texto libre, y desde v0.30 solo deja un warning en su propio log, una sola vez. Usa `response_format` con `json_schema` o `structured_outputs` . La otra: leer `chunk.choices[0]` sin mirar. Con `include_usage` , el chunk de usage trae `choices: []` y el cliente revienta justo al final de una respuesta que salió bien.

En el curso: M14 · Prefill y decode: TTFT, TPOT, throughput y goodput · M14 · Guardrails: inyección, PII, salidas estructuradas y cuotas · C10 · Serving · M7 · Protocolos: REST, gRPC, KServe v2 y OpenAI-compatible · M7 · Servidores de LLMs: vLLM, SGLang, TGI, TensorRT-LLM · Hoja · El gateway · contrato y 499 · Hoja · El gateway · timeouts y reintentos

6 Continuous batching y el scheduler: prefill contra decode en el mismo batch, chunked prefill, max_num_batched_tokens, latencia contra throughput

¿Qué decide el scheduler de vLLM en cada paso, por qué un prompt largo frena a los que ya están generando y qué perilla mueve el TTFT contra el TPOT?

SINTAXIS CON VARIABLES

`--max-num-seqs` S `--max-num-batched-tokens` B

Los dos techos de cada **paso** (una pasada del modelo por la GPU): como mucho S secuencias corriendo y B tokens procesados, sumando el token de cada **decode** (una secuencia que ya genera: 1 token por paso) y los trozos de **prefill** (la lectura de un prompt nuevo). El **scheduler** (la parte de vLLM que decide quién entra en cada paso) arma el batch de nuevo en cada paso.

`--enable-chunked-prefill --long-prefill-token-threshold` u

Chunked prefill: un prompt que no entra en el presupuesto del paso se parte en trozos. u limita el trozo de un prompt largo para que no se coma el paso entero; 0, el default, no pone límite. Trozos grandes le dan mejor TTFT (tiempo hasta el primer token) al prompt nuevo y más tirón a los demás; trozos chicos, al revés.

`--scheduling-policy priority` `# y en cada request: "priority":` p

Con `priority`, la cola de espera se ordena por número (menor = antes; empate: el que llegó primero) y, si faltan bloques de KV cache (la memoria de lo que cada secuencia ya leyó), se desaloja primero el número más alto. No saca a nadie del batch para que entre otro: eso lo resuelven los lugares reservados del gateway.

```
sum by (pod) (vllm:num_requests_running)
sum by (pod) (vllm:num_requests_waiting)
```

El batch de verdad (`running`) y la cola (`waiting`) de cada réplica. Si `waiting` sube con `running` pegado al techo, falta capacidad o sobra contexto.

```
histogram_quantile(0.5, sum by (le)
(rate(vllm:iteration_tokens_total_bucket[5m])))
```

Tokens por paso. Una mediana cerca del número de secuencias corriendo es decode puro; los pasos de miles de tokens son prefills. Ojo: vLLM anota el prompt entero (sin lo que estaba en caché) en el paso donde termina su prefill, aunque lo haya partido en trozos. Los bordes de este histograma van de 1 a 16,384, con 8,192 incluido.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
S	máximo de secuencias corriendo en un paso	`128 en el C10 (default 1,024 en una H100)`
B	máximo de tokens por paso: decodes + trozos de prefill	`8,192`
u	tope del trozo de un prompt largo; 0 = sin tope	`0 (default)`
p	prioridad del request: menor = antes	`0 interactivo, 10 batch`
n_{decode}	secuencias que generan: 1 token cada una por paso	`≈ 39 planeadas, 72 en el codo`
c_j, P	trozo de prefill del request j en este paso · tokens de su prompt que no están en caché	`P ≈ 1,000 con el prefijo en caché`
L, λ, W	secuencias en vuelo · requests por segundo · segundos por request	`≈ 39 · 3.75 · 10.5`
n_i	tokens que genera el request i	`A = 3 … B = 8 en la figura`

FÓRMULAS

$$\underbrace{n_{\text{decode}}}_{\text{1 token c/u}} + \sum_j c_j \le B \qquad n_{\text{decode}} + n_{\text{prefill}} \le S$$

$$\text{C10 en el codo: } 72 + c \le 8{,}192 \Rightarrow c \le 8{,}120 \text{ tokens de prefill por paso}$$

$$\text{pasos de un prefill} = \left\lceil \frac{P}{B - n_{\text{decode}}} \right\rceil \qquad \left\lceil \frac{6{,}000}{2{,}048 - 72} \right\rceil = 4$$

$$L = \lambda \times W \qquad \frac{60 \text{ rps} \times 10.5 \text{ s}}{16} \approx 39 \text{ por réplica} \qquad 6.06 \times 11.89 \approx 72$$

$$\text{ocupación} = \frac{\sum_i n_i}{\text{lugares} \times \text{pasos}} \qquad \text{estático: } \frac{25}{4 \times 12} \approx 52\% \qquad \text{continuo: } \frac{25}{4 \times 8} \approx 78\%$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`continuous batching`	El batch se rearma en cada paso: cuando una secuencia termina, otra entra en el paso siguiente. También se llama iteration-level scheduling.	`cada réplica del C10`
`dynamic batching (Triton)`	Junta requests completos y los ejecuta juntos; para generar token a token, el grupo esperaría al más largo.	`hoja de Triton: el bus que espera 2 ms`
`paso (step)`	Una pasada del modelo: cada secuencia en decode suma 1 token y cada prefill, su trozo.	`≈ 45 ms con 72 secuencias en los fixtures`
`scheduler de vLLM V1`	V1 es el motor actual. No tiene fase de prefill ni de decode: en cada paso le da tokens a cada request para que lo calculado alcance a su prompt más lo generado.	`num_computed_tokens alcanza a num_tokens`
`primero los que corren`	Primero entran los que ya están en el batch (decodes y prefills a medias); con el presupuesto que sobra, los que esperan.	`por eso un decode no espera detrás de un prompt nuevo`

PIEZA	QUÉ HACE	EJEMPLO
max_num_seqs	Techo de secuencias corriendo. Más alto = más throughput y más TPOT para cada una, hasta que se acaba el KV.	**128 > las 92 que caben de 2,250 tokens**
max_num_batched_tokens	Presupuesto de tokens por paso. Chico (≈ 2,048) mejora el ITL; grande mejora el TTFT y el throughput (la doc sugiere más de 8,192 para throughput).	**8,192 de partida; se ajusta con el barrido**
chunked prefill	Parte el prompt según el presupuesto que queda en el paso. Default en v0.30.	**un prompt de 6,000 entra en un paso de 8,192**
long_prefill_token_threshold	Tope por request para el trozo de un prompt largo. 0 = sin tope.	**0**
prefix caching	Los tokens del prefijo en caché no se recalculan: el prefill arranca donde termina el acierto.	**≈ 1,000 de 2,000 tokens**
preemption	Si faltan bloques de KV para seguir, el scheduler saca una secuencia, libera sus bloques y la recalcula después (sección 3). Con fcfs sale la última que entró; con priority, el número más alto.	**el batch, de noche**
max_num_queued_reqs	Tope propio de vLLM para esperando + corriendo: lo que pasa recibe 503. Default: sin tope.	**el C10 hace la admisión en el gateway**
vllm:num_requests_running · waiting	Gauges (valores que suben y bajan) del batch y de la cola de cada réplica, ahora.	**KVCacheSaturation mira waiting > 0**
vllm:iteration_tokens_total	Histograma de tokens por paso.	**muestras de miles = prefills que terminan**
latencia contra throughput	Más secuencias por paso = más tokens/s de la réplica y cada persona más lenta: el decode lee pesos + KV de todas en cada paso.	**72 → 1,514 tokens/s con TPOT p95 49.6 ms (fixtures)**

6 REQUESTS, 4 LUGARES: **A** 3 tokens · **B** 8 tokens · **C** 2 tokens · **D** 5 tokens · **E** 4 tokens · **F** 3 tokens

Estático: el grupo sale junto y espera al más largo

paso	1	2	3	4	5	6	7	8	9	10	11	12
1	A1	A2	A3						E1	E2	E3	E4
2	B1	B2	B3	B4	B5	B6	B7	B8	F1	F2	F3	
3	C1	C2										
4	D1	D2	D3	D4	D5							

12 pasos · 25 de 48 lugares con trabajo = **52 %** · primer token de E en el paso 9 y de F en el paso 9

Continuo: en cada paso sube el que espera

paso	1	2	3	4	5	6	7	8	9	10	11	12
1	A1	A2	A3	F1	F2	F3						
2	B1	B2	B3	B4	B5	B6	B7	B8				
3	C1	C2	E1	E2	E3	E4						
4	D1	D2	D3	D4	D5							

8 pasos · 25 de 32 lugares con trabajo = **78 %** · primer token de E en el paso 3 y de F en el paso 4

- cada paso, cada secuencia del batch saca un token
- estático: el grupo siguiente entra cuando termina el más largo
- continuo: un lugar que se libera se ocupa en el paso siguiente

X1 el paso del prefill: sale el primer token · Xn un paso de decode: un token más · · lugar vacío: la GPU no tiene trabajo para él

6 requests que llegan juntos, 4 lugares en el batch, un token por paso (el primero sale del paso de su prefill). Estático: el grupo espera al más largo (B, 8 tokens) y E y F recién empiezan en el paso 9. Continuo: en el lugar que libera C entra E en el paso 3, y todo termina en 8 pasos. Es la salida de bus.py, del lab.

EJEMPLO REAL · CUÁNTAS SECUENCIAS HAY EN EL BATCH: LA LEY DE LITTLE EN FLEET_CALC.PY

PYTHON reference/c10/capacity/fleet_calc.py (líneas 136–143 y 214–216; # … marca líneas omitidas)

```python
def request_seconds(ttft_s: float, output_tokens: float, tpot_s: float) -> float:
    """Duración de un request de streaming ≈ TTFT + tokens de salida × TPOT."""
    return ttft_s + output_tokens * tpot_s


def littles_law_concurrency(rps: float, seconds_per_request: float) -> float:
    """L = λ × W (#m03.little): secuencias vivas a la vez = llegadas por segundo × segundos que dura cada una."""
    return rps * seconds_per_request
# …
    secs = request_seconds(0.5, inter.output_tokens, 0.040)
    conc = littles_law_concurrency(CASE_DESIGN_RPS, secs)
```

```
    print(f"Little: {CASE_DESIGN_RPS:.0f} rps × {secs:.1f} s = {conc:.0f} secuencias vivas ≈ {conc / fl.replicas_deployed:.1f} por réplica")
```

- Con continuous batching no fijas el tamaño del batch: lo fija el tráfico. `request_seconds(0.5, 250, 0.040)` da 10.5 s por request, y 60 rps × 10.5 s = 630 secuencias vivas en la flota: ≈ 39.4 por réplica. `python3 capacity/fleet_calc.py` lo imprime así.
- Cuatro números de batch del C10, de menor a mayor: ≈ 39 planeadas (Little); 72 en el codo del barrido (sección 7); ≈ 92 que caben en el KV con 2,250 tokens (sección 3); y 128 de `--max-num-seqs`. El que manda es un quinto: el tope del gateway, 288 en vuelo por cada una de sus 4 réplicas, o sea 288 × 4 ÷ 16 = 72 por réplica de vLLM, justo el codo. El techo de vLLM queda arriba como margen.
- `--max-num-batched-tokens 8192` (manifest, línea 198): «más = mejor TTFT y peor TPOT; se ajusta con el bench». Con 72 decodes por paso sobran 8,120 tokens para prefill: un prompt de 6,000 sin caché entra en un solo paso, y ese paso tarda más para las 72 secuencias que esperaban su token.
- La alerta `LLMTPOTHigh` lista las causas de batching que suben el TPOT (monitoring/alerts.yml, línea 107): «¿Subió la concurrencia por réplica (vllm:num_requests_running), hay preemptions (LLMPreemptionsRising) o prefills largos intercalados (max-num-batched-tokens)?».
- Lo que no está aquí: el tiempo por request sale de un TTFT y un TPOT supuestos. Con los del barrido (sección 7), Little da justo la concurrencia del punto: 6.06 req/s × 11.89 s ≈ 72.

CÓMO RECORDARLA

continuous batching es un bus que deja bajar y subir pasajeros en cada parada, en vez de esperar a que todos lleguen al final. El de Triton espera 2 ms en la terminal y sale con todos; este no vuelve nunca a la terminal, pero cada parada tiene cupo: `max_num_seqs` asientos y `max_num_batched_tokens` de equipaje. Un prompt largo es equipaje grande: o sube entero y esa parada tarda (buen TTFT para él, tirón para los demás), o sube en partes (chunked prefill).

TRAMPA

subir `--max-num-batched-tokens` para que «vaya más rápido». El throughput sube, pero cada paso con un prefill grande frena a todas las secuencias que están generando: el TPOT promedio casi no cambia y el p99 del ITL se dispara (los tirones). En el modelo de juguete del lab, 8,192 da un TTFT de 495 ms y un hueco de 495 ms para los 72 que generaban; 2,048 da 630 ms y 193 ms. Decídelo con el barrido, mirando el p99 del ITL y no solo el promedio.

En el curso: M14 · Batching, caching, speculative decoding y paralelismo · M7 · Servidores de LLMs: vLLM, SGLang, TGI, TensorRT-LLM · M3 · Little's law: la cuenta que ordena todo · M3 · Batching: throughput a cambio de latencia · C10 · Serving · Hoja · De PyTorch a Triton · dynamic batching

7 Medir: TTFT, TPOT/ITL, E2E, tokens/s, goodput; métricas vllm: en Prometheus; el barrido de concurrencia del C10

¿Cuántos requests aguanta una réplica dentro del SLO, cómo se mide sin engañarse y qué métricas de vLLM miras en producción?

SINTAXIS CON VARIABLES

```
vllm bench serve --backend openai-chat --endpoint
/v1/chat/completions \
  --base-url url --model nombre --tokenizer ruta \
  --dataset-name dataset --num-prompts N --max-
concurrency c --request-rate inf \
  --percentile-metrics ttft,tpot,itl,e2el --metric-
percentiles 50,95,99 \
  --goodput ttft:ms tpot:ms --save-result --result-dir
dir --result-filename archivo
```

Barrido cerrado: como mucho `c` requests en vuelo; el siguiente sale cuando termina uno. Lo repites con varios `c` y te quedas con el último que cumple. Sin `--metric-percentiles` solo hay p99 (el tiempo bajo el que queda el 99 % de los requests; el default es `99`), y sin `--percentile-metrics` no hay E2E (el default es `ttft,tpot,itl`).

```
vllm bench serve … --request-rate λ --burstiness 1.0
```

Barrido abierto: llegadas de Poisson (al azar, con un promedio de `λ` por segundo), como el tráfico real. Sin `--max-concurrency`, la cola se forma en el servidor y se ve en el TTFT; el cerrado nunca deja que esa cola crezca (**coordinated omission**: el cliente deja de mandar justo cuando el servidor se frena, y esa espera nunca se mide).

```
python3 bench/pick_concurrency.py results/sweep-c*.json
--ttft-ms 1000 --tpot-ms 50 --percentile 95
```

El lector del C10: ordena los puntos, anota las fallas de cada uno y elige el último bueno antes del primero malo. Sale con código 2 si a un JSON le falta el percentil pedido.

```
histogram_quantile(0.95, sum by (le)
(rate(vllm:time_to_first_token_seconds_bucket[5m])))
```

El p95 de toda la flota: `rate`, `sum by (le)` y recién ahí el cuantil, como en la hoja de Prometheus. Por pod: `sum by (le, pod)`. Nunca promedies los p95 de cada pod.

```
sum(rate(vllm:generation_tokens_total[5m]))
# tokens de salida/s
sum(rate(vllm:request_time_per_output_token_seconds_sum[5m])
)
  /
sum(rate(vllm:request_time_per_output_token_seconds_count[5m
]))  # TPOT medio
```

Los counters (contadores que solo suben) llevan `_total` en Prometheus y se leen con `rate`. La media de un histograma es `_sum` ÷ `_count`; la cola (p95, p99) sale de los buckets.

```
curl -s host:8000/metrics | grep -E '^vllm:
(num_requests_(running|waiting)|kv_cache_usage_perc|num_pree
mptions_total)'
```

Lo que expone una réplica, sin Prometheus de por medio. Cada serie trae los labels `model_name` y `engine`; el PodMonitor del C10 agrega `pod` y `track`.

```
Engine 000: Avg prompt throughput: x tokens/s, Avg
generation throughput: y tokens/s, Running: r reqs,
Waiting: w reqs, GPU KV cache usage: k%, Prefix cache hit
rate: h%
```

La línea de estadísticas que vLLM escribe en su log cada 10 s (`VLLM_LOG_STATS_INTERVAL`; se apaga con `--disable-log-stats`). Si en ese intervalo hubo preemptions, agrega `Preemptions: N` antes del KV: en V1 es el único rastro en el log, porque el scheduler no escribe una línea por cada preemption. Con la réplica ociosa, la línea pasa a DEBUG y no aparece.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
TTFT	time to first token: de que llega el request al primer token	p95 ≤ 1 s (el SLO, la meta de servicio, medido en el gateway)
TPOT	time per output token: el promedio entre tokens de un request, sin contar el primero	p95 ≤ 50 ms
ITL	inter-token latency: el hueco entre dos tokens seguidos; el TPOT es su promedio dentro de un request	p99 109 ms a c = 72 (fixtures)
E2E	end-to-end: el request completo	≈ 11.9 s a c = 72
n_{out}	tokens de salida del request (usage.completion_tokens)	250, con --ignore-eos
c	requests en vuelo (--max-concurrency)	16 … 96
λ	requests por segundo completados (request_throughput)	6.06 a c = 72
W	segundos que dura cada request	11.89
goodput	requests por segundo que cumplen todos los SLOs, uno por uno (request_goodput)	5.75
url · nombre · ruta	la réplica · su --served-model-name · el tokenizer local	http://localhost:8000 · chat-70b-fp8-2026-09-28 · /models/chat-70b-fp8-2026-09-28
dataset	de dónde salen los prompts	prefix_repetition: 3 prefijos de 1,000 + 1,000 propios, 250 de salida
N	requests por punto	1,000 (con menos de 500, el p95 es ruido)
ms	umbrales del SLO para el goodput, en ms	ttft:1000 tpot:50
λ	tasa del barrido abierto	4.2 (lo planeado)
dir · archivo	dónde guardar cada JSON	results · sweep-c072.json
x · y · r · w · k · h	tokens/s de prompt y de generación del intervalo · requests corriendo · esperando · % del KV ocupado · % de aciertos del prefix cache	Running: 72 reqs con el tope del gateway lleno

FÓRMULAS

$$\text{TTFT} = t_{\text{primer token}} - t_{\text{llegada}} \qquad \text{E2E} = t_{\text{último token}} - t_{\text{llegada}}$$

$$\text{TPOT} = \frac{\text{E2E} - \text{TTFT}}{n_{\text{out}} - 1} \quad (n_{\text{out}} > 1) \qquad \text{ITL}_k = t_{k+1} - t_k$$

$$\text{E2E} \approx \text{TTFT} + (n_{\text{out}} - 1) \times \text{TPOT} = 0.56 + 249 \times 0.0455 = 11.89\ \text{s} \quad (c = 72)$$

$$\lambda = \frac{c}{W} = \frac{72}{11.89\ \text{s}} = 6.06\ \text{req/s} \qquad \text{tokens/s} = \lambda \times n_{\text{out}} = 6.06 \times 250 \approx 1{,}514$$

$$\text{goodput} = \frac{\#\{\text{requests con TTFT} \le 1\ \text{s y TPOT} \le 50\ \text{ms}\}}{\text{duración}} = 5.75\ \text{req/s} = 95\%\ \text{de}\ 6.06$$

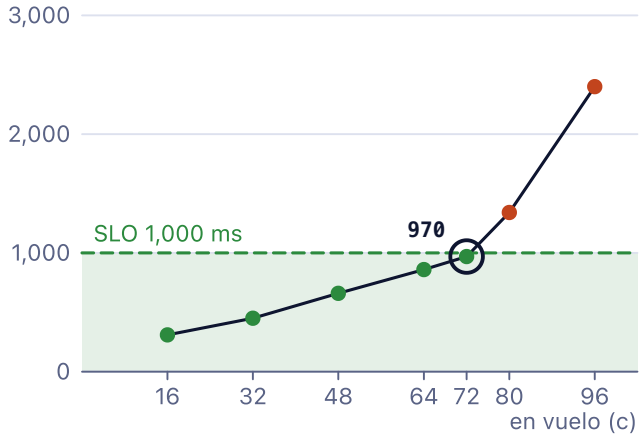
$$\text{planeado} = 0.70 \times 6.06 = 4.24\ \text{req/s por réplica}$$

LAS PIEZAS

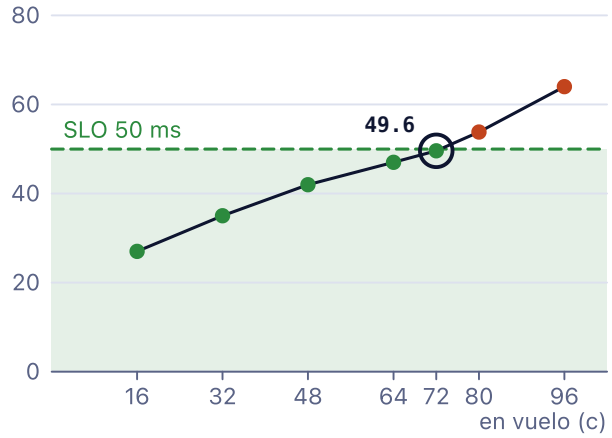
PIEZA	QUÉ HACE	EJEMPLO
vllm bench serve	El cliente de carga de vLLM: manda requests de verdad a la API y mide del lado del cliente.	viene con vLLM: corre donde lo tengas instalado
--max-concurrency	Barrido cerrado: tope de requests en vuelo. Lo que un request espera turno en el cliente no entra en su TTFT: v0.30 lo guarda aparte (la lista queue_times del JSON; sus percentiles, solo si agregas client_queue_time a --percentile-metrics).	16, 32, 48, 64, 72, 80, 96
--request-rate	inf (default): todos salen en t = 0 y los frena --max-concurrency. Un número: llegadas de Poisson (con --burstiness 1).	inf en el cerrado; 4.2 en el abierto
--dataset-name prefix_repetition	Prompts sintéticos con prefijos compartidos, para que el prefix caching pese como en producción.	3 prefijos, 1,000 + 1,000 tokens, 250 de salida
--ignore-eos	Que cada request genere sus 250 tokens aunque el modelo quiera parar.	sin esto, el texto al azar corta en largos raros
--percentile-metrics · --metric-percentiles	Qué métricas y qué percentiles van al JSON. Defaults: ttft,tpot,itl y 99.	ttft,tpot,itl,e2el y 50,95,99
--goodput ttft:1000 tpot:50	Cuenta los requests que cumplen todos los umbrales (en ms), uno por uno.	request_goodput = 5.75 a c = 72
--save-result	Un JSON plano por corrida: request_throughput, output_throughput, request_goodput, mean_, median_, std_ y p<P>_ (uno por percentil pedido) de cada métrica, max_concurrency, request_rate…	sweep-c072.json
TTFT de vllm bench	Desde antes del POST hasta el primer chunk con choices (cuenta el del rol).	incluye red, cola y prefill
TPOT de vllm bench	(latencia – TTFT) ÷ (tokens – 1), por request; los de 1 token no entran. Los tokens salen de usage.	promedio por request
ITL de vllm bench	Todos los huecos entre chunks de todos los requests, en una sola lista.	su p99 muestra los tirones
vllm:time_to_first_token_seconds	Histograma: de que el request llega al motor hasta el primer token. No ve la cola del gateway ni la red.	llm:vllm_ttft_p95:rate5m, por pod
vllm:inter_token_latency_seconds	Histograma del ITL. Antes se llamaba vllm:time_per_output_token_seconds: obsoleta desde v0.11, oculta en v0.12 y borrada en v0.15.	bordes 0.01, 0.025, 0.05, 0.075… s
vllm:request_time_per_output_token_seconds	Histograma del TPOT de cada request: su tiempo de decode ÷ (tokens – 1).	la contraparte del SLO
vllm:e2e_request_latency_seconds · vllm:request_queue_time_seconds	El request completo y lo que esperó turno dentro de vLLM. También hay request_prefill_time_seconds y request_decode_time_seconds.	¿la espera está en vLLM o antes?
vllm:num_requests_running · vllm:num_requests_waiting	Gauges: el batch y la cola de la réplica, ahora.	KVCacheSaturation; absent() en LLMCapacityHeadroomLow
vllm:kv_cache_usage_perc	Gauge de 0 a 1: el KV cache ocupado. Antes vllm:gpu_cache_usage_perc (oculto en v0.11, borrado en v0.12).	> 0.95 con waiting > 0: KVCacheSaturation
vllm:num_preemptions_total · vllm:request_num_preemptions	Counter de preemptions (eventos) e histograma por request; su primer bucket es le="1" y junta 0 y 1.	LLMPreemptionsRising: eventos ÷ requests > 0.05
vllm:prompt_tokens_total · vllm:generation_tokens_total	Counters de tokens de entrada y de salida.	tokens/s de la réplica, con rate()
vllm:prefix_cache_queries_total · vllm:prefix_cache_hits_total	Counters en tokens: hits ÷ queries = la tasa de aciertos del prefix caching.	cae cuando alguien cambia un system prompt
vllm:request_success_total	Requests terminados, con el label finished_reason (stop, length, abort…).	el denominador de las preemptions por request

PIEZA	QUÉ HACE	EJEMPLO
`vllm:admission_rejections_total`	Nueva en v0.32: requests rechazados con 503 por un tope de cola de vLLM, con el label reason (max_num_queued_reqs o max_num_queued_tokens). En v0.30 los topes ya existen, pero esta métrica no.	el pin del C10 no la tiene
`la línea de estadísticas del log`	Cada 10 s: tokens/s de prompt y de generación, Running, Waiting, % de KV y aciertos del prefix cache; Preemptions: N solo si hubo.	en V1, el scheduler no escribe una línea por preemption

TTFT p95 (ms)



TPOT p95 (ms)



72 el codo: el último punto que cumple 80 TTFT y TPOT fuera

EN VUELO	REQ/S	GOODPUT	TOKENS/S	TTFT P95	TPOT P95
16	2.6	2.6	650	310 ms	27 ms
32	4.01	4.01	1,003	450 ms	35 ms
48	4.88	4.83	1,221	660 ms	42 ms
64	5.72	5.54	1,429	860 ms	47 ms
72	6.06	5.75	1,514	970 ms	49.6 ms
80	6.19	5.02	1,548	1,340 ms	53.8 ms
96	6.38	2.68	1,595	2,400 ms	64 ms

Se planea al 70 %: 6.06 × 0.7 = **4.24 req/s** por réplica.

Datos: `reference/c10/bench/tests/fixtures/sweep-c*.json (sintéticos)`

cumple su SLO fuera del SLO la zona que cumple el elegido: 72 en vuelo

pasado el codo, más concurrencia casi no da throughput (6.06 → 6.38 req/s) y sí da cola (TTFT p95 970 → 2,400 ms)

El barrido de una réplica con los fixtures del C10: SINTÉTICOS (números inventados con la forma típica y las claves reales de vLLM v0.30.0), no una medición. Hasta 72 en vuelo se cumplen TTFT p95 ≤ 1 s y TPOT p95 ≤ 50 ms; en 80 se rompen los dos, y en 96 hay 5 % más de requests/s que en 72 con 2.5× el TTFT p95.

EJEMPLO REAL · EL BARRIDO DEL C10: RUN_SWEEP.SH Y EL LECTOR QUE ELIGE LA CONCURRENCIA

BASH · ZSH reference/c10/bench/run_sweep.sh (líneas 18–49)

```
for c in 16 32 48 64 72 80 96; do  # alrededor del codo esperado (≈ 40 planeado × 1/0.7 ≈ 57, y más allá)
  vllm bench serve \
    --backend openai-chat \
    --endpoint /v1/chat/completions \
    --base-url "$BASE_URL" \
    --model "$MODEL" \
    --tokenizer "$TOKENIZER" \
    `# Mezcla sintética del caso: 3 productos (3 prefijos) × ≈ 1,000 tokens compartidos + ≈ 1,000 propios,` \
    `# 250 de salida. Los prompts se barajan (default). Mejor aún: reproducir prompts reales redactados.` \
    --dataset-name prefix_repetition \
    --prefix-repetition-num-prefixes 3 \
    --prefix-repetition-prefix-len 1000 \
    --prefix-repetition-suffix-len 1000 \
    --prefix-repetition-output-len 250 \
    `# ignore_eos: que cada request genere sus 250 tokens; sin esto, texto aleatorio corta en largos raros.` \
    --ignore-eos \
    --num-prompts "$NUM_PROMPTS" \
    `# Barrido CERRADO: hasta c requests en vuelo, el siguiente sale cuando termina uno.` \
    --max-concurrency "$c" \
    --request-rate inf \
```

```
    `# El default de --metric-percentiles es "99": sin esto no hay p95 en el JSON.` \
    --percentile-metrics ttft,tpot,itl,e2el \
    --metric-percentiles 50,95,99 \
    `# goodput: req/s que cumplen TTFT ≤ 1,000 ms Y TPOT ≤ 50 ms uno por uno (valores en ms).` \
    --goodput ttft:1000 tpot:50 \
    --save-result \
    --result-dir "$OUT" \
    --result-filename "sweep-c$(printf '%03d' "$c").json"
done

# Elegir: el último punto que cumple antes del primero que no, y la capacidad planeada al 70 %.
python "$(dirname "$0")/pick_concurrency.py" "$OUT"/sweep-c*.json --ttft-ms 1000 --tpot-ms 50 --percentile 95
```

- Contra UNA réplica (port-forward a un pod), no contra el gateway: se mide la unidad que escala. Y con la mezcla del caso, 3 prefijos de 1,000 tokens + 1,000 propios y 250 de salida: con otra mezcla, el número no sirve.
- Cada flag está en la página de `vllm bench serve` de v0.30.0. El script esquiva dos trampas: `--metric-percentiles` vale `99` por defecto (sin `50,95,99` no hay `p95_ttft_ms` y `pick_concurrency.py` se niega a adivinar) y, sin `--ignore-eos`, las respuestas cortan en largos al azar.
- `pick_concurrency.py` elige el último punto bueno antes del primero malo: si 80 falla y 96 «cumple», 96 es ruido (pocos requests, un warmup, otro proceso en la GPU) y lo avisa. Con los fixtures elige 72: 6.06 req/s, planeado al 70 % = 4.24. De ahí sale el ≈ 4.2 del plan de flota.
- Los fixtures (`bench/tests/fixtures/sweep-c*.json`) son SINTÉTICOS: claves reales de v0.30.0 (`request_throughput` , `output_throughput` , `request_goodput` , `p95_ttft_ms` …) con números inventados para que el codo caiga en ≈ 6 rps. Los 12 tests de `test_pick_concurrency.py` pasan.
- Un JSON real de v0.30.0 con este comando trae más claves que los fixtures: las de E2E (`p95_e2el_ms` y compañía, que el script sí pide), `max_output_tokens_per_s` , `max_concurrent_requests` , `rtfx` (0 fuera de los benchmarks de audio) y dos listas por request que quedan aunque no pidas `--save-detailed` : `latencies` y `queue_times` . Los percentiles de `client_queue_time` solo salen si los pides en `--percentile-metrics` . El lector solo lee las que usa.
- Las líneas 51–53 lo dicen: después se confirma con un barrido ABIERTO a la tasa planeada (`--request-rate 4.2` , sin `--max-concurrency`), porque el cerrado nunca deja que se forme la cola que en producción rompe el TTFT.

CÓMO RECORDARLA

TTFT es cuánto tardas en empezar a hablar; TPOT, a qué ritmo hablas; E2E, cuánto dura todo. Para el plan de capacidad, sube la concurrencia de a poco hasta que el primer SLO se rompa: el punto anterior es la capacidad (el codo), y planeas al 70 % de eso. En producción, con los histogramas: `rate` , `sum by (le)` y recién ahí el cuantil.

TRAMPA

creerle al p99 de un barrido cerrado. Cuando la réplica se frena, el cliente también deja de mandar, y la cola que en producción rompe el TTFT nunca se forma (coordinated omission); además, lo que un request espera turno en el cliente no entra en su TTFT. Por eso el C10 confirma con un barrido abierto a 4.2 req/s. La otra: el TPOT p95 puede estar en 49.6 ms con el ITL p99 en 109 ms (c = 72, en los fixtures). El promedio por request esconde los tirones, y una alerta que solo mira el TPOT no los ve.

En el curso: M14 · Prefill y decode: TTFT, TPOT, throughput y goodput · M3 · Load testing: cómo se mide de verdad · M3 · p50, p95, p99: por qué el promedio miente · M12 · Prometheus, Grafana y alertas por SLO · C10 · Monitoreo, alertas y reentrenamiento · C10 · Archivos de referencia · Hoja · Prometheus y Grafana · histogramas y percentiles

8 Paralelismo: tensor, pipeline y réplicas; NVLink contra PCIe; cuándo un 70B necesita TP

¿Cuándo hay que partir un modelo entre GPUs, de qué formas se parte y por qué el C10 usa TP=2 dentro de un nodo, y no una GPU por réplica ni un TP más grande?

SINTAXIS CON VARIABLES

`vllm serve` *modelo* `--tensor-parallel-size` *tp*

Tensor parallelism (TP): cada capa se parte entre *tp* GPUs; cada una tiene un pedazo de cada matriz y, al final de la atención y del MLP (las dos partes de cada capa), juntan resultados con un **all-reduce** (una suma entre todas las GPUs). Son dos por capa: con 80 capas, 160 por paso. Necesita una conexión muy rápida entre GPUs: **NVLink**, el enlace directo entre las GPUs de un mismo servidor.

`vllm serve` *modelo* `--tensor-parallel-size` *gpus_por_nodo* `--pipeline-parallel-size` *nodos* \ `--distributed-executor-backend ray`

Pipeline parallelism (PP): grupos de capas en GPUs distintas (etapas); entre etapas viajan las activaciones de cada token (en vLLM, dos vectores: hidden_states y residual), una vez por paso. Para un modelo que no cabe en un nodo, la regla de la documentación es TP = GPUs por nodo y PP = número de nodos. También sirve dentro de un nodo sin NVLink.

`vllm serve` *modelo* `--data-parallel-size` *dp* `--tensor-parallel-size` *tp* `# dp × tp GPUs`

Data parallelism (DP): copias completas e independientes, cada una con su KV cache. Con `--data-parallel-size`, un solo `vllm serve` reparte los requests entre las copias; para un modelo denso es lo mismo que N servidores separados. El C10 usa réplicas de Kubernetes: 16 pods de TP=2.

`resources: {limits: {nvidia.com/gpu: tp × pp }}` `# + nodeSelector a un nodo con NVLink`

Un pod pide TP × PP GPUs, y todas tienen que estar en el mismo nodo. En el C10: 2, en un p5.48xlarge (8 H100 unidas por NVSwitch: cualquier par sirve).

`nvidia-smi topo -m` `# NV# = NVLink; PIX, PXB, PHB, NODE, SYS = caminos por PCIe`
`NCCL_DEBUG=INFO vllm serve …` `# por dónde va cada canal: P2P, SHM o NET`

Antes de culpar al modelo, mira el cable. En H100 SXM con NVSwitch, `topo -m` muestra `NV18` entre cada par (así sale en un DGX H100). Con `NCCL_DEBUG=INFO`, NCCL (la librería de NVIDIA que comunica GPUs) dice qué transporte usa: P2P es directo entre GPUs (por NVLink o PCIe), SHM pasa por la memoria de la CPU (el host) y `NET/Socket` es TCP, lento para TP entre nodos.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
tp	GPUs que se reparten cada capa	`2`
gpus_por_nodo · *nodos*	para un modelo que no cabe en un nodo: TP y PP	`no aplica al C10`
dp	copias completas dentro de un vllm serve	`el C10 usa 16 pods en vez de --data-parallel-size`
TP, PP, DP	tensor, pipeline y data parallelism	`2 · 1 · 16 réplicas`
n_{tokens}	tokens del paso	`72 decodes, o un paso de 8,192`
8, 192, 2 B	hidden size del modelo (el largo del vector de cada token) · bytes por valor en BF16 (las activaciones)	`config.json de Llama 3.3 70B`
450 GB/s, 64 GB/s	ancho de banda por sentido: el NVLink de la H100 SXM (900 sumando los dos) y PCIe Gen5 x16 (128)	`hoja de datos de NVIDIA`
2.5 GB	runtime y activaciones por GPU	`≈ 5 GB por réplica (estimación del C10)`

FÓRMULAS

$$\text{pesos por GPU} = \frac{70.6\ \text{GB}}{\text{TP}\times\text{PP}} \qquad \text{TP} = 2\text{: } 35.3\ \text{GB}$$

$$1\ \text{GPU: } 0.90\times 80 - 70.6 - 2.5 = -1.1\ \text{GB} \Rightarrow \text{no cabe} \qquad \text{TP} = 2\text{: } 0.90\times 160 - 70.6 - 5 = 68.4\ \text{GB de KV}$$

$$\text{KV heads por GPU} = \max\left(1, \left\lfloor \frac{8}{\text{TP}} \right\rfloor\right) \qquad 64 \bmod \text{TP} = 0 \Rightarrow \text{TP} \in \{1, 2, 4, 8\}\ \text{en un nodo de 8}$$

$$\text{all-reduce por paso (TP=2)} = 2\times 80\times n_{\text{tokens}}\times 8{,}192\times 2\ \text{B} \qquad n = 72\text{: } 189\ \text{MB} \qquad n = 8{,}192\text{: } 21.5\ \text{GB}$$

$$t \approx \frac{\text{bytes}}{\text{ancho de banda}} \qquad 189\ \text{MB: } 0.42\ \text{ms por NVLink (450 GB/s)contra2.9 ms por PCIe Gen5 (64 GB/s)}$$

$$\text{GPUs} = \text{DP}\times\text{TP}\times\text{PP} \qquad 16\times 2\times 1 = 32 = 4\ \text{nodos de 8}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`--tensor-parallel-size (-tp)`	Parte cada capa entre GPUs. Default 1.	`2`
`--pipeline-parallel-size (-pp)`	Parte el modelo por capas, en etapas. Default 1.	`1`
`--data-parallel-size (-dp)`	Copias del modelo dentro de un mismo servidor. Default 1.	`1 por pod; 16 pods`
`regla de TP`	Las heads de atención (64) tienen que dividirse por TP; si no: «Total number of attention heads (64) must be divisible by tensor parallel size (3).»	`TP ∈ {1, 2, 4, 8} en un nodo`

PIEZA	QUÉ HACE	EJEMPLO
KV heads por GPU	Con TP, cada GPU guarda max(1, 8 ÷ TP) KV heads: con TP=2, 4 de 8, o sea 160 KiB por token por GPU. Con más TP que KV heads, se replican.	`320 KiB por token entre las dos`
all-reduce	La suma entre GPUs al final de la atención y del MLP de cada capa: 160 por paso. Cada paso espera a la GPU más lenta.	`lo hace NCCL o, con mensajes chicos, un kernel más rápido (de vLLM, FlashInfer o PyTorch)`
--disable-custom-all-reduce	Apaga el kernel de all-reduce propio de vLLM. Default False. En v0.30 quedan los de FlashInfer y PyTorch: para que todo vaya por NCCL, también VLLM_ALLREDUCE_USE_FLASHINFER=0 y VLLM_ALLREDUCE_USE_SYMM_MEM=0.	`útil para descartar un bug`
NCCL	La librería de NVIDIA para comunicar GPUs (all-reduce y demás). Dentro del nodo usa NVLink si lo hay y crea segmentos en /dev/shm.	`sin espacio: failed to extend /dev/shm/nccl-…`
NVLink · NVSwitch	Conexión directa entre GPUs: 900 GB/s en la H100 SXM sumando los dos sentidos, ≈ 7× PCIe Gen5. NVSwitch une las 8 GPUs de un nodo.	`p5.48xlarge: cualquier par sirve`
PCIe	El bus de la tarjeta: 128 GB/s en Gen5 x16, sumando sentidos. Con TP=2, el all-reduce de un paso de decode le suma ≈ 3 ms a un piso de ≈ 10.5; el de un prefill de 8,192, ≈ 0.34 s.	`GPUs sin NVLink (L40S): la doc sugiere PP`
/dev/shm	Memoria compartida: NCCL crea ahí sus segmentos y vLLM la usa para repartir cada paso entre los procesos de la réplica.	`emptyDir Memory de 16Gi`
--distributed-executor-backend	mp (procesos de Python) o ray. Por defecto, mp en un nodo; para varios nodos la doc usa ray, o mp con --nnodes y --node-rank.	`mp: todo dentro de un pod`
PP y las burbujas	Las etapas van en serie para cada token; los ratos en que una etapa espera vacía son las burbujas. vLLM las tapa con tantos batches en vuelo como etapas.	`piso del token ≈ 2 × 10.5 ms con PP=2`
TP entre nodos	Cada paso cruza la red 160 veces: por NET/Socket (TCP) es muy lento. TP dentro del nodo, PP entre nodos.	`ninguna réplica del C10 cruza nodos`
expert parallelism (EP)	Para modelos MoE (mixture of experts: muchos expertos y pocos activos por token): reparte los expertos entre GPUs (--enable-expert-parallel). No aplica a un modelo denso.	`gpt-oss, si algún día lo eligen`
H200 (141 GB)	Con más memoria por GPU, el 70B en FP8 cabe en una con ≈ 54 GB de KV: TP=1.	`otra GPU, otra cuenta`

p5.48xlarge: 8 × H100 SXM 80 GB · NVSwitch: 900 GB/s por GPU sumando sentidos (≈ 450 por sentido)

TP=2 (el C10) — cabe



cada capa partida en 2: 32 de las 64 heads de query y 4 de las 8 KV heads por GPU

CADA GPU, A ESCALA (DE 0 A 80 GB)



KV por réplica: **68.4 GB** · **92** secuencias de 2,250 tokens

COMUNICACIÓN: 2 all-reduce por capa: 160 por paso

- Un paso de decode con 72 tokens: **188.7 MB** entre GPUs → **0.42 ms** por NVLink contra **2.9 ms** por PCIe Gen5; el piso del paso es 10.5 ms
- Un paso de prefill con 8,192 tokens: **21,474.8 MB** entre GPUs → **47.7 ms** por NVLink contra **335.5 ms** por PCIe Gen5; el piso del paso es 292.2 ms

elegido: el TP más chico donde caben los pesos y el KV

PP=2 — cabe



GPU 1: capas 1–40 · GPU 2: capas 41–80

CADA GPU, A ESCALA (DE 0 A 80 GB)



KV por réplica: **68.4 GB** · **92** secuencias de 2,250 tokens

COMUNICACIÓN: un envío por paso entre etapas (hidden_states y residual de cada token): 2.36 MB con 72 tokens

LATENCIA: las etapas van en serie: piso del token ≈ 2 × 10.5 = 21 ms, contra 10.5 con TP=2

para nodos sin NVLink o modelos que no caben en un nodo

TP=1 (réplica en 1 GPU) — no cabe: −1.1 GB



8 × 1 GPU

el modelo entero en una GPU

CADA GPU, A ESCALA (DE 0 A 80 GB)



se pasa por **1.1 GB** del tope (72 GB por GPU): sin lugar para el KV

COMUNICACIÓN: ninguna

no cabe: 0.90 × 80 − 70.6 − 2.5 = −1.1 GB (en INT4 o en una H200, sí)

CON MÁS TP, EN EL MISMO NODO

TP	KV POR RÉPLICA	SECUENCIAS DE 2,250	RÉPLICAS POR NODO
TP=4	**207.4 GB**	**281**	**2**

TP	KV POR RÉPLICA	SECUENCIAS DE 2,250	RÉPLICAS POR NODO
TP=8	485.4 GB	658	1

DP × TP × PP = 16 × 2 × 1 = 32 GPUs = 4 nodos; las réplicas son pods de Kubernetes

pesos FP8 · runtime y activaciones (≈ 2.5 GB, est.) · KV cache · lo que vLLM no toca: (1 − 0.90) × 80 GB · lo que no cabe · el tope de vLLM

Tres formas de repartir el 70B en FP8 en un p5 de 8 H100. TP=2 es la del C10: cada GPU tiene la mitad de cada capa y las dos se sincronizan 160 veces por paso por NVLink. PP=2 ocupa la misma memoria, pero las etapas van en serie. Una copia entera no entra en una H100 con 0.90 de utilización. Bytes y tiempos: la cuenta de tp.py, del lab (cotas: solo ancho de banda).

EJEMPLO REAL · EL PARALELISMO DEL C10 EN EL MANIFEST: UN NODO CON NVLINK, TP=2, DOS GPUS POR POD Y /DEV/SHM

YAML reference/c10/k8s/vllm-70b.yaml (líneas 97–102, 185–186, 225–233, 272–273 y 281–284; # … marca líneas omitidas)

```yaml
      nodeSelector:
        node.kubernetes.io/instance-type: p5.48xlarge  # 8 × H100 SXM con NVLink: TP=2 dentro del nodo
      tolerations:
        - key: nvidia.com/gpu  # los nodos GPU tienen taint: nada que no use GPU ocupa una máquina de ≈ US$55/h
          operator: Exists
          effect: NoSchedule
# …
            - --tensor-parallel-size
            - "2"  # 70.6 GB de pesos FP8 + KV no entran cómodos en 80 GB; en 2 GPUs quedan ≈ 68 GB para KV
# …
          resources:
            requests:
              cpu: "40"  # p5.48xlarge: 192 vCPU para 4 réplicas; tokenización, scheduler y el API server son CPU
              memory: 256Gi  # 2 TiB en el nodo; page cache de los safetensors (mmap) + /dev/shm + buffers
              ephemeral-storage: 200Gi  # el emptyDir de los pesos cuenta aquí
              nvidia.com/gpu: 2  # TP=2: dos H100 del mismo nodo
            limits:
              memory: 256Gi  # = request: pasarse es un OOMKilled claro
              nvidia.com/gpu: 2  # las GPUs no se sobreasignan
# …
            - name: shm
              mountPath: /dev/shm  # NCCL y los procesos de TP comparten memoria por aquí
# …
        - name: shm
          emptyDir:
            medium: Memory  # el /dev/shm por defecto de un contenedor es 64 MiB: TP con NCCL necesita más
            sizeLimit: 16Gi  # cuenta contra el límite de memoria del contenedor
```

- `nodeSelector` fija el tipo de nodo: un p5.48xlarge trae 8 H100 SXM unidas por NVSwitch (900 GB/s por GPU, sumando sentidos), así que cualquier par sirve para TP=2 y ninguna réplica cruza nodos. El taint `nvidia.com/gpu` (una marca del nodo que rechaza a los pods sin la toleration) deja esas máquinas de ≈ US$55/h solo para pods de GPU.
- `--tensor-parallel-size "2"` : con 0.90 de utilización, una H100 deja 72 GB para vLLM y los pesos FP8 ocupan 70.6: no queda lugar para el KV. En dos, quedan 68.4 GB de KV (≈ 92 secuencias de 2,250 tokens). Es el TP más chico que alcanza; después se agregan réplicas.
- `nvidia.com/gpu: 2` en requests y limits = TP × PP. El device plugin (el agente de NVIDIA que le presenta las GPUs a Kubernetes) entrega dos GPUs enteras del mismo nodo, que no se comparten con otros pods.
- `/dev/shm` en RAM, de 16Gi: NCCL crea ahí sus segmentos de memoria compartida y vLLM reparte cada paso entre sus procesos por ahí. Con los 64 MiB por defecto, NCCL puede fallar con `failed to extend /dev/shm/nccl-…` .
- Por qué no TP=4 u 8: más TP baja el TPOT (el tiempo por token: cada GPU lee menos pesos por paso) y da más KV por réplica (207 y 485 GB), pero el all-reduce de cada capa no se achica (pesa más en cada paso) y deja menos réplicas: con TP=8, una réplica caída es 1/4 de la flota. El C10 se queda con TP=2 mientras el barrido no muestre una diferencia que pague eso.
- Por qué no DP dentro de un pod (`--data-parallel-size 4` con las 8 GPUs): para un modelo denso da lo mismo que 4 réplicas, pero un pod caído se lleva las 4 y un rolling update baja de a 4. Con pods de 2 GPUs, todo pasa de a 1/16.

CÓMO RECORDARLA

el TP más chico donde caben los pesos y el KV; después, réplicas. TP es partir cada página de un libro entre dos lectores que leen a la vez y se pasan notas en cada renglón: tienen que estar sentados juntos (NVLink). PP es darle a cada lector la mitad de los capítulos: se pasan la posta una vez por paso, pero uno espera al otro. Las réplicas son libros completos, uno por mesa.

TRAMPA

partir con TP entre dos nodos «porque sobran GPUs en el otro». Cada paso hace 160 all-reduce; por la red (`NET/Socket` en el log de NCCL es TCP) cada uno cuesta mucho más que por NVLink, y el TPOT se multiplica. Entre nodos va PP; dentro del nodo, TP. La otra: TP=3 o TP=6 para «usar las GPUs que quedan». 64 heads no se dividen por 3 ni por 6, y vLLM no arranca: `Total number of attention heads (64) must be divisible by tensor parallel size (3).`

En el curso: M14 · Batching, caching, speculative decoding y paralelismo · C10 · Serving · M4 · Cálculo de VRAM paso a paso · M4 · T4, L4, A10G, A100, H100: cuándo usar cada una · M6 · GPUs en Kubernetes · Números · GPUs: memoria, ancho de banda, cómputo y precio

9 Cuantización: FP8, INT8, INT4 (AWQ y GPTQ), KV cache en FP8 y la calidad

¿Cuántos bytes por parámetro necesita el 70B del C10, qué formato corre en qué GPU, cómo se lo dices a vLLM y cómo sabes que la calidad no se rompió?

SINTAXIS CON VARIABLES

`vllm serve` *checkpoint* `--served-model-name` *nombre* `--tensor-parallel-size` *tp*

Cuantizar es guardar los pesos con menos bits: cada número se vuelve un número chico (8 o 4 bits) multiplicado por una **escala** que comparten muchos pesos. Un checkpoint (la carpeta con los pesos y su config) ya cuantizado (**pre-quantized**) trae su receta en `config.json`, bajo `quantization_config`, y vLLM la lee sola: no pasas `--quantization`. Si lo pasas y no coincide con la del checkpoint, no arranca. Así sirve el C10 sus pesos FP8.

`python3 -c "import json; print(json.load(open('`*carpeta*`/config.json')).get('quantization_config'))"`

Lee la receta antes de servir: `quant_method` (`compressed-tensors`, `fp8`, `awq`, `gptq` …), los bits y el tipo, si las activaciones se cuantizan al vuelo (`dynamic: true`), qué capas quedaron en 16 bits (`ignore`, casi siempre `lm_head`, la última capa, la que elige el token) y si trae escalas para el KV (`kv_cache_scheme`; `null` = no trae).

`vllm serve` *checkpoint_bf16* `--quantization fp8`

Cuantización **online** (al cargar): vLLM baja un checkpoint BF16 a FP8 sin datos de calibración, con una escala por tensor. Sirve para probar rápido; la doc actual avisa que así la latencia mejora poco. Para producción, un checkpoint hecho con llm-compressor (la librería de vLLM para cuantizar; escalas por canal en los pesos y por token en las activaciones). Desde v0.31, `fp8` redirige a `fp8_per_tensor`, el nombre que usa la doc actual.

`vllm serve` *checkpoint* `--kv-cache-dtype` *tipo_kv*

El **KV cache** (lo que el modelo ya leyó de cada conversación, guardado en la GPU para no recalcularlo) pasa de 2 bytes a 1 por valor: caben ≈ el doble de tokens. `auto`, el default y el del C10, usa el dtype del modelo (BF16). `fp8` es `fp8_e4m3`. Si un checkpoint cuantizado, como el del C10, no trae escalas para K y V, vLLM usa 1.0 y lo avisa en el log; con uno sin cuantizar usa 1.0 sin avisar.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
checkpoint	carpeta o repo con los pesos ya cuantizados y su config.json	`/models/chat-70b-fp8-2026-09-28`
nombre	el nombre del modelo que ven los clientes	`chat-70b-fp8-2026-09-28`
tp	entre cuántas GPUs se reparte cada capa	`2`
carpeta	dónde quedaron los safetensors	`/models/chat-70b-fp8-2026-09-28`
checkpoint_bf16	los pesos originales, en 16 bits	`meta-llama/Llama-3.3-70B-Instruct (≈ 141 GB)`
tipo_kv	en qué tipo se guarda el KV cache	`auto en el C10; fp8 solo si pasa el golden set`
P	parámetros del modelo	`70.6 B`
b	bytes por parámetro del formato	`2 en BF16 · 1 en FP8 o INT8 · 0.5 en INT4`
KV libre	memoria para KV en una réplica: GPUs × 80 GB × gpu_memory_utilization − pesos − runtime (≈ 2.5 GB por GPU, estimado)	`68.4 GB con pesos FP8`
s	la escala que comparten muchos pesos	`max|x| ÷ 448 en FP8 E4M3`
q	el número chico que se guarda en lugar de x	`un entero de −127 a 127, o un float de 8 bits`

FÓRMULAS

$$\text{pesos} = P \times b \qquad 70.6\ \text{B} \times \{2,\ 1,\ 0.5\}\ \text{B} = 141.2,\ 70.6,\ 35.3\ \text{GB}$$

$$\text{KV libre} = 2 \times 80 \times 0.90 - \text{pesos} - 5 \Rightarrow \text{BF16: } -2.2 \quad \text{FP8: } 68.4 \quad \text{INT4: } 103.7\ \text{GB}$$

$$\text{secuencias} = \left\lfloor \frac{68.4\ \text{GB}}{2{,}250 \times 320\ \text{KiB}} \right\rfloor = 92 \qquad \text{con KV en FP8 (160 KiB): } 185$$

$$x \approx s \cdot q \qquad s_{\text{INT8}} = \frac{\max|x|}{127} \qquad s_{\text{FP8 E4M3}} = \frac{\max|x|}{448}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
BF16	16 bits con el rango de FP32. Es el formato en que se entrenan y se publican los LLMs, y la referencia de calidad.	`141.2 GB de pesos: en 2 × H100 casi no deja KV`
FP8 W8A8	Pesos y activaciones en 8 bits de punto flotante (W8A8: weights y activations en 8 bits). La mitad de memoria que BF16 y tensor cores (las unidades que multiplican matrices) FP8 para el cómputo.	`los pesos del C10: ≈ 70.6 GB`
E4M3 · E5M2	Los dos FP8: 4 bits de exponente y 3 de mantisa (más precisión, máximo 448) o 5 y 2 (más rango). Como es punto flotante, un valor atípico casi no le quita resolución a los chicos.	`--kv-cache-dtype fp8 = fp8_e4m3`
FP8 weight-only (W8A16)	Solo los pesos en FP8; el cómputo sigue en 16 bits. vLLM lo usa en GPUs sin FP8 nativo (Turing y Ampere) con los kernels Marlin: ahorra memoria, no cómputo.	`el mismo checkpoint en una A100`
INT8 W8A8	Enteros de 8 bits con escala, en pesos y activaciones (hechos con llm-compressor). Corre desde Turing.	`una salida para A100 y A10G, que no tienen FP8`
INT4 W4A16 (AWQ, GPTQ)	Pesos de 4 bits con una escala por grupo de pesos (por ejemplo, cada 128); se descuantizan al vuelo y el cómputo va en 16 bits. AWQ y GPTQ son dos formas de elegir esos 4 bits perdiendo poco.	`el 70B en ≈ 39.5 GB: cabe en una sola H100`

PIEZA	QUÉ HACE	EJEMPLO
GPTQ con act-order (desc_act)	Una variante de GPTQ que cuantiza las columnas en otro orden y guarda un índice (g_idx) para deshacerlo. vLLM v0.30.0 dejó de soportarla: un checkpoint así ni arranca (GPTQ group activation ordering (desc_act=True) is no longer supported).	usa GPTQ sin act-order de grupo (llm-compressor usa static por defecto) o AWQ
Marlin	Kernels rápidos (los programas que corren en la GPU) para pesos de 4 u 8 bits con activaciones de 16: GPTQ, AWQ y FP8 weight-only. Con FP8 en una GPU sin soporte nativo, vLLM los elige solo.	la A100 con el checkpoint FP8 del C10
quantization_config	La receta dentro del config.json: método, bits, grupos y capas que no se tocaron. vLLM la lee y elige el método.	quant_method: compressed-tensors · ignore: [lm_head]
--quantization (-q)	Fuerza un método. Con un checkpoint ya cuantizado no hace falta; con uno BF16 y fp8, cuantiza al cargar.	el C10 no lo pasa
--kv-cache-dtype	El tipo del KV cache, aparte de los pesos: auto (el del modelo), fp8 (= fp8_e4m3), fp8_e5m2 y otros más nuevos.	auto en el C10: KV en BF16
k_scale · v_scale	Las escalas del KV en FP8 que la calibración guarda en el checkpoint. Sin ellas, vLLM usa 1.0 (y lo avisa solo si el checkpoint está cuantizado).	kv_cache_scheme: null en un checkpoint FP8-dynamic
Calibración (PTQ)	Pasar unos cientos de ejemplos reales para medir rangos y fijar escalas, después del entrenamiento (post-training quantization).	FP8_DYNAMIC no la necesita; INT4 y el KV sí
Compute capability	La generación de la GPU como número: 7.5 Turing (T4), 8.0 Ampere (A100), 8.9 Ada (L4, L40S), 9.0 Hopper (H100). FP8 nativo desde 8.9.	H100 = 9.0
Golden set y gate	El examen fijo de cada producto y la regla que decide si una revisión pasa. Ninguna cuantización entra sin él.	eval/golden_eval.py del C10

El 70B del C10 (arquitectura de Llama 3.x 70B: 70.6 B de parámetros, 80 capas, 8 KV heads × 128). Una réplica: 2 × 80 GB × 0.9 = **144 GB** para vLLM, con ≈ 2.5 GB de runtime por GPU. KV por token: 320 KiB en BF16, 160 KiB en FP8.

CADA FORMATO, A ESCALA (TODAS LAS BARRAS VAN DE 0 A 150 GB)

BF16 2 B por parámetro

pesos **141.2 GB** · checkpoint real 141.1 GB

2 × H100 (TOPE 144 GB)



no cabe: −2.2 GB sin lugar para el KV

1 × H100 (TOPE 72 GB)

no aplica: los pesos solos pasan la memoria de la GPU

0 30 60 90 120 150 GB

no cabe con KV útil en 2 × H100: la cuenta nominal da −2.2 GB y, con la memoria que reporta la GPU (≈ 85.5 GB), quedan ≈ 7.7 GB (≈ 10 secuencias). Pide TP=4

FP8 (W8A8) 1 B por parámetro

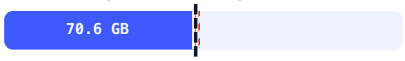
pesos **70.6 GB** · checkpoint real 72.7 GB

2 × H100 (TOPE 144 GB)



KV libre **68.4 GB** → **92** secuencias de 2,250 tokens con KV en BF16, **185** con KV en FP8

1 × H100 (TOPE 72 GB)



0 30 60 90 120 150 GB

no cabe: −1.1 GB

el C10; en 1 H100 no deja KV útil (nominal −1.1 GB; con la memoria reportada, ≈ 2–4 GB)

INT4 (W4A16) 0.5 B por parámetro

pesos **35.3 GB** · checkpoint real 39.5 GB

2 × H100 (TOPE 144 GB)



KV libre **103.7 GB** → **140** secuencias de 2,250 tokens con KV en BF16, **281** con KV en FP8

1 × H100 (TOPE 72 GB)



0 30 60 90 120 150 GB

KV libre **34.2 GB**

cabe en 1 H100; más riesgo de calidad

LOS CHECKPOINTS REALES

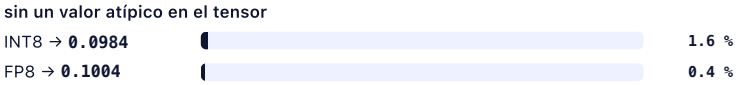
`meta-llama/Llama-3.3-70B-Instruct`
BF16 · 70,553,706,496 parámetros · **141.1 GB**

`RedHatAI/Llama-3.3-70B-Instruct-FP8-dynamic`
68.45 B en FP8 E4M3 + 2.11 B en BF16 · **72.7 GB**

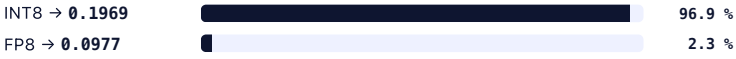
`RedHatAI/Llama-3.3-70B-Instruct-quantized.w4a16`
INT4 con escalas por grupo + embeddings en BF16 · **39.5 GB**

REDONDEAR 0.1 A 8 BITS: LO QUE SE GUARDA Y SU ERROR RELATIVO (A ESCALA, DE 0 A 100 %)

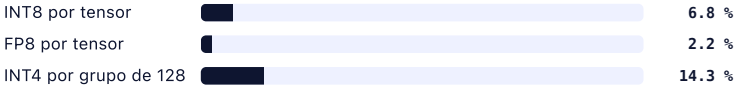
sin un valor atípico en el tensor



con un atípico de 25 en el mismo tensor



mediana del error en 4,096 pesos con un atípico



pesos · runtime: activaciones, CUDA graphs y buffers · KV libre · lo que no cabe · el tope de vLLM

Bytes por parámetro y memoria del 70B del C10 en una réplica de 2 × H100 (0.90 × 160 GB = 144 GB) y en una sola H100: pesos, KV libre y secuencias de 2,250 tokens que caben. Los pesos son la cuenta del caso (70.6 B × bytes); los checkpoints reales en FP8 e INT4 pesan algo más porque dejan embeddings y lm_head en BF16 (el BF16 real pesa algo menos: 70.55 B de parámetros). Números del lab de esta sección.

EJEMPLO REAL · C10: EL GATE QUE DECIDE SI UNA CUANTIZACIÓN (O EL KV EN FP8) ENTRA

PYTHON reference/c10/eval/golden_eval.py (líneas 218–228, 238–245 y 251–261; las líneas # … resumen lo omitido)

```python
@dataclass(frozen=True)
class GateThresholds:
    """Umbrales ILUSTRATIVOS: los fija cada producto con su dueño. Se versionan con el código."""

    min_json_validity: float = 0.99  # extracción: 1 % de JSON roto ya es 20k items por noche (de 2 M)
    min_field_f1: float = 0.90  # extracción: piso absoluto
    max_f1_drop: float = 0.01  # extracción: contra la revisión actual (si se da baseline)
    min_win_rate_lower: float = 0.45  # chat: no-inferioridad con margen de 5 puntos (borde inferior de Wilson)
    min_position_consistency: float = 0.80  # chat: debajo de esto el juez decide por posición, no por calidad
    min_judge_kappa: float = 0.60  # chat: acuerdo juez-humano medido en la muestra de calibración
    min_pairs: int = 300  # chat: con menos, el intervalo es demasiado ancho para decidir


def gate_extraction(rep: ExtractionReport, t: GateThresholds, baseline_f1: float | None = None) -> GateResult:
    reasons = []
    if rep.json_validity < t.min_json_validity:
        reasons.append(f"JSON válido {rep.json_validity:.3f} < {t.min_json_validity}")
    if rep.f1 < t.min_field_f1:
        reasons.append(f"F1 por campo {rep.f1:.3f} < {t.min_field_f1}")
    if baseline_f1 is not None and baseline_f1 - rep.f1 > t.max_f1_drop:
        reasons.append(f"F1 cae {baseline_f1 - rep.f1:.3f} contra la revisión actual (> {t.max_f1_drop})")
    # … return GateResult(not reasons, reasons, {…})


def gate_pairwise(rep: PairwiseReport, t: GateThresholds, judge_kappa: float) -> GateResult:
    lo, hi = wilson_interval(rep.wins + 0.5 * rep.ties, rep.pairs)
    reasons = []
    if rep.pairs < t.min_pairs:
        reasons.append(f"solo {rep.pairs} pares (< {t.min_pairs})")
    if judge_kappa < t.min_judge_kappa:
        reasons.append(f"kappa juez-humano {judge_kappa:.2f} < {t.min_judge_kappa}: recalibra el juez antes de decidir")
    if rep.position_consistency < t.min_position_consistency:
        reasons.append(f"consistencia de posición {rep.position_consistency:.2f} < {t.min_position_consistency}")
    if lo < t.min_win_rate_lower:
        reasons.append(f"win rate {rep.win_rate:.3f}, IC95 [{lo:.3f}, {hi:.3f}]: el borde inferior no pasa {t.min_win_rate_lower}")
    # … return GateResult(not reasons, reasons, {…})
```

- En el C10 los pesos FP8 son una revisión del modelo más: `chat-70b-fp8-2026-09-28`. Pasa el mismo gate que un modelo distinto, por producto, y el reporte va al model registry. Para comparar FP8 contra el original, el caso corre la versión BF16 en una réplica TP=4 temporal: en 2 H100 no cabe con KV.
- Chat: `gate_pairwise` compara contra la revisión actual con un juez calibrado. El borde inferior del intervalo de Wilson tiene que pasar 0.45: con 1,000 pares, un win rate de 0.50 da [0.469, 0.531] y pasa; con 100, [0.404, 0.596] y no pasa (es la salida de `python eval/golden_eval.py`).
- Extracción: JSON válido ≥ 0.99, F1 por campo ≥ 0.90 y no más de 1 punto debajo de la revisión actual. Con 2 M items por noche, 1 % de JSON roto son 20,000 items.
- El KV en FP8 espera este gate. El manifest lo dice al lado del flag: `auto # KV en BF16. "fp8" duplica las secuencias; se activa solo si pasa el golden set`. Córrelo también con los prompts largos del set: M14 avisa que el riesgo del KV en FP8 crece con el contexto.
- Lo que el gate no mide es la velocidad. Una revisión cuantizada vuelve al barrido de `vllm bench serve` (`bench/run_sweep.sh`), porque el «≈ 6 rps por réplica» cambia con el formato.

CÓMO RECORDARLA

2, 1, ½: bytes por parámetro en BF16, FP8 e INT4, o sea **141, 71 y 35 GB** para el 70B. FP8 nativo desde la generación **8.9** (Ada, Hopper, Blackwell); en Turing y Ampere vLLM lo corre solo como pesos (W8A16). Y el orden del C10: **pesos en FP8 primero; KV en FP8 solo si pasa el golden set**.

TRAMPA

tratar la cuantización como un cambio de memoria y no de modelo. Cambia los números que el modelo calcula: se mide con el golden set de cada producto, en sus casos difíciles (prompts largos, español de México, JSON), no con un benchmark público ni con el promedio. Y dos de configuración: pasar `--quantization` a un checkpoint que ya trae la suya (si no coincide, vLLM no arranca), y activar `--kv-cache-dtype fp8` sin escalas calibradas: corre con 1.0 y, como mucho, te avisa en el log.

En el curso: M8 · Cuantización: FP16, INT8, FP8 e INT4 · M4 · Bytes por dtype y cuánto pesa un modelo · M4 · T4, L4, A10G, A100, H100: cuándo usar cada una · M14 · KV cache: la memoria que decide la concurrencia · C10 · Serving · C10 · Trade-offs · C10 · Evaluación offline y online · Hoja · De PyTorch a Triton · precisión en TensorRT

10 Optimizaciones: prefix caching, speculative decoding, LoRA múltiple, autoscaling por cola y cold start

Ya cabe y ya cumple: ¿qué palanca baja el TTFT, cuál el TPOT, cuál sirve varios fine-tunings con una sola flota, cuál acorta el arranque, y por qué el C10 no escala con KEDA?

SINTAXIS CON VARIABLES

```
sum(rate(vllm:prefix_cache_hits_total[ventana])) /
sum(rate(vllm:prefix_cache_queries_total[ventana]))
```

Prefix caching (reusar el KV de un comienzo de prompt que ya se calculó) viene activado en v0.30 (`--enable-prefix-caching`; lo apaga `--no-enable-prefix-caching`). Guarda bloques completos de 16 tokens, cada uno identificado por un hash (una huella numérica) de todo lo anterior: solo acierta lo idéntico desde el primer token. Ahorra prefill (TTFT y GPU), no decode. Esta query da la fracción de tokens de prompt que salieron del caché.

```
vllm serve modelo --speculative-config '{"method":
"ngram", "num_speculative_tokens": k, "prompt_lookup_max":
n_max}'
vllm serve modelo --speculative-config '{"method":
"metodo", "model": "draft", "num_speculative_tokens":
k}'
```

Speculative decoding (algo barato propone k tokens y el modelo grande los verifica todos en una sola pasada). `method` dice de dónde salen las propuestas: `ngram` (copia trozos del prompt), `draft_model` (un modelo chico con el mismo tokenizer), `eagle3`, `mtp` o `suffix`. La doc de vLLM lo recomienda para bajar el tiempo entre tokens con QPS medio o bajo, cuando el decode es **memory-bound** (lo limita leer los pesos, no calcular). En teoría la salida es la misma: cambia la velocidad, no la calidad.

```
vllm serve modelo_base --enable-lora --max-loras n
--max-lora-rank r \
  --lora-modules nombre=ruta
# el cliente lo pide como cualquier modelo: "model":
"nombre"
```

LoRA (un adaptador chico entrenado encima del modelo base): un solo vLLM sirve el base y varios adaptadores. `--max-loras` = cuántos adaptadores distintos entran en un mismo batch (default 1); `--max-lora-rank` = el rank (el ancho de las matrices del adaptador) más grande que aceptas (default 16): uno mayor no carga. En `/v1/models` aparecen con el modelo base como `parent`.

```
triggers:
  - type: prometheus
    metadata:
      serverAddress: url_prometheus
      query:
sum(vllm:num_requests_waiting{namespace="ns"})
      threshold: "umbral"
```

Con **KEDA** (el autoscaler de Kubernetes que escala con métricas de afuera, como una query de Prometheus), escala por **cola** (requests esperando adentro de vLLM), no por utilización de GPU, que está cerca de 100 % con 5 secuencias o con 90. Con el `metricType` por defecto (`AverageValue`), el HPA (el autoscaler de pods de Kubernetes) que crea KEDA pide ceil(cola total ÷ umbral) réplicas. Escribe `minReplicaCount`: su default es 0, y un LLM que baja a cero paga minutos de arranque en el primer request.

```
vllm serve modelo --enforce-eager
# en el log de arranque, sin --enforce-eager:
#   Model loading took gib GiB memory and seg seconds
#   torch.compile took seg s in total
#   Graph capturing finished in seg secs, took gib GiB
```

CUDA graphs graba la secuencia de kernels de un paso y la repite sin el costo de lanzarlos uno por uno: decode más rápido, a cambio de segundos de captura y algo de memoria al arrancar. `--enforce-eager` apaga torch.compile y CUDA graphs: arranca antes y genera más lento. La caché de **torch.compile** (el grafo del modelo compilado a kernels propios) vive en `~/.cache/vllm/torch_compile_cache` (`VLLM_CACHE_ROOT`); con ella, el segundo arranque no recompila.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ventana	ventana de rate()	`5m`
modelo	el modelo que sirves	`/models/chat-70b-fp8-2026-09-28`
k	cuántos tokens propone el método por paso	`4`
n_max	el trozo más largo que ngram busca en el prompt	`5 (si no fijas ni el mínimo ni el máximo, los dos quedan en 5)`
metodo	de dónde salen las propuestas	`draft_model · eagle3 · mtp`
draft	el modelo chico o la cabeza entrenada para tu modelo	`uno de la misma familia, con el mismo tokenizer`
modelo_base	el modelo sobre el que se entrenaron los adaptadores	`el 70B del C10 (el caso no usa LoRA)`
n	adaptadores distintos en un mismo batch	`1 por defecto`
r	el rank más grande que aceptas	`16 por defecto; hasta 512`
nombre = *ruta*	el nombre que piden los clientes y dónde está el adaptador	`listings-lora=/models/adapters/listings (ilustrativo)`
url_prometheus	el Prometheus que lee KEDA	`http://prometheus-operated.monitoring.svc:9090`
ns	namespace de vLLM	`c10`
umbral	requests esperando por réplica que aceptas	`5 (el tutorial de KEDA del production-stack de vLLM)`
gib, *seg*	memoria y segundos que vLLM informa al arrancar	`los del log de tu pod`
λ	requests por segundo que recibe una réplica	`60 ÷ 16 = 3.75`
α	probabilidad de que se acepte cada token propuesto	`0.5–0.8 según la tarea`

FÓRMULAS

$$\text{prefill} = \lambda \times (n_{\text{prompt}} - n_{\text{caché}}) = 3.75 \times (2{,}000 - 992) \approx 3{,}780 \text{ tokens/s por réplica} \quad (\text{sin caché: } 7{,}500)$$

$$E[\text{tokens por paso}] = \frac{1 - \alpha^{k+1}}{1 - \alpha} \qquad \alpha = 0.7,\ k = 4 \Rightarrow 2.77$$

$$\text{mean acceptance length} = 1 + \frac{\text{aceptados}}{\text{drafts}} \qquad \text{acceptance rate} = \frac{\text{aceptados}}{\text{propuestos}} = \frac{2.77 - 1}{4} \approx 44\,\%$$

$$\text{réplicas} = \left\lceil \frac{\sum \text{num_requests_waiting}}{\text{umbral}} \right\rceil \qquad t_{\text{arranque}} \approx \underbrace{2\text{–}5}_{\text{nodo}} + \underbrace{1\text{–}3}_{\text{imagen}} + \underbrace{1\text{–}3}_{\text{pesos}} + \underbrace{1\text{–}3}_{\text{carga y graphs}} \text{ min}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Prefix caching (APC)	Reusa los bloques de KV de un comienzo de prompt idéntico. Ahorra prefill: baja el TTFT y libera GPU; no acelera el decode.	**≈ 1,000 tokens de instrucciones por producto: ≈ la mitad de cada prompt**
Bloque completo de 16 tokens	Solo se cachean bloques llenos, encadenados por hash desde el primer token. Si todo el prompt acierta, vLLM recalcula igual el último token para tener los logits (en la práctica, su bloque completo).	**un prefijo de 1,000 tokens deja 62 bloques (992 tokens) en caché**
Lo fijo primero	El nombre de la persona, la hora exacta o un id al comienzo cambian el hash de todos los bloques que siguen: la tasa de aciertos cae a ≈ 0.	**la regla de prompts del C10**
cache_salt	Un texto por request que entra en el hash del primer bloque: solo comparten caché los requests con la misma sal. Evita que un tenant adivine por tiempos lo que mandó otro.	**una sal por tenant (pendiente en la referencia)**
Speculative decoding	Algo barato propone k tokens y el modelo grande los verifica juntos. Baja el TPOT con poca concurrencia; con la GPU llena, lo rechazado es trabajo tirado.	**fuera del diseño base del C10 (≈ 40 secuencias por réplica)**
ngram · draft_model · eagle3 · mtp · suffix	De dónde salen las propuestas: copiar del prompt, un modelo chico, una cabeza entrenada para tu modelo, capas de predicción del propio modelo o un árbol de sufijos.	**ngram para extracción: la salida copia trozos del input**
Mean acceptance length	Tokens por paso del modelo grande: 1 + aceptados ÷ drafts. La acceptance rate de vLLM es otra cosa: aceptados ÷ propuestos.	**2.77 tokens por paso con α = 0.7 y k = 4**
Multi-LoRA	Un modelo base con varios adaptadores encima; cada request elige el suyo por nombre. Cuesta memoria y algo de latencia por adaptador.	**una opción para el C10 si cada producto tuviera su fine-tuning**
--max-loras · --max-lora-rank · --max-cpu-loras	Cuántos adaptadores distintos por batch (default 1), el rank máximo (default 16) y cuántos quedan en memoria de CPU (al menos max-loras).	**rank 32 con el default: LoRA rank 32 is greater than max_lora_rank 16.**
Autoscaling por cola	Escalar con vllm:num_requests_waiting (o el tiempo en cola), no con la utilización de GPU, que está cerca de 100 % con 5 o con 90 secuencias.	**el C10 tiene réplicas fijas: el porqué está en la trampa**
Cold start	El tiempo hasta servir: nodo, imagen, pesos, carga a la GPU, torch.compile y CUDA graphs.	**≈ 5–10 min según el C10 (la suma de sus tramos, estimados: mídelo)**
Pesos en NVMe local	Copiar los safetensors a un disco del nodo antes de arrancar: vLLM los lee por mmap y un disco de red convierte la carga en minutos.	**init container: aws s3 sync + sha256sum -c**
Caché de torch.compile	Lo compilado se guarda en ~/.cache/vllm/torch_compile_cache; con caché, el arranque siguiente no recompila. VLLM_DISABLE_COMPILE_CACHE=1 la apaga.	**con HOME=/home/vllm, en el sistema de archivos del contenedor**
CUDA graphs · --enforce-eager	Graphs: el decode repite pasos grabados sin lanzar kernels uno por uno. --enforce-eager apaga graphs y torch.compile: arranca antes y genera más lento.	**el C10 deja los graphs (sin --enforce-eager)**

PALANCA	QUÉ BAJA	QUÉ CUESTA	EN EL C10
--enable-prefix-caching	prefill: TTFT y GPU (≈ 50 % del prompt en el C10)	nada si los prompts empiezan igual; no ayuda si empiezan distinto	activado (default en v0.30)
--speculative-config	TPOT con poca concurrencia	GPU en lo rechazado; memoria si hay un draft model	no: ≈ 40 secuencias por réplica
--enable-lora	flotas: un modelo base para varios productos	memoria y algo de latencia por adaptador	no (sin fine-tunings)
KEDA con num_requests_waiting	GPUs ociosas en el valle	arranques de minutos y H100 que quizá no existan	no: flota fija; la admisión absorbe los picos

PALANCA	QUÉ BAJA	QUÉ CUESTA	EN EL C10
Pesos en NVMe y caché de compile	minutos de arranque	disco local y más piezas	pesos sí (init container); caché de compile no
CUDA graphs (default) · --enforce-eager	graphs: tiempo por paso de decode; eager: tiempo de arranque	graphs: memoria y segundos al arrancar; eager: TPOT	graphs (sin --enforce-eager)

Ninguna se deja prendida sin medirla con el barrido de goodput y la mezcla real (bench/run_sweep.sh).

Las palancas de vLLM una vez que el modelo cabe: qué baja cada una, qué cuesta y qué decidió el C10.

EJEMPLO REAL · C10: POR QUÉ LA FLOTA ES FIJA Y QUÉ SE HIZO PARA QUE ARRANQUE RÁPIDO

YAML reference/c10/k8s/vllm-70b.yaml (líneas 12–17, 195, 215–217 y 238–245; las líneas # … resumen lo omitido)

```yaml
# Por qué réplicas FIJAS (sin HPA/KEDA): un nodo p5 nuevo + la imagen (≈ 8.7 GB comprimida) + ≈ 70 GB de pesos +
# CUDA graphs ≈ 5–10 min (est., mídelo). Un autoscaler no sigue un pico de 1 minuto con eso; y de noche la
# flota no queda ociosa, porque la usa el batch de catálogo. Se mantiene la flota del pico prendida y los picos
# los absorbe la admisión del gateway (gateway/admission.py). Los nodos los da la capacidad reservada
# (On-Demand Capacity Reservation o Capacity Blocks for ML; verifica disponibilidad de p5 en tu región), no
# "on-demand cuando haga falta": la H100 es el recurso escaso, no el dinero.
# …
            - --enable-prefix-caching  # default en v0.30; explícito porque el plan cuenta con él (system prompts)
# …
          env:
            - name: HOME
              value: /home/vllm  # caches de vLLM, torch.compile y Triton en un lugar escribible para UID 2000
# …
          startupProbe:
            # Cargar ≈ 70 GB desde NVMe, perfilar memoria, capturar CUDA graphs y (sin caché) torch.compile: minutos.
            # 90 × 10 s = 15 min antes de que cuente la liveness. Si se pasa, el pod reinicia (y lo ves).
            httpGet:
              path: /health
              port: http
            periodSeconds: 10
            failureThreshold: 90
```

- Sin autoscaling: el arranque de una réplica (≈ 5–10 min estimados) es más largo que cualquier pico de minutos, y las H100 son el recurso escaso. La flota del pico queda prendida, el 30 % de margen y la admisión del gateway absorben los picos, y de noche la usa el batch.
- Prefix caching ya viene activado en v0.30; el flag explícito documenta que el plan cuenta con él. Con 1,000 tokens de instrucciones por producto, cada réplica baja de 7,500 a ≈ 3,780 tokens/s de prefill en el pico, y 40 secuencias ocupan 53,000 tokens de KV en vez de 90,000 (`kv_tokens_with_shared_prefix` en `capacity/vram_calc.py`).
- `HOME=/home/vllm` pone las cachés de vLLM y de torch.compile en el sistema de archivos del contenedor: un contenedor nuevo empieza sin caché y vuelve a compilar. Un volumen montado en `~/.cache/vllm` (o en `VLLM_CACHE_ROOT`) la conservaría entre reinicios del contenedor dentro del mismo pod; el C10 no lo hace.
- La startupProbe da 15 min y, en el C10, es el único límite: con un solo API server, vLLM v0.30 espera al engine sin plazo. `VLLM_ENGINE_READY_TIMEOUT_S` (600 s por defecto) solo corta cuando el API corre en procesos aparte (`--api-server-count` mayor que 1, o data parallel), con `Timed out waiting for engine core processes to start` ; si un día los usas y el arranque puede pasar de 10 min, sube esa variable.
- Para reiniciar más rápido hay piezas nuevas. Desde v0.30, un daemon por GPU guarda los pesos ya cuantizados y repartidos, y un engine que reinicia los toma por CUDA IPC (memoria de la GPU compartida entre procesos) con `--load-format ipc_cache` ; en v0.30 el daemon se lanza con `python -m vllm.model_executor.model_loader.weight_cache.daemon` y desde v0.31 con `vllm preload` . En v0.32, `vllm download-kernels` instala antes los kernels precompilados de FlashInfer que se bajarían o compilarían al arrancar. El C10 no usa ninguna.

CÓMO RECORDARLA

prefix caching baja el TTFT, speculative decoding baja el TPOT, LoRA ahorra flotas, KEDA ahorra GPUs ociosas y la caché ahorra arranques. Cada una se paga: aciertos que dependen de cómo armas el prompt, GPU tirada en lo rechazado, memoria, minutos de arranque.

TRAMPA

copiar el autoscaling del C01. Una réplica del C10 tarda ≈ 5–10 min en servir (nodo p5, imagen, 70 GB de pesos, CUDA graphs) y la H100 puede no existir en ese momento: un pico de un minuto termina antes de que llegue la réplica nueva. Por eso hay 16 réplicas fijas y la admisión del gateway absorbe los picos. La otra: un prompt que empieza con el nombre de la persona o la hora exacta cambia el hash de todos sus bloques, y la tasa de aciertos cae a 0 aunque el resto sea idéntico.

En el curso: M14 · Batching, caching, speculative decoding y paralelismo · C10 · Serving · M6 · KEDA: escalar por eventos y hasta cero · M6 · Cold starts de punta a punta · C10 · Modos de falla y degradación · M7 · Servidores de LLMs: vLLM, SGLang, TGI, TensorRT-LLM · Números · Optimización y serving: rangos y umbrales · Hoja · Kubernetes · autoscaling con KEDA

11 El gateway de LLMs del C10: cuotas en tokens, admisión por prioridad, streaming detrás de proxies, cancelación y PII

¿Qué le pasa a un request en el llm-gateway antes de llegar a vLLM, qué código recibe el producto cuando no hay lugar, y cómo evitas que un proxy trabe el stream o que la GPU siga generando para alguien que ya se fue?

SINTAXIS CON VARIABLES

```
cobro = min(prompt_estimado + max_tokens, max_model_len)   # prompt_estimado = caracteres ÷ 3 + 4 por mensaje
ok, espera_s = balde[ tenant ].try_take(cobro)
if not ok:
    return 429, {"Retry-After": str(max(1, math.ceil(espera_s)))}
# … al terminar, con el usage del último chunk:
balde[ tenant ].settle(cobro, usage["prompt_tokens"] + usage["completion_tokens"])
```

Un **token bucket** (un balde de fichas que se recarga a ritmo fijo) contado en **tokens**, no en requests: un prompt de 8,000 tokens cuesta ≈ 27 veces uno de 300. El costo real se sabe al final (`usage`), así que al entrar se cobra una estimación pesimista y al salir se **concilia**: se devuelve lo que no se usó. Cada réplica del gateway tiene su propio balde con la cuota ÷ 4.

```
try:
    await admision.acquire( clase )       # interactive o batch: un lugar libre, o esperar en una cola con tope
except Overloaded as e:                    # cola llena o se esperó demasiado
    return 503, {"Retry-After": str(math.ceil(e.retry_after_s))}   # o el vendor, si el tenant puede
try:
    …                                      # el request a vLLM
finally:
    admision.release( clase )
```

El **control de admisión** (decidir en la puerta quién entra a la flota) limita los requests en vuelo: 288 por réplica del gateway, de los cuales 72 nunca los ocupa el batch. Cuando se libera un lugar, pasa primero un interactivo. La espera tiene tope de tamaño y de tiempo: 64 y 0.5 s el interactivo, 256 y 30 s el batch. **429** = te pasaste de tu cuota; **503** = la plataforma está llena.

```
location /v1/ {
    proxy_pass         http:// gateway ;
    proxy_buffering    off;                # o que la respuesta traiga el header X-Accel-Buffering: no
    proxy_read_timeout  silencio_max s;  # default 60s: lo máximo sin bytes entre dos lecturas
}
```

Con **SSE** (Server-Sent Events: líneas `data: …` que llegan una por una en la misma respuesta HTTP), cada proxy del camino tiene que reenviar al instante. nginx guarda la respuesta en buffers (memoria intermedia) por defecto (`proxy_buffering on`): el texto llega a ráfagas o todo al final, y el TTFT que ve la persona se vuelve el tiempo total (la cadena lo muestra con un proxy de prueba). Además un proxy corta una conexión callada: nginx a los 60 s (`proxy_read_timeout`) y el ALB a los 60 s (idle timeout). vLLM puede mandar comentarios `: keep-alive` con `--sse-keep-alive-interval` (0 = apagado). El gateway del C10 no los reenvía ni los necesita: su silencio hacia el cliente ya queda debajo de 60 s (≤ 30 s entre eventos; antes del primer byte, la cola + conectar + 10 s), y en vLLM los deja apagados porque un comentario reiniciaría el read timeout con el que corta un stream trabado.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
tenant	el producto que llama, sacado de su API key	`support-assistant`
clase	la prioridad del tenant (la decide la plataforma, no el cliente)	`interactive · batch`
gateway	a dónde reenvía el proxy	`las réplicas del llm-gateway`
silencio_max	segundos sin bytes que el proxy tolera	`más que el primer token más lento (10 s en el C10) y que cualquier pausa`
url_gateway	la URL del gateway, nunca la de vLLM	`http://llm-gateway.c10.svc:8080 (ilustrativa: el C10 no trae el manifest del gateway)`
key	la API key del tenant; el gateway guarda solo su sha256	`sk-c10-demo-support (solo en la demo)`
alias	el nombre estable del modelo	`chat-default`
reintentos	reintentos automáticos del cliente de openai	`0 o 1 en interactivo (el default es 2)`
timeout_s	espera máxima del cliente entre bytes	`lo que acepte tu producto (el default es 600)`
max_tokens	tope de salida de este request	`≤ 1,024 en soporte; si no lo mandas, el gateway pone el tope`
Q	cuota total del tenant en tokens por minuto	`5.4 M (soporte)`
N	réplicas del gateway	`4`
B, r	capacidad del balde y recarga por segundo, en una réplica	`1.35 M y 22,500`
$\hat{p}$	prompt estimado: caracteres ÷ 3 + 4 por mensaje	`≈ 2,000`
m	max_tokens del request	`1,024`
L	max_model_len	`8,192`

```
from openai import OpenAI
client = OpenAI(base_url="url_gateway/v1", api_key="key",
max_retries=reintentos, timeout=timeout_s)
stream = client.chat.completions.create(model="alias",
messages=mensajes, stream=True, max_tokens=max_tokens)
for chunk in stream:
    if chunk.choices and chunk.choices[0].delta.content:   #
el chunk de usage trae choices vacío
        print(chunk.choices[0].delta.content, end="",
flush=True)
```

Los productos usan el cliente de openai contra el gateway, con el **alias** en `model`. Ojo con sus defaults (openai-python 3.37): reintenta 2 veces los 408, 409, 429 y 5xx, respeta `Retry-After` hasta 2 min y espera hasta 600 s (5 s para conectar). En el camino interactivo usa `max_retries=0` o 1: si no, un 503 de admisión vuelve solo, dos veces, justo cuando la flota está llena.

FÓRMULAS

$$B = \frac{Q}{N} \qquad r = \frac{Q}{60N} \qquad \frac{5.4\ \text{M}}{4} = 1.35\ \text{M tokens}, \quad r = 22{,}500\ \text{tokens/s}$$

$$\text{cobro} = \min(\hat{p} + m,\ L) = 2{,}000 + 1{,}024 = 3{,}024 \qquad \text{devolución} = 3{,}024 - 2{,}250 = 774$$

$$\text{Retry-After} = \left\lceil \frac{\text{cobro} - \text{saldo}}{r} \right\rceil \qquad \left\lfloor \frac{1{,}350{,}000}{3{,}024} \right\rfloor = 446 \text{ en una ráfaga} \qquad \left\lfloor \frac{1{,}350{,}000}{2{,}250} \right\rfloor = 600 \text{ conciliando}$$

$$\text{tope} = \frac{16 \times 72}{4} = 288 \qquad L = \lambda W = 60 \times 10.5 \approx 630 \Rightarrow \frac{630}{4 \times 288} \approx 55\,\%$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
API key → sha256 → tenant	El gateway no guarda las keys: guarda su hash y busca al tenant con él. Sin tenant, 401, que igual se cuenta (tenant unknown) y deja su línea de log.	`sk-c10-demo-support → support-assistant`
Alias	El nombre estable que piden los productos; el gateway lo cambia por la revisión servida. Cambiar de modelo es editar tenants.yaml.	`chat-default → chat-70b-fp8-2026-09-28`
Pin por tenant (canario)	Manda a un tenant, o a un % de sus usuarios finales por hash del campo user, a otro Service y otra revisión.	`{upstream: vllm-70b-canary, model: chat-70b-fp8-2026-10-12, percent: 5}`
Cuota en tokens/min	Un balde por tenant y por réplica del gateway: capacidad = 1 min de cuota; recarga = cuota ÷ 60 por segundo.	`soporte: 5.4 M ÷ 4 = 1.35 M por réplica`
Cobro estimado y conciliación	Al entrar cobra prompt estimado + max_tokens; al final devuelve lo que no se usó según usage, o cobra lo que faltó.	`cobra 3,024, usa 2,250, devuelve 774`
max_tokens obligatorio	Si el cliente no lo manda, el gateway pone el tope del tenant: sin tope, vLLM genera hasta llenar el contexto.	`1,024 soporte · 600 publicaciones · 400 catálogo`
Admisión por prioridad	Un tope de requests en vuelo hacia la flota, con lugares que el batch nunca ocupa. Al liberarse uno, pasa primero un interactivo; dentro de cada clase, el que llegó antes.	`288 por réplica del gateway (16 × 72 ÷ 4), 72 reservados`
Cola acotada	Esperar un lugar con tope de tamaño y de tiempo. Lo que no entra recibe 503 con Retry-After.	`interactivo: 64 y 0.5 s · batch: 256 y 30 s`
priority hacia vLLM	El gateway manda 0 (interactive) o 10 (batch). Con --scheduling-policy priority, vLLM atiende antes el número menor y, si falta KV, desaloja primero al mayor; nunca saca a uno que corre para meter a uno que espera.	`la documentación dice que sin el flag da error; en el código de v0.30.0 no se encontró esa validación (la cola fcfs no lo lee): no cuentes con el error, pon el flag`
Primer evento con contenido	Hasta leer el primer token con texto (o un fin normal), el gateway no le manda nada al cliente: una falla todavía puede ser un 502/504 limpio o ir al vendor, también un stream que llega a [DONE] sin texto (un abort antes del primer token).	`first_token_timeout_s: 10`
[DONE] e include_usage	El stream termina con data: [DONE]. El gateway pide stream_options.include_usage para conciliar, y esconde ese chunk si el cliente no lo pidió.	`el chunk de usage trae choices: []`
Cancelación y 499	Si el cliente se va, el gateway cierra su conexión con vLLM, y vLLM aborta el request y libera su KV. Cuenta como 499 (la convención de nginx), fuera de la tasa de errores.	`curl --max-time 0.5 → 499 e inflight vuelve a 0`
stream_error	El upstream falló con el stream ya empezado: el cliente vio un 200 incompleto y un evento de error. Sí cuenta como error.	`un nodo que muere a mitad de 4 streams`
Sin keep-alive hacia vLLM	Cada request abre su conexión TCP: kube-proxy reparte conexiones, no requests, y una reusada seguiría yendo a un pod que ya salió.	`max_keepalive_connections=0 · max_connections=None`

PIEZA	QUÉ HACE	EJEMPLO
Fallback gobernado	Solo tenants con vendor_fallback, solo campos de una lista permitida y con la PII redactada. Un 429 del vendor se vuelve nuestro 503; un 401/403, un 502.	`listing-assistant sí; support-assistant no`
Redacción de PII	Regex para correos, teléfonos, tarjetas, CLABE, CURP y RFC (tarjetas y CLABE, además, con su dígito verificador), con marcadores estables. No ve nombres ni direcciones.	`ana.p@mail.com → [EMAIL_1]`

EL REQUEST BAJA

EL CLIENTE · SOPORTE

`POST /v1/chat/completions {model: chat-default, stream: true, max_tokens: 512}`

1 Auth — < 1 ms

Bearer key → sha256 → tenant support-assistant (interactive, 5.4 M tokens/min, tope 1,024, sin vendor)

SI FALLA: `401`

2 Validación — ≈ 1–5 ms

alias permitido, n = 1, max_tokens ≤ 1,024, prompt estimado ≤ 6,000

SI FALLA: `400` `403` `404`

3 Cuota en tokens — < 1 ms

cobra ≈ 2,000 + 512 del balde de esta réplica (5.4 M ÷ 4 por minuto)

SI FALLA: `429 + Retry-After`

4 Admisión — ≤ 0.5 s

un lugar interactivo de los 288, o cola (≤ 64, ≤ 0.5 s)

SI FALLA: `503 + Retry-After` (publicaciones: vendor con la PII redactada)

5 vLLM — ≈ 0.3–1 s al primer token

Service vllm-70b, model = chat-70b-fp8-2026-09-28, priority 0, include_usage; conexión TCP nueva

SI FALLA: `502 / 504` (antes del primer byte)

6 Stream SSE — ≈ 40 ms por token

cada evento se reenvía al llegar; el TTFT se mide en el primer evento con contenido

SI FALLA: `evento de error` (stream_error) `499 si el cliente se va`

7 Cierre

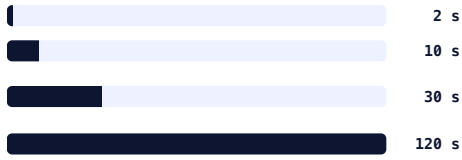
[DONE]; libera el lugar, concilia la cuota con usage, métricas y una línea de log sin prompt

EL STREAM VUELVE

LOS TOPES

288 en vuelo por réplica del gateway
72 reservados para interactive
64 · 0.5 s cola de interactive (lugares · espera máxima)
256 · 30 s cola de batch (lugares · espera máxima)
Retry-After: **2 s** en interactive · **60 s** en batch

LOS TIMEOUTS DEL GATEWAY, A ESCALA (DE 0 A 120 S)



LOS TENANTS (TENANTS.YAML)

TENANT	CLASE	TOKENS POR MINUTO	MAX_TOKENS	PROMPT MÁXIMO	¿VENDOR?
support-assistant	interactive	5,400,000	1,024	6,000	no
listing-assistant	interactive	3,000,000	600	6,000	sí
catalog-enrichment	batch	6,000,000	400	2,000	no

espera en la cola · vLLM · el stream: tokens que vuelven · cierre · el código si falla

Un request de soporte por el llm-gateway del C10: cada etapa con su tiempo y el código que devuelve si falla; el stream vuelve por el mismo camino. Números del README del C10 y de tenants.yaml.

EJEMPLO REAL · C10: EL CAMINO DE UN REQUEST EN LLM_GATEWAY.PY, DE LA KEY A VLLM

PYTHON — reference/c10/gateway/llm_gateway.py (Gateway.chat_completions, líneas 597–606, 619–626, 635–668 y 673–678; las líneas # … resumen lo omitido)

```python
    async def chat_completions(self, request: Request) -> Response:
        t0 = self.clock()
        rid = uuid.uuid4().hex
        auth = request.headers.get("authorization", "")
```

```
        key = auth[7:] if auth.lower().startswith("bearer ") else ""
        tenant = self.cfg.by_key.get(hashlib.sha256(key.encode()).hexdigest()) if key else None
        if tenant is None:
            # También cuenta y deja su línea (tenant "unknown"): una key rotada en un producto es una ola de 401
            return self._reject(rid, t0, "unknown", "unknown", 401, "API key inválida", "authentication_error",
                                "invalid_api_key")
        # … el cuerpo tiene que ser un objeto JSON (si no, 400; si el cliente corta a mitad de mandarlo, 499)
        alias_name = body.get("model")
        alias = self.cfg.aliases.get(alias_name) if isinstance(alias_name, str) else None
        if alias is None:
            return self._reject(rid, t0, tenant.name, "unknown", 404, f"el modelo {alias_name!r} no existe",
                                "invalid_request_error", "model_not_found")
        if alias.name not in tenant.allowed_aliases:
            return self._reject(rid, t0, tenant.name, alias.name, 403, f"el tenant no puede usar {alias.name}",
                                "permission_error", "model_not_allowed")
        # … messages no vacío y n = 1 (si no, 400)
        max_tokens = body.get("max_completion_tokens", body.get("max_tokens"))
        if max_tokens is None:
            max_tokens = tenant.max_tokens_cap
            body["max_completion_tokens"] = max_tokens  # sin tope, vLLM genera hasta llenar el contexto
        if not isinstance(max_tokens, int) or isinstance(max_tokens, bool) or not 1 <= max_tokens <= tenant.max_tokens_cap:
            return self._reject(rid, t0, tenant.name, alias.name, 400,
                                f"max_tokens debe estar entre 1 y {tenant.max_tokens_cap}", "invalid_request_error",
                                "max_tokens_exceeded")
        stream = bool(body.get("stream", False))

        # ---- cuota: prompt estimado + max_tokens, sin pasar del contexto (un request no puede usar más que eso)
        est_prompt = estimate_prompt_tokens(messages)
        if est_prompt > tenant.max_prompt_tokens:
            return self._reject(rid, t0, tenant.name, alias.name, 400,
                                f"prompt de ≈ {est_prompt} tokens (tope {tenant.max_prompt_tokens})",
                                "invalid_request_error", "prompt_too_long")
        charged = min(est_prompt + max_tokens, self.cfg.max_model_len)
        ok, wait_s = self.buckets[tenant.name].try_take(charged)
        if not ok:
            return self._reject(rid, t0, tenant.name, alias.name, 429, "cuota de tokens por minuto agotada",
                                "rate_limit_error", "rate_limit_exceeded",
                                {"Retry-After": str(max(1, math.ceil(wait_s)))})

        # ---- admisión a la flota
        priority = tenant.priority
        try:
            await self.admission.acquire(priority)
        except Overloaded as e:
            if self._can_fallback(tenant, alias):
                self.metrics.vendor.labels(tenant.name, "overloaded").inc()
                return await self._to_vendor(request, tenant, alias, body, stream, charged, est_prompt, t0, rid)
            self.buckets[tenant.name].settle(charged, 0)  # no se usó nada: se devuelve la cuota
            return self._reject(rid, t0, tenant.name, alias.name, 503, f"plataforma sin capacidad ({e.reason})",
                                "overloaded_error", "overloaded", {"Retry-After": str(math.ceil(e.retry_after_s))})
        # … si el cliente se va mientras espera lugar, también se devuelve la cuota
        up_name, model = tenant.route_for(alias, body.get("user"))
        upstream = self.upstreams[up_name]
        self_body = self._self_body(body, model, priority)
        if stream:
            return await self._stream_self(request, tenant, alias, body, upstream, self_body, charged, est_prompt, t0, rid)
        return await self._complete_self(request, tenant, alias, body, upstream, self_body, charged, est_prompt, t0, rid)
```

- **Quién eres**: la key se convierte en sha256 y se busca en `by_key`. tenants.yaml solo guarda hashes: si se filtra, no sirve para llamar al gateway. Sin tenant, 401: también se cuenta (con tenant `unknown`) y deja su línea, así una key rotada en un producto se ve como una ola de 401.
- **Cuánto te queda**: `charged` es pesimista (caracteres ÷ 3 + 4 por mensaje + max_tokens, sin pasar de 8,192) y se ajusta al final en `finish()` con el usage real. Si el cliente se fue o el stream falló sin usage, se cobra el prompt estimado: el prefill ya se pagó en la GPU.
- **Hay lugar**: si no, publicaciones va al vendor (`overloaded`) y soporte recibe 503 con `Retry-After: 2`; la cuota se devuelve entera, porque no se usó nada.
- **A qué modelo**: `route_for` cambia el Service y la revisión juntos, porque cada pod de vLLM sirve solo su `--served-model-name` y otro nombre da 404.
- **El stream** (en `_stream_self`): `_open_and_first` espera el primer evento con contenido hasta 10 s, y un `[DONE]` o un corte antes de ese evento también es falla (502 o el vendor): así llega un abort antes del primer token, sin ningún chunk de abort. Después no hay fallback, porque no se pegan dos streams de dos modelos. `race_disconnect` y `GuardedStream` cierran la conexión con vLLM si el cliente se va: vLLM v0.30 atrapa la cancelación y aborta el request (`vllm/v1/engine/async_llm.py`), y el gateway anota 499.
- Toda respuesta deja su conteo en `llm_gateway_requests_total` y una línea JSON sin contenido (`_account`): los 200, los rechazos (el 401 con tenant `unknown`), las fallas antes del primer byte (502, 504), los 4xx reenviados de vLLM, los 499 y los `stream_error`. La línea lleva el código del error, nunca su mensaje, y el mismo `request_id` que el header `x-request-id` de la respuesta. `finish()` concilia la cuota de lo que llegó a un upstream y después llama a `_account`.

CÓMO RECORDARLA

quién eres → cuánto te queda → hay lugar → a qué modelo → stream → cuenta final, o sea 401 · 429 · 503 · alias · SSE · conciliación. Y tres códigos: **429 = tu cuota; 503 = la plataforma llena; 499 = te fuiste y vLLM abortó**.

TRAMPA

algo en el camino que bufferea, reintenta o no avisa. Un proxy con buffer convierte el stream en una respuesta al final: el TTFT se vuelve el tiempo total sin que cambie nada en vLLM. El cliente de openai con sus defaults reintenta por su cuenta los 429 y los 503, dos veces: el gateway ve el triple de intentos justo cuando está lleno. Y si un proxy intermedio no cierra la conexión con vLLM cuando el cliente se va, la GPU genera hasta max_tokens para nadie: suben los 499 y no bajan los tokens generados.

En el curso: C10 · Serving · C10 · Archivos de referencia · M15 · Rate limiting y protección contra abuso · M15 · Datos personales: dónde están, cuánto duran y quién los ve · M3 · Timeouts, reintentos y hedging · M14 · Guardrails: inyección, PII, salidas estructuradas y cuotas · M15 · Degradación con gracia: contener la falla · C10 · Modos de falla y degradación · Hoja · El gateway · admisión · Hoja · El gateway · timeouts y reintentos · Hoja · El gateway · apagado ordenado

12 Diagnóstico: el método y la tabla de síntomas

Alguien dice «tarda» o «se cortó la respuesta», o el pod no arranca: ¿qué miras primero, qué después y qué te dice cada número?

SINTAXIS CON VARIABLES

```
curl -si http://host:puerto/health | head -1
kubectl get pods -n ns -l app.kubernetes.io/name=vllm-70b -o wide
```

¿Está vivo? `/health` responde 200 con el cuerpo vacío si el **engine** (el proceso que corre el modelo) está vivo, y 503 si murió (`EngineDeadError`): eso no se cura solo, se reinicia. No dice si está lleno: un pod sano puede tener 50 requests esperando. Llegas al puerto con `kubectl port-forward`; la NetworkPolicy del C10 solo deja entrar al gateway y a Prometheus.

```
curl -s http://host:puerto/metrics | grep -E '^vllm:(num_requests_(running|waiting)|kv_cache_usage_perc|num_preemptions_total)'
```

¿Hay cola adentro y alcanza la memoria? Tres gauges (números que suben y bajan) y un counter (uno que solo sube): corriendo, esperando, la fracción del KV ocupada (1 = 100 %) y las **preemptions** (secuencias que vLLM sacó de la GPU por falta de KV y va a recalcular) desde que arrancó el pod; para la tasa, `rate()` en Prometheus. Las series traen `model_name` y `engine`; Prometheus les suma `pod`, `namespace` y `track`.

```
kubectl logs -n ns pod -c vllm | grep -E 'Model loading took|KV cache size|Graph capturing finished'
kubectl logs -n ns pod -c vllm --since=ventana | grep 'Engine 000' | tail -1
```

¿Qué midió al arrancar y qué hace ahora? Al arrancar, vLLM escribe cuánto pesó el modelo, `GPU KV cache size: N tokens, Maximum concurrency for 8,192 tokens per request: X.XXx` (tu capacidad real) y la captura de CUDA graphs. Después, cada 10 s (`VLLM_LOG_STATS_INTERVAL`), una línea con el throughput de prompt y de generación, corriendo, esperando, `Preemptions: N` (solo si hubo), el uso del KV y la tasa de aciertos del prefix cache. V1 no escribe una línea por cada preemption.

```
kubectl describe pod -n ns pod | sed -n '/^Events:/,$p'
kubectl logs -n ns pod -c vllm --previous | tail -50
kubectl exec -n ns pod -c vllm -- nvidia-smi --query-gpu=index,memory.used,memory.total,utilization.gpu --format=csv
```

¿Qué le pasó al pod? Los eventos dicen Pending (`Insufficient nvidia.com/gpu`), probes que fallan, `OOMKilled` (la RAM del contenedor, no la GPU) o reinicios; `--previous` trae el log del intento que murió. `nvidia-smi` muestra ≈ 90 % de memoria ocupada desde el arranque: vLLM la reserva entera para pesos y KV.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
host, *puerto*	la réplica vista desde donde corres curl	`localhost:8000 (kubectl port-forward a un pod)`
ns	namespace de vLLM	`c10`
pod	un pod de vllm-70b (o deploy/vllm-70b, que toma uno)	`vllm-70b-6c9f7d8b4-q7k2m (ilustrativo)`
ventana	cuánto log reciente lees	`1m`
GPU KV cache size	tokens de KV que caben en la réplica, según vLLM al arrancar	`208,736 en la cuenta del C10`
max_model_len	el contexto máximo configurado	`8,192`
$W_{\text{admisión}}$	espera en la cola del gateway	`≤ 0.5 s en interactivo`
W_{cola}	espera adentro de vLLM: vllm:request_queue_time_seconds	`≈ 0 si num_requests_waiting = 0`
t_{prefill}	procesar el prompt hasta el primer token	`≈ 0.3–1 s en el C10, con prefix cache`
Δ	lo que creció un counter entre dos lecturas	`en 60 s`

FÓRMULAS

$$\text{concurrencia máx.} = \frac{\text{GPU KV cache size}}{\text{max_model_len}} = \frac{208{,}736}{8{,}192} \approx 25.48\text{x} \qquad \frac{208{,}736}{2{,}250} \approx 92 \text{ secuencias típicas}$$

$$\text{TTFT}_{\text{gateway}} \approx W_{\text{admision}} + W_{\text{cola}} + t_{\text{prefill}} \qquad \text{TTFT}_{\text{vLLM}} \approx W_{\text{cola}} + t_{\text{prefill}}$$

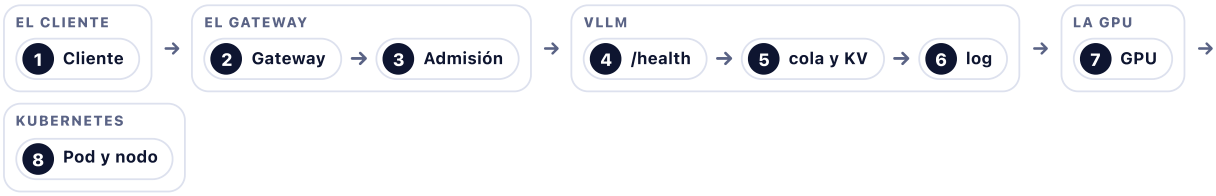
$$\frac{\Delta\text{vllm:num_preemptions_total}}{\Delta\text{vllm:request_success_total}} = \frac{24}{207} \approx 0.12 > 0.05 \Rightarrow \text{LLMPreemptionsRising}$$

$$\text{KV de una secuencia} = (n_{\text{prompt}} + n_{\text{salida}}) \times 320 \text{ KiB} = 5{,}300 \times 320 \text{ KiB} \approx 1.74 \text{ GB} \Rightarrow 40 \times 1.74 \approx 69.5 > 68.4 \text{ GB}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`/health 200`	El engine responde. No dice nada de cola ni de KV.	`la readinessProbe del C10`
`/health 503`	EngineDeadError: el engine murió y no se recupera solo. Reiniciar es lo correcto.	`la liveness reinicia tras 6 × 10 s`
`GPU KV cache size: N tokens`	Cuántos tokens de KV caben en la réplica: tu capacidad real, medida por vLLM al arrancar.	`208,736 en la cuenta del C10`
`Maximum concurrency for 8,192 tokens per request: X.XXx`	Cuántas secuencias de largo máximo caben a la vez. Las más cortas caben más.	`25.48x (≈ 92 de 2,250 tokens)`
`vllm:num_requests_waiting > 0`	Requests dentro de vLLM esperando lugar: falta KV o se llegó a max_num_seqs.	`KVCacheSaturation lo pide junto con KV > 95 %`

PIEZA	QUÉ HACE	EJEMPLO
vllm:kv_cache_usage_perc	Fracción del KV ocupada (1 = 100 %). Cerca de 1 de día y con cola: la memoria limita.	de noche, 100 % es el batch: normal
Preemptions: N	En la línea de stats, solo si hubo: vLLM sacó secuencias de la GPU por falta de KV y las va a recalcular.	LLMPreemptionsRising: > 0.05 por request terminado
finish_reason length	La respuesta paró en max_tokens o en el contexto. No es un error HTTP: es un 200 corto.	max_completion_tokens = 1,024 en soporte
finish_reason abort	vLLM abortó: el pod entró en drain o el cliente se fue.	antes del primer token, el gateway lo convierte en 502 o en el vendor: sin streaming lo ve en el finish_reason; con streaming, v0.30 no manda ese chunk y lo delata un [DONE] sin texto
TTFT alto en el gateway, normal en vLLM	La espera está antes de vLLM: admisión, red, el propio gateway.	llm_gateway_queue_depth
TTFT alto también en vLLM	Cola dentro de la réplica o prefill largo.	vllm:request_queue_time_seconds · vllm:request_prompt_tokens
nvidia-smi con ≈ 90 % de memoria sin tráfico	Normal: vLLM reserva gpu_memory_utilization al arrancar. La ocupación real es kv_cache_usage_perc.	≈ 72 de 80 GB por GPU con 0.90
EngineCore encountered an issue	El engine murió con el servidor corriendo; la causa está más arriba en el log.	kubectl logs --previous
Timed out waiting for engine core processes to start	El engine tardó más de 600 s (VLLM_ENGINE_READY_TIMEOUT_S) en estar listo. En v0.30 solo pasa con varios procesos de API (--api-server-count > 1); con uno solo, como el C10, el tope es la startupProbe.	pesos lentos o torch.compile sin caché



1 Cliente EL CLIENTE

¿Qué recibió: un código, una demora o un texto cortado?

TERMINAL

```
curl -sN "$URL/v1/chat/completions" -H "Authorization: Bearer $KEY" -H 'content-type: application/json' -d @req.json | tail -3
```

429 = la cuota del tenant; 503 = la admisión llena; 502/504 = vLLM no contestó o terminó sin texto (un abort antes del primer token, como en un drain); un 200 sin [DONE], o con finish_reason length, = cortado.

2 Gateway EL GATEWAY

¿La espera está antes de vLLM o adentro?

PROMQL

```
llm:gateway_ttft_p95:rate5m
```

Compáralo con max(llm:vllm_ttft_p95:rate5m): si solo sube el del gateway, la espera es la admisión o la red.

3 Admisión EL GATEWAY

¿Quedan lugares en vuelo?

PROMQL

```
sum(llm_gateway_inflight) / sum(llm_gateway_max_inflight)
```

Más de 0.85 = LLMCapacityHeadroomLow. Mira también llm_gateway_queue_depth por clase.

4 vLLM: /health VLLM

¿El engine está vivo?

TERMINAL

```
curl -si localhost:8000/health | head -1
```

200 = vivo; 503 = EngineDeadError: lee kubectl logs --previous.

5 vLLM: cola y KV VLLM

¿Hay cola adentro y alcanza la memoria?

TERMINAL

```
curl -s localhost:8000/metrics | grep -E '^vllm:(num_requests_waiting|kv_cache_usage_perc)'
```

Esperando > 0 con KV > 0.95 = la memoria limita; esperando > 0 con KV bajo = mira max-num-seqs y el tope del gateway.

6 vLLM: log VLLM

¿Hay preemptions? ¿Acierta el prefix cache?

TERMINAL

```
kubectl logs -n c10 deploy/vllm-70b -c vllm --since=1m | grep 'Engine 000' | tail -1
```

Preemptions: N aparece solo si hubo. Si la tasa de aciertos cae a la mitad, alguien cambió un prompt.

7 GPU LA GPU

¿La GPU tiene errores o la ocupa otro proceso?

TERMINAL

```
kubectl exec -n c10 deploy/vllm-70b -c vllm -- nvidia-smi --query-gpu=index,memory.used,memory.total,utilization.gpu --format=csv
```

≈ 90 % de memoria siempre (vLLM la reserva); los errores XID (fallas de la GPU que reporta el driver) salen en el DCGM exporter, el que pasa las métricas de la GPU a Prometheus.

8 Pod y nodo KUBERNETES

¿Kubernetes lo dejó correr?

TERMINAL

```
kubectl describe pod -n c10 -l app.kubernetes.io/name=vllm-70b | grep -A12 '^Events:'
```

Pending sin GPU, probes que fallan, OOMKilled (la RAM, no la VRAM) o reinicios.

> **El primer eslabón que falla es la causa.**

TERMINAL un comando: va en la terminal PROMQL una consulta: va en Prometheus o en Grafana

El método: recorre la cadena en orden, del cliente al nodo. El primer eslabón que falla es la causa; lo que viene después son consecuencias.

EJEMPLO REAL · C10: UN TTFT ALTO QUE ERA MEMORIA (Y LA MEMORIA, PROMPTS MÁS LARGOS)

BASH · ZSH Comandos reales con los nombres de reference/c10 (namespace c10, PodMonitor de k8s/vllm-70b.yaml, recording rules de monitoring/alerts.yml); las salidas son ilustrativas, con los nombres y el formato de vLLM v0.30.0 (vllm/v1/metrics/loggers.py)

```
# 12:10 suena LLMTTFTHigh para chat-default (TTFT p95 interactivo en el gateway > 1 s durante 10 min).
# 1) ¿La espera está antes de vLLM o adentro? El Prometheus del cluster:
kubectl port-forward -n monitoring svc/prometheus-operated 9090 &
curl -s localhost:9090/api/v1/query --data-urlencode 'query=llm:gateway_ttft_p95:rate5m' | jq -r '.data.result[].value[1]'
# 1.62
curl -s localhost:9090/api/v1/query --data-urlencode 'query=max(llm:vllm_ttft_p95:rate5m)' | jq -r '.data.result[].value[1]'
# 1.48        ← vLLM también subió: es la réplica, no la cola del gateway

# 2) Adentro de una réplica: cola, KV y la línea de stats
kubectl port-forward -n c10 pod/vllm-70b-6c9f7d8b4-q7k2m 8000 &
curl -s localhost:8000/metrics | grep -E '^vllm:(num_requests_(running|waiting)|kv_cache_usage_perc)\{'
# vllm:num_requests_running{engine="0",model_name="chat-70b-fp8-2026-09-28"} 38.0
# vllm:num_requests_waiting{engine="0",model_name="chat-70b-fp8-2026-09-28"} 7.0
# vllm:kv_cache_usage_perc{engine="0",model_name="chat-70b-fp8-2026-09-28"} 0.98
kubectl logs -n c10 vllm-70b-6c9f7d8b4-q7k2m -c vllm --since=1m | grep 'Engine 000' | tail -1
# … Engine 000: Avg prompt throughput: 15108.4 tokens/s, Avg generation throughput: 702.3 tokens/s, Running: 38 reqs, Waiting: 7 reqs,
Preemptions: 4, GPU KV cache usage: 98.0%, Prefix cache hit rate: 21.4%

# 3) ¿Por qué se llenó? El largo de los prompts, en toda la flota:
curl -s localhost:9090/api/v1/query --data-urlencode 'query=sum(rate(vllm:request_prompt_tokens_sum[30m])) /
sum(rate(vllm:request_prompt_tokens_count[30m]))' | jq -r '.data.result[].value[1]'
# 5050        ← el plan supone ≈ 2,000
```

- El orden importa: primero la comparación entre el TTFT del gateway y el de vLLM, que alerts.yml ya graba (`llm:gateway_ttft_p95:rate5m` y `llm:vllm_ttft_p95:rate5m`). Si solo subiera el del gateway, la espera estaría en la admisión o en la red, y la réplica estaría sana.
- Los gauges dicen memoria: 7 esperando con el KV al 98 %. La línea de stats lo confirma con `Preemptions: 4` en 10 s y una tasa de aciertos del prefix cache de 21 % (lo normal es ≈ 50 %).
- La causa: prompts de ≈ 5,050 tokens contra ≈ 2,000 del plan. Cada secuencia pide (5,050 + 250) × 320 KiB ≈ 1.74 GB de KV; las ≈ 40 en vuelo que da la ley de Little (3.75 req/s × ≈ 10.5 s) piden ≈ 69.5 GB y la réplica tiene 68.4. Con ese largo, los 208,736 tokens alcanzan para 39 secuencias, no para 92.
- El arreglo va en orden: que el tenant que alargó los prompts los recorte (un `max_prompt_tokens` más bajo en tenants.yaml es un cambio revisado), bajar el tope en vuelo mientras tanto, KV en FP8 si ya pasó su golden set, y réplicas al final. Reiniciar no sirve: el mismo tráfico vuelve a llenar el KV.
- Las salidas son de ejemplo. El formato de la línea de stats sale de `vllm/v1/metrics/loggers.py` de v0.30.0; el número de línea de cada log y el prefijo del proceso dependen de la versión.

CÓMO RECORDARLA

cliente → gateway → admisión → /health → cola y KV → log → GPU → pod. Tres números cuentan casi todo: `num_requests_waiting` (¿hay cola adentro?), `kv_cache_usage_perc` (¿falta memoria?) y `Preemptions: N` en el log (¿ya se está recalculando trabajo?).

TRAMPA

mirar `nvidia-smi` y concluir. Muestra ≈ 90 % de memoria ocupada desde el arranque (vLLM la reserva) y una utilización cerca de 100 % con 5 secuencias o con 90: ninguno de los dos dice si la réplica está llena. La otra: reiniciar el pod por preemptions o por un TTFT alto. El reinicio tarda minutos, se lleva el log del intento (usa `--previous`) y el problema vuelve con el mismo tráfico: se arregla con más KV por réplica o con menos secuencias.

En el curso: C10 · Monitoreo, alertas y reentrenamiento C10 · Modos de falla y degradación M14 · KV cache: la memoria que decide la concurrencia

M4 · Cálculo de VRAM paso a paso

M12 · Dashboards, runbooks y on-call

Docker y kubectl · Depurar pods: síntoma → comandos → causa típica

Hoja · Kubernetes · diagnóstico

13 Errores que debes reconocer rápido

Lees el mensaje: ¿quién lo dijo (vLLM al arrancar, vLLM al atender, CUDA o PyTorch, NCCL, Hugging Face, el cliente de openai o el gateway), qué significa, qué corres primero y dónde está la explicación?

SINTAXIS CON VARIABLES

NIVEL *mm-dd hh:mm:ss* **[** *archivo* **.py:** *linea* **]** *mensaje*

Una línea del log de vLLM (`kubectl logs … -c vllm`): el nivel, la fecha sin año y el archivo de vLLM que la escribió. Las del engine y de los workers llevan adelante el proceso, como `(EngineCore pid=271)`. Al arrancar (`kv_cache_utils.py`, `gpu_model_runner.py`) te dice capacidad y tiempos; después, cada 10 s, viene la línea `Engine 000: …` (`loggers.py`).

modulo **.** *Excepcion* **:** *mensaje*

…

RuntimeError: Engine core initialization failed. See root cause above. Failed core proc(s): *procesos*

Un traceback de Python (el rastro de llamadas que se imprime con un error). El engine de vLLM corre en otro proceso, así que el error del final es un envoltorio: la causa es la primera excepción más arriba, casi siempre un `ValueError` de configuración o un `OutOfMemoryError`.

{"error": {"message": " *mensaje* **", "type": "** *Tipo* **Error", "param":** *param* **, "code":** *status* **}}**

El cuerpo de un error HTTP de vLLM, con la forma de OpenAI: `message`, `type` (`BadRequestError` para un 400, `NotFoundError` para un 404), `param` y `code` (el status). El gateway del C10 contesta con la misma forma, sin `param` y con `code` en texto (`rate_limit_exceeded`, `overloaded`).

openai. *Clase* **: Error code:** *status* **–** *cuerpo*

Lo que ve el producto: el cliente de openai convierte el status en una clase (400 `BadRequestError`, 404 `NotFoundError`, 429 `RateLimitError`, 5xx `InternalServerError`) y le pega el cuerpo JSON entero, escrito como dict de Python: `{'error': {'message': …}}`. El texto suelto queda en `e.body['message']` y el code en `e.code`. Sin respuesta, `APIConnectionError` o `APITimeoutError`. Antes de lanzarla ya reintentó 2 veces los 429 y 5xx.

NCCL WARN *detalle*

huggingface_hub.errors. *Error* **:** *status* **Client Error. …** *url*

Dos fuentes de afuera de vLLM: **NCCL** (la librería de NVIDIA que comunica las GPUs de una réplica) escribe `NCCL WARN`, y cuenta más con `NCCL_DEBUG=INFO` o `TRACE`; **huggingface_hub** dice qué pasó al buscar el modelo en el Hub.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
NIVEL	qué tan grave	`INFO, WARNING o ERROR`
archivo .py: *linea*	de qué parte de vLLM salió	`kv_cache_utils.py · loggers.py`
mensaje	qué pasó	`GPU KV cache size: 208,736 tokens`
modulo . *Excepci on*	la clase del error	`torch.OutOfMemoryEr ror · ValueError`
procesos	los procesos del engine que murieron y su código de salida	`un diccionario con el nombre del proceso y su código`
Tipo Error, *status*	la clase del error HTTP y su código	`BadRequestError, 400 · NotFoundError, 404`
param	el campo culpable, si se sabe	`"input_tokens" o null`
Clase	la excepción del cliente de openai	`RateLimitError (429) · InternalServerError (5xx)`
Error , *url*	el error de Hugging Face y la URL que pidió	`GatedRepoError · …/resolve/main/conf ig.json`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Log de arranque de vLLM	Lo que midió al cargar: pesos, KV, CUDA graphs. Si el arranque falla, la causa está aquí.	`GPU KV cache size: 208,736 tokens`
Línea de stats (Engine 000)	Cada 10 s: throughput, corriendo, esperando, Preemptions (si hubo), KV y prefix cache.	`Preemptions: 4, GPU KV cache usage: 98.0%`
ValueError al arrancar	vLLM revisó la configuración contra el modelo y la GPU y algo no cierra: memoria, contexto, TP, cuantización. Los que salen de revisar la configuración llegan envueltos por pydantic: 1 validation error for ModelConfig (o VllmConfig) y, abajo, Value error, con el texto.	`Total number of attention heads (64) must be divisible by tensor parallel size (3).`
Envoltorios del EngineCore	El engine corre en otro proceso; si muere, ves un mensaje genérico y la causa queda más arriba.	`Engine core initialization failed. See root cause above.`
torch.OutOfMemoryError	PyTorch no consiguió memoria de GPU: otro proceso, gpu_memory_utilization alto o un modelo que no cabe.	`CUDA out of memory. Tried to allocate …`
NCCL WARN	La comunicación entre las GPUs de la réplica (TP=2) falló: /dev/shm, P2P o NVLink, o una GPU que murió.	`failed to extend /dev/shm/nccl-… to … bytes`
400 de vLLM (BadRequestError)	El request no entra: el contexto, un parámetro o un campo inválido. Repetirlo igual falla igual.	`This model's maximum context length is 8192 tokens…`
404 de vLLM (NotFoundError)	El model del request no es el --served-model-name de ese pod (ni un LoRA cargado).	`The model `chat-default` does not exist.`
Errores del cliente de openai	La clase dice el status; sin respuesta, APIConnectionError o APITimeoutError. Ya reintentó 2 veces.	`openai.RateLimitError: Error code: 429 - …`

PIEZA	QUÉ HACE	EJEMPLO
Errores del llm-gateway	La forma de OpenAI con su code: invalid_api_key, model_not_allowed, rate_limit_exceeded, overloaded, upstream_timeout.	`429: cuota de tokens por minuto agotada`
huggingface_hub	La descarga o la búsqueda del modelo en el Hub: repo gated, token, nombre o modo offline.	`GatedRepoError · LocalEntryNotFoundError`
finish_reason	No es un error HTTP: dice por qué terminó la respuesta: stop, length, abort, error o repetition.	`length = max_tokens o max_model_len`
Eventos del pod	Kubernetes: Pending sin GPU, probes que fallan, OOMKilled de la RAM del contenedor.	`Insufficient nvidia.com/gpu`

LOS 26 MENSAJES

`Value error, Model architectures ['…ForCausalLM'] are not supported for now. Supported architectures: … [type=value_error, …]`

vLLM no tiene implementación para la arquitectura que pide el config.json del modelo (el campo architectures), ni propia ni de respaldo. Pasa con modelos muy nuevos. Llega envuelto por pydantic (la librería con la que vLLM valida su configuración): la línea de arriba dice pydantic_core._pydantic_core.ValidationError: 1 validation error for ModelConfig. No es tu configuración: necesitas una versión de vLLM que la soporte, otro motor (SGLang, TensorRT-LLM) u otro modelo.

PRIMERO: `jq '.architectures, .model_type' config.json`

→ Sección 1 · Por qué servir un LLM es distinto y qué hace vLLM

`This model's maximum context length is 8192 tokens. However, you requested 1024 output tokens and your prompt contains at least 7169 input tokens, for a total of at least 8193 tokens. Please reduce the length of the input prompt or the number of requested output tokens.`

Un 400 de vLLM v0.30: reserva max_tokens dentro del contexto, y 7,500 + 1,024 no entran en 8,192. El gateway del C10 lo reenvía tal cual (es un 4xx: sin fallback). Desde v0.32 el mismo request se acepta y la salida se corta en 692 tokens, con finish_reason length.

PRIMERO:

```
curl -s localhost:8000/tokenize -H 'content-type: application/json' -d '{"model": "chat-70b-fp8-2026-09-28", "messages": [{"role": "user", "content": "…"}]}' | jq '.count, .max_model_len'
```

→ Sección 2 · Tokens y contexto: tokenizer, chat template, max_model_len, costo por token, español contra inglés

`As of transformers v4.44, default chat template is no longer allowed, so you must provide a chat template if the tokenizer does not define one.`

El tokenizer del modelo no trae plantilla de chat (el chat template: cómo se juntan system, user y assistant en un solo texto), y /v1/chat/completions no sabe armar el prompt. Típico de un modelo base, no instruct.

PRIMERO:

```
jq 'has("chat_template")' tokenizer_config.json; ls chat_template.jinja
```

→ Sección 2 · Tokens y contexto: tokenizer, chat template, max_model_len, costo por token, español contra inglés

`torch.OutOfMemoryError: CUDA out of memory. Tried to allocate … GiB. GPU 0 has a total capacity of … GiB of which … MiB is free. …`

PyTorch no consiguió memoria de GPU. Al arrancar: otro proceso usa la GPU, los pesos no caben con ese TP o la captura de CUDA graphs no entró. Con tráfico casi nunca es el KV: vLLM lo reservó entero al arrancar.

PRIMERO:

```
nvidia-smi --query-compute-apps=pid,used_memory --format=csv
```

→ Sección 3 · Memoria: pesos + KV cache, la fórmula con GQA, gpu_memory_utilization, bloques, max_num_seqs y preemption

`ValueError: Free memory on device … (…/… GiB) on startup is less than desired GPU memory utilization (0.9, … GiB). Decrease GPU memory utilization or reduce GPU memory used by other processes.`

Al arrancar, vLLM mide la memoria libre de cada GPU y exige que alcance para su fracción (0.90 × el total en el C10). Algo más ya la ocupa: un pod viejo, otro vLLM, un proceso colgado.

PRIMERO:

```
nvidia-smi --query-compute-apps=pid,used_memory --format=csv
```

→ Sección 3 · Memoria: pesos + KV cache, la fórmula con GQA, gpu_memory_utilization, bloques, max_num_seqs y preemption

`ValueError: To serve at least one request with the model's max seq len (131072), (… GiB KV cache is needed, which is larger than the available KV cache memory (… GiB). Based on the available memory, the estimated maximum model length is …. Try increasing `gpu_memory_utilization` (which also controls CPU memory on the CPU backend) or decreasing `max_model_len` when initializing the engine. See https://docs.vllm.ai/en/latest/configuration/conserving_memory/ for more details.`

El KV libre no alcanza para una sola secuencia del largo máximo. Casi siempre es no haber fijado --max-model-len: vLLM toma el del config.json (131,072 en los Llama 3.x), y un 8B en una L4 necesitaría 16 GiB de KV para una sola. El mensaje trae el largo que sí cabe. En el 70B del C10 una de 131,072 sí cabe (40 GiB), pero apenas ≈ 1.6 a la vez.

PRIMERO:

```
kubectl logs -n c10 deploy/vllm-70b -c vllm --previous | grep -E 'max seq len|Available KV cache memory'
```

→ Sección 3 · Memoria: pesos + KV cache, la fórmula con GQA, gpu_memory_utilization, bloques, max_num_seqs y preemption

`TimeoutError: Timed out waiting for engine core processes to start. This is often caused by slow weight loading for large models. Waited 600s (configured by VLLM_ENGINE_READY_TIMEOUT_S). To increase the timeout, set the environment variable: VLLM_ENGINE_READY_TIMEOUT_S=<seconds>`

Un proceso del API esperó 10 minutos a que el engine terminara de arrancar (cargar pesos, perfilar, compilar, capturar CUDA graphs) y se rindió. En v0.30 solo pasa cuando el API corre en procesos aparte (--api-server-count mayor que 1, o data parallel, que lo sube solo): con un único API server, como en el C10, vLLM espera al engine sin límite y el tope es la startupProbe (15 min). Causas típicas: pesos en un disco lento o torch.compile sin caché.

PRIMERO:

```
kubectl logs -n c10 deploy/vllm-70b -c vllm --previous | grep -E 'Model loading took|torch.compile took|Graph capturing'
```

→ Sección 4 · Arrancar vLLM: vllm serve y sus flags, la imagen de Docker, el manifest del C10 en Kubernetes (GPUs, shm, probes, caché del modelo, secreto de HF)

`RuntimeError: Engine core initialization failed. See root cause above. Failed core proc(s): {…}`

Un envoltorio: el engine corre en otro proceso y murió al arrancar. La causa real es la primera excepción más arriba en el log: un ValueError de configuración, un OutOfMemoryError, un error de NCCL.

PRIMERO:

```
kubectl logs -n c10 deploy/vllm-70b -c vllm --previous | grep -m1 -B2 -A8 -E 'Error|Exception'
```

→ Sección 4 · Arrancar vLLM: vllm serve y sus flags, la imagen de Docker, el manifest del C10 en Kubernetes (GPUs, shm, probes, caché del modelo, secreto de HF)

`Value error, User-specified max_model_len (262144) is greater than the derived max_model_len (max_position_embeddings=131072 or model_max_length=… in model's config.json). To allow overriding this maximum, set the env var VLLM_ALLOW_LONG_MAX_MODEL_LEN=1. … [type=value_error, …]`

Pediste un contexto más largo que el que el modelo dice soportar. vLLM se niega porque las posiciones de más dan salidas malas o errores de CUDA; VLLM_ALLOW_LONG_MAX_MODEL_LEN=1 lo permite, bajo tu riesgo. Llega envuelto por pydantic: arriba dice 1 validation error for ModelConfig.

PRIMERO: `jq '.max_position_embeddings, .rope_scaling' config.json`

→ Sección 4 · Arrancar vLLM: vllm serve y sus flags, la imagen de Docker, el manifest del C10 en Kubernetes (GPUs, shm, probes, caché del modelo, secreto de HF)

`The model `chat-default` does not exist.`

Un 404 de vLLM: el model del request no es el --served-model-name de ese pod (ni un LoRA cargado). Pasa si alguien llama a vLLM con el alias en vez de la revisión, o si un Service mezcla pods de dos revisiones (por eso el canario va en su propio Service).

PRIMERO: `curl -s localhost:8000/v1/models | jq -r '.data[].id'`

→ Sección 5 · La API compatible con OpenAI: chat completions, streaming SSE, parámetros de sampling, structured outputs, el cliente de openai

`openai.APIConnectionError: Connection error.`

El cliente de openai no pudo hablar con el servidor: nadie escucha en ese puerto, el nombre no resuelve o falló TLS. No hubo respuesta HTTP, así que no hay status. Antes de lanzarla ya reintentó 2 veces.

PRIMERO:

```
curl -sv "$OPENAI_BASE_URL/models" -H "Authorization: Bearer $OPENAI_API_KEY" 2>&1 | head -20
```

→ Sección 5 · La API compatible con OpenAI: chat completions, streaming SSE, parámetros de sampling, structured outputs, el cliente de openai

`openai.APITimeoutError: Request timed out.`

No llegó nada a tiempo: el default es 600 s entre bytes y 5 s para conectar. Si el destino descarta paquetes en silencio (una NetworkPolicy), también sale timeout y no Connection error. El gateway del C10 corta antes: 504 si vLLM no da el primer token en 10 s.

PRIMERO:

```
curl -s -o /dev/null -w 'conecta %{time_connect} s · primer byte %{time_starttransfer} s\n' "$OPENAI_BASE_URL/models"
```

→ Sección 5 · La API compatible con OpenAI: chat completions, streaming SSE, parámetros de sampling, structured outputs, el cliente de openai

`The engine is currently busy and cannot accept new requests. Please try again later or on a different instance.`

Un 503 de vLLM por su propio tope: solo pasa si alguien puso --max-num-queued-reqs (cuántos requests puede tener a la vez, esperando o corriendo; sin límite por defecto). El C10 no lo usa: la admisión vive en el gateway, que trataría este 503 como una falla del upstream (502 o el vendor).

PRIMERO:

```
kubectl get deploy vllm-70b -n c10 -o jsonpath='{.spec.template.spec.containers[0].args}'
```

→ Sección 6 · Continuous batching y el scheduler: prefill contra decode en el mismo batch, chunked prefill, max_num_batched_tokens, latencia contra throughput

`error: bench/tests/fixtures/p99-only.json: faltan ['p95_ttft_ms', 'p95_tpot_ms'] (hay ['p99_ttft_ms']). Corre vllm bench serve con --percentile-metrics ttft,tpot,itl,e2el --metric-percentiles 50,95,99`

Lo dice bench/pick_concurrency.py del C10 (sale con código 2): el JSON de vllm bench serve no tiene p95 porque el default de --metric-percentiles es solo 99. El script no reemplaza p95 por p99 en silencio.

PRIMERO:

```
jq -r 'keys[] | select(test("^p[0-9]+_(ttft|tpot)_ms$"))' results/sweep-c072.json
```

→ Sección 7 · Medir: TTFT, TPOT/ITL, E2E, tokens/s, goodput; métricas vllm: en Prometheus; el barrido de concurrencia del C10

`Value error, Total number of attention heads (64) must be divisible by tensor parallel size (3). [type=value_error, …]`

Con TP, cada GPU se queda con una parte de los heads de atención: los 64 heads (y 8 KV heads) del 70B se reparten parejo entre 1, 2, 4 u 8 GPUs de un nodo. Llega envuelto por pydantic: arriba dice 1 validation error for VllmConfig.

PRIMERO: `jq '.num_attention_heads, .num_key_value_heads' config.json`

→ Sección 8 · Paralelismo: tensor, pipeline y réplicas; NVLink contra PCIe; cuándo un 70B necesita TP

`NCCL WARN Error: failed to extend /dev/shm/nccl-03v824 to 4194660 bytes`

NCCL (la librería que comunica las GPUs de una réplica con TP=2) usa memoria compartida en /dev/shm y no le alcanzó: el /dev/shm de un contenedor es de 64 MiB por defecto. Según la doc de NCCL, así no logra inicializarse: vLLM falla al arrancar o se queda colgado esperando a la otra GPU.

PRIMERO:

```
kubectl exec -n c10 deploy/vllm-70b -c vllm -- df -h /dev/shm
```

→ Sección 8 · Paralelismo: tensor, pipeline y réplicas; NVLink contra PCIe; cuándo un 70B necesita TP

`Value error, The quantization method fp8 is not supported for the current GPU. Minimum capability: 75. Current capability: 70. [type=value_error, …]`

La GPU es más vieja de lo que pide el método. En vLLM v0.30, FP8 corre desde Turing (7.5) como pesos (W8A16) y nativo desde 8.9 (Ada, Hopper, Blackwell); una V100 (7.0) no puede. Llega envuelto por pydantic: arriba dice 1 validation error for VllmConfig.

PRIMERO: `nvidia-smi --query-gpu=name,compute_cap --format=csv`

→ Sección 9 · Cuantización: FP8, INT8, INT4 (AWQ y GPTQ), KV cache en FP8 y la calidad

`Using KV cache scaling factor 1.0 for fp8_e4m3. If this is unintended, verify that k/v_scale scaling factors are properly set in the checkpoint.`

Un warning al arrancar con --kv-cache-dtype fp8: el checkpoint cuantizado (como el FP8 del C10) no trae escalas para K y V (kv_cache_scheme: null) y vLLM usa 1.0. Corre, pero sin calibrar: mide la calidad con el golden set antes de dejarlo. Con un checkpoint sin cuantizar también usa 1.0, pero sin avisar.

PRIMERO: `jq '.quantization_config.kv_cache_scheme' config.json`

→ Sección 9 · Cuantización: FP8, INT8, INT4 (AWQ y GPTQ), KV cache en FP8 y la calidad

`LoRA rank 32 is greater than max_lora_rank 16.`

El adaptador tiene rank 32 y vLLM arrancó con el default de --max-lora-rank (16). vLLM reserva memoria para el rank máximo, así que no lo carga.

PRIMERO: `jq .r adapter_config.json`

→ Sección 10 · Optimizaciones: prefix caching, speculative decoding, LoRA múltiple, autoscaling por cola y cold start

`Skipping CUDA graph capture. To turn on CUDA graph capture, ensure `cudagraph_mode` was not manually set to `NONE``

Un warning: vLLM arrancó sin CUDA graphs porque cudagraph_mode quedó en NONE, casi siempre a mano (-cc.cudagraph_mode=none) o con -O0. Con --enforce-eager no sale este aviso sino Enforce eager set, disabling torch.compile and CUDAGraphs. En los dos casos arranca antes y decodifica más lento: en cada paso de decode la GPU espera a que la CPU lance los kernels.

PRIMERO:

```
kubectl get deploy vllm-70b -n c10 -o
jsonpath='{.spec.template.spec.containers[0].args}' | tr ',' '\n'
| grep -E 'eager|compilation|optimization|-O|-cc'
```

→ Sección 10 · Optimizaciones: prefix caching, speculative decoding, LoRA múltiple, autoscaling por cola y cold start

`openai.RateLimitError: Error code: 429 - {'error': {'message': 'cuota de tokens por minuto agotada', 'type': 'rate_limit_error', 'code': 'rate_limit_exceeded'}}`

Lo dice el llm-gateway del C10 (code rate_limit_exceeded): el balde de tokens del tenant en esa réplica no alcanza para el prompt estimado + max_tokens. Trae Retry-After; el cliente de openai ya reintentó 2 veces respetándolo.

PRIMERO:

```
sum by (tenant) (rate(llm_gateway_requests_total{status="429"}
[5m]))
```

→ Sección 11 · El gateway de LLMs del C10: cuotas en tokens, admisión por prioridad, streaming detrás de proxies, cancelación y PII

`{"error":{"message":"plataforma sin capacidad (queue_timeout)","type":"overloaded_error","code":"overloaded"}}`

Un 503 del gateway: no hubo lugar en vuelo y el request esperó más de 0.5 s (queue_timeout; queue_full = la cola ya estaba llena). Es la plataforma, no el tenant: cuenta en LLMGatewayErrorRate. Publicaciones iría al vendor; soporte recibe esto con Retry-After: 2.

PRIMERO: `sum(llm_gateway_inflight) / sum(llm_gateway_max_inflight)`

→ Sección 11 · El gateway de LLMs del C10: cuotas en tokens, admisión por prioridad, streaming detrás de proxies, cancelación y PII

`{"error":{"message":"el modelo no respondió a tiempo","type":"api_error","code":"upstream_timeout"}}`

Un 504 del gateway antes del primer byte: vLLM no dio el primer token en 10 s, no se pudo conectar en 2 s, o el pool HTTP no tuvo una conexión libre (PoolTimeout). Con el default de httpx2 (100 conexiones) pasaría desde el request 101 en vuelo de una réplica; el C10 usa max_connections=None.

PRIMERO: `max(llm:vllm_ttft_p95:rate5m)` · `sum(vllm:num_requests_waiting)`

→ Sección 11 · El gateway de LLMs del C10: cuotas en tokens, admisión por prioridad, streaming detrás de proxies, cancelación y PII

`GatedRepoError: 401 Client Error. … Cannot access gated repo for url https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct/resolve/main/config.json.`

huggingface_hub: el repo es gated (hay que pedir acceso y que lo aprueben). 401 = sin token; 403 = un token sin acceso aprobado o sin permiso de lectura.

PRIMERO:

```
curl -s -o /dev/null -w '%{http_code}\n' -H "Authorization:
Bearer $HF_TOKEN" https://huggingface.co/meta-llama/Llama-3.3-
70B-Instruct/resolve/main/config.json
```

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

`LocalEntryNotFoundError: An error happened while trying to locate the file on the Hub and we cannot find the requested files in the local cache. Please check your connection and try again or make sure your Internet connection is on.`

huggingface_hub no pudo preguntarle al Hub (sin red, o con HF_HUB_OFFLINE=1 como en el C10) y el archivo no está en la caché local. En el C10 quiere decir que vLLM recibió un nombre de repo en vez de la carpeta /models/….

PRIMERO:

```
kubectl get deploy vllm-70b -n c10 -o
jsonpath='{.spec.template.spec.containers[0].args[0]}'
```

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

`EngineDeadError: EngineCore encountered an issue. See stack trace (above) for the root cause.`

El engine murió con el servidor ya corriendo (un XID, o sea una falla de la GPU que reporta el driver; un error de CUDA; un bug). No se recupera: /health pasa a 503, la liveness reinicia el pod y los streams en curso se cortan (stream_error en el gateway).

PRIMERO:

```
kubectl logs -n c10 deploy/vllm-70b -c vllm --previous | grep -m1
-B5 -A20 Traceback
```

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

EJEMPLO REAL · CÓMO LEER UN 400 DE CONTEXTO, DEL PRODUCTO HASTA VLLM

TEXTO — Formatos reales de vLLM v0.30.0 (vllm/renderers/params.py y el cuerpo de error de vllm/entrypoints/serve/exception_handling) y de openai-python 3.37.0 (_base_client.py); la conversación y los números son ilustrativos

```
$ python3 producto.py          # el asistente de soporte, con el cliente de openai contra el llm-gateway
Traceback (most recent call last):
  …
openai.BadRequestError: Error code: 400 - {'error': {'message': "This model's maximum context length is 8192 tokens. However, you requested 1024 output tokens and your prompt contains at least 7169 input tokens, for a total of at least 8193 tokens. Please reduce the length of the input prompt or the number of requested output tokens.", 'type': 'BadRequestError', 'param': 'input_tokens', 'code': 400}}

# El cuerpo que devolvió vLLM v0.30.0 y que el gateway reenvió sin tocar (es un 4xx: no hay fallback):
{"error":{"message":"This model's maximum context length is 8192 tokens. However, you requested 1024 output tokens and your prompt contains at least 7169 input tokens, for a total of at least 8193 tokens. Please reduce the length of the input prompt or the number of requested output tokens.","type":"BadRequestError","param":"input_tokens","code":400}}

# Lo que quedó en el gateway: la muestra del counter y una línea de log, sin el prompt ni el mensaje
llm_gateway_requests_total{alias="chat-default",status="400",tenant="support-assistant"} 1.0
{"request_id": "9f2c41d07b8e4a6d8c13e5b2a7f06d91", "tenant": "support-assistant", "alias": "chat-default", "backend": "self", "status": "400", "error": "upstream_rejected", "prompt_tokens": null, "completion_tokens": null, "ttft_s": null, "duration_s": 0.0412}
```

- Léelo por partes: `you requested 1024 output tokens` es el max_tokens (aquí, el tope de soporte que puso el gateway porque el cliente no mandó uno); `at least 7169 input tokens` : vLLM deja de tokenizar un token después del límite (8,192 − 1,024 + 1 = 7,169), así que el prompt real, ya con la plantilla de chat, puede ser más largo (aquí, 7,500); `at least 8193` ya no entra en 8,192.
- ¿Por qué pasó la puerta del gateway? Estima caracteres ÷ 3 + 4 por mensaje: una conversación que pega una tabla de pedidos de ≈ 17,500 caracteres da ≈ 5,860 estimados (bajo el tope de 6,000), pero el tokenizer de Llama 3 cuenta los números de a 3 dígitos o menos y son 7,500 reales. La estimación pesimista falla justo con texto raro.

- Quién lo dijo: vLLM (`type` `BadRequestError` , `param` `input_tokens`). El gateway reenvía los 4xx tal cual y no prueba el vendor: un request inválido no se arregla mandándolo afuera. El cliente de openai no lo reintenta: solo reintenta 408, 409, 429 y 5xx.
- En vLLM v0.32 este mismo request ya no falla: max_tokens deja de reservarse y la salida se corta en 692 tokens (8,192 – 7,500), con finish_reason length. Un upgrade convierte un 400 claro en una respuesta corta, en silencio.
- El gateway lo cuenta y deja su línea JSON: status 400, error `upstream_rejected` , backend `self` , sin el texto del error. Su `request_id` es el header `x-request-id` de la respuesta que recibió el producto: con él se cruzan el log del producto y el del gateway.

CÓMO RECORDARLA

quién habla dice dónde mirar. Una línea `INFO … [archivo.py:N]` : el log de vLLM. Un `ValueError` al arrancar (o un `Value error, …` de pydantic): tu configuración contra el modelo y la GPU. `Engine core initialization failed` o `EngineCore encountered an issue` : la causa está más arriba. `openai.…Error: Error code: N` : el status que devolvió el gateway o vLLM. `NCCL WARN` : las GPUs entre sí. `GatedRepoError` : el Hub.

TRAMPA

quedarte con la última línea del traceback. En vLLM el engine corre en otro proceso, y `Engine core initialization failed. See root cause above` no es la causa: es el aviso de que la causa está arriba. La otra: confiar en textos de versiones viejas. En V1 no existe la línea `Sequence group … is preempted` de V0 (las preemptions solo salen en el counter y en `Preemptions: N`), y el 400 por max_tokens de v0.30 desaparece en v0.32.

En el curso: C10 · Archivos de referencia · C10 · Modos de falla y degradación · M14 · KV cache: la memoria que decide la concurrencia · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Herramientas · Stack de LLMs (cambia rápido)

Flags, defaults, endpoints, métricas y mensajes revisados en octubre de 2026 contra la documentación oficial de vLLM, su código en GitHub, la referencia de la API de OpenAI y los config.json de los modelos en Hugging Face, y contra los archivos de `reference/c10` del curso. Las versiones son las del C10 (vLLM v0.30.0 en la imagen; el gateway con fastapi 0.142.2, uvicorn 0.54.0, httpx2 2.13.1) y la actual de vLLM (v0.32.0); cuando la versión actual cambia algo, la sección lo dice. De los 15 laboratorios, los 15 se corrieron de verdad en un Mac, sin GPU. Si tu terminal dice otra cosa, créele a tu terminal.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error