Facsímil 03

# Arquitecturas y modelos

Capítulo 07

## Inferencia optimizada, edge AI y hardware

686f6c61 · 2026

# Capítulo 07: Inferencia optimizada, edge AI y hardware

## La IA también es esperar

Hay una escena muy común: eliges un modelo, haces una demo, todo parece brillante… y luego alguien lo prueba con diez usuarios a la vez. El primer token tarda demasiado. La respuesta completa llega tarde. La GPU se llena aunque el modelo «cabía». El portátil se calienta. La factura sube. El problema ya no es si el modelo sabe contestar; es si puede contestar **a tiempo, con memoria suficiente y a un coste razonable**.

Este capítulo trata de esa parte menos vistosa y absolutamente decisiva: la inferencia. En el capítulo anterior hablamos de adaptar, destilar, cuantizar y elegir modelos abiertos. Ahora preguntamos qué pasa cuando ese modelo se usa de verdad: cuánta memoria consume, qué limita la velocidad, qué pinta la caché KV, por qué el hardware importa y cuándo tiene sentido llevar IA al dispositivo.

La idea de fondo es sencilla: un modelo útil no es solo el que responde bien. Es el que responde bien **dentro de tus restricciones**.

## Estado del arte con fecha de corte

**Fecha de corte:** 10 de junio de 2026.
**Fuentes consultadas ese día:** papers de FlashAttention, PagedAttention/vLLM, speculative decoding, GQA y cuantización; documentación vigente de vLLM, TensorRT-LLM, llama.cpp y MLPerf Inference; y una lectura pedagógica de Turing Post sobre QKV y KV cache.

Esta sección no pretende congelar el mercado. Los nombres de runtimes, formatos y aceleradores cambian. Lo que sí es más estable son las tensiones técnicas: memoria, ancho de banda, lotes, caché KV, latencia, calidad y coste.

| CAPA DEL PROBLEMA | ESTADO OBSERVADO A 10 DE JUNIO DE 2026 | QUÉ DEBES MIRAR |
|---|---|---|
| **Kernels de atención** | FlashAttention y variantes IO-aware son referencia para reducir movimiento de memoria en atención exacta.[1] | Si tu runtime usa kernels modernos para tu GPU y tu longitud de contexto. |
| **KV cache** | PagedAttention y cachés paginadas reducen desperdicio y fragmentación al servir muchas peticiones.[2] | Memoria real con batch, contexto largo y usuarios simultáneos. |
| **Planificación de peticiones** | Continuous batching, chunked prefill, prefix caching y separación prefill/decode aparecen en runtimes modernos como vLLM y TensorRT-LLM.[3] | Latencia p50/p95, no solo tokens/s máximos. |
| **Decodificación** | Speculative decoding acelera cuando un modelo pequeño propone tokens que el grande puede aceptar.[4] | Si la tasa de aceptación compensa el coste del modelo auxiliar. |
| **Arquitectura del modelo** | MQA y GQA reducen el número de cabezas de clave/valor y, por tanto, la memoria de KV cache.[5] | No solo tamaño en parámetros: también número de capas, contexto y cabezas KV. |
| **Cuantización** | INT8, FP8, INT4, GPTQ, AWQ, SmoothQuant y GGUF conviven según hardware, runtime y objetivo.[6] | Calidad tras cuantizar, memoria, velocidad y soporte real del runtime. |
| **Edge AI** | llama.cpp/GGUF, MLX, ONNX Runtime, NPUs y GPUs integradas permiten casos locales, pero con límites térmicos y de memoria.[7] | Si necesitas privacidad, offline, baja latencia local o menor coste por uso. |
| **Benchmarking** | MLPerf Inference sigue siendo una referencia neutral para medir sistemas completos, no solo modelos.[8] | Comparar sistema completo: modelo, runtime, hardware, batch, precisión y latencia. |

La lectura corta: a fecha de corte, optimizar inferencia no significa tocar un botón. Significa coordinar modelo, caché, kernel, cuantización, scheduler, hardware y evaluación.

## Qué no es optimizar inferencia

Optimizar inferencia no es solo elegir un modelo más pequeño. Un modelo pequeño puede ser más rápido, sí, pero quizá no tenga la calidad mínima. Un modelo grande puede ser aceptable si usa buen batching, buena caché y cuantización adecuada. La pregunta no es «pequeño o grande», sino «qué combinación cumple calidad, latencia, memoria y coste».

Tampoco es medir una única petición. Si pruebas un prompt aislado en local, quizá veas 80 tokens/s. En producción, con varias peticiones simultáneas, contexto largo y respuestas de distinta longitud, la historia cambia. Importan la cola, los percentiles, el tamaño de lote, el tiempo hasta primer token y el comportamiento cuando el sistema se llena.

Y optimizar no es perseguir el número más alto de tokens por segundo. A veces un sistema con menos throughput bruto da mejor experiencia porque entrega antes el primer token. O porque no se cae cuando llegan peticiones largas. O porque mantiene calidad tras cuantizar.

---

## El bucle real: prefill y decode

Cuando escribes una petición a un LLM, desde fuera parece una sola acción: envías texto y el modelo responde. Por dentro no ocurre así. La inferencia tiene dos momentos con personalidad muy distinta.

Primero está el **prefill**. El modelo lee todo lo que ya existe: instrucciones del sistema, conversación previa, documentos recuperados, pregunta del usuario y cualquier otro token de contexto. En esta fase todavía no está «escribiendo» la respuesta. Está construyendo una representación interna de ese contexto y llenando la caché que usará después. Si el prompt es largo, el prefill pesa mucho. Si metes veinte páginas de documentación en contexto, el coste no aparece solo al final: aparece aquí.

Después viene el **decode**. Ahora el modelo ya ha leído el contexto y empieza a generar. Pero un LLM autoregresivo genera de forma secuencial: produce un token, lo añade al contexto, usa ese nuevo contexto para producir el siguiente, y así sucesivamente. Por eso generar 300 tokens no es como calcular una tabla de 300 celdas independientes; es una cadena donde cada eslabón depende del anterior.

Una forma humana de verlo: el prefill es leer el enunciado completo antes de contestar; el decode es dictar la respuesta palabra a palabra. Leer mucho puede retrasar el arranque. Dictar mucho puede alargar el final.

También hay una diferencia técnica importante: el prefill suele aprovechar mejor el paralelismo porque procesa muchos tokens de entrada juntos; el decode tiene menos margen, porque el token $t + 1$ depende de haber generado antes el token $t$. Por eso algunas mejoras optimizan el arranque de la respuesta y otras optimizan la velocidad de escritura. Si mezclamos ambas fases en una sola métrica, dejamos de ver dónde duele de verdad.

| FASE | QUÉ HACE | QUÉ SUELE LIMITAR |
|---|---|---|
| **Prefill** | Procesa todos los tokens de entrada y construye la KV cache. | Cómputo y longitud del contexto. |
| **Decode** | Genera tokens nuevos uno a uno usando la KV cache. | Memoria, ancho de banda y planificación. |

Esta separación explica por qué dos usuarios pueden sentir experiencias muy distintas con el mismo modelo. Un usuario con un prompt corto y una respuesta larga puede ver el primer token pronto, pero esperar bastante hasta el final. Otro usuario con un prompt enorme puede esperar mucho al principio aunque luego la respuesta salga a buen ritmo.

Piénsalo con situaciones concretas. En una asesoría, alguien pega un contrato de treinta páginas y pregunta «¿qué cláusulas debo revisar primero?». Ahí el prefill manda: el sistema tiene que leer mucho antes de empezar. En una app de escritura, alguien pide «redáctame una propuesta completa de dos páginas». El contexto puede ser corto, pero el decode pesa: hay que producir muchos tokens seguidos. En un panel de soporte, veinte personas preguntan cosas parecidas con las mismas instrucciones del sistema. Ahí no basta con un modelo rápido; importa reutilizar prefijos, organizar lotes y no desperdiciar KV cache.

Si el usuario manda una entrada de $n_{\text{entrada}}$ tokens y queremos generar $n_{\text{salida}}$ tokens, conviene separar lectura de contexto y generación.

**Ejemplo de fórmula.** Como calculadora de primer orden, no como benchmark universal, podemos escribir:

$$T_{\text{total}} \approx T_{\text{prefill}} + n_{\text{salida}} \cdot T_{\text{decode}}$$

| SÍMBOLO | SIGNIFICADO | EJEMPLO |
|---|---|---|
| $T_{\text{total}}$ | Tiempo total de respuesta. | 7,5 segundos. |
| $T_{\text{prefill}}$ | Tiempo de procesar el contexto inicial. | 0,85 segundos. |
| $n_{\text{salida}}$ | Tokens generados. | 300 tokens. |
| $T_{\text{decode}}$ | Tiempo medio por token generado. | 0,022 segundos/token. |

En producto solemos mirar dos métricas porque responden a preguntas distintas. La primera mide cuándo empieza la sensación de respuesta; la segunda mide cuánto tarda en completarse.

| MÉTRICA | PREGUNTA QUE RESPONDE |
|---|---|
| **TTFT** (*time to first token*) | ¿Cuánto tarda el usuario en ver que algo empieza? |
| **Tokens/s** | ¿A qué velocidad se completa la respuesta una vez empieza? |

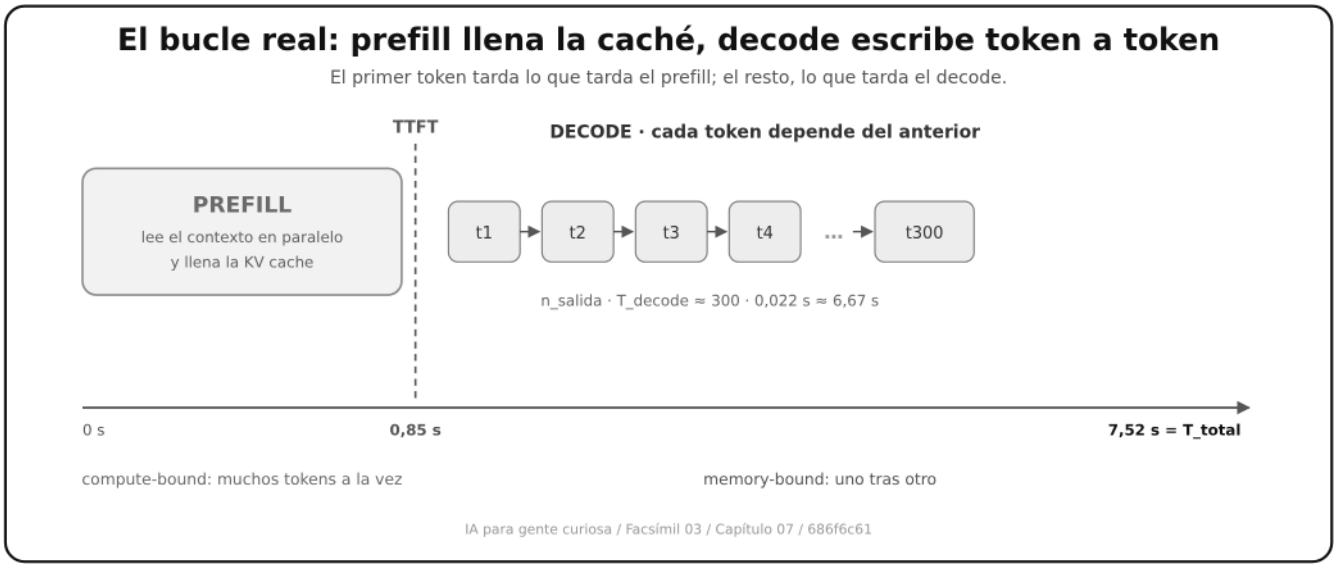
Con tokens por segundo, la misma idea se escribe así:

$$T_{\text{total}} \approx \frac{\text{TTFT}_{ms}}{1000} + \frac{n_{\text{salida}}}{\text{tokens/s}}$$

Ejemplo:

$$T_{\text{total}} \approx 0{,}85 + \frac{300}{45} \approx 7{,}52 \text{ s}$$

Esto explica una sensación habitual: el sistema puede «empezar rápido» pero terminar lento, o empezar lento pero luego escribir deprisa. Ambas cosas importan.




## La memoria invisible: KV cache

En los capítulos de atención vimos $Q$, $K$ y $V$. Durante inferencia, las claves y valores del contexto pasado se vuelven especialmente importantes. Si el modelo ya procesó los primeros 4096 tokens, no queremos recalcularlos enteros cada vez que genera un token nuevo. Sería como releer todo el libro antes de escribir cada palabra de un resumen.

La solución es guardar parte de ese trabajo en una **KV cache**. Esa caché permite que el decode consulte el pasado sin recomputarlo desde cero. A cambio, consume memoria. Y esa memoria crece con cuatro cosas que suelen crecer justo cuando el producto tiene éxito: más

usuarios simultáneos, más contexto, más capas y más cabezas de clave/valor.

Alyona Vert lo explica con una imagen muy clara: durante generación autoregresiva, las claves y valores de tokens anteriores se guardan para que el modelo no tenga que recalcular todo el pasado en cada nuevo token.[9] Esa lectura no sustituye los papers de inferencia, pero ayuda a que la intuición técnica aterrice.

**Ejemplo de fórmula.** Una aproximación de memoria para KV cache es:

$$M_{\mathrm{KV}} \approx 2 \cdot L \cdot B \cdot S \cdot H_{\mathrm{KV}} \cdot d_h \cdot \mathrm{bytes}$$

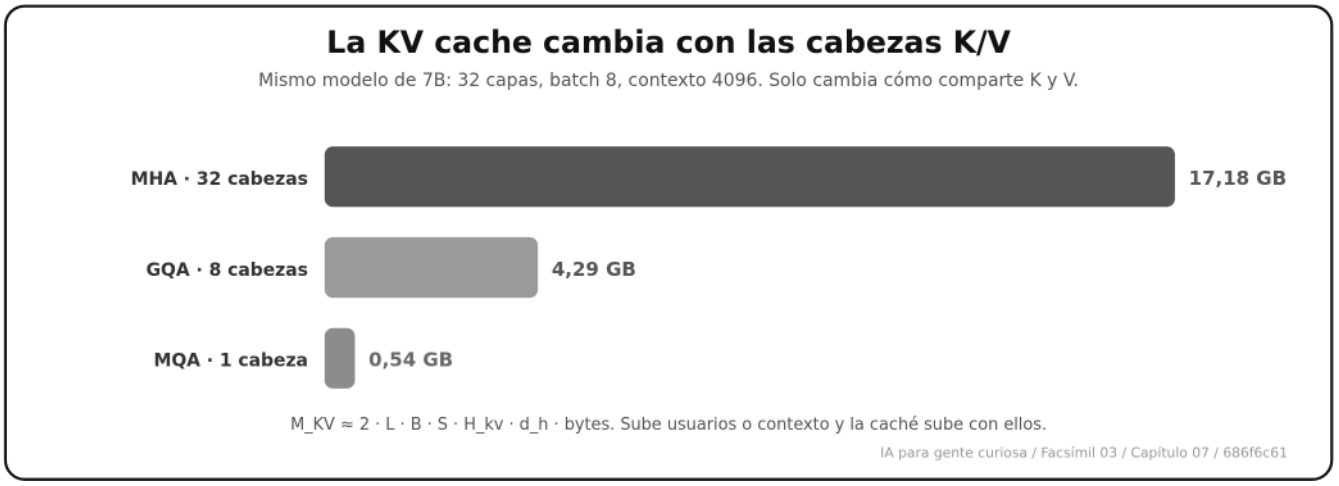
El factor 2 aparece porque guardamos claves y valores. $L$ son capas, $B$ es batch o conversaciones simultáneas, $S$ es longitud de contexto, $H_{\mathrm{KV}}$ son cabezas de clave/valor, $d_h$ es dimensión por cabeza y `bytes` depende de la precisión usada. Esta aproximación ayuda a detectar órdenes de magnitud; el consumo real depende del runtime, paginación de caché, fragmentación, buffers, activaciones, paralelismo y margen operativo.

| SÍMBOLO | SIGNIFICADO | EJEMPLO |
|---|---|---|
| $2$ | Guardamos claves y valores. | $K$ y $V$. |
| $L$ | Número de capas. | 32 capas. |
| $B$ | Batch o peticiones activas. | 8 peticiones. |
| $S$ | Tokens de contexto mantenidos. | 4096 tokens. |
| $H_{\mathrm{KV}}$ | Cabezas de clave/valor. | 32 en MHA, 8 en GQA, 1 en MQA. |
| $d_h$ | Dimensión por cabeza. | 128. |
| $\mathrm{bytes}$ | Bytes por valor. | 2 para FP16/BF16. |

Con $L = 32$, $B = 8$, $S = 4096$, $d_h = 128$ y 2 bytes por valor, el mismo modelo puede tener memorias de caché muy distintas según cómo organice sus cabezas $K/V$:

| ATENCIÓN | $H_{\mathrm{KV}}$ | KV CACHE APROXIMADA |
|---|---|---|
| Multi-head attention | 32 | 17,18 GB |
| Grouped-query attention | 8 | 4,29 GB |
| Multi-query attention | 1 | 0,54 GB |

Aquí aparece una lección importante: dos modelos con los mismos parámetros pueden tener costes de inferencia distintos si gestionan de forma distinta $K$ y $V$. Por eso GQA o MQA no son detalles académicos: afectan directamente a cuántas conversaciones caben a la vez.




Otro ejemplo sencillo: imagina un asistente de una universidad que responde dudas de matrícula. El modelo cabe en memoria cuando lo prueba una persona. Pero el primer lunes de septiembre entran cien estudiantes con conversaciones largas, documentos de normativa y respuestas a medio generar. Lo que antes era «el modelo cabe» se convierte en «el modelo más la caché de todas las conversaciones no cabe». La KV cache es ese coste que no se ve en la ficha del modelo, pero aparece justo cuando el sistema empieza a usarse de verdad.

---

## Por qué el decode va más lento: memory-bound y compute-bound

Hay una pregunta que da una intuición muy potente: cuando la GPU genera un token, ¿qué la frena, hacer cuentas o mover datos? La respuesta separa dos regímenes que el modelo roofline hizo famosos.[10]

En el **prefill**, la GPU procesa muchos tokens de entrada a la vez. Reutiliza cada peso del modelo para muchos tokens, así que hace muchas operaciones por cada byte que lee de memoria. Está **compute-bound**: el límite es la potencia de cálculo. La GPU trabaja a tope.

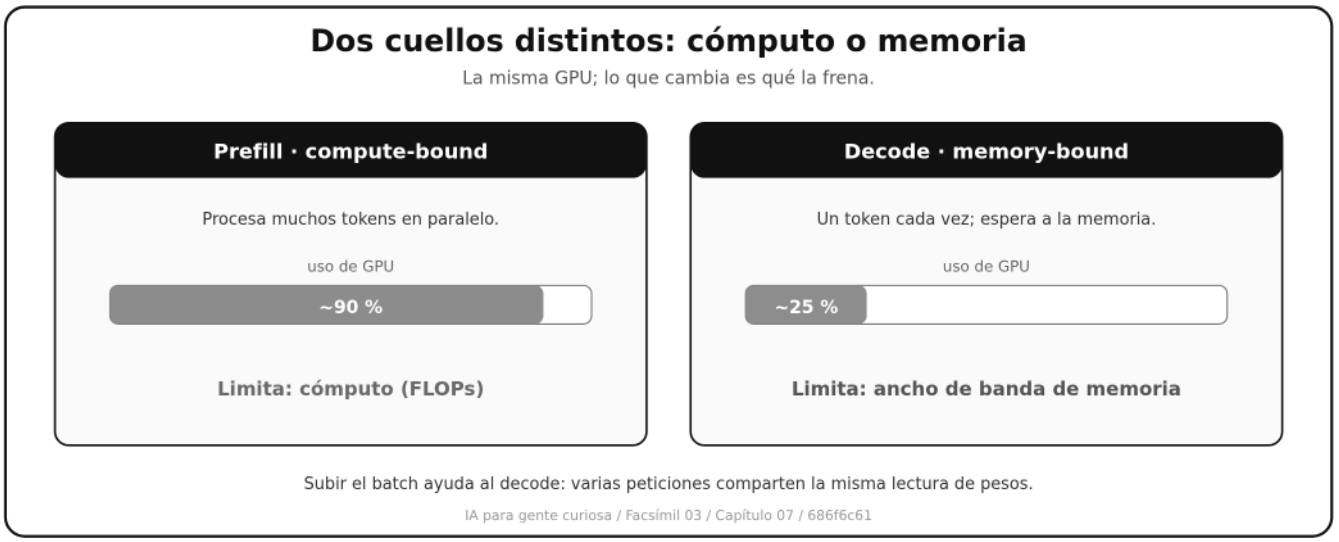
En el **decode**, la GPU genera un solo token cada vez. Para producirlo tiene que leer todos los pesos del modelo y toda la KV cache de memoria, y con esa lectura enorme hace muy pocas operaciones. Está **memory-bound**: el límite es el ancho de banda de memoria, no el cálculo. La GPU pasa buena parte del tiempo esperando datos, medio vacía.

| FASE | RÉGIMEN | QUÉ LA FRENA | POR QUÉ |
|---|---|---|---|
| **Prefill** | Compute-bound | Cómputo (FLOPs) | Muchos tokens por lectura de pesos: mucha cuenta por byte. |
| **Decode** | Memory-bound | Ancho de banda de memoria | Un token por lectura de todos los pesos: poca cuenta por byte. |

Esta idea reordena toda la lista de técnicas. Si el decode está limitado por mover datos, entonces:

- **Subir el batch** ayuda mucho: varias peticiones comparten la misma lectura de pesos, así que cada byte leído sirve para más tokens.
- **Cuantizar** ayuda al decode aunque no cambie la calidad: menos bits por peso son menos bytes que mover.
- **Speculative decoding** ayuda: el modelo grande verifica varios tokens propuestos en una sola pasada de lectura, en vez de uno por pasada.
- **GQA y MQA** ayudan: menos cabezas K/V son menos KV cache que leer en cada token.

No es casualidad que casi todas las optimizaciones de decode ataquen la memoria, no el cálculo. Saber en qué régimen estás te dice qué palanca mover.




## Qué acelera de verdad

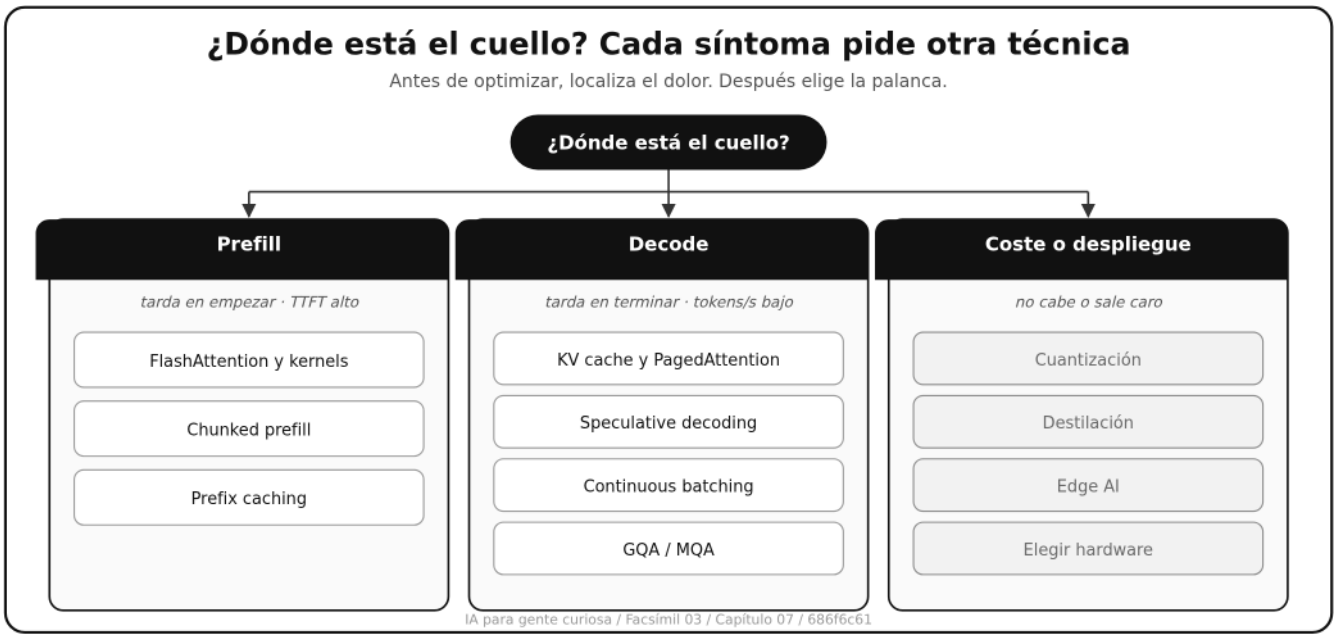
La inferencia no tiene un único cuello de botella. A veces el problema es que la atención mueve demasiados datos entre memorias. A veces la caché KV está fragmentada o ocupa demasiado. A veces la GPU está medio vacía porque las peticiones llegan en tamaños raros. A

veces el decode es lento porque cada token depende del anterior. Y a veces el modelo simplemente no cabe.

Por eso las técnicas de optimización no son intercambiables. Cada una toca una parte distinta del sistema. Antes de elegir una, conviene preguntar: ¿mi problema está en leer el contexto, en generar tokens, en servir muchos usuarios, en memoria, en coste o en calidad?

| TÉCNICA | QUÉ MEJORA | INTUICIÓN |
|---|---|---|
| **FlashAttention** | Atención más rápida y con menos memoria intermedia. | No cambia la fórmula; cambia cómo se mueven los datos. |
| **PagedAttention** | Gestión de KV cache con menos desperdicio. | Trata la caché como páginas, no como bloques rígidos enormes. |
| **Continuous batching** | Throughput con muchas peticiones. | Entra y sale gente del lote sin esperar a que todas las respuestas terminen. |
| **Prefix caching** | Peticiones con prefijos repetidos. | Si muchas consultas comparten sistema o contexto, no recalculas todo. |
| **Speculative decoding** | Menos tiempo por tokens generados. | Un modelo pequeño propone; el grande verifica. |
| **Cuantización** | Memoria y a veces velocidad. | Menos bits por peso o activación. |
| **GQA/MQA** | Memoria de KV cache. | Menos cabezas $K/V$ compartidas entre cabezas de consulta. |
| **Distilación** | Tamaño y coste. | Un estudiante más pequeño imita una parte útil del profesor. |

Si el cuello está en el **prefill**, suelen importar kernels de atención, longitud de contexto, chunked prefill y reutilización de prefijos. Si el cuello está en el **decode**, suelen importar KV cache, speculative decoding, batching y ancho de banda de memoria. Si el cuello está en **coste o despliegue**, entran cuantización, destilación, edge y elección de hardware.




No todas se suman limpiamente. Una técnica puede mejorar throughput y empeorar TTFT. Otra puede funcionar bien en una GPU y no tener soporte real en otra. Por eso el estado del arte no se adopta leyendo una lista: se adopta midiendo tu caso.

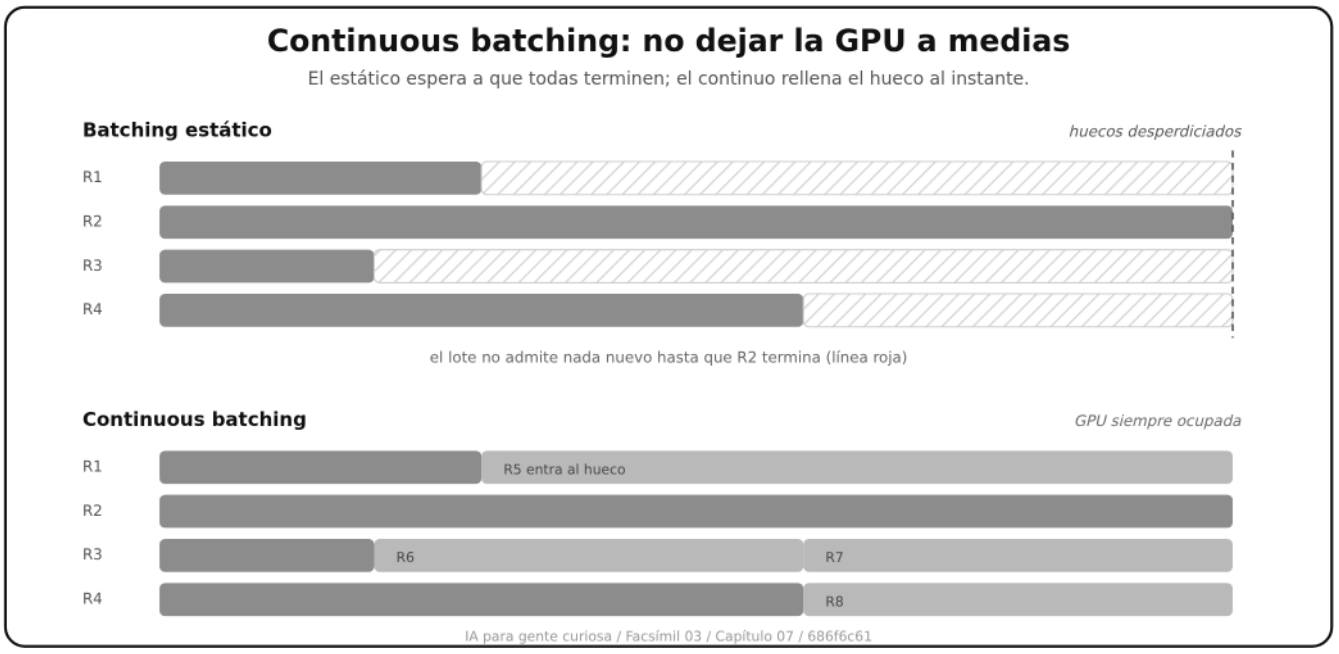
En una empresa pequeña, por ejemplo, puede parecer natural pasar de una API a un modelo local para ahorrar. Si cada consulta usa un prompt enorme con políticas internas, quizá el ahorro se pierde porque el prefill consume mucho. Si las consultas comparten el mismo contexto base, prefix caching puede cambiar la película. Si el problema es que cada respuesta debe ser larga, speculative decoding puede ayudar más que seguir recortando bits. La técnica correcta depende de dónde está el dolor.

---

## Continuous batching: no dejar la GPU a medias

El batching clásico junta varias peticiones en un lote y las procesa a la vez. El problema aparece cuando tienen longitudes distintas: el lote no termina hasta que acaba la respuesta más larga, y mientras tanto las peticiones que ya terminaron dejan su sitio vacío. La GPU sigue ocupada, pero haciendo trabajo inútil.

El **continuous batching** (también llamado *in-flight batching*) rompe esa rigidez. En vez de esperar a que todo el lote termine, cuando una petición acaba, su hueco se rellena al instante con otra que estaba en cola. La GPU no espera: siempre tiene tokens útiles que generar. Por eso es una de las mayores ganancias de throughput en runtimes modernos como vLLM y TensorRT-LLM, y suele importar más que cambiar de modelo cuando el cuello es servir muchas peticiones a la vez.



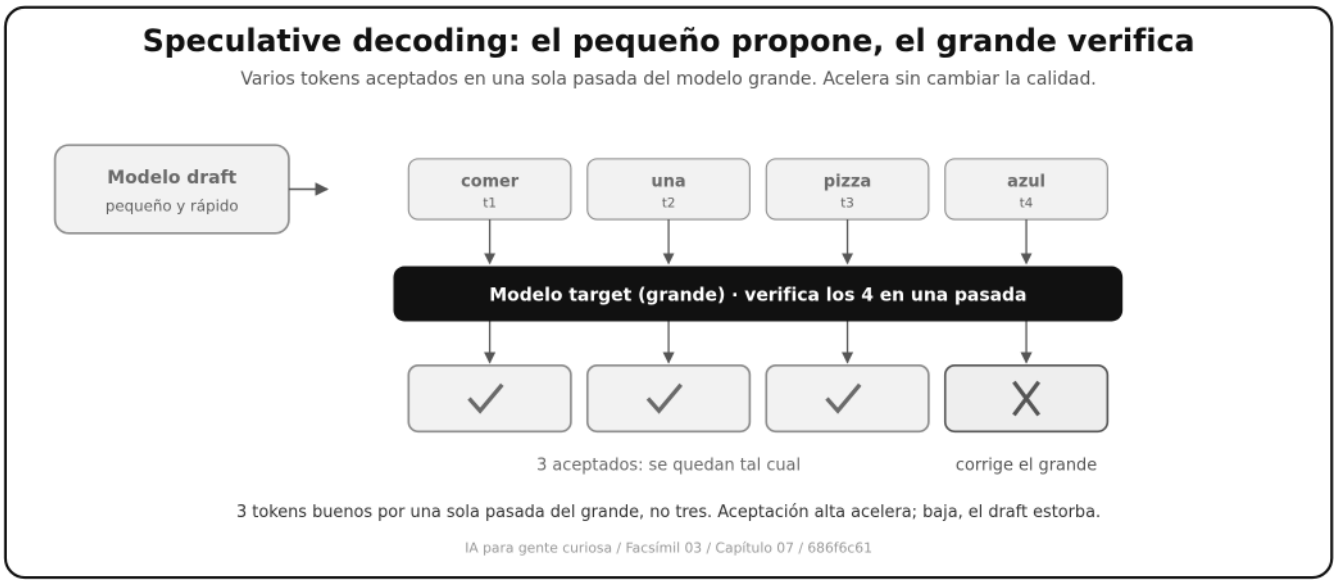

El detalle fino: continuous batching mejora mucho el throughput, pero hay que vigilar el TTFT. Si el sistema mete demasiadas peticiones nuevas en mitad del decode de otras, el primer token de alguien puede retrasarse. Por eso los runtimes combinan esta idea con chunked prefill y límites de lote: el objetivo no es solo «más tokens por segundo», sino un buen equilibrio entre throughput y latencia percibida.

---

## Speculative decoding: proponer y verificar

El decode es lento porque es secuencial: un token por pasada del modelo grande. La idea del **speculative decoding** es romper esa secuencialidad sin cambiar el resultado. Un modelo pequeño y barato (el *draft*) propone varios tokens de golpe; el modelo grande (el *target*) los verifica todos en una sola pasada, en paralelo. Acepta el tramo inicial que coincide con lo que él habría generado y descarta el resto, corrigiendo el primer fallo con su propio token.

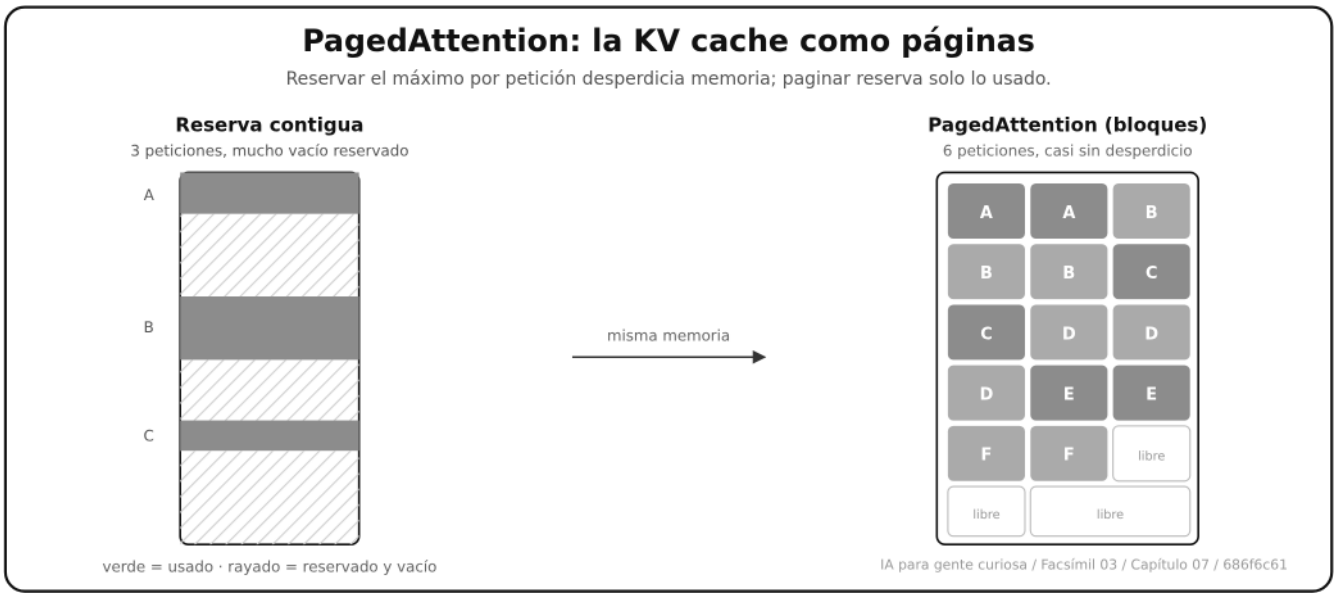
La clave es que verificar varios tokens en paralelo cuesta casi lo mismo que generar uno, porque el decode es memory-bound: la lectura de pesos ya estaba pagada. Si el draft acierta tres de cada cuatro, avanzas tres tokens por pasada del grande en vez de uno. No mejora la calidad (la distribución final es la del target); mejora la velocidad, y solo si la **tasa de aceptación** compensa el coste de ejecutar el draft.




## PagedAttention: la KV cache como páginas

La KV cache tiene un problema de gestión de memoria, no solo de tamaño. La forma ingenua de reservarla es dar a cada petición un bloque contiguo del tamaño de su contexto máximo, por si crece. Pero la mayoría de respuestas no llegan a ese máximo, así que queda mucha memoria reservada y vacía: fragmentación interna. Con esa reserva pesimista caben pocas conversaciones a la vez.

**PagedAttention**, la idea sobre la que se construyó vLLM, aplica algo que los sistemas operativos llevan décadas haciendo con la memoria virtual: dividir la KV cache en bloques pequeños de tamaño fijo (páginas) y asignarlos bajo demanda según crece cada secuencia. No hace falta memoria contigua ni reservar el máximo por adelantado. El desperdicio baja casi a cero y, en la misma GPU, caben muchas más peticiones simultáneas.



## Modo ingeniero: el presupuesto de serving en código

Las tres cuentas del capítulo (memoria de pesos, KV cache y latencia con concurrencia) caben en unas líneas.

```
# 1) Memoria de pesos: N * bits / 8
N = 7_000_000_000
for bits in (16, 8, 4):
    gb = N * bits / 8 / 1e9
    print(bits, "bits ->", round(gb, 1), "GB")     # 14.0, 7.0, 3.5

# 2) KV cache: 2 * L * B * S * H_kv * d_h * bytes
L, B, S, d_h, byts = 32, 8, 4096, 128, 2
for nombre, h_kv in {"MHA": 32, "GQA": 8, "MQA": 1}.items():
    gb = 2 * L * B * S * h_kv * d_h * byts / 1e9
    print(nombre, round(gb, 2), "GB")               # 17.18, 4.29, 0.54

# 3) Latencia: TTFT + tokens_salida / (tokens_por_segundo)
ttft_s, n_out = 0.85, 300
solo = ttft_s + n_out / 45                          # 7.52 s, un usuario
capacidad, usuarios = 180, 12
compartido = ttft_s + n_out / (capacidad / usuarios) # 20.85 s, capacidad repartida
print(round(solo, 2), round(compartido, 2))
```

Tres lecciones en un vistazo. Los pesos bajan con los bits, sí, pero la KV cache de MHA (17 GB) puede pesar más que los propios pesos en 16 bits (14 GB). Y la latencia que siente un usuario aislado (7,5 s) casi se triplica (20,85 s) cuando doce comparten la misma capacidad. Cambia `h_kv`, `S`, `B` o `usuarios` y verás moverse justo lo que duele en producción.

## Edge AI: cuando el modelo vive cerca

Edge AI significa ejecutar el modelo cerca del usuario, del sensor o del dispositivo: portátil, móvil, mini PC, servidor local, navegador, fábrica, aula o consulta. No siempre es mejor. Es una decisión de producto e infraestructura.

La pregunta no es «¿puedo correrlo local?», sino «¿qué gano al correrlo local y qué pierdo?». A veces ganas privacidad y respuesta sin red. A veces pierdes calidad, facilidad de actualización o estabilidad térmica. La tabla ayuda a separar motivos razonables de entusiasmo tecnológico.

| MOTIVO | POR QUÉ PUEDE INTERESAR |
|---|---|
| Offline | El sistema sigue funcionando sin red. |
| Latencia local | Evitas ida y vuelta a un servidor lejano. |
| Control de datos | Algunos datos no salen del dispositivo o de la red local. |
| Coste marginal | Muchas inferencias locales pueden salir más baratas que llamadas remotas. |
| Personalización | El sistema puede adaptarse al entorno concreto. |

Pero edge también trae límites:

| LÍMITE | QUÉ IMPLICA |
|---|---|
| Memoria | El modelo debe caber junto con KV cache y aplicación. |
| Energía | Un portátil o móvil no puede sostener carga alta indefinidamente. |
| Temperatura | El rendimiento puede bajar si el dispositivo se calienta. |
| Runtimes | No todos los formatos corren igual en CPU, GPU, NPU o Apple Silicon. |
| Mantenimiento | Actualizar modelos locales puede ser más complejo que cambiar una API. |

Una NPU no convierte cualquier modelo en viable. Una GPU no siempre gana. Una CPU moderna con un modelo cuantizado puede ser suficiente para una tarea pequeña. La pregunta buena es: ¿qué calidad necesito, con qué latencia, para cuántos usuarios, durante cuánto tiempo y con qué presupuesto térmico?

---

## Un caso cercano: el asistente de almacén

Imagina una empresa con un almacén grande. Quiere un asistente que lea incidencias de pedidos y proponga la siguiente acción: reponer, avisar, revisar lote, generar etiqueta o pedir foto.

Hay tres diseños posibles:

| DISEÑO | VENTAJA | COSTE |
|---|---|---|
| API externa potente | Calidad alta y mantenimiento simple. | Depende de red y coste por uso. |
| Servidor local con GPU | Control y buen rendimiento para muchos puestos. | Compra, operación y monitorización. |
| Modelo cuantizado en cada terminal | Funciona incluso con mala red. | Menor calidad y límites de memoria/temperatura. |

No hay una respuesta universal. Si el almacén necesita funcionar sin conexión, edge pesa mucho. Si la tarea cambia cada semana y necesita consultar políticas vivas, RAG y servidor central pueden ser más naturales. Si hay muchas peticiones simultáneas, la clave quizá no sea el modelo, sino batching, KV cache y colas.

Ahora bajémoslo un poco más. Si cada terminal manda una foto convertida a descripción, el número de tokens de entrada puede ser pequeño y el decode de la acción recomendada domina poco. Un modelo local pequeño puede bastar. Si cada incidencia arrastra historial de cliente, condiciones de entrega, normativa interna y correos anteriores, el prefill se vuelve caro y conviene pensar en servidor, caché de prefijos y recuperación selectiva. Si el turno de mañana genera cientos de incidencias a la vez, el problema ya no es «qué modelo responde mejor», sino cómo se ordenan las peticiones para que ninguna persona espere demasiado.

## En el día a día

**En una aplicación de soporte.** El usuario no mide tu media. Siente la espera. Si el primer token tarda cinco segundos, la experiencia parece rota aunque luego escriba rápido. TTFT importa tanto como tokens/s.

**En un dashboard interno.** Muchas personas lanzan consultas parecidas por la mañana. Prefix caching y batching pueden valer más que cambiar de modelo, porque el cuello está en repetir contexto y servir picos.

**En un producto local.** Un modelo GGUF de 4 bits puede ser perfecto para resumir notas privadas en un portátil, pero quizá no sirva para razonamiento largo. La decisión no es ideológica; es medición.

**En una demo ejecutiva.** Una sola petición luce bien. Lo profesional es enseñar también qué pasa con 20 peticiones, contexto largo y una respuesta de 500 tokens. Ahí aparece la verdad del sistema.

## Por qué debería importarte

Porque la arquitectura se vuelve coste. El número de capas, la longitud de contexto, el tipo de atención, los bits de precisión, la KV cache y el runtime acaban convertidos en euros, segundos, vatios y temperatura.

También porque la optimización sin evaluación puede engañar. Si cuantizas y el modelo responde más rápido pero falla justo en los casos delicados, no has optimizado: has movido el problema. Si subes batch y mejora throughput pero empeora p95, quizá has hecho feliz al benchmark y triste al usuario.

## Dónde solía tropezar yo

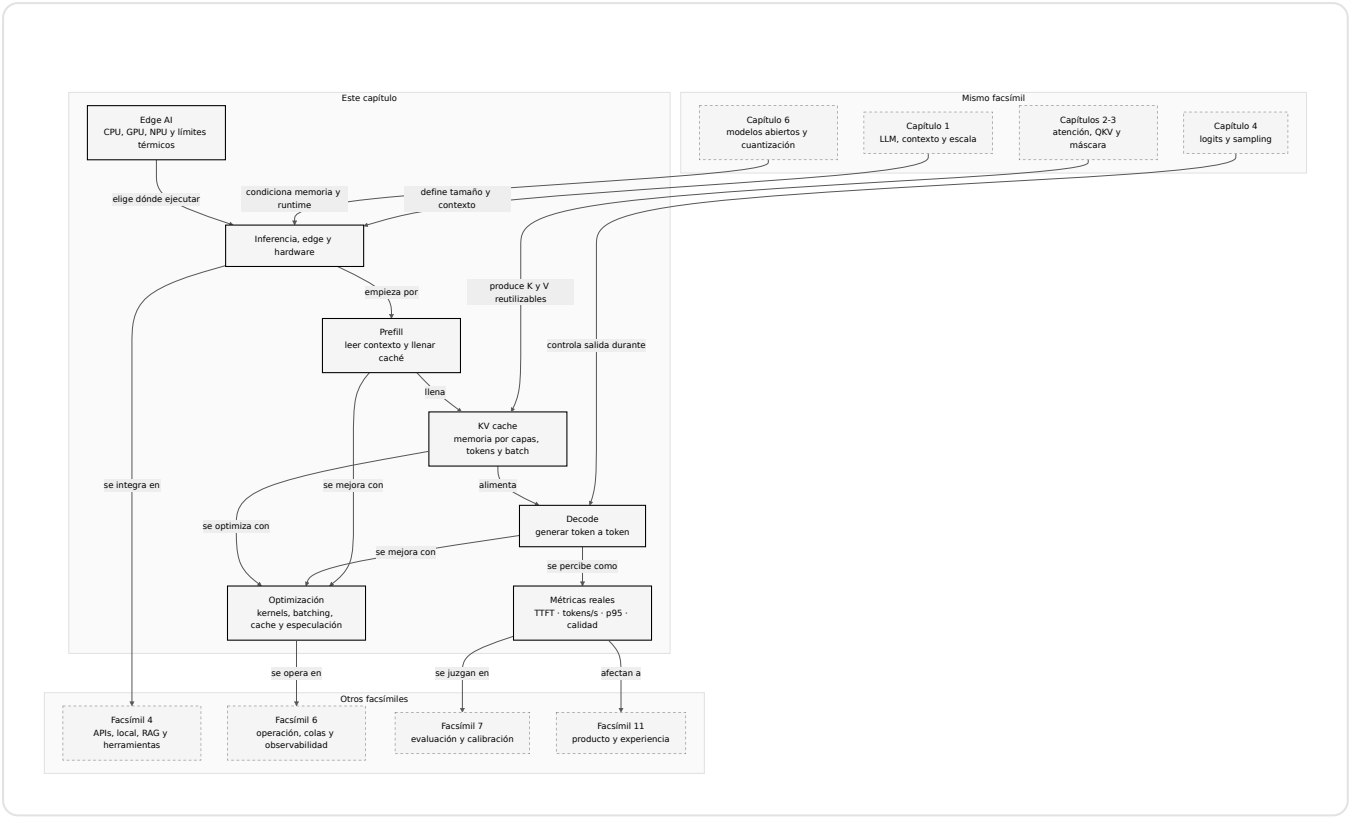
Estos son los tropiezos que más distorsionan decisiones reales. Casi todos tienen la misma raíz: mirar una métrica aislada y olvidarse del sistema completo.

| ERROR | POR QUÉ ES UN ERROR | ANTÍDOTO |
|---|---|---|
| **Mirar solo parámetros** | Dos modelos de 7B pueden tener inferencias muy distintas por contexto, GQA, runtime y cuantización. | Mide pesos, KV cache, TTFT y tokens/s. |
| **Confundir tokens/s con buena experiencia** | El usuario nota mucho el primer token y los percentiles altos. | Mide TTFT, latencia total y p95/p99. |
| **Cuantizar sin comparar calidad** | Menos bits pueden romper justo los casos que importan. | Crea una evaluación antes y después. |
| **Olvidar la KV cache** | El modelo cabe, pero las conversaciones simultáneas no. | Calcula memoria de pesos y memoria de KV cache. |
| **Creer que edge siempre es mejor** | Local puede ser lento, caliente o difícil de actualizar. | Decide por caso: offline, datos, coste, latencia y mantenimiento. |

## Cómo encaja todo

Este mapa conecta inferencia con los capítulos anteriores y con la operación futura. QKV deja de ser una fórmula: se vuelve caché. Sampling deja de ser un parámetro: se vuelve experiencia de usuario. Cuantización deja de ser una tabla de bits: se vuelve memoria, calidad y coste.

La decisión que prepara el siguiente facsímil es clara: no basta con elegir modelo; hay que elegir dónde vive, cómo se sirve, qué latencia promete y cómo se mide.




## Vocabulario aprendido

Nos quedamos con las palabras que conviene tener a mano para leer documentación de runtimes, benchmarks y hardware sin perder el hilo.

| TÉRMINO | DEFINICIÓN |
|---|---|
| **Inferencia** | Uso de un modelo entrenado para responder a una entrada nueva. |
| **Prefill** | Fase que procesa el contexto inicial antes de generar tokens. |
| **Decode** | Fase que genera tokens de salida uno a uno. |
| **TTFT** | Tiempo hasta recibir el primer token visible. |
| **Throughput** | Trabajo procesado por unidad de tiempo, por ejemplo tokens/s. |
| **KV cache** | Caché de claves y valores de atención para reutilizar el contexto pasado. |
| **FlashAttention** | Kernel de atención que reduce movimiento de memoria. |
| **PagedAttention** | Gestión paginada de KV cache para servir muchas peticiones con menos desperdicio. |
| **Continuous batching** | Técnica para mezclar peticiones activas y mejorar uso del hardware. |
| **Speculative decoding** | Método donde un modelo pequeño propone tokens y el grande los verifica. |
| **Cuantización** | Reducir precisión numérica para ahorrar memoria y, a veces, ganar velocidad. |
| **Edge AI** | Ejecutar modelos cerca del usuario o dispositivo. |
| **NPU** | Acelerador especializado para redes neuronales. |
| **Compute-bound** | Régimen limitado por el cálculo; típico del prefill. |
| **Memory-bound** | Régimen limitado por el ancho de banda de memoria; típico del decode. |

## Antes de pasar página

- [ ] ¿Puedo explicar la diferencia entre prefill y decode? (Si no, vuelve a «El bucle real».)
- [ ] ¿Entiendo por qué la KV cache puede ocupar varios GB? (Si no, vuelve a «La memoria invisible».)
- [ ] ¿Sé distinguir cuándo el decode es memory-bound y el prefill compute-bound? (Si no, vuelve a «Por qué el decode va más lento».)
- [ ] ¿Sé distinguir TTFT, tokens/s y throughput? (Si no, vuelve a «El bucle real».)

- [ ] ¿Entiendo por qué el continuous batching mantiene la GPU ocupada frente al batching estático? (Si no, vuelve a «Continuous batching».)
- [ ] ¿Sé cómo el speculative decoding acelera sin cambiar la calidad y por qué importa la tasa de aceptación? (Si no, vuelve a «Speculative decoding».)
- [ ] ¿Puedo explicar por qué PagedAttention desperdicia menos memoria que la reserva contigua? (Si no, vuelve a «PagedAttention».)
- [ ] ¿He reproducido en código la memoria de pesos, la KV cache y la latencia con concurrencia? (Si no, vuelve a «Modo ingeniero».)
- [ ] ¿Puedo explicar por qué GQA reduce memoria de inferencia? (Si no, vuelve a la tabla de KV cache.)
- [ ] ¿Sé por qué cuantizar no basta sin medir calidad? (Si no, vuelve a «Dónde solía tropezar yo».)
- [ ] ¿Puedo decidir cuándo edge AI tiene sentido? (Si no, vuelve a «Edge AI».)
- [ ] ¿Sé cómo cambian memoria y latencia al variar batch, longitud de secuencia y número de usuarios? (Si no, vuelve a «Modo ingeniero: el presupuesto de serving en código».)

## En resumen

La versión corta del capítulo no es una lista de herramientas: es una forma de pensar. Primero entiende qué fase estás pagando, después mide y solo entonces optimiza.

| IDEA FUERZA | DETALLE |
|---|---|
| Inferir no es una llamada simple. | Hay prefill, decode, KV cache, scheduler, kernels y hardware. |
| La memoria no son solo pesos. | La KV cache puede dominar cuando hay contexto largo y usuarios simultáneos. |
| Optimizar es medir compromisos. | TTFT, tokens/s, p95, calidad, coste y temperatura cuentan juntos. |
| Edge AI es una decisión de producto. | Sirve cuando offline, datos, latencia o coste local compensan sus límites. |
| El estado del arte tiene fecha. | A 10 de junio de 2026, las prácticas cambian rápido; el mecanismo importa más que la marca concreta. |

## Para saber más


Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. https://arxiv.org/abs/2305.13245

Dao, T., Fu, D. Y., Ermon, S., Rudra, A. y Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2205.14135

Dettmers, T., Lewis, M., Belkada, Y. y Zettlemoyer, L. (2022). LLM.int8(): 8-bit matrix multiplication for Transformers at scale. *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2208.07339

Frantar, E., Ashkboos, S., Hoefler, T. y Alistarh, D. (2022). *GPTQ: Accurate post-training quantization for generative pre-trained Transformers*. https://arxiv.org/abs/2210.17323

ggml-org. (2026). *llama.cpp: LLM inference in C/C++*. Consultado el 10 de junio de 2026. https://github.com/ggml-org/llama.cpp

Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. https://arxiv.org/abs/2309.06180

Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast inference from Transformers via speculative decoding. *Proceedings of ICML*. https://arxiv.org/abs/2211.17192

Lin, J. y otros (2024). AWQ: Activation-aware weight quantization for LLM compression and acceleration. *Proceedings of Machine Learning and Systems*. https://arxiv.org/abs/2306.00978

MLCommons. (2026). *MLPerf Inference: Datacenter benchmark*. Consultado el 10 de junio de 2026. https://mlcommons.org/benchmarks/inference-datacenter/

NVIDIA. (2026). *TensorRT-LLM documentation*. Consultado el 10 de junio de 2026. https://docs.nvidia.com/tensorrt-llm/index.html

vLLM Project. (2026). *vLLM documentation*. Consultado el 10 de junio de 2026. https://docs.vllm.ai/en/stable/

Vert, A. (2026, 13 de mayo). *AI 101: Your Ultimate Guide to Attention: Mechanism, QKV, and KV Cache*. Turing Post. https://www.turingpost.com/p/your-ultimate-guide-to-attention-mechanism-qkv-and-kv-cache

Xiao, G. y otros (2023). SmoothQuant: Accurate and efficient post-training quantization for large language models. *Proceedings of ICML*. https://arxiv.org/abs/2211.10438

## Notas

1. Dao, T., Fu, D. Y., Ermon, S., Rudra, A. y Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2205.14135. La idea central es optimizar lecturas y escrituras entre memorias rápidas y lentas, no cambiar la matemática de la atención. ↩

2. Kwon, W. y otros (2023). Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*. https://arxiv.org/abs/2309.06180. vLLM se construye sobre esta idea para gestionar KV cache con menos memoria desperdiciada. ↩

3. vLLM Project. (2026). *vLLM documentation*. https://docs.vllm.ai/en/stable/. Consultado el 10 de junio de 2026. La documentación lista serving online/offline, prefix caching, speculative decoding, cuantización y métricas de producción. ↩

4. Leviathan, Y., Kalman, M. y Matias, Y. (2023). Fast inference from Transformers via speculative decoding. *Proceedings of ICML*. https://arxiv.org/abs/2211.17192. El método aprovecha un modelo auxiliar para proponer varios tokens y verificar en paralelo con el modelo principal. ↩

5. Ainslie, J. y otros (2023). GQA: Training generalized multi-query Transformer models from multi-head checkpoints. *Proceedings of EMNLP*. https://arxiv.org/abs/2305.13245. GQA ocupa un punto intermedio entre multi-head attention y multi-query attention. ↩

6. Dettmers, T., Lewis, M., Belkada, Y. y Zettlemoyer, L. (2022). LLM.int8(): 8-bit matrix multiplication for Transformers at scale. *Advances in Neural Information Processing Systems 35*. https://arxiv.org/abs/2208.07339. El trabajo muestra una ruta para inferencia 8-bit en modelos grandes cuidando valores atípicos. ↩

7. ggml-org. (2026). *llama.cpp: LLM inference in C/C++*. https://github.com/ggml-org/llama.cpp. Consultado el 10 de junio de 2026. El proyecto documenta GGUF, cuantización, servidor compatible con OpenAI y herramientas de conversión. ↩

8. MLCommons. (2026). *MLPerf Inference: Datacenter benchmark*. https://mlcommons.org/benchmarks/inference-datacenter/. Consultado el 10 de junio de 2026. MLPerf publica resultados, divisiones, categorías de disponibilidad y metadatos de sistemas completos. ↩

9. Vert, A. (2026, 13 de mayo). *AI 101: Your Ultimate Guide to Attention: Mechanism, QKV, and KV Cache*. Turing Post. https://www.turingpost.com/p/your-ultimate-guide-to-attention-mechanism-qkv-and-kv-cache. Consultado el 10 de junio de 2026. ↩

10. Williams, S., Waterman, A. y Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM, 52*(4), 65-76. https://doi.org/10.1145/1498765.1498785. El modelo relaciona rendimiento con intensidad aritmética y distingue cuándo limita el cómputo y cuándo limita el ancho de banda de memoria. ↩