Facsímil 04

La caja de herramientas

Capítulo 03

Tokens, coste, contexto y caché

686f6c61 · 2026

Capítulo 03: Tokens, coste, contexto y caché

El presupuesto invisible de cada pregunta

Cuando una persona escribe “resúmeme este PDF” parece que está enviando una frase. En realidad puede estar enviando miles de tokens: instrucciones, historial, documento, herramientas disponibles, schema de salida y la propia pregunta. La factura no mira si la frase sonaba sencilla. Mira lo que entró, lo que salió, lo que se pudo cachear, el modelo elegido y cómo se sirvió la petición.

Este capítulo continúa el capítulo 02. Allí aprendimos a construir una llamada de API completa. Aquí hacemos la cuenta que decide si esa llamada cabe, cuánto cuesta, cuánto tarda y qué podemos reutilizar.

La idea central es esta: **un sistema con IA no solo se diseña con prompts; se diseña con presupuestos de tokens**.

Estado del arte con fecha de corte

Fecha de corte: 25 de mayo de 2026.
Fuentes consultadas ese día: documentación oficial de OpenAI sobre conteo de tokens, prompt caching, coste, Batch API, latencia y precios; documentación de Anthropic sobre token counting, prompt caching y ventanas de contexto; documentación de Google sobre token counting, long context, context caching y precios de Gemini API; el repositorio oficial de `tiktoken`; y artículos primarios sobre MoE, Switch Transformer, GShard y Mixtral.

Lo estable es el mecanismo: los modelos procesan tokens, la ventana de contexto es finita, la entrada y la salida se cobran de forma distinta, el streaming mejora la espera percibida, el batch puede abaratar trabajos diferidos y la caché solo ayuda cuando repites prefijos de forma reconocible.

Lo cambiante son los precios, modelos, nombres de campos, ventanas máximas, umbrales de cache, descuentos y límites. Para que las notas no aparezcan como una ristra de números pegados, las dejamos ordenadas en dos grupos: primero la documentación oficial de cada proveedor, después los artículos primarios que sustentan la sección de MoE.

PROVEEDOR	NOTA QUE CONVIENE REVISAR	POR QUÉ IMPORTA
OpenAI	Conteo de tokens.[1]	Estimar entrada, salida y compatibilidad de tokenizador.
OpenAI	Prompt caching.[2]	Diseñar prefijos repetibles y medir cache hit.
OpenAI	Optimización de coste.[3]	Separar coste de entrada, salida, cache y modelo.
OpenAI	Batch API.[4]	Pasar trabajos no interactivos a procesamiento diferido.
OpenAI	Optimización de latencia.[5]	Distinguir prefill, decode, streaming y tiempo percibido.
Anthropic	Token counting.[6]	Comparar conteos antes de migrar prompts.
Anthropic	Prompt caching.[7]	Pensar qué prefijos se reutilizan y durante cuánto tiempo.
Anthropic	Context windows.[8]	Saber qué entra, qué sale y qué se queda fuera.
Google	Conteo de tokens.[9]	Medir prompts, archivos y respuestas antes de desplegar.
Google	Long context.[10]	Evaluar cuándo una ventana larga ayuda y cuándo añade ruido.
Google	Context caching.[11]	Revisar TTL, prefijos y reutilización de contexto.
Google	Pricing de Gemini API.[12]	Presupuestar con tarifas vigentes.

Y los artículos primarios sobre Mixture-of-Experts, que sustentan la sección de MoE de este capítulo:

ARTÍCULO	POR QUÉ IMPORTA
Capa sparsely-gated MoE.[13]	Entender expertos, router y cómputo condicional.
Switch Transformer.[14]	Ver por qué top-1 simplifica routing, comunicación y entrenamiento.
GShard.[15]	Conectar MoE con sharding y entrenamiento distribuido.
Mixtral of Experts.[16]	Ver un LLM sparse MoE moderno con routing top-2 por token.

Qué no es un token

Un token no es una palabra. “Universidad” puede ocupar un token o varios, según el tokenizer. Un emoji puede ocupar más de uno. Un espacio, una coma o una tilde pueden cambiar la cuenta. Por eso contar palabras en un documento no sirve para estimar coste con precisión.

Tampoco es una unidad humana de significado. Para el modelo, un token es una pieza de codificación aprendida para representar texto de forma eficiente. A veces coincide con algo que reconocemos; a veces es una sílaba, una terminación, un símbolo o un fragmento raro.

Y no es igual en todos los modelos. Cada familia puede usar tokenizer, reglas multimodales y contabilidad distinta. OpenAI mantiene `tiktoken` como tokenizador rápido para sus modelos, pero eso no implica que el mismo conteo valga para Claude, Gemini o un modelo local.[17]

Cómo se construye un tokenizador

Un tokenizador, o constructor de tokens, no es una lista escrita a mano con todas las palabras posibles. Es una pieza entrenada sobre un corpus: mira mucho texto, aprende piezas frecuentes y produce una tabla estable de `texto -> id`. El modelo se entrena después con esos ids. Por eso no puedes cambiar el tokenizador de un modelo ya entrenado como quien cambia una fuente tipográfica: cambiarías el idioma interno con el que ese modelo aprendió.

La familia más intuitiva para empezar es BPE, Byte Pair Encoding. En NLP moderno se popularizó para manejar palabras raras y vocabularios abiertos en traducción neuronal.[18] La idea es sencilla: empezar con unidades pequeñas y fusionar pares frecuentes hasta construir piezas útiles.

El paso a paso mental es este:

PASO	QUÉ HACES	DECISIÓN DE INGENIERÍA
1	Reúnes un corpus representativo.	Si entrenas con textos legales, contarán mucho las piezas legales; si entrenas con código, contarán símbolos y nombres técnicos.
2	Normalizas lo mínimo necesario.	Minúsculas, Unicode, espacios y acentos cambian el vocabulario. No lo improvises.
3	Partes el texto en unidades pequeñas.	Caracteres, bytes o piezas iniciales. Los tokenizadores modernos suelen preferir variantes robustas a bytes.
4	Cuentas pares vecinos.	`("c", "a")`, `("a", "s")`, `("s", "a")`, etc.
5	Fusionas el par más frecuente.	Ese par pasa a ser una pieza nueva del vocabulario.
6	Repites hasta llegar al tamaño de vocabulario.	8 000, 32 000, 100 000 o lo que pida el modelo y el dominio.
7	Guardas vocabulario y reglas de merge.	Es un artefacto versionado, igual que pesos, configuración y model card.
8	Codificas texto nuevo aplicando esas reglas.	El resultado son ids numéricos que entran al modelo.

Este es el algoritmo Byte Pair Encoding (BPE), adaptado del campo de la compresión a la tokenización de modelos de lenguaje.[19] Su regla de fusión, en cada paso, elige el par de símbolos más frecuente:

$$(u^{\backslash *}, v^{\backslash *}) = \arg\max_{(u,v)} \text{freq}(u, v)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
u, v	Dos piezas vecinas candidatas.	`c` y `a`.
$\text{freq}(u, v)$	Frecuencia del par en el corpus.	`ca` aparece 28 000 veces.
$(u^{\backslash *}, v^{\backslash *})$	Par ganador que se fusiona.	`c` + `a` pasa a ser `ca`.
$\arg\max$	Elige el candidato con mayor frecuencia.	No devuelve la frecuencia, devuelve el par.

SentencePiece añade una idea muy útil para ingeniería multilingüe: puede entrenarse directamente sobre texto crudo y tratar los espacios como parte de la segmentación, en lugar de depender de un preprocesado específico de cada idioma.[20] Esto importa porque “separar

por espacios” funciona regular en inglés y español, pero se vuelve frágil con japonés, chino, emojis, código, nombres propios y formatos raros.

Detalles que un ingeniero debe tener muy presentes:

DECISIÓN	QUÉ ROMPE SI SE DECIDE MAL
Normalización Unicode	Dos textos visualmente iguales pueden tokenizar distinto.
Tamaño de vocabulario	Vocabulario pequeño alarga secuencias; vocabulario enorme aumenta tabla y puede memorizar rarezas inútiles.
Tratamiento de espacios	Cambia la cuenta de tokens y la reversibilidad del decode.
Tokens especiales	`system`, `tool`, separadores, imágenes o fin de texto deben tener ids reservados y documentados.
Dominio del corpus	Un tokenizador entrenado en conversación puede ser torpe con código o biomedicina.
Versionado	Modelo, tokenizer y plantilla de mensajes forman un paquete; si uno cambia, se revalida todo.

Qué sí es: la unidad que paga, cabe y tarda

Un token es la unidad que conecta tres preguntas de ingeniería:

PREGUNTA	QUÉ MIDE	POR QUÉ IMPORTA
¿Cabe?	Tokens de entrada más salida esperada frente a ventana de contexto.	Si no cabe, tienes que resumir, trocear, recuperar o rechazar.
¿Cuesta?	Tokens de entrada, salida, cache, batch y modelo.	Dos prompts parecidos pueden tener facturas muy distintas.
¿Tarda?	Tokens procesados en prefill y generados en decode.	Una entrada enorme tarda antes de empezar; una salida larga tarda mientras se genera.

La llamada que hicimos en el capítulo anterior no estaba completa hasta mirar tokens. `input`, `tools`, `schema`, documentos e imágenes ocupan presupuesto. La respuesta también. Si pides “razona mucho y dame una respuesta larga”, no solo estás pidiendo calidad: estás comprando tokens de salida y tiempo de generación.

La cuenta mínima: entrada, salida y ventana

La primera fórmula es sencilla:

$$T_{\text{total}} = T_{\text{entrada}} + T_{\text{salida}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{entrada}	Tokens enviados al modelo.	Instrucciones, historial, PDF, schema y pregunta suman 18 000 tokens.
T_{salida}	Tokens generados por el modelo.	El resumen y el JSON final ocupan 900 tokens.
T_{total}	Tokens de la llamada completa.	$18\,000 + 900 = 18\,900$.

Pero que el total exista no significa que quepa. Cada modelo tiene una ventana de contexto:

$$T_{\text{entrada}} + T_{\text{salida\_max}} \leq W$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$T_{\text{salida\_max}}$	Máximo de salida reservado.	Reservas 1 500 tokens para contestar.
W	Ventana de contexto del modelo elegido.	Un modelo con 32 000 tokens de ventana.
$\leq$	Restricción de cabida.	$18\,000 + 1\,500 \leq 32\,000$.

El detalle importante: reservar salida también consume ventana. Si llenas toda la ventana con documentos, quizá el modelo no tiene espacio para responder. Por eso el presupuesto de contexto debe decidir cuánto se queda cada pieza.

PIEZA	QUÉ SUELE OCUPAR	DECISIÓN PRÁCTICA
Instrucciones	Poco, pero se repite siempre.	Mantenerlas cortas, estables y cacheables.
Historial	Crece sin pedir permiso.	Resumir, compactar o guardar solo turnos relevantes.
Documentos	Puede dominar toda la llamada.	Usar RAG, citas, páginas o troceo.
Tools y schemas	No parecen contenido, pero cuentan.	Versionar y no inflar campos innecesarios.
Salida	Se paga y tarda.	Limitar `max_output_tokens` con criterio.

Coste: la factura no mira la dificultad, mira el uso

Los proveedores suelen separar precio de entrada y precio de salida. Algunos añaden categorías para cache writes, cache reads, batch, prioridad, procesamiento flexible, audio, imagen o razonamiento. Por eso la fórmula útil no es “precio por pregunta”, sino “precio por componentes”.

Este no es un truco didáctico: es el modelo de facturación real que publican OpenAI, Anthropic y Google en sus páginas de precios, donde cada categoría de token (entrada, salida, lectura de caché, escritura de caché) lleva su propia tarifa por millón.[21] El coste de una llamada es la suma ponderada del recuento de tokens de cada categoría por su precio unitario:

$$C = \frac{T_i P_i + T_o P_o + T_{cr} P_{cr} + T_{cw} P_{cw}}{1\,000\,000}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
C	Coste estimado de la llamada.	0,0068 euros o dólares, según tarifa.
T_i	Tokens de entrada frescos.	8 000 tokens no cacheados.
P_i	Precio por millón de tokens de entrada.	2,00 por millón (precio ilustrativo).
T_o	Tokens de salida.	700 tokens generados.
P_o	Precio por millón de tokens de salida.	8,00 por millón (precio ilustrativo).
T_{cr}	Tokens leídos desde caché.	12 000 tokens reutilizados.
P_{cr}	Precio por millón de tokens cacheados leídos.	Menor que P_i si el proveedor descuenta cache read.
T_{cw}	Tokens escritos en caché.	12 000 tokens de prefijo guardado.
P_{cw}	Precio por millón de cache write.	Puede ser igual, mayor o no aplicar según proveedor.

No uses los números del ejemplo para presupuestar un producto real. Usa la fórmula y consulta la página oficial de precios el día que diseñes el sistema.[22] Lo profesional no es saberse una tarifa de memoria; es guardar en configuración qué modelo, proveedor, fecha y precio estás usando para cada estimación.

La caché no es gratis el primer día: amortización

La caché de prompt cambia la aritmética anterior, pero no de forma mágica. Escribir un prefijo en caché suele costar más que procesarlo una vez (los proveedores que separan esta categoría cobran la escritura por encima del precio de entrada), y leerlo después cuesta mucho menos. Por eso la caché es una inversión: paga de más una vez para pagar de menos muchas. Conviene cuando el prefijo se reutiliza, no cuando cambia en cada llamada.

Pongámoslo en números con tarifas ilustrativas de 2,00 (entrada), 8,00 (salida), 0,50 (lectura de caché) y 2,50 (escritura de caché) por millón. Una llamada tiene un prefijo estable de 12 000 tokens (una normativa), 8 000 tokens frescos de pregunta y genera 700 tokens. La primera llamada escribe el prefijo en caché; las siguientes lo leen:

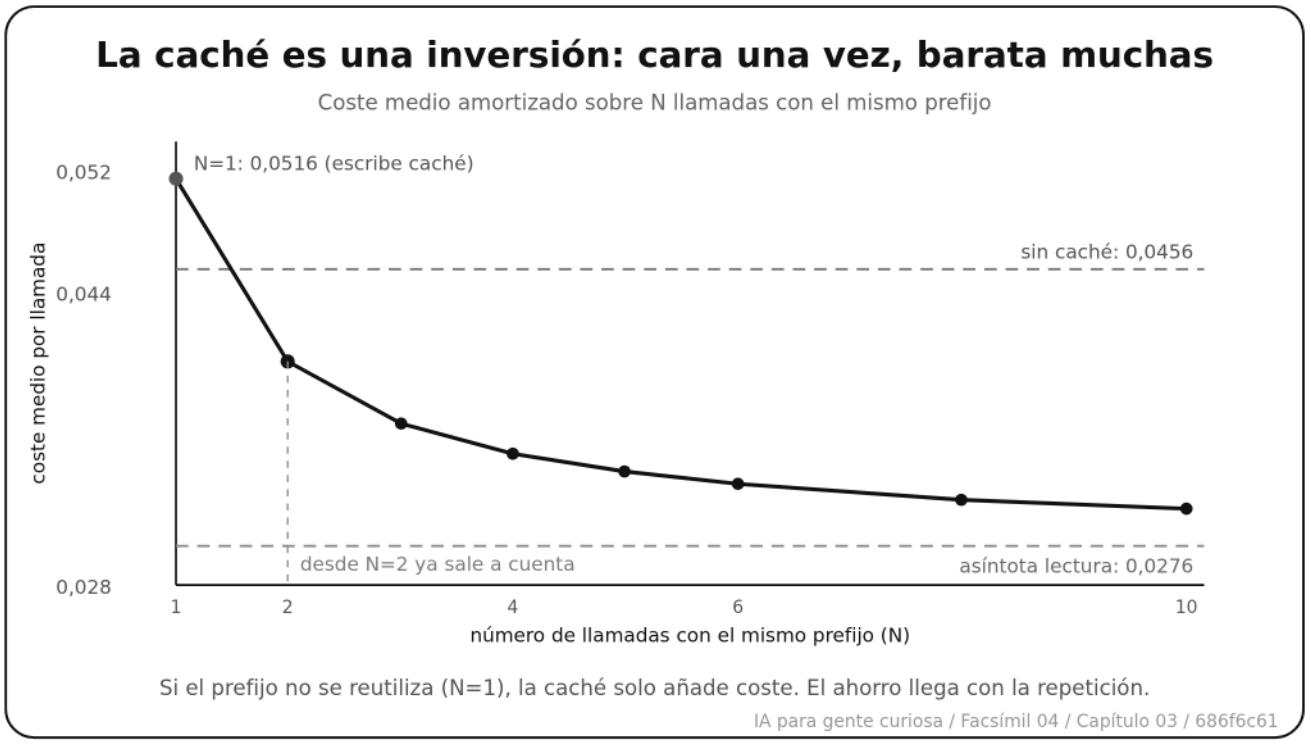
$$C_{\text{write}} = \frac{8000 \cdot 2{,}00 + 12000 \cdot 2{,}50 + 700 \cdot 8{,}00}{10^6} = 0{,}0516$$

$$C_{\text{read}} = \frac{8000 \cdot 2{,}00 + 12000 \cdot 0{,}50 + 700 \cdot 8{,}00}{10^6} = 0{,}0276$$

Sin caché, cada llamada paga el prefijo a precio de entrada completo: $C_{\text{full}} = 0{,}0456$. La primera llamada con caché sale más cara que sin ella (0,0516 frente a 0,0456), pero cada repetición ahorra. El coste medio amortizado sobre N llamadas con el mismo prefijo es:

$$\bar{C}(N) = \frac{C_{\text{write}} + (N-1)\,C_{\text{read}}}{N}$$

Con estos números, la caché empieza a salir a cuenta frente a no cachear a partir de la segunda llamada ($N \geq 2$), y cuando N crece, $\bar{C}(N)$ tiende a $C_{\text{read}} = 0{,}0276$: un 40 % menos que el coste sin caché. La regla de ingeniería: cachea prefijos que se reutilizan al menos un puñado de veces dentro del TTL; no cachees lo que cambia en cada petición.



Contexto: meter más no siempre ayuda

La ventana larga es una bendición cuando necesitas leer un expediente grande, comparar documentos o mantener una conversación compleja. Pero más contexto también compra coste, latencia y ruido. Un documento irrelevante dentro del prompt no es gratis: ocupa presupuesto y puede distraer.

La ventana de contexto W es una capacidad fija: un número máximo de tokens que el modelo puede procesar en una llamada. Diseñar el contexto es, en términos formales, un problema de asignación de un recurso escaso. Todo lo que metas debe caber, y la entrada y la salida comparten esa misma cuenta:

$$B_{\text{instrucciones}} + B_{\text{historial}} + B_{\text{documentos}} + B_{\text{tools}} + B_{\text{salida}} \;\leq\; W$$

lo que reservas

SÍMBOLO	SIGNIFICADO	EJEMPLO
W	Capacidad total de la ventana del modelo.	128 000 tokens en un modelo concreto.
$B_{\text{instrucciones}}$	Presupuesto reservado a reglas estables.	600 tokens.
$B_{\text{historial}}$	Presupuesto de conversación previa.	2 000 tokens.
$B_{\text{documentos}}$	Presupuesto para evidencia externa.	20 000 tokens.
B_{tools}	Presupuesto para definiciones de herramientas y schemas.	1 400 tokens.
B_{salida}	Presupuesto reservado para responder.	1 500 tokens.

La desigualdad obliga a un reparto explícito: si reservas 1 500 tokens para la salida, esos no están disponibles para documentos. Tratar la asignación como una restricción dura, y no como un deseo, es lo que distingue un sistema que falla con un error claro («el contexto no cabe, reduce documentos») de uno que trunca en silencio y degrada la respuesta. El error típico es decidir el contexto al final. En realidad conviene decidirlo antes de llamar a la API: “para esta tarea, el modelo puede ver tres fragmentos, no veinte; debe citar página; y si no cabe, se resume o se pregunta de nuevo”.

Contexto, memoria y KV cache no son lo mismo

En producto solemos llamar “contexto” a todo lo que acompaña a la pregunta: instrucciones del sistema, mensajes anteriores, documentos recuperados, resultados de herramientas, imágenes, tablas, preferencias del usuario y formato esperado. Para el modelo, eso no llega como recuerdos. Llega como una secuencia de ids de token en una llamada concreta.

El recorrido real es más técnico:

ETAPA	QUÉ OCURRE	QUÉ QUEDA GUARDADO
Texto a tokens	El tokenizador convierte texto y partes estructuradas en ids.	La lista de ids de entrada.
Tokens a vectores	Cada id se convierte en embedding y se combina con información de posición.	Tensores de entrada para el transformer.
Atención	Cada capa calcula consultas, claves y valores: Q, K, V.	Activaciones temporales de la llamada.
Prefill	El servidor procesa todo el prefijo de entrada.	Claves y valores listos para generar.
Decode	El modelo genera un token, lo añade a la secuencia y repite.	La KV cache crece token a token.
Fin de llamada	Se devuelve texto, JSON o eventos de streaming.	Nada queda en los pesos del modelo por defecto.

La atención original del transformer compara cada consulta con claves y usa valores para mezclar información relevante.[23] En inferencia autoregresiva, no queremos recalcular las claves y valores de todos los tokens anteriores cada vez que generamos uno nuevo. Por eso los servidores guardan una KV cache temporal.

La fórmula conceptual de atención es:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{d_k}\right)V$$

SÍMBOLO	QUÉ SIGNIFICA	INTUICIÓN
Q	Queries o consultas.	Qué está buscando el token actual.
K	Keys o claves.	Qué ofrece cada token anterior para ser encontrado.
V	Values o valores.	Información que se mezcla si la atención la considera útil.
d_k	Dimensión de las claves.	Factor de escala para estabilizar el producto.

La KV cache guarda K y V ya calculados. No es memoria del usuario. Es memoria de cálculo durante la inferencia. Si una conversación tiene 20 000 tokens de entrada y genera 800 tokens de salida, el servidor va manteniendo claves y valores para evitar rehacer todo el prefijo en cada paso.

La memoria que ocupa esa KV cache se calcula con la fórmula estándar de serving, la misma que usan los sistemas de inferencia para planificar batch y memoria de GPU:

$$M_{\text{KV}} = 2 \cdot L \cdot B \cdot S \cdot H_{\text{kv}} \cdot d_{\text{head}} \cdot \text{bytes}$$

SÍMBOLO	SIGNIFICADO	QUÉ MUEVE EN LA PRÁCTICA
2	Guardamos claves y valores.	K y V, no solo una matriz.
L	Número de capas.	Modelos más profundos consumen más cache.
B	Batch o secuencias simultáneas.	Más usuarios concurrentes, más memoria.
S	Longitud de secuencia.	Contexto largo y salida larga hacen crecer la cache.
H_{kv}	Cabezas KV.	MQA/GQA reducen cabezas KV frente a atención multi-head clásica.[24]
d_{head}	Dimensión por cabeza.	Depende de arquitectura.
bytes	Precisión usada.	FP16, BF16, INT8 o variantes cuantizadas cambian memoria.

Esta fórmula explica por qué el contexto largo no es solo “más texto”. Es memoria de GPU, planificación de batch, colas y throughput. Sistemas como vLLM investigan precisamente cómo gestionar esa memoria de KV cache con menos desperdicio mediante PagedAttention.[25]

Ahora sí podemos separar conceptos que a menudo se mezclan:

CONCEPTO	VIVE DÓNDE	DURA CUÁNTO	LO CONTROLA QUIÉN	PARA QUÉ SIRVE
Contexto de llamada	En el payload enviado al modelo.	Una petición.	Tu aplicación.	Dar instrucciones, evidencia y formato de salida.
Historial	En tu base de datos o en el cliente.	Hasta que lo borres o resumas.	Tu producto.	Reconstruir conversación útil.
Memoria de producto	En una base de datos, perfil, resumen o vector store.	Persistente o con política de expiración.	Tu producto y sus permisos.	Recordar preferencias o hechos útiles entre sesiones.
KV cache	En memoria del servidor de inferencia.	Durante una secuencia o sesión gestionada por el runtime.	El proveedor o tu servidor local.	Acelerar decode y evitar recomputar prefijos.
Prompt cache	En la infraestructura del proveedor o runtime.	Según reglas, TTL y coincidencia de prefijo.	Proveedor más diseño de prompt.	Reutilizar trabajo de prefijos repetidos entre llamadas.

La frase correcta sería: “nuestro producto recuerda algo porque lo guardamos y lo volvemos a meter en el contexto”. El modelo base no actualiza sus pesos por leer un mensaje de un usuario. Para que “recuerde” en otra llamada, alguien tiene que almacenar, recuperar, filtrar y reinyectar esa información.

Un MoE capa a capa dentro de este proceso

Ahora metamos una arquitectura MoE en la misma película. No cambia la idea básica de este capítulo: entran tokens, se procesan dentro de una ventana, se genera salida y se mide coste/latencia. Lo que cambia está dentro de algunas capas del Transformer: en lugar de que todos los tokens pasen por el mismo MLP denso, un router aprende a mandar cada token a uno o varios expertos.

En un bloque Transformer denso, cada capa aplica atención y una red feed-forward, ambas con conexión residual y normalización, tal como define la arquitectura original:[26]

$$u_t^{(\ell)} = h_t^{(\ell)} + \mathrm{Attention}^{(\ell)}(\mathrm{LN}(h_t^{(\ell)}))$$

$$h_t^{(\ell+1)} = u_t^{(\ell)} + \mathrm{MLP}^{(\ell)}(\mathrm{LN}(u_t^{(\ell)}))$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$h_t^{(\ell)}$	Estado del token t al entrar en la capa ℓ.	Vector del token “cache” en la capa 12.
$\mathrm{Attention}$	Parte que mezcla información del contexto mediante Q, K y V.	Lee tokens anteriores y usa la KV cache.
MLP	Red feed-forward densa del bloque.	La misma red para todos los tokens de esa capa.
LN	LayerNorm.	Normaliza antes de atención o MLP.
$u_t^{(\ell)}$	Estado tras atención y residual.	El token ya trae información del contexto.

En un MoE, la parte que se suele sustituir es el MLP por un conjunto de expertos con un router que activa solo unos pocos por token, el mecanismo de capa sparsely-gated y su simplificación top-1.[27] La atención sigue mezclando contexto; después entra el router:

$$s_t^{(\ell)} = W_r^{(\ell)}\,\mathrm{LN}(u_t^{(\ell)})$$

$$p_t^{(\ell)} = \mathrm{softmax}(s_t^{(\ell)})$$

$$C_t^{(\ell)} = \mathrm{TopK}(p_t^{(\ell)}, k)$$

$$\mathrm{MoE}^{(\ell)}(u_t) = \sum_{i \in C_t^{(\ell)}} \alpha_i E_i^{(\ell)}(\mathrm{LN}(u_t^{(\ell)}))$$

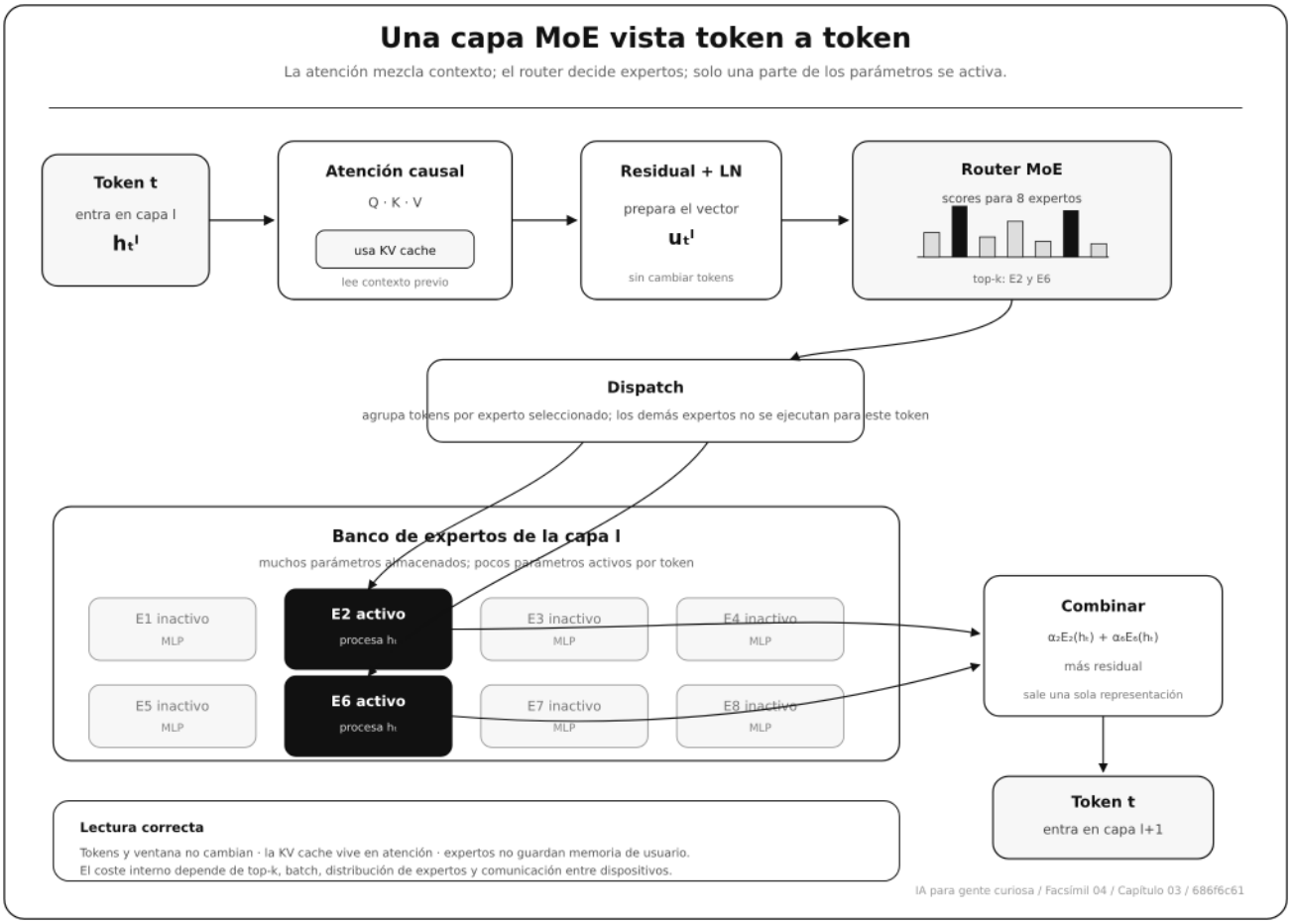
$$h_t^{(\ell+1)} = u_t^{(\ell)} + \mathrm{MoE}^{(\ell)}(u_t)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$W_r^{(\ell)}$	Matriz aprendida del router en la capa ℓ.	Produce una puntuación por experto.
$s_t^{(\ell)}$	Logits del router para el token t.	Ocho puntuaciones si hay ocho expertos.
$p_t^{(\ell)}$	Probabilidades de routing tras softmax.	`[0.02, 0.61, 0.04, 0.21, ...]`.
k	Número de expertos activados por token.	Switch usa top-1; Mixtral usa top-2.
$C_t^{(\ell)}$	Conjunto de expertos elegidos para ese token y esa capa.	Expertos 2 y 4 en la capa 18.
$E_i^{(\ell)}$	Experto i, normalmente una red feed-forward.	Un MLP independiente dentro de la capa.
α_i	Peso normalizado con el que se combina cada experto seleccionado.	Si top-2, cada experto aporta una parte.

La película layer a layer queda así:

PASO	QUÉ LE PASA AL TOKEN	QUÉ IMPORTA PARA COSTE, CONTEXTO Y CACHÉ
1	El texto ya fue tokenizado y convertido en embeddings.	El MoE no cambia el conteo de tokens de entrada.
2	El token entra en la capa ℓ con un vector $h_t^{(\ell)}$.	La secuencia sigue teniendo longitud S.
3	La atención calcula Q, K, V y mezcla contexto.	La KV cache sigue siendo de atención, no de “memoria del experto”.
4	El resultado pasa por residual y normalización.	El token ya trae información de tokens anteriores.
5	El router calcula puntuaciones para expertos.	Aparece cómputo extra pequeño: routing.
6	Se eligen k expertos.	Cambian los parámetros activos por token.
7	El runtime agrupa tokens por experto.	En GPU distribuida puede haber comunicación entre dispositivos.
8	Cada experto procesa los tokens que le tocaron.	No se ejecutan todos los expertos para cada token.
9	Se combinan las salidas de los expertos elegidos.	En top-2 hay mezcla; en top-1 se simplifica.
10	Se suma residual y el token pasa a la siguiente capa.	En la capa siguiente puede elegir expertos distintos.

Visualmente, una capa MoE se entiende mejor si la dibujamos como una cinta de procesamiento: el token pasa por atención, el router decide, solo algunos expertos trabajan y sus salidas se recombinan.



Esto explica una frase que en fichas técnicas suele confundirse: parámetros totales no son parámetros activos. Un MoE puede tener muchos parámetros almacenados porque tiene muchos expertos, pero cada token usa solo una parte. Mixtral, citado en el bloque de estado del arte, describe capas con ocho bloques feed-forward y selección de dos expertos por token y por capa; el propio artículo distingue entre parámetros accesibles y parámetros activos durante inferencia.

En serving, la parte delicada no es solo matemática. Es logística:

PROBLEMA OPERATIVO	QUÉ OCURRE
Balanceo de carga	Si demasiados tokens van al mismo experto, ese experto se convierte en cuello de botella.
Capacidad por experto	Los runtimes suelen limitar cuántos tokens procesa cada experto por lote.
Comunicación	Si los expertos viven en dispositivos distintos, los tokens o activaciones tienen que moverse.
Batch irregular	Dos peticiones con los mismos tokens de entrada no tienen por qué activar exactamente la misma distribución de expertos.
Métrica de coste	El usuario paga tokens, pero el proveedor opera con parámetros activos, routing, memoria y comunicación.

Por eso MoE es muy relevante para este capítulo: te obliga a leer “coste por token” con más finura. Desde fuera sigues viendo tokens de entrada, tokens de salida, ventana y precio. Por dentro, cada token atraviesa todas las capas, pero en las capas MoE solo activa una ruta dispersa. El contexto no se “guarda en los expertos”; la memoria temporal de contexto sigue estando en la atención y su KV cache. Los expertos transforman representaciones, no almacenan recuerdos de usuario.

Caché: repetir bien para pagar y esperar menos

Prompt caching aprovecha un hecho simple: muchas llamadas repiten prefijos. Las instrucciones del sistema, las herramientas, el schema, una normativa larga o un conjunto de ejemplos pueden ser iguales durante muchas peticiones. Si el proveedor reconoce ese prefijo, puede reutilizar trabajo ya hecho.

En las guías citadas al inicio, OpenAI describe caché para mensajes, imágenes, tool use y structured outputs, y recomienda colocar el contenido estático o repetido al principio y lo dinámico al final. Anthropic explica breakpoints explícitos y automáticos, con especial atención al orden `tools`, `system` y `messages` en el prefijo cacheable. Gemini distingue caché implícita y caché explícita con TTL configurable.

Hay dos cachés que se confunden mucho:

CACHÉ	QUÉ REUTILIZA	CUÁNDO LA NOTAS	QUÉ MIRAR
KV cache	K y V ya calculados dentro de una secuencia.	En decode, porque cada token nuevo no recalcula todo el prefijo.	Memoria, longitud de secuencia, batch, throughput.
Prompt cache o context cache	Un prefijo repetido entre llamadas.	En coste, latencia o campos de usage del proveedor.	Orden estable del prompt, TTL, cache hit y contenido dinámico al final.

El diseño de prompt cache es casi diseño de APIs: si serializas un JSON con claves en orden aleatorio, metes timestamps arriba o cambias ejemplos sin necesidad, rompes coincidencias. Si colocas primero instrucciones, tools, schema y documentos estables, y dejas al final la pregunta concreta del usuario, aumentas la probabilidad de reutilización.

La métrica que queremos mirar es:

$$H = \frac{T_{\text{cache\_hit}}}{T_{\text{entrada}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
H	Proporción de entrada servida desde caché.	$0{,}72$, es decir, 72 %.
$T_{\text{cache\_hit}}$	Tokens de entrada que fueron cache hit.	14 400 tokens.
T_{entrada}	Tokens de entrada totales.	20 000 tokens.

Para entenderlo: si cada petición incluye una normativa de 15 000 tokens y solo cambia la pregunta final, la caché puede tener sentido. Si cada petición mete documentos distintos, timestamps dentro del prefijo y orden aleatorio de fragmentos, la caché probablemente no ayudará.

Latencia: prefill, decode y espera percibida

La latencia no es una sola cosa. Hay tiempo de red, cola, prefill, generación y renderizado. Un modelo afín de primer orden separa entrada y salida, y se apoya en las dos fases reales de la inferencia autoregresiva: el coeficiente α aproxima el coste de *prefill* por token de entrada y β el de *decode* por token de salida (el inverso del ritmo de generación).[28]

$$L \approx L_0 + \alpha T_{\text{entrada\_fresca}} + \beta T_{\text{salida}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Latencia total aproximada.	4,8 segundos.
L_0	Coste fijo de red, cola y preparación.	350 ms.
α	Coste medio por token de entrada fresca.	Depende de modelo y hardware.
$T_{\text{entrada\_fresca}}$	Entrada no cubierta por caché.	3 000 tokens.
β	Coste medio por token generado.	Depende de decode.
T_{salida}	Tokens generados.	900 tokens.

El streaming no reduce necesariamente el tiempo total, pero reduce el tiempo hasta ver el primer fragmento. Batch puede reducir coste o mejorar operación cuando no necesitas respuesta inmediata. La caché puede reducir prefill si el prefijo se reutiliza. Elegir un modelo menor puede reducir coste y latencia, pero quizá empeora calidad. Todo vuelve al triángulo: calidad, coste y tiempo.

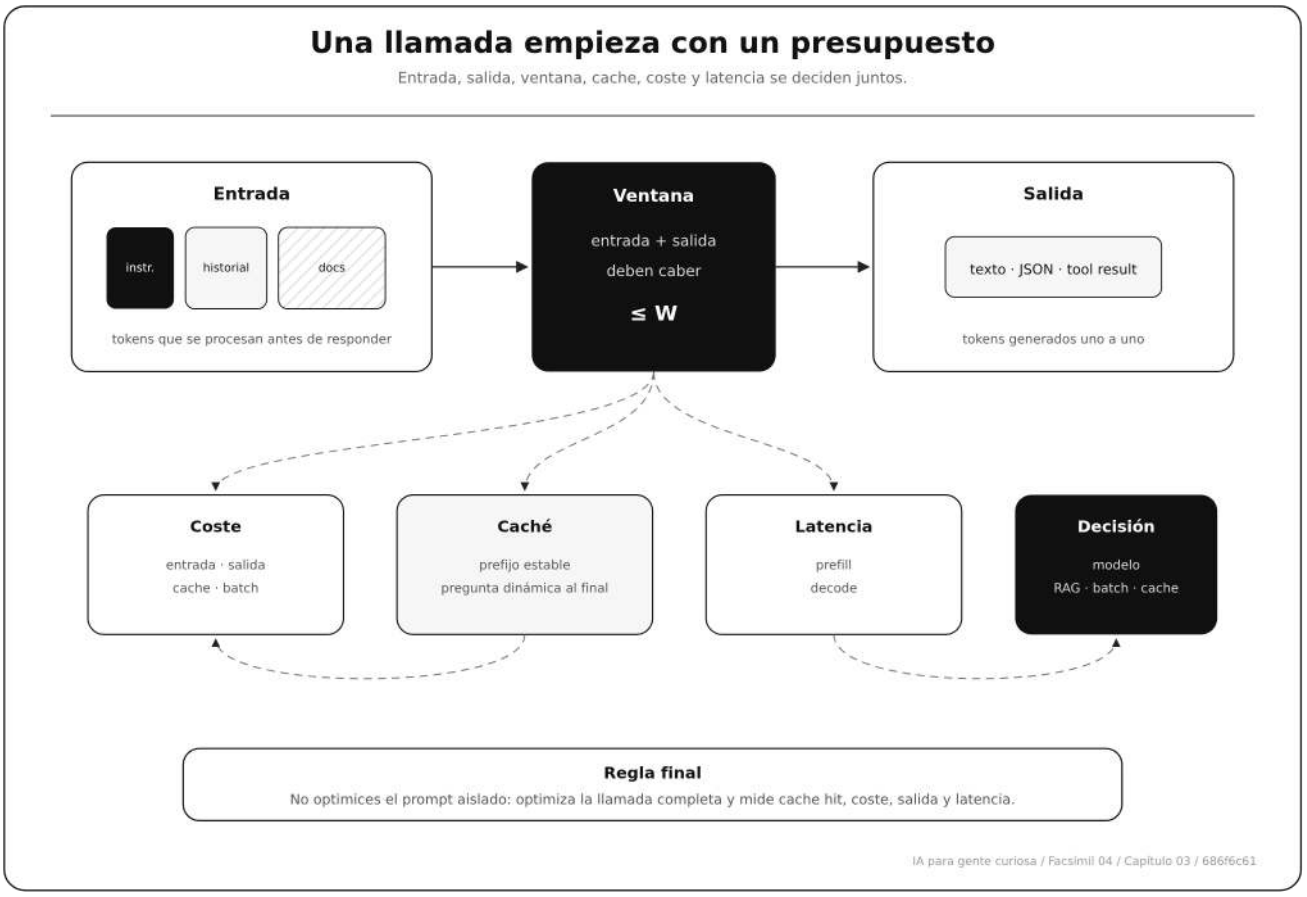
Para entenderlo antes de tocar código

Pensemos en cuatro casos cercanos:

CASO	QUÉ PESA	QUÉ HARÍA
Tutor que corrige respuestas cortas	Salida estructurada y muchas llamadas.	Modelo menor, schema corto, batch si no es interactivo.
Asistente de normativa universitaria	Documento largo repetido.	Prefijo estable, cache, RAG si la normativa es grande.
Chat de soporte con historial largo	Historial que crece cada turno.	Compactar, resumir y guardar solo lo útil.
Analizador de facturas con imágenes	Imagen, OCR, schema y salida JSON.	Medir tokens/latencia multimodal y limitar campos.
Copiloto interno con preferencias	Memoria de producto y contexto recuperado.	Guardar preferencias fuera del modelo y reinyectar solo las relevantes.
Servicio con muchas sesiones simultáneas	KV cache, batch y cola.	Vigilar memoria de inferencia, longitud media y tokens por segundo.
Modelo MoE en producción	Routing, expertos y parámetros activos.	Medir latencia real; no comparar solo parámetros totales.

La pregunta útil no es “¿cuántos tokens acepta el modelo más grande?”. La pregunta útil es “¿cuántos tokens necesita esta tarea para dar una respuesta fiable sin pagar ruido?”.

Mapa visual de presupuesto de tokens



En el día a día

En un proyecto real, este capítulo aparece cuando alguien pregunta: “¿por qué esto cuesta tanto?” o “¿por qué tarda tanto?”. Muchas veces la respuesta no está en cambiar de modelo, sino en mirar el payload completo.

Si cada llamada manda el mismo documento largo, quizá toca cache. Si cada llamada manda veinte fragmentos RAG y solo dos eran relevantes, toca mejorar retrieval. Si el usuario necesita respuesta inmediata, quizá no puedes batchar. Si el proceso es nocturno, batch puede ser perfecto. Si la salida siempre se alarga, limita tokens de salida y cambia el contrato.

Si eliges un modelo MoE, añade una pregunta más: ¿cuántos parámetros son totales y cuántos activos por token? No necesitas ver el router interno para operar una API comercial, pero sí necesitas entender que “47B parámetros” y “13B activos” no significan lo mismo, y que la latencia real dependerá también de routing, expertos, batch y hardware.

Una integración madura registra al menos: tokens de entrada, tokens de salida, tokens cacheados si el proveedor los devuelve, modelo, arquitectura si se conoce, latencia, coste estimado, schema y motivo de selección del modelo. Sin esos datos, optimizar es opinar con cara seria.

Por qué debería importarte

Porque los tokens convierten decisiones aparentemente literarias en decisiones de producto. “Añadamos más contexto” puede duplicar coste. “Respondamos con más detalle” puede multiplicar salida. “Metamos todos los documentos” puede romper ventana o empeorar la respuesta. “Usemos el modelo grande siempre” puede ser innecesario.

MoE añade otra trampa sana: un modelo puede ser enorme en parámetros totales y, aun así, activar solo una fracción por token. Eso puede mejorar capacidad sin multiplicar igual el cómputo, pero también complica serving, balanceo y lectura de fichas técnicas.

También importa para enseñar y aprender. Cuando entiendes tokens, dejas de pensar en la IA como una caja opaca y empiezas a verla como un sistema con restricciones medibles.

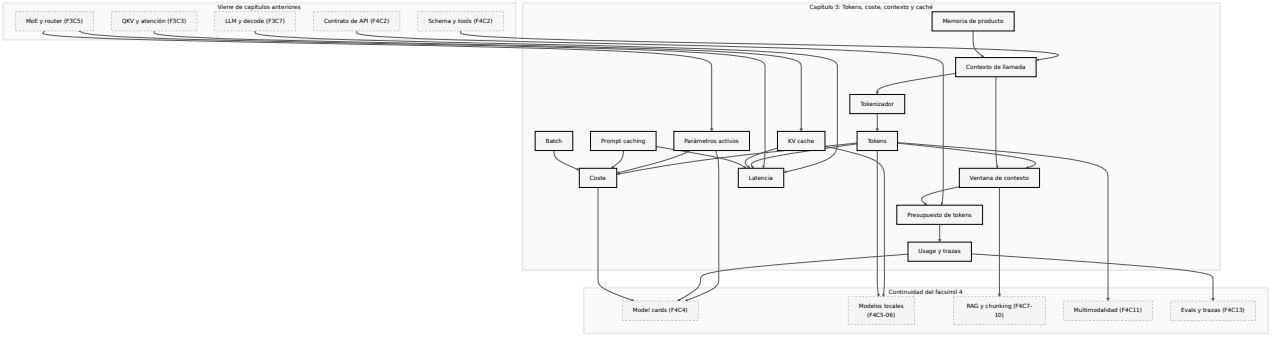
Dónde solía tropezar yo

Estos tropiezos son muy comunes cuando se pasa de una demo a una aplicación con usuarios.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Contar palabras y no tokens	La factura y la ventana no entienden palabras humanas.	Usar el contador del proveedor o tokenizer compatible.
Pensar que todos los tokenizadores son equivalentes	Un mismo texto puede producir ids y longitudes distintas según modelo.	Versionar tokenizer, plantilla de mensajes y modelo como un conjunto.
Llenar la ventana hasta el borde	Si no reservas salida, el modelo no tiene espacio para contestar.	Separar presupuesto de entrada y salida máxima.
Meter contexto por tranquilidad	El contexto irrelevante cuesta, tarda y puede confundir.	Recuperar menos, citar mejor y medir calidad.
Confundir contexto con memoria	El modelo no recuerda por defecto lo que no le vuelves a enviar.	Guardar memoria en producto y reinyectarla con permisos y criterio.
Leer parámetros totales como coste por token	En MoE, muchos parámetros existen, pero solo algunos expertos se activan por token.	Mirar parámetros activos, routing y latencia medida.
Pensar que el experto MoE guarda conocimiento humano etiquetado	Un experto es una subred aprendida, no “el experto de matemáticas” de forma garantizada.	Hablar de rutas internas y medir comportamiento, no inventar etiquetas humanas.
Esperar demasiado de la caché	La caché solo ayuda si repites prefijos estables.	Ordenar prompt: estable primero, dinámico al final.
Mezclar prompt cache y KV cache	Una ahorra trabajo entre llamadas; la otra acelera el decode dentro de una secuencia.	Medir ambas con métricas distintas.
No registrar usage	Sin tokens reales no puedes explicar coste ni latencia.	Guardar usage, cache hit, modelo y traza por llamada.

Cómo encaja todo

Este mapa conecta tokens con las decisiones que ya venimos construyendo: APIs, schemas, RAG, elección de modelo y operación.



Vocabulario aprendido

Estos términos nos permiten hablar de coste y contexto sin quedarnos en “el prompt es largo”.

TÉRMINO	DEFINICIÓN
Token	Unidad mínima que el modelo procesa; puede ser una palabra, fragmento, signo o espacio.
Tokenizador	Software que convierte texto en ids de tokens y reconstruye texto desde esos ids.
Vocabulario de tokens	Tabla versionada que asigna piezas a identificadores numéricos.
BPE	Técnica que aprende subpalabras fusionando pares frecuentes.
SentencePiece	Enfoque de tokenización de subpalabras que puede aprender desde texto crudo y tratar espacios explícitamente.
Ventana de contexto	Límite de tokens que pueden viajar juntos en una llamada.
Token de entrada	Token enviado al modelo antes de generar.
Token de salida	Token producido por el modelo durante la respuesta.
Prefill	Procesamiento inicial de la entrada.
Decode	Generación de salida token a token.
KV cache	Claves y valores de atención guardados temporalmente para acelerar inferencia.
Memoria de producto	Información persistida por la aplicación y reinyectada en contexto cuando toca.
MoE	Arquitectura con varios expertos donde cada token activa solo algunos en ciertas capas.
Router MoE	Módulo aprendido que puntúa expertos y elige top-k para cada token y capa.
Parámetros activos	Parámetros usados realmente por un token, distintos de todos los parámetros almacenados.
Prompt caching	Reutilización de un prefijo repetido cuando el proveedor lo soporta.
Cache hit	Tokens que coinciden con contenido cacheado.
Batch	Procesamiento diferido de muchas peticiones.
Presupuesto de tokens	Reparto planificado entre instrucciones, historial, documentos, tools y salida.

Antes de pasar página

- [ ] ¿Puedo explicar por qué un token no es una palabra? (Si no, vuelve a «Qué no es un token».)
- [ ] ¿Sé construir mentalmente un tokenizador BPE mínimo: corpus, pares, merges, vocabulario e ids? (Si no, vuelve a «Cómo se construye un tokenizador».)
- [ ] ¿Sé calcular $T_{\text{entrada}} + T_{\text{salida\_max}} \leq W$? (Si no, vuelve a «La cuenta mínima: entrada, salida y ventana».)
- [ ] ¿Entiendo por qué reservar salida también consume ventana? (Si no, vuelve a «Contexto: meter más no siempre ayuda».)
- [ ] ¿Puedo distinguir contexto, historial, memoria de producto, KV cache y prompt cache? (Si no, vuelve a «Contexto, memoria y KV cache no son lo mismo».)
- [ ] ¿Puedo explicar qué hace un MoE capa a capa: atención, router, top-k, expertos y combinación? (Si no, vuelve a «Un MoE capa a capa dentro de este proceso».)
- [ ] ¿Sé distinguir parámetros totales y parámetros activos en un modelo MoE? (Si no, vuelve a «Un MoE capa a capa dentro de este proceso».)
- [ ] ¿Puedo estimar coste separando entrada, salida, cache read y cache write? (Si no, vuelve a «Coste: la factura no mira la dificultad, mira el uso».)
- [ ] ¿Sé cuándo la caché puede ayudar y cuándo no? (Si no, vuelve a «Caché: repetir bien para pagar y esperar menos».)
- [ ] ¿Puedo distinguir latencia de prefill y latencia de decode? (Si no, vuelve a «Latencia: prefill, decode y espera percibida».)
- [ ] ¿Sé cuándo batch tiene sentido y cuándo rompería la experiencia? (Si no, vuelve a «Latencia: prefill, decode y espera percibida».)
- [ ] ¿Sé cómo cambian el coste y la latencia al variar el corpus, el `cache_hit` y los `documentos`? (Si no, repasa la sección correspondiente.)

En resumen

Los tokens son la contabilidad básica de una integración con modelos. No son solo un detalle técnico: determinan cabida, coste, latencia, diseño de contexto, selección de modelo y estrategia de operación.

IDEA FUERZA	DETALLE
El token es la unidad que paga, cabe y tarda.	Palabras, documentos y herramientas se traducen a tokens.
El tokenizador es parte del modelo.	Si cambias tokenizer, vocabulario o plantilla de mensajes, cambias la entrada real.
La ventana de contexto se reparte.	Instrucciones, historial, documentos, tools y salida compiten por el mismo espacio.
El modelo no recuerda por defecto.	La memoria útil vive en producto y se vuelve a meter en contexto.
La KV cache no es memoria de usuario.	Es memoria temporal de inferencia para no recalcular claves y valores.
MoE cambia el MLP, no la naturaleza de los tokens.	Cada token sigue atravesando capas, pero activa solo algunos expertos en las capas MoE.
Parámetros totales no son parámetros activos.	En un MoE hay que preguntar cuántos expertos se activan por token y cómo afecta a latencia.
El coste es por componentes.	Entrada, salida, cache y batch pueden tener precios distintos.
La caché necesita prefijos estables.	Lo repetido va al principio; lo dinámico al final.
Más contexto no siempre mejora.	Puede aumentar ruido, coste y latencia.
Sin usage no hay optimización seria.	Hay que registrar tokens, cache hit, modelo, coste y latencia.

Para saber más

Ainslie, J. y otros (2023). *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. https://arxiv.org/abs/2305.13245

Anthropic. (2026). *Context windows*. https://platform.claude.com/docs/en/build-with-claude/context-windows

Anthropic. (2026). *Prompt caching*. https://platform.claude.com/docs/en/build-with-claude/prompt-caching

Anthropic. (2026). *Token counting*. https://platform.claude.com/docs/en/build-with-claude/token-counting

Fedus, W., Zoph, B. y Shazeer, N. (2022). *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. https://jmlr.org/papers/v23/21-0998.html

Google. (2026). *Context caching*. https://ai.google.dev/gemini-api/docs/caching

Google. (2026). *Gemini Developer API pricing*. https://ai.google.dev/gemini-api/docs/pricing

Google. (2026). *Long context*. https://ai.google.dev/gemini-api/docs/long-context

Google. (2026). *Token counting*. https://ai.google.dev/gemini-api/docs/tokens

Jiang, A. Q. y otros (2024). *Mixtral of Experts*. https://arxiv.org/abs/2401.04088

Kudo, T. y Richardson, J. (2018). *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. https://aclanthology.org/D18-2012/

Kwon, W. y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. https://arxiv.org/abs/2309.06180

Lepikhin, D. y otros (2021). *GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding*. https://arxiv.org/abs/2006.16668

OpenAI. (2026). *Batch API*. https://developers.openai.com/api/docs/guides/batch

OpenAI. (2026). *Cost optimization*. https://developers.openai.com/api/docs/guides/cost-optimization

OpenAI. (2026). *Counting tokens*. https://developers.openai.com/api/docs/guides/token-counting

OpenAI. (2026). *Latency optimization*. https://developers.openai.com/api/docs/guides/latency-optimization

OpenAI. (2026). *Pricing*. https://developers.openai.com/api/docs/pricing

OpenAI. (2026). *Prompt caching*. https://developers.openai.com/api/docs/guides/prompt-caching

OpenAI. (2026). *tiktoken*. https://github.com/openai/tiktoken

Sennrich, R., Haddow, B. y Birch, A. (2016). *Neural Machine Translation of Rare Words with Subword Units*. https://aclanthology.org/P16-1162/

Shazeer, N. y otros (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. https://arxiv.org/abs/1701.06538

Vaswani, A. y otros (2017). *Attention Is All You Need*. https://papers.nips.cc/paper/7181-attention-is-all-you-need

Notas

1. OpenAI. (2026). *Counting tokens*. https://developers.openai.com/api/docs/guides/token-counting. Consultado el 25 de mayo de 2026. ↩

2. OpenAI. (2026). *Prompt caching*. https://developers.openai.com/api/docs/guides/prompt-caching. Consultado el 25 de mayo de 2026. ↩

3. OpenAI. (2026). *Cost optimization*. https://developers.openai.com/api/docs/guides/cost-optimization. Consultado el 25 de mayo de 2026. ↩

4. OpenAI. (2026). *Batch API*. https://developers.openai.com/api/docs/guides/batch. Consultado el 25 de mayo de 2026. ↩

5. OpenAI. (2026). *Latency optimization*. https://developers.openai.com/api/docs/guides/latency-optimization. Consultado el 25 de mayo de 2026. ↩

6. Anthropic. (2026). *Token counting*. https://platform.claude.com/docs/en/build-with-claude/token-counting. Consultado el 25 de mayo de 2026. ↩

7. Anthropic. (2026). *Prompt caching*. https://platform.claude.com/docs/en/build-with-claude/prompt-caching. Consultado el 25 de mayo de 2026. ↩

8. Anthropic. (2026). *Context windows*. https://platform.claude.com/docs/en/build-with-claude/context-windows. Consultado el 25 de mayo de 2026. ↩

9. Google. (2026). *Token counting*. https://ai.google.dev/gemini-api/docs/tokens. Consultado el 25 de mayo de 2026. ↩

10. Google. (2026). *Long context*. https://ai.google.dev/gemini-api/docs/long-context. Consultado el 25 de mayo de 2026. ↩

11. Google. (2026). *Context caching*. https://ai.google.dev/gemini-api/docs/caching. Consultado el 25 de mayo de 2026. ↩

12. Google. (2026). *Gemini Developer API pricing*. https://ai.google.dev/gemini-api/docs/pricing. Consultado el 25 de mayo de 2026. ↩

13. Noam Shazeer y otros (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. ICLR. https://arxiv.org/abs/1701.06538. ↩

14. William Fedus, Barret Zoph y Noam Shazeer. (2022). *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. *Journal of Machine Learning Research, 23*(120), 1-39. https://jmlr.org/papers/v23/21-0998.html. ↩

15. Dmitry Lepikhin y otros (2021). *GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding*. ICLR. https://arxiv.org/abs/2006.16668. ↩

16. Albert Q. Jiang y otros (2024). *Mixtral of Experts*. https://arxiv.org/abs/2401.04088. ↩

17. OpenAI. (2026). *tiktoken*. https://github.com/openai/tiktoken. Consultado el 25 de mayo de 2026. ↩

18. Rico Sennrich, Barry Haddow y Alexandra Birch. (2016). *Neural Machine Translation of Rare Words with Subword Units*. ACL. https://aclanthology.org/P16-1162/. ↩

19. Rico Sennrich, Barry Haddow y Alexandra Birch. (2016). *Neural Machine Translation of Rare Words with Subword Units*. ACL. https://arxiv.org/abs/1508.07909. ↩

20. Taku Kudo y John Richardson. (2018). *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. EMNLP. https://aclanthology.org/D18-2012/. ↩

21. OpenAI. (2026). *Pricing*. https://developers.openai.com/api/docs/pricing. Consultado el 25 de mayo de 2026. ↩

22. OpenAI. (2026). *Pricing*. https://developers.openai.com/api/docs/pricing. Consultado el 25 de mayo de 2026. ↩

23. Ashish Vaswani y otros (2017). *Attention Is All You Need*. NeurIPS. https://papers.nips.cc/paper/7181-attention-is-all-you-need. ↩

24. Joshua Ainslie y otros (2023). *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. EMNLP. https://arxiv.org/abs/2305.13245. ↩

25. Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. https://arxiv.org/abs/2309.06180. ↩

26. Ashish Vaswani y otros (2017). *Attention Is All You Need*. NeurIPS. https://papers.nips.cc/paper/7181-attention-is-all-you-need. ↩

27. Noam Shazeer y otros (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. ICLR. https://arxiv.org/abs/1701.06538. William Fedus, Barret Zoph y Noam Shazeer. (2022). *Switch Transformers*. JMLR, 23(120). https://jmlr.org/papers/v23/21-0998.html. ↩

28. Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. https://arxiv.org/abs/2309.06180. ↩

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.

Tokens, contexto y coste: la factura de un LLM

https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2004/01-tokens-contexto-coste.ipynb