

Facsímil 04

La caja de herramientas

Coleccionable completo · 13 capítulos

686f6c61 · 2026

Índice

01	Elegir la intervención correcta: prompt, RAG, tool o ajuste
02	APIs de modelos: mensajes, streaming y salidas estructuradas
03	Tokens, coste, contexto y caché
04	Model cards y elección de modelos
05	Modelos locales: Ollama, LM Studio, GGUF y cuantización
06	Cloud frente a local: privacidad, latencia y coste
07	Embeddings aplicados y búsqueda semántica
08	Bases vectoriales, filtros y búsqueda híbrida
09	RAG básico: chunking, retrieval, citas y abstención
10	Evaluar RAG: retrieval, groundedness y abstención
11	Agentic RAG y GraphRAG: cuándo complicar
12	Text-to-SQL y herramientas de datos
13	Lo que deberías saber: la caja de herramientas

Capítulo 01: Elegir la intervención correcta: prompt, RAG, tool o ajuste

La herramienta no es el punto de partida

El error más frecuente en IA aplicada es elegir la herramienta antes que el problema: «hagamos RAG», «probemos fine-tuning», «pongamos un agente», «llevémoslo a local». En ingeniería ese salto tiene nombre: es resolver sin haber hecho el análisis de requisitos. Una solución elegida antes del diagnóstico no es una decisión, es una preferencia.

Una herramienta es la respuesta a un síntoma concreto, y cada síntoma apunta a una capa distinta del sistema. Si el modelo no respeta un formato, falla el contrato de salida: salida estructurada. Si no conoce la normativa interna, falta evidencia: RAG. Si debe consultar stock real, falta estado externo: una tool. Si responde bien pero con criterio inestable en miles de casos repetidos, falla la conducta aprendida: ajuste. Si los datos no pueden salir de una máquina, manda el entorno de despliegue: local. La herramienta no se elige por su prestigio, sino por la capa que repara.

Este capítulo, por tanto, se ordena alrededor de una sola pregunta de diseño: **¿qué parte del sistema falla y cuál es el cambio mínimo que la corrige con la mejor relación entre mejora medible y coste de operación?**

Estado del arte con fecha de corte

Fecha de corte: 25 de mayo de 2026.

Fuentes consultadas ese día: documentación oficial de Anthropic sobre prompt engineering, OpenAI sobre salidas estructuradas y function calling, Hugging Face PEFT/LoRA, documentación de Ollama API y Cloud, y el paper original de RAG.

Esta foto cambia rápido. Lo estable no es el nombre de una herramienta concreta, sino el patrón de decisión: contexto, evidencia, schema, tool, ajuste, local, evaluación y coste.

En 2026, las plataformas de modelos ya no ofrecen solo “texto dentro, texto fuera”. Las APIs modernas permiten schemas, tool calls, streaming, batch, cache y distintos modelos para coste o calidad. OpenAI documenta salidas estructuradas para forzar que una respuesta siga un schema.¹ Anthropic mantiene guías de prompt engineering como primera capa de control antes de añadir arquitectura.²

También se consolidó una separación práctica: RAG para conocimiento externo, PEFT/LoRA para adaptar comportamiento con pocos parámetros y modelos locales o cloud local-compatible cuando la restricción está en privacidad, coste, portabilidad o experimentación.³ Ollama documenta una API local y una variante cloud con una experiencia parecida para modelos que no caben en hardware personal.⁴⁵

A 2026 hay además un consenso práctico sobre el orden por defecto: prompt y salida estructurada como primera capa de control, RAG como forma habitual de aportar conocimiento que cambia, tools para estado real y cálculo, y ajuste de pesos solo cuando una conducta repetida no se estabiliza de otra forma. Los flujos agentic (varios pasos, herramientas y memoria) ganan terreno para tareas complejas, pero no cambian la regla de fondo: no se empieza por la pieza más cara, se llega a ella cuando una evaluación lo justifica. La novedad de los últimos años no es una herramienta nueva que sustituya a las demás, sino mejores contratos entre capas (schemas, tool calls tipados, trazas) que hacen que combinar varias sea más fiable y más fácil de depurar.

Qué no es una caja de herramientas de IA

Una caja de herramientas no es una lista de marcas. Si no sabes qué síntoma resuelve cada pieza, el catálogo solo añade ruido. La persona que sabe usar herramientas no es quien instala más librerías, sino quien simplifica antes de complicar.

Tampoco es una escalera de prestigio. Prompt no es “poco serio” y fine-tuning no es “más profesional” por defecto. RAG no es superior a una consulta SQL si la pregunta exige datos exactos. Un modelo local no es mejor que una API si no cumple calidad o mantenimiento. Cada elección compra algo y paga algo.

Y una caja de herramientas no sustituye a evaluación. Puedes montar RAG, tool calling, LoRA y modelo local; si no tienes casos de prueba, no sabes si has mejorado o solo has construido una máquina más difícil de entender.

La pregunta correcta: qué quieres cambiar

Antes de elegir intervención, separa cuatro planos. El primero es **entrada**: quizá el modelo ya puede hacer la tarea si le das mejores instrucciones, ejemplos y formato esperado. El segundo es **contexto**: quizá necesita documentos, datos o memoria externa. El tercero es **acción**: quizá debe consultar una API, calcular algo o escribir en un sistema. El cuarto es **comportamiento aprendido**: quizá quieres que responda de una forma estable en una tarea repetida.

Esa separación evita mezclar herramientas:

SI EL PROBLEMA ES...	PRIMERA INTERVENCIÓN RAZONABLE	POR QUÉ
Formato irregular	Prompt claro y salida estructurada	El modelo sabe responder, pero falta contrato de salida.
Conocimiento vivo	RAG o consulta a fuente externa	El dato cambia y debe poder citarse o verificarse.
Cálculo, estado o acción	Tool con schema y permisos	El modelo no debe inventar resultados de sistemas externos.
Tono o taxonomía repetida	Ejemplos, luego fine-tuning si escala	Cambias comportamiento estable, no conocimiento vivo.
Privacidad u offline	Modelo local o entorno privado	La restricción vive en dónde se ejecuta.
Coste o latencia	Modelo más pequeño, cache, batch o local	El problema puede estar en serving, no en "inteligencia".

Para verlo con un caso cercano: una universidad quiere contestar dudas de matrícula. Si las reglas cambian cada curso, RAG o consulta a la base oficial. Si el problema es devolver siempre JSON para integrarlo en una app, salida estructurada. Si debe comprobar si una persona ya pagó, tool contra el sistema de pagos. Si cada respuesta debe tener tono institucional, ejemplos y quizá ajuste más adelante.

¿Qué capa del sistema falla? Ahí actúa la intervención

No cambies de modelo por costumbre: cambia la capa que duele.

Entrada
formato o criterio irregular

Prompt y ejemplos · Salida estructurada

Contexto
conocimiento vivo, hay que citar

RAG: recuperar evidencia externa

Acción
necesita estado real o cálculo

Tool con schema y permisos

Conducta aprendida
tarea repetida, tono inestable

Ajuste: SFT o LoRA

Entorno
privacidad, offline, coste

Modelo local o entorno privado

IA para gente curiosa / Facsímil 04 / Capítulo 01 / 686f6c61

El criterio de decisión: utilidad esperada y valor de la información

Elegir una intervención bajo incertidumbre no es una corazonada: es un problema clásico de teoría de la decisión, y conviene usar su formalismo en lugar de una regla casera.

El primer pilar es la **utilidad esperada**. Una decisión racional elige la acción que maximiza la utilidad esperada sobre los resultados posibles, formalizada por von Neumann y Morgenstern.⁶

$$\mathbb{E}[U(a)] = \sum_s P(s) U(a, s)$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
a	Acción o intervención que evaluamos.	Añadir RAG, una tool o un schema.
s	Estado o resultado posible del sistema.	El dato estaba vigente o no, el formato se valida o no.
$P(s)$	Probabilidad de cada resultado.	Estimada con casos reales, no supuesta.
$U(a, s)$	Utilidad de la acción si ocurre s .	Respuesta correcta y citada frente a respuesta inventada.
$\mathbb{E}[U(a)]$	Utilidad esperada de la acción.	Promedio ponderado de los resultados.

La regla de decisión es directa: adoptas una intervención frente al baseline solo si su utilidad esperada, descontado el coste total, supera a la de no hacer nada.

$$\mathbb{E}[U(a)] - K(a) > \mathbb{E}[U(\text{baseline})]$$

Aquí $K(a)$ agrega el coste real de operar la intervención: económico, latencia, mantenimiento y riesgo. Esto evita la mala costumbre de contar solo la mejora y olvidar lo que habrá que sostener después.

El segundo pilar explica por qué RAG, las tools o cualquier recuperación de evidencia tienen sentido. No cambian el modelo: le dan información antes de decidir. La teoría del **valor de la información** de Howard cuantifica cuánto vale esa evidencia.⁷

$$\text{VoI}(e) = \mathbb{E}_e \left[\max_a \mathbb{E}[U(a) \mid e] \right] - \max_a \mathbb{E}[U(a)] \geq 0$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
e	Evidencia que se recupera antes de responder.	El fragmento de normativa vigente, el stock real.
$\mathbb{E}[U(a) \mid e]$	Utilidad esperada de la acción dada esa evidencia.	Decidir con el documento delante.
$\text{VoI}(e)$	Valor de la información de esa evidencia.	Cuánto mejora la decisión por tenerla.

En palabras: añadir RAG o una tool solo merece la pena si la evidencia que aportan **cambia lo que el sistema haría**. Si la respuesta es la misma con o sin ese documento, su valor de información es cero, aunque la recuperación funcione a la perfección. Por eso el diagnóstico no pregunta «¿puedo recuperar algo?», sino «¿hay una decisión que esa evidencia modifique?».

Veámoslo con números. Un asistente debe decidir si una solicitud encaja en una política que puede decir «aprobar» o «denegar». Sin leer el documento, su creencia a priori es $P(\text{aprobar}) = 0,6$ y $P(\text{denegar}) = 0,4$. Las utilidades del caso: acertar vale +10, equivocarse -20 (una respuesta de política errónea es cara) y abstenerse para escalar a una persona vale +2.

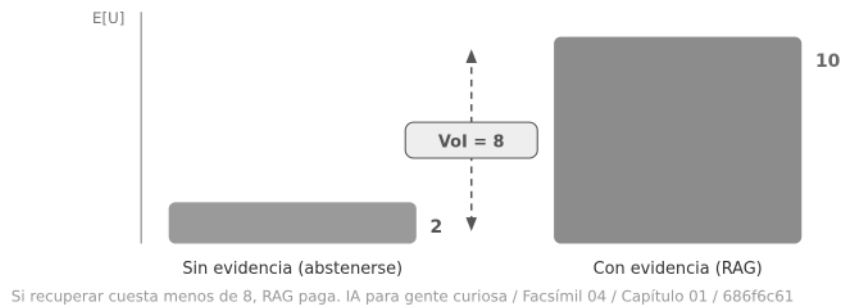
ACCIÓN SIN EVIDENCIA	UTILIDAD ESPERADA
Decir «aprobar»	$0,6 \cdot 10 + 0,4 \cdot (-20) = -2$
Decir «denegar»	$0,6 \cdot (-20) + 0,4 \cdot 10 = -8$
Abstenerse y escalar	+2

Sin evidencia, la mejor acción es **abstenerse**, con $\mathbb{E}[U] = 2$. Ahora recuperamos el documento, que revela el estado real: con él, el asistente siempre elige la acción correcta y obtiene +10 en cualquiera de los dos estados, así que $\mathbb{E}[U \mid e] = 10$. El valor de la información de ese documento es:

$$\text{VoI}(e) = 10 - 2 = 8$$

El valor de la información, en números

Recuperar el documento sube la utilidad esperada de 2 a 10: vale hasta 8.



Ese 8 es el techo de lo que deberías estar dispuesto a pagar por recuperar: si el coste de RAG (tokens, latencia, mantener el índice) es menor que 8, la intervención paga; si es mayor, no. Y observa el otro extremo: si equivocarse costara solo -1 en vez de -20 , responder «aprobar» sin leer nada ya daría $0,6 \cdot 10 + 0,4 \cdot (-1) = 5,6$, mejor que abstenerse, y la evidencia apenas cambiaría la decisión. El valor de la información cae cuando el coste de equivocarse es bajo. Esa es la diferencia entre añadir RAG porque toca y añadirlo porque mueve una decisión cara.

La fórmula no pretende dar una verdad exacta. Pretende impedir una mala costumbre: contar solo la mejora y olvidar todo lo que tendrás que operar después.

Las cinco intervenciones básicas

Primero hay que explicar el síntoma. Luego la herramienta empieza a tener sentido. Estas cinco piezas vuelven durante todo el facsímil.

La tabla siguiente es una síntesis didáctica, no una taxonomía oficial única. Se apoya en [prompt engineering⁸](#) para controlar tarea, contexto y ejemplos; en [salidas estructuradas⁹](#) para imponer contratos de respuesta; en [RAG¹⁰](#) para añadir evidencia recuperada; en [function calling¹¹](#) para conectar el modelo con funciones externas; y en [LoRA/PEFT¹²](#) para adaptar comportamiento mediante pocos parámetros entrenables.

INTERVENCIÓN	CAMBIA	SIRVE CUANDO	NO SIRVE PARA
Prompt y ejemplos	La entrada	Falta claridad, formato o criterio de respuesta.	Datos vivos grandes o acciones externas.
Salida estructurada	El contrato de salida	Necesitas JSON, campos obligatorios o validación automática.	Saber más contenido.
RAG	El contexto recuperado	Hay documentos, políticas o conocimiento cambiante.	Cambiar el estilo profundo del modelo.
Tool	La capacidad de consultar o actuar	Hay estado real, cálculo, base de datos o sistema externo.	Sustituir permisos o validación.
Fine-tuning/LoRA	Algunos pesos o adaptadores	Tarea repetida, estable y medible.	Actualizar información que cambia cada día.

RAG nació como una forma de combinar memoria paramétrica con recuperación externa en tareas intensivas en conocimiento.¹³ LoRA propuso adaptar modelos grandes entrenando matrices pequeñas de bajo rango.¹⁴ QLoRA redujo aún más memoria al ajustar sobre modelos cuantizados.¹⁵

Para entenderlo antes de tocar código

Antes de escribir una línea de código conviene hacer una prueba mental muy simple: describe el fallo como lo vería una persona usuaria, no como lo nombraría una librería. Después tradúcelo a una capa del sistema. Esa traducción es el músculo que queremos entrenar en este capítulo.

La misma frase “el modelo falla” puede significar cinco cosas distintas. Puede fallar porque no recibió instrucciones claras, porque le falta evidencia, porque necesita consultar un sistema externo, porque debe repetir un criterio muy específico o porque el entorno de despliegue impone límites. Si llamas a todo “modelo malo”, acabarás cambiando piezas equivocadas.

CASO	SEÑAL OBSERVABLE	DIAGNÓSTICO	PRIMERA INTERVENCIÓN	QUÉ MEDIR
Asesoría con normativa cambiante	Responde bien hasta que cambia una norma.	Falta evidencia viva y trazable.	RAG o consulta a fuente oficial con cita.	Porcentaje de respuestas con documento correcto y fecha vigente.
CRM que necesita campos exactos	El texto es bueno, pero el backend rompe al parsear.	Falta contrato de salida.	Salida estructurada con schema y validación.	Campos válidos, errores de schema y casos que requieren revisión.
Tienda con stock real	Preguntan por talla, precio o disponibilidad.	Hace falta estado externo.	Tool pequeña contra inventario o pedidos.	Exactitud del dato, latencia de consulta y manejo de “no disponible”.
Soporte con miles de tickets parecidos	Responde con criterio parecido, pero formato y tono varían.	Conducta repetida poco estable.	Prompt con ejemplos; si escala, SFT, LoRA o adapter.	Consistencia de rúbrica, coste por ticket y regresiones.
Equipo con documentos sensibles	La mejor API externa no puede recibir ciertos textos.	Restricción de entorno.	Modelo local, entorno privado o arquitectura híbrida.	Calidad mínima aceptable, latencia, coste operativo y trazabilidad.

Fíjate en el tercer caso. Si una persona pregunta “¿quedan zapatillas talla 42?”, el modelo puede escribir una respuesta convincente, pero no conoce el almacén. La intervención correcta no es pedirle que “razone mejor”. Es darle una función con un contrato estrecho: `consultar_stock(producto, talla, tienda)`. La inteligencia del sistema no está solo en el modelo; está en saber cuándo el modelo debe dejar de completar texto y pedir un dato.

Ahora mira el primer caso. Si la asesoría trabaja con normas que cambian, ajustar pesos puede incluso empeorar la situación: convierte conocimiento vivo en memoria opaca. RAG no entra porque sea una palabra de moda, sino porque necesitamos tres cosas muy concretas: recuperar el fragmento vigente, citarlo y poder actualizarlo sin entrenar de nuevo.

El caso del CRM enseña otra lección. A veces el modelo “entiende” perfectamente la tarea, pero el producto necesita datos que se puedan validar. Ahí no queremos literatura: queremos `{categoria, prioridad, siguiente_paso, confianza}` y una regla clara para rechazar respuestas inválidas. La salida estructurada no mejora el mundo interno del modelo; mejora el contrato entre el modelo y el software que lo rodea.

Y el equipo de soporte muestra cuándo empieza a tener sentido ajustar. Si el conocimiento ya está resuelto, los documentos aparecen bien y el schema se cumple, pero la respuesta no respeta una rúbrica interna en miles de ejemplos parecidos, entonces sí: quizá toca entrenar una adaptación pequeña. Pero esa decisión llega después de medir, no antes.

Una regla útil: **si puedes arreglarlo cambiando entrada, contexto o contrato, no empieces cambiando pesos**. Los pesos se tocan cuando quieres estabilizar una conducta repetida, tienes ejemplos buenos, sabes medir el resultado y aceptas mantener otra versión del sistema.



Criterios de elección: la matriz que decide contigo

Si esto fuera una asignatura universitaria, aquí no bastaría con decir “depende”. El “depende” tiene que descomponerse en variables observables. Una buena decisión técnica no es la que suena más moderna, sino la que explica qué restricción pesa más y qué evidencia aceptaríamos para cambiar de opinión.

Esta es la forma práctica de la utilidad esperada cuando hay varios criterios en juego: el **análisis multicriterio**. Su versión más simple, el Weighted Sum Model, puntúa cada intervención como la suma de sus valoraciones por criterio, ponderadas por la importancia de cada criterio.¹⁶¹⁷

$$\text{score}(a) = \sum_j w_j r_{aj}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a	Intervención evaluada.	RAG, tool, ajuste, local.
j	Criterio de decisión.	Conocimiento vivo, coste, privacidad.
w_j	Peso o importancia del criterio j .	Cuánto pesa la privacidad en tu caso.
r_{aj}	Valoración de la intervención a en el criterio j .	Del 0 al 5.
$\text{score}(a)$	Puntuación total de la intervención.	La que ordena el ranking.

Es justo lo que practicas en el cuaderno del facsímil: defines los pesos w_j de tu caso y dejas que el modelo ordene. Su límite académico también conviene decirlo: el Weighted Sum Model solo es válido si los criterios son comparables y están en la misma escala; si un criterio es de naturaleza distinta (por ejemplo, un requisito legal que no se puede compensar con coste), no entra en la suma, sino que actúa como restricción dura.

La siguiente matriz sirve para discutir una intervención en clase, en un equipo o en una revisión de arquitectura. No da una respuesta automática, pero obliga a justificarla.

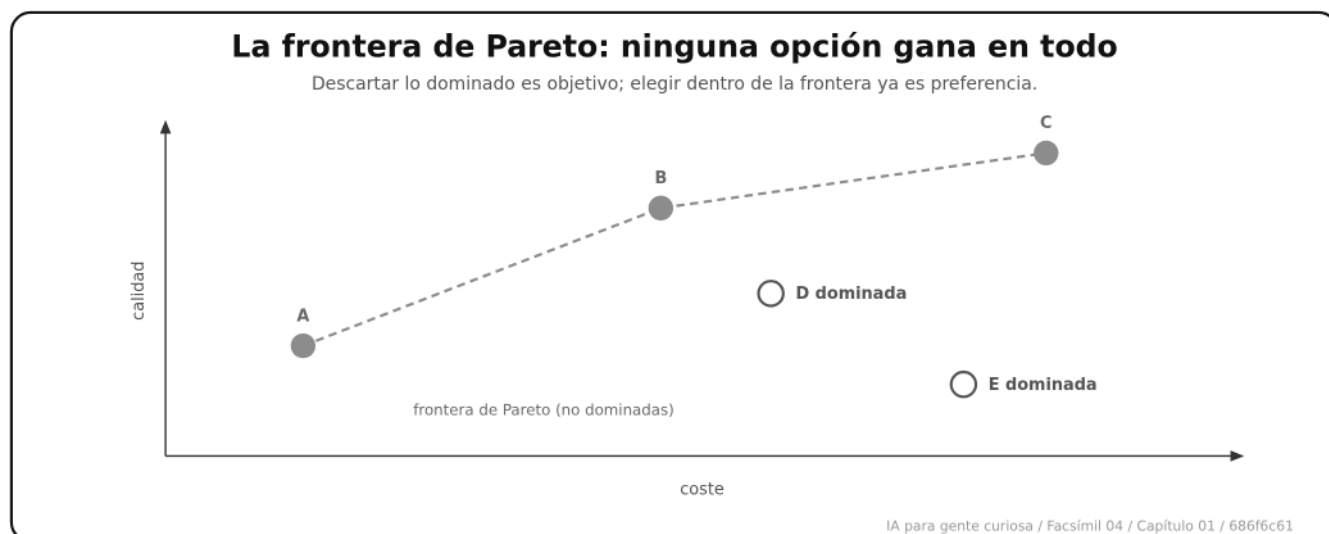
CRITERIO	PREGUNTA QUE DEBES HACER	SI PESA MUCHO, SUELE EMPUJAR HACIA...	EVIDENCIA MÍNIMA
Conocimiento cambiante	¿La respuesta depende de datos que cambian con frecuencia?	RAG, base de datos o tool.	Documentos con fecha, fuente y casos donde el dato cambia.
Necesidad de cita	¿La persona debe poder revisar de dónde sale la respuesta?	RAG con citas o consulta verificable.	Porcentaje de respuestas con evidencia correcta.
Estado real	¿Hace falta mirar inventario, pagos, agenda, expediente o cálculo externo?	Tool con schema estrecho.	Función definida, entrada validada y salida comprobable.
Formato estricto	¿Otro software consume la respuesta?	Salida estructurada.	JSON válido, campos obligatorios y tests de schema.
Conducta repetida	¿La misma tarea aparece miles de veces con criterios estables?	Prompt con ejemplos; si no basta, SFT, LoRA o adapter.	Dataset pequeño pero limpio, rúbrica y comparación contra baseline.
Privacidad o despliegue	¿Dónde puede ejecutarse el sistema y dónde pueden vivir los datos?	Modelo local, entorno privado o arquitectura híbrida.	Política de datos, latencia aceptable y calidad mínima medida.
Coste y latencia	¿El problema es calidad o servirlo sin arruinar la experiencia?	Modelo menor, cache, batch, cuantización o local.	Coste por consulta, TTFT, tokens/s y percentiles de latencia.
Reversibilidad	¿Podemos deshacer el cambio si empeora?	Prompt, schema, RAG o tool antes que ajuste de pesos.	Plan de rollback y comparación antes/después.

Esta matriz también evita discusiones tramposas. Si alguien propone fine-tuning para un problema de stock, puedes preguntar: “¿qué criterio de la tabla justifica cambiar pesos?”. Si nadie puede responder, todavía no hay diagnóstico.

La frontera de Pareto: cuando ninguna opción gana en todo

El Weighted Sum Model da un único ganador porque colapsa todo en un número. Pero esconde una decisión: los pesos. A veces no existe una opción que domine a las demás en todos los criterios; una es más barata, otra más precisa. Para esos casos la herramienta correcta no es la suma ponderada, sino la **frontera de Pareto**, el concepto central de la optimización multiobjetivo.¹⁸

Una opción está **dominada** si otra es al menos igual de buena en todos los criterios y estrictamente mejor en alguno. Las opciones que nadie domina forman la frontera de Pareto. Lo útil es que descartar lo dominado no exige elegir pesos: es objetivo. Solo cuando te quedas en la frontera aparece el juicio, porque allí toda mejora en un criterio cuesta empeorar otro.



En el dibujo, B domina a D: es más barata y de más calidad, así que D no debería ni discutirse. A, B y C, en cambio, son defendibles: A es la más barata, C la de más calidad y B el punto intermedio. Elegir entre ellas es declarar cuánto vale para ti un punto extra de calidad. La frontera no decide, pero limpia la mesa de las opciones que nadie debería elegir.

Un mismo caso, cinco soluciones posibles

Tomemos un caso único para no perdernos: una universidad quiere un asistente de matrícula. El objetivo superficial parece uno solo, “responder dudas”, pero debajo hay varios problemas distintos. Según cuál duela, la intervención cambia.

LECTURA DEL PROBLEMA	SOLUCIÓN RAZONABLE	QUÉ MEJORA	QUÉ NO ARREGLA
El alumnado pregunta de formas muy distintas y el sistema responde desordenado.	Prompt con ejemplos y criterios de estilo.	Claridad, tono, estructura inicial.	Normativa nueva o datos personales.
La app necesita guardar la respuesta en una ficha.	Salida estructurada con campos como <code>tema</code> , <code>respuesta</code> , <code>fuentes</code> , <code>confianza</code> .	Integración con software y validación.	Saber si la norma está vigente.
Las normas de matrícula cambian cada curso.	RAG sobre normativa oficial, con fecha y cita.	Respuestas trazables y actualizables.	Consultar si una persona concreta pagó.
El alumno pregunta “¿me falta pagar algo?”.	Tool contra el sistema académico o de pagos.	Estado real de esa persona.	Explicar bien una norma general.
El equipo responde miles de tickets con una rúbrica estable.	Ajuste ligero si prompt, schema y RAG ya no bastan.	Consistencia en una tarea repetida.	Conocimiento que cambia cada semana.

Lo importante es que ninguna fila “gana” siempre. En un producto real quizá uses varias: RAG para normativa, tool para expediente, salida estructurada para integrar y prompt para tono. Pero las añades por capas, no por entusiasmo.

Combinar intervenciones: el orden importa

Un sistema real casi nunca usa una sola intervención. Un asistente de atención puede tener prompt cuidado, salida estructurada para integrar, RAG para políticas y una tool para consultar pedidos. La pregunta no es «¿cuál uso?», sino «¿en qué orden las añado y cómo sé que cada capa ayuda?».

La regla práctica es añadir de barata a cara, y medir entre capa y capa. Cada intervención nueva tiene que justificar su coste con una mejora visible en la evaluación; si no la mueve, sobra. Así se evita el sistema que tiene de todo y no se sabe por qué funciona ni por qué falla.

ORD EN	CAPA QUE AÑADES	ANTES DE AÑADIRLA, COMPRUEBA	LA EVAL QUE LA JUSTIFICA
1	Prompt y ejemplos	Que el modelo base no resuelve ya la tarea con instrucciones claras.	Tasa de respuestas útiles sobre 30 casos.
2	Salida estructurada	Que otro software consume la respuesta y el formato rompe.	JSON válido y campos correctos.
3	RAG	Que el fallo es de conocimiento vivo, no de formato ni de tono.	Evidencia correcta y abstención cuando falta fuente.
4	Tool	Que hace falta estado real o cálculo que el modelo no debe inventar.	Exactitud de la llamada y manejo de «no disponible».
5	Ajuste	Que prompt, schema y RAG ya no mueven la conducta repetida.	Mejora contra baseline sin regresiones.

El orden no es dogma, pero su lógica sí: las capas de abajo son más baratas y reversibles, y muchas veces hacen innecesarias las de arriba. Si añades RAG y la tarea mejora, quizá ya no necesitas ajustar. Si saltas directo a fine-tuning, puede que estés pagando la capa más cara para arreglar un problema que un schema resolvía.

Una advertencia: combinar no es acumular. Cada capa añade superficie que mantener, evaluar y depurar. Dos intervenciones bien medidas valen más que cinco puestas por entusiasmo. La señal de un buen sistema no es cuántas herramientas tiene, sino que cada una se pueda explicar con la eval que la sostiene.

Cómo evaluar cada intervención

Una intervención no está terminada cuando funciona en la demo. Está terminada cuando puedes repetir una prueba y decidir si mejoró. Si no sabes qué medir, la arquitectura se convierte en opinión.

INTERVENCIÓN	MÉTRICA PRINCIPAL	PRUEBA MÍNIMA	SEÑAL DE QUE NO BASTA
Prompt y ejemplos	Tasa de respuestas útiles según rúbrica.	30 casos reales antes/después.	Mejora solo en ejemplos vistos o se rompe con variaciones simples.
Salida estructurada	Validez de schema y tasa de campos correctos.	Tests con campos obligatorios, tipos y casos incompletos.	El JSON es válido pero el contenido sigue siendo incorrecto.
RAG	Evidencia correcta, cobertura y abstención cuando falta fuente.	Preguntas con documento esperado y preguntas sin respuesta en corpus.	Recupera texto parecido pero no el fragmento que justifica la respuesta.
Tool	Exactitud de llamada, validación de argumentos y latencia.	Casos con entradas válidas, incompletas y ambiguas.	La tool se invoca cuando no toca o con parámetros mal formados.
Fine-tuning/LoRA	Mejora contra baseline sin perder casos importantes.	Eval fija antes/después y revisión de regresiones.	Mejora el formato pero empeora factualidad o flexibilidad.
Modelo local o privado	Calidad mínima, latencia, coste y mantenimiento.	Misma eval que la API base, con medición de recursos.	Cumple privacidad pero no alcanza la calidad necesaria.

Esas métricas no son vagas: tienen definición formal y conviene usar la real. Para recuperación, la métrica estándar es el `recall@k`, la fracción de consultas cuyo documento relevante aparece entre los k recuperados.¹⁹ Para RAG, la *groundedness* mide qué fracción de las afirmaciones de la respuesta están soportadas por la evidencia citada.²⁰

$$\text{recall}@k = \frac{1}{N} \sum_{i=1}^N 1[d_i^* \in R_k(q_i)]$$

INTERVENCIÓN	MÉTRICA FORMAL	DEFINICIÓN
Salida estructurada	Tasa de validez de schema	Respuestas que validan contra el schema, sobre el total.
RAG (recuperación)	recall@k	Consultas con el documento relevante en el top-k, sobre el total.
RAG (generación)	Groundedness	Afirmaciones soportadas por la evidencia citada, sobre el total.
Tool	Exactitud de llamada	Llamadas con nombre y argumentos correctos, sobre el total.

Y un punto que separa una medición de una anécdota: toda métrica es una **proporción estimada con una muestra**, así que tiene incertidumbre. Para n casos y una tasa observada $\hat{p}$, el intervalo de confianza al 95 % por la aproximación normal (intervalo de Wald) es:²¹

$$\hat{p} \pm 1,96 \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

Con $\hat{p} = 0,8$ y $n = 100$ el margen es $\pm 0,08$; con $n = 30$, $\pm 0,14$. Por eso una mejora de dos puntos medida sobre 30 casos no es señal: cabe entera dentro del ruido. La eval mínima de 30 casos vale para cazar fallos gruesos; afirmar una mejora fina exige más muestra o una comparación pareada.

En clase, yo pediría siempre tres números antes de aceptar una propuesta: calidad, coste y reversibilidad. Calidad sin coste puede ser inviable. Coste sin calidad no sirve. Y una mejora que no puedes revertir exige mucha más evidencia.

Errores de diagnóstico que conviene detectar

Estos errores parecen razonables cuando tienes prisa, por eso conviene nombrarlos. No son fallos de principiante: aparecen en equipos buenos cuando el problema se describe demasiado pronto con el nombre de una herramienta.

ERROR DE DIAGNÓSTICO	CÓMO SE VE	CÓMO CORREGIRLO
Confundir conocimiento con comportamiento	"Ajustemos el modelo para que sepa la normativa nueva".	Si cambia con frecuencia, sácalo a documentos, base de datos o tool.
Confundir formato con inteligencia	"El modelo no entiende", cuando lo único que falla es el JSON.	Primero schema, validación y ejemplos de salida.
Confundir búsqueda con respuesta	El retrieval trae documentos parecidos, pero no justifican la conclusión.	Evaluar recuperación por fragmento correcto, no solo por similitud.
Confundir acción con texto	El modelo "dice" que ha comprobado algo, pero no ha consultado ningún sistema.	Tool real, logs y permisos mínimos.
Confundir benchmark con caso propio	Se elige modelo por ranking general.	Eval con idioma, dominio, coste y latencia del proyecto.
Confundir privacidad con peor producto	"Local" se acepta sin medir calidad.	Comparar local, privado y API con la misma rúbrica.

El antídoto común es escribir una frase de diagnóstico antes de escribir una frase de solución: "El sistema falla porque...". Si esa frase ya contiene el nombre de una herramienta, sospecha un poco.

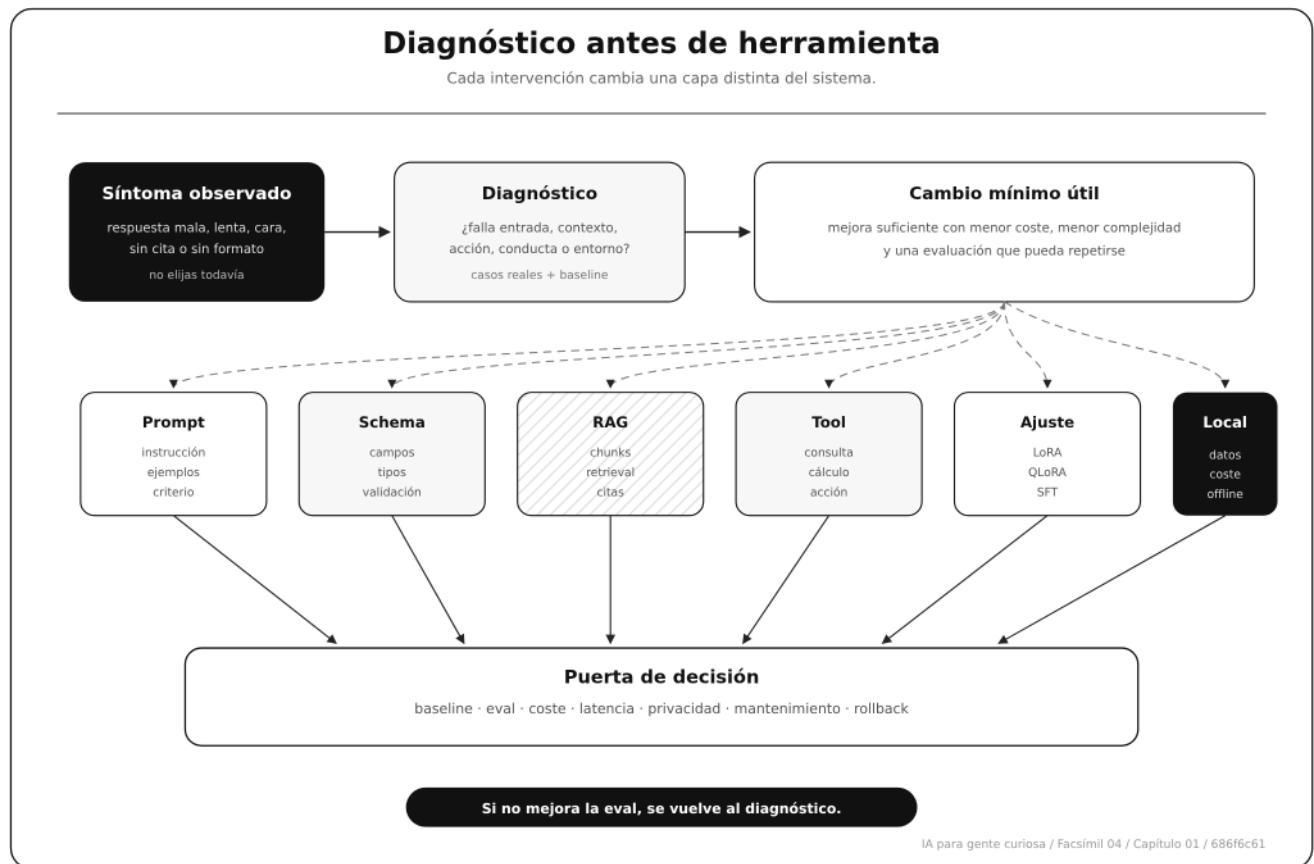
Mini práctica de decisión

Para entrenar el criterio, resuelve estos cuatro casos sin programar. En cada uno escribe: síntoma, intervención principal, alternativa descartada y métrica de aceptación. Después compara con la solución modelo.

CASO	SÍNTOMA	INTERVENCIÓN PRINCIPAL	ALTERNATIVA DESCARTADA	MÉTRICA DE ACEPTACIÓN
Biblioteca universitaria con horarios cambiantes.	Responde horarios antiguos.	RAG o consulta a fuente oficial.	Fine-tuning.	Respuestas con horario vigente y fuente correcta.
App médica que necesita clasificar mensajes en tres colas internas.	El texto es bueno, pero la integración falla.	Salida estructurada.	RAG.	JSON válido y cola correcta según rúbrica humana.
Ecommerce con preguntas de disponibilidad por tienda.	El modelo estima stock.	Tool de inventario.	Prompt más insistente.	Stock correcto, latencia aceptable y manejo de ausencia de dato.
Equipo legal con contratos sensibles en portátiles sin conexión.	No puede enviar documentos fuera y necesita asistencia básica.	Modelo local cuantizado o entorno privado.	API externa directa.	Calidad mínima en una eval interna y tiempo de respuesta usable.

La solución modelo no pretende cerrar todos los matices. Pretende mostrar el razonamiento: cada respuesta identifica la capa que falla. Si cambias el enunciado, puede cambiar la intervención. Si la biblioteca también necesita guardar campos exactos, añadirías salida estructurada. Si el ecommerce además debe explicar políticas de devolución, quizá combinarías tool con RAG. La arquitectura final puede tener varias piezas; el diagnóstico inicial decide cuál entra primero.

Mapa visual de diagnóstico



En el día a día

En un equipo real, este capítulo se usa antes de abrir el editor. Pones encima de la mesa tres cosas: el caso de uso, diez ejemplos reales y una forma de medir. Solo entonces eliges herramienta.

Si un jefe de producto pide “un chatbot con todos los documentos”, tradúcelo: quizá quiere búsqueda con citas, quizá quiere navegación guiada, quizá quiere reducir tickets, quizá quiere extraer campos. Cada objetivo produce una arquitectura distinta.

Si un equipo técnico dice “hagamos RAG”, pregunta por el corpus, el tipo de preguntas, el criterio de cita, los permisos por documento y cómo sabremos que el retrieval encontró evidencia. Si esas respuestas no existen, todavía no hay arquitectura: hay deseo.

Por qué debería importarte

Porque cada intervención mal elegida deja deuda. Un RAG innecesario añade índices, chunks y evals. Un fine-tuning prematuro añade dataset, entrenamiento y versiones. Una tool amplia añade permisos y validación. Un modelo local añade soporte, hardware y medición de calidad.

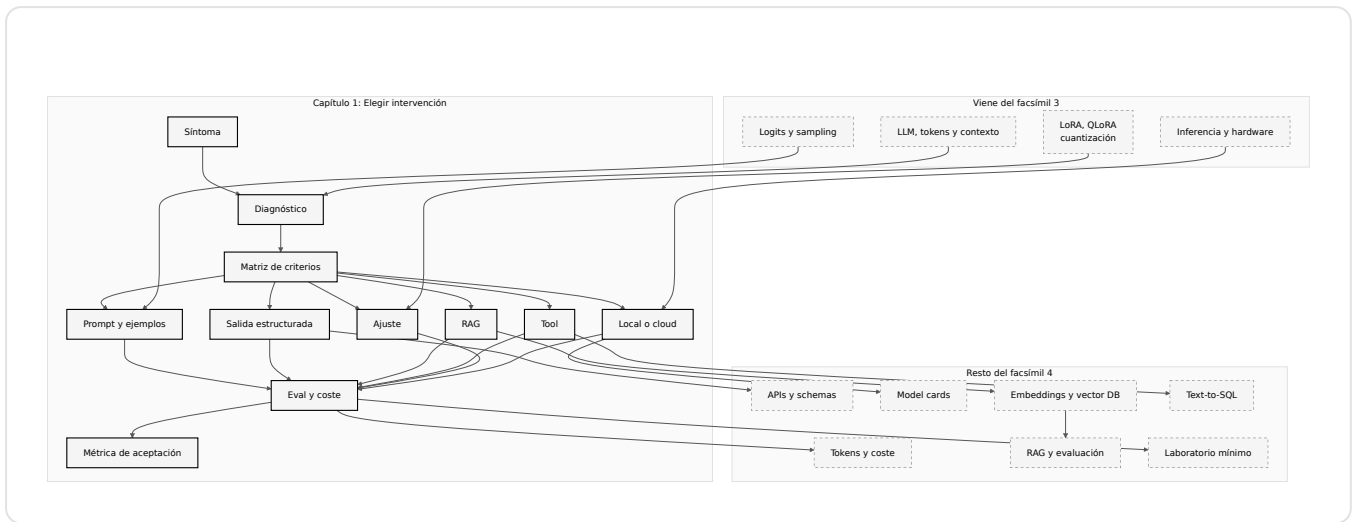
La buena noticia es que una intervención bien elegida suele simplificar. Un schema puede eliminar cientos de líneas de parsing frágil. Una tool puede evitar que el modelo invente un dato. Un RAG con citas puede convertir una respuesta bonita en una respuesta revisable.

Dónde solía tropezar yo

Estos tropiezos aparecen mucho al empezar proyectos con IA. Casi todos nacen de confundir síntoma con solución.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Empezar por RAG sin corpus claro	Si no sabes qué documentos, preguntas y citas necesitas, el índice será decoración cara.	Definir 30 preguntas reales antes de indexar.
Ajustar pesos para datos vivos	El dato que cambia mañana no debería quedar escondido en un adapter.	Usar RAG, base de datos o tool para conocimiento cambiante.
Pedir formato con súplicas	“Responde en JSON” no es contrato suficiente si el backend depende de campos exactos.	Usar salida estructurada y validación.
Usar una tool para todo	Una función gigante es difícil de validar y fácil de usar mal.	Tools pequeñas, schemas claros y permisos mínimos.
Comparar modelos sin tarea	Un ranking genérico no sabe qué coste, idioma, latencia o riesgo tiene tu producto.	Comparar con tus casos, tus métricas y tus límites.

Cómo encaja todo



Vocabulario aprendido

Estas palabras aparecerán en todo el facsímil. Conviene que queden limpias desde el principio.

TÉRMINO	DEFINICIÓN
Intervención	Cambio concreto en el sistema para mejorar un síntoma medible.
Prompt	Entrada que define tarea, contexto, ejemplos y criterio.
Schema	Estructura esperada de una salida o una tool.
Salida estructurada	Respuesta obligada a cumplir un formato validable.
RAG	Recuperar evidencia externa y pasarla al modelo como contexto.
Tool	Función externa que consulta, calcula o modifica algo bajo reglas.
Fine-tuning	Ajustar pesos de un modelo con datos propios.
LoRA	Ajuste eficiente con matrices pequeñas de bajo rango.
Modelo local	Modelo ejecutado en una máquina o infraestructura controlada.
Baseline	Comparación mínima antes de añadir complejidad.
Eval	Prueba repetible que mide calidad, coste, latencia o seguridad del sistema.
Matriz de decisión	Tabla que obliga a justificar una elección con criterios observables.
Reversibilidad	Facilidad para volver atrás si una intervención empeora el sistema.
Abstención	Capacidad de reconocer que falta evidencia suficiente para responder.

Antes de pasar página

- ☐ ¿Puedo explicar por qué no se empieza eligiendo herramienta? (Si no, vuelve a «La herramienta no es el punto de partida».)
- ☐ ¿Sé distinguir si un problema es de entrada, contexto, acción, conducta o entorno? (Si no, vuelve a «La pregunta correcta: qué quieres cambiar».)
- ☐ ¿Puedo decir cuándo basta prompt/schema y cuándo hace falta RAG? (Si no, vuelve a «Las cinco intervenciones básicas».)
- ☐ ¿Entiendo por qué fine-tuning no es buena memoria para datos vivos? (Si no, vuelve a «Para entenderlo antes de tocar código».)
- ☐ ¿Sé cuándo una tool es más adecuada que pedirle al modelo que “razone”? (Si no, vuelve a «Para entenderlo antes de tocar código».)

- ☐ ¿Puedo explicar una decisión con utilidad esperada y valor de la información en vez de con una razonada? (Si no, vuelve a «El criterio de decisión: utilidad esperada y valor de la información».)
- ☐ ¿Puedo justificar una intervención usando criterios como cita, estado real, formato, coste y reversibilidad? (Si no, vuelve a «Criterios de elección: la matriz que decide contigo».)
- ☐ ¿Sé resolver un mismo caso con prompt, schema, RAG, tool o ajuste según el síntoma? (Si no, vuelve a «Un mismo caso, cinco soluciones posibles».)
- ☐ ¿Puedo proponer una métrica mínima para evaluar cada intervención? (Si no, vuelve a «Cómo evaluar cada intervención».)
- ☐ ¿Sé detectar cuándo estoy confundiendo conocimiento, formato, acción o despliegue? (Si no, vuelve a «Errores de diagnóstico que conviene detectar».)
- ☐ ¿Sé ajustar los pesos de la matriz para que gane otra intervención y por qué? (Si no, repasa «Criterios de elección: la matriz que decide contigo».)

En resumen

La caja de herramientas empieza por diagnóstico. Si eliges bien el problema, la herramienta suele volverse evidente.

IDEA FUERZA	DETALLE
La herramienta no es el punto de partida.	Primero síntoma, casos reales y baseline.
Prompt/schema arreglan contrato de entrada y salida.	No hacen que el modelo sepa datos vivos.
RAG aporta evidencia externa.	Sirve para documentos cambiantes y respuestas citables.
Tool conecta con estado real.	Sirve para consultar, calcular o actuar sin inventar.
Ajustar pesos cambia comportamiento.	Tiene sentido en tareas repetidas, estables y medibles.
Local/cloud son decisiones de entorno.	Privacidad, coste, latencia y mantenimiento mandan.
Eval decide si la intervención se queda.	Sin medición, solo hay impresión.
Una buena decisión debe poder explicarse.	Criterio, alternativa descartada, métrica y rollback forman parte de la respuesta.

Para saber más

Anthropic. (2026). *Prompt engineering overview*. <https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/overview>

Dettmers, T. y otros (2023). QLoRA: Efficient Finetuning of Quantized LLMs. *Advances in Neural Information Processing Systems 36*. <https://arxiv.org/abs/2305.14314>

Hugging Face. (2026). *PEFT: LoRA developer guide*.
https://huggingface.co/docs/peft/developer_guides/lora

Hu, E. J. y otros (2022). LoRA: Low-Rank Adaptation of Large Language Models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2106.09685>

Lewis, P. y otros (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474.
<https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>

Ollama. (2026). *Introduction to the Ollama API*. <https://docs.ollama.com/api/introduction>

Ollama. (2026). *Ollama Cloud*. <https://docs.ollama.com/cloud>

OpenAI. (2026). *Function calling*. <https://developers.openai.com/api/docs/guides/function-calling>

OpenAI. (2026). *Structured model outputs*.
<https://developers.openai.com/api/docs/guides/structured-outputs>

Notas

1. OpenAI. (2026). *Structured model outputs*.
<https://developers.openai.com/api/docs/guides/structured-outputs>. Consultado el 25 de mayo de 2026. La guía distingue entre salidas estructuradas para respuesta final y function calling cuando el modelo se conecta a herramientas o datos. ↵
2. Anthropic. (2026). *Prompt engineering overview*. <https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/overview>. Consultado el 25 de mayo de 2026. La guía trata el prompt como mecanismo de especificación de tarea, contexto, ejemplos y restricciones. ↵
3. Hugging Face. (2026). *PEFT: LoRA developer guide*.
https://huggingface.co/docs/peft/developer_guides/lora. Consultado el 25 de mayo de 2026. PEFT documenta LoRA y variantes para entrenar adaptadores pequeños sobre modelos base. ↵

4. Ollama. (2026). *Introduction to the Ollama API*. <https://docs.ollama.com/api/introduction>. Consultado el 25 de mayo de 2026. La documentación indica la URL local por defecto y la URL cloud compatible. ↵
5. Ollama. (2026). *Ollama Cloud*. <https://docs.ollama.com/cloud>. Consultado el 25 de mayo de 2026. La guía describe modelos cloud que se ejecutan sin requerir una GPU local potente. ↵
6. von Neumann, J. y Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press. El teorema de la utilidad esperada establece que, bajo axiomas de racionalidad, las preferencias se representan maximizando la esperanza de una función de utilidad. ↵
7. Howard, R. A. (1966). Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1), 22-26. <https://doi.org/10.1109/TSSC.1966.300074>. El valor de la información es la mejora de utilidad esperada que produce observar una evidencia antes de decidir, y es siempre no negativa. ↵
8. Anthropic. (2026). *Prompt engineering overview*. <https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/overview>. Consultado el 25 de mayo de 2026. ↵
9. OpenAI. (2026). *Structured model outputs*. <https://developers.openai.com/api/docs/guides/structured-outputs>. Consultado el 25 de mayo de 2026. ↵
10. Lewis, P. y otros (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474. <https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>. ↵
11. OpenAI. (2026). *Function calling*. <https://developers.openai.com/api/docs/guides/function-calling>. Consultado el 25 de mayo de 2026. ↵
12. Hu, E. J. y otros (2022). LoRA: Low-Rank Adaptation of Large Language Models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2106.09685>. Hugging Face. (2026). *PEFT: LoRA developer guide*. https://huggingface.co/docs/peft/developer_guides/lora. Consultado el 25 de mayo de 2026. ↵
13. Lewis, P. y otros (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474. <https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>. El trabajo formuló modelos que recuperan documentos y los combinan con generación. ↵
14. Hu, E. J. y otros (2022). LoRA: Low-Rank Adaptation of Large Language Models. *International Conference on Learning Representations*. <https://arxiv.org/abs/2106.09685>. ↵
15. Dettmers, T. y otros (2023). QLoRA: Efficient Finetuning of Quantized LLMs. *Advances in Neural Information Processing Systems 36*. <https://arxiv.org/abs/2305.14314>. ↵

- .6. Triantaphyllou, E. (2000). *Multi-Criteria Decision Making Methods: A Comparative Study*. Springer. <https://doi.org/10.1007/978-1-4757-3157-6>. El Weighted Sum Model es el método multicriterio más conocido; su validez exige criterios comparables y normalizados. ↵
- .7. Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley. Formaliza cómo combinar varios objetivos con funciones de valor y pesos. ↵
- .8. Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley. La frontera eficiente de compromisos recoge las alternativas no dominadas entre las que la elección depende de las preferencias. ↵
- .9. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. Define precisión, exhaustividad (recall) y sus variantes por corte, base de la evaluación de recuperación. ↵
- .10. Es, S., James, J., Espinosa-Anke, L. y Schockaert, S. (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. <https://arxiv.org/abs/2309.15217>. Formaliza métricas como faithfulness y context relevance para evaluar sistemas RAG. ↵
- .11. El intervalo de Wald usa la aproximación normal de la binomial; para n pequeño o $\hat{p}$ cercano a 0 o 1 conviene el intervalo de Wilson. Lawrence D. Brown, T. Tony Cai y Anirban DasGupta. (2001). *Interval Estimation for a Binomial Proportion*. Statistical Science, 16(2), 101-133. <https://doi.org/10.1214/ss/1009213286>. ↵

Capítulo 02: APIs de modelos: mensajes, streaming y salidas estructuradas

El modelo no vive solo en una caja de texto

Cuando usamos un chat en el navegador, parece que el sistema funciona así: escribes una pregunta y aparece una respuesta. Para aprender a construir productos con IA, esa imagen se queda corta. Entre tu aplicación y el modelo hay un contrato: qué modelo llamas, qué mensajes envías, qué herramientas están disponibles, qué formato esperas, si quieres streaming, qué harás si la salida no valida y cómo guardarás la traza.

Este capítulo baja un escalón desde el criterio del [capítulo 01](#). Allí decidimos cuándo tiene sentido usar prompt, schema, RAG, tool o ajuste. Aquí aprendemos a convertir esa decisión en una petición concreta a una API de modelos.

La idea central es sencilla: **una API de modelos no es un buzón de texto; es una frontera entre software probabilístico y software verificable.**

Estado del arte con fecha de corte

Fecha de corte: 10 de junio de 2026.

Fuentes consultadas ese día: documentación oficial de OpenAI sobre Responses API, generación de texto, archivos, visión, salidas estructuradas, function calling, streaming, SDKs y Agents SDK; documentación de Anthropic sobre Messages API, visión, PDF, streaming y structured outputs; documentación de Google sobre Gemini API, documentos, visión, structured outputs, ADK, MCP y A2A; documentación oficial del protocolo A2A; JSON Schema; y la especificación WHATWG de Server-Sent Events.¹

La parte estable es el patrón: enviar una petición con modelo, instrucciones, entrada, herramientas opcionales, formato esperado y opciones de ejecución. La parte cambiante son los nombres exactos de endpoints, SDKs, modelos, campos y límites. Por eso conviene aprender la arquitectura mental antes que memorizar una firma de cliente.

OpenAI documenta la generación de texto y el uso de salidas estructuradas para pedir respuestas que cumplan un schema.²³ También documenta function calling para que el modelo solicite llamadas a funciones definidas por la aplicación.⁴ Anthropic organiza su API alrededor de mensajes y documenta streaming y salidas estructuradas como patrones de construcción.⁵⁶⁷ Google documenta Gemini como una API de generación de contenido con entrada textual y multimodal, además de structured outputs basadas en schema.⁸⁹

La revisión del 10 de junio no cambia la tesis del capítulo, pero sí refuerza una decisión de ingeniería: **no acoples tu dominio al JSON exacto de un proveedor**. OpenAI mantiene como piezas centrales Responses, structured outputs, function calling y streaming; Anthropic conserva Messages API, streaming y patrones de salida estructurada; Google añade además superficies recientes alrededor de Interactions API, ADK y protocolos de agentes.¹⁰¹¹¹² La conclusión práctica es aburrida y muy importante: escribe un adaptador por proveedor, valida la salida en tu código, guarda trazas comparables y deja que tu aplicación hable en objetos propios, no en campos de moda.

Qué no es una API de modelos

Una API de modelos no es “el prompt por HTTP”. Si la tratas así, acabarás pegando instrucciones, datos, historial, formato esperado y lógica de negocio en una sola cadena. Eso funciona en una demo; se vuelve frágil cuando hay usuarios reales.

Tampoco es una promesa de verdad. La API puede devolver texto muy convincente, pero tu aplicación sigue necesitando validación, métricas, trazas y reglas de producto. Si pides JSON y no validas JSON, no tienes contrato: tienes esperanza.

Y no es una interfaz idéntica entre proveedores. Muchos conceptos se parecen, pero los detalles cambian: roles disponibles, nombre del campo de entrada, eventos de streaming, formato de tools, límites, modelos, errores y objetos de respuesta. Por eso el diseño de tu aplicación debería tener una capa propia que traduzca “lo que necesita mi producto” a “lo que espera este proveedor”.

Qué sí es: un contrato entre capas

Una API de modelos es un contrato entre tres mundos. El primero es tu aplicación: usuarios, permisos, pantallas, flujos y datos. El segundo es el proveedor del modelo: endpoint, modelo, tokens, eventos, herramientas y respuesta. El tercero es tu código de integración: validadores, retries, logs, evals y transformación a objetos de dominio.

Formalmente, una petición no es una cadena: es un valor de un **tipo producto**, lo que en programación llamamos un registro o `struct`. Tiene un campo por cada decisión, y cada campo se puede tipar y validar por separado, igual que un modelo de Pydantic o una interfaz de TypeScript. En notación de tipos:

$$R = M \times I \times C \times F \times T \times S$$

Una petición concreta $r \in R$ fija un valor en cada campo:

CAMPO	SIGNIFICADO	EJEMPLO
<i>M</i>	Modelo elegido.	Un modelo rápido para clasificación.
<i>I</i>	Instrucciones y mensajes.	Rol del sistema, contexto y pregunta del usuario.
<i>C</i>	Contexto externo.	Fragmentos RAG, datos de producto o historial relevante.
<i>F</i>	Formato de salida.	JSON con <code>categoria</code> , <code>prioridad</code> y <code>siguiente_paso</code> .
<i>T</i>	Tools disponibles.	<code>consultar_expediente(id_alumno)</code> .
<i>S</i>	Opciones de servicio.	Streaming, límite de salida, metadata o timeout.

Verlo como registro tipado, y no como una cadena opaca, es lo que permite localizar el fallo en un campo concreto. Si falla *F*, quizá necesitas schema. Si falla *C*, quizá necesitas retrieval. Si falta *T*, el modelo no debería inventar estado externo. Si falla *S*, puede que el problema sea latencia, no inteligencia.

Qué APIs se usan hoy y quién las usa

A 10 de junio de 2026, cuando una empresa dice “vamos a llamar a un LLM desde nuestra app”, normalmente está hablando de una de estas familias. No son las únicas, pero sí sirven para entender el mercado: APIs directas de laboratorios, APIs cloud que empaquetan varios modelos, gateways que unifican proveedores y runtimes locales que imitan una interfaz remota.

OpenAI organiza gran parte de su construcción moderna alrededor de la Responses API: una llamada que puede recibir entrada multimodal, instrucciones, tools, formato de salida, streaming y opciones de ejecución.¹³ La usan equipos que quieren construir asistentes, clasificadores, flujos con herramientas, extracción estructurada, análisis de documentos, experiencias con visión o productos donde una respuesta textual debe convertirse en objeto de software.

Anthropic expone Claude principalmente mediante Messages API para conversaciones y turnos sin estado: envías una lista estructurada de mensajes y el modelo genera el siguiente mensaje.¹⁴ Es habitual en productos que necesitan lectura y escritura largas, análisis documental, asistentes internos, ayuda a programación, tutoría o flujos donde interesa controlar muy bien el historial enviado.

Google ofrece la Gemini API alrededor de `generateContent` , con entradas que pueden combinar texto, imágenes, vídeo y audio, además de configuración de generación y salidas estructuradas.¹⁵ Encaja especialmente en productos multimodales, prototipos integrados con

el ecosistema Google, procesamiento de documentos y aplicaciones que quieren aprovechar modelos con ventanas largas o capacidades visuales.

Además existen plataformas como Amazon Bedrock, Vertex AI, Azure AI Foundry, Mistral, Cohere, Groq, Together, Fireworks, OpenRouter o Vercel AI SDK. La lección no es aprenderlas todas de memoria. La lección es que casi todas terminan pidiendo lo mismo con nombres distintos: modelo, entrada, instrucciones, parámetros de generación, herramientas, schema, streaming, límites y metadatos.

Los parámetros: no memorizar nombres, entender familias

Cuando un alumno ve por primera vez una referencia de API, se pierde porque parece una lista interminable de campos. La forma útil de estudiarla es agrupar parámetros por intención, es decir, agrupar los campos del registro *R* en seis familias: **qué modelo** usamos, **qué entra**, **cuánto y cómo responde**, **qué herramientas puede pedir**, **qué forma debe tener la salida** y cómo se **opera** el sistema. Memorizar nombres concretos es inútil; reconocer a qué familia pertenece cada campo sirve en cualquier proveedor.

La tabla siguiente no pretende congelar una API que cambia. Sirve para reconocer esas equivalencias entre familias:

FAMILIA	OPENAI RESPONSES API	ANTHROPIC MESSAGES API	GEMINI API
Modelo	<code>model</code>	<code>model</code>	modelo en <code>models.generateContent</code> o ruta REST.
Entrada	<code>input</code> con mensajes y bloques de contenido.	<code>messages</code> con contenido por turnos.	<code>contents</code> con <code>parts</code> .
Instrucciones estables	<code>instructions</code> o mensajes equivalentes según SDK.	<code>system</code> como campo superior.	<code>systemInstruction</code> o configuración equivalente.
Límite de salida	<code>max_output_tokens</code> según modelo y endpoint.	<code>max_tokens</code> .	<code>maxOutputTokens</code> dentro de configuración.
Aleatoriedad	<code>temperature</code> , <code>top_p</code> cuando el modelo lo permita.	<code>temperature</code> , <code>top_p</code> , <code>top_k</code> según modelo.	<code>temperature</code> , <code>topP</code> , <code>topK</code> .
Parada	secuencias de parada si están disponibles.	<code>stop_sequences</code> .	<code>stopSequences</code> .
Salida estructurada	formato de texto/schema o Structured Outputs.	<code>output_config.format</code> o tool estricta, según caso.	<code>responseMimeType</code> y <code>responseJsonSchema</code> .
Tools	<code>tools</code> y <code>tool_choice</code> .	<code>tools</code> y <code>tool_choice</code> .	<code>tools</code> y function calling.
Streaming	<code>stream</code> y eventos.	<code>stream</code> con eventos SSE.	métodos de streaming del SDK o REST.
Metadatos y operación	<code>metadata</code> , trazas, <code>user</code> o campos de servicio cuando existan.	<code>metadata</code> , uso de tokens y estado de parada.	<code>usageMetadata</code> , configuración y metadatos del entorno.

El parámetro más peligroso no es siempre el más técnico. Muchas integraciones fallan por no fijar bien `max_tokens` o `max_output_tokens`, por mezclar instrucciones con datos del usuario, por no versionar el schema o por asumir que `temperature=0` convierte una salida probabilística en una función matemática.

Entradas y salidas: qué ocurre en cada caso

Una API de modelos no devuelve siempre “texto”. Puede devolver texto, JSON, una petición de tool, eventos parciales o una combinación de bloques. Por eso conviene diseñar el flujo antes de escribir el cliente.

CASO	ENTRA	SALE	QUÉ DEBE HACER TU APLICACIÓN
Texto a texto	Pregunta, instrucciones y contexto breve.	Respuesta natural.	Mostrar, registrar y quizá evaluar calidad.
Texto a JSON	Texto y schema.	Objeto estructurado.	Parsear, validar y transformar a objeto de dominio.
Documento a resumen	Archivo, páginas o fragmentos.	Resumen, citas o extracción.	Conservar referencia a página, versión y documento original.
Imagen a explicación	Imagen más pregunta.	Descripción, clasificación o lectura visual.	Revisar límites visuales y pedir evidencia cuando importe.
Texto a tool call	Petición y definición de función.	Nombre de tool y argumentos.	Validar argumentos, comprobar permisos y ejecutar código.
Tool result a respuesta	Resultado externo.	Explicación final.	Separar dato consultado de redacción generada.
Streaming	Petición normal con <code>stream</code> .	Eventos parciales.	Acumular, cancelar, manejar errores y cerrar estado.

La salida estructurada y las tool calls se parecen porque usan schemas, pero no cumplen la misma misión. Una salida estructurada es el resultado final con forma de dato. Una tool call es una solicitud intermedia para que el sistema consulte, calcule o actúe. Si confundes ambas cosas, tu app termina ejecutando texto como si fuera una decisión cerrada.

Documentos e imágenes: multimodal no significa comprensión automática

Enviar una imagen o un PDF a un modelo multimodal no equivale a “el modelo lo sabe todo sobre ese archivo”. Equivale a darle una entrada rica que el modelo puede procesar dentro de sus límites. OpenAI documenta bloques como `input_file` para archivos y `input_image` para imágenes dentro de la entrada.¹⁶¹⁷ Anthropic documenta visión y soporte de PDF para Claude.¹⁸¹⁹ Gemini permite pasar imágenes como datos inline o mediante Files API, y documenta límites específicos para PDF.²⁰²¹

Para una sola imagen, suele bastar con enviar la imagen y una pregunta clara: “lee esta factura y extrae fecha, proveedor e importe”. Para muchas imágenes o archivos repetidos, conviene subirlos una vez y referenciarlos. Para un PDF largo, hay que decidir si se manda

entero, si se divide por páginas, si se indexa en un sistema RAG o si se usa una combinación: retrieval para localizar fragmentos y modelo multimodal para interpretar tablas, figuras o páginas concretas.

Hay cuatro cosas que no deberíamos olvidar:

CUIDAD O	POR QUÉ IMPORTA	BUENA PRÁCTICA
Tamaño y coste	Imágenes y documentos consumen contexto y pueden aumentar latencia.	Medir tokens, páginas, resolución y tiempo de respuesta.
Referencias	Una respuesta sin página o fragmento es difícil de revisar.	Pedir <code>pagina</code> , <code>fragmento</code> o <code>evidencia</code> en el schema.
Privacidad	Los documentos pueden contener datos personales o internos.	Minimizar, redactar cuando sea posible y revisar política del proveedor.
Lectura visual	Un modelo puede interpretar mal tablas, sellos o capturas pequeñas.	Comprobar con OCR, reglas o revisión humana cuando importe.

La regla práctica: si el documento es fuente de verdad, no lo trates como “contexto decorativo”. Guárdalo con identificador, versión, fecha de carga y forma de recuperación. Si mañana alguien pregunta por qué la app contestó eso, debes poder reconstruir qué archivo vio, qué páginas entraron y qué schema validó la salida.

Mensajes: separar instrucciones, contexto y petición

La mayoría de APIs modernas no reciben solo una cadena. Reciben mensajes o una estructura equivalente. El objetivo no es teatralizar una conversación, sino separar responsabilidades: qué reglas gobiernan la tarea, qué dijo la persona usuaria, qué contestó antes el modelo y qué resultados devolvieron herramientas.

PIEZA	QUÉ REPRESENTA	RIESGO SI SE MEZCLA
Instrucciones de sistema o desarrollador	Comportamiento estable que quieres mantener.	El usuario puede pisar reglas importantes con texto accidental.
Mensaje de usuario	La petición concreta de esta interacción.	El sistema no distingue objetivo de contexto.
Contexto recuperado	Evidencia externa añadida por la aplicación.	El modelo no sabe qué parte citar o priorizar.
Mensaje del asistente	Respuesta anterior o salida actual.	Se pierde trazabilidad en conversaciones largas.
Resultado de tool	Dato externo obtenido por código.	Se confunde texto generado con dato comprobado.

Un patrón robusto consiste en construir la petición desde piezas separadas y solo al final traducirlas al formato del proveedor. Así puedes cambiar de modelo sin reescribir la lógica de negocio.

Para entenderlo: si una app universitaria pregunta “¿puede Ana matricularse de Sistemas Inteligentes?”, no deberías mandar solo esa frase. Deberías separar la política académica vigente, el identificador de Ana, las reglas de formato de salida y, si hace falta, la tool que consulta expediente.

Salidas estructuradas: cuando el texto debe convertirse en dato

Pedir “responde en JSON” es una intención. Usar un schema es un contrato. JSON Schema define vocabulario para describir tipos, propiedades, campos requeridos y reglas de validación sobre documentos JSON.²² Las APIs de modelos aprovechan esa idea para reducir la distancia entre respuesta natural y objeto que tu software puede consumir.

La métrica mínima de una salida estructurada es la tasa de conformidad:

$$\text{validez} = \frac{N_{\text{válidas}}}{N_{\text{total}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$N_{\text{válidas}}$	Respuestas que cumplen schema.	97 de 100 respuestas.
N_{total}	Respuestas evaluadas.	100 casos de prueba.
validez	Proporción de salidas estructuralmente correctas.	0,97.

Como toda proporción medida sobre una muestra, la validez tiene incertidumbre: con $N_{\text{total}} = 100$ y $\text{validez} = 0,97$, el intervalo de confianza al 95 % es $0,97 \pm 1,96 \cdot 0,03/100 \approx \pm 0,034$. La lectura práctica es la contraria a la de una eval de calidad: en validez de schema, un solo fallo de cada 100 no es ruido, es un patrón roto que tu validador debe rechazar siempre, no un porcentaje aceptable.

Pero cuidado: una respuesta puede cumplir schema y seguir siendo mala. Si el schema pide `prioridad`, el valor puede ser válido como texto y estar mal como decisión. Por eso necesitamos dos validaciones:

VALIDACIÓN	PREGUNTA	EJEMPLO
Estructural	¿Cumple tipos, campos y restricciones?	<code>prioridad</code> existe y vale <code>alta</code> , <code>media</code> o <code>baja</code> .
Semántica	¿El contenido es correcto para el caso?	El mensaje realmente requiere prioridad alta.

Un detalle de ingeniería que la práctica de este capítulo aprovecha: declarar `additionalProperties: false` junto con el modo estricto convierte el schema en un contrato de **mundo cerrado**, en el que el modelo no puede añadir campos que no estén declarados. Sin eso, una respuesta puede validar y aun así arrastrar claves inventadas que tu backend no espera. Un buen contrato no solo dice qué campos deben estar; dice también qué campos no pueden estar.

La salida estructurada arregla el contrato con el software. No reemplaza la evaluación del criterio.

Streaming: que la respuesta llegue por partes

Streaming significa que la aplicación no espera a tener toda la respuesta para empezar a recibir fragmentos. En la web moderna suele implementarse con eventos o flujos parecidos a Server-Sent Events, donde el servidor envía datos progresivamente al cliente.²³ OpenAI y

Anthropic documentan streaming para respuestas de modelos.²⁴²⁵

La razón no es solo estética. El streaming cambia la experiencia percibida.

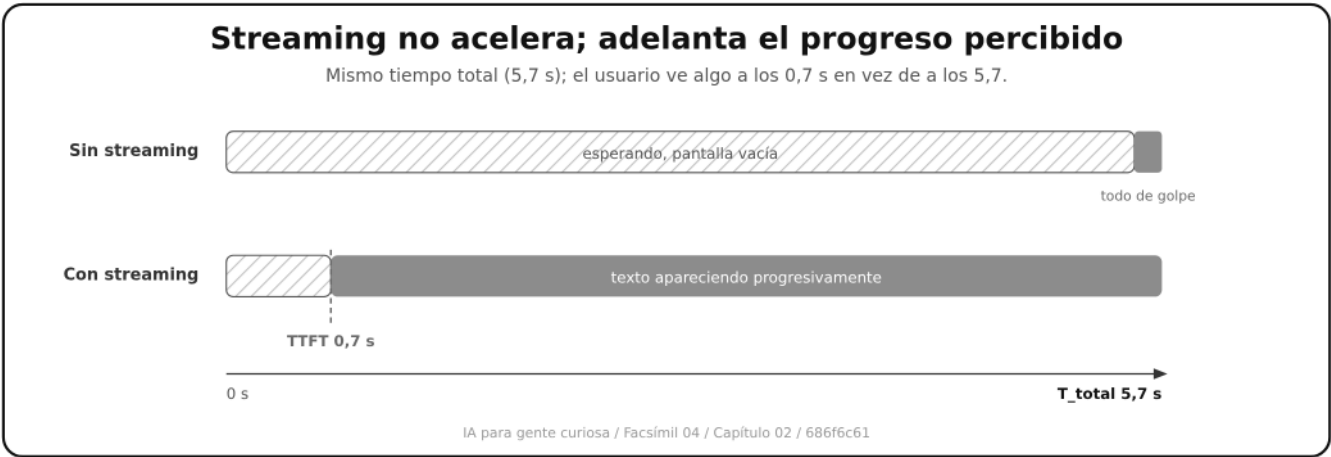
La clave está en separar el tiempo total de la latencia percibida. El tiempo total hasta terminar una respuesta se descompone, según el modelo de dos fases (*prefill* y *decode*) de la inferencia autoregresiva, en el tiempo hasta el primer token y el tiempo de generar el resto:²⁶

$$T_{\text{total}} = \text{TTFT} + \frac{n_{\text{salida}}}{v}$$

Donde v es el ritmo de generación en tokens por segundo, de modo que n_{salida}/v es la fase de decode y equivale a $(n_{\text{salida}}) \cdot \text{TPOT}$, el tiempo por token de salida.

SÍMBOLO	SIGNIFICADO	EJEMPLO
TTFT	Tiempo hasta el primer token (<i>time to first token</i>).	0,7 s.
n_{salida}	Tokens que hay que generar.	300 tokens.
v	Velocidad de generación (tokens por segundo).	60 tokens/s.
T_{total}	Tiempo total de la respuesta.	$0,7 + 300/60 = 5,7$ s.

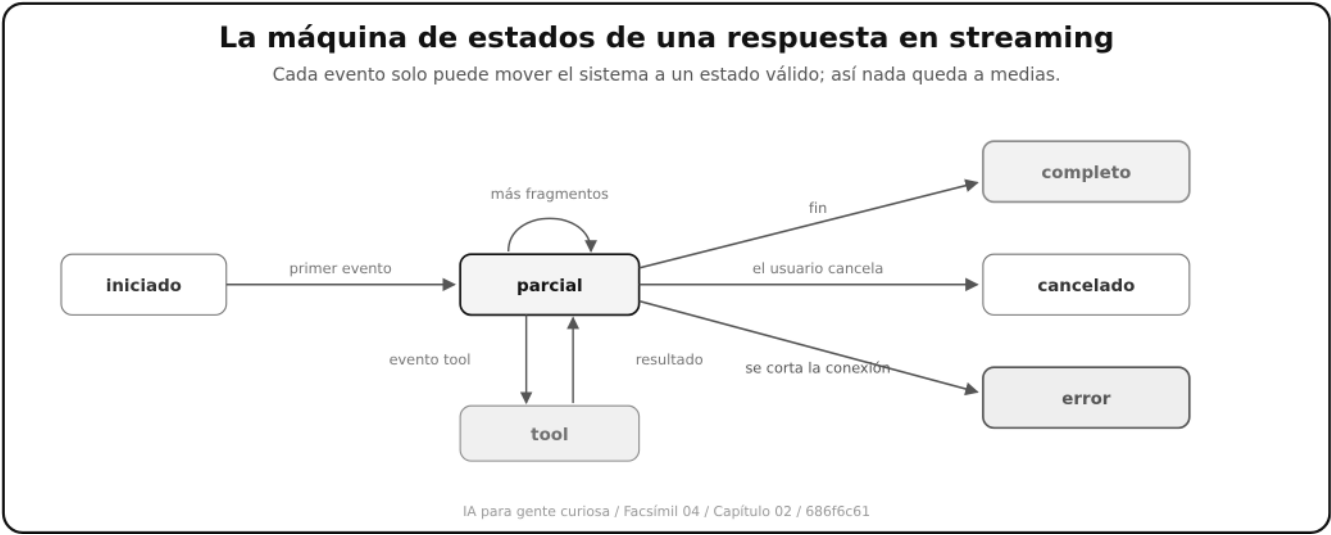
Streaming no cambia T_{total} : el modelo tarda lo mismo en terminar. Lo que cambia es **cuándo el usuario percibe progreso**. Sin streaming, no ve nada hasta los 5,7 segundos; con streaming, ve el primer fragmento a los 0,7 (la TTFT) y el resto va apareciendo. La latencia percibida cae de T_{total} a TTFT, aunque el coste total no se mueva.



Streaming no hace que el modelo “piense mejor”. Hace que el producto pueda mostrar progreso, cancelar, actualizar UI y registrar eventos. También complica: debes ensamblar fragmentos, manejar cortes, distinguir eventos de texto y eventos de tool, y decidir cuándo

una salida estructurada está lista para validarse.

Por eso una respuesta en streaming se modela mejor como una **máquina de estados finita**: un conjunto de estados y las transiciones permitidas entre ellos. Tratarla así evita el error más común, dejar la interfaz o la traza a medias cuando la conexión se corta, porque cada evento que llega solo puede mover el sistema a un estado válido.



Tool calls: cuando responder no basta

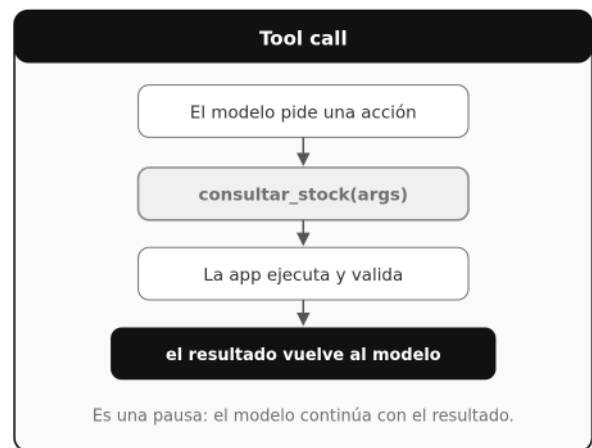
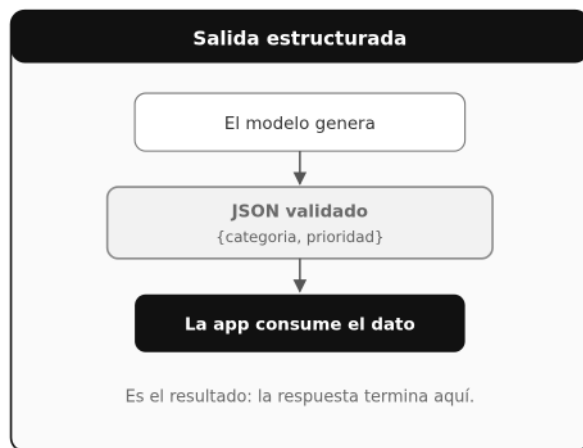
Una salida estructurada devuelve datos. Una tool call pide que tu aplicación ejecute algo. Esa diferencia parece pequeña y es enorme.

NECESIDAD	SALIDA ESTRUCTURADA	TOOL CALL
Clasificar un mensaje	Devuelve <code>{categoria, prioridad}</code> .	Normalmente no hace falta.
Consultar stock	Puede devolver intención de consulta.	Llama <code>consultar_stock(producto, talla)</code> .
Calcular una cuota	Puede proponer fórmula.	Llama <code>calcular_cuota(importe, plazo)</code> .
Abrir un ticket	Puede redactar el contenido.	Llama <code>crear_ticket(...)</code> si el usuario confirma.

La regla práctica: **si el dato existe fuera del modelo, no lo conviertas en adivinanza**. Define una tool pequeña, valida sus argumentos, ejecuta el código y devuelve el resultado como contexto para que el modelo lo explique.

Salida estructurada frente a tool call

Las dos usan schema, pero una termina la respuesta y la otra la interrumpe para pedir algo.



IA para gente curiosa / Facsímil 04 / Capítulo 02 / 686f6c61

Para entenderlo antes de tocar código

Pensemos en cuatro productos que parecen parecidos porque todos “usan IA”, pero que piden contratos distintos.

PRODUCTO	QUÉ ENVÍA A LA API	QUÉ ESPERA RECIBIR	PIEZA CRÍTICA
Clasificador de correos internos	Texto del correo e instrucciones.	JSON con cola, prioridad y motivo breve.	Schema y validador.
Tutor universitario	Pregunta, nivel del curso y rúbrica.	Explicación paso a paso.	Mensajes bien separados.
Asistente de matrícula	Pregunta y contexto normativo recuperado.	Respuesta con cita y quizá tool de expediente.	RAG, tool y trazabilidad.
Redactor con respuesta larga	Brief, tono y ejemplos.	Texto progresivo en pantalla.	Streaming y cancelación.

El mismo modelo puede participar en los cuatro, pero la API no se usa igual. En uno importa más el schema; en otro, el streaming; en otro, tools; en otro, trazabilidad del contexto.

Buenas prácticas de integración

Una integración madura no empieza llamando al SDK desde cualquier pantalla. Empieza con un contrato propio de la aplicación. Ese contrato dice: “para clasificar una solicitud necesito estos campos, esta política, este schema, estas tools permitidas y esta forma de guardar trazas”. Después un adaptador traduce ese contrato al proveedor elegido.

El adaptador evita que el resto del producto sepa si por debajo hay OpenAI, Claude, Gemini, un modelo local o un gateway. También te obliga a decidir lo importante en un sitio: versionar prompts, schemas y modelos; convertir errores de proveedor en errores propios; medir coste; registrar identificadores; y probar la misma tarea con casos de evaluación.

PRÁCTICA	QUÉ RESUELVE	CÓMO SE VE EN CÓDIGO O PRODUCTO
Adaptador por proveedor	Evita acoplar pantallas a nombres de campos externos.	<code>crearRespuestaTutor(...)</code> traduce a OpenAI, Claude o Gemini.
Schema versionado	Permite cambiar formato sin romper consumidores.	<code>respuesta_matricula.v2.json</code> .
Validador propio	No delega toda la corrección al proveedor.	Pydantic, Zod, JSON Schema o validación del backend.
Trazas mínimas	Permite reproducir fallos sin guardar más de lo necesario.	<code>trace_id</code> , modelo, versión de prompt, schema y resumen de entrada.
Timeouts y reintentos	Evita dejar al usuario esperando sin cierre.	Reintento solo en operaciones idempotentes.
Tools pequeñas	Reduce ambigüedad y facilita permisos.	<code>consultar_expediente(id)</code> en vez de <code>hacer_cosas(datos)</code> .
Evals antes de publicar	Mide si la integración mejora o empeora.	Conjunto de casos fijos con salida esperada.
Streaming con máquina de estados	Evita UI a medias.	<code>iniciado</code> , <code>parcial</code> , <code>tool</code> , <code>completo</code> , <code>cancelado</code> , <code>error</code> .

Los reintentos tienen una lógica que conviene cuantificar. Si una llamada falla de forma transitoria (un `429`, un `503`, un `timeout`) con probabilidad $1 - p$ en cada intento, y los fallos son independientes, la probabilidad de éxito tras k intentos es:

$$p_{\text{eff}} = 1 - (1 - p)^k$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p	Probabilidad de éxito de un intento.	0,9.
k	Número de intentos permitidos.	3.
p_{eff}	Probabilidad de éxito tras los reintentos.	$1 - (1 - 0,9)^3 = 0,999$.

Con $p = 0,9$, pasar de un intento a tres sube la fiabilidad del 90 % al 99,9 %. La trampa es la palabra **independientes**: la fórmula solo vale para fallos transitorios. Si la petición está mal formada o el schema es inválido, fallará igual las tres veces, y reintentar solo multiplica el coste. Por eso los reintentos van sobre operaciones idempotentes y sobre códigos de error transitorios (429 , 500 , 502 , 503 , 504), nunca sobre un 400 de validación.

La integración más limpia tiene forma de **pipeline cerrado**: la petición sale del producto, atraviesa cada capa y vuelve al producto ya convertida en algo validado, no en texto suelto.

```
producto -> contrato propio -> adaptador -> proveedor -> validador -> producto
```

PIEZA	PREGUNTA QUE RESPONDE
Producto	¿Qué necesita conseguir la persona usuaria?
Contrato propio	¿Qué datos, formato, tools y reglas exige nuestro flujo?
Adaptador	¿Cómo se expresa eso en la API concreta?
Proveedor	¿Qué modelo genera, pide tool o devuelve eventos?
Validador	¿La salida cumple estructura y criterio mínimo?
Producto	¿Mostramos, pedimos confirmación, guardamos o repetimos?

Cómo sería una API perfecta para integrar

Una API perfecta no sería “la que siempre acierta”. Eso no existe. Sería la que hace fácil construir software fiable alrededor de una capacidad probabilística. Si diseñáramos una interfaz ideal para una app profesional, tendría estas propiedades:

RASGO	POR QUÉ IMPORTA
Entrada multimodal tipada	Texto, imágenes y documentos no llegan como una cadena opaca.
Instrucciones separadas	Las reglas estables no se mezclan con lo que escribe la persona usuaria.
Salida schema-first	El contrato de datos se declara antes de generar.
Tools tipadas y pequeñas	La API distingue pedir una acción de ejecutar una acción.
Eventos normalizados	Streaming, tool calls y errores siguen una secuencia predecible.
Uso y coste visibles	La respuesta trae tokens, latencia y modelo usado.
Versionado explícito	Prompt, schema, modelo y toolset se pueden congelar y comparar.
Errores tipados	La app sabe si hubo límite, timeout, validación fallida o contenido incompleto.
Idempotencia	Reintentar no duplica acciones sensibles ni crea registros repetidos.
Privacidad configurable	Puedes decidir qué se guarda, qué se omite y durante cuánto tiempo.
Evals integradas	El mismo contrato puede probarse con casos antes de publicarse.

En pseudocódigo, una llamada ideal se parecería menos a “envía este texto” y más a esto:

```
respuesta = modelo.generar({
  tarea: "clasificar_solicitud_matricula",
  contrato: "solicitud_matricula.v2",
  entrada: {texto, documentos, usuario_contexto},
  salida: schema_respuesta,
  tools: [consultar_expediente],
  ejecucion: {stream: true, timeout_ms: 12000, trace_id},
  politica: {confirmar_antes_de_crear_ticket: true}
})
```

La clave no es que todos los proveedores adopten exactamente ese formato. La clave es que tu aplicación sí tenga esa claridad interna. Si tu dominio está bien modelado, cambiar de proveedor es una migración. Si tu dominio vive pegado al prompt, cambiar de proveedor es cirugía.

El contrato de OpenAI Responses, campo a campo

Antes de armar la práctica, conviene ver una petición real a la Responses API de OpenAI con sus campos principales, agrupados por las seis familias del registro *R*. Tómallo como un ejemplo de contrato: no para memorizarlo, sino para saber qué decide cada pieza. La referencia exacta y siempre actualizada vive en la documentación de OpenAI; aquí mostramos los campos estables y explicamos cada trozo.


```

peticion = {
  # Familia 1 (M): qué modelo
  "model": "modelo-vigente",

  # Familia 2 (I, C): qué entra
  "instructions": "Reglas estables del sistema, separadas del usuario.",
  "input": [
    {
      "role": "user",
      "content": [
        {"type": "input_text", "text": "Texto de la persona usuaria."},
        {"type": "input_file", "file_id": "file_normativa_2026"},
        {"type": "input_image", "image_url": "https://example.edu/captura.png"},
      ],
    },
  ],
  "previous_response_id": None, # encadenar con una respuesta anterior, si la hubo

  # Familia 3 (g): cuánto y cómo responde
  "max_output_tokens": 900,
  "temperature": 0.2,
  "top_p": 0.9,
  "reasoning": {"effort": "medium"}, # solo en modelos de razonamiento
  "truncation": "auto", # qué hacer si el contexto se desborda

  # Familia 4 (T): qué herramientas puede pedir
  "tools": [TOOL_CONSULTAR_EXPEDIENTE],
  "tool_choice": "auto", # auto | none | required | una tool concreta
  "parallel_tool_calls": False,

  # Familia 5 (F): qué forma tiene la salida
  "text": {
    "format": {
      "type": "json_schema",
      "name": "clasificacion",
      "strict": True,
      "schema": RESPUESTA_SCHEMA,
    },
  },

  # Familia 6 (S): operación
  "stream": True,
  "store": False, # guardar o no la respuesta en el proveedor
  "include": [], # datos extra a incluir en la respuesta
  "metadata": {"trace_id": "trc_matricula_00042"},
}

```

Ahora cada trozo, por familia:

Qué modelo. `model` fija el modelo y su versión. No uses un alias que pueda cambiar bajo tus pies en producción: usa la versión exacta y guárdala en la traza.

Qué entra. `instructions` lleva las reglas estables del sistema, separadas del `input`, donde va la petición de la persona usuaria. El `input` no es una cadena: es una lista de mensajes con bloques de contenido tipados, `input_text` para texto, `input_file` para documentos e `input_image` para imágenes, que es como entra lo multimodal. `previous_response_id` permite encadenar con una respuesta anterior reutilizando estado del servidor, en vez de reenviar todo el historial.

Cuánto y cómo responde. `max_output_tokens` acota la longitud de salida (y el coste). `temperature` y `top_p` controlan la aleatoriedad; para una tarea estructurada se bajan. `reasoning` con su `effort` ajusta cuánto piensa un modelo de razonamiento antes de responder. `truncation` decide qué pasa si la entrada se pasa de contexto: recortar de forma automática o fallar.

Qué herramientas. `tools` declara las funciones tipadas que el modelo puede pedir; `tool_choice` decide si puede elegir (`auto`), no debe usar ninguna (`none`), está obligado (`required`) o debe usar una concreta; `parallel_tool_calls` permite o no varias llamadas a la vez, que conviene desactivar cuando el orden importa.

Qué forma tiene la salida. El bloque `text.format` con `type: "json_schema"` es lo que convierte el texto en dato: `name` identifica el schema, `strict: True` activa el modo estricto y `schema` declara los campos. Combinado con `additionalProperties: false`, fija un contrato de mundo cerrado.

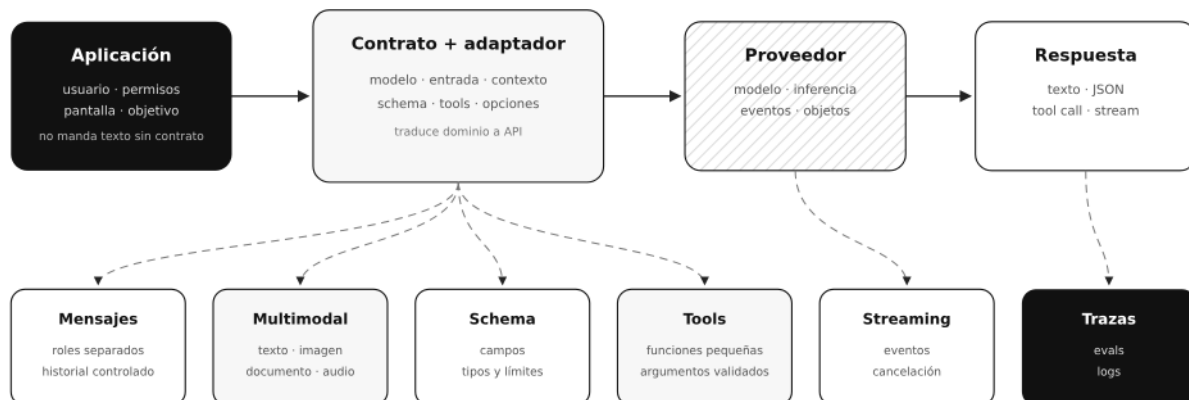
Operación. `stream` activa la entrega progresiva; `store` decide si el proveedor guarda la respuesta; `include` pide datos extra en la respuesta; `metadata` lleva tu `trace_id` y versiones para poder reproducir el caso. Y un matiz importante: `timeout`, la política de reintentos y la `idempotency_key` no viajan en este JSON, son parte de tu cliente HTTP o SDK, pero forman parte igual del contrato de operación.

Mapa visual de una petición robusta

El diagrama resume la idea práctica del capítulo: la aplicación no debería hablar con el modelo como quien manda una frase suelta, sino como quien prepara un contrato que luego valida.

Una API de modelos es un contrato

Entrada, formato, tools, streaming y validación viajan separados.



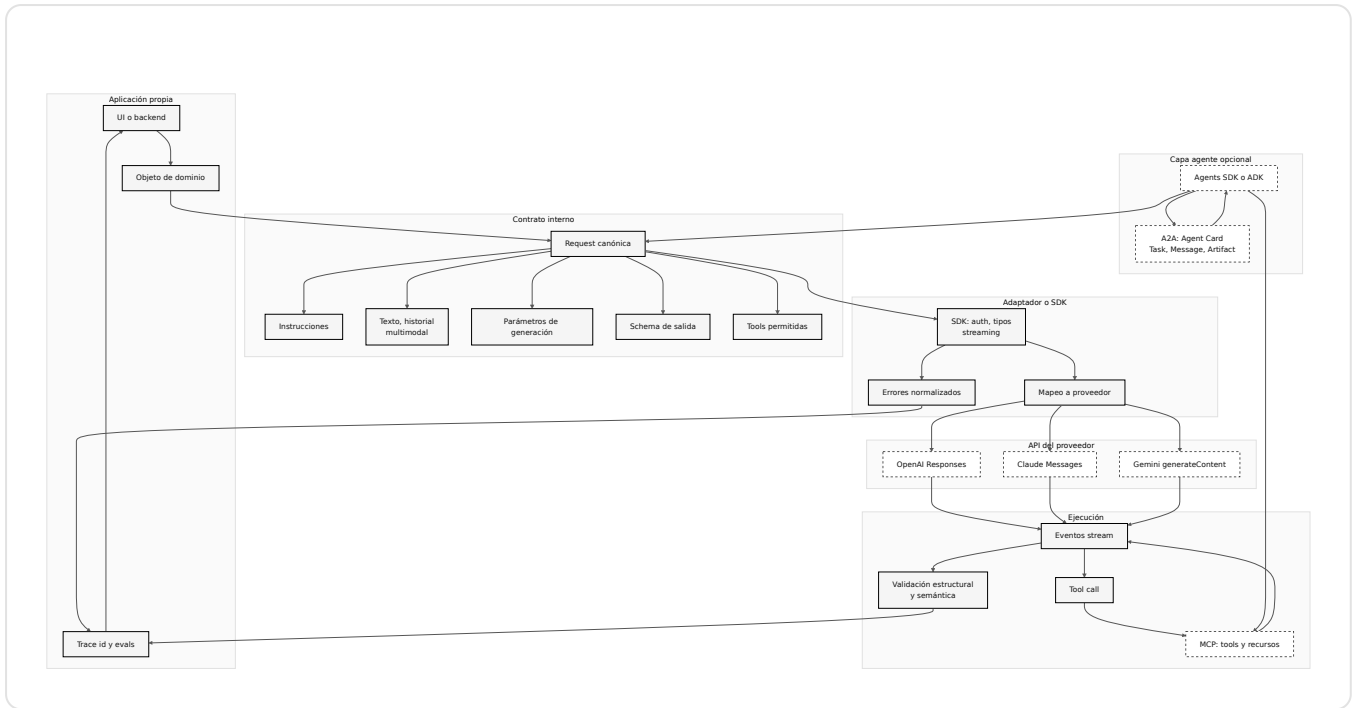
Regla final

El modelo genera; tu aplicación valida, ejecuta, registra y decide qué hacer después.

IA para gente curiosa / Facsímil 04 / Capítulo 02 / 686f6c61

Mapa Mermaid: todo lo que viaja en una API

El SVG anterior da la intuición editorial. Ahora conviene ver la llamada como arquitectura técnica: qué construye tu aplicación, qué transforma el SDK o adaptador, qué recibe el proveedor y qué vuelve a tu sistema.



SDKs, ADK, MCP y A2A: cada cosa en su capa

Aquí suele nacer mucha confusión porque todo parece “la API”. No lo es. Una API es el contrato de red y datos. Un SDK es una biblioteca en tu lenguaje que envuelve esa API. Un framework de agentes es una capa que orquesta pasos, estado, tools y decisiones. Y un protocolo de interoperabilidad define cómo se comunican sistemas que quizá ni comparten proveedor ni framework.

OpenAI documenta SDKs oficiales para lenguajes como JavaScript/TypeScript y Python, pensados para llamar a la API desde código de aplicación.²⁷ El SDK no cambia la semántica: si el endpoint espera `input`, `tools`, `stream` o un schema, el SDK lo expresa con tipos, métodos, helpers de streaming, subida de archivos y manejo de errores. Es cómodo, pero no sustituye al diseño del contrato.

OpenAI también documenta Agents SDK para casos donde tu servidor posee la orquestación, la ejecución de tools, el estado y las aprobaciones del flujo.²⁸ Google ADK va en esa misma familia de herramientas de construcción de agentes: su documentación organiza piezas como agentes, equipos de agentes, workflows, ejecución, observabilidad, evaluación, tools, sesiones, memoria, artefactos, MCP y A2A.²⁹ La propia documentación de ADK incluye guías para exponer agentes a otros sistemas y consumir agentes remotos mediante A2A.³⁰

MCP y A2A resuelven problemas distintos. ADK describe MCP como un estándar para que LLMs y agentes se comuniquen con aplicaciones externas, fuentes de datos y herramientas mediante recursos, prompts y tools.³¹ A2A, en cambio, es para comunicación entre agentes: la documentación oficial lo presenta como un estándar abierto para interoperabilidad entre agentes construidos con distintos frameworks o proveedores.³²

Técnicamente, A2A introduce piezas que no aparecen en una simple llamada a un modelo: AgentCard , Message , Part , Task , TaskStatus , Artifact , métodos para enviar mensajes, consultar tareas, cancelar, suscribirse a eventos y entregar resultados por streaming o notificaciones.³³ La AgentCard dice qué agente hay al otro lado, qué capacidades ofrece y cómo se accede. Un Message lleva partes de contenido; una Task representa trabajo con estado; un Artifact es una salida producida por el agente remoto.

CAPA	OBJETO PRINCIPAL	QUIÉN CONTROLA EL ESTADO	PARA QUÉ SIRVE
API de modelos	Request y response.	Tu aplicación y el proveedor.	Generar, estructurar, llamar tools o recibir eventos.
SDK	Cliente tipado del lenguaje.	Tu aplicación.	Autenticación, tipos, helpers, streaming y errores.
Agents SDK / ADK	Run, sesión, agente, workflow, tool context.	Tu servidor o runtime de agentes.	Orquestar pasos, tools, memoria, evaluación y observabilidad.
MCP	Tool, recurso, prompt.	Host de IA y servidor MCP.	Conectar agentes o apps a herramientas y datos externos.
A2A	Agent Card, Message, Task, Part, Artifact.	Agente cliente y agente remoto.	Delegar tareas entre agentes independientes y recibir progreso o resultados.

Para entenderlo con una situación concreta: si una app de la universidad pregunta a un modelo “clasifica esta solicitud”, quizá basta la API y un SDK. Si necesita consultar expediente, el modelo puede pedir una tool; esa tool puede venir de tu backend o de un servidor MCP. Si además hay un agente remoto especializado en normativa académica, tu agente podría descubrirlo mediante una AgentCard , enviarle un Message , recibir una Task en estado working , escuchar eventos y recoger un Artifact final. Ahí ya no estás “llamando a un modelo”: estás coordinando sistemas.

En el día a día

En un proyecto real, este capítulo aparece cuando alguien dice: “ya tenemos el prompt, vamos a integrarlo”. Ahí empieza el trabajo serio. Hay que decidir qué parte será configuración, qué parte será código, qué parte será schema y qué parte quedará en logs para poder depurar.

Si una respuesta alimenta otra pantalla, no basta con que “se lea bien”. Debe llegar como objeto fiable. Si una respuesta se muestra mientras se genera, debes pensar en streaming y cancelación. Si el modelo pide una tool, debes decidir quién ejecuta, con qué permisos, cómo

se registra y qué ocurre si faltan argumentos.

La integración buena suele tener una capa intermedia: tu aplicación habla en términos de dominio, y esa capa traduce a la API concreta. Así evitas que cada pantalla dependa de detalles de un proveedor.

Por qué debería importarte

Porque una mala integración convierte una capacidad potente en un sistema difícil de mantener. Si guardas texto sin estructura, mañana no podrás medir. Si no validas schema, el backend se rompe tarde. Si no separas mensajes, no sabrás qué instrucción produjo qué comportamiento. Si no registras eventos de streaming y tools, no podrás explicar por qué una respuesta salió como salió.

La buena noticia: una API bien tratada como contrato permite cambiar modelos, añadir RAG, introducir tools y medir calidad sin rehacer todo el producto.

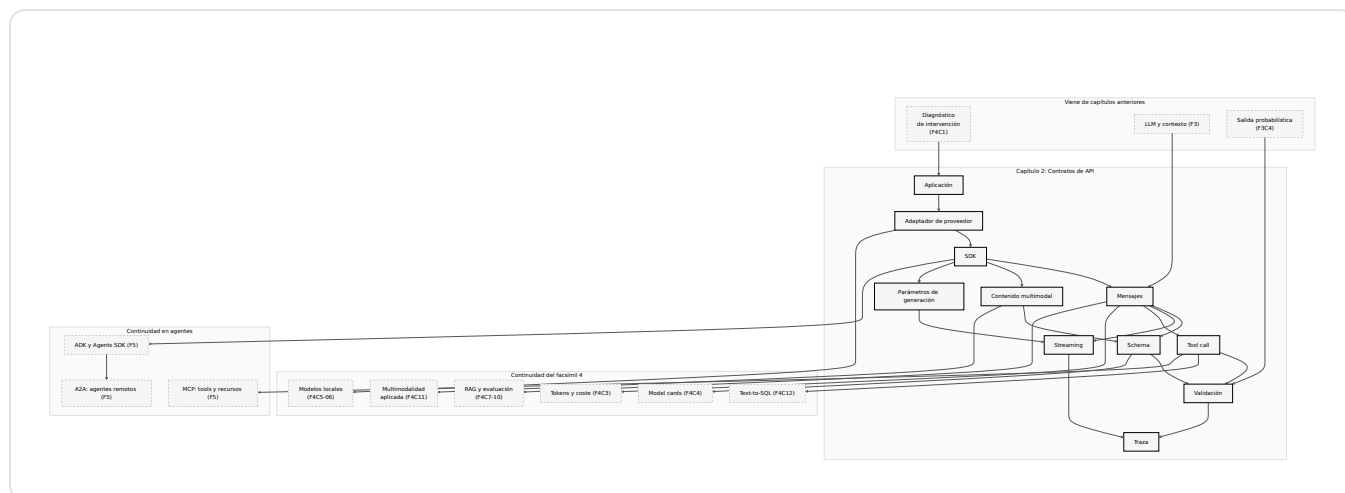
Dónde solía tropezar yo

Estos tropiezos aparecen cuando uno pasa de probar prompts a construir una aplicación que tiene que vivir.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Meter todo en un único prompt	Instrucción, contexto, formato y datos se vuelven inseparables.	Construir mensajes y contexto por capas.
Validar solo que haya JSON	Un objeto puede estar bien formado y contener una decisión mala.	Separar validación estructural y semántica.
Confundir tool call con ejecución	Que el modelo pida una función no significa que deba ejecutarse sin más.	Validar argumentos y aplicar reglas de negocio antes de ejecutar.
Usar streaming sin estado claro	Si se corta el flujo, puedes dejar la UI o la traza a medias.	Diseñar estados: iniciado, parcial, completo, cancelado y error.
Mandar documentos sin referencia	La respuesta queda desligada del archivo, página o versión que la produjo.	Guardar identificador, páginas usadas y schema de extracción.
Acoplarse al proveedor demasiado pronto	Cada pantalla acaba hablando el dialecto de una API concreta.	Usar un contrato propio y adaptadores por proveedor.
Confundir SDK con arquitectura	El SDK facilita la llamada, pero no decide schemas, permisos, trazas ni evaluación.	Diseñar primero contrato y flujo; elegir SDK después.
Mezclar MCP y A2A	MCP conecta tools y recursos; A2A conecta agentes completos con tareas y artefactos.	Dibujar qué sistema habla con qué sistema antes de integrar.
No guardar trazas mínimas	Sin request, respuesta, schema y versión de modelo no puedes reproducir fallos.	Registrar lo necesario para depurar sin guardar datos innecesarios.

Cómo encaja todo

Este mapa sitúa el capítulo dentro del facsímil. El capítulo anterior decide qué intervención toca; este convierte esa decisión en contrato de API.



Vocabulario aprendido

Estos términos nos permiten hablar de integración sin mezclarlo todo en “el prompt”.

TÉRMINO	DEFINICIÓN
API de modelos	Interfaz para enviar entradas a un modelo y recibir salidas bajo un contrato técnico.
Mensaje	Pieza de conversación con rol y contenido.
Rol	Etiqueta que indica qué función cumple un mensaje dentro de la petición.
Streaming	Entrega progresiva de la respuesta por eventos o fragmentos.
Salida estructurada	Respuesta obligada a cumplir un schema.
Schema	Contrato que define campos, tipos y restricciones de una salida.
Tool call	Petición del modelo para que el sistema ejecute una función externa.
Validador	Código que comprueba si una salida cumple el contrato esperado.
Contenido multimodal	Entrada formada por texto, imágenes, documentos, audio o vídeo según lo permita el modelo.
Adaptador de proveedor	Capa que traduce el contrato propio de la aplicación al formato de una API concreta.
Tool choice	Opción que limita o fuerza qué herramienta puede pedir el modelo durante una llamada.
Idempotencia	Propiedad de repetir una operación sin duplicar efectos cuando hay reintentos.
SDK	Biblioteca cliente que envuelve una API desde un lenguaje concreto.
ADK	Framework para construir agentes con tools, sesiones, memoria, workflows y observabilidad.
MCP	Protocolo para conectar aplicaciones de IA con herramientas, recursos y contexto externos.
A2A	Protocolo para que agentes independientes se descubran, se envíen mensajes y coordinen tareas.
Agent Card	Documento de metadatos que publica identidad, endpoint, capacidades y requisitos de acceso de un agente.
Traza	Registro mínimo de lo enviado, recibido y validado para poder depurar.

Antes de pasar página

- ☐ ¿Puedo explicar por qué una API de modelos no es solo un prompt por HTTP? (Si no, vuelve a «Qué no es una API de modelos».)
- ☐ ¿Sé comparar OpenAI Responses API, Claude Messages API y Gemini API sin memorizar cada SDK? (Si no, vuelve a «Qué APIs se usan hoy y quién las usa».)
- ☐ ¿Reconozco las familias de parámetros: modelo, entrada, generación, tools, schema, streaming y operación? (Si no, vuelve a «Los parámetros: no memorizar nombres, entender familias».)
- ☐ ¿Sé separar instrucciones, mensaje de usuario, contexto recuperado, tool y salida? (Si no, vuelve a «Mensajes: separar instrucciones, contexto y petición».)
- ☐ ¿Entiendo qué cambia cuando la entrada incluye documentos o imágenes? (Si no, vuelve a «Documentos e imágenes: multimodal no significa comprensión automática».)
- ☐ ¿Puedo distinguir salida estructurada de tool call? (Si no, vuelve a «Tool calls: cuando responder no basta».)
- ☐ ¿Entiendo por qué JSON válido no significa decisión correcta? (Si no, vuelve a «Salidas estructuradas: cuando el texto debe convertirse en dato».)
- ☐ ¿Sé explicar cuándo streaming mejora experiencia y qué complica? (Si no, vuelve a «Streaming: que la respuesta llegue por partes».)
- ☐ ¿Puedo describir una capa de adaptador que proteja mi producto de cambios de proveedor? (Si no, vuelve a «Buenas prácticas de integración».)
- ☐ ¿Puedo distinguir API, SDK, Agents SDK, ADK, MCP y A2A sin meterlos en el mismo saco? (Si no, vuelve a «SDKs, ADK, MCP y A2A: cada cosa en su capa».)
- ☐ ¿Sé explicar qué son `AgentCard`, `Task`, `Message`, `Part` y `Artifact` dentro de A2A? (Si no, vuelve a «SDKs, ADK, MCP y A2A: cada cosa en su capa».)
- ☐ ¿Puedo diseñar un schema mínimo para una respuesta que usará otro software? (Si no, vuelve a «Salidas estructuradas: cuando el texto debe convertirse en dato».)
- ☐ ¿Sé diseñar un caso que el validador de salida estructurada deba rechazar y por qué? (Si no, repasa «Salidas estructuradas: cuando el texto debe convertirse en dato».)

En resumen

Una buena integración trata la API como contrato. El modelo produce una salida; tu aplicación decide cómo construir la petición, validar la respuesta y registrar lo necesario.

IDEA FUERZA	DETALLE
La API no es una caja de texto.	Es una frontera entre software probabilístico y software verificable.
Los proveedores cambian de dialecto, no de problema.	OpenAI, Claude y Gemini piden piezas parecidas con nombres y límites distintos.
Los mensajes separan responsabilidades.	Instrucciones, usuario, contexto y tools no deberían vivir en una sola cadena.
La multimodalidad exige trazabilidad.	Si entran documentos o imágenes, guarda fuente, versión, página y criterio de validación.
El schema convierte texto en dato.	Pero aún necesitas validar si el contenido tiene sentido.
Streaming mejora la experiencia percibida.	También obliga a manejar estados parciales y cancelación.
Una tool call no es ejecución automática.	La aplicación valida argumentos y decide qué hacer.
El adaptador protege tu producto.	Tu dominio debería hablar su propio contrato y traducirlo al proveedor.
SDK y ADK no son lo mismo.	El SDK llama APIs; ADK o Agents SDK orquestan agentes, tools, sesiones y workflows.
MCP y A2A resuelven fronteras distintas.	MCP conecta herramientas y recursos; A2A conecta agentes completos mediante tareas y artefactos.
Sin trazas no hay depuración seria.	Guardar contrato, versión y resultado ayuda a reproducir problemas.

Para saber más

Anthropic. (2026). *Messages API*. <https://platform.claude.com/docs/en/api/messages>

Anthropic. (2026). *PDF support*. <https://platform.claude.com/docs/en/build-with-claude/pdf-support>

Anthropic. (2026). *Streaming messages*. <https://platform.claude.com/docs/en/build-with-claude/streaming>

Anthropic. (2026). *Structured outputs*. <https://platform.claude.com/docs/en/build-with-claude/structured-outputs>

Anthropic. (2026). *Vision*. <https://platform.claude.com/docs/en/build-with-claude/vision>

A2A Protocol. (2026). *Agent2Agent (A2A) Protocol*. <https://a2a-protocol.org/latest/>

A2A Protocol. (2026). *Overview specification*. <https://a2a-protocol.org/latest/specification/>

Google. (2026). *ADK with Agent2Agent (A2A) Protocol*. <https://adk.dev/a2a/>

Google. (2026). *Agent Development Kit*. <https://adk.dev/>

Google. (2026). *Document understanding*. <https://ai.google.dev/gemini-api/docs/document-processing>

Google. (2026). *Image understanding*. <https://ai.google.dev/gemini-api/docs/image-understanding>

Google. (2026). *Model Context Protocol (MCP)*. <https://adk.dev/mcp/>

Google. (2026). *Structured outputs*. <https://ai.google.dev/gemini-api/docs/structured-output>

Google. (2026). *Text generation*. <https://ai.google.dev/gemini-api/docs/text-generation>

JSON Schema. (2020). *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. <https://json-schema.org/draft/2020-12/json-schema-validation>

OpenAI. (2026). *Create a model response*. <https://developers.openai.com/api/docs/api-reference/responses/create>

OpenAI. (2026). *File inputs*. <https://developers.openai.com/api/docs/guides/file-inputs>

OpenAI. (2026). *Function calling*. <https://developers.openai.com/api/docs/guides/function-calling>

OpenAI. (2026). *Images and vision*. <https://developers.openai.com/api/docs/guides/images-vision>

OpenAI. (2026). *Agents SDK*. <https://developers.openai.com/api/docs/guides/agents>

OpenAI. (2026). *SDKs and CLI*. <https://developers.openai.com/api/docs/libraries>

OpenAI. (2026). *Streaming API responses*. <https://developers.openai.com/api/docs/guides/streaming-responses>

OpenAI. (2026). *Structured model outputs*. <https://developers.openai.com/api/docs/guides/structured-outputs>

OpenAI. (2026). *Text generation*. <https://developers.openai.com/api/docs/guides/text>

WHATWG. (2026). *Server-sent events*. <https://html.spec.whatwg.org/multipage/server-sent-events.html>

Notas

1. Lista de documentos consultados el 10 de junio de 2026. OpenAI: *Create a model response* (<https://developers.openai.com/api/docs/api-reference/responses/create>), *Text generation* (<https://developers.openai.com/api/docs/guides/text>), *File inputs* (<https://developers.openai.com/api/docs/guides/file-inputs>), *Images and vision* (<https://developers.openai.com/api/docs/guides/images-vision>), *Structured model outputs* (<https://developers.openai.com/api/docs/guides/structured-outputs>), *Function calling* (<https://developers.openai.com/api/docs/guides/function-calling>), *Streaming API responses* (<https://developers.openai.com/api/docs/guides/streaming-responses>), *SDKs and CLI* (<https://developers.openai.com/api/docs/libraries>), *Agents SDK* (<https://developers.openai.com/api/docs/guides/agents>). Anthropic: *Messages API* (<https://platform.claude.com/docs/en/api/messages>), *Vision* (<https://platform.claude.com/docs/en/build-with-claude/vision>), *PDF support* (<https://platform.claude.com/docs/en/build-with-claude/pdf-support>), *Streaming messages* (<https://platform.claude.com/docs/en/build-with-claude/streaming>), *Structured outputs* (<https://platform.claude.com/docs/en/build-with-claude/structured-outputs>). Google: *Text generation* (<https://ai.google.dev/gemini-api/docs/text-generation>), *Document understanding* (<https://ai.google.dev/gemini-api/docs/document-processing>), *Image understanding* (<https://ai.google.dev/gemini-api/docs/image-understanding>), *Structured outputs* (<https://ai.google.dev/gemini-api/docs/structured-output>), *Agent Development Kit* (<https://adk.dev/>), *Model Context Protocol* (<https://adk.dev/mcp/>), *ADK with A2A* (<https://adk.dev/a2a/>). A2A Protocol: especificación oficial (<https://a2a-protocol.org/latest/specification/>). JSON Schema 2020-12: *Validation* (<https://json-schema.org/draft/2020-12/json-schema-validation>). WHATWG: *Server-sent events* (<https://html.spec.whatwg.org/multipage/server-sent-events.html>). ↵
2. OpenAI. (2026). *Text generation*. <https://developers.openai.com/api/docs/guides/text>. Consultado el 10 de junio de 2026. ↵
3. OpenAI. (2026). *Structured model outputs*. <https://developers.openai.com/api/docs/guides/structured-outputs>. Consultado el 10 de junio de 2026. ↵
4. OpenAI. (2026). *Function calling*. <https://developers.openai.com/api/docs/guides/function-calling>. Consultado el 10 de junio de 2026. ↵
5. Anthropic. (2026). *Messages API*. <https://platform.claude.com/docs/en/api/messages>. Consultado el 10 de junio de 2026. ↵
6. Anthropic. (2026). *Streaming messages*. <https://platform.claude.com/docs/en/build-with-claude/streaming>. Consultado el 10 de junio de 2026. ↵
7. Anthropic. (2026). *Structured outputs*. <https://platform.claude.com/docs/en/build-with-claude/structured-outputs>. Consultado el 10 de junio de 2026. ↵

8. Google. (2026). *Text generation*. <https://ai.google.dev/gemini-api/docs/text-generation>. Consultado el 10 de junio de 2026. ↵
9. Google. (2026). *Structured outputs*. <https://ai.google.dev/gemini-api/docs/structured-output>. Consultado el 10 de junio de 2026. ↵
10. Google. (2026). *Interactions API overview*. <https://ai.google.dev/gemini-api/docs/interactions/interactions-overview>. Consultado el 10 de junio de 2026. ↵
11. Google. (2026). *Agent Development Kit*. <https://adk.dev/>. Consultado el 10 de junio de 2026. ↵
12. A2A Protocol. (2026). *Agent2Agent Protocol*. <https://a2a-protocol.org/latest/>. Consultado el 10 de junio de 2026. ↵
13. OpenAI. (2026). *Create a model response*. <https://developers.openai.com/api/docs/api-reference/responses/create>. Consultado el 10 de junio de 2026. ↵
14. Anthropic. (2026). *Messages API*. <https://platform.claude.com/docs/en/api/messages>. Consultado el 10 de junio de 2026. ↵
15. Google. (2026). *Text generation*. <https://ai.google.dev/gemini-api/docs/text-generation>. Consultado el 10 de junio de 2026. ↵
16. OpenAI. (2026). *File inputs*. <https://developers.openai.com/api/docs/guides/file-inputs>. Consultado el 10 de junio de 2026. ↵
17. OpenAI. (2026). *Images and vision*. <https://developers.openai.com/api/docs/guides/images-vision>. Consultado el 10 de junio de 2026. ↵
18. Anthropic. (2026). *Vision*. <https://platform.claude.com/docs/en/build-with-claude/vision>. Consultado el 10 de junio de 2026. ↵
19. Anthropic. (2026). *PDF support*. <https://platform.claude.com/docs/en/build-with-claude/pdf-support>. Consultado el 10 de junio de 2026. ↵
20. Google. (2026). *Image understanding*. <https://ai.google.dev/gemini-api/docs/image-understanding>. Consultado el 10 de junio de 2026. ↵
21. Google. (2026). *Document understanding*. <https://ai.google.dev/gemini-api/docs/document-processing>. Consultado el 10 de junio de 2026. ↵
22. JSON Schema. (2020). *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. <https://json-schema.org/draft/2020-12/json-schema-validation>. ↵
23. WHATWG. (2026). *Server-sent events*. <https://html.spec.whatwg.org/multipage/server-sent-events.html>. Consultado el 10 de junio de 2026. ↵

14. OpenAI. (2026). *Streaming API responses*. <https://developers.openai.com/api/docs/guides/streaming-responses>. Consultado el 10 de junio de 2026. ↵
15. Anthropic. (2026). *Streaming messages*. <https://platform.claude.com/docs/en/build-with-claude/streaming>. Consultado el 10 de junio de 2026. ↵
16. Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. <https://arxiv.org/abs/2309.06180>. La métrica TTFT y el ritmo de decodificación se formalizan en la literatura de serving, p. ej. Yinmin Zhong y otros (2024). *DistServe*. OSDI 2024. <https://arxiv.org/abs/2401.09670>. ↵
17. OpenAI. (2026). *SDKs and CLI*. <https://developers.openai.com/api/docs/libraries>. Consultado el 10 de junio de 2026. ↵
18. OpenAI. (2026). *Agents SDK*. <https://developers.openai.com/api/docs/guides/agents>. Consultado el 10 de junio de 2026. ↵
19. Google. (2026). *Agent Development Kit*. <https://adk.dev/>. Consultado el 10 de junio de 2026. ↵
20. Google. (2026). *ADK with Agent2Agent (A2A) Protocol*. <https://adk.dev/a2a/>. Consultado el 10 de junio de 2026. ↵
21. Google. (2026). *Model Context Protocol (MCP)*. <https://adk.dev/mcp/>. Consultado el 10 de junio de 2026. ↵
22. A2A Protocol. (2026). *Agent2Agent (A2A) Protocol*. <https://a2a-protocol.org/latest/>. Consultado el 10 de junio de 2026. ↵
23. A2A Protocol. (2026). *Overview specification*. <https://a2a-protocol.org/latest/specification/>. Consultado el 10 de junio de 2026. ↵

Cuadernos para practicar

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



Salidas estructuradas: que el modelo rellene tu formulario

<https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%3ADmil%2004/05-salidas-estructuradas.ipynb>

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:

PA SO	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:

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

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^*, v^*)	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	<code>system</code> , <code>tool</code> , 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:

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

SÍMBOL O	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:

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

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{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 <code>max_output_tokens</code> 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:

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

SÍMBOL O	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:

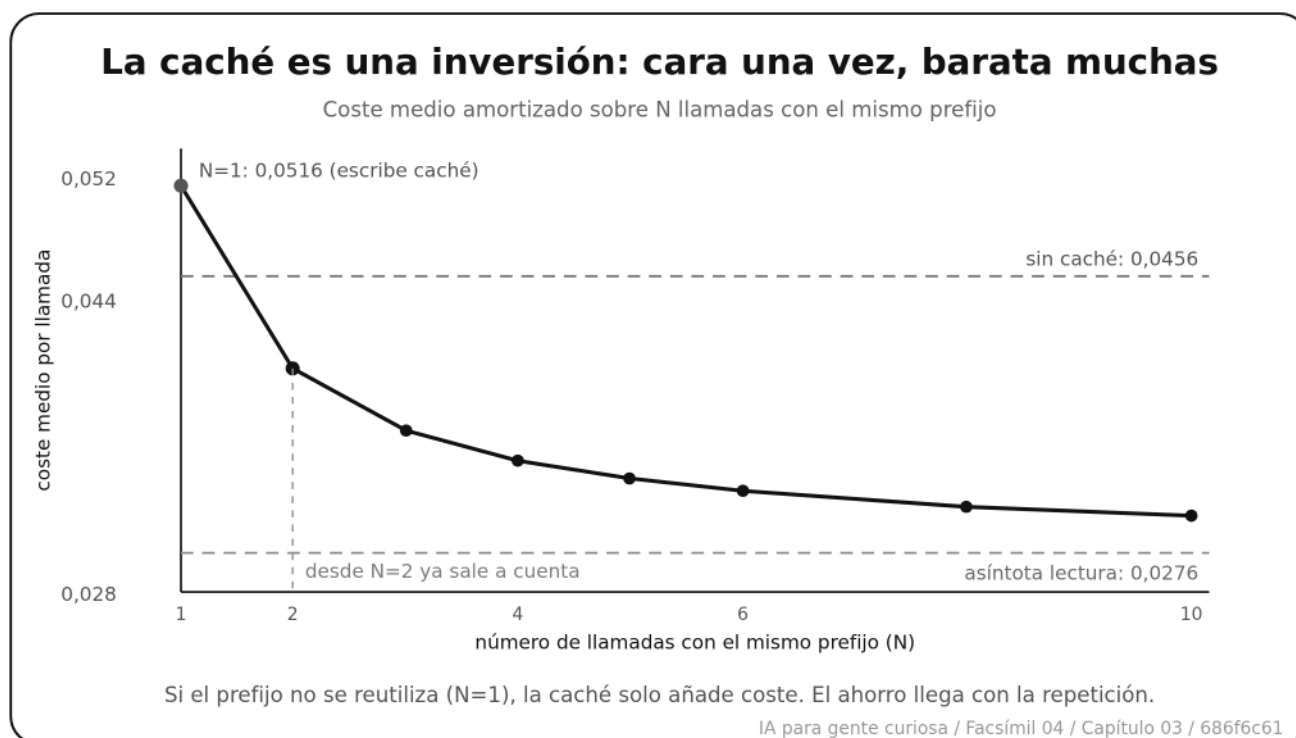
$$C_{\text{write}} = \frac{8000 \cdot 2,00 + 12000 \cdot 2,50 + 700 \cdot 8,00}{10^6} = 0,0516$$

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

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

SÍMBOL O	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:

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

SÍMBOL O	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

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

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

SÍMBOL O	SIGNIFICADO	EJEMPLO
$h_t^{(\ell)}$	Estado del token t al entrar en la capa ℓ .	Vector del token “cache” en la capa 12.
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:

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

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

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

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

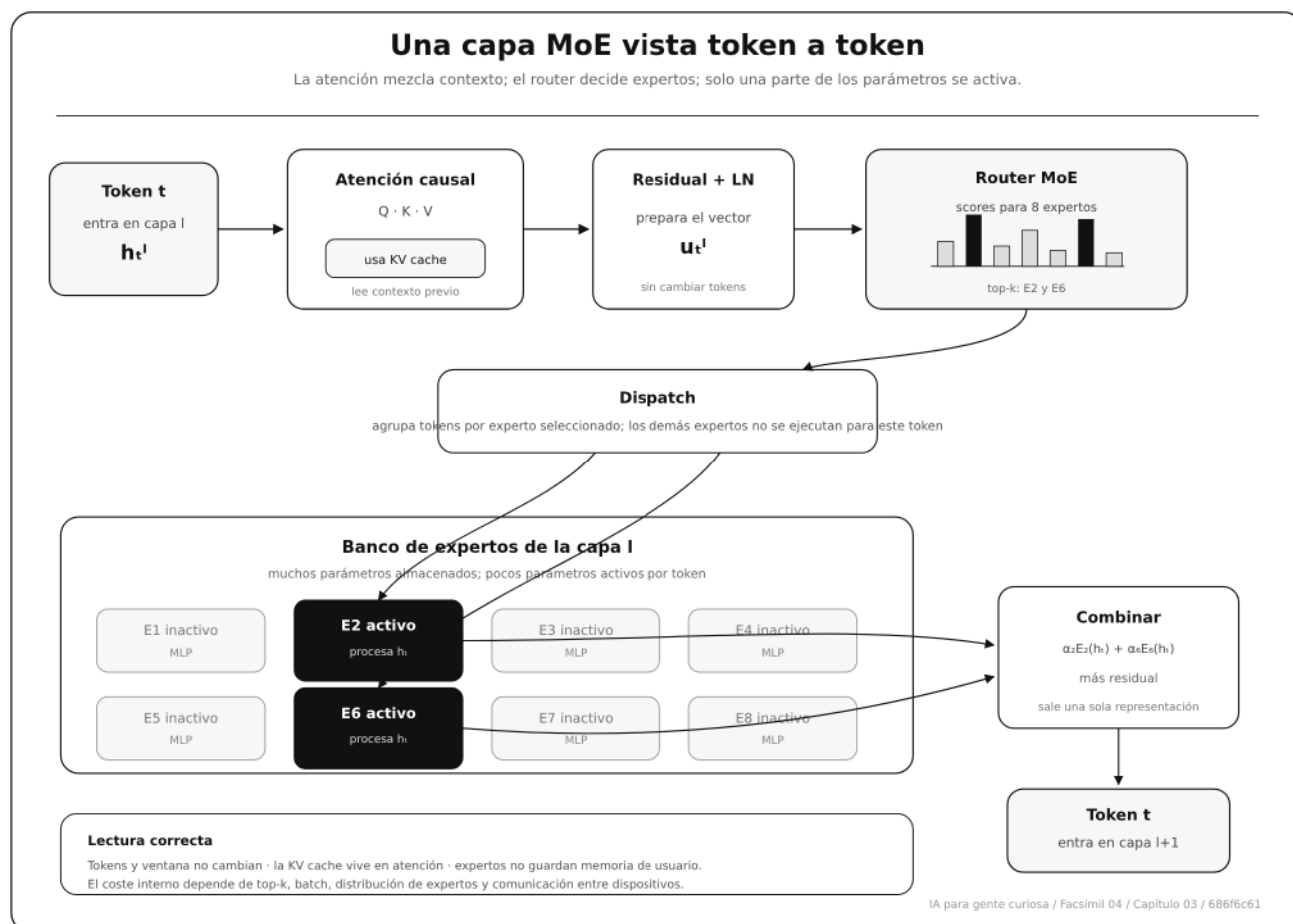
$$h_t^{(\ell+1)} = u_t^{(\ell)} + \text{MoE}^{(\ell)}(u_t)$$

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.
$C_i^{(\ell)}$	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 CACHE
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.

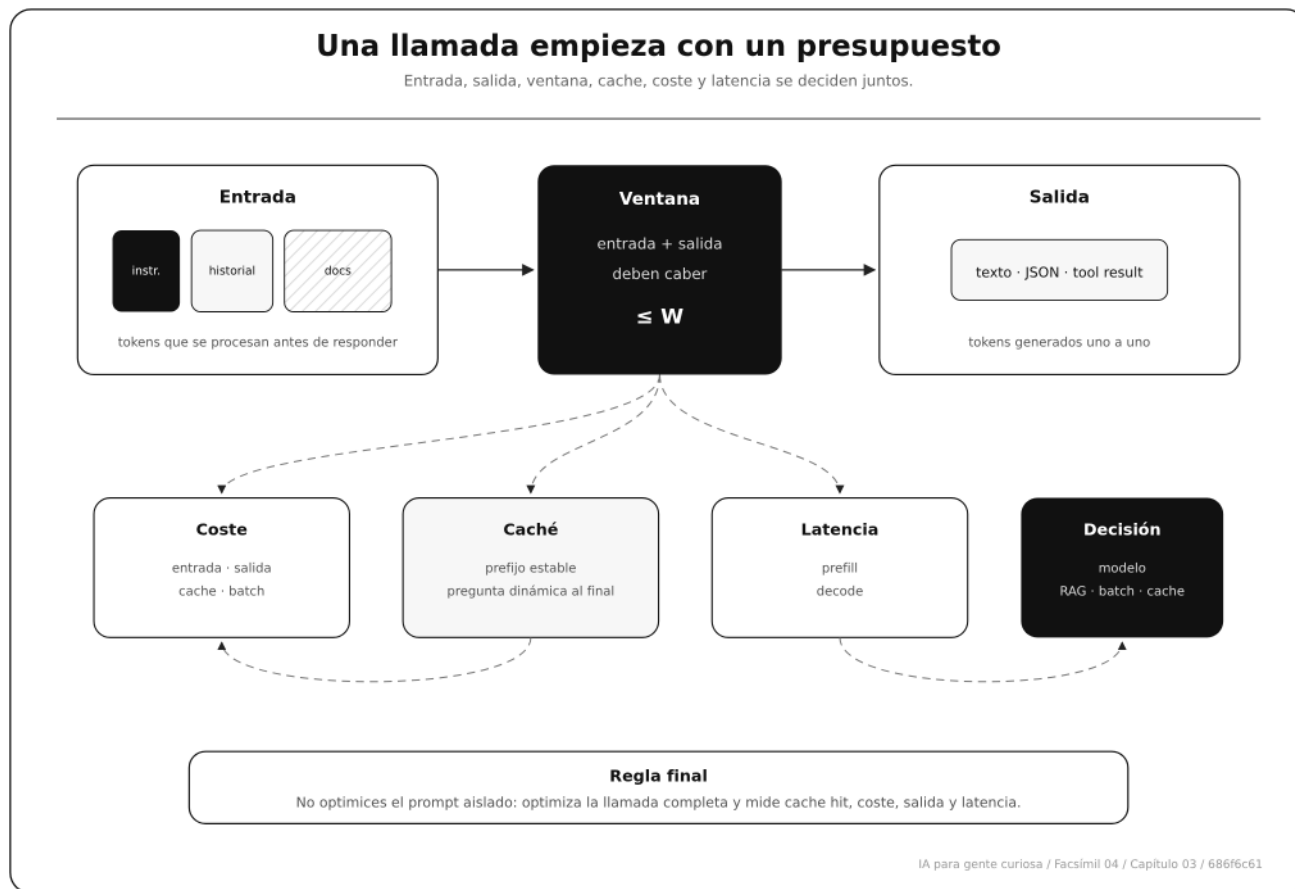
Para entenderlo antes de tocar código

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.

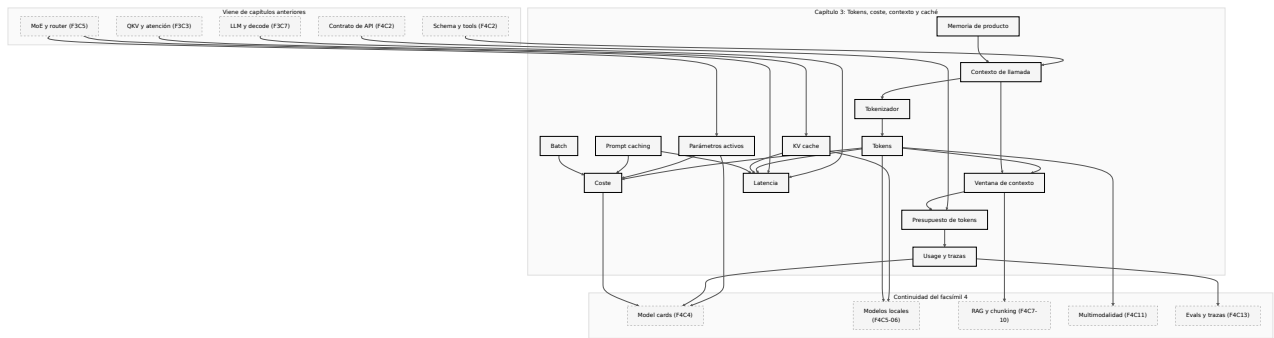
Dónde solía tropezar yo

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>. ↵

- .6. Albert Q. Jiang y otros (2024). *Mixtral of Experts*. <https://arxiv.org/abs/2401.04088>. ↵
- .7. OpenAI. (2026). *tiktoken*. <https://github.com/openai/tiktoken>. Consultado el 25 de mayo de 2026. ↵
- .8. Rico Sennrich, Barry Haddow y Alexandra Birch. (2016). *Neural Machine Translation of Rare Words with Subword Units*. ACL. <https://aclanthology.org/P16-1162/>. ↵
- .9. Rico Sennrich, Barry Haddow y Alexandra Birch. (2016). *Neural Machine Translation of Rare Words with Subword Units*. ACL. <https://arxiv.org/abs/1508.07909>. ↵
- !0. 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/>. ↵
- !1. OpenAI. (2026). *Pricing*. <https://developers.openai.com/api/docs/pricing>. Consultado el 25 de mayo de 2026. ↵
- !2. OpenAI. (2026). *Pricing*. <https://developers.openai.com/api/docs/pricing>. Consultado el 25 de mayo de 2026. ↵
- !3. Ashish Vaswani y otros (2017). *Attention Is All You Need*. NeurIPS. <https://papers.nips.cc/paper/7181-attention-is-all-you-need>. ↵
- !4. Joshua Ainslie y otros (2023). *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. EMNLP. <https://arxiv.org/abs/2305.13245>. ↵
- !5. Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. <https://arxiv.org/abs/2309.06180>. ↵
- !6. Ashish Vaswani y otros (2017). *Attention Is All You Need*. NeurIPS. <https://papers.nips.cc/paper/7181-attention-is-all-you-need>. ↵
- !7. 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>. ↵
- !8. 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>

Capítulo 04: Model cards y elección de modelos

Elegir modelo sin dejarse arrastrar por el escaparate

Elegir modelo parece una decisión de catálogo: abres una tabla, miras cuál aparece arriba y lo usas. En proyectos reales casi nunca funciona así. El modelo que gana un benchmark general puede ser caro, lento, excesivo, poco conveniente por licencia, incómodo para tu runtime o flojo justo en el idioma, formato y dominio que necesitas.

Venimos del [capítulo 03](#), donde vimos que tokens, coste, contexto y caché convierten una llamada aparentemente sencilla en una decisión de ingeniería. Aquí añadimos la siguiente capa: **cómo leer la ficha de un modelo y decidir con criterio.**

La idea central es esta: **no eliges el mejor modelo en abstracto; eliges el modelo que cumple una tarea, bajo restricciones, con evidencia suficiente.**

Estado del arte con fecha de corte

Fecha de corte: 14 de junio de 2026.

Fuentes consultadas: documentación oficial de modelos y precios de OpenAI, Anthropic y Google Gemini API; documentación de Hugging Face sobre model cards; definición de Open Source AI y Open Weights de la Open Source Initiative; fichas/licencias de gpt-oss, Qwen, Mistral, DeepSeek, Gemma y Llama; artículos académicos sobre model cards, datasheets y evaluación holística; y benchmarks de sistemas de inferencia.

Lo estable es el método: documentar el modelo, leer uso previsto, capacidades, límites, datos, licencia, coste, contexto, evaluación y condiciones de despliegue. Lo cambiante son los nombres de modelos, ventanas máximas, precios, disponibilidad regional, versiones `preview`, modelos retirados, rate limits y capacidades por API.

Actualización de apertura de modelos: el 14 de junio de 2026 se revisó además la distinción entre modelos cerrados, pesos abiertos, código abierto y Open Source AI. Es una parte especialmente cambiante porque los proveedores publican nuevas familias, licencias y repositorios con mucha frecuencia. Por eso en esta sección no basta con decir “modelo abierto”: hay que decir **qué está abierto, bajo qué licencia, qué puedes modificar, qué no puedes reproducir y qué KPI vas a medir.**

Para no convertir el capítulo en una lista que caduca, dejamos las fuentes ordenadas por función:

FUENTE	QUÉ APORTA	CÓMO USARLA
Model Cards for Model Reporting. ¹	Marco original para documentar modelos, usos, métricas y límites.	Como plantilla mental de lectura.
Datasheets for Datasets. ²	Documentación de datos, motivación, composición, recogida y mantenimiento.	Para no leer una model card sin preguntar por datos.
Hugging Face Model Cards. ³	Implementación práctica: README con metadatos, licencia, datasets, evaluación y texto explicativo.	Para leer modelos abiertos o open weights.
OpenAI Models. ⁴	Catálogo vivo de modelos, modalidades y usos previstos.	Como foto actual, no como verdad permanente.
OpenAI Pricing. ⁵	Precios por modalidad, entrada, salida, cache, batch y variantes de servicio.	Para calcular coste por tarea, no solo coste por token.
Anthropic Models. ⁶	Comparación de modelos Claude, ventanas y capacidades declaradas.	Para contrastar familia, contexto y capacidades.
Anthropic Pricing. ⁷	Coste de entrada, salida, cache y operaciones por lote.	Para no confundir modelo potente con solución asumible.
Google Gemini Models. ⁸	Modelos Gemini, estados estable/preview/experimental y modalidades.	Para vigilar estabilidad de versión y API concreta.
Google Gemini Pricing. ⁹	Precios por modelo, modalidad, contexto y uso de API.	Para revisar el coste justo el día de la decisión.
HELM. ¹⁰	Evaluación amplia con múltiples escenarios, métricas y transparencia de resultados.	Para desconfiar de rankings de una sola métrica.
MLPerf Inference. ¹¹	Benchmark de sistemas completos de inferencia.	Para recordar que modelo, runtime y hardware van juntos.

Modelo cerrado, pesos abiertos y código abierto no son lo mismo

Aquí conviene ir despacio, porque esta confusión aparece muchísimo. En software tradicional, “código abierto” suele significar que puedes inspeccionar, modificar, compilar y redistribuir el código bajo ciertas condiciones. En IA generativa, el objeto que usamos no es solo código. Hay arquitectura, pesos, tokenizer, datos de entrenamiento, filtros, scripts de entrenamiento, receta de post-training, evaluaciones, plantillas de chat, runtime y, a veces, una API gestionada que no expone nada de eso.

Por eso decir “este modelo es abierto” sin apellido es demasiado vago. Puede significar una de varias cosas:

1. El modelo está disponible por API, pero los pesos son cerrados.
2. Los pesos se pueden descargar, pero no se publican los datos ni la receta completa de entrenamiento.
3. Hay código de inferencia abierto, pero los pesos o datos no lo son.
4. Hay pesos, código y suficiente información de datos/proceso para estudiar y modificar el sistema de forma profunda.

La Open Source Initiative formaliza una definición de Open Source AI basada en libertades de uso, estudio, modificación y compartición. Además, exige acceso a la forma preferida para modificar el sistema: información suficientemente detallada de datos, código usado para procesar y entrenar, y parámetros o pesos bajo términos adecuados.¹² Esa definición es más exigente que “puedo descargar un checkpoint”.

La propia OSI distingue también los **open weights**: publicar los pesos finales ayuda mucho, pero normalmente no incluye el código de entrenamiento completo, los datos, los filtros, los checkpoints intermedios ni la receta suficiente para reproducir el modelo.¹³ Dicho de forma cercana: tener los pesos es como tener el edificio terminado; ayuda a vivir dentro, reformar algunas habitaciones y medir su consumo, pero no te da necesariamente los planos completos, el origen de todos los materiales ni el diario de obra.

Qué son exactamente los pesos

Los pesos son los números aprendidos durante el entrenamiento. En una red neuronal, cada capa transforma vectores usando matrices y funciones no lineales. Esas matrices contienen parámetros. Después de entrenar, esos parámetros quedan fijados en archivos: `safetensors`, `gguf`, checkpoints de PyTorch, formatos optimizados o variantes cuantizadas. Cuando alguien dice “pesos abiertos”, normalmente quiere decir: “puedes obtener esos números y cargarlos en un runtime compatible”.

Pero los pesos no son el modelo completo en sentido práctico. Para usarlos bien necesitas, como mínimo:

PIEZA	POR QUÉ IMPORTA
Arquitectura	Define cómo se conectan los pesos: capas, atención, MoE, normalización, activaciones.
Configuración	Define dimensiones, número de capas, vocabulario, contexto, precisión y detalles de carga.
Tokenizer	Convierte texto en tokens. Si cambias tokenizer, cambias la entrada real.
Chat template	Convierte roles y mensajes en el formato que el modelo espera.
Runtime	Ejecuta el modelo: Transformers, vLLM, SGLang, llama.cpp, Ollama, TensorRT-LLM, etc.
Licencia	Decide qué puedes hacer legalmente: usar, modificar, vender, redistribuir, servir.
Evals	Te dicen si ese modelo, con esa configuración, sirve para tu caso.

Por eso un modelo con pesos abiertos puede ser tremendamente útil y, aun así, no ser “Open Source AI” en sentido estricto. Puede que no puedas reproducir su entrenamiento, auditar todos sus datos, conocer sus filtros o verificar por qué aprendió ciertos sesgos. En ingeniería, eso cambia cómo lo documentas: no lo vendes como sistema transparente total, sino como **artefacto ejecutable con más control operativo que una API cerrada**.

Las cuatro categorías que conviene enseñar al alumno

CATEGORÍA	QUÉ TIENES	QUÉ NO TIENES NECESARIAMENTE	EJEMPLOS A 14 DE JUNIO DE 2026	DECISIÓN DE INGENIERÍA
Modelo cerrado o por API	Endpoint, documentación, precios, SLA o contrato, herramientas del proveedor.	Pesos, datos, receta de entrenamiento, capacidad de servirlo tú.	GPT-5.5/GPT-5.4 en OpenAI API, Claude en Anthropic API, Gemini en Gemini API. ¹⁴	Útil si necesitas capacidad alta, menor operación propia, herramientas integradas y contrato de servicio. Mide dependencia, coste, privacidad y plan de salida.
Pesos abiertos permisivos	Pesos descargables y licencia amplia, normalmente compatible con uso comercial.	Datos completos, receta reproducible, garantías de seguridad o coste operativo bajo.	gpt-oss-120b/20b bajo Apache 2.0; Qwen3.6 open-weight Apache 2.0; Mistral Large 3 bajo Apache 2.0; DeepSeek-R1 con licencia MIT. ¹⁵	Bueno para control, despliegue propio, fine-tuning, privacidad y coste a escala. Mide VRAM, throughput, equipo de operación y calidad en tu eval.
Pesos abiertos con licencia propia	Pesos accesibles, documentación y permiso condicionado.	Libertad tipo MIT/Apache. Puede haber restricciones de escala, uso, redistribución o política aceptable.	Llama 4 bajo Llama 4 Community License; Gemma con open weights y términos propios de Google. ¹⁶	Técnicamente puede ser muy atractivo, pero legalmente no lo trates como “open source clásico”. Revisa licencia con el caso de uso concreto.
Open Source AI estricto	Libertades de uso, estudio, modificación y compartición, más forma preferida de modificación: datos/información de datos, código y parámetros.	No siempre existe en modelos frontier modernos.	La OSI define el criterio; muchos modelos llamados “abiertos” solo cumplen parte.	Útil como vara conceptual y de auditoría. Pregunta siempre: ¿qué falta para reproducir, auditar y modificar de verdad?

La tabla ayuda, pero el cuerpo de la idea es este: **cada nivel de apertura compra una libertad distinta y deja una deuda distinta**. Una API cerrada compra velocidad de adopción, pero deja dependencia. Un open weight compra control, pero deja operación. Un open weight permisivo compra posibilidad de modificar y servir, pero no necesariamente transparencia científica. Un Open Source AI completo compraría auditabilidad profunda, pero todavía no es la norma en modelos frontier.

El espectro de apertura

“Abierto” sin apellido no dice nada: lo que importa es qué expone cada nivel

	Pesos	Lic. libre	Código	Datos
más cerrado				
Cerrado (solo API)	X	X	X	X
Pesos abiertos, licencia propia	✓	X	X	X
Pesos abiertos permisivos	✓	✓	X	X
Open Source AI (OSI)	✓	✓	✓	✓

más abierto

Tener los pesos no es tener el modelo: la licencia, el código y los datos deciden qué puedes servir, modificar, auditar y reproducir.

IA para gente curiosa / Facsímil 04 / Capítulo 04 / 686f6c61

KPIs para decidir entre cerrado, pesos abiertos y open source

Una discusión madura no pregunta “¿abierto o cerrado?” como si fuera una preferencia moral. Pregunta qué KPI importa para el sistema. KPI aquí no significa poner números por ponerlos; significa elegir indicadores que cambian la decisión.

KPI	QUÉ MIDE	CÓMO SE CALCULA O SE OBSERVA	QUÉ VALOR SERÍA RAZONABLE
Calidad propia	Si resuelve tus casos, no el benchmark general.	Eval con casos de tu dominio: exactitud, rúbrica, tasa de formato válido, revisión humana.	Se fija por tarea: por ejemplo, >95 % JSON válido y >85 % acierto en casos críticos.
Coste por tarea completada	Coste real por respuesta útil, no por token aislado.	Tokens de entrada + salida + cache + batch + repeticiones + fallos.	El modelo barato que repite tres veces puede salir caro; mide coste por caso aprobado.
Latencia p95	Tiempo que sufre el 5 % más lento de usuarios.	Medición bajo carga con red, runtime, batch y contexto reales.	Interactivo: quizá <2-5 s; batch: puede ser minutos si está justificado.
Throughput	Cuántas peticiones o tokens procesas por segundo.	Requests/s, tokens/s, concurrencia y cola.	Importa mucho en open weights: una GPU infrautilizada destruye el TCO.
Coste de operación	Trabajo humano y técnico de mantener el sistema.	Horas de SRE/ML, actualizaciones, incidentes, monitorización, GPUs, parches.	API cerrada suele bajar operación propia; self-hosting la sube.
Riesgo de licencia	Probabilidad de que el uso viole términos o bloquee producto.	Revisión de licencia, restricciones de uso, redistribución, derivados, atribución.	Si el caso es comercial o regulado, licencia dudosa es filtro duro, no penalización pequeña.
Control de datos	Qué ocurre con prompts, documentos, logs y salidas.	Retención, región, entrenamiento con datos, cifrado, contratos, despliegue on-prem.	Datos sensibles pueden empujar a self-hosting o contrato enterprise fuerte.
Reproducibilidad	Capacidad de repetir la misma evaluación y volver a la misma versión.	Versión exacta, commit, hash de pesos, configuración, seeds, prompt, dataset.	En producción: modelo y configuración fijados; alias latest solo si aceptas cambio.
Capacidad de adaptación	Facilidad para fine-tuning, LoRA, cuantización o routing.	Acceso a pesos, licencia, tooling PEFT, soporte runtime.	Open weights gana aquí si el equipo sabe medir degradación.
Portabilidad	Facilidad de cambiar proveedor o mover despliegue.	Compatibilidad OpenAI-like, vLLM/SGLang, formatos, prompts, schemas, evals.	Cuanto más crítica la app, más importante mantener segundo candidato.
Observabilidad	Capacidad de ver fallos, costes y comportamiento.	Logs, trazas, métricas, prompts, outputs, tokens, latencia, errores por slice.	No uses un modelo que no puedes medir en el nivel que exige el riesgo.

KPI	QUÉ MIDE	CÓMO SE CALCULA O SE OBSERVA	QUÉ VALOR SERÍA RAZONABLE
Gobernanza	Capacidad de auditar qué se usa, por qué y bajo qué condiciones.	Model card interna, fecha, licencia, DPIA si aplica, evals, aprobaciones.	En entornos profesionales, una decisión sin ficha interna es memoria frágil.

Estos KPIs no pesan igual siempre. Para un prototipo de clase, calidad y facilidad de ejecución pueden bastar. Para un asistente con expedientes privados, privacidad, región, trazas y licencia pesan mucho más. Para una herramienta de código interna, quizá throughput, coste por tarea y capacidad de adaptación sean los criterios principales.

Ejemplo de lectura. Si una organización compara GPT-5.5 por API, Claude por API, Gemini por API, `gpt-oss-120b`, Qwen3.6 y Mistral Large 3, no debería empezar por “cuál es más inteligente”. Debería construir una matriz:

1. Filtros duros: privacidad, modalidad, licencia, región, presupuesto máximo.
2. Eval propia: 100 casos reales con salida esperada o rúbrica.
3. Medición operativa: p50/p95, tokens/s, coste por caso, tasa de retry.
4. Riesgo: dependencia de proveedor, licencia, plan de salida, estabilidad de versión.
5. Decisión: modelo principal, modelo alternativo, fecha de revisión y condiciones de cambio.

La conclusión puede ser híbrida. Por ejemplo: API cerrada para razonamiento difícil y multimodalidad; open weights para tareas repetitivas, datos sensibles o coste a escala; modelo pequeño local para clasificación barata; RAG para conocimiento vivo. Eso no es indecisión. Es arquitectura.

Qué significa “abierto” en una model card

Hugging Face permite declarar licencia en los metadatos de la model card y enlazar archivos `LICENSE`; también permite especificar datasets, `pipeline_tag` y resultados de evaluación estructurados.¹⁷ Eso está muy bien, pero hay que leerlo con cuidado.

Cuando veas una model card, separa estas preguntas:

PREGUNTA	DÓNDE MIRAR	DECISIÓN QUE CAMBIA
¿Puedo descargar pesos?	<code>Files and versions</code> , tamaño, formato, gated access.	Self-hosting, fine-tuning, cuantización.
¿Qué licencia tiene?	Metadata <code>license</code> , archivo <code>LICENSE</code> , términos externos.	Uso comercial, redistribución, derivados.
¿Hay código de inferencia?	README, ejemplos, <code>config.json</code> , runtime recomendado.	Facilidad de ejecución y compatibilidad.
¿Hay código de entrenamiento?	Paper, repo, scripts, argumentos, filtros.	Reproducibilidad y auditoría profunda.
¿Hay datos o información de datos?	Dataset card, README, paper, datasheets.	Riesgo de sesgo, cumplimiento, trazabilidad.
¿Hay evals comparables?	<code>model-index</code> , benchmark, paper, harness.	Calidad relativa y gaps de evaluación.
¿Hay política de uso?	Model card, terms, acceptable use policy.	Riesgo de producto y cumplimiento interno.

Un ejemplo típico: un repo puede tener pesos Apache 2.0 y ser muy útil para producción, pero no publicar dataset completo ni receta reproducible. En el facsímil lo llamaríamos “pesos abiertos permisivos”, no “open source completo”. No es un desprecio; es precisión.

Cómo lo explicaría en una revisión técnica

Una frase mala sería: “usamos un modelo open source porque es gratis”.

Una frase profesional sería: “para esta tarea usamos un modelo con pesos abiertos bajo licencia Apache 2.0, servido con vLLM en infraestructura propia, porque necesitamos control de datos y coste estable a volumen. No afirmamos que sea Open Source AI completo: no tenemos todos los datos de entrenamiento ni la receta reproducible. Lo compensamos con eval propia, model card interna, revisión de licencia, medición de p95 y alternativa API si la calidad cae”.

Y otra frase igualmente profesional sería: “para esta tarea usamos un modelo cerrado por API porque la calidad multimodal y las herramientas integradas superan el coste operativo de servir pesos abiertos. Lo documentamos como dependencia de proveedor, fijamos versión cuando el proveedor lo permite, medimos coste por tarea completada y mantenemos una eval de regresión para migrar”.

La madurez no está en elegir siempre abierto o siempre cerrado. Está en poder explicar qué libertad compras, qué deuda aceptas y qué KPI vigila que la decisión siga siendo buena.

Qué no es una model card

Una model card no es marketing. Puede estar mejor o peor escrita, pero su función no es decir “este modelo es increíble”. Su función es permitir una decisión responsable: qué es, para qué se pensó, dónde se evaluó, qué límites declara, qué licencia tiene y qué condiciones debes revisar antes de usarlo.

Tampoco es una garantía. Que una ficha diga “razonamiento”, “multimodal”, “contexto largo” o “excelente en código” no te dice automáticamente si funcionará en tu flujo. Te da pistas para construir pruebas.

Y no es un benchmark suelto. Una tabla de MMLU, SWE-bench, HumanEval, MMMU o cualquier otra métrica puede ser útil, pero solo mide lo que mide. Si tu producto clasifica incidencias en castellano, resume expedientes internos o genera SQL contra tu esquema, el ranking general es una señal débil.

La trampa más común es leer una model card como si fuera un menú. En realidad hay que leerla como una ficha de compatibilidad: “¿encaja con mi problema, mis datos, mi latencia, mi presupuesto, mis permisos y mi forma de evaluar?”.

Qué sí es: una ficha para decidir sin autoengaño

Una buena model card responde a seis bloques de preguntas:

BLOQUE	PREGUNTAS QUE DEBE RESPONDER
Identidad	¿Qué modelo es, qué versión, qué familia, qué arquitectura y qué modalidad?
Uso previsto	¿Para qué fue diseñado? ¿Qué usos desaconseja?
Entrada y salida	¿Texto, imagen, audio, vídeo, tools, JSON, embeddings? ¿Qué límites tiene?
Evaluación	¿Con qué benchmarks, datasets, idiomas y condiciones se midió?
Operación	¿Contexto, coste, latencia, rate limits, runtime, hardware, versiones estables?
Condiciones	¿Licencia, privacidad, retención de datos, restricciones y obligaciones de atribución?

La palabra “modelo” además puede esconder capas distintas:

NIVEL	QUÉ MIRAS	EJEMPLO DE PREGUNTA
Modelo base	Arquitectura y preentrenamiento.	¿Es base, instruct, MoE, multimodal o de embeddings?
Modelo servido por API	Capacidades y contrato del proveedor.	¿Acepta documentos? ¿Devuelve JSON validable? ¿Tiene tools?
Modelo local	Pesos, formato, cuantización y runtime.	¿GGUF, safetensors, vLLM, Ollama, TensorRT-LLM?
Sistema completo	RAG, tools, memoria, permisos y evals.	¿El fallo viene del modelo o del contexto que le damos?

Una elección madura empieza separando esos niveles. Cambiar de modelo no arregla una mala recuperación de documentos. Un modelo con contexto enorme no sustituye una política de permisos. Un benchmark alto no te exonera de evaluar tus casos.

El tamaño tampoco decide solo. Las leyes de escala ayudaron a entender cómo bajaba la pérdida al aumentar parámetros, datos y cómputo durante el entrenamiento.¹⁸ Después, el trabajo conocido como Chinchilla puso el foco en el equilibrio entre tamaño de modelo y cantidad de datos de entrenamiento.¹⁹ Para elegir en un producto, esa lección se traduce así: no preguntes solo “cuántos parámetros tiene”, pregunta si el modelo resuelve tu tarea con el coste, la latencia y la trazabilidad que puedes sostener.

La matriz mínima de decisión

Elegir con criterio exige convertir preferencias vagas en criterios comparables. No hace falta convertirlo todo en una hoja de cálculo infinita, pero sí conviene explicitar qué pesa más. Esto es exactamente lo que estudia el análisis de decisión multicriterio (MCDA): un campo formal con décadas de literatura sobre cómo agregar criterios heterogéneos en una decisión justificable.²⁰

El método más directo es el **Weighted Sum Model** (modelo de suma ponderada): cada modelo candidato recibe una puntuación que combina sus criterios normalizados con los pesos que tú declaras. Aquí lo extendemos con un término de penalización para criterios que restan en vez de sumar:

$$S(m) = \sum_{j=1}^n w_j \cdot q_j(m) - \sum_{k=1}^r \lambda_k \cdot p_k(m)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$S(m)$	Puntuación final del modelo m .	0,79 para el candidato ganador.
w_j	Peso del criterio positivo j , con $\sum_j w_j = 1$.	Calidad vale 0,22; control de datos vale 0,16.
$q_j(m)$	Valor normalizado del criterio en $[0, 1]$.	Calidad propia 0,80; latencia 0,70.
λ_k	Peso de una penalización.	Penalizar licencia incompatible con 1,0.
$p_k(m)$	Penalización activada.	1 si no cumple privacidad; 0 si cumple.

Dos condiciones hacen que esta fórmula sea rigurosa y no decorativa. Primera: los pesos w_j deben sumar 1, de modo que $S(m)$ es una combinación convexa y las puntuaciones de distintos modelos son comparables en la misma escala. Segunda: los criterios q_j deben estar normalizados al mismo rango $[0, 1]$, porque sumar “latencia en milisegundos” con “calidad en porcentaje” sin normalizar no significa nada.

Los requisitos obligatorios no entran en la suma: actúan **antes**, como cribado conjuntivo. En MCDA esto es el método conjuntivo (satisficing): un candidato que no supera un umbral mínimo en un criterio innegociable queda eliminado, por bien que puntúe en el resto. Si privacidad es obligatoria, no es “un criterio más” con su peso: es un filtro que decide quién llega siquiera a la suma ponderada. Si la latencia p95 tiene que ser menor de 2 segundos, el modelo que no lo cumple queda fuera aunque gane un benchmark.

El flujo práctico suele ser:

FASE	QUÉ HACES	RESULTADO
1. Requisitos	Defines tarea, entrada, salida, usuarios, idioma, latencia y presupuesto.	Lista de restricciones.
2. Filtros duros	Eliminas modelos que no cumplen licencia, modalidad, región, privacidad o contexto.	Lista corta inicial.
3. Lectura de fichas	Revisas model cards, docs de proveedor y versiones.	Hipótesis de encaje.
4. Eval propia	Pruebas casos representativos con salida esperada o rúbrica.	Evidencia en tu tarea.
5. Coste y operación	Mides tokens, p50/p95, fallos, rate limits y mantenimiento.	Coste total razonable.
6. Decisión	Documentas modelo elegido, alternativa y fecha de revisión.	Decisión trazable.

Primero el filtro duro, luego la suma ponderada

Solo puntúan los modelos que superan los requisitos innegociables

Pasan el filtro y se puntúan ($S = \sum w_j \cdot q_j$)



Eliminados por filtro duro (no llegan a puntuar)

open_weight_license_propia	licencia incompatible
open_weight_quantized_no_json	sin salida JSON

Un benchmark alto no rescata a un modelo que no cumple un requisito innegociable.

IA para gente curiosa / Facsímil 04 / Capítulo 04 / 686f6c61

También conviene separar “calidad” de “fiabilidad”. Calidad es que responda bien cuando todo va bien. Fiabilidad es que falle de forma manejable cuando el caso es raro, falta contexto, la entrada está sucia o el formato de salida importa.

La ficha que yo leería antes de elegir

Cuando abras una model card o una página de modelos, no empieces por la frase grande. Empieza por esta lista:

DATO	PREGUNTA INCÓMODA
Nombre exacto y versión	¿Estoy usando una versión estable o un alias que puede cambiar?
Modalidades	¿Texto solo, imagen, audio, vídeo, embeddings, tools?
Contexto	¿Cuánto entra realmente y cuánto debo reservar para salida?
Salida máxima	¿Puede devolver la respuesta completa o tengo que trocear?
Precio	¿Entrada, salida, cache, batch, imágenes, audio, razonamiento?
Latencia esperada	¿Me importa tiempo total o tiempo hasta primer token?
Datos y fecha de entrenamiento	¿Hay conocimiento que no puede saber sin RAG?
Evaluación	¿Los benchmarks se parecen a mi tarea?
Idiomas	¿Se evaluó de verdad en castellano o solo se declara soporte?
Formato de chat	¿Tiene plantilla de mensajes, system, tools, JSON, function calling?
Licencia	¿Puedo usarlo en mi producto, modificarlo, redistribuirlo o servirlo?
Retención y privacidad	¿Qué ocurre con los datos que envío?
Deprecación	¿Hay fecha de retirada, migración o versión recomendada?

Google, por ejemplo, distingue modelos estables, preview, latest y experimentales en su documentación de modelos. Esa clasificación importa porque `latest` o `preview` puede ser útil para explorar, pero no siempre es lo que quieres fijar en producción. OpenAI y Anthropic mantienen páginas vivas de modelos y precios; por eso una decisión profesional debería guardar fecha de consulta y versión exacta, no solo “usamos el modelo bueno”.

Anatomía de una model card en Hugging Face

Hugging Face convierte una model card en una página viva: parte README, parte ficha técnica, parte repositorio de archivos y parte panel operativo. Por eso conviene leerla en capas. Primero miras la identidad del modelo. Después los metadatos. Después los archivos. Después cómo se ejecuta. Y solo al final miras los benchmarks.

La idea no es aprender un ritual de botones. Es saber qué pregunta de ingeniería hay detrás de cada etiqueta.

ZONA DE LA PÁGINA	QUÉ ESTÁS MIRANDO	PREGUNTA QUE DEBES HACER
owner/model	Organización y nombre exacto del repositorio.	¿Estoy mirando el modelo oficial, una copia, un fine-tune o una cuantización?
Tarea visible	Etiqueta como <code>Text Generation</code> , <code>Image-Text-to-Text</code> o <code>Feature Extraction</code> .	¿La tarea coincide con mi caso o estoy forzando el modelo?
Biblioteca	<code>Transformers</code> , <code>Diffusers</code> , <code>Sentence Transformers</code> , <code>timm</code> u otra.	¿Con qué librería se espera cargar o servir?
Formato	<code>Safetensors</code> , <code>GGUF</code> , <code>ONNX</code> , <code>PyTorch</code> , <code>TensorRT</code> u otros.	¿Es el formato que mi runtime puede abrir?
Licencia	MIT, Apache-2.0, llama, custom, research-only u otra.	¿Puedo usarlo, modificarlo, servirlo o redistribuirlo en mi contexto?
Tags	Palabras como <code>conversational</code> , <code>fp8</code> , <code>eval results</code> , <code>long-context</code> .	¿Son metadatos útiles o solo señales que debo comprobar?
Downloads y likes	Popularidad y uso reciente.	¿Hay adopción o solo ruido? Nunca es una prueba de calidad.
Model tree	Relación con fine-tunes, adapters y cuantizaciones.	¿Estoy viendo el tronco principal o una rama derivada?
Files and versions	Archivos, commits, pesos, config, tokenizer, licencia e historial.	¿Puedo auditar qué estoy descargando y cuándo cambió?
Inference Providers	Empresas que lo sirven desde la nube.	¿La calidad y el coste vienen del modelo o del proveedor que lo sirve?
Spaces	Demos o aplicaciones que usan el modelo.	¿Es una demo útil o una evidencia técnica? Normalmente es lo primero.
Evaluation results	Resultados integrados desde <code>model-index</code> o evaluaciones enlazadas.	¿Qué métrica, dataset, configuración y fuente produjo ese número?

La parte superior suele incluir metadatos que Hugging Face usa para buscar, filtrar y mostrar modelos. Algunos aparecen escritos en YAML dentro del README, otros se infieren desde archivos como `config.json` o desde la integración de la librería.

TÉRMINO	TRADUCCIÓN PRÁCTICA	QUÉ NO DEBES ASUMIR
<code>pipeline_tag</code>	Tarea principal del modelo. Decide filtros, widget y parte de la experiencia de inferencia.	Que el modelo sea bueno en todas las tareas parecidas.
<code>library_name</code>	Librería esperada para usarlo.	Que otra librería lo cargue igual sin conversión.
<code>license</code>	Condiciones de uso declaradas.	Que todo lo derivado tenga automáticamente la misma licencia sin revisar.
<code>language</code>	Idiomas declarados o detectados.	Que haya evaluación seria en todos esos idiomas.
<code>datasets</code>	Datasets de entrenamiento o evaluación que el autor declara.	Que conozcas todo el corpus real de entrenamiento.
<code>base_model</code>	Modelo del que parte un fine-tune, adapter o destilación.	Que conserve exactamente las capacidades del modelo base.
<code>new_version</code>	Repositorio recomendado como versión posterior.	Que puedas migrar sin repetir evals.
<code>tags</code>	Señales de búsqueda: modalidad, precisión, dominio, familia, técnica.	Que sean una especificación formal.
<code>model-index</code>	Resultados de evaluación estructurados.	Que el benchmark represente tu producto.
<code>widget</code>	Ejemplo interactivo en la página.	Que el prompt del widget sea tu contrato de producción.
<code>extra gated_fields</code>	Campos que el usuario acepta antes de acceder a un modelo restringido.	Que aceptar la pantalla baste para resolver privacidad o permisos internos.

Ahora leamos un caso real. El 25 de mayo de 2026, la card de `deepseek-ai/DeepSeek-V4-Pro` aparece en Hugging Face como modelo de generación de texto, con etiquetas de `Transformers`, `Safetensors`, `deepseek_v4`, `conversational`, resultados de evaluación y precisión `fp8`; declara licencia MIT; y describe DeepSeek-V4-Pro como un modelo MoE de 1,6T parámetros totales, 49B parámetros activados y contexto de 1M tokens.²¹

Lo importante no es memorizar esos números. Lo importante es saber leerlos. Para no convertirlo en una pared, vamos término a término, con el ejemplo de DeepSeek-V4-Pro como caso de lectura. Cuando una ficha técnica menciona DeepSeek-V4, también conviene contrastar con la documentación de Transformers, porque ahí aparecen detalles de arquitectura como tipos de atención, `max_position_embeddings` y clases de carga.²²

La regla de esta sección es: **ningún término se queda en definición de diccionario**. Para cada dato preguntamos qué mide, qué recurso toca, con qué se compara, qué sería razonable y qué prueba haría antes de creerlo.

Identidad, tarea y repositorio

Estos términos responden a una pregunta básica: “¿qué estoy mirando exactamente?”. Parece trivial, pero muchísimos errores empiezan por usar una variante distinta de la que se quería evaluar.

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
deepseek-ai/DeepSeek-V4-Pro	Es el identificador completo del repositorio: organización o usuario antes de / , nombre del modelo después. En Hugging Face no basta con decir “DeepSeek V4”; puede haber forks, quantizations, fine-tunes y mirrors.	deepseek-ai indica la organización; DeepSeek-V4-Pro indica el repo concreto.	Guarda este valor exacto en documentación, evals y configuración. Si pruebas Rooc/DeepSeek-V4-Pro , ya no estás probando el mismo repositorio.
Namespace	Es la parte izquierda del identificador. Puede ser una empresa, una comunidad, una persona o una organización académica.	deepseek-ai no es lo mismo que nvidia , unsloth , mlx-community o una cuenta personal.	Antes de descargar pesos, mira si el repo es oficial, derivado o una adaptación para otro formato.
Repository	Es la carpeta pública del modelo: README, pesos, tokenizer, configuración, licencia, historial y discusiones.	En un repo grande encontrarás README , archivos safetensors , configuración, tokenizer, licencia, scripts y carpetas auxiliares.	Trata el repo como expediente técnico, no como una tarjeta de marketing.
Text Generation	Es la tarea principal declarada. Significa que el modelo genera texto token a token a partir de un contexto. No significa automáticamente “buen chat”, “buen agente” o “buen programador”.	DeepSeek-V4-Pro se muestra como generación de texto.	Si quieres embeddings, clasificación o visión, esta etiqueta por sí sola no basta. Busca el pipeline_tag correcto y evalúa tu tarea.
conversational	Tag que sugiere uso conversacional. Normalmente indica que el modelo está pensado para turnos de usuario/asistente.	Puede aparecer junto a Text Generation .	Revisa la plantilla de chat. Un modelo conversacional mal formateado puede fallar por entrada, no por capacidad.
Evaluation Results	Señal de que Hugging Face puede mostrar resultados de evaluación asociados al modelo.	La card puede enseñar métricas como GSM8K, SWE-bench, GPQA o benchmarks de contexto.	Lee dataset, métrica, configuración y fuente. El número sin protocolo vale poco.

Ejemplo cercano: si en un proyecto interno dices “vamos a usar DeepSeek”, esa frase no basta. Una decisión trazable diría algo como: “probamos deepseek-ai/DeepSeek-V4-Pro , consultado el 25 de mayo de 2026, frente a una alternativa local cuantizada y una API comercial, con estos 80 casos de evaluación”.

Librería, formato y archivos

La siguiente capa responde a: “¿cómo se carga y qué estoy descargando?”. Aquí aparecen términos que parecen de infraestructura, pero deciden si el modelo se puede probar hoy o si necesitas una semana de entorno.

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
Transformers	Indica integración con la librería de Hugging Face para tokenizers, configuración y modelos. No garantiza que tu portátil pueda cargarlo ni que tu versión instalada tenga soporte completo.	La card enseña ejemplos con <code>AutoTokenizer</code> y <code>AutoModelForCausalLM</code> .	Comprueba versión de <code>transformers</code> , memoria, <code>trust_remote_code</code> si aplica y soporte de arquitectura.
Safetensors	Formato de pesos basado en tensores, diseñado para ser simple, rápido y evitar dependencias de carga como <code>pickle</code> . ²³	La zona lateral puede mostrar <code>Safetensors</code> .	Bien para distribuir pesos; no implica que el modelo quepa en GPU ni que tu runtime soporte todas sus capas.
Files and versions	Pestaña donde están los archivos reales y su historial. Es donde miras pesos, <code>config.json</code> , tokenizer, licencia, scripts y commits.	Si ves una carpeta <code>encoding</code> , no la ignores: puede explicar cómo convertir mensajes en texto para el modelo.	Para producción, fija commit o versión. No dependas solo del nombre del repo.
config.json	Archivo que describe arquitectura, dimensiones, vocabulario, contexto, tipos de capas y parámetros de inferencia.	Puede incluir <code>max_position_embeddings</code> o <code>layer_types</code> .	Si un número de la card no cuadra, abre la configuración antes de asumir que la card está mal o bien.
Tokenizer	Componente que parte texto en tokens y reconstruye texto desde tokens.	El mismo prompt puede convertirse en secuencias distintas según tokenizer.	No cambies tokenizer entre evals salvo que quieras medir otra cosa.
Tensor type	Tipos numéricos presentes en archivos: BF16, F32, FP8, enteros u otros.	La zona lateral puede listar varios tipos a la vez.	No leas “tensor type” como precisión única de inferencia. Puede mezclar pesos, escalas, índices y archivos auxiliares.
Licenses: mit	Licencia declarada para repo y pesos, según la card. MIT suele ser permisiva, pero debes leer el archivo de licencia.	DeepSeek-V4-Pro declara MIT en la card.	Comprueba <code>LICENSE</code> , condiciones internas de tu organización y si usas derivados con otra licencia.

Ejemplo cercano: si un compañero dice “está en Safetensors, lo cargamos fácil”, la respuesta de ingeniería es: “formato de archivo sí; ahora dime tamaño, precisión, runtime, memoria, tokenizer, plantilla y licencia”.

Tamaño, precisión, contexto y memoria

Esta parte responde a: “¿cuánto pesa operar esto?”. Aquí se confunden mucho los términos porque parecen números comparables, pero no todos miden lo mismo. Un número útil debe decirte tres cosas: **qué recurso toca, contra qué lo comparas y qué decisión cambia.**

La cuenta mínima que debe tener un ingeniero en la cabeza es esta:

$$M_{\text{pesos}} \approx N_{\text{parametros}} \cdot \frac{b}{8}$$

SÍMBOL O	QUÉ SIGNIFICA	EJEMPLO
M_{pesos}	Memoria aproximada solo de pesos.	No incluye KV cache, activaciones, runtime ni fragmentación.
$N_{\text{parametros}}$	Número de parámetros almacenados.	7B, 70B, 1.6T.
b	Bits por parámetro.	F32 usa 32; BF16/F16 usa 16; FP8/I8 usa 8; FP4/I4 usa 4.

Regla de bolsillo: **1B parámetros ocupa unos 4 GB en F32, 2 GB en BF16/F16, 1 GB en FP8/INT8 y 0,5 GB en FP4/INT4**, antes de sobrecostes. En producción añade memoria para KV cache, buffers, escalas de cuantización, runtime y margen de seguridad. Por eso “70B en 4-bit son 35 GB” es solo el principio de la conversación, no el dimensionamiento completo.

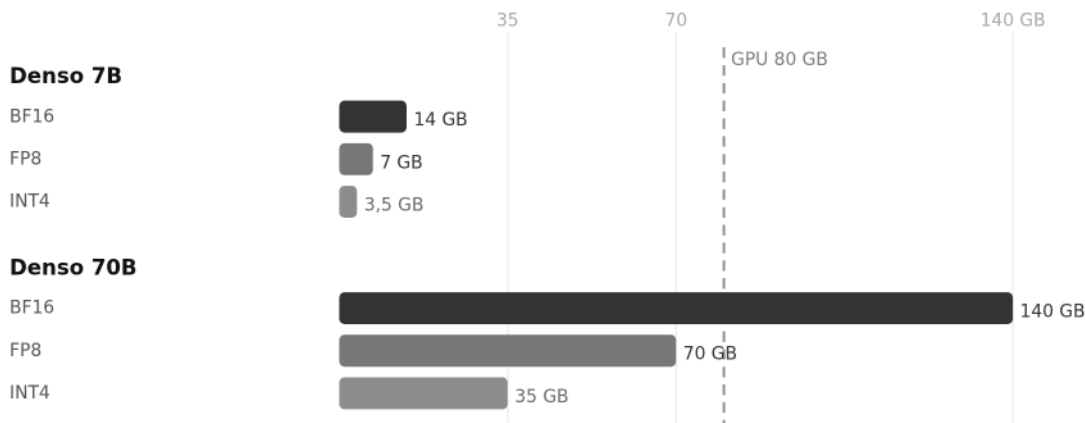
Hagamos la cuenta de verdad para un modelo denso de 70B. Con b bits por parámetro, la memoria solo de pesos es $M = 70 \cdot 10^9 \cdot b/8$:

- BF16 (16 bits): $70 \cdot 10^9 \cdot 16/8 = 140 \cdot 10^9$ bytes = **140 GB**.
- FP8/INT8 (8 bits): $70 \cdot 10^9 \cdot 8/8 = 70$ GB.
- INT4/FP4 (4 bits): $70 \cdot 10^9 \cdot 4/8 = 35$ GB.

La lectura es inmediata: una GPU de 80 GB no sirve ese 70B en BF16 (no cabe), lo sirve justo en FP8 (sin apenas margen para KV cache) y lo sirve con holgura en INT4. Esa cuenta de servilleta decide, antes de cualquier benchmark, en qué hardware puede vivir el modelo.

El mismo modelo encoge con la precisión

Memoria solo de pesos, $M = N \cdot b/8$ (antes de KV cache, buffers y runtime)



Cada salto de precisión parte la memoria a la mitad (16→8→4 bits).
El 70B no cabe en 80 GB en BF16; en FP8 entra justo; en INT4, con holgura.

IA para gente curiosa / Facsímil 04 / Capítulo 04 / 686f6c61

COMPARACIÓN RÁPIDA	BF16/F16	FP8/INT8	FP4/INT4	LECTURA DE INGENIERÍA
Modelo denso 7B	~14 GB	~7 GB	~3,5 GB	En local, 4-bit suele ser el punto de entrada; BF16 pide GPU más holgada.
Modelo denso 70B	~140 GB	~70 GB	~35 GB	Normalmente necesitas varias GPUs, servidor grande o cuantización fuerte.
MoE 1.6T con 49B activados	Pesos totales enormes	Menos memoria por peso	Menos memoria por peso	El cómputo por token se parece más a los activados, pero tienes que almacenar y servir el total o repartirlo.

Lo “adecuado” no es universal. Para entrenamiento o referencia científica, BF16/F16 suele ser base razonable; F32 queda para partes sensibles, depuración o cálculos concretos. Para inferencia de producción en hardware moderno, FP8/INT8 puede ser un buen compromiso si el runtime lo soporta. Para local, demos y coste bajo, INT4/FP4/GGUF puede ser aceptable, pero solo después de eval propia. NVIDIA documenta el uso de BF16, FP8 y formatos más bajos como parte de entrenamiento e inferencia de baja precisión; Hugging Face y vLLM mantienen documentación específica para cuantización en inferencia.²⁴

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
1.6T total parameters	Mide cuántos parámetros hay almacenados. Aporta una idea de memoria total, descarga, reparto entre GPUs y complejidad operativa. En MoE no equivale al cómputo por token.	Si fueran 1,6T en BF16, solo pesos serían ~3,2 TB; con FP8/FP4 mixto baja, pero sigue siendo infraestructura seria.	Compáralo con modelos densos 7B/70B. Si no tienes plan de serving distribuido o provider, este número ya te dice que no es “modelo local normal”.
49B activated parameters	Mide la parte aproximada que participa por token. Aporta una pista de coste de cómputo, no de memoria total.	49B activados se parece más a servir un modelo grande denso por token, pero con pesos totales mucho mayores detrás.	Úsalo para estimar latencia y throughput, pero no para decidir memoria de GPU. Memoria y cómputo son dos columnas distintas.
Model size	Estimación que muestra el Hub. Aporta una lectura rápida, pero puede mezclar detección automática, ficheros publicados y metadatos.	La UI puede resumir algo que el README detalla de otra manera.	Si hay discrepancia, abre <code>config.json</code> , README y archivos. El valor adecuado es el que puedes reproducir desde el repo.
1M context length	Mide ventana máxima teórica. Aporta capacidad para meter muchos tokens, pero también presión sobre KV cache, latencia y calidad de recuperación.	DeepSeek-V4-Pro declara contexto de 1M tokens.	Adecuado solo si tu tarea necesita contexto largo y lo evalúas. Para muchos productos, RAG con buenos fragmentos gana a meter todo.
max_position_embeddings	Mide posiciones máximas soportadas por la arquitectura/configuración. Aporta el límite estructural, no la promesa de experiencia barata.	En DeepSeek-V4 la documentación de Transformers menciona <code>1048576</code> , aproximadamente 1M posiciones.	Compáralo con tu longitud real: si tus casos tienen 8K-32K tokens, 1M quizá no aporta nada salvo coste.
FP4 + FP8 Mixed	Indica precisión mixta: algunas partes usan 4 bits y otras 8 bits. Aporta reducción de memoria/ancho de banda manteniendo más precisión donde conviene.	DeepSeek-V4-Pro declara expertos MoE en FP4 y la mayoría del resto en FP8.	Adecuado cuando el modelo fue entrenado/publicado para ese formato y tu runtime lo soporta. No lo equipares a una cuantización casera hecha después.
BF16	Flotante de 16 bits con rango amplio parecido a F32 y menos precisión fina. Aporta buena estabilidad con mitad de memoria que F32.	7B en BF16 son ~14 GB solo en pesos; 70B son ~140 GB.	Buen baseline de calidad para inferencia seria si tienes memoria. Si BF16 no cabe, cuantizas; si cabe, úsalo como referencia de comparación.
F32	Flotante de 32 bits. Aporta máxima comodidad numérica,	7B en F32 son ~28 GB solo pesos; 70B son	No suele ser adecuado para servir LLM grandes completos.

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
	pero cuesta el doble que BF16 y cuatro veces más que FP8/INT8 en memoria.	~280 GB.	Útil para partes sensibles, depuración, entrenamiento clásico o modelos pequeños.
FP8	Flotante de 8 bits. Aporta ahorro de memoria y ancho de banda manteniendo comportamiento mejor que muchos enteros si está bien calibrado y soportado.	70B en FP8 son ~70 GB de pesos, antes de sobrecostes.	Adecuado en GPUs/runtimes modernos con soporte real. Evalúa porque FP8 no garantiza calidad: depende de escalas, kernels y arquitectura.
I8 / INT8	Entero de 8 bits. Aporta compresión y aceleración si el runtime tiene kernels adecuados. Suele necesitar escalas para reconstruir valores.	Un modelo de 70B en INT8 ronda ~70 GB de pesos más escalas y sobrecostes.	Adecuado para inferencia cuando la degradación medida es pequeña. No asumas que todo INT8 conserva igual matemáticas, código o formato JSON.
FP4 / INT4	4 bits por peso. Aporta gran reducción de memoria, a costa de mayor riesgo de pérdida de calidad.	Un 70B en 4-bit ronda ~35 GB de pesos más sobrecostes.	Adecuado para local, coste bajo o prototipos cuando la eval propia lo confirma. Para extracción exacta, SQL, código o razonamiento duro, compara contra BF16/FP8.
Tensor type	Lista tipos numéricos presentes en archivos. Aporta pistas, pero no dice por sí sola la precisión principal de inferencia.	Puede aparecer BF16, F32, FP8, I64 o I8 porque hay pesos, escalas, índices y metadatos.	No tomes el primer tipo como “el modelo corre en eso”. Pregunta: qué tensors son pesos, cuáles escalas, cuáles índices y qué usa el runtime.
Cuantización	Técnica para representar pesos o activaciones con menos bits. Aporta reducción de memoria y coste, pero puede cambiar calidad, velocidad y compatibilidad.	Puede aparecer como GGUF, GPTQ, AWQ, bitsandbytes, FP8, INT8, INT4 o repo derivado.	Lo adecuado depende de tu restricción: BF16 para referencia, FP8/INT8 para producción eficiente, 4-bit para local/coste bajo si pasa evals.

Ejemplo cercano: si tienes un asistente que responde a documentos de 40 páginas, un contexto de 1M tokens quizá no es la primera solución. Puede ser mejor recuperar 10 fragmentos bien citados con RAG, como veremos en capítulos 09 y 10.

Arquitectura: MoE, atención y conexiones internas

Estos términos explican cómo está construido el modelo. No siempre necesitas dominarlos para usar una API, pero sí para entender por qué un modelo tiene ciertas necesidades de runtime.

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
MoE	Mixture of Experts. El modelo tiene varios expertos y un mecanismo de enrutamiento decide cuáles se usan para cada token.	DeepSeek-V4-Pro es MoE: total enorme, activación parcial.	Necesitas servir expertos, routing y paralelismo con cuidado. Un runtime pobre puede arruinar la ventaja.
Expertos	Subredes especializadas dentro de un MoE. No son “personas”; son bloques de parámetros.	Un token puede activar ciertos expertos y no otros.	En inferencia distribuida importa dónde vive cada experto y cuánto tráfico genera.
Routing	Decisión interna de qué expertos se activan.	Cada token se enruta a una parte del modelo.	Puede afectar latencia, balanceo de carga y reproducibilidad de rendimiento.
CSA	Compressed Sparse Attention. Atención comprimida y dispersa para manejar contexto largo de forma más eficiente.	DeepSeek-V4 menciona CSA como parte de su atención híbrida.	No basta con leer “1M tokens”; mide si recupera bien información lejana en tu tarea.
HCA	Heavily Compressed Attention. Otra rama de atención comprimida orientada a señales de largo alcance.	La documentación describe capas HCA y CSA intercaladas.	Útil para contexto largo, pero exige pruebas de latencia, memoria y calidad.
mHC	Manifold-Constrained Hyper-Connections. Conexiones internas que sustituyen o refuerzan conexiones residuales tradicionales.	Aparece como cambio arquitectónico de DeepSeek-V4.	Interesa para entender estabilidad y diseño, pero no decide por sí solo si te sirve.
Sliding attention	Atención en ventana local. Mira solo un tramo cercano del contexto.	En algunos bloques se usa una ventana local.	Buena para eficiencia local; no sustituye por sí sola recuperación global.
KV cache	Memoria de claves y valores de atención durante generación. Ya la vimos en capítulo 03 .	Contexto largo puede disparar KV cache si no hay compresión.	Para producto, KV cache es coste real: GPU, batch, latencia y throughput.

Ejemplo cercano: imagina una biblioteca. Un modelo denso abre todas las salas para cada consulta. Un MoE intenta abrir solo algunas salas especializadas. Eso ahorra trabajo por consulta, pero obliga a tener un edificio enorme disponible y un sistema de pasillos muy bien organizado.

Entrenamiento, ajuste y modos de razonamiento

Esta capa responde a: “¿qué se hizo para que el modelo se comporte así?”. Son términos de entrenamiento, no botones de producto.

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
Pretraining	Entrenamiento base sobre grandes cantidades de texto, código u otros datos. Aprende patrones generales.	La familia V4 declara preentrenamiento a gran escala antes del post-training.	No esperes que pretraining conozca tus documentos privados. Para eso entra RAG o fine-tuning.
Post-training	Fase posterior para ajustar instrucciones, formato, razonamiento, preferencias o herramientas.	La card habla de un pipeline posterior al pretraining.	Afecta cómo responde, no solo cuánto sabe. Evalúa estilo, obediencia de formato y consistencia.
SFT	Supervised fine-tuning: ajuste con pares entrada-salida. Enseña formato y comportamiento deseado.	“Si el usuario pregunta X, responde con Y” en muchos ejemplos.	Muy útil para tono y patrón de respuesta; no es buena vía para información que cambia cada día.
RL	Reinforcement learning. El modelo mejora usando señales de recompensa sobre sus respuestas.	Se usa en modelos de razonamiento para reforzar soluciones mejores.	Puede mejorar resolución, pero debes medir longitud, coste, formato y estabilidad.
GRPO	Group Relative Policy Optimization, variante usada en trabajos de DeepSeekMath para mejorar razonamiento con menor coste de memoria que PPO. ²⁵	DeepSeek-V4 menciona RL con GRPO en su post-training.	Interpreta “GRPO” como pista de entrenamiento, no como garantía de que tu problema saldrá bien.
on-policy distillation	Destilación usando salidas generadas bajo la propia política del sistema durante el proceso de mejora.	La card lo describe como consolidación de capacidades.	Es relevante para entender la receta; tu aplicación sigue necesitando eval propia.
Muon optimizer	Optimizador usado durante entrenamiento para estabilidad o convergencia.	DeepSeek-V4 lo menciona como parte de sus mejoras.	No cambia tu llamada API. Sirve para leer el informe técnico, no para configurar un chatbot.
Non-think	Modo de respuesta más directa, sin gran presupuesto de razonamiento.	Útil para tareas rutinarias.	Si la tarea es simple, evita pagar latencia extra por razonamiento largo.
Think	Modo con más análisis interno y respuesta más cuidadosa.	Útil para planificación, código o problemas con varias restricciones.	Mide coste y tiempo. No lo actives por defecto en todo.
Think Max	Modo de esfuerzo máximo.	Pensado para problemas difíciles o	Reserva para casos donde el coste adicional

TÉRMI NO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4- PRO	DECISIÓN PRÁCTICA
		evaluación de frontera.	compense.

Ejemplo cercano: para clasificar tickets de soporte, `Non-think` puede bastar. Para revisar una migración de base de datos, quizá `Think` tenga sentido. Para explorar una demostración matemática o un problema de programación complejo, `Think Max` puede ser una prueba, no necesariamente el modo de producción.

Plantilla de chat, encoding y parámetros de generación

Aquí aparece una de las causas más tontas y más caras de errores: usar bien el modelo, pero formatear mal la entrada. Hugging Face documenta las chat templates como la forma de convertir mensajes en el formato exacto que el modelo vio durante entrenamiento.²⁶

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
Chat Template	Plantilla que convierte mensajes <code>system</code> , <code>user</code> y <code>assistant</code> en texto/tokens con separadores especiales.	Algunas familias usan Jinja; DeepSeek-V4-Pro indica una carpeta <code>encoding</code> en lugar de una plantilla Jinja clásica.	Si ignoras la plantilla, puedes convertir un buen modelo en un mal modelo.
<code>encoding folder</code>	Carpeta con scripts para codificar mensajes y parsear salidas.	Puede incluir funciones tipo “mensajes a string” y “texto generado a respuesta”.	Señal de contrato de entrada. No improvises el prompt final sin leerla.
Roles	Estructura de conversación: sistema, usuario, asistente y a veces herramientas.	Un mensaje de sistema puede fijar comportamiento; uno de usuario contiene la tarea.	Mantén roles consistentes entre eval y producción.
<code>temperature</code>	Controla aleatoriedad de muestreo. Valores bajos suelen ser más conservadores; altos, más variados.	La card puede recomendar <code>temperature = 1.0</code> .	Para extracción JSON o clasificación, baja variación. Para ideación, quizá más variación.
<code>top_p</code>	Muestreo por núcleo: limita candidatos a una masa de probabilidad acumulada.	La card puede recomendar <code>top_p = 1.0</code> .	No cambies <code>temperature</code> y <code>top_p</code> a ciegas; registra configuración en tus evals.
<code>max_new_tokens</code>	Máximo de tokens que puede generar la respuesta.	Si pides resumen largo con límite bajo, cortará la salida.	Reserva salida suficiente. Contexto total no es solo entrada.
Stop tokens	Secuencias que detienen generación.	Un token especial puede cerrar un bloque de razonamiento o respuesta.	Útiles para formato; peligrosos si cortan respuestas válidas.

Ejemplo cercano: dos equipos pueden “usar el mismo modelo” y obtener resultados distintos si uno aplica bien la plantilla de chat y otro manda un string plano. La diferencia no está en inteligencia; está en protocolo.

Runtimes, proveedores y despliegue

Esta parte responde a: “¿dónde corre y con qué contrato?”. El modelo no vive en el aire: necesita runtime, hardware, límites, observabilidad y presupuesto. vLLM y SGLang, por ejemplo, exponen servidores compatibles con APIs tipo OpenAI para servir modelos grandes.²⁷

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
vLLM	Runtime de inferencia orientado a throughput, batch, KV cache y API compatible con OpenAI.	La card puede enseñar cómo servir el modelo con <code>vllm serve</code> .	Mide p50, p95, tokens por segundo, memoria y límites de contexto.
SGLang	Framework de serving para modelos de lenguaje y multimodales, con foco en baja latencia, throughput, cache y paralelismo.	La card puede dar comandos <code>sglang.launch_server</code> .	Útil si necesitas exprimir serving, paralelismo o flujos estructurados.
Docker Model Runner	Forma de ejecutar modelos desde Docker usando comandos familiares y un endpoint gestionado por Model Runner. ²⁸	La card puede mostrar <code>docker model run hf.co/...</code> .	Muy cómodo para probar; en producción revisa memoria, logs, versiones y despliegue real.
Inference Providers	Proveedores integrados en Hugging Face que sirven modelos alojados. ²⁹	Puede aparecer Novita u otros proveedores para un modelo.	Compara proveedor concreto, región, precio, latencia, privacidad y versión servida.
OpenAI-compatible API	API que imita contratos de OpenAI, normalmente <code>/v1/chat/completions</code> u otros endpoints parecidos.	vLLM y SGLang pueden exponer endpoints compatibles.	Compatible no significa idéntico: revisa streaming, tools, JSON, errores y parámetros soportados.
Throughput	Cantidad de tokens o peticiones procesadas por unidad de tiempo.	Un modelo puede ser bueno para batch y peor para chat interactivo.	Si tienes muchos usuarios, throughput importa tanto como calidad.
Latencia p95	Tiempo por debajo del cual acaba el 95 por ciento de peticiones.	Una demo rápida puede esconder colas lentas.	Decide con p95, no solo con “a mí me respondió rápido”.
Batch	Agrupar peticiones para aprovechar hardware.	Muy útil en procesos nocturnos o clasificación masiva.	Puede mejorar coste, pero empeorar tiempo de espera interactivo.
Observabilidad	Logs, métricas, trazas, errores, coste y calidad en producción.	Necesitas saber cuándo falla, cuánto cuesta y qué versión respondió.	Sin observabilidad, elegir modelo es solo el inicio de una caja negra.

Ejemplo cercano: “lo sirve un provider” no responde a “¿me sirve a mí?”. Si tu aplicación trata documentos internos, necesitas saber retención, región, contrato, trazas, límites y cómo se comporta el proveedor cuando hay picos.

Evaluación y resultados

Las cards enseñan números porque necesitamos señales, pero cada número tiene una historia. Un benchmark sin protocolo es como una nota sin examen.

TÉRMINO EN LA CARD	EXPLICACIÓN COMPLETA	EJEMPLO CON DEEPSEEK-V4-PRO	DECISIÓN PRÁCTICA
Evaluation Results	Bloque de resultados visibles en el Hub o enlazados desde la ficha.	Puede mostrar GSM8K, SWE-bench, GPQA, Terminal Bench u otros.	Úsalo para orientar, nunca para decidir sin eval propia.
Benchmark	Prueba estandarizada sobre un conjunto de tareas.	HumanEval mide programación; MMLU conocimiento general; GPQA razonamiento científico.	Pregunta si se parece a tu caso real.
Dataset	Conjunto de datos usado para medir.	GSM8K contiene problemas matemáticos de primaria/secundaria.	Si tu tarea es legal, médica o administrativa, GSM8K no te la resuelve.
Metric	Regla que convierte respuestas en puntuación.	Pass@1 , EM , ACC , F1 .	La métrica define qué cuenta como acierto. Léela antes de comparar.
Shots	Número de ejemplos que se dan en el prompt.	0-shot , 3-shot , 5-shot , 25-shot .	Más ejemplos consumen tokens y pueden cambiar mucho el resultado.
Harness	Código y protocolo que ejecutan la evaluación.	Puede controlar prompts, herramientas, timeouts y validación.	Dos resultados en el mismo benchmark pueden no ser comparables si cambia el harness.
tools enabled	La evaluación permite usar herramientas externas, como terminal, navegador, ejecución de código o buscador.	Algunos benchmarks de agentes lo indican explícitamente.	No compares contra resultados sin herramientas como si fueran la misma prueba.
Source	Origen del resultado: autor del modelo, leaderboard externo, paper o tercero.	Hugging Face puede enlazar fuente de evaluación.	Prefiere resultados reproducibles o, como mínimo, con protocolo claro.

Ejemplo cercano: si un modelo saca muy buena nota en SWE-bench , eso no significa que clasifique bien incidencias de alumnos. Significa que merece pasar a tu lista corta para tareas de código, si tus restricciones de coste y privacidad encajan.

Hay un detalle muy de ingenieros: las cards grandes a veces muestran números que no parecen encajar a primera vista. En DeepSeek-V4-Pro puedes ver una tabla del README con parámetros totales, activados, contexto y precisión, y también una zona lateral generada por Hugging Face con “model size” y “tensor type”. Si algo no cuadra, no lo ignores: abre `Files and versions` , busca `config.json` , `README` , `LICENSE` , `tokenizer`, `scripts` de `encoding` y notas del repositorio. La card no es un oráculo; es la entrada al expediente.

Las métricas también tienen vocabulario propio. Conviene leerlas como leerías un contrato: una palabra pequeña cambia lo que realmente se está midiendo.

MÉTRICA	QUÉ MIDE APROXIMADAMENTE	EJEMPLO SENCILLO	CUIDADO
EM	Exact match: la respuesta generada debe coincidir exactamente con la esperada.	Esperado: Madrid . Generado: Madrid . Acierta. Generado: La respuesta es Madrid . Puede fallar si el evaluador es estricto.	Penaliza respuestas correctas con formato distinto. Es buena para respuestas cerradas, mala para explicaciones.
F1	Solapamiento entre partes de la respuesta esperada y generada. Se usa mucho cuando hay varias palabras relevantes.	Esperado: revisar matrícula y pago . Generado: revisar el pago de matrícula . Tiene bastante solapamiento.	Puede dar buena nota aunque falte un detalle crítico. En tareas sensibles, mira también errores cualitativos.
Pas@s@1	Si la primera solución generada pasa la prueba. Es común en código y problemas con verificador automático.	El modelo escribe una función; los tests se ejecutan una vez; si pasan, cuenta como acierto.	Depende muchísimo del harness, tests, timeout y formato esperado. No es "calidad general".
ACC	Accuracy: porcentaje de aciertos sobre el total.	Clasifica 100 tickets; acierta 87; ACC = 0.87 .	No dice qué clase falla. Si la clase rara es la importante, accuracy puede engañar.
MMR	Acrónimo que debes definir en el benchmark concreto. En recuperación suele ser Maximal Marginal Relevance; en algunas tablas puede aparecer con otro significado operacional.	En búsqueda semántica puede premiar resultados relevantes pero no repetidos.	Nunca asumas el significado por las siglas. Abre la ficha del benchmark.
Elo	Puntuación relativa por comparaciones entre modelos. Suele venir de duelos: respuesta A frente a respuesta B.	Un evaluador humano o automático prefiere una respuesta; el ranking se actualiza.	Depende de participantes, prompts, evaluador y protocolo. No es una unidad absoluta de inteligencia.
0-shot	El modelo responde sin ejemplos dentro del prompt.	"Clasifica este ticket" sin mostrar tickets anteriores resueltos.	Si tu producto sí usa ejemplos, este resultado quizá subestima tu caso.
few-shot	El prompt incluye algunos ejemplos antes de la tarea.	Das 3 tickets con categoría correcta y luego pides clasificar uno nuevo.	Puede mejorar mucho, pero consume contexto y puede sobreajustarse al formato de los ejemplos.
tool-enabled	La evaluación permite usar herramientas externas: ejecución de código, terminal, buscador, base de datos o navegación controlada.	Para resolver un bug, el sistema puede ejecutar tests en vez de responder solo de memoria.	No compares con un resultado sin herramientas. Es otro sistema, no solo otro modelo.

Mi lectura práctica de una card de Hugging Face siempre termina con siete preguntas:

PREGUNTA	DÓNDE BUSCARLA
¿Qué modelo exacto es?	Nombre del repo, organización, commits y versiones.
¿Qué tarea dice resolver?	<code>pipeline_tag</code> , tags, README y ejemplos.
¿Cómo se ejecuta bien?	<code>Use this model</code> , runtime, chat template, tokenizer y scripts.
¿Qué coste operativo tendrá?	Tamaño, precisión, contexto, parámetros activados, runtime y proveedores.
¿Qué evidencia trae?	<code>model-index</code> , tablas de evaluación, paper y fuente del benchmark.
¿Qué condiciones tiene?	Licencia, privacidad del proveedor, gating y restricciones internas.
¿Qué no me está diciendo?	Datos de entrenamiento, idiomas evaluados, fallos conocidos, prompts exactos y límites reales.

Si una card no responde a varias de estas preguntas, no significa que el modelo sea malo. Significa que tu decisión tiene más incertidumbre. Y la incertidumbre se compensa con pruebas propias, límites claros y una alternativa preparada.

Benchmarks: útiles, pero no soberanos

Los benchmarks son necesarios porque evitan discutir solo con impresiones. Pero no todos los benchmarks sirven para todas las decisiones. HELM nació precisamente para evaluar modelos de lenguaje de forma más holística: no solo exactitud, también escenarios, métricas y transparencia de resultados.³⁰ Esa idea es más importante que cualquier posición concreta en una tabla.

Hay una razón de fondo por la que un benchmark alto no basta, y tiene nombre: la **ley de Goodhart**, que popularizada dice «cuando una métrica se convierte en objetivo, deja de ser buena métrica».³¹ Si toda la industria optimiza para los mismos exámenes, los modelos aprenden a aprobar esos exámenes, no necesariamente a resolver tu tarea. A esto se suma la **contaminación de datos**: si las preguntas del benchmark estuvieron, aun sin querer, en el corpus de entrenamiento, la nota mide memoria, no capacidad; por eso la literatura reciente insiste en medir la contaminación benchmark a benchmark antes de creerse una cifra.³² La defensa no es despreciar los benchmarks, sino añadir una evaluación propia con casos que el modelo no pudo ver.

Tres preguntas ayudan:

PREGUNTA	POR QUÉ IMPORTA
¿Qué tarea mide?	Matemáticas, código, lectura, conversación, visión, SQL o seguridad no son lo mismo.
¿Cómo se evaluó?	Prompt, few-shot, temperatura, herramientas, idioma y versión pueden cambiar resultados.
¿Qué coste tuvo acertar?	Un modelo puede ganar usando más tokens, más tiempo o más cómputo de inferencia.

El capítulo anterior nos da el antídoto: mide tokens, coste y latencia además de calidad. Una respuesta que mejora un 2 % en exactitud pero triplica coste y p95 quizá no es mejor para tu producto.

Y hay otra capa: benchmark de modelo no es benchmark de sistema. MLPerf Inference, por ejemplo, mide sistemas completos bajo escenarios definidos. En aplicaciones con IA, el sistema incluye modelo, runtime, hardware, batch, cache, red, RAG, herramientas, validadores y observabilidad. Si solo miras el modelo, te faltan piezas.

Coste total: no solo precio por millón de tokens

El precio público es una parte, no toda la decisión. En una API pagas tokens, modalidades, cache, batch o prioridad según proveedor. En local pagas GPUs, electricidad, memoria, mantenimiento, actualización y tiempo del equipo. En ambos casos pagas también integración, evaluación y cambios de versión.

Esto no es una metáfora: es el **coste total de propiedad** (TCO), un marco de análisis de compras con literatura propia que insiste en que el precio de adquisición suele ser la parte pequeña frente a los costes de uso, operación y cambio a lo largo de la vida del sistema.³³ Aplicado a servir un modelo, el TCO mensual descompone así:

$$TCO = C_{\text{tokens}} + C_{\text{infra}} + C_{\text{operacion}} + C_{\text{cambio}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
TCO	Coste total de propiedad.	Coste mensual real de servir una función.
C_{tokens}	Coste de entrada, salida, cache y batch.	Factura del proveedor.
C_{infra}	Infraestructura propia o gestionada.	GPU, CPU, memoria, almacenamiento, red.
$C_{operacion}$	Monitorización, fallos, evals y soporte.	Tiempo del equipo y alertas.
C_{cambio}	Migración entre versiones o proveedores.	Adaptar prompts, schemas y evals.

Este coste cambia según el patrón de uso. Un asistente interactivo necesita p95 bajo. Un proceso nocturno puede aceptar batch. Una tarea con normativa fija puede aprovechar caché. Una tarea con documentos privados puede empujar hacia local o hacia una API con garantías contractuales concretas.

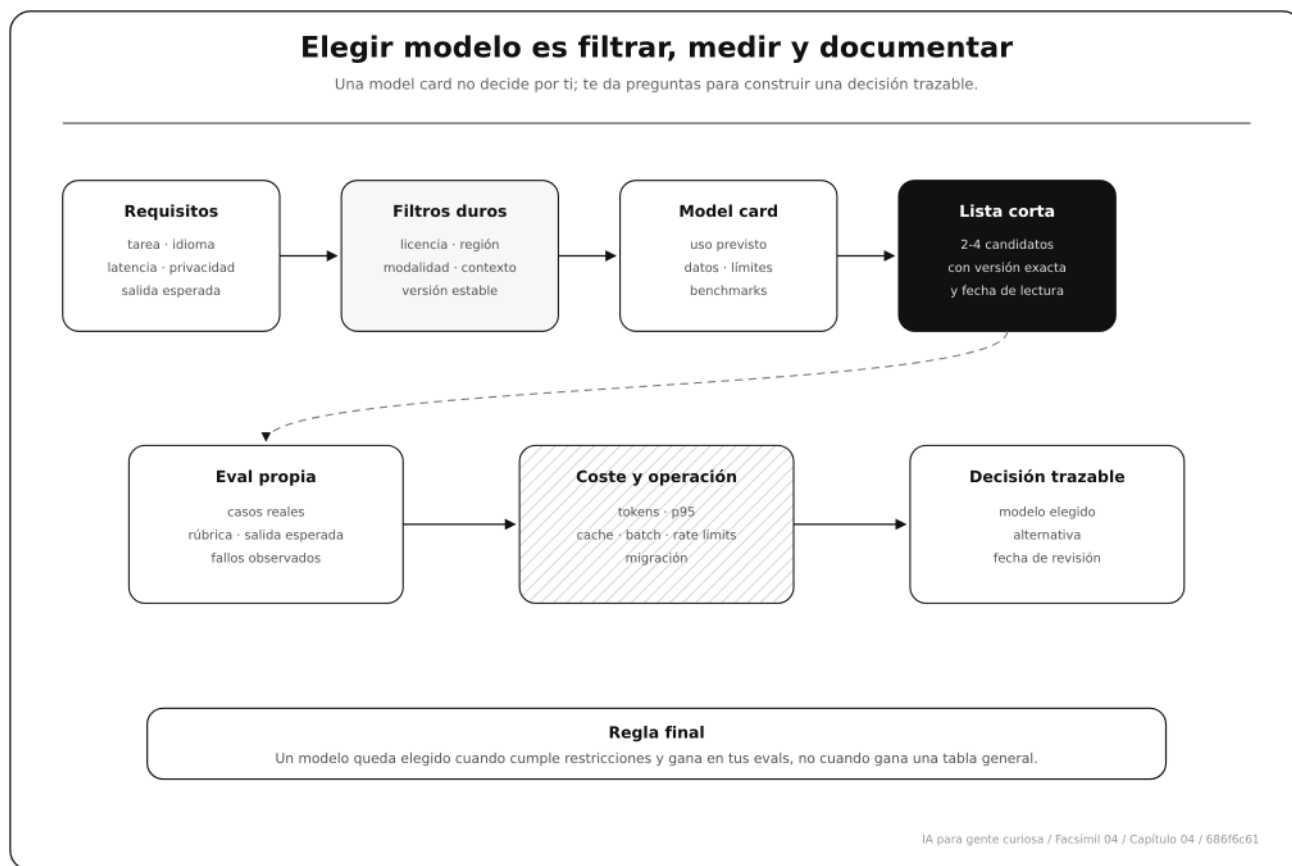
Para entenderlo: tres elecciones distintas

Pensemos en situaciones concretas:

CASO	MODELO TENTADOR	DECISIÓN MÁS SENSATA
Chat de orientación universitaria	El modelo más capaz disponible.	Modelo fiable, barato, con RAG, citas y buena evaluación en castellano.
Clasificador de tickets internos	Un LLM grande generalista.	Modelo menor con salida estructurada, eval propia y batch si no es interactivo.
Análisis de contratos extensos	El modelo con más contexto.	Contexto largo si aporta valor; si no, RAG con citas y control de fragmentos.
Generación de SQL	Modelo de código muy alto en benchmark.	Eval con tu esquema, permisos, consultas esperadas y validación antes de ejecutar.
Asistente local para datos sensibles	API más cómoda.	Revisar privacidad, modelo local, cuantización y coste operativo real.

La pregunta no es “¿cuál es mejor?”. La pregunta es “¿qué falla si me equivoco?”. Si el fallo cuesta poco, puedes experimentar. Si el fallo rompe una decisión importante, necesitas más evaluación, trazas y límites.

Mapa visual de la decisión



En el día a día

En un proyecto real, este capítulo aparece cuando alguien dice: “probemos con el modelo más potente”. A veces tiene sentido. Muchas otras veces la decisión correcta es un modelo más barato, una salida estructurada, RAG mejor hecho, caché, batch o una evaluación más honesta.

El trabajo profesional no es enamorarse de una familia de modelos. Es mantener una lista corta con versión exacta, fecha de consulta, coste estimado, eval propia y plan de salida si el proveedor cambia una versión o retira un endpoint.

Un equipo maduro guarda, junto al prompt y el código, una pequeña ficha interna: modelo elegido, alternativas descartadas, motivo, dataset de evaluación, resultados, costes, límites y próxima revisión. Esa ficha evita discusiones circulares cuando tres meses después alguien pregunta por qué no usamos “el nuevo”.

Por qué debería importarte

Porque la elección de modelo decide coste, experiencia de usuario, privacidad, mantenimiento y calidad. Si eliges solo por capacidad máxima, puedes construir un sistema que funciona en demo y duele en producción. Si eliges solo por precio, puedes ahorrar justo en la parte que sostenía la calidad.

También importa para aprender. Leer model cards te entrena a pensar como ingeniero: cada número pide una pregunta, cada benchmark pide contexto y cada promesa pide verificación.

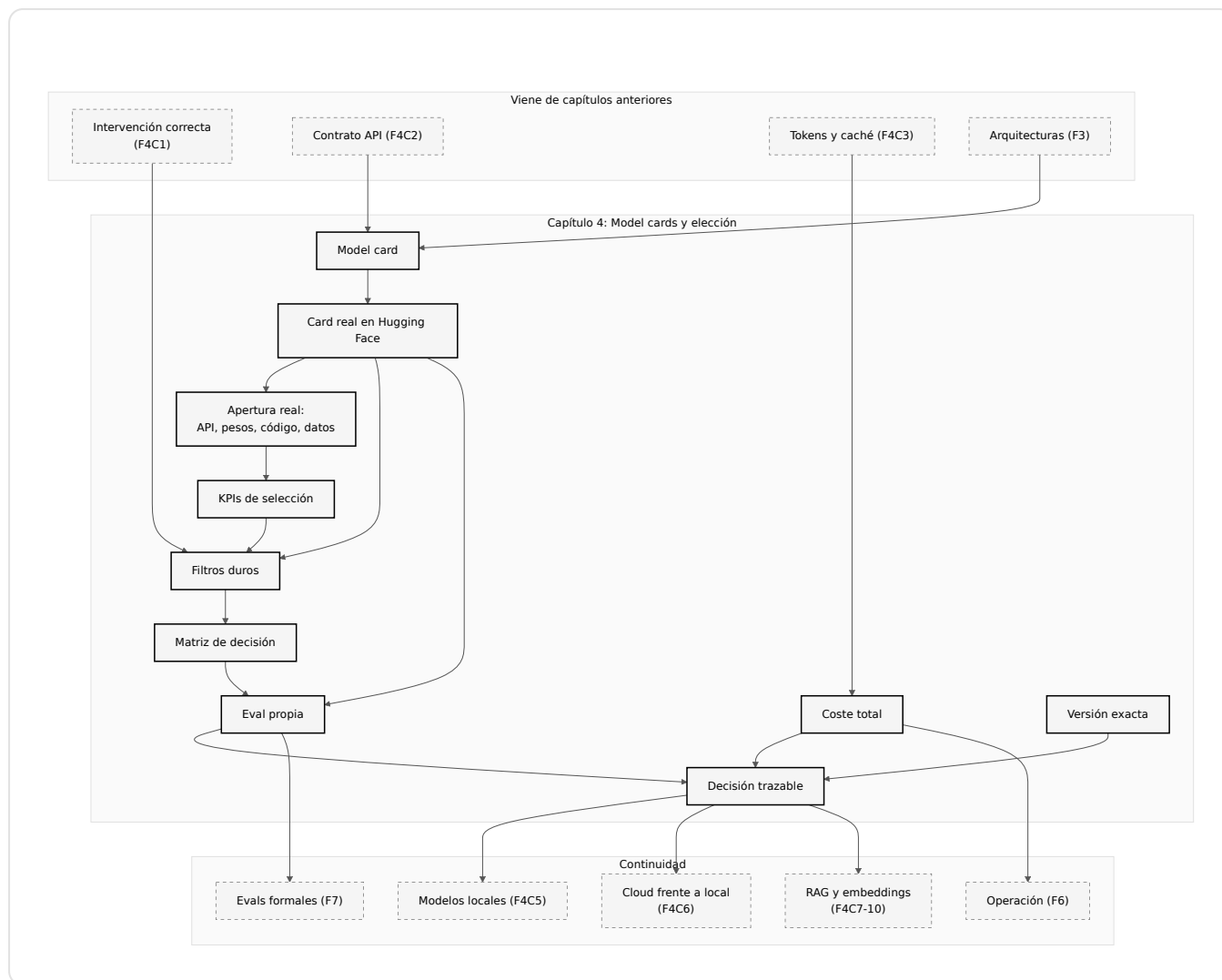
Dónde solía tropezar yo

Estos errores aparecen mucho cuando la conversación se queda en nombres de modelos.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Elegir por ranking general	Un benchmark amplio no mide tu flujo, tus datos ni tu coste.	Crear una eval propia pequeña antes de decidir.
No fijar versión exacta	Un alias puede cambiar y romper comparabilidad.	Guardar modelo, fecha, proveedor y configuración.
Comparar precio sin salida	Un modelo barato puede generar más tokens o fallar más.	Medir coste por tarea completada, no solo por millón de tokens.
Confundir contexto largo con calidad	Más ventana puede añadir ruido y latencia.	Medir qué fragmentos son realmente necesarios.
Olvidar licencia o privacidad	El modelo puede funcionar técnicamente y no encajar legalmente.	Revisar condiciones antes de hacer pruebas profundas.
No tener alternativa	Cuando cambia una versión, el producto queda atado.	Mantener segundo candidato y eval de regresión.

Cómo encaja todo

Este mapa conecta la elección de modelos con lo que ya vimos y con lo que viene en el facsímil.



Vocabulario aprendido

Estos términos convierten “me gusta este modelo” en una conversación técnica.

TÉRMINO	DEFINICIÓN
Model card	Ficha técnica que documenta uso previsto, datos, evaluación, límites y condiciones.
System card	Documento que describe un sistema completo, no solo el modelo aislado.
Benchmark	Prueba estandarizada bajo una metodología concreta.
Eval propia	Conjunto de casos representativos de tu proyecto.
Latencia p95	Tiempo que cubre el 95 por ciento de peticiones.
TCO	Coste total de propiedad: tokens, infraestructura, operación y cambios.
Modelo estable	Versión concreta pensada para producción.
Modelo preview	Versión de avance rápido que puede cambiar antes.
Matriz de decisión	Comparación ponderada de modelos según criterios explícitos.
pipeline_tag	Tarea principal declarada en Hugging Face.
Safetensors	Formato de pesos basado en tensores.
Parámetros activados	Parte del modelo MoE usada para cada token.
Chat template	Plantilla que convierte mensajes en tokens.
model-index	Metadatos de evaluación que Hugging Face puede mostrar de forma estructurada.
Cuantización	Uso de menos bits para reducir memoria y coste operativo.
Namespace	Organización o usuario propietario del repositorio en Hugging Face.
Tokenizer	Componente que convierte texto en tokens y tokens en texto.
Tensor type	Tipo numérico de los tensores publicados o auxiliares.
BF16	Formato de 16 bits usado como baseline de buena calidad cuando cabe en memoria.
F32	Formato de 32 bits, cómodo numéricamente pero caro para servir LLM grandes.
FP8 / INT8	Formatos de 8 bits para reducir memoria y ancho de banda con evaluación obligatoria.
FP4 / INT4	Formatos de 4 bits para local o coste bajo, con mayor riesgo de pérdida de calidad.

TÉRMINO	DEFINICIÓN
MoE	Arquitectura con expertos y enrutamiento por token.
Modelo cerrado	Modelo usado como servicio o producto sin acceso a pesos ni receta completa de entrenamiento.
Pesos abiertos	Pesos descargables o accesibles para servir, ajustar o cuantizar un modelo según licencia.
Código abierto	Código disponible bajo licencia abierta; en IA no implica automáticamente datos, pesos y entrenamiento abiertos.
Open Source AI	Sistema que ofrece libertades de uso, estudio, modificación y compartición junto con la forma preferida para modificarlo.
KPI de selección	Indicador que cambia una decisión de modelo: calidad propia, coste por tarea, p95, licencia, reproducibilidad o control de datos.
GRPO	Variante de optimización por refuerzo usada en trabajos de razonamiento.
temperature	Parámetro que controla variación en la generación.
top_p	Muestreo que limita candidatos por probabilidad acumulada.
Throughput	Capacidad de procesar tokens o peticiones por unidad de tiempo.
Pass@1	Métrica que cuenta si la primera solución pasa el verificador.
0-shot	Evaluación sin ejemplos dentro del prompt.
few-shot	Evaluación con ejemplos dentro del prompt.

Antes de pasar página

- ☐ ¿Puedo explicar por qué una model card no es marketing ni garantía? (Si no, vuelve a «Qué no es una model card».)
- ☐ ¿Sé separar modelo base, modelo servido por API, modelo local y sistema completo? (Si no, vuelve a «Modelo cerrado, pesos abiertos y código abierto no son lo mismo».)
- ☐ ¿Sé distinguir modelo cerrado, pesos abiertos, código abierto y Open Source AI? (Si no, vuelve a «Modelo cerrado, pesos abiertos y código abierto no son lo mismo».)
- ☐ ¿Puedo explicar qué son los pesos y por qué no equivalen a todo el proceso de entrenamiento? (Si no, vuelve a «Modelo cerrado, pesos abiertos y código abierto no son lo mismo».)

- ☐ ¿He definido KPIs de selección antes de discutir marcas de modelos? (Si no, vuelve a «Modelo cerrado, pesos abiertos y código abierto no son lo mismo».)
- ☐ ¿Puedo construir filtros duros antes de comparar puntuaciones? (Si no, vuelve a «La matriz mínima de decisión».)
- ☐ ¿Sé leer una model card de Hugging Face sin confundirme con tags, likes o downloads? (Si no, vuelve a «Anatomía de una model card en Hugging Face».)
- ☐ ¿Distingo parámetros totales, parámetros activados, precisión, formato y runtime? (Si no, vuelve a «Anatomía de una model card en Hugging Face».)
- ☐ ¿Sé por qué un benchmark general no sustituye una eval propia? (Si no, vuelve a «Benchmarks: útiles, pero no soberanos».)
- ☐ ¿Puedo calcular una puntuación ponderada y explicar sus pesos? (Si no, vuelve a «La matriz mínima de decisión».)
- ☐ ¿Distingo precio público de coste total de propiedad? (Si no, vuelve a «Coste total: no solo precio por millón de tokens».)
- ☐ ¿Sé qué datos debo guardar para que la decisión sea trazable? (Si no, vuelve a «La ficha que yo leería antes de elegir».)
- ☐ ¿Sé cómo cambia el ranking al ajustar los pesos de la matriz? (Si no, repasa «La matriz mínima de decisión».)

En resumen

Elegir modelo es una decisión de ingeniería, producto y operación. La model card no te da una respuesta automática, pero sí una lista de preguntas que evitan elegir por entusiasmo.

IDEA FUERZA	DETALLE
No existe “el mejor modelo” sin contexto.	Existe el modelo adecuado para una tarea, restricciones y evidencia.
La model card se lee como ficha de compatibilidad.	Uso previsto, límites, datos, evaluación, licencia y operación importan.
“Abierto” necesita apellido.	Modelo cerrado, pesos abiertos, licencia propia y Open Source AI no significan lo mismo.
Los pesos abiertos compran control, no magia.	Puedes servir, adaptar o cuantizar según licencia, pero quizá no tienes datos ni receta reproducible.
Los KPIs mandan sobre las etiquetas.	Calidad propia, p95, coste por tarea, licencia, privacidad y reproducibilidad pesan más que el eslogan.
Los benchmarks orientan, no deciden.	Tu eval propia decide si el modelo funciona en tu caso.
La versión exacta importa.	Alias, previews y modelos retirados pueden romper comparaciones.
El coste real no es solo precio por token.	Latencia, cache, batch, operación y migración entran en la cuenta.
La decisión debe quedar escrita.	Modelo elegido, alternativas, fecha y próxima revisión evitan memoria frágil.

Para saber más

Anthropic. (2026). *Models overview*. <https://platform.claude.com/docs/en/about-claude/models/overview>

Anthropic. (2026). *Pricing*. <https://platform.claude.com/docs/en/about-claude/pricing>

Gebu, T. y otros (2021). *Datasheets for Datasets*. <https://doi.org/10.1145/3458723>

DeepSeek-AI. (2025). *DeepSeek-R1*. <https://huggingface.co/deepseek-ai/DeepSeek-R1>

DeepSeek-AI. (2026). *deepseek-ai/DeepSeek-V4-Pro*. <https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro>

Docker. (2026). *docker model run*. <https://docs.docker.com/reference/cli/docker/model/run/>

Google. (2026). *Gemma 4 model overview*. <https://ai.google.dev/gemma/docs/core>

Google. (2026). *Gemini API: Models*. <https://ai.google.dev/gemini-api/docs/models>

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

Hugging Face. (2026). *Chat templates*.

https://huggingface.co/docs/transformers/chat_templates

Hugging Face. (2026). *DeepSeek-V4*.

https://huggingface.co/docs/transformers/model_doc/deepseek_v4

Hugging Face. (2026). *Inference Providers*. <https://huggingface.co/docs/inference-providers/en/index>

Hugging Face. (2026). *Model Cards*. <https://huggingface.co/docs/hub/model-cards>

Hugging Face. (2026). *Quantization*.

https://huggingface.co/docs/transformers/main_classes/quantization

Hugging Face. (2026). *Safetensors*. <https://huggingface.co/docs/safetensors/en/index>

Hoffmann, J. y otros (2022). *Training Compute-Optimal Large Language Models*.

<https://doi.org/10.48550/arXiv.2203.15556>

Kaplan, J. y otros (2020). *Scaling Laws for Neural Language Models*.

<https://doi.org/10.48550/arXiv.2001.08361>

Liang, P. y otros (2022). *Holistic Evaluation of Language Models*.

<https://arxiv.org/abs/2211.09110>

Mitchell, M. y otros (2019). *Model Cards for Model Reporting*.

<https://doi.org/10.1145/3287560.3287596>

Meta. (2025). *Llama 4 Community License Agreement*. <https://github.com/meta-llama/llama-models/blob/main/models/llama4/LICENSE>

Mistral AI. (2025). *Introducing Mistral 3*. <https://mistral.ai/news/mistral-3/>

MLCommons. (2026). *MLPerf Inference: Datacenter benchmark*.

<https://mlcommons.org/benchmarks/inference-datacenter/>

NVIDIA. (2026). *Transformer Engine: Low Precision Training*.

https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/features/low_precision_training/introduction/introduction.html

OpenAI. (2025). *Introducing gpt-oss*. <https://openai.com/index/introducing-gpt-oss/>

OpenAI. (2026). *Models*. <https://developers.openai.com/api/docs/models>

OpenAI. (2026). *OpenAI open-weight models (gpt-oss)*.

<https://help.openai.com/en/articles/11870455-openai-open-weight-models-gpt-oss>

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

Open Source Initiative. (2024). *The Open Source AI Definition 1.0*.
<https://opensource.org/ai/open-source-ai-definition>

Open Source Initiative. (2026). *Open Weights: not quite what you've been told*.
<https://opensource.org/ai/open-weights>

Qwen Team. (2026). *Qwen3.6*. <https://github.com/QwenLM/Qwen3.6>

SGLang. (2026). *Welcome to SGLang*. <https://docs.sglang.io/index.html>

Shao, Z. y otros (2024). *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. <https://arxiv.org/abs/2402.03300>

vLLM. (2026). *OpenAI-Compatible Server*.
https://docs.vllm.ai/en/latest/serving/openai_compatible_server/

vLLM. (2026). *Quantization*. <https://docs.vllm.ai/en/stable/features/quantization/>

Notas

1. Margaret Mitchell y otros (2019). *Model Cards for Model Reporting*. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 220-229.
<https://doi.org/10.1145/3287560.3287596>. ↵
2. Timnit Gebru y otros (2021). *Datasheets for Datasets*. *Communications of the ACM*, 64(12), 86-92. <https://doi.org/10.1145/3458723>. ↵
3. Hugging Face. (2026). *Model Cards*. <https://huggingface.co/docs/hub/model-cards>. Consultado el 25 de mayo de 2026. ↵
4. OpenAI. (2026). *Models*. <https://developers.openai.com/api/docs/models>. Consultado el 25 de mayo de 2026. ↵
5. OpenAI. (2026). *Pricing*. <https://developers.openai.com/api/docs/pricing>. Consultado el 25 de mayo de 2026. ↵
6. Anthropic. (2026). *Models overview*. <https://platform.claude.com/docs/en/about-claude/models/overview>. Consultado el 25 de mayo de 2026. ↵
7. Anthropic. (2026). *Pricing*. <https://platform.claude.com/docs/en/about-claude/pricing>. Consultado el 25 de mayo de 2026. ↵
8. Google. (2026). *Gemini API: Models*. <https://ai.google.dev/gemini-api/docs/models>. Consultado el 25 de mayo de 2026. ↵

9. Google. (2026). *Gemini Developer API pricing*. <https://ai.google.dev/gemini-api/docs/pricing>. Consultado el 25 de mayo de 2026. ↵
10. Percy Liang y otros (2022). *Holistic Evaluation of Language Models*. <https://arxiv.org/abs/2211.09110>. ↵
1. MLCommons. (2026). *MLPerf Inference: Datacenter benchmark*. <https://mlcommons.org/benchmarks/inference-datacenter/>. Consultado el 25 de mayo de 2026. ↵
2. Open Source Initiative. (2024). *The Open Source AI Definition 1.0*. <https://opensource.org/ai/open-source-ai-definition>. Consultado el 14 de junio de 2026. ↵
3. Open Source Initiative. (2026). *Open Weights: not quite what you've been told*. <https://opensource.org/ai/open-weights>. Consultado el 14 de junio de 2026. ↵
4. OpenAI. (2026). *Models*. <https://developers.openai.com/api/docs/models>. Consultado el 14 de junio de 2026. Anthropic. (2026). *Models overview*. <https://platform.claude.com/docs/en/about-claude/models/overview>. Consultado el 14 de junio de 2026. Google. (2026). *Gemini API: Models*. <https://ai.google.dev/gemini-api/docs/models>. Consultado el 14 de junio de 2026. ↵
5. OpenAI. (2025). *Introducing gpt-oss*. <https://openai.com/index/introducing-gpt-oss/>. Consultado el 14 de junio de 2026. OpenAI. (2026). *OpenAI open-weight models (gpt-oss)*. <https://help.openai.com/en/articles/11870455-openai-open-weight-models-gpt-oss>. Consultado el 14 de junio de 2026. Qwen Team. (2026). *Qwen3.6*. <https://github.com/QwenLM/Qwen3.6>. Consultado el 14 de junio de 2026. Mistral AI. (2025). *Introducing Mistral 3*. <https://mistral.ai/news/mistral-3/>. Consultado el 14 de junio de 2026. DeepSeek-AI. (2025). *DeepSeek-R1*. <https://huggingface.co/deepseek-ai/DeepSeek-R1>. Consultado el 14 de junio de 2026. ↵
6. Meta. (2025). *Llama 4 Community License Agreement*. <https://github.com/meta-llama/llama-models/blob/main/models/llama4/LICENSE>. Consultado el 14 de junio de 2026. Google. (2026). *Gemma 4 model overview*. <https://ai.google.dev/gemma/docs/core>. Consultado el 14 de junio de 2026. ↵
7. Hugging Face. (2026). *Model Cards*. <https://huggingface.co/docs/hub/model-cards>. Consultado el 14 de junio de 2026. ↵
8. Jared Kaplan y otros (2020). *Scaling Laws for Neural Language Models*. <https://doi.org/10.48550/arXiv.2001.08361>. ↵
9. Jordan Hoffmann y otros (2022). *Training Compute-Optimal Large Language Models*. <https://doi.org/10.48550/arXiv.2203.15556>. ↵
10. Evangelos Triantaphyllou. (2000). *Multi-Criteria Decision Making Methods: A Comparative Study*. Springer. <https://doi.org/10.1007/978-1-4757-3157-6>. Y el tratamiento clásico de objetivos y compensaciones: Ralph L. Keeney y Howard Raiffa. (1976). *Decisions with Multiple*

Objectives: Preferences and Value Tradeoffs. Wiley. ↵

1. DeepSeek-AI. (2026). *deepseek-ai/DeepSeek-V4-Pro*. <https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro>. Consultado el 25 de mayo de 2026. ↵
2. Hugging Face. (2026). *DeepSeek-V4*. https://huggingface.co/docs/transformers/model_doc/deepseek_v4. Consultado el 25 de mayo de 2026. ↵
3. Hugging Face. (2026). *Safetensors*. <https://huggingface.co/docs/safetensors/en/index>. Consultado el 25 de mayo de 2026. ↵
4. NVIDIA. (2026). *Transformer Engine: Low Precision Training*. https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/features/low_precision_training/introduction/introduction.html. Hugging Face. (2026). *Quantization*. https://huggingface.co/docs/transformers/main_classes/quantization. vLLM. (2026). *Quantization*. <https://docs.vllm.ai/en/stable/features/quantization/>. Consultados el 25 de mayo de 2026. ↵
5. Zhihong Shao y otros (2024). *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. <https://arxiv.org/abs/2402.03300>. ↵
6. Hugging Face. (2026). *Chat templates*. https://huggingface.co/docs/transformers/chat_templating. Consultado el 25 de mayo de 2026. ↵
7. vLLM. (2026). *OpenAI-Compatible Server*. https://docs.vllm.ai/en/latest/serving/openai_compatible_server/. Consultado el 25 de mayo de 2026. SGLang. (2026). *Welcome to SGLang*. <https://docs.sglang.io/index.html>. Consultado el 25 de mayo de 2026. ↵
8. Docker. (2026). *docker model run*. <https://docs.docker.com/reference/cli/docker/model/run/>. Consultado el 25 de mayo de 2026. ↵
9. Hugging Face. (2026). *Inference Providers*. <https://huggingface.co/docs/inference-providers/en/index>. Consultado el 25 de mayo de 2026. ↵
10. Percy Liang y otros (2022). *Holistic Evaluation of Language Models (HELM)*. Stanford CRFM. <https://arxiv.org/abs/2211.09110>. Consultado el 14 de junio de 2026. ↵
11. Charles A. E. Goodhart. (1975). *Problems of Monetary Management: The U.K. Experience*. Reserva de Australia. La formulación citada es la versión divulgada por Marilyn Strathern (1997). ↵
12. Oscar Sainz y otros (2023). *NLP Evaluation in Trouble: On the Need to Measure LLM Data Contamination for each Benchmark*. Findings of EMNLP 2023. <https://arxiv.org/abs/2310.18018>. ↵

13. Lisa M. Ellram. (1995). *Total Cost of Ownership: An Analysis Approach for Purchasing*. International Journal of Physical Distribution & Logistics Management, 25(8), 4-23.
<https://doi.org/10.1108/09600039510099928>. ↵

Capítulo 05: Modelos locales: Ollama, LM Studio, GGUF y cuantización

Cuando el modelo se queda en tu máquina

Hay una frase que suena sencilla: “lo corremos en local”. Parece que significa privacidad, control, coste bajo y ausencia de dependencias externas. A veces es verdad. Otras veces significa otra cosa: descargar un fichero enorme, pelear con memoria, descubrir que el contexto no cabe, que el modelo responde lento, que la cuantización cambia el comportamiento o que la API local está expuesta de una forma que nadie revisó.

Venimos del [capítulo 04](#), donde aprendimos a leer model cards. Ahora bajamos un nivel: **qué ocurre cuando eliges un modelo descargable y quieres ejecutarlo tú**. Ya no basta con preguntar si el modelo es bueno. Hay que preguntar si cabe, si responde a tiempo, si el runtime entiende su formato, si la licencia encaja, si la calidad tras cuantizar sigue siendo suficiente y si puedes medirlo sin engañarte.

La idea central es esta: **un modelo local no es solo un modelo; es la suma de pesos, formato, runtime, hardware, configuración, API, licencia y evaluación**.

Estado del arte con fecha de corte

Fecha de corte: 10 de junio de 2026.

Fuentes consultadas ese día: documentación oficial de Ollama, LM Studio, Hugging Face Hub sobre GGUF, repositorio de llama.cpp y papers de cuantización LLM.int8, SmoothQuant, GPTQ, AWQ, QLoRA y cuantización entera clásica.

Lo estable es el mecanismo: descargar pesos, elegir formato, cargar en un runtime, repartir memoria entre CPU/GPU, configurar contexto y generación, exponer una API si hace falta, medir calidad y latencia. Lo cambiante son nombres de modelos, soporte de GPU, formatos concretos, variantes de cuantización, límites de contexto y compatibilidad de cada aplicación.

FUENTE	QUÉ APORTA	QUÉ DECISIÓN PERMITE TOMAR
Ollama API. ¹	API local por defecto, endpoints, librerías y compatibilidad básica.	Saber si tu app puede llamar al modelo como servicio local.
Ollama Modelfile. ²	FROM , PARAMETER , TEMPLATE , SYSTEM , ADAPTER , LICENSE y MESSAGE .	Saber qué parte de la conducta se fija en la definición del modelo.
Ollama context length. ³	Relación entre VRAM, contexto por defecto y memoria necesaria.	No subir contexto sin calcular memoria.
Ollama hardware support. ⁴	Soporte de NVIDIA, AMD, Metal y Vulkan.	Comprobar si tu máquina acelera o cae a CPU.
Ollama OpenAI compatibility. ⁵	Compatibilidad con parte de la API de OpenAI.	Reutilizar clientes existentes sabiendo que “compatible” no significa idéntico.
LM Studio basics. ⁶	Flujo de descarga y ejecución local de modelos con pesos accesibles.	Entender qué se descarga y qué significa correr un modelo desde una UI.
LM Studio REST API. ⁷	API nativa local y endpoints compatibles con OpenAI y Anthropic.	Decidir si LM Studio será UI, servidor local o ambas cosas.
LM Studio load. ⁸	Carga con contexto, GPU offload, TTL y estimación de memoria.	Probar si un modelo cabe antes de cargarlo.
Hugging Face GGUF. ⁹	GGUF como formato con tensores y metadatos; visor de metadata y tipos de cuantización.	Leer un .gguf como fichero técnico, no como etiqueta comercial.
llama.cpp. ¹⁰	Runtime C/C++ base del ecosistema GGUF.	Entender de dónde vienen muchas piezas de Ollama, LM Studio y herramientas locales.
LLM.int8. ¹¹	Cuantización 8-bit cuidando valores atípicos en LLMs grandes.	Entender por qué bajar bits no es solo redondear números.
SmoothQuant. ¹²	Reescalado de pesos y activaciones para cuantización post-entrenamiento.	Entender por qué activaciones y pesos se tratan juntos en algunos despliegues.
GPTQ. ¹³	Cuantización post-entrenamiento de pesos usando información aproximada de segundo orden.	Leer GPTQ como método de compresión medible, no como sufijo decorativo.
AWQ. ¹⁴	Cuantización orientada a pesos importantes según activaciones.	Saber por qué algunas cuantizaciones conservan mejor calidad que otras.

FUENTE	QUÉ APORTA	QUÉ DECISIÓN PERMITE TOMAR
QLoRA. ¹⁵	Fine-tuning eficiente sobre modelos cuantizados de 4 bits.	Separar servir un modelo cuantizado de ajustar adaptadores sobre una base cuantizada.
Cuantización entera clásica. ¹⁶	Base de cuantización para inferencia eficiente con enteros.	Recordar que cuantizar es aproximar cálculo, no comprimir un ZIP.

La revisión del 10 de junio añade un matiz importante: “uso Ollama” ya no equivale necesariamente a “todo corre en mi portátil”. Ollama documenta Cloud como una forma de usar modelos remotos grandes con herramientas locales, mientras que la API local sigue viviendo por defecto en `localhost:11434`.¹⁷ LM Studio documenta un servidor local que puede exponerse en la máquina o en red y que ofrece API nativa, compatibilidad OpenAI y compatibilidad Anthropic.¹⁸ llama.cpp, por su parte, documenta `llama-server` con endpoints compatibles, batching, métricas y salidas restringidas por schema.¹⁹

La consecuencia práctica es clara: cuando digas “modelo local”, documenta **dónde se ejecuta realmente**, qué puerto abre, si acepta tráfico de red, qué autenticación tiene, qué contexto reserva, qué cuantización usa, qué plantilla de chat aplica y qué versión del runtime lo sirve. La diferencia entre una práctica de clase y un incidente de seguridad puede ser una bandera de host mal puesta.

Qué no es correr un modelo en local

Correr un modelo en local **no** significa automáticamente que sea privado en sentido fuerte. Si descargas pesos desde un repositorio externo, ejecutas una app, expones un puerto, instalas extensiones o conectas herramientas, sigues teniendo superficie técnica que revisar. Local reduce ciertos problemas de envío de datos a un proveedor remoto, pero no elimina los problemas de licencia, procedencia de ficheros, logs, permisos, backups o red.

Tampoco significa “gratis”. Dejas de pagar por token a una API, pero pagas en hardware, electricidad, tiempo de instalación, memoria, mantenimiento, actualizaciones y evaluación. Si un portátil tarda 40 segundos en responder a una tarea que una API resuelve en 2 segundos por céntimos, puede que local no sea más barato para ese flujo.

Y no significa “igual que el modelo original, pero comprimido”. Un GGUF Q4 no es el mismo objeto operativo que un checkpoint BF16. Puede ser suficientemente bueno, pero hay que demostrarlo con casos propios. Si no mides, no has elegido cuantización; has elegido esperanza.

Qué sí es un modelo local

Un modelo local es un sistema de inferencia donde tú controlas la máquina que carga los pesos. Esa frase es más precisa que “lo tengo en mi ordenador”. Controlar la máquina implica decidir formato, runtime, memoria, configuración, API, permisos y actualización.

PIEZA	QUÉ ES	QUÉ PREGUNTA RESPONDE
Pesos	Ficheros con matrices aprendidas durante entrenamiento o ajuste.	¿Qué modelo estoy ejecutando realmente?
Formato	GGUF, safetensors, ONNX, TensorRT u otro contenedor.	¿Qué runtime puede abrirlo?
Runtime	Ollama, LM Studio, llama.cpp, vLLM, SGLang, Transformers u otro.	¿Quién gestiona memoria y generación?
Hardware	CPU, GPU, NPU, VRAM, RAM, memoria unificada.	¿Cabe y a qué velocidad responde?
Configuración	Contexto, cuantización, GPU offload, temperatura, top-p, stops.	¿Cómo se comporta en cada llamada?
API local	Endpoint HTTP o SDK para llamarlo desde una app.	¿Cómo lo integro en un producto o notebook?
Evaluación	Casos propios con calidad, latencia y memoria.	¿Sirve para mi tarea, no solo para una demo?

La pregunta profesional no es “¿puedo ejecutarlo?”. Es: **¿puedo ejecutarlo con calidad, latencia, memoria, licencia y mantenimiento aceptables?**

La pila local por dentro

Un modelo local atraviesa una pila. Si una pieza falla, el resultado final falla aunque el modelo sea bueno.

CAPA	QUÉ MIDE	QUÉ APORTA	QUÉ SERÍA RAZONABLE
Licencia	Permisos de uso, modificación y redistribución.	Reduce riesgo jurídico y de producto.	Leer licencia del modelo y de variantes derivadas antes de automatizar.
Procedencia	Quién publicó el fichero y con qué historial.	Evita comparar copias, forks o variantes sin saberlo.	Guardar repo, archivo exacto, hash o commit cuando sea posible.
Formato	Cómo están empaquetados pesos y metadatos.	Determina runtime compatible.	GGUF para llama.cpp/Ollama/LM Studio; safetensors para ecosistema Transformers.
Bits por peso	Memoria aproximada de pesos.	Estima si cabe y cuánta calidad podrías perder.	BF16 como referencia; Q4/Q5 si necesitas local barato; Q8 si tienes memoria.
Contexto	Tokens que entran en memoria.	Permite tareas largas, pero aumenta KV cache.	No subir contexto sin medir VRAM/RAM y TTFT.
GPU offload	Capas o partes movidas a GPU.	Reduce latencia si cabe en VRAM.	Usar GPU para capas máximas sin expulsar KV cache ni forzar intercambio lento.
API	Contrato de integración.	Permite usarlo desde una app.	Probar streaming, errores, JSON y límites antes de cambiar proveedor.
Observabilidad	Logs, tiempo, tokens/s, memoria, errores.	Permite saber qué pasa cuando algo va lento.	Medir p50/p95, TTFT, tokens/s y memoria por caso.

La segunda pasada obligatoria es preguntarse qué falta. En un primer intento solemos mirar solo “modelo y cuantización”. Falta comprobar licencia, procedencia, hash, contexto real, plantilla de chat, memoria de KV cache, parámetros de muestreo, endpoint, exposición de red, eval propia y plan de actualización. Si no está escrito, no existe.

La tercera pasada es rehacer la decisión como si tuviera que explicarse en una reunión: “elegimos este modelo local porque cabe con esta cuantización, responde en este p95, conserva esta calidad frente al baseline, usa esta licencia, se integra por esta API y tiene esta alternativa si falla”.

Memoria: la cuenta que evita la fantasía

Antes de descargar nada, haz una estimación. No será perfecta, pero evita decisiones imposibles.

La memoria mínima de pesos se aproxima así:

$$M_{\text{pesos}} \approx N_{\text{parametros}} \cdot \frac{b_{\text{peso}}}{8}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
M_{pesos}	Memoria aproximada ocupada solo por pesos.	Un 7B Q4 ronda 3,5 GB antes de sobrecostes.
$N_{\text{parametros}}$	Número de parámetros del modelo.	7B, 14B, 32B, 70B.
b_{peso}	Bits por peso tras cuantizar.	16, 8, 5, 4, 3.

Pero un modelo cargado no son solo pesos. La memoria que el runtime reserva de verdad al cargar el modelo se descompone en cuatro sumandos, todos reales y medibles:

$$M_{\text{total}} \approx M_{\text{pesos}} + M_{\text{KV}} + M_{\text{runtime}} + M_{\text{margen}}$$

SÍMBOLO	SIGNIFICADO	POR QUÉ IMPORTA
M_{KV}	Memoria de la KV cache.	Crece con contexto, capas, batch y dimensiones de atención (la fórmula está en el capítulo 3).
M_{runtime}	Memoria del programa, buffers y kernels.	No aparece en el tamaño del fichero.
M_{margen}	Colchón para sistema operativo, UI, navegador, otros procesos y picos.	Si no hay margen, el sistema intercambia memoria y se vuelve lento.

Conviene tratar esta suma como una restricción, no como una curiosidad: M_{total} debe caber en la memoria disponible (VRAM de la GPU, o RAM si haces offload a CPU). Hagamos la cuenta para un caso concreto, el mismo que estimarás en el cuaderno del facsímil: un modelo de 8B en Q4 (4 bits) con contexto de 8192 tokens.

$$M_{\text{total}} \approx \underset{\text{pesos: } 8 \cdot 10^9 \cdot 4/8}{4 \text{ GB}} + \underset{\text{KV cache}}{3 \text{ GB}} + \underset{\text{runtime + margen}}{4 \text{ GB}} = 11 \text{ GB}$$

Con una GPU de 16 GB, esos 11 GB caben y dejan unos 5 GB de margen: suficiente para un prototipo de un usuario, no para prometer concurrencia. Si subes el contexto, la KV cache crece y ese margen se come; si pasas a Q8, los pesos se duplican a 8 GB y el total roza el límite. La cuenta, no la intuición, te dice qué cabe.

Cargar un modelo no son solo pesos

8B en Q4, contexto 8192 → $M_{total} = 11$ GB dentro de una GPU de 16 GB



Sube el contexto y la KV cache como el margen; pasa a Q8 y los pesos se duplican a 8 GB.

“Cabe” significa: con este contexto, esta cuantización y este margen.

IA para gente curiosa / Facsímil 04 / Capítulo 05 / 686f6c61

Regla de bolsillo para pesos:

MODELO DENSO	BF16/F 16	Q8/IN T8	Q5 APROX.	Q4 APROX.	LECTURA PRÁCTICA
3B	~6 GB	~3 GB	~1,9 GB	~1,5 GB	Buen candidato para portátiles modestos.
7B	~14 GB	~7 GB	~4,4 GB	~3,5 GB	Punto de entrada local serio.
14B	~28 GB	~14 GB	~8,8 GB	~7 GB	Empieza a exigir GPU/RAM holgada.
32B	~64 GB	~32 GB	~20 GB	~16 GB	Normalmente workstation o servidor.
70B	~140 GB	~70 GB	~44 GB	~35 GB	Multi-GPU, mucha RAM o paciencia.

Lo adecuado depende de la tarea:

SITUACIÓN	PUNTO DE PARTIDA RAZONABLE	POR QUÉ
Notebook, aprendizaje, privacidad personal	3B-8B en Q4/Q5.	Cabe mejor y permite experimentar.
Clasificación o extracción sencilla	7B-14B Q4/Q5 con salida estructurada.	Calidad suficiente si la tarea está bien acotada.
Código, razonamiento o agente local	14B-32B si cabe, comparado contra API.	Estos casos sufren más con modelos pequeños.
Producción con usuarios simultáneos	Runtime de servidor, medición p95 y modelo alternativo.	El problema ya no es una conversación aislada.
Datos sensibles con restricción fuerte	Local o entorno privado con auditoría.	El criterio principal es control, no solo calidad.

La frase “este modelo cabe” debe significar algo concreto: cabe con **este contexto, esta cuantización, este batch, esta GPU offload, este margen de memoria y esta calidad medida**.

Ollama: cómodo no significa invisible

Ollama es una forma práctica de ejecutar modelos y hablar con ellos por CLI, app o API. La documentación oficial indica que su API local se sirve por defecto en `http://localhost:11434/api`, y que también ofrece librerías oficiales para Python y JavaScript. Esa comodidad es valiosa porque permite convertir un modelo local en un servicio consumible por scripts, notebooks o aplicaciones.

Lo importante es no tratar Ollama como una caja opaca. Ollama decide cómo cargar el modelo, qué contexto usar, qué parámetros aplicar y cómo exponer endpoints. Debes saber leer esas decisiones.

TÉRMINO	QUÉ MIDE O CONTROLA	EJEMPLO	DECISIÓN PRÁCTICA
<code>ollama run</code>	Conversación rápida por terminal.	<code>ollama run gemma3</code>	Bien para probar; no basta para evaluar producto.
API local	Contrato HTTP local.	<code>POST /api/generate</code> o <code>POST /api/chat</code> .	Útil para integrar en Python, JS o una app interna.
OpenAI compatibility	Adaptación parcial a clientes existentes.	<code>/v1/chat/completions</code> .	Reutiliza SDKs, pero valida parámetros soportados y errores.
Modelfile	Receta reproducible de modelo, plantilla y parámetros.	<code>FROM llama3.2 ,</code> <code>PARAMETER num_ctx 4096 ,</code> <code>SYSTEM ...</code> .	No guardes solo “uso llama”; guarda el Modelfile.
<code>FROM</code>	Modelo base o fichero GGUF/safetensors.	<code>FROM ./modelo.gguf</code> .	Define qué pesos cargas realmente.
<code>PARAMETER num_ctx</code>	Tamaño de contexto.	<code>num_ctx 4096</code> .	Más contexto usa más memoria; no es gratis.
<code>TEMPLATE</code>	Plantilla de prompt.	Formato de roles y separadores.	Si está mal, el modelo parece peor.
<code>ADAPTER</code>	Adaptador LoRA/QLoRA.	<code>ADAPTER ./adapter.gguf</code> .	Solo encaja si base y adaptador corresponden.
<code>LICENSE</code>	Texto legal asociado.	Licencia incluida en el Modelfile.	Necesario si empaquetas o compartes.
<code>ollama ps</code>	Modelos cargados, procesador y contexto.	<code>100% GPU ,</code> <code>CONTEXT 65536</code> .	Comprueba si de verdad está en GPU y qué contexto usa.

Ollama documenta valores por defecto de contexto según VRAM: por debajo de 24 GiB, 4k; entre 24 y 48 GiB, 32k; desde 48 GiB, 256k. También recomienda contextos altos para tareas como web search, agentes y herramientas de código. La lectura correcta no es “sube contexto a 64k siempre”, sino “si la tarea lo requiere, calcula memoria y verifica `ollama ps`”.

Situación concreta: quieres usar un modelo local para revisar un repositorio. Si el agente mete muchos ficheros en contexto, 4k no alcanza. Pero subir a 64k puede llenar VRAM. La solución profesional no es elegir al azar: mide cuántos tokens entran, decide qué se recupera, sube contexto solo si aporta y comprueba TTFT y p95.

LM Studio: visual, local y medible

LM Studio entra por otro camino: hace cómoda la experiencia visual de buscar, descargar, cargar y probar modelos locales. Su documentación recuerda una distinción importante: para correr local necesitas acceso a los pesos, normalmente en formatos como `.gguf` o `.safetensors`.

Además de UI, LM Studio ofrece API REST local, endpoints compatibles con OpenAI y Anthropic, y CLI. En su REST API v1 aparecen endpoints como `/api/v1/chat`, `/api/v1/models`, `/api/v1/models/load`, `/api/v1/models/unload` y `/api/v1/models/download`. Eso convierte LM Studio en algo más que una app de chat: puede ser un servidor local de desarrollo.

TÉRMINO	QUÉ MIDE O CONTROLA	EJEMPLO	DECISIÓN PRÁCTICA
Modelo descargado	Fichero local con pesos.	Un <code>.gguf</code> de Qwen, Mistral o Gemma.	Comprueba licencia, tamaño, cuantización y procedencia.
<code>lms load</code>	Cargar un modelo en memoria.	<code>lms load <model_key></code> .	Separa descargar de cargar; no todo lo descargado está activo.
<code>--context-length</code>	Tokens de contexto al cargar.	<code>--context-length 8192</code> .	Mide memoria y calidad con ese contexto, no con el máximo teórico.
<code>--gpu</code>	Proporción de offload a GPU.	<code>--gpu 0.5</code> , <code>--gpu max</code> , <code>--gpu off</code> .	Si no usas GPU, la experiencia puede cambiar mucho.
<code>--estimate-only</code>	Estimar memoria sin cargar.	<code>lms load --estimate-only <model_key></code> .	Úsalo antes de romper la sesión por falta de memoria.
TTL	Descargar de memoria tras inactividad.	<code>--ttl 3600</code> .	Evita dejar modelos ocupando RAM/VRAM todo el día.
API nativa	Endpoint local propio.	<code>/api/v1/chat</code> .	Útil si quieres capacidades específicas de LM Studio.
OpenAI-compatible	Endpoint familiar para clientes existentes.	<code>/v1/chat/completions</code> .	Valida streaming, structured output y parámetros.

LM Studio también permite configurar parámetros de inferencia, como `temperature`, `maxTokens` y `topP`, y parámetros de carga, como longitud de contexto y GPU offload. La separación es crucial: `temperature` cambia cómo se elige el siguiente token; `contextLength` cambia cuánta memoria reserva el sistema al cargar.

Situación concreta: en una clase o equipo, LM Studio es excelente para enseñar porque el lector ve el modelo, el fichero, la carga, la conversación y el servidor. En un backend repetible, quizá prefieras Ollama o llama.cpp directamente. La decisión no es “cuál mola más”, sino qué necesitas: UI, script, servidor, compatibilidad, trazabilidad o control fino.

Una prueba local que sí se puede repetir

Una prueba local no debería consistir en abrir una ventana, hacer una pregunta y decidir por impresión. Eso sirve para orientarse, pero no para elegir. Una prueba mínima debe dejar huella: qué modelo era, qué fichero, qué cuantización, qué runtime, qué contexto, qué máquina, qué prompt, qué salida y qué métricas.

Con Ollama, una llamada mínima a la API local puede parecer así:

```
curl http://localhost:11434/api/chat \
-H "Content-Type: application/json" \
-d '{
  "model": "llama3.2",
  "messages": [
    {"role": "system", "content": "Responde en español claro y devuelve JSON válido."},
    {"role": "user", "content": "Clasifica esta incidencia: no puedo entrar al campus virtual."}
  ],
  "stream": false,
  "options": {
    "temperature": 0.2,
    "num_ctx": 4096
  }
}'
```

Con LM Studio en modo compatible con OpenAI, una llamada de integración puede tener esta forma:

```
curl http://localhost:1234/v1/chat/completions \
-H "Content-Type: application/json" \
-d '{
  "model": "modelo-local-cargado",
  "messages": [
    {"role": "system", "content": "Extrae campos con precisión. Si falta un dato, usa
null."},
    {"role": "user", "content": "Factura 2026-05: total 184,30 EUR; proveedor Norte S.L."}
  ],
  "temperature": 0.1,
  "max_tokens": 300
}'
```

Lo importante no es copiar esos comandos tal cual, sino entender qué contrato crean:

CAMPO	QUÉ CONTROLA	QUÉ ANOTAR
model	Identificador que el runtime resuelve a pesos concretos.	Nombre exacto, origen y, si es posible, hash o ruta.
messages	Plantilla de conversación convertida a tokens.	System prompt, user prompt y versión del caso de prueba.
temperature	Aleatoriedad de muestreo.	Valor fijo durante eval; si cambia, cambia la comparación.
num_ctx / contexto	Ventana que reserva memoria.	Contexto usado, no el máximo anunciado.
max_tokens	Límite de salida.	Evita comparar respuestas truncadas con completas.
stream	Si la salida llega por partes.	Afecta percepción de latencia y forma de medir TTFT.

Una tabla de evaluación mínima tendría estas columnas:

COLUMNA	QUÉ SIGNIFICA	POR QUÉ IMPORTA
fecha	Día de la prueba.	Los modelos, runtimes y drivers cambian.
runtime	Ollama, LM Studio, llama.cpp, vLLM, etc.	El mismo fichero puede rendir distinto.
version_runtime	Versión instalada.	Una actualización puede cambiar memoria o salida.
modelo_origen	Repositorio o proveedor de pesos.	Evita comparar copias distintas.
fichero	Nombre exacto del GGUF/safetensors.	Incluye cuantización y variante.
contexto	Tokens de contexto configurados.	Cambia memoria y a veces calidad.
caso_id	Identificador del caso de prueba.	Permite repetir y depurar.
ok_formato	Si cumple contrato de salida.	Importa tanto como “suena bien”.
ok_contenido	Si la respuesta es correcta.	Métrica principal de utilidad.
ttft_s	Tiempo hasta primer token.	Mide sensación inicial de respuesta.
tokens_s	Tokens de salida por segundo.	Mide velocidad de generación.
memoria_pico_gb	Pico observado de RAM/VRAM.	Detecta candidatos que caben sin margen real.
observacion	Fallo o matiz concreto.	Ayuda a saber qué mejorar.

La regla profesional es comparar siempre contra algo: contra una API fuerte, contra Q8, contra otro runtime o contra el mismo modelo con menos contexto. Un resultado aislado no dice si el sistema es bueno; solo dice que funcionó una vez.

Medir latencia y velocidad sin engañarse

En local se habla mucho de “va rápido” o “va lento”. Para que esa frase sirva, hay que separarla en las métricas estándar del serving de modelos, que corresponden a las dos fases de la inferencia (prefill para el primer token, decode para el ritmo de generación):²⁰

$$TTFT = t_{\text{primer token}} - t_{\text{envio}}$$

$$\text{tokens/s} = \frac{N_{\text{tokens salida}}}{t_{\text{fin}} - t_{\text{primer token}}}$$

$$p95 = \text{percentil}_{95}(\text{latencias})$$

MÉTRICA	QUÉ APORTA	QUÉ VALOR ES ADECUADO
TTFT	Cuánto espera la persona hasta ver que algo empieza.	Bajo si hay interacción humana; menos crítico en batch.
Tokens/s	Ritmo de generación una vez arrancó.	Depende de longitud de salida; compáralo en los mismos casos.
Latencia total	Tiempo desde petición hasta respuesta completa.	Importa cuando la salida debe estar cerrada para continuar.
p95	Cómo se comporta el sistema en casos lentos.	Más útil que el promedio si hay usuarios reales.
Memoria pico	Máximo de RAM/VRAM observado.	Debe dejar margen; si va al límite, el sistema será frágil.
Tasa de formato válido	Porcentaje de salidas parseables.	Crucial en JSON, SQL, extracción o agentes.

Ejemplo: un modelo que da 35 tokens/s puede parecer mejor que otro de 22 tokens/s. Pero si el primero falla JSON en 8 de cada 100 casos y el segundo falla en 1, el segundo puede ser mejor para una integración. La métrica adecuada depende de qué duele más: espera, coste, errores de formato, memoria o mantenimiento.

GGUF: el fichero no es el modelo, es el contenedor operativo

GGUF aparece en muchas páginas de modelos y causa una confusión frecuente. No es una arquitectura. No es una familia. No es una licencia. Es un formato de fichero pensado para guardar pesos y metadatos de forma útil para runtimes como llama.cpp.

Hugging Face describe GGUF como un formato binario optimizado para carga y guardado eficiente de modelos; a diferencia de formatos solo de tensores, GGUF incorpora tensores y metadatos estandarizados. Además, el Hub ofrece visor de metadatos y tipos de tensor para archivos GGUF.

DATO EN UN GGUF	QUÉ MIDE	QUÉ APORTA	QUÉ SERÍA RAZONABLE
Arquitectura	Familia que espera el runtime.	Saber si puede cargarse.	No forzar un runtime que no soporta esa arquitectura.
Contexto declarado	Longitud máxima o recomendada.	Límite operativo inicial.	Probar contexto real, no solo máximo.
Tipo de cuantización	Bits y método de aproximación.	Memoria y calidad esperada.	Q4/Q5 para equilibrio, Q8 para calidad si cabe, Q2/Q3 solo con eval fuerte.
Tensor info	Nombre, forma y precisión de tensores.	Auditar qué hay dentro.	Revisar si algo no cuadra entre card y fichero.
Tokenizer metadata	Cómo partir texto.	Evitar prompts mal codificados.	No mezclar tokenizer de otro modelo.
Chat template	Formato de conversación.	Convertir roles a tokens correctos.	Probar con plantilla del modelo, no inventarla.

Nombres como `Q4_K_M`, `Q5_K_M` o `Q8_0` no son decoración. Indican una estrategia de cuantización. Aun así, no hay que tratarlos como una escala universal de calidad. Dos modelos distintos pueden perder capacidades distintas al pasar a Q4. Una tarea de resumen puede sobrevivir bien; una tarea de código o extracción exacta puede sufrir más.

VARIANTE ORIENTATIVA	QUÉ APORTA	CUÁNDO PROBARLA	QUÉ COMPARAR
F16/BF16	Referencia de más calidad y más memoria.	Si tienes RAM/VRAM suficiente.	Sirve como baseline contra cuantizadas.
Q8	Cerca de calidad alta con menos memoria.	Si Q4 falla o la tarea es delicada.	Calidad frente a F16/BF16 y latencia.
Q5	Equilibrio entre memoria y calidad.	Buen primer candidato si Q4 queda justo.	Errores en casos difíciles.
Q4	Mucho ahorro de memoria.	Portátiles, demos, prototipos y tareas acotadas.	Formato, razonamiento, código y alucinaciones.
Q2/Q3	Ahorro agresivo.	Solo si la restricción de memoria manda.	Degradación fuerte con eval propia.

La pregunta “¿qué cuantización uso?” debería reformularse así: “¿cuál es la menor cuantización que mantiene calidad suficiente en mis casos, con latencia y memoria aceptables?”.

Cuantizar no es comprimir un archivo

Cuantizar significa aproximar números. En un modelo, los pesos son valores numéricos. Si los representas con menos bits, ocupan menos y pueden moverse más rápido, pero pierdes resolución. La cuestión es dónde se pierde, cuánto se pierde y si tu tarea lo nota.

La idea matemática mínima es la cuantización afín, el esquema estándar de inferencia entera.²¹ Un peso real x , que antes vivía como FP16, BF16 o FP32, se guarda como un entero pequeño q . Para volver a usarlo, el runtime reconstruye una aproximación $\hat{x}$:

$$q = \text{clip} \left(\text{round} \left(\frac{x}{s} \right) + z, q_{\min}, q_{\max} \right)$$
$$\hat{x} = s \cdot (q - z)$$

SÍMBOLO	QUÉ SIGNIFICA	QUÉ APORTA
x	Valor original del peso o activación.	Referencia de calidad.
q	Entero almacenado con pocos bits.	Ahorro de memoria y movimiento de datos.
s	Escala de cuantización.	Dice cuánto vale un paso entero en el mundo real.
z	Zero-point.	Permite desplazar el cero en cuantización asimétrica.
$q_{\min}, q_{\max}$	Rango posible del entero.	En 4 bits hay muchos menos valores que en 8 o 16.
$\hat{x}$	Valor reconstruido.	Lo que realmente usa el cálculo tras cuantizar.

Si cuantizas a 4 bits, cada valor no puede tomar infinitos matices: solo hay 16 códigos posibles por grupo. El truco está en elegir bien la escala, el grupo y qué tensores reciben más precisión. Por eso dos ficheros Q4 pueden comportarse diferente aunque ambos “sean Q4”.

La pérdida local de información se puede mirar así:

$$\epsilon_W = \frac{\|W - \hat{W}\|_2}{\|W\|_2}$$

Pero esa fórmula no basta para decidir. Un error pequeño en una matriz puede afectar mucho a una tarea de código, y un error mayor puede ser tolerable en una tarea de clasificación sencilla. **La calidad final no se certifica mirando solo el error de pesos; se certifica con casos de uso.**

Qué se puede cuantizar

No todo se cuantiza igual. Esta distinción evita muchas confusiones:

QUÉ SE CUANTIZA	QUÉ CAMBIA	QUÉ APORTA	RIESGO
Pesos	Se guardan matrices del modelo con menos bits.	Reduce tamaño del fichero y memoria de carga.	Puede degradar razonamiento, código o extracción fina.
Activaciones	Se aproximan valores intermedios durante inferencia.	Puede acelerar cálculo si el hardware lo aprovecha.	Más delicado: cambia señales que dependen de cada entrada.
KV cache	Se guarda la memoria de atención con menos precisión.	Ahorra mucho en contextos largos.	Puede empeorar recuperación de detalles lejanos.
Adaptadores	Se ajustan piezas pequeñas sobre una base cuantizada.	Permite fine-tuning barato con QLoRA.	No convierte una base pobre en una buena por sí solo.
Embeddings	Se reducen vectores de representación.	Ahorra almacenamiento y búsqueda.	Puede afectar ranking semántico y vecinos cercanos.

En modelos locales de escritorio, muchas veces estás usando **cuantización de pesos**. Eso no significa que todas las operaciones internas sean enteras, ni que las activaciones estén cuantizadas, ni que el runtime use la GPU de la misma forma que un servidor optimizado. Esta frase es importante: **peso cuantizado no significa inferencia entera de punta a punta.**

Granularidad: dónde se decide el daño

Una misma cuantización de 4 bits puede ser muy distinta según cuántos pesos comparten escala.

GRANULARIDAD	QUÉ OCURRE	VENTAJA	COSTE
Por tensor	Toda la matriz comparte escala.	Muy simple y pocos metadatos.	Mala si hay valores con rangos muy distintos.
Por canal o fila	Cada fila/canal tiene su escala.	Mejor preservación de matrices grandes.	Más metadatos.
Por grupo o bloque	Grupos de 32, 64, 128 u otro tamaño comparten escala.	Buen equilibrio en LLMs locales.	Más complejo; cada formato lo concreta distinto.
Mixta por tensor	Algunos tensores reciben más bits que otros.	Protege partes sensibles del modelo.	El nombre del fichero ya no cuenta toda la historia.

La regla práctica: **grupos más pequeños suelen conservar mejor la señal, pero añaden metadatos y complejidad**. En GGUF, los sufijos como `Q4_K_M` resumen una receta; no sustituyen leer metadata, card del cuantizador y evaluación.

Familias importantes

LLM.int8 mostró que en LLMs grandes no basta con pasar todo a 8 bits de forma ingenua: hay valores atípicos que conviene tratar con cuidado.²² SmoothQuant redistribuye dificultad entre activaciones y pesos para hacer más viable la cuantización de activaciones.²³ GPTQ propuso una ruta de cuantización post-entrenamiento para modelos generativos usando información aproximada de segundo orden.²⁴ AWQ se fijó en qué pesos importan más según activaciones para preservar mejor comportamiento.²⁵ QLoRA popularizó ajustar modelos usando una base cuantizada de 4 bits con adaptadores entrenables.²⁶ La cuantización clásica de redes ya venía de antes, pero los LLMs hicieron visible que no todos los pesos duelen igual.

MÉTODO O FAMILIA	QUÉ CAMBIA	QUÉ APORTA	QUÉ PRUEBA HARÍA
INT8 / LLM.int8	Baja a 8 bits cuidando valores atípicos.	Menos memoria con pérdida pequeña si se hace bien.	Comparar exactitud y formato contra BF16.
SmoothQuant	Reescala pesos y activaciones antes de cuantizar.	Hace más manejable cuantizar activaciones.	Medir latencia real en el runtime elegido.
GPTQ	Cuantización post-entrenamiento de pesos.	Ficheros pequeños y rápidos en ciertos runtimes.	Probar código, matemáticas y extracción.
AWQ	Conserva pesos relevantes según activaciones.	Buen equilibrio para despliegue eficiente.	Comparar frente a GPTQ/GGUF de mismo tamaño.
GGUF Q4/Q5/Q8	Recetas prácticas para llama.cpp y derivados.	Uso local amplio y sencillo.	Medir calidad por tarea, no por sufijo.
QLoRA / NF4	Fine-tuning eficiente sobre base cuantizada.	Ajustar comportamiento con mucha menos memoria.	No confundir ajustar con servir un modelo cuantizado.

Qué significa elegir Q4, Q5 o Q8

Los bits por peso reducen memoria, pero no de forma aislada. Hay metadatos, escalas, grupos y tensores con formatos distintos. Aun así, como intuición:

OPCIÓN	LECTURA TÉCNICA	CUÁNDO SUELE ENCAJAR	CUÁNDO SOSPECHAR
BF16/ F16	Baseline de alta fidelidad.	Evaluación de referencia y tareas delicadas.	Si no cabe o la latencia es inaceptable.
Q8	Aproximación conservadora.	Cuando quieres ahorrar memoria sin perder demasiado.	Si el ahorro no basta para tu máquina.
Q6/Q5	Punto intermedio.	Código, extracción y razonamiento moderado si Q4 falla.	Si el modelo sigue quedando lento o justo de memoria.
Q4	Compromiso local popular.	Chat, resumen, clasificación acotada, aprendizaje y prototipos.	Si hay errores de formato, cálculo, SQL o instrucciones largas.
Q3/Q2	Compromiso agresivo.	Solo cuando la máquina manda y la tarea tolera pérdida.	En casi todo lo que requiera precisión sostenida.

Un ejemplo concreto: si un 7B Q4 cabe y responde fluido, puede ser buena elección para resumir tickets internos. Si el mismo modelo debe devolver JSON contractual con importes, fechas y campos obligatorios, Q5, Q8 o una API fuerte pueden ser más razonables. No porque Q4 sea “malo”, sino porque el coste del error cambió.

Cómo evaluar una cuantización

Cuantizar es siempre un canje: ganas memoria y velocidad, arriesgas calidad. Evaluarlo bien consiste en medir las dos caras del canje sobre los mismos casos, no en confiar en que «4 bits seguro que vale». Medir la degradación de calidad al bajar de precisión es práctica estándar en la literatura de cuantización.²⁷ Las dos métricas son:

$$\Delta_{\text{calidad}} = \text{score}_{\text{baseline}} - \text{score}_{\text{cuantizado}}$$

$$\text{ahorro} = 1 - \frac{M_{\text{cuantizado}}}{M_{\text{baseline}}}$$

Con números: un 7B pasa de ~7 GB en Q8 a ~3,5 GB en Q4, así que ahorro = 1 – 3,5/7 = 0,5, un 50 % menos de memoria. Lo que ese 50 % cuesta en Δ_{calidad} no se adivina: se mide en tus casos. La decisión es aceptable cuando el ahorro es grande y Δ_{calidad} es pequeño *en la tarea que te importa*.

PRUEBA	QUÉ DETECTA	SEÑAL DE ALARMA
Formato exacto	Si respeta JSON, CSV, SQL o campos obligatorios.	Respuestas bonitas pero no parseables.
Casos largos	Si conserva información al crecer el contexto.	Olvida restricciones del principio.
Cálculo simple	Si degrada operaciones numéricas.	Errores en sumas, importes o comparaciones.
Código	Si mantiene sintaxis y pruebas.	Soluciones que parecen plausibles pero no ejecutan.
Recuperación de datos	Si extrae hechos sin inventar.	Cambia nombres, fechas o cantidades.
Latencia	Si el ahorro de memoria mejora la experiencia.	Menos memoria, pero más lentitud por ruta de runtime.

La decisión final debería escribirse así: “frente a BF16/Q8, esta cuantización ahorra X GB, mantiene Y de calidad en nuestros casos, empeora Z, y aun así compensa porque la tarea tolera ese error”. Si no puedes completar esa frase, todavía no has elegido; solo has descargado un fichero.

Ejemplo cercano: si el modelo debe escribir primeras versiones de correos, Q4 puede ser suficiente. Si debe extraer importes exactos de contratos, Q4 puede fallar de forma cara. Si debe generar SQL, un pequeño error puede romper la consulta. La cuantización adecuada depende del coste del error.

El criterio de elección local

Antes de instalar nada, escribe la decisión como una matriz. No hace falta que sea perfecta; hace falta que obligue a pensar.

PREGUNTA	SI LA RESPUESTA ES SÍ	SI LA RESPUESTA ES NO
¿Necesito que los datos no salgan de mi máquina o red?	Local gana peso.	API puede ser más simple.
¿La tarea tolera algo menos de calidad?	Cuantización agresiva puede valer.	Baseline fuerte o API.
¿Necesito baja latencia interactiva?	Mide TTFT y tokens/s local.	Batch o API pueden bastar.
¿Tengo VRAM/RAM suficiente?	Prueba Q5/Q8 o modelos mayores.	Baja tamaño, baja contexto o usa cloud.
¿Necesito integrar en app?	Ollama/LM Studio API/local server.	UI puede ser suficiente para aprendizaje.
¿Necesito control fino de runtime?	llama.cpp/vLLM/SGLang.	Ollama o LM Studio simplifican.
¿Puedo mantener actualizaciones?	Local es viable.	API gestionada reduce carga.

Fíjate en que ninguna pregunta dice “¿qué modelo está de moda?”. El orden correcto es restricción, memoria, calidad, latencia, integración y mantenimiento.

Qué ocurre cuando cargas un modelo local

“Cargar un modelo” no es abrir un archivo. Es convertir un conjunto de ficheros en un proceso de inferencia que ocupa memoria, reserva contexto, aplica una plantilla de chat y queda disponible para recibir peticiones.

El recorrido real suele ser este:

PASO	QUÉ PASA	QUÉ PUEDE FALLAR
1. Resolver el identificador	<code>gemma3</code> , <code>llama3.2</code> , un GGUF concreto o un modelo importado se traducen a ficheros locales.	Creer que dos nombres parecidos son el mismo modelo.
2. Leer metadatos	El runtime mira arquitectura, tokenizer, cuantización, contexto y plantilla.	Usar plantilla o tokenizer incorrectos.
3. Mapear pesos	Los tensores se leen desde disco y se preparan para CPU, GPU o memoria unificada.	El fichero cabe en disco, pero no en memoria.
4. Decidir offload	Algunas capas o cálculos pasan a GPU si hay VRAM o memoria unificada suficiente.	Parte cae a CPU y la latencia se dispara.
5. Reservar KV cache	El runtime reserva memoria para claves y valores de atención según contexto.	Subir contexto llena memoria aunque el modelo pese lo mismo.
6. Aplicar plantilla	Los mensajes <code>system</code> , <code>user</code> y <code>assistant</code> se convierten a tokens con el formato esperado.	El modelo “parece malo” porque se le habla con formato equivocado.
7. Generar tokens	El modelo predice token a token usando sampling, stops, temperatura y límites.	La salida no respeta formato, tarda demasiado o consume más memoria de la prevista.

En Ollama, el servidor local expone una API en `localhost:11434` y permite comprobar modelos cargados con `ollama ps` . En LM Studio, la app permite cargar desde interfaz y el CLI `lms` permite listar, cargar, descargar, iniciar servidor y ver qué está en memoria. En ambos casos hay una idea común: **descargar no es cargar, cargar no es evaluar, evaluar no es integrar**.

Hardware y dependencias que sí importan

El hardware no se resume en “tengo GPU”. Para modelos locales, importan memoria, ancho de banda, drivers, disco, sistema operativo y puerto de servicio.

RECURSO	QUÉ MIRAR	LECTURA PRÁCTICA
Disco	Espacio para modelos, duplicados, cachés y versiones.	Un proyecto local puede ocupar decenas o cientos de GB. No lo metas sin pensar en el disco del sistema.
RAM	Memoria principal para CPU, buffers, runtime y partes no aceleradas.	Si no hay margen, el sistema intercambia memoria y todo parece roto.
VRAM o memoria unificada	Pesos, KV cache y buffers en GPU o Apple Silicon.	Un modelo Q4 pequeño puede ir fluido; uno grande con contexto alto puede dejar de caber.
CPU	Fallback y preparación de datos.	Sirve para ejecutar, pero puede ser demasiado lento para uso interactivo.
GPU y drivers	NVIDIA, AMD, Metal, ROCm, Vulkan o CPU.	Asegura soporte antes de prometer latencia. Ollama documenta NVIDIA, AMD, Metal y Vulkan.
Contexto	Tokens máximos disponibles en memoria.	Más contexto aumenta memoria; no es una barra “gratis”.
Puerto local	11434 para Ollama, 1234 habitual en LM Studio.	Revisa si escucha solo en localhost o si lo expones en red.
Herramientas de trabajo	Terminal, curl, Python 3, runtime elegido y drivers.	Sin medición por terminal, todo queda en sensación visual.

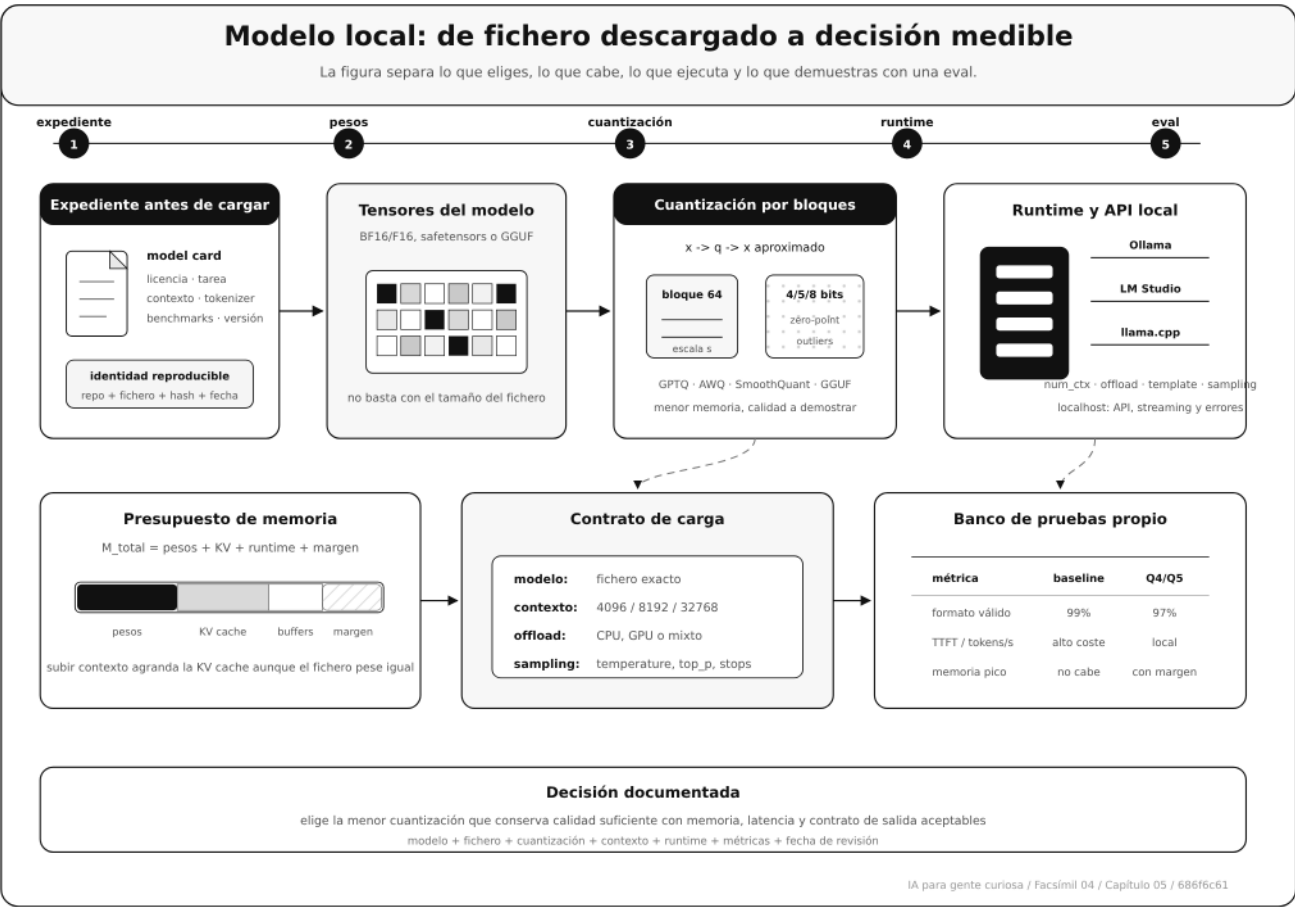
Dónde se guardan los modelos también importa. Ollama documenta rutas por defecto: macOS usa `~/.ollama/models`, Linux usa `/usr/share/ollama/.ollama/models` y Windows usa `C:\Users\%username%\ollama\models`; si necesitas moverlos, `OLLAMA_MODELS` define otra ubicación. LM Studio lo gestiona desde “My Models” y `lms ls` refleja el directorio de modelos configurado en la app.

Una instalación local mínima tiene estas dependencias conceptuales:

DEPENDENCIA	PARA QUÉ SIRVE	SEÑAL DE QUE ESTÁ BIEN
Runtime	Cargar y ejecutar el modelo.	<code>ollama -v</code> o <code>lms --help</code> responden.
Modelo descargado	Tener pesos reales en disco.	<code>ollama list</code> o <code>lms ls</code> muestran el modelo.
Modelo cargado	Ocupar memoria para inferencia.	<code>ollama ps</code> o <code>lms ps</code> muestran un modelo activo.
API local	Integrar con scripts o apps.	<code>curl localhost:11434</code> o <code>curl localhost:1234</code> responde.
Driver GPU	Acelerar inferencia.	<code>ollama ps</code> indica GPU o <code>lms load --estimate-only</code> estima uso razonable.
Prueba propia	Ver si el sistema sirve para tu tarea.	Tienes métricas de latencia, memoria y formato válido.

Mi recomendación para clase o primer montaje: empieza con un modelo pequeño o mediano, una cuantización conservadora (`Q4_K_M` , `Q5_K_M` o equivalente), contexto 4096 u 8192, y una tarea concreta. Luego sube tamaño o contexto solo si puedes explicar qué ganaste.

Mapa visual del sistema local



En el día a día

En un proyecto real, los modelos locales aparecen en cinco situaciones: prototipado rápido, privacidad, trabajo offline, coste por volumen y aprendizaje técnico. Cada una exige una lectura distinta.

Si prototipas, quieres fricción baja. Ollama o LM Studio te permiten probar rápido. Si el objetivo es privacidad, ya no basta con “local”: revisa qué app abre red, dónde quedan logs, qué permisos tiene el servidor y si el puerto escucha solo en localhost. Si el objetivo es coste, calcula coste de hardware y tiempo del equipo. Si el objetivo es aprendizaje, acepta modelos pequeños y mide para entender.

La señal de madurez no es correr el modelo más grande posible. Es poder decir: “en esta máquina, con este modelo, esta cuantización y este contexto, obtenemos esta calidad, este TTFT, estos tokens/s y esta memoria”.

Por qué debería importarte

Porque local cambia el equilibrio del sistema. En una API externa, pagas por uso y delegas hardware. En local, controlas pesos y datos, pero compras complejidad: memoria, runtime, temperatura, contexto, actualizaciones y fallos.

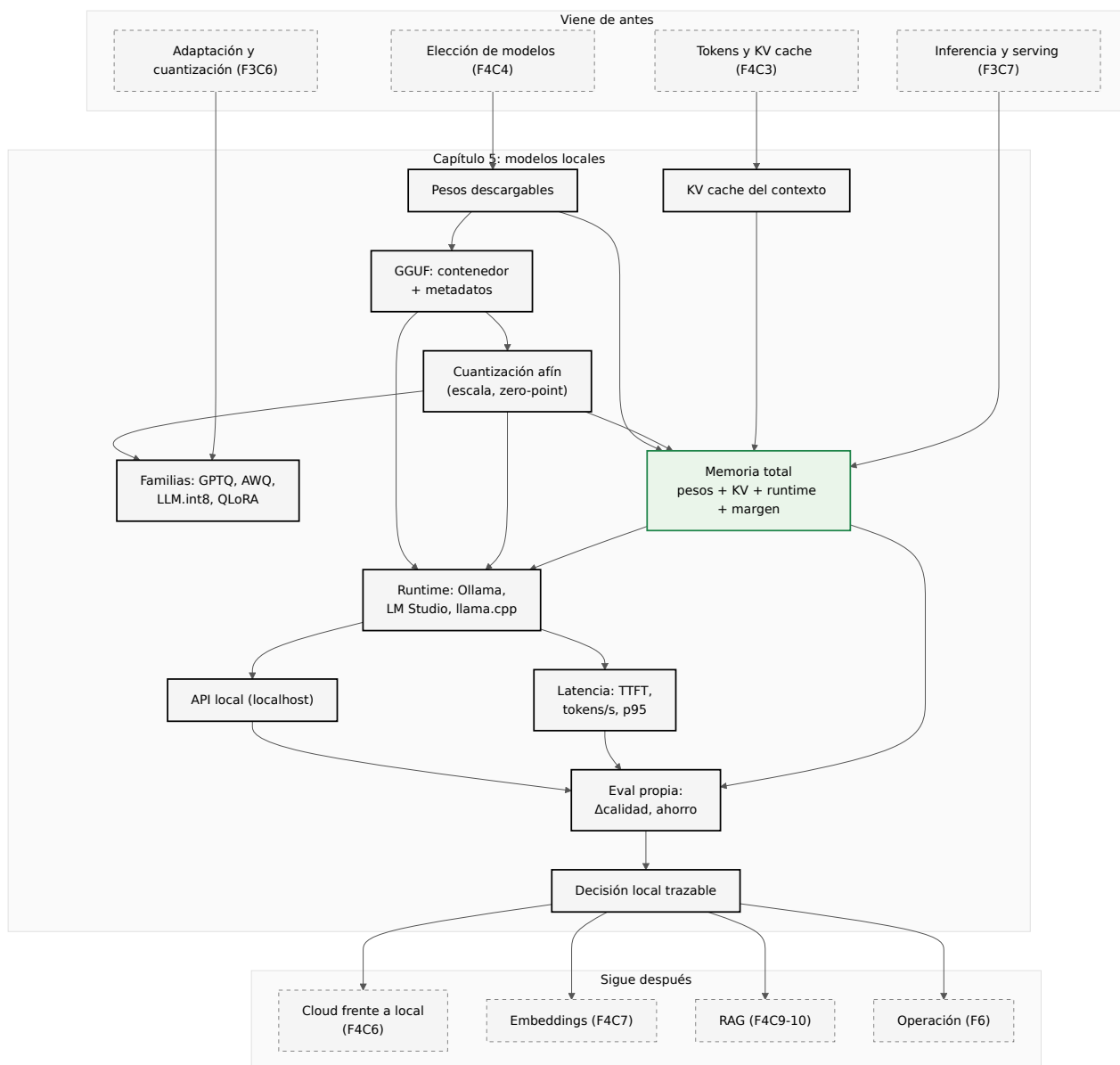
También importa porque los próximos capítulos dependen de esto. RAG local, embeddings, text-to-SQL, agentes y herramientas internas pueden apoyarse en modelos locales, pero solo si sabes cuándo local es suficiente y cuándo estás intentando ahorrar en el sitio equivocado.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir local con privado absoluto	Local reduce envío remoto, pero no elimina permisos, logs, red ni procedencia.	Revisar puerto, logs, permisos, licencia y origen de pesos.
Elegir por tamaño descargado	El fichero no incluye KV cache, runtime ni margen de memoria.	Estimar pesos + KV + runtime + margen.
Subir contexto sin medir	Más contexto puede llenar memoria y empeorar TTFT.	Probar contexto real y mirar memoria durante la carga.
Comparar Q4 contra API sin baseline	Quizá el fallo viene de la cuantización, no del modelo.	Comparar contra Q8/BF16 o una API fuerte en los mismos casos.
Pensar que OpenAI-compatible significa igual	Puede haber diferencias en herramientas, JSON, streaming, errores y parámetros.	Ejecutar tests de contrato antes de cambiar backend.
No guardar configuración	No puedes reproducir un resultado si no sabes contexto, temperatura, offload y fichero.	Guardar Modelfile, modelo exacto, cuantización y métricas.

Cómo encaja todo

Este mapa conecta los modelos locales con lo que ya vimos (tokens, KV cache y elección de modelo) y con lo que viene después. Separa tres planos: lo que **decides** (qué pesos, formato y cuantización), lo que **ejecutas** (runtime, memoria y API) y lo que **mides** (latencia y eval propia) antes de cerrar una decisión trazable.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Modelo local	Modelo que se carga y ejecuta en una máquina controlada por ti.
Open weights	Pesos descargables bajo licencia concreta.
Runtime	Programa que ejecuta la inferencia y gestiona memoria.
GGUF	Formato que guarda tensores y metadatos para runtimes como llama.cpp.
GPU offload	Movimiento de capas o trabajo a GPU para acelerar inferencia.
VRAM	Memoria de GPU usada por pesos, KV cache y buffers.
KV cache	Memoria que crece con contexto y generación autoregresiva.
Cuantización	Representar pesos con menos bits para ahorrar memoria.
Escala de cuantización	Factor que convierte un entero pequeño en una aproximación del valor real.
Zero-point	Desplazamiento entero usado para representar el cero en cuantización asimétrica.
Granularidad	Tamaño del tensor, fila o bloque que comparte escala y metadatos.
PTQ	Cuantización aplicada después de entrenar, sin reentrenar todo el modelo.
Q4_K_M	Variante GGUF de 4 bits usada como equilibrio frecuente.
Modelfile	Receta de Ollama para modelo, parámetros, plantilla y licencia.
Local API	Endpoint HTTP para llamar al modelo desde una app local.
TTFT	Tiempo hasta recibir el primer token.
Tokens por segundo	Velocidad de generación durante la salida.

Antes de pasar página

☐ ¿Puedo explicar por qué local no significa automáticamente privado, barato o mejor? (Si no, vuelve a «Qué no es correr un modelo en local».)

- ☐ ¿Sé separar pesos, formato, runtime, hardware, contexto y API? (Si no, vuelve a «La pila local por dentro».)
- ☐ ¿Puedo estimar memoria de pesos con parámetros y bits? (Si no, vuelve a «Memoria: la cuenta que evita la fantasía».)
- ☐ ¿Sé por qué el contexto aumenta memoria aunque el fichero pese lo mismo? (Si no, vuelve a «Memoria: la cuenta que evita la fantasía».)
- ☐ ¿Distingo GGUF, safetensors, cuantización y arquitectura? (Si no, vuelve a «GGUF: el fichero no es el modelo, es el contenedor operativo».)
- ☐ ¿Sé cuándo usar Ollama y cuándo LM Studio por su forma de trabajo? (Si no, vuelve a «Ollama: cómodo no significa invisible».)
- ☐ ¿Sé dónde se guardan los modelos y cómo mover la ruta si hace falta? (Si no, vuelve a «Qué ocurre cuando cargas un modelo local».)
- ☐ ¿Puedo explicar qué mide `ollama ps`, `lms ps` o `lms load --estimate-only`? (Si no, vuelve a «Medir latencia y velocidad sin engañarse».)
- ☐ ¿He levantado una API local y probado una petición real con `curl` o Python? (Si no, vuelve a «Una prueba local que sí se puede repetir».)
- ☐ ¿Sé comparar Q4, Q5, Q8 y BF16 con una eval propia? (Si no, vuelve a «Cuantizar no es comprimir un archivo».)
- ☐ ¿Sé distinguir si la respuesta de un modelo local sirve para una aplicación, no solo para leerla? (Si no, repasa la sección correspondiente.)

En resumen

IDEA FUERZA	DETALLE
Local es una pila completa.	Pesos, formato, runtime, memoria, API, licencia y eval trabajan juntos.
El tamaño del fichero no basta.	Hay que sumar KV cache, runtime, margen y contexto real.
GGUF es un contenedor operativo.	Guarda tensores y metadatos para runtimes locales, no una garantía de calidad.
Cuantizar cambia el sistema.	Puede ahorrar memoria y coste, pero debe medirse contra un baseline.
Ollama y LM Studio resuelven problemas distintos.	Uno favorece flujo simple y API; el otro añade UI, gestión visual y servidor local.
Instalar no es integrar.	Debes probar descarga, carga, API, latencia y contrato de salida.
La decisión local debe quedar escrita.	Modelo, cuantización, contexto, hardware, métricas y alternativa.

Para saber más

Dettmers, T. y otros (2022). *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. <https://doi.org/10.52202/068431-2198>

Frantar, E. y otros (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++*. <https://github.com/ggml-org/llama.cpp>

Hugging Face. (2026). *GGUF*. <https://huggingface.co/docs/hub/en/gguf>

Jacob, B. y otros (2018). *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. <https://doi.org/10.1109/CVPR.2018.00286>

Lin, J. y otros (2024). *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. <https://arxiv.org/abs/2306.00978>

Dettmers, T. y otros (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. <https://arxiv.org/abs/2305.14314>

LM Studio. (2026). *Configuring the Model*. <https://lmstudio.ai/docs/typescript/llm-prediction/parameters>

LM Studio. (2026). *Get started with LM Studio*. <https://lmstudio.ai/docs/app/basics>

LM Studio. (2026). *LM Studio API*. <https://lmstudio.ai/docs/developer/rest>

LM Studio. (2026). *LM Studio Developer Docs*. <https://lmstudio.ai/docs/developer>

LM Studio. (2026). *lms: LM Studio's CLI*. <https://lmstudio.ai/docs/cli>

LM Studio. (2026). *lms load*. <https://lmstudio.ai/docs/cli/local-models/load>

Ollama. (2026). *FAQ*. <https://docs.ollama.com/faq>

Ollama. (2026). *Linux*. <https://docs.ollama.com/linux>

Ollama. (2026). *macOS*. <https://docs.ollama.com/macos>

Ollama. (2026). *Quickstart*. <https://docs.ollama.com/quickstart>

Ollama. (2026). *Windows*. <https://docs.ollama.com/windows>

Ollama. (2026). *Context length*. <https://docs.ollama.com/context-length>

Ollama. (2026). *Hardware support*. <https://docs.ollama.com/gpu>

Ollama. (2026). *Introduction to the Ollama API*. <https://docs.ollama.com/api/introduction>

Ollama. (2026). *Modelfile Reference*. <https://docs.ollama.com/modelfile>

Ollama. (2026). *OpenAI compatibility*. <https://docs.ollama.com/api/openai-compatibility>

Xiao, G. y otros (2023). *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models*. <https://arxiv.org/abs/2211.10438>

Notas

1. Ollama. (2026). *Introduction to the Ollama API*. <https://docs.ollama.com/api/introduction>. Consultado el 10 de junio de 2026. ↵
2. Ollama. (2026). *Modelfile Reference*. <https://docs.ollama.com/modelfile>. Consultado el 10 de junio de 2026. ↵
3. Ollama. (2026). *Context length*. <https://docs.ollama.com/context-length>. Consultado el 10 de junio de 2026. ↵
4. Ollama. (2026). *Hardware support*. <https://docs.ollama.com/gpu>. Consultado el 10 de junio de 2026. ↵

5. Ollama. (2026). *OpenAI compatibility*. <https://docs.ollama.com/api/openai-compatibility>. Consultado el 10 de junio de 2026. ↵
6. LM Studio. (2026). *Get started with LM Studio*. <https://lmstudio.ai/docs/app/basics>. Consultado el 10 de junio de 2026. ↵
7. LM Studio. (2026). *LM Studio API*. <https://lmstudio.ai/docs/developer/rest>. Consultado el 10 de junio de 2026. ↵
8. LM Studio. (2026). *Imgs load*. <https://lmstudio.ai/docs/cli/local-models/load>. Consultado el 10 de junio de 2026. ↵
9. Hugging Face. (2026). *GGUF*. <https://huggingface.co/docs/hub/en/gguf>. Consultado el 10 de junio de 2026. ↵
10. ggml-org. (2026). *llama.cpp: LLM inference in C/C++*. <https://github.com/ggml-org/llama.cpp>. Consultado el 10 de junio de 2026. ↵
11. 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://doi.org/10.52202/068431-2198>. ↵
12. 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>. ↵
13. 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>. ↵
14. 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>. ↵
15. Dettmers, T., Pagnoni, A., Holtzman, A. y Zettlemoyer, L. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. *Advances in Neural Information Processing Systems* 36. <https://arxiv.org/abs/2305.14314>. ↵
16. Jacob, B. y otros (2018). *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. *Proceedings of CVPR*, 2704-2713. <https://doi.org/10.1109/CVPR.2018.00286>. ↵
17. Ollama. (2026). *Cloud*. <https://docs.ollama.com/cloud>. Consultado el 10 de junio de 2026. ↵
18. LM Studio. (2026). *Local Server*. <https://lmstudio.ai/docs/developer/core/server>. Consultado el 10 de junio de 2026. ↵
19. ggml-org. (2026). *llama-server*. <https://github.com/ggml-org/llama.cpp/blob/master/tools/server/README.md>. Consultado el 10 de junio de 2026. ↵

- !0. Yinmin Zhong y otros (2024). *DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving*. OSDI 2024. <https://arxiv.org/abs/2401.09670>.
- Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. <https://arxiv.org/abs/2309.06180>. ↵
- !1. Benoit Jacob y otros (2018). *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. CVPR. <https://arxiv.org/abs/1712.05877>. ↵
- !2. Tim Dettmers y otros (2022). *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. NeurIPS. <https://arxiv.org/abs/2208.07339>. ↵
- !3. Guangxuan Xiao y otros (2023). *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models*. ICML. <https://arxiv.org/abs/2211.10438>. ↵
- !4. Elias Frantar y otros (2022). *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. <https://arxiv.org/abs/2210.17323>. ↵
- !5. Ji Lin y otros (2023). *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. MLSys 2024. <https://arxiv.org/abs/2306.00978>. ↵
- !6. Tim Dettmers y otros (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. NeurIPS. <https://arxiv.org/abs/2305.14314>. ↵
- !7. Ji Lin y otros (2023). *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. MLSys 2024. <https://arxiv.org/abs/2306.00978>. ↵

Capítulo 06: Cloud frente a local: privacidad, latencia y coste

La decisión que no cabe en un eslogan

Después de montar un modelo local, aparece una tentación muy humana: convertirlo en bandera. “Local es privado”. “Cloud es escalable”. “Local es barato”. “Cloud es caro”. Ninguna de esas frases aguanta mucho si la llevas a producción.

Venimos del capítulo 05, donde vimos que un modelo local es una pila: pesos, runtime, memoria, API, configuración y evaluación. Ahora comparamos esa pila con una API gestionada o una plataforma cloud. No para escoger por ideología, sino para responder una pregunta de ingeniería: **dónde debe ejecutarse esta inferencia, con estos datos, este volumen, esta latencia, este presupuesto y este nivel de operación.**

La idea central es esta: **local y cloud no son bandos; son posiciones distintas dentro de una arquitectura.**

Estado del arte con fecha de corte

Fecha de corte: 10 de junio de 2026.

Fuentes consultadas ese día: documentación oficial de controles de datos y precios de OpenAI, Anthropic, Google Cloud Vertex AI y Amazon Bedrock; documentación oficial de latencia; y artículos académicos sobre latencia de cola y edge computing.

Lo estable es el método: mirar frontera de confianza, retención, región, latencia, coste total, elasticidad, operación y plan de salida. Lo cambiante son precios, modelos disponibles, regiones, controles de retención, límites de API, multiplicadores por residencia, descuentos y disponibilidad de hardware.

FUENTE	QUÉ APORTA	CÓMO USARLA
OpenAI data controls. ¹	Explica uso de datos, retención por endpoint y controles como retención cero o monitorización modificada cuando aplica.	Para no decir “API” sin revisar qué se guarda y durante cuánto tiempo.
OpenAI models API. ²	Devuelve modelos disponibles para una clave y metadatos básicos como <code>id</code> , <code>created</code> y <code>owned_by</code> .	Para no usar identificadores copiados de ejemplos viejos.
OpenAI pricing. ³	Precios por tokens, cache, batch y modalidades.	Para calcular coste por flujo real, no por intuición.
OpenAI latency optimization. ⁴	Criterios para reducir latencia de aplicaciones con modelos.	Para separar tokens, streaming, modelo y arquitectura.
Anthropic data retention. ⁵	Diferencia arreglos de retención, alcance de ZDR y funciones que necesitan almacenamiento.	Para preguntar qué modo contractual tienes, no qué marca usas.
Anthropic models API. ⁶	Lista modelos disponibles con paginación y fechas de creación.	Para separar “familia Claude” de identificador concreto de API.
Anthropic pricing. ⁷	Precios por entrada, salida, cache y geografía.	Para ver que cache y residencia también cambian coste.
Gemini models. ⁸	Lista modelos, modalidades y patrones de versión estable, preview, latest y experimental.	Para fijar versiones estables y no depender de alias que cambian.
Vertex AI data governance. ⁹	Controles de gobernanza, retención y condiciones de servicios generativos.	Para tratar cloud como contrato de datos, no solo endpoint.
Vertex AI pricing. ¹⁰	Precios por modelo, tokens, herramientas y modalidades.	Para revisar token, imagen, vídeo, grounding y extras.
Amazon Bedrock data protection. ¹¹	Modelo de responsabilidad compartida, cifrado, IAM, logs y separación con proveedores de modelos.	Para entender qué controla AWS y qué sigues controlando tú.
Amazon Bedrock pricing. ¹²	Precios por modelo y modo de inferencia.	Para comparar marketplaces, regiones y modalidades.

FUENTE	QUÉ APORTA	CÓMO USARLA
OpenRouter models API. ¹³	Expone modelos de muchos proveedores con <code>context_length</code> , modalidades, precios, parámetros soportados y enlaces a endpoints.	Para construir inventario técnico de modelos cloud comparables.
OpenRouter routing. ¹⁴	Permite controlar proveedores, preferencias, rutas y sustitución cuando un proveedor no encaja.	Para entender un gateway como capa de selección, no como “un modelo más”.
Ollama Cloud y API. ¹⁵	Mantiene herramientas locales mientras ejecuta modelos grandes en el servicio cloud de Ollama; la API local usa <code>localhost:11434/api</code> , cloud <code>https://ollama.com/api</code> y compatibilidad parcial OpenAI en <code>/v1</code> .	Para distinguir “uso Ollama” de “la inferencia está en mi máquina”.
vLLM, llama.cpp, TGI y SGLang. ¹⁶	Runtimes para servir modelos propios con APIs compatibles, batching, plantillas de chat, GPU/CPU y opciones de producción.	Para montar servidor local con parámetros, observabilidad y contrato, no solo “ejecutar un modelo”.
Alquiler de GPU. ¹⁷	Permite alquilar VM o instancia con acelerador, pagar por tiempo, reservar capacidad o usar capacidad con descuento bajo condiciones.	Para separar coste de API de coste de infraestructura propia en cloud.
GPU clouds y VM GPU. ¹⁸	Documentan familias GPU, billing, almacenamiento, pods dedicados, contenedores y disponibilidad.	Para entender que alquilar GPU incluye storage, imagen, datos, arranque y operación.
The Tail at Scale. ¹⁹	Explica por qué p95 y p99 importan más que el promedio en servicios interactivos.	Para medir latencia como experiencia real, no media bonita.
Edge Computing: Vision and Challenges. ²⁰	Sitúa edge/local como forma de acercar cómputo a datos y usuarios.	Para entender local como ubicación arquitectónica, no como capricho.

La revisión del 10 de junio refuerza que cloud frente a local no se decide solo con precio por millón de tokens. OpenAI separa controles de datos, optimización de coste, optimización de latencia y selección de modelos; Anthropic documenta retención, residencia, uso/coste y límites de tasa; AWS Bedrock presenta la protección de datos desde responsabilidad compartida; y Vertex AI publica capacidades, release notes y SLA por servicio.²¹

La pregunta de ingeniería queda así: ¿puedo medir coste, latencia, retención, región, límites, errores y salida por proveedor con la misma rúbrica? Si no puedes, todavía no estás comparando local y cloud; estás comparando sensaciones. Un diseño serio incluye inventario de modelos, contrato de datos, presupuesto por flujo, p95/p99, plan de degradación, logs filtrados y salida de emergencia si un proveedor cambia modelo, precio o disponibilidad.

Qué no significa “privado”

Privado no significa “no usa internet”. Un portátil con un modelo local puede tener logs, copias de seguridad, extensiones, puertos abiertos, carpetas sincronizadas y usuarios con permisos amplios. Cloud no significa automáticamente “lo ve todo el proveedor”: puede haber contratos, controles de retención, cifrado, regiones, IAM, redes privadas y auditoría. La pregunta correcta no es si algo suena local o remoto; es **dónde existe texto claro, quién puede acceder, cuánto tiempo queda guardado y qué contrato lo gobierna**.

Tampoco conviene confundir privacidad con cumplimiento. Puedes proteger datos en local y aun así incumplir una política interna por falta de trazabilidad. Puedes usar cloud y cumplir mejor porque tienes auditoría, control de accesos y residencia definida. Depende del caso.

Y privacidad tampoco equivale a calidad. El modelo local más controlado puede no servir para la tarea. El modelo cloud más potente puede no encajar con los datos. La decisión empieza por la frontera de confianza, pero no termina ahí.

La frontera de confianza

Antes de hablar de coste, dibuja por dónde viajan los datos.

LUGAR DONDE EXISTE EL DATO	PREGUNTA QUE DEBES HACER	SEÑAL DE CONTROL
Navegador o app cliente	¿El usuario escribe datos sensibles?	Minimización antes de enviar.
Backend propio	¿Qué se loguea antes de llamar al modelo?	Logs filtrados, cifrado y permisos.
Proveedor cloud	¿Qué retiene el endpoint usado?	Contrato, región, DPA/BAA cuando aplique, controles de retención.
Runtime local	¿Quién puede leer disco, memoria, logs y puerto?	Usuario dedicado, permisos, localhost , rutas controladas.
Herramientas conectadas	¿El modelo llama sistemas externos?	Scopes mínimos, auditoría y validación de salida.
Caché o vector store	¿Se guardan prompts, fragmentos o embeddings?	TTL, borrado, cifrado y separación por cliente.

La frontera no es un punto; es una cadena. Si haces RAG local pero subes la respuesta completa a una API cloud para “mejorarla”, la frontera cambió. Si usas cloud pero anonimizas antes, haces hashing de identificadores y evitas enviar documentos completos, también cambió.

Latencia: no mires solo el promedio

La latencia total de una llamada no es un número único: es la suma de las etapas por las que pasa la petición. Esta descomposición es una identidad de contabilidad de tiempo (el total es la suma de sus partes), útil para localizar dónde duele:

$$L_{total} = L_{red} + L_{cola} + L_{prefill}(T_{entrada}) + L_{decode}(T_{salida}) + L_{herramientas} + L_{postproceso}$$

TÉRMINO	QUÉ SIGNIFICA	QUÉ CAMBIA LOCAL/CLOUD
L_{red}	Viaje entre cliente, backend y modelo.	Local puede reducirlo; cloud depende de región y red.
L_{cola}	Espera antes de ejecutar.	Cloud puede absorber picos; local se satura antes.
$L_{prefill}$	Procesar tokens de entrada.	Crece con contexto. RAG y documentos largos lo disparan.
L_{decode}	Generar tokens de salida.	Depende de modelo, hardware, cuantización y runtime.
$L_{herramientas}$	Consultas a bases de datos, APIs o buscadores.	A veces domina más que el modelo.
$L_{postproceso}$	Validar JSON, guardar, renderizar o reintentar.	Suele olvidarse en demos.

Dos de estos términos no son intuición de ingeniería, sino que tienen una base teórica que conviene conocer.

La cola obedece a la teoría de colas. La Ley de Little relaciona el número medio de peticiones en el sistema L , la tasa de llegada λ y el tiempo medio que cada una pasa en el sistema W :²²

$$L = \lambda \cdot W$$

La consecuencia práctica: cuando la tasa de llegada se acerca a la capacidad de servicio, la espera en cola se dispara de forma no lineal. Por eso un servidor al 90 por ciento de utilización ya sufre colas largas, y cloud, al poder escalar réplicas, absorbe picos que saturarían a un servidor local fijo.

El modelo sigue las dos fases de la inferencia autoregresiva. L_{prefill} procesa todo el prompt de entrada de una vez; L_{decode} genera la salida token a token reutilizando la KV cache.²³ La latencia de la parte de modelo se expresa con las dos métricas estándar del serving, TTFT (*time to first token*, dominada por el prefill) y TPOT (*time per output token*, dominada por el decode):

$$L_{\text{modelo}} = \text{TTFT} + (N_{\text{salida}} - 1) \cdot \text{TPOT}$$

Esta es la descomposición que usan los sistemas de serving modernos para optimizar latencia, hasta el punto de separar prefill y decode en máquinas distintas porque tienen perfiles de cómputo opuestos.²⁴

El promedio engaña. Si una app tiene 1000 usuarios y el 5 por ciento sufre esperas largas, ese 5 por ciento puede ser el que abandona. Dean y Barroso explican por qué la cola de latencias importa en sistemas a escala: cuando una petición depende de muchos componentes, la probabilidad de que al menos uno vaya lento crece, y por eso hay que mirar percentiles altos, no la media.²⁵ No basta con que “normalmente vaya bien”.

MÉTRICA	QUÉ MIDE	DECISIÓN
p50	Experiencia típica.	Sirve para ver sensación normal.
p95	Casos lentos frecuentes.	Útil para producto.
p99	Cola larga.	Útil para flujos críticos o muchos usuarios.
TTFT	Tiempo hasta ver el primer token.	Mejora percepción si usas streaming.
tokens/s	Velocidad de salida.	Importa en respuestas largas.
timeouts	Peticiones que no terminan a tiempo.	Marcan límites de arquitectura.

Local puede ganar si el usuario y los datos están cerca del modelo y el modelo cabe bien. Cloud puede ganar si necesita hardware optimizado, batch, escalado, modelos grandes, alta concurrencia o regiones específicas. La única forma seria de decidir es medir el mismo caso en ambas rutas.

Coste: token barato no significa sistema barato

Igual que en el capítulo 4, lo que decide no es el precio por token, sino el coste total de propiedad: la suma de todo lo que pagas por servir la función al mes.²⁶ En cloud, ese coste mensual para N peticiones es:

$$C_{\text{cloud}} = N \cdot \left(\frac{T_{\text{entrada}}}{10^6} P_{\text{entrada}} + \frac{T_{\text{salida}}}{10^6} P_{\text{salida}} \right) + C_{\text{cache}} + C_{\text{herramientas}} + C_{\text{almacenamiento}} + C_{\text{observabilidad}}$$

En local, el coste no escala con las peticiones sino con la capacidad: es un coste mayormente fijo, donde el hardware se amortiza a lo largo de su vida útil. La cuenta mínima es:

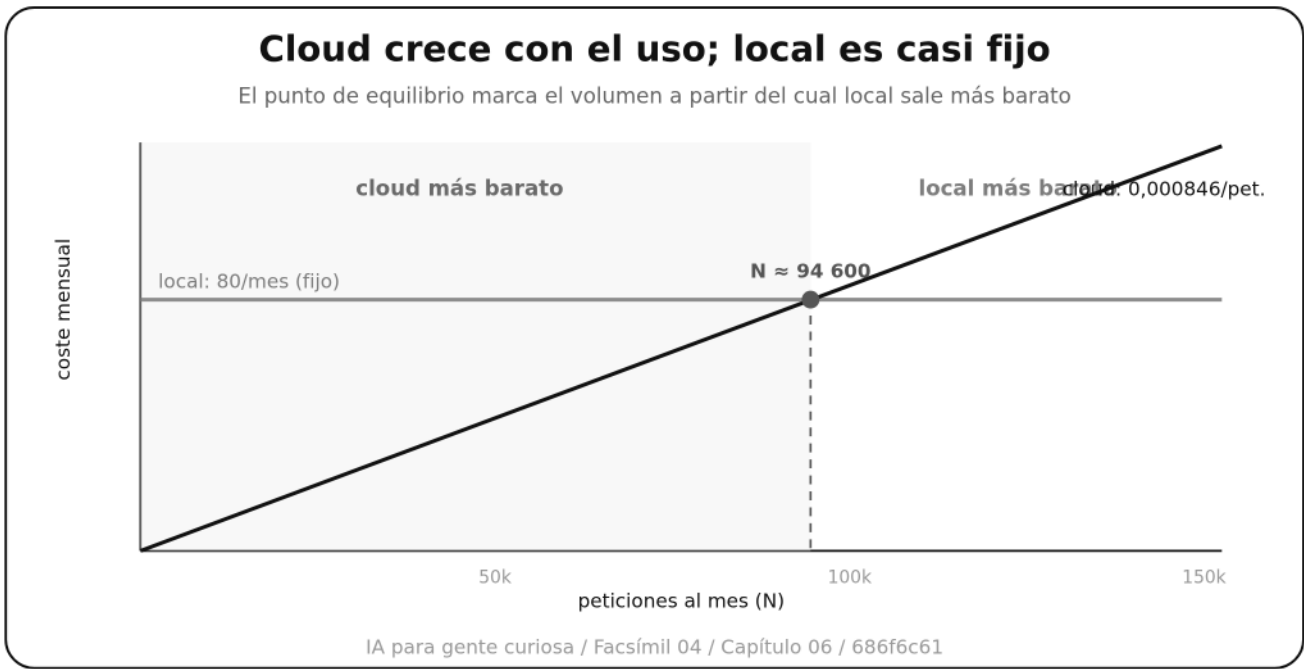
$$C_{\text{local}} = \frac{C_{\text{hardware}}}{M_{\text{amortizacion}}} + C_{\text{energia}} + C_{\text{operacion}} + C_{\text{mantenimiento}} + C_{\text{fallos}}$$

COSTE	CLOUD	LOCAL
Tokens	Visible y variable.	No se paga por token, pero sí por capacidad.
Hardware	Incluido en precio o instancia.	Compra, alquiler o servidor propio.
Picos	Elasticidad bajo demanda.	Necesitas sobredimensionar o aceptar cola.
Mantenimiento	Delegado en gran parte.	Drivers, runtime, modelos, disco, seguridad, monitorización.
Fallos	SLA, regiones, límites y dependencia externa.	Tú operas la pila.
Privacidad	Contratos y controles del proveedor.	Control físico/lógico, pero también responsabilidad propia.
Cambio de modelo	Más fácil si la API lo ofrece.	Depende de pesos, runtime y hardware disponible.

Como cloud crece con N y local es casi fijo, existe un **punto de equilibrio**: el volumen a partir del cual local sale más barato. Es el clásico análisis coste-volumen (break-even) de la contabilidad de gestión, donde el umbral es el coste fijo dividido entre el margen de contribución por unidad.²⁷ Se obtiene igualando ambas cuentas y despejando N :

$$N_{\text{equilibrio}} = \frac{C_{\text{local mensual}}}{\frac{T_{\text{entrada}}}{10^6} P_{\text{entrada}} + \frac{T_{\text{salida}}}{10^6} P_{\text{salida}}}$$

Pongamos números, los mismos que mide la práctica de este capítulo: un servidor local que cuesta 80 al mes y un coste cloud de 0,000846 por petición. El equilibrio está en $N_{\text{equilibrio}} = 80/0,000846 \approx 94\,600$ peticiones al mes. Por debajo de ese volumen, cloud es más barato; por encima, el coste fijo local se reparte entre tantas peticiones que gana. Por eso para 1200 peticiones al mes cloud es claramente más barato (estás muy por debajo del equilibrio), y solo cuando hablas de decenas de miles o millones de peticiones homogéneas empieza a compensar la infraestructura propia. Ese número no decide solo (faltan privacidad, latencia y operación), pero baja la conversación a tierra.



Tres decisiones que se confunden

Cuando alguien dice “lo hacemos local o cloud”, en realidad hay tres decisiones mezcladas:

DECISIÓN	OPCIONES	QUÉ PREGUNTA RESPONDE
Dónde corre la generación	API cloud, servidor propio, portátil, edge, híbrido.	¿Dónde se ejecuta el modelo que genera texto?
Dónde viven los datos	Base propia, documentos locales, vector store cloud, almacenamiento regional.	¿Dónde están los documentos antes y después de inferir?
Dónde se opera el producto	App local, backend propio, cloud gestionado, marketplace.	¿Quién escala, observa, actualiza y responde cuando falla?

Puedes tener generación cloud con datos minimizados localmente. Puedes tener generación local con logs sincronizados a una nube corporativa. Puedes tener embeddings locales y generación cloud. Puedes tener RAG cloud y clasificación local. Lo profesional es nombrar la arquitectura exacta.

Estrategias reales de despliegue

No hay dos caminos; hay una familia de estrategias. La pregunta no es “¿cloud o local?”, sino **qué capa quieres delegar y qué capa quieres controlar**.

ESTRATEGIA	QUÉ CONTROLAS	QUÉ DELEGAS	CUÁNDO ENCAJA	CUIDADO PRINCIPAL
API directa a un laboratorio	Prompt, contrato de salida, observabilidad de tu app.	Modelo, servidor, escalado, optimizaciones de inferencia.	Necesitas calidad alta, velocidad de desarrollo y modelos actuales.	Dependencia de precios, límites, región y política de datos.
Plataforma a cloud gestionada	Región, IAM, redes, trazabilidad cloud, despliegue corporativo.	Servir el modelo y actualizar infraestructura base.	Empresa con cloud ya gobernada, auditoría y compras centralizadas.	El contrato cloud no elimina tu responsabilidad de diseño.
Gateway de modelos	Un endpoint común, selección de modelo, fallback, comparación rápida.	Relación con muchos proveedores y normalización parcial de APIs.	Quieres probar modelos o tener plan B sin reescribir toda la app.	No todos los parámetros significan lo mismo en todos los modelos.
Ollama Cloud	Herramienta local, CLI/API Ollama, cambio suave desde modelos pequeños.	Ejecución de modelos cloud de Ollama cuando no caben en tu equipo.	Quieres seguir usando flujo Ollama con modelos más grandes.	“Uso Ollama” no siempre significa “el cálculo ocurre localmente”.
Servidor propio de inferencia	Runtime, modelo exacto, hardware, red, logs, versionado y costes fijos.	Poco: tú operas casi todo.	Volumen estable, datos cerca, requisitos offline o control fino.	Operación, capacidad, actualizaciones y degradación bajo carga.
Híbrida por flujo	Qué parte va local, qué parte va cloud, qué se cachea y qué se deriva.	Solo las piezas elegidas.	La mayoría de productos reales con distintas sensibilidades y costes.	Sin reglas explícitas se convierte en una mezcla difícil de depurar.

OpenRouter entra en la tercera categoría: no es un modelo, es un **router/gateway** con una API compatible en la que eliges modelos de distintos proveedores. Su endpoint de modelos publica campos como `id`, `context_length`, modalidades, precios y parámetros soportados. Eso sirve para hacer inventario, pero no sustituye tu evaluación: dos modelos con la misma ventana de contexto pueden comportarse distinto con JSON, herramientas, español, razonamiento o latencia.

Ollama Cloud es otra cosa. Ollama puede seguir pareciendo local desde tu terminal, pero los modelos marcados como cloud se ejecutan en Ollama Cloud para poder usar modelos que no caben en tu GPU. Esto es cómodo para probar, pero cambia la frontera de confianza y el coste: la interfaz local no garantiza inferencia local.

Cómo saber qué modelos cloud tienes de verdad

Nunca elijas un modelo copiando un nombre de una entrada antigua, una captura o una demo. El identificador de modelo es una dependencia de producción. Tiene versión, fecha, capacidades, precio, modalidad, límites y, a veces, política de retirada.

PROVEEDOR O GATEWAY	CÓMO INVENTARIARLO	QUÉ MIRAR ANTES DE USARLO
OpenAI	GET <code>https://api.openai.com/v1/models</code> con tu clave.	<code>id</code> , familia, endpoint soportado, precio actual, entrada multimodal, herramientas, structured outputs y modelo recomendado para tu tarea.
Anthropic	GET <code>https://api.anthropic.com/v1/models</code> con <code>anthropic-version</code> .	<code>id</code> , <code>display_name</code> , fecha de creación, ventana de contexto, coste, soporte de tools y modo de pensamiento si aplica.
Gemini API	Página de modelos y API de listado cuando trabajas con clave.	Si el nombre es estable, preview, latest o experimental; modalidades, herramientas, contexto, rate limits y fecha de retirada.
Bedrock o Vertex AI	Catálogo de modelos dentro de la región y cuenta.	Modelo disponible por región, precio por modalidad, IAM, logging, residencia y cuotas.
OpenRouter	GET <code>https://openrouter.ai/api/v1/models</code> .	<code>id</code> , proveedor, <code>context_length</code> , <code>pricing</code> , <code>supported_parameters</code> , modalidades y endpoints concretos.
Ollama local	GET <code>http://localhost:11434/api/tags</code> o <code>ollama list</code> .	Modelo descargado, tamaño, cuantización, fecha, template y si responde bien a tu contrato.
Ollama Cloud	Catálogo de modelos cloud y base URL <code>https://ollama.com/api</code> .	Si el modelo se ejecuta cloud, cuenta/API key, precio, límites y qué datos salen de tu máquina.
Servidor local OpenAI-compatible	GET <code>http://host:puerto/v1/models</code> si el runtime lo expone.	Nombre servido, plantilla de chat, límites de contexto, dtype, cuantización y parámetros aceptados.

Un inventario mínimo debería quedar así, aunque lo guardes en una hoja o en JSON:

CAMPO	EJEMPLO	POR QUÉ IMPORTA
provider	openai , anthropic , openrouter , local-vllm	Te dice quién opera la inferencia.
model_id	gpt-... , claude-... , meta-llama/...	Es la dependencia exacta de código.
endpoint	/v1/chat/completions , /v1/responses , /api/chat	No todos los modelos sirven en todos los endpoints.
context_tokens	128000 , 1000000 , 4096	Define cuánto texto puedes meter sin partir.
input_price y output_price	USD por millón de tokens	El coste de salida suele ser más alto que el de entrada.
modalities	texto, imagen, audio, embeddings	Evita elegir texto para un problema multimodal.
tools_json_schema	sí/no/parcial	Afecta agentes, validación y salidas estructuradas.
retention_region	política y región	Afecta cumplimiento y arquitectura.
version_policy	estable, preview, latest	Afecta reproducibilidad.
checked_at	2026-06-10	Hace explícito cuándo era verdad.

Comandos de inventario, no de producción:

```
# OpenAI: modelos accesibles por tu clave
curl https://api.openai.com/v1/models \
  -H "Authorization: Bearer $OPENAI_API_KEY"

# Anthropic: modelos accesibles por tu workspace
curl https://api.anthropic.com/v1/models \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01"

# OpenRouter: modelos, contexto, precios y parámetros soportados
curl https://openrouter.ai/api/v1/models \
  -H "Authorization: Bearer $OPENROUTER_API_KEY"

# Ollama local: modelos descargados en tu máquina
curl http://localhost:11434/api/tags

# Servidor local compatible con OpenAI, si lo expone
curl http://localhost:8000/v1/models \
  -H "Authorization: Bearer $LOCAL_API_KEY"
```

Lo importante es no mezclar “modelo que existe”, “modelo que mi cuenta puede usar”, “modelo que mi endpoint acepta” y “modelo que mi evaluación aprueba”. Son cuatro filtros distintos.

Servir inferencia local en serio

Bajar un GGUF, escribir `ollama run` y ver una respuesta es una prueba de vida. Un servidor de inferencia es otra cosa: es una pieza de infraestructura que debe cargar pesos, reservar memoria, aceptar peticiones, encolar trabajo, generar tokens, devolver errores comprensibles, medir latencia y sobrevivir a usos repetidos.

La memoria que reserva un servidor de inferencia no es el tamaño del fichero, sino el mismo presupuesto que vimos en el capítulo 5, ahora con el batch como variable de primer orden:

$$\text{memoria}_{\text{total}} \approx \text{pesos}_{\text{cuantizados}} + \text{KV cache}(L, T, B) + \text{runtime} + \text{margen}$$

Donde L son capas, T tokens de contexto, B peticiones simultáneas y la KV cache guarda claves y valores de atención para no recalcular todo en cada token. Por eso un modelo puede “caber” con una conversación corta y romperse cuando subes contexto, batch o concurrencia. En MoE tampoco basta mirar parámetros totales: importan parámetros activados por token, memoria de pesos, comunicación entre GPUs y eficiencia del runtime.

CAPA	DECISIÓN TÉCNICA	QUÉ MIRAR
Hardware	CPU, GPU, VRAM, RAM, disco NVMe, red.	VRAM útil, ancho de banda, drivers, consumo, refrigeración y margen.
Artefacto	GGUF, safetensors, AWQ/GPTQ/FP8/BF16, revisión exacta.	Licencia, checksum, tokenizer, chat template y versión fija.
Runtime	Ollama, llama.cpp, vLLM, SGLang, TGI, LM Studio.	Batching, KV cache, cuantización, multi-GPU, tools, JSON y compatibilidad OpenAI.
Configuración	<code>max_model_len</code> , dtype, batch, concurrencia, tensor parallel, cache.	Que el límite declarado sea sostenible con tus usuarios reales.
API	<code>/v1/chat/completions</code> , <code>/v1/embeddings</code> , <code>/api/chat</code> , streaming.	Contrato estable, errores, timeouts y parámetros aceptados.
Entrada	Plantilla de chat, system prompt, roles, documentos, imágenes.	Si la plantilla es incorrecta, el modelo parece peor de lo que es.
Operación	proceso, reinicio, logs, métricas, health checks, despliegue.	p50/p95/p99, TTFT, tokens/s, cola, VRAM, errores de JSON y coste eléctrico.
Acceso	bind de red, autenticación, TLS, rate limits, CORS.	No publiques <code>0.0.0.0</code> sin proxy, clave, límites y logs útiles.
Evolución	canary, rollback, evals, cambio de modelo.	Cambiar cuantización o template puede cambiar respuestas aunque el nombre parezca igual.

Runtimes habituales:

RUNTIME	MEJOR PARA	PUNTOS TÉCNICOS
Ollama	Desarrollo local, demos, herramientas personales, API sencilla.	Muy cómodo; distingue local de cloud cuando uses modelos cloud.
LM Studio	Exploración visual, pruebas con modelos descargados, endpoint local.	Bueno para aprender; no lo confundas con una plataforma multiusuario.
llama.cpp llama-server	GGUF, CPU/edge, GPU modesta, despliegues ligeros.	Expone servidor HTTP compatible, opciones de host/puerto, GPU offload y endpoints de chat/embeddings.
vLLM	Alto throughput en GPU, servicio multiusuario, OpenAI-compatible.	Continuous batching, KV cache eficiente, tensor parallel, cuantización y <code>--api-key</code> .
SGLang	Baja latencia, alto throughput, modelos grandes/multimodales.	Runtime optimizado, OpenAI API, RadixAttention, prefix caching y gateway.
Hugging Face TGI	Servir modelos HF con API REST y Messages API.	Streaming, tensor parallel, Prometheus/Grafana, despliegue cloud o propio.

Un arranque local mínimo con vLLM no debería terminar en “funciona”. Debería fijar modelo, dtype, contexto, nombre servido y clave:

```
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen3-8B-Instruct \
  --served-model-name local-qwen3-8b \
  --host 127.0.0.1 \
  --port 8000 \
  --dtype auto \
  --max-model-len 8192 \
  --gpu-memory-utilization 0.86 \
  --api-key "$LOCAL_API_KEY"
```

Y una prueba de vida técnica debería medir contrato, streaming y coste aproximado de tokens, no solo leer una respuesta bonita:

```
URL="http://127.0.0.1:8000/v1"
```

```
curl "$URL/chat/completions" \  
-H "Authorization: Bearer $LOCAL_API_KEY" \  
-H "Content-Type: application/json" \  
-d '{  
  "model": "local-qwen3-8b",  
  "messages": [  
    {"role": "system", "content": "Devuelve solo JSON valido."},  
    {"role": "user", "content": "Clasifica: acceso al campus virtual."}  
  ],  
  "temperature": 0.1,  
  "max_tokens": 160,  
  "stream": false  
'
```

Si esa llamada falla, no has “fallado con IA”; has descubierto una capa concreta: modelo no cargado, plantilla incorrecta, falta de VRAM, endpoint mal expuesto, clave ausente, límite de contexto, JSON inestable o timeout. Eso es mucho más útil que una demo.

Alquilar GPUs para inferencia

Entre una API gestionada y una GPU en tu mesa existe una opción muy usada: **alquilar GPU en cloud y servir tú el modelo**. Es cloud en infraestructura, pero local en responsabilidad técnica: eliges artefacto, runtime, contenedor, plantilla, métricas, límites y política de despliegue.

Hay tres formas habituales:

FORMA	QUÉ ALQUILAS	QUÉ CONTROLAS	CUÁNDO ENCAJA	RIESGO PRINCIPAL
VM GPU clásica	Una máquina con GPU, CPU, RAM, disco y red.	Sistema operativo, Docker, runtime, modelo, API y logs.	Servicio propio estable, pruebas serias, fine-tuning ligero o inferencia dedicada.	Pagas mientras está asignada, aunque esté esperando peticiones.
Pod GPU especializado	Una instancia GPU más directa, a menudo con plantillas o contenedor propio.	Imagen, volumen, librerías, runtime y arranque.	Probar modelos abiertos, levantar endpoints rápidos, trabajos por horas.	Disponibilidad por región/GPU y coste de almacenamiento persistente.
Serverless GPU o endpoint dedicado	Workers que arrancan bajo demanda o endpoint gestionado con GPU detrás.	Menos infraestructura; normalmente controlas imagen y escalado.	Tráfico irregular, demos, colas batch, picos previsibles.	Cold start, límites de ejecución, cola y menor control fino.

AWS EC2, Google Compute Engine y Azure permiten VMs con familias aceleradas por GPU. Runpod y proveedores similares simplifican el alquiler de pods con GPU y contenedores propios. Esto no sustituye a Bedrock, Vertex AI u OpenAI: es otra capa. En vez de comprar tokens de un modelo servido por otro, alquilas capacidad y corres tu pila de inferencia.

Alquilar una GPU se factura por hora, pero el coste real incluye las horas ociosas y los extras de operación. El coste mensual es:

$$C_{\text{gpu}} = H_{\text{activa}} P_{\text{hora}} + H_{\text{idle}} P_{\text{hora}} + C_{\text{volumen}} + C_{\text{egress}} + C_{\text{imagenes}} + C_{\text{operacion}}$$

Lo que de verdad importa para comparar con cloud es convertir ese precio por hora en **coste por token**, y ahí la utilización lo cambia todo. Si una GPU genera R tokens por segundo pero solo está ocupada una fracción U del tiempo, el coste por millón de tokens generados es:

$$C_{1M} \approx \frac{P_{\text{hora}}}{3600 \cdot R_{\text{tokens/s}} \cdot U} \cdot 10^6$$

Donde $R_{\text{tokens/s}}$ es la velocidad útil del servidor y U es la utilización real. Con números: una GPU a 2,50 por hora que genera 40 tokens/s sale a $C_{1M} = 2,50 / (3600 \cdot 40 \cdot 0,5) \cdot 10^6 \approx 34,7$ por millón de tokens si está ocupada la mitad del tiempo ($U = 0,5$). Si solo está ocupada el 10 por ciento ($U = 0,1$), ese mismo hardware cuesta **173,6** por millón: cinco veces más. Este es el punto que más se olvida: una GPU barata por hora puede ser carísima por token si está casi vacía. Por eso el alquiler de GPU solo bate al precio por token de una API cuando consigues mantener una utilización alta y sostenida.

Para inferencia, la GPU no se elige por nombre bonito:

NECESIDAD	GPU O ACELERADOR TÍPICO	POR QUÉ
Modelo 7B-13B cuantizado, baja concurrencia	L4, A10, RTX 4090/5090, RTX 6000, T4 si aceptas menor margen.	Suele bastar para pilotos, herramientas internas y modelos pequeños.
Modelo 30B-70B, contexto mayor o más usuarios	A100 80GB, H100, H200, B200 o multi-GPU.	Más VRAM, ancho de banda y margen para KV cache.
Inferencia muy optimizada en AWS	Inf2 u otros aceleradores específicos.	Pueden ser eficientes, pero exigen stack y compilación propios.
Tráfico irregular	Serverless GPU o workers autoscalados.	Pagas menos espera, pero introduces arranque en frío.
Tráfico estable	GPU dedicada con reserva o compromiso.	Mejora disponibilidad y coste si sabes que la usarás muchas horas.

El tamaño de modelo no basta. Para servir bien necesitas medir:

SEÑAL	QUÉ MIDE	DECISIÓN
VRAM libre tras cargar pesos	Margen para KV cache y batch.	Si queda poco margen, baja contexto, cuantización o concurrencia.
TTFT	Tiempo hasta primer token.	Si es alto, revisa cola, prefill, cold start o modelo demasiado grande.
tokens/s por petición	Velocidad de generación.	Compara runtime, cuantización y GPU.
throughput agregado	Tokens/s o peticiones/s con concurrencia.	Sirve para decidir batch, réplicas y autoscaling.
utilización de GPU	Porcentaje real de uso.	Si es baja, estás pagando idle; si es alta, aparecerá cola.
errores por memoria	Peticiones que fallan por VRAM/contexto.	Define límites de entrada y concurrencia.
tiempo de arranque	Descargar imagen, montar volumen, cargar pesos.	Crítico en serverless y pods efímeros.

Checklist mínimo antes de usar GPU alquilada para una API de inferencia:

- Imagen Docker reproducible con CUDA, runtime, versión de Python y dependencias fijadas.

- Modelo y tokenizer en volumen persistente o cache precalentada; no descargar 80 GB en cada arranque.
- Health check que no diga “vivo” hasta haber cargado el modelo.
- Warmup con una petición corta para inicializar kernels, plantilla y cache.
- Límite de contexto y `max_tokens` por endpoint, no solo por buena voluntad del cliente.
- Autenticación delante del runtime, aunque sea interno.
- Métricas de p50, p95, p99, TTFT, tokens/s, cola, VRAM y errores.
- Política de apagado: deallocated/delete cuando no se usa; “stopped” no siempre significa coste cero según proveedor.
- Plan de fallback: otro modelo, otra región, API gestionada o cola de espera.

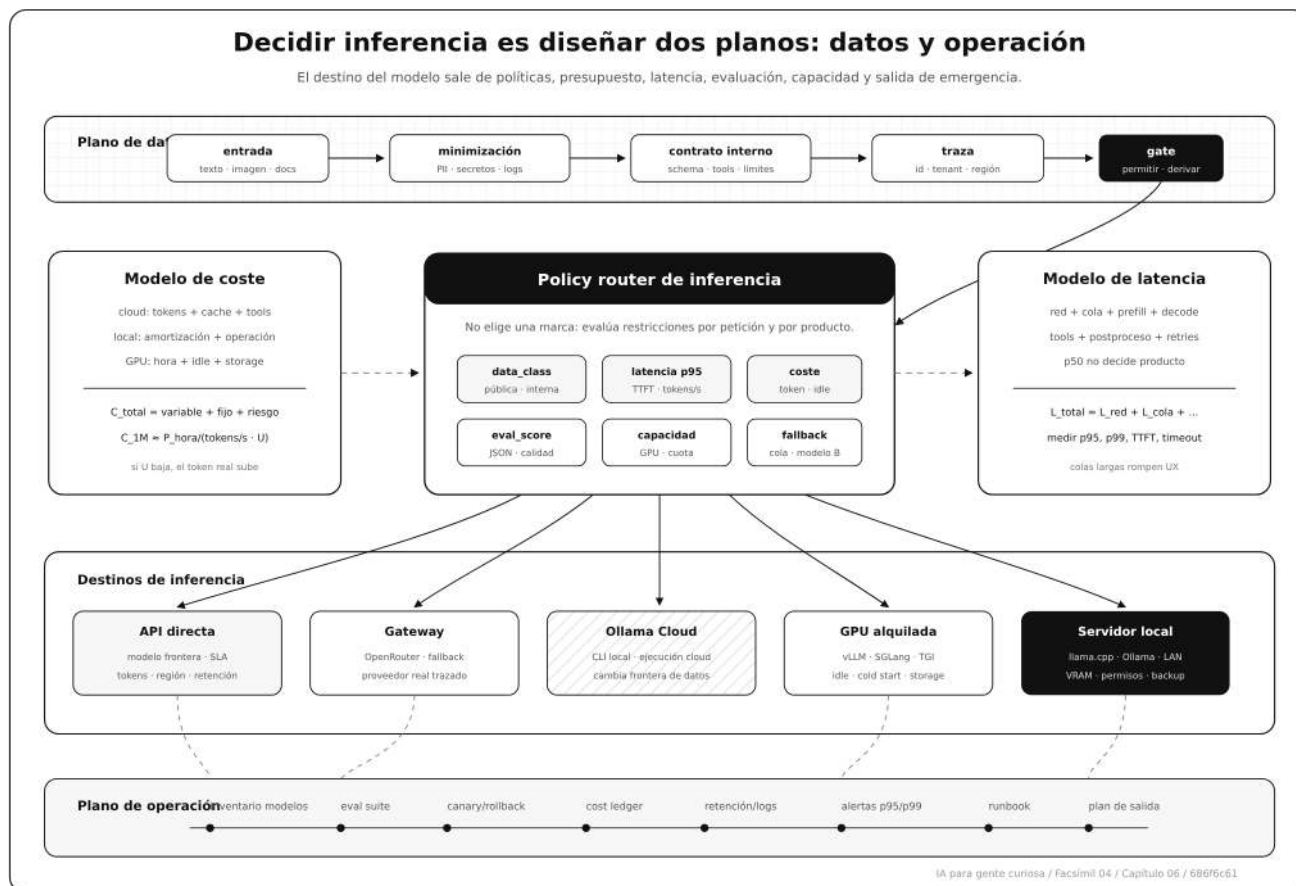
La regla práctica: si vas a usar una GPU alquilada como API, trátala como producto en producción desde el minuto uno. Si solo la enciendes para experimentar, trátala como laboratorio caro y pon alarma de apagado.

Cuándo elegir cada ruta

SITUACIÓN	ruta PROBABLE	POR QUÉ
Prototipo rápido con usuarios internos	Cloud o LM Studio local.	Aprendes rápido sin comprar infraestructura.
Datos sensibles y flujo simple	Local o cloud con controles contractuales fuertes.	La decisión depende de política, no de marca.
Mucha concurrencia variable	Cloud.	La elasticidad suele compensar.
Volumen estable y tarea acotada	Local/propio puede competir.	Puedes amortizar hardware y optimizar.
Necesitas modelo frontera	Cloud.	Los mejores modelos no suelen estar todos como pesos descargables.
Offline, aula, demo, entorno cerrado	Local.	Funciona sin depender de red externa.
Latencia de milisegundos cerca del usuario	Local/edge si el modelo cabe.	La distancia de red importa.
Cumplimiento con región definida	Cloud regional o local controlado.	Se decide por residencia, auditoría y contrato.
Comparar muchos modelos sin reescribir clientes	Gateway como OpenRouter.	Normaliza entrada y permite inventariar precios, contexto y parámetros.
Usar herramientas Ollama con modelos que no caben	Ollama Cloud.	Mantienes flujo Ollama pero cambias la ubicación real de inferencia.
Servir modelos abiertos con control técnico	GPU alquilada con vLLM, SGLang, TGI o llama.cpp.	Controlas artefacto, runtime y API sin comprar hardware.
Tráfico con picos y largas pausas	Serverless GPU o endpoint autoscalado.	Puede reducir idle, a cambio de cold start y menos control fino.

La ruta híbrida es común: clasificación local, RAG con datos propios, generación cloud para casos difíciles, cache de respuestas frecuentes y fallback local si la API externa no está disponible. Híbrido no significa improvisado; significa que cada pieza tiene una razón.

Mapa visual de decisión



En el día a día

Imagina una universidad que quiere clasificar incidencias de estudiantes. Si el texto contiene solo categorías generales, una API cloud puede dar calidad alta y mantenimiento bajo. Si el texto incluye expedientes, datos médicos o información contractual, quizá convenga anonimizar, resumir localmente o ejecutar todo en un entorno controlado.

Imagina una asesoría que analiza miles de facturas al mes. Si el volumen es bajo, una API potente evita comprar hardware. Si el volumen es constante, el modelo local puede ahorrar coste. Pero si cada error de extracción cuesta una llamada humana, el coste real no está en tokens: está en fallos.

Imagina una app de escritorio que debe funcionar en una fábrica sin red estable. Local gana por disponibilidad. Pero si el modelo debe razonar sobre documentos complejos y actualizados, quizá necesites sincronizar, cachear o derivar algunos casos a cloud. La arquitectura buena suele ser menos épica y más concreta.

Por qué debería importarte

Porque la elección local/cloud afecta a producto, privacidad, presupuesto, operación y experiencia de usuario. No es una decisión que pueda tomar solo compras, solo legal o solo ingeniería. Hay que poner a todos mirando la misma tabla.

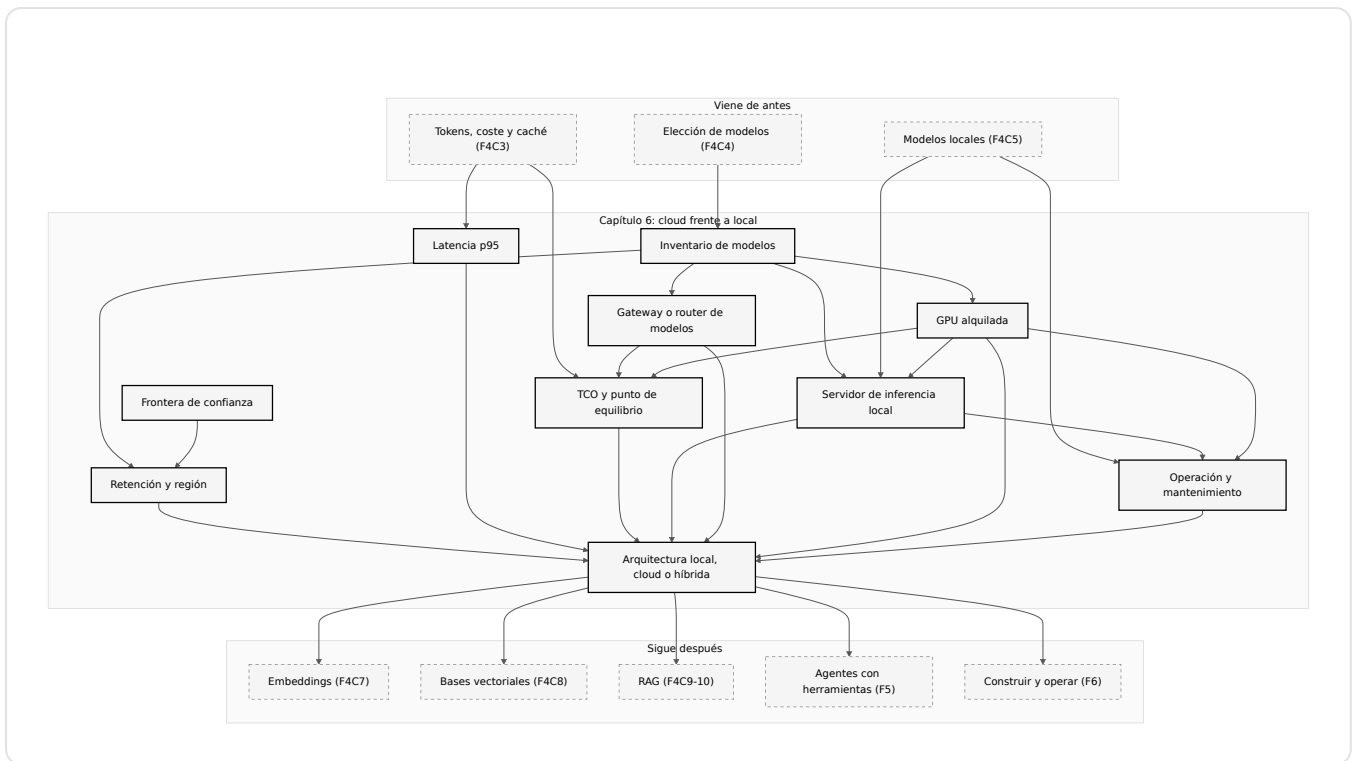
También importa porque los próximos capítulos del facsímil se apoyan en esta decisión. Los embeddings, las bases vectoriales, el RAG y las herramientas de datos pueden correr local, cloud o híbrido. Si no sabes elegir ubicación, cada capítulo posterior se convierte en una colección de piezas sin arquitectura.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Decir “local” como sinónimo de seguro	Local también tiene logs, backups, puertos y permisos.	Dibujar la frontera de confianza completa.
Decir “cloud” como sinónimo de caro	Pocas peticiones o modelos frontera pueden ser más baratos por API que operar hardware.	Calcular coste mensual, no coste emocional.
Comparar latencia media	El promedio oculta esperas largas.	Medir p50, p95, p99 y timeouts.
Olvidar el coste de operación local	Drivers, actualizaciones, disco, monitorización y guardias cuestan tiempo.	Incluir horas humanas y mantenimiento en el TCO.
No mirar la retención por endpoint	Distintas funciones pueden guardar estado de forma distinta.	Revisar documentación y contrato antes de enviar datos reales.
Construir sin plan de salida	Cambiar de proveedor o runtime tarde puede doler mucho.	Mantener pruebas comunes y contrato de API propio.
Creer que un gateway es un modelo	OpenRouter, por ejemplo, puede enrutar a modelos y proveedores distintos.	Guardar modelo, proveedor, endpoint, precio, fecha y parámetros soportados.
Confundir Ollama con inferencia local siempre	Ollama Cloud mantiene la experiencia de herramienta local, pero ejecuta modelos cloud.	Mirar si el modelo está descargado localmente o se ejecuta en cloud.
Montar servidor local sin contrato operativo	Una respuesta en terminal no mide colas, contexto, JSON, p95 ni reinicios.	Definir runtime, API, clave, métricas, límites, salud y rollback.
Alquilar GPU y olvidar el idle	Una GPU por hora puede salir cara si espera vacía.	Calcular coste por token útil con utilización real.
Confundir stopped con deallocated	En algunos clouds parar una VM no libera todo el coste asignado.	Revisar estado facturable, discos, IPs, snapshots y volúmenes.

Cómo encaja todo

Este mapa conecta la decisión cloud/local con lo que ya vimos y con lo que viene. Fíjate en que no sale de “gusto por herramientas”, sino de restricciones medibles.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Frontera de confianza	Punto o cadena donde los datos pasan a otro dominio técnico o contractual.
TCO	Coste total de propiedad, incluyendo uso, infraestructura, operación y mantenimiento.
Latencia p95	Tiempo por debajo del cual termina el 95 por ciento de peticiones.
TTFT	Tiempo hasta recibir el primer token.
Throughput	Capacidad de procesar peticiones o tokens por unidad de tiempo.
Región	Ubicación geográfica donde se procesa o guarda una carga.
Retención	Tiempo durante el que se conservan datos, logs o estado.
Residencia de datos	Restricción sobre dónde deben vivir o procesarse los datos.
Capacidad elástica	Capacidad de escalar recursos bajo demanda.
Punto de equilibrio	Volumen donde coste local y cloud se igualan bajo hipótesis dadas.
Gateway de modelos	Capa que ofrece un endpoint común y enruta peticiones a distintos proveedores o modelos.
Servidor de inferencia	Servicio que carga pesos, gestiona memoria, cola peticiones y expone una API para generar salidas.
GPU alquilada	Acelerador en cloud que pagas por tiempo para ejecutar tu propio runtime o contenedor.
KV cache	Memoria usada para guardar claves y valores de atención ya calculados durante la generación.
Modelo servido	Nombre de modelo que expone tu API; puede ser distinto del repositorio o archivo interno.

Antes de pasar página

☐ ¿Puedo explicar por qué local no significa automáticamente privado? (Si no, vuelve a «Qué no significa “privado”».)

- ☐ ¿Puedo explicar por qué cloud no significa automáticamente caro? (Si no, vuelve a «Coste: token barato no significa sistema barato».)
- ☐ ¿Sé dibujar la frontera de confianza de un flujo completo? (Si no, vuelve a «La frontera de confianza».)
- ☐ ¿Sé inventariar modelos disponibles por endpoint y no por memoria? (Si no, vuelve a «Cómo saber qué modelos cloud tienes de verdad».)
- ☐ ¿Sé distinguir API directa, plataforma cloud, gateway, Ollama Cloud y servidor propio? (Si no, vuelve a «Estrategias reales de despliegue».)
- ☐ ¿Sé calcular coste cloud con entrada, salida, cache y extras? (Si no, vuelve a «Coste: token barato no significa sistema barato».)
- ☐ ¿Sé calcular coste local incluyendo hardware, energía y operación? (Si no, vuelve a «Servir inferencia local en serio».)
- ☐ ¿Sé calcular coste de GPU alquilada incluyendo idle, storage, red y operación? (Si no, vuelve a «Alquilar GPUs para inferencia».)
- ☐ ¿Estoy midiendo p95 y no solo promedio? (Si no, vuelve a «Latencia: no mires solo el promedio».)
- ☐ ¿Sé qué datos se retienen y durante cuánto tiempo en el endpoint elegido? (Si no, vuelve a «La frontera de confianza».)
- ☐ ¿Sé qué runtime, contexto, cuantización y nombre servido tiene mi servidor local? (Si no, vuelve a «Servir inferencia local en serio».)
- ☐ ¿Sé comparar local y cloud con el mismo caso y los mismos criterios? (Si no, repasa «Cuándo elegir cada ruta».)

En resumen

IDEA FUERZA	DETALLE
Local y cloud son posiciones arquitectónicas.	No son bandos; responden a restricciones distintas.
Privacidad exige dibujar la frontera de confianza.	Hay que saber dónde existe el dato, quién accede y cuánto se retiene.
La latencia útil se mide en p95 y p99.	El promedio no captura la experiencia de usuarios lentos.
El coste real es TCO.	Tokens, hardware, operación, mantenimiento, cache y herramientas cuentan.
Un gateway no es un modelo.	OpenRouter u otros routers pueden cambiar proveedor, parámetros y precio detrás de un endpoint común.
Ollama Cloud no equivale a inferencia local.	Conserva la experiencia Ollama, pero cambia ubicación, coste y frontera de datos.
Servir local en serio requiere infraestructura.	Modelo, runtime, KV cache, API, claves, métricas, colas, límites y rollback forman parte del sistema.
Alquilar GPU es cloud con responsabilidad propia.	Pagas tiempo, storage e idle; tú sirves, mides, actualizas y apagas.
La ruta híbrida suele ser la más realista.	Minimiza datos localmente, escala cloud cuando aporta y mantiene una eval común.

Para saber más

Amazon Web Services. (2026). *Amazon Bedrock pricing*.

<https://aws.amazon.com/bedrock/pricing/>

Amazon Web Services. (2026). *Data protection in Amazon Bedrock*.

<https://docs.aws.amazon.com/bedrock/latest/userguide/data-protection.html>

Amazon Web Services. (2026). *Amazon EC2 Pricing*. <https://aws.amazon.com/ec2/pricing/>

Amazon Web Services. (2026). *Specifications for Amazon EC2 accelerated computing instances*. <https://docs.aws.amazon.com/ec2/latest/instancetypes/ac.html>

Anthropic. (2026). *API and data retention*. <https://platform.claude.com/docs/en/manage-claude/api-and-data-retention>

Anthropic. (2026). *List Models*. <https://platform.claude.com/docs/en/api/models/list>

Anthropic. (2026). *Pricing*. <https://platform.claude.com/docs/en/about-claude/pricing>

Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. *Communications of the ACM*, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>

Google. (2026). *Gemini API models*. <https://ai.google.dev/gemini-api/docs/models>

Google Cloud. (2026). *About GPU instances*. <https://docs.cloud.google.com/compute/docs/gpus/about-gpus>

Google Cloud. (2026). *GPU machine types*. <https://docs.cloud.google.com/compute/docs/gpus>

Google Cloud. (2026). *Vertex AI and zero data retention*. <https://cloud.google.com/vertex-ai/generative-ai/docs/data-governance>

Google Cloud. (2026). *Vertex AI pricing*. <https://cloud.google.com/vertex-ai/generative-ai/pricing>

Hugging Face. (2026). *Text Generation Inference: HTTP API Reference*. https://huggingface.co/docs/text-generation-inference/reference/api_reference

llama.cpp. (2026). *llama-server*. <https://www.mintlify.com/ggml-org/llama.cpp/api/tools/llama-server>

Microsoft Azure. (2026). *Linux Virtual Machines Pricing*. <https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/>

Microsoft Azure. (2026). *Virtual machine sizes overview*. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview>

Ollama. (2026). *API introduction*. <https://docs.ollama.com/api/introduction>

Ollama. (2026). *Cloud*. <https://docs.ollama.com/cloud>

Ollama. (2026). *OpenAI compatibility*. <https://docs.ollama.com/api/openai-compatibility>

OpenAI. (2026). *Data controls in the OpenAI platform*. <https://developers.openai.com/api/docs/guides/your-data>

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

OpenAI. (2026). *List models*. <https://developers.openai.com/api/reference/resources/models/methods/list>

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

OpenRouter. (2026). *List all models and their properties*. <https://openrouter.ai/docs/api/api-reference/models/get-models>

OpenRouter. (2026). *Provider routing*. <https://openrouter.ai/docs/guides/routing/provider-selection>

Runpod. (2026). *Cloud GPU Instances for AI Workloads*. <https://www.runpod.io/product/cloud-gpus>

Runpod. (2026). *Pods pricing*. <https://docs.runpod.io/pods/pricing>

Shi, W., Cao, J., Zhang, Q., Li, Y. y Xu, L. (2016). *Edge Computing: Vision and Challenges*. *IEEE Internet of Things Journal*, 3(5), 637-646. <https://doi.org/10.1109/JIOT.2016.2579198>

SGLang. (2026). *Welcome to SGLang*. <https://docs.sglang.io/index.html>

vLLM. (2026). *OpenAI-Compatible Server*. https://docs.vllm.ai/en/stable/serving/openai_compatible_server/

Notas

1. OpenAI. (2026). *Data controls in the OpenAI platform*. <https://developers.openai.com/api/docs/guides/your-data>. Consultado el 10 de junio de 2026. ↵
2. OpenAI. (2026). *List models*. <https://developers.openai.com/api/reference/resources/models/methods/list>. Consultado el 10 de junio de 2026. ↵
3. OpenAI. (2026). *Pricing*. <https://developers.openai.com/api/docs/pricing>. Consultado el 10 de junio de 2026. ↵
4. OpenAI. (2026). *Latency optimization*. <https://developers.openai.com/api/docs/guides/latency-optimization>. Consultado el 10 de junio de 2026. ↵
5. Anthropic. (2026). *API and data retention*. <https://platform.claude.com/docs/en/manage-claude/api-and-data-retention>. Consultado el 10 de junio de 2026. ↵
6. Anthropic. (2026). *List Models*. <https://platform.claude.com/docs/en/api/models/list>. Consultado el 10 de junio de 2026. ↵
7. Anthropic. (2026). *Pricing*. <https://platform.claude.com/docs/en/about-claude/pricing>. Consultado el 10 de junio de 2026. ↵

8. Google. (2026). *Gemini API models*. <https://ai.google.dev/gemini-api/docs/models>. Consultado el 10 de junio de 2026. ↵
9. Google Cloud. (2026). *Vertex AI and zero data retention*. <https://cloud.google.com/vertex-ai/generative-ai/docs/data-governance>. Consultado el 10 de junio de 2026. ↵
10. Google Cloud. (2026). *Vertex AI pricing*. <https://cloud.google.com/vertex-ai/generative-ai/pricing>. Consultado el 10 de junio de 2026. ↵
11. Amazon Web Services. (2026). *Data protection in Amazon Bedrock*. <https://docs.aws.amazon.com/bedrock/latest/userguide/data-protection.html>. Consultado el 10 de junio de 2026. ↵
12. Amazon Web Services. (2026). *Amazon Bedrock pricing*. <https://aws.amazon.com/bedrock/pricing/>. Consultado el 10 de junio de 2026. ↵
13. OpenRouter. (2026). *List all models and their properties*. <https://openrouter.ai/docs/api/api-reference/models/get-models>. Consultado el 10 de junio de 2026. ↵
14. OpenRouter. (2026). *Provider routing*. <https://openrouter.ai/docs/guides/routing/provider-selection>. Consultado el 10 de junio de 2026. ↵
15. Ollama. (2026). *Cloud*. <https://docs.ollama.com/cloud>. Consultado el 10 de junio de 2026. Véase también *API introduction*: <https://docs.ollama.com/api/introduction> y *OpenAI compatibility*: <https://docs.ollama.com/api/openai-compatibility>. ↵
16. vLLM. (2026). *OpenAI-Compatible Server*. https://docs.vllm.ai/en/stable/serving/openai_compatible_server/. Consultado el 10 de junio de 2026. llama.cpp. (2026). *llama-server*. <https://www.mintlify.com/ggml-org/llama.cpp/api/tools/llama-server>. Hugging Face. (2026). *TGI HTTP API Reference*. https://huggingface.co/docs/text-generation-inference/reference/api_reference. SGLang. (2026). *Welcome to SGLang*. <https://docs.sglang.io/index.html>. ↵
17. AWS. (2026). *Amazon EC2 Pricing*. <https://aws.amazon.com/ec2/pricing/>. Consultado el 10 de junio de 2026. AWS. (2026). *Specifications for Amazon EC2 accelerated computing instances*. <https://docs.aws.amazon.com/ec2/latest/instancetype/ac.html>. Google Cloud. (2026). *GPU machine types*. <https://docs.cloud.google.com/compute/docs/gpus>. Google Cloud. (2026). *About GPU instances*. <https://docs.cloud.google.com/compute/docs/gpus/about-gpus>. ↵
18. Microsoft Azure. (2026). *Virtual machine sizes overview*. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview>. Microsoft Azure. (2026). *Linux Virtual Machines Pricing*. <https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/>. Runpod. (2026). *Pods pricing*. <https://docs.runpod.io/pods/pricing>. Runpod. (2026). *Cloud GPU Instances for AI Workloads*. <https://www.runpod.io/product/cloud-gpus>. Consultado el 10 de junio de 2026. ↵
19. Jeffrey Dean y Luiz André Barroso. (2013). *The Tail at Scale*. *Communications of the ACM*, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>. ↵

- !0. Weisong Shi y otros (2016). *Edge Computing: Vision and Challenges*. *IEEE Internet of Things Journal*, 3(5), 637-646. <https://doi.org/10.1109/IJOT.2016.2579198>. ↵
- !1. Fuentes de esta revisión: OpenAI. (2026). *Cost optimization*. <https://developers.openai.com/api/docs/guides/cost-optimization>. OpenAI. (2026). *Model selection*. <https://developers.openai.com/api/docs/guides/model-selection>. Anthropic. (2026). *Usage and Cost Admin API*. <https://platform.claude.com/docs/en/manage-claude/usage-cost-api>. Anthropic. (2026). *Data residency*. <https://platform.claude.com/docs/en/manage-claude/data-residency>. Amazon Web Services. (2026). *Data protection in Amazon Bedrock*. <https://docs.aws.amazon.com/bedrock/latest/userguide/data-protection.html>. Google Cloud. (2026). *Vertex AI Generative AI release notes*. <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/release-notes>. Google Cloud. (2026). *Vertex AI Generative AI SLA*. <https://cloud.google.com/vertex-ai/generative-ai/sla>. Todas consultadas el 10 de junio de 2026. ↵
- !2. John D. C. Little. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . *Operations Research*, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>. ↵
- !3. Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. <https://arxiv.org/abs/2309.06180>. ↵
- !4. Yinmin Zhong y otros (2024). *DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving*. OSDI 2024. <https://arxiv.org/abs/2401.09670>. ↵
- !5. Jeffrey Dean y Luiz André Barroso. (2013). *The Tail at Scale*. *Communications of the ACM*, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>. ↵
- !6. Lisa M. Ellram. (1995). *Total Cost of Ownership: An Analysis Approach for Purchasing*. *International Journal of Physical Distribution & Logistics Management*, 25(8), 4-23. <https://doi.org/10.1108/09600039510099928>. ↵
- !7. Charles T. Horngren, Srikant M. Datar y Madhav V. Rajan. (2015). *Cost Accounting: A Managerial Emphasis* (15.^a ed.). Pearson. Capítulo sobre análisis coste-volumen-beneficio. ↵

Capítulo 07: Embeddings aplicados y búsqueda semántica

Cuando las palabras exactas no bastan

Imagina que alguien busca en una intranet: “no puedo entrar al campus virtual”. El documento que resuelve el problema quizá se titula “Restablecer acceso a Moodle con doble factor”. No comparte demasiadas palabras con la consulta, pero para una persona está claro que hablan de lo mismo.

La búsqueda clásica por palabras exactas funciona muy bien cuando el usuario sabe cómo se llama algo. Falla más cuando el usuario describe una necesidad, usa sinónimos, escribe con otro registro o mezcla conceptos. Aquí entran los embeddings: convertir textos en vectores para poder buscar por cercanía aproximada de significado.

Venimos del [capítulo 03](#), donde hablamos de tokens, contexto y coste, y del [capítulo 06](#), donde decidimos dónde ejecutar modelos. Ahora usamos esa base para montar la pieza que alimenta RAG, memoria de producto, recomendadores y buscadores internos.

Estado del arte con fecha de corte

Fecha de corte: 25 de mayo de 2026.

Fuentes consultadas ese día: documentación oficial de embeddings de OpenAI, Gemini, Cohere, Voyage AI y Sentence Transformers; y trabajos académicos sobre sentence embeddings, búsqueda vectorial, HNSW, FAISS y evaluación de recuperación.

Lo estable es el mecanismo: un modelo convierte entradas en vectores, esos vectores se comparan con una métrica y el sistema devuelve un ranking. Lo cambiante son modelos disponibles, dimensiones, precios, límites de contexto, soporte multimodal, compresión, tipos de salida, librerías y benchmarks.

FUENTE	QUÉ APORTA	CÓMO USARLA
OpenAI embeddings. 1	Explica embeddings como vectores de números y enumera usos como búsqueda, clustering, recomendaciones y clasificación.	Para entender el contrato básico: texto entra, vector sale, distancia mide relación.
text-embedding-3-large. 2	Documenta un modelo de embedding concreto y su ficha de modelo.	Para no hablar de “OpenAI embeddings” como si fuera un único artefacto.
Gemini embeddings. 3	Muestra cómo generar embeddings en Gemini API y usar salida vectorial para recuperación.	Para comparar API, dimensiones, tareas y límites.
Cohere embeddings. 4	Introduce <code>input_type</code> , soporte multilingüe, embeddings de imagen, contenido mixto, Matryoshka y compresión.	Para recordar que query y documento pueden tratarse de forma distinta.
Voyage embeddings. 5	Lista modelos, dimensiones, contexto e <code>input_type</code> para query/document.	Para elegir modelos orientados a retrieval, código, legal, finanzas o uso general.
Sentence Transformers. 6	Ofrece una forma local y reproducible de generar embeddings y hacer búsqueda semántica.	Para aprender el mecanismo sin depender de una API externa.
Sentence-BERT. 7	Populariza embeddings de frases eficientes para similitud semántica.	Para ver por qué no basta usar cualquier vector interno de un modelo.
FAISS. 8	Muestra técnicas de búsqueda eficiente de vectores a gran escala, especialmente en GPU.	Para entender por qué una base vectorial no compara siempre todo contra todo.
HNSW. 9	Describe un índice por grafo muy usado para vecinos aproximados.	Para entender el intercambio entre rapidez, memoria y exactitud.
BEIR. 10	Propone evaluar retrieval en tareas diversas, no solo en un dataset cómodo.	Para no validar un buscador con tres ejemplos elegidos a mano.

Qué no es un embedding

Un embedding no es una traducción secreta del texto. No contiene una definición legible de cada palabra. No es una base de datos comprimida con todos los documentos. Tampoco es una garantía de verdad: dos textos pueden estar cerca en el espacio vectorial y aun así no responder la pregunta correcta.

Tampoco es “memoria” por sí mismo. Guardar embeddings de documentos permite recuperar fragmentos parecidos, pero el sistema no entiende permisos, vigencia, autoría ni contexto de negocio a menos que tú lo diseñes. Un vector no sabe si un reglamento está derogado.

Y un embedding no sustituye a la búsqueda por filtros. Si el usuario pregunta por “normativa de matrícula 2025” y tu sistema devuelve un documento semánticamente parecido de 2021, el vector ha hecho parte del trabajo; falta metadata, filtros y evaluación.

Qué sí es: una coordenada útil

Un modelo de embeddings es una función que mapea un texto a un vector denso de d dimensiones. La idea de representar el significado como vectores viene de los word embeddings y hoy se extiende a frases y documentos completos con modelos tipo Sentence-BERT, que producen embeddings comparables directamente con similitud coseno:¹¹

$$e = f_{\theta}(x) \in \mathbb{R}^d$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Entrada que queremos representar.	“Restablecer acceso al campus virtual”.
f_{θ}	Modelo de embeddings con parámetros aprendidos.	Un modelo de Sentence Transformers o una API de embeddings.
e	Vector resultante.	[0,12, -0,03, 0,44, ...].
d	Número de dimensiones del vector.	384, 768, 1024, 1536 o 3072 según modelo.
$\mathbb{R}^d$	Espacio de vectores reales de dimensión d .	Una tabla con d columnas numéricas.

La intuición: textos que el modelo considera parecidos quedan cerca. Textos que el modelo considera distintos quedan lejos. Esa cercanía no aparece porque el modelo “sepa” como una persona; aparece porque durante entrenamiento aprendió a colocar ejemplos relacionados cerca y ejemplos no relacionados más lejos.

En búsqueda semántica hacemos lo mismo con documentos y consultas:

$$q = f_{\theta}(\text{consulta})$$

$$d_i = f_{\theta}(\text{documento}_i)$$

Después comparamos q con cada d_i y ordenamos.

Cómo se aprende esa geometría: entrenamiento contrastivo

Esa colocación no es magia ni se programa a mano: se aprende con un objetivo **contrastivo**. Al modelo se le muestran pares positivos (una consulta y su documento correcto) y se le pide acercarlos en el espacio, mientras aleja los negativos (otros documentos del lote). La función de pérdida habitual es InfoNCE, que maximiza la similitud del par correcto frente a todos los demás candidatos del batch:¹²

$$\mathcal{L} = -\log \frac{\exp(\text{sim}(q, d^+)/\tau)}{\sum_j \exp(\text{sim}(q, d_j)/\tau)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\mathcal{L}$	Pérdida a minimizar: baja cuando el par correcto domina sobre los demás.	Tiende a 0 si el documento correcto es, con mucho, el más similar.
q	Vector de la consulta (o del ancla).	Embedding de “no puedo entrar al campus”.
d^+	Documento positivo: el correcto para esa consulta.	“Restablecer acceso al campus virtual”.
d_j	Cada candidato del lote, positivo y negativos.	Los demás documentos del batch, incluidos los hard negatives.
$\text{sim}(\cdot, \cdot)$	Similitud entre dos vectores, normalmente el coseno.	$\text{sim}(q, d^+) = 0,82$.
τ	Temperatura: escala las similitudes antes del softmax.	0,05. Más baja hace la pérdida más exigente con los negativos cercanos.
$\sum_j$	Suma sobre todos los candidatos del denominador.	Recorre el positivo y todos los negativos del lote.

La expresión es un softmax sobre similitudes: el numerador premia que el positivo sea muy similar a la consulta, y el denominador penaliza que los negativos también lo sean. Para un ingeniero esto tiene tres consecuencias prácticas: primero, los **hard negatives** (documentos casi idénticos pero incorrectos) son el material que más enseña, por eso conviene incluirlos al afinar y al evaluar; segundo, un modelo solo separa bien lo que vio separar durante su entrenamiento, así que un embedding generalista puede flojear en tu dominio; y tercero, como la geometría depende de los pesos θ , cambiar de modelo cambia el espacio entero y **obliga a reindexar**.

Qué significa la dimensión

La dimensión de un embedding es el número de componentes del vector. Si un modelo devuelve un embedding de dimensión 384, cada texto se convierte en 384 números. Si devuelve 3072, cada texto se convierte en 3072 números. No es una “nota de inteligencia”; es el ancho de la representación.

Piensa en una tabla. Cada fila es un texto y cada columna es una coordenada aprendida por el modelo:

$$e = [e_1, e_2, e_3, \dots, e_d]$$

PIEZA	QUÉ SIGNIFICA	EJEMPLO
e	Embedding completo de un texto.	El vector de “no puedo entrar al campus”.
e_1, e_2, e_3	Primeras coordenadas del vector.	0.12 , -0.31 , 0.08 .
d	Número total de coordenadas.	384 columnas numéricas.

En un ejemplo pequeño de cuatro dimensiones, dos textos podrían quedar así:

TEXTO	E_1	E_2	E_3	E_4
“acceso al campus”	0,20	0,81	-0,10	0,33
“entrar en Moodle”	0,18	0,77	-0,08	0,29
“calendario de matrícula”	-0,42	0,05	0,71	-0,11

Los dos primeros textos se parecen porque sus coordenadas apuntan en una dirección parecida. El tercero queda más lejos porque su patrón numérico es distinto.

Conviene decirlo con cuidado: una dimensión no suele significar “Moodle”, “matrícula” o “problema técnico” de forma aislada. En embeddings modernos, el significado aparece distribuido entre muchas coordenadas a la vez. Una coordenada puede participar en varios patrones; un patrón puede necesitar cientos o miles de coordenadas. Por eso no miramos una dimensión suelta para interpretar el texto: comparamos el vector completo.

La dimensión importa por cuatro razones:

RAZÓN	QUÉ CAMBIA	CONSECUENCIA PRÁCTICA
Memoria	Cada vector ocupa d números.	Más dimensión implica más RAM, disco, backup y red.
Latencia	Comparar vectores cuesta más si d crece.	El ranking exacto y el índice trabajan más.
Señal	El vector tiene más espacio para codificar matices.	Puede mejorar retrieval, pero no siempre en tu dominio.
Compatibilidad	Todos los vectores del índice deben tener la misma dimensión.	Cambiar modelo o dimensión suele exigir reindexar.

La memoria bruta se calcula así:

$$M_{\text{vectores}} = N \cdot d \cdot b$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
M_{vectores}	Memoria bruta para guardar vectores.	Bytes antes de índice y metadata.
N	Número de vectores guardados.	1.000.000 chunks.
d	Dimensión de cada vector.	384, 1024 o 3072.
b	Bytes por número.	4 bytes para <code>float32</code> , 2 para <code>float16</code> .

Para 1 millón de vectores en `float32` :

DIMENSIÓN	MEMORIA BRUTA	LECTURA DE INGENIERÍA
384	1,54 GB	Cómodo para prototipos y muchos casos internos.
768	3,07 GB	Dobla memoria y cómputo respecto a 384.
1536	6,14 GB	Empieza a exigir pensar en índice, RAM y backups.
3072	12,29 GB	Puede tener más señal, pero no sale gratis.

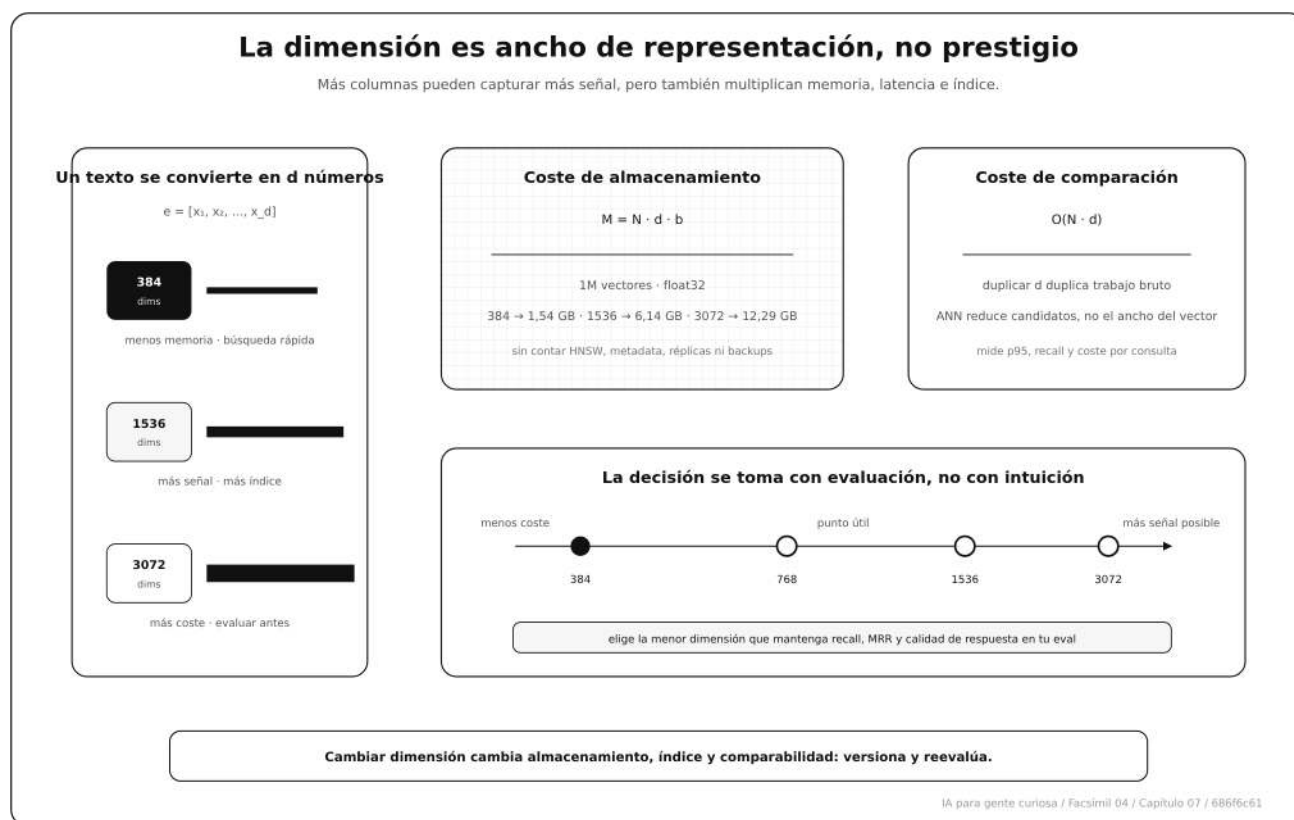
El coste de comparación exacta crece igual:

$$C_{\text{comparar}} = O(N \cdot d)$$

Si duplicas d , duplicas el trabajo bruto de comparar una consulta contra todos los vectores. Un índice aproximado reduce el número de comparaciones, pero no elimina que cada comparación tenga d componentes.

Algunos modelos y proveedores permiten reducir dimensión de salida o usar representaciones tipo Matryoshka, donde los primeros bloques del vector intentan conservar señal útil al truncar dimensiones.¹³ Cohere, por ejemplo, documenta embeddings con Matryoshka y compresión para equilibrar calidad, memoria y coste.¹⁴ Eso no significa que puedas cortar cualquier vector arbitrariamente y esperar el mismo resultado: hay que evaluarlo.

Dimensión, coste y calidad en una sola imagen



Acortar la dimensión sin reentrenar: Matryoshka

«Elige la menor dimensión que funcione» suena bien, pero con un embedding normal no puedes simplemente quedarte con las primeras 256 de 1536 columnas: el modelo repartió la información por todo el vector y truncarlo lo rompe. La solución moderna es el **aprendizaje de representaciones Matryoshka** (MRL): entrenar el modelo para que la información se concentre de forma anidada, de modo que los primeros m números ya sean un buen embedding por sí solos.¹⁵ Modelos como `text-embedding-3` de OpenAI o los de Nomic se entrenan así.

La consecuencia para ingeniería es muy concreta: con un modelo Matryoshka puedes **truncar el vector y renormalizar** (quedarte con $1536 \rightarrow 512 \rightarrow 256$ dimensiones) y recuperar casi toda la calidad, sin reentrenar ni cambiar de modelo. Eso convierte la dimensión en un dial de coste que ajustas por colección: guardas el vector completo, pero sirves la búsqueda con un prefijo más corto donde la latencia y la memoria aprietan. La regla no cambia (mide recall, MRR y nDCG en tu caso a cada dimensión, justo lo que hace la práctica de este capítulo), pero ahora el recorte es barato y reversible en vez de una reindexación con otro modelo.

La métrica que decide el ranking

La similitud coseno compara dirección, no tamaño bruto. Es la medida clásica del modelo de espacio vectorial de la recuperación de información, donde documentos y consultas se representan como vectores y se ordenan por el coseno del ángulo entre ellos:¹⁶

$$\cos(q, d_i) = \frac{q \cdot d_i}{\|q\| \|d_i\|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q	Vector de la consulta.	$[0,2, 0,8]$.
d_i	Vector del documento i .	$[0,1, 0,9]$.
$q \cdot d_i$	Producto punto: suma de productos componente a componente.	$0,2 \cdot 0,1 + 0,8 \cdot 0,9 = 0,74$.
$\ q\ $	Norma o longitud del vector de consulta.	$0,2^2 + 0,8^2 = 0,824$.
$\ d_i\ $	Norma del vector de documento.	$0,1^2 + 0,9^2 = 0,906$.
$\cos(q, d_i)$	Similitud final.	$0,74 / (0,824 \cdot 0,906) = 0,992$.

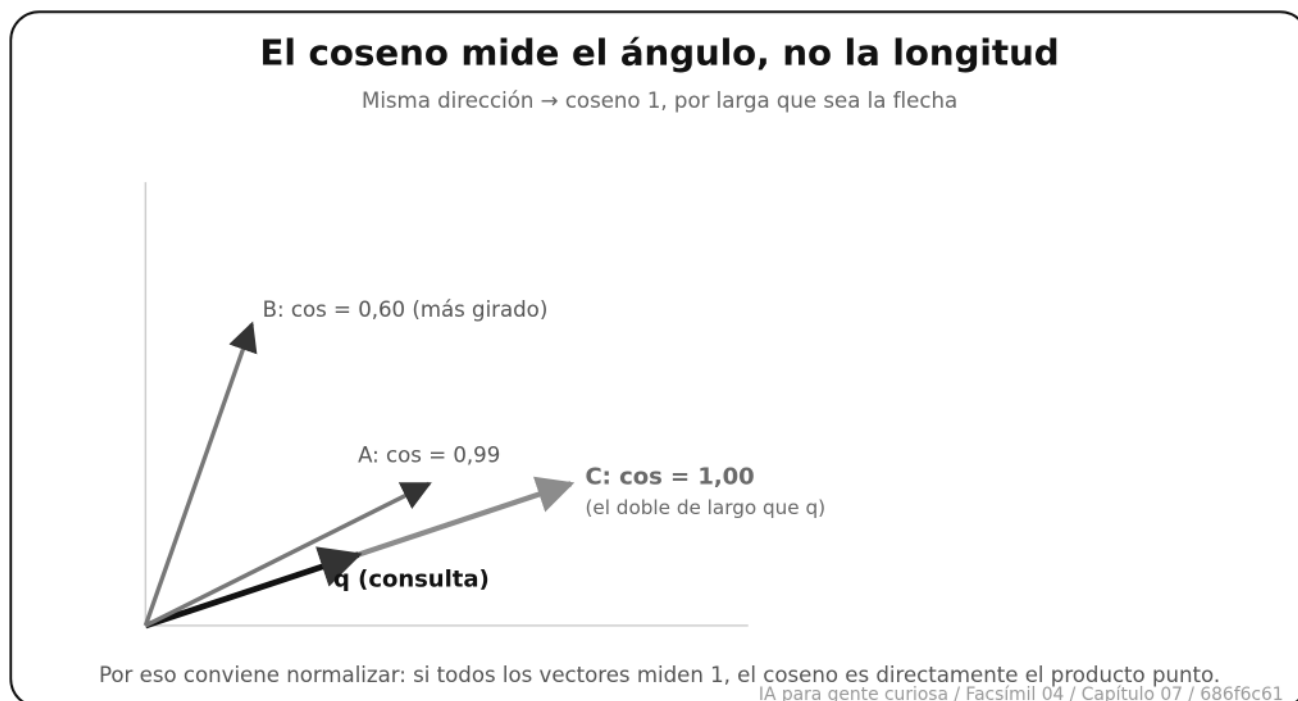
Si normalizas todos los vectores para que tengan norma 1, el coseno se convierte en producto punto:

$$\hat{q} = \frac{q}{\|q\|}, \quad \hat{d}_i = \frac{d_i}{\|d_i\|}$$

$$\cos(q, d_i) = \hat{q} \cdot \hat{d}_i$$

Eso importa en producción porque muchas bases vectoriales trabajan más rápido con producto punto si tus vectores ya están normalizados.

Lo que hace especial al coseno es que mide el **ángulo** entre vectores, no su longitud. Dos textos con la misma dirección semántica tienen coseno 1 aunque uno sea mucho más largo que el otro. En la figura, el vector *C* es el doble de largo que la consulta *q*, pero apunta en la misma dirección: su coseno es 1, por encima de *A* (casi alineado) y de *B* (más girado).



No todo el mundo usa coseno. Una base vectorial te obliga a elegir una métrica al crear el índice, y debe cuadrar con cómo entrenaste y normalizaste:

MÉTRICA	QUÉ MIDE	CUÁNDO USARLA
Coseno	Ángulo entre vectores; ignora la longitud.	Por defecto en búsqueda semántica; lo más común.
Producto punto (inner product)	Alineación y magnitud; favorece vectores largos.	Cuando el modelo se entrenó para que la magnitud codifique algo, o tras normalizar (entonces equivale a coseno). Es el problema de <i>maximum inner product search</i> , MIPS.
Distancia euclídea (L2)	Distancia recta entre puntos.	Si el modelo se entrenó con objetivo L2; en vectores normalizados, ordena igual que el coseno.

La trampa clásica: elegir producto punto en un índice con vectores **sin** normalizar hace que documentos largos ganen por tamaño, no por relevancia. Por eso muchos sistemas normalizan al indexar y usan producto punto, que es más rápido de calcular a escala.¹⁷

El ranking top-k se expresa así:

$$\text{TopK}(q, D, k) = \{d_{(1)}, \dots, d_{(k)}\}, \quad s(q, d_{(1)}) \geq \dots \geq s(q, d_{(k)})$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
D	Colección de documentos vectorizados.	50.000 fragmentos de una intranet.
k	Número de resultados que queremos devolver.	5.
$s(q, d_i)$	Función de puntuación.	Coseno, producto punto o distancia negativa.
TopK	Devuelve los identificadores con mayor puntuación.	Documentos 17, 42, 8, 91 y 3.

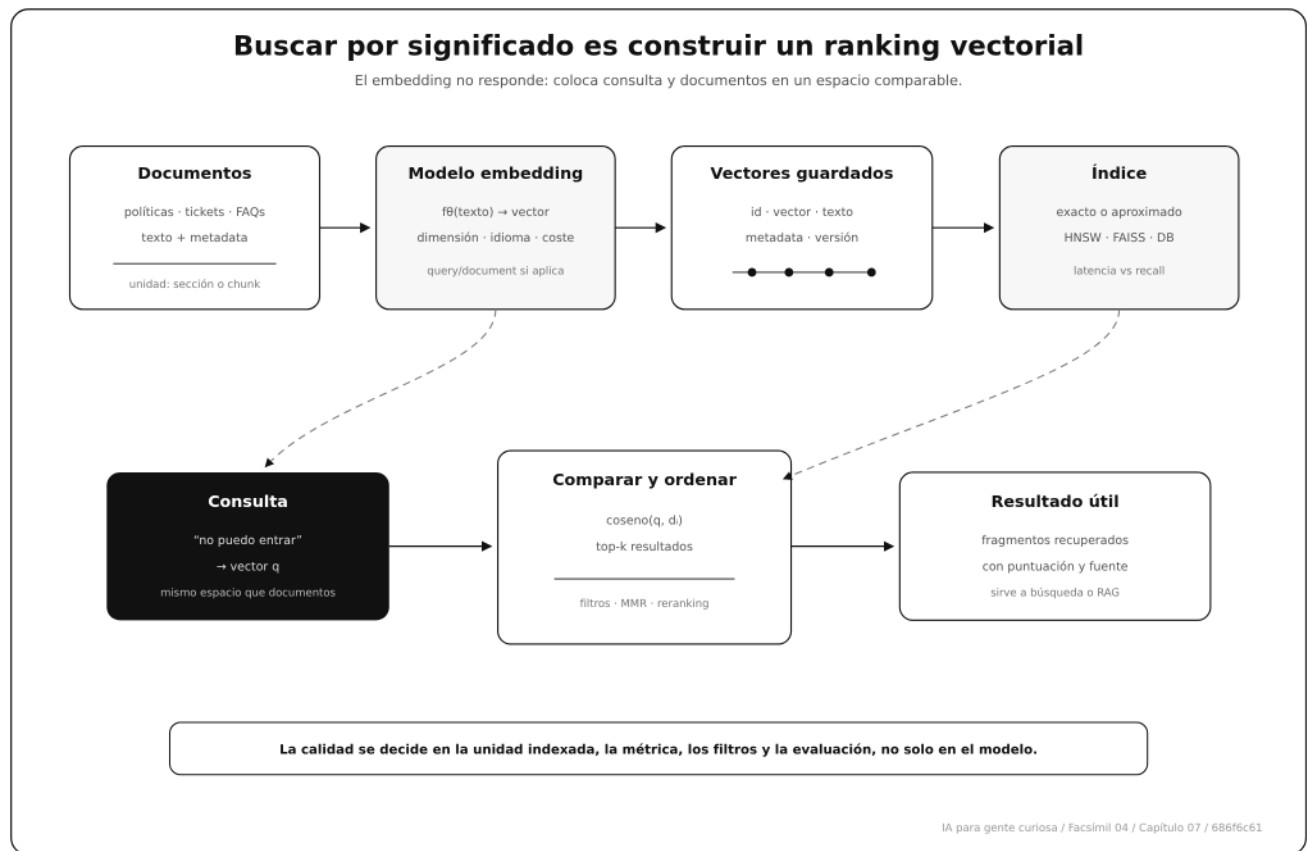
El proceso completo

Una búsqueda semántica mínima tiene dos fases: indexación y consulta. En indexación conviertes documentos en vectores y guardas esos vectores con sus metadatos. En consulta conviertes la pregunta en otro vector, buscas vecinos cercanos y devuelves resultados.

PASO	QUÉ OCURRE	DECISIÓN TÉCNICA
1. Preparar documentos	Limpias, separas, titulas o partes contenido.	Qué unidad se busca: documento entero, sección, párrafo o chunk.
2. Generar embeddings	Cada unidad pasa por el modelo.	Modelo, dimensión, idioma, coste, privacidad y batch.
3. Normalizar	Opcionalmente reescalas vectores.	Coseno/producto punto y compatibilidad con el índice.
4. Guardar	Vector + texto + metadata + versión.	Base vectorial, tabla propia o índice en memoria.
5. Embedding de consulta	La pregunta se transforma con el mismo modelo o modelo compatible.	<code>input_type=query</code> si el proveedor lo usa.
6. Recuperar top-k	Buscas vecinos exactos o aproximados.	Exactitud, latencia, memoria y filtros.
7. Reordenar	Puedes aplicar reranking, filtros o MMR.	Mejorar precisión y diversidad.
8. Usar resultados	Mostrar documentos o pasarlos a un LLM.	Búsqueda, RAG, recomendación o clasificación.

La unidad de búsqueda es decisiva. Si indexas documentos enormes, el resultado puede ser “parecido” pero poco accionable. Si indexas frases demasiado cortas, pierdes contexto. Si indexas chunks sin título, una frase como “plazo máximo” puede quedar huérfana.

Una imagen mental del pipeline



Exacto, aproximado y lo que cuesta

Si tienes pocos documentos, puedes comparar la consulta con todos los vectores. Eso se llama búsqueda exacta por fuerza directa. Si tienes millones de vectores, comparar contra todos puede ser caro y lento; entonces aparecen índices aproximados.

El coste de comparar una consulta contra N documentos de dimensión d es aproximadamente:

$$C_{\text{exacto}} = O(N \cdot d)$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
N	Número de vectores guardados.	1.000.000 chunks.
d	Dimensión de cada vector.	1024.
$O(N \cdot d)$	Trabajo proporcional a comparar N vectores de d números.	Unos 1.024 millones de multiplicaciones/sumas por consulta.

Los índices ANN reducen latencia buscando candidatos probables, no revisando todo. HNSW lo hace con un grafo de vecinos navegable; FAISS agrupa varias técnicas como índices planos, cuantización y búsqueda en GPU. La palabra aproximado no significa “malo”: significa que aceptas una probabilidad de no encontrar exactamente el vecino más cercano a cambio de velocidad.

La memoria también importa:

$$M_{\text{vectores}} = N \cdot d \cdot b$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
M_{vectores}	Memoria bruta para guardar vectores.	Bytes antes de índice y metadata.
N	Número de vectores.	1.000.000.
d	Dimensión.	1536.
b	Bytes por número.	4 bytes si usas <code>float32</code> .

Con $N = 1.000.000$, $d = 1536$ y `float32`, solo los vectores ocupan:

$$1.000.000 \cdot 1536 \cdot 4 = 6.144.000.000 \text{ bytes}$$

Eso son unos 6,1 GB antes de contar índices, texto, metadata, réplicas y backups. Si además guardas 10 millones de chunks, el problema deja de ser “llamar a embeddings” y pasa a ser arquitectura de almacenamiento.

Recuperar barato, reordenar caro: dos fases

El modelo de embeddings que hemos visto es un **bi-encoder**: codifica la consulta y cada documento por separado, lo que permite precalcular todos los vectores e indexarlos. Es rapidísimo y escala a millones, pero paga un precio: como nunca mira la consulta y el documento juntos, su precisión en el orden fino es limitada.

La arquitectura de producción estándar resuelve esto en **dos fases**:

FASE	MODELO	QUÉ HACE	COSTE
1. Recuperación	Bi-encoder (embeddings)	Compara la consulta contra millones de vectores y devuelve los <i>k</i> mejores (por ejemplo, top-50).	Barato, precalculado, escala.
2. Reordenación	Cross-encoder (reranker)	Mira cada par (consulta, documento) juntos en un mismo modelo y les da una puntuación de relevancia más fina.	Caro: solo se aplica a esos 50 candidatos.

El cross-encoder es mucho más preciso porque ve la interacción entre consulta y documento, pero por eso mismo no se puede precalcular ni indexar: habría que pasarlo por cada documento en cada consulta.¹⁸ La división del trabajo es la clave: recuperas 50 candidatos baratos con embeddings, los reordenas con el reranker caro y pasas solo los 5 mejores al LLM. Cuando **no** hacerlo: si tu corpus es pequeño o el bi-encoder ya acierta el top-k en tu evaluación, el reranker añade latencia y coste sin mejorar la respuesta. Como siempre, lo decide tu medición, no la moda.

Elegir modelo de embedding

Elegir un modelo de embeddings no es elegir “el más grande”. Es elegir el que recupera mejor tus documentos con tu idioma, tu dominio, tu latencia y tu presupuesto.

CRITERIO	QUÉ MIRAR	POR QUÉ
Idioma	Español, multilingüe, mezcla de idiomas.	Un modelo fuerte en inglés puede perder matices en español.
Dominio	General, código, legal, financiero, médico, soporte.	El vocabulario y las relaciones cambian.
Dimensión	384, 768, 1024, 1536, 3072...	Afecta memoria, coste y velocidad de búsqueda.
Contexto	Tokens máximos por entrada.	Documentos largos se truncarán o habrá que trocearlos.
input_type	Query/document si el proveedor lo distingue.	Algunas APIs optimizan consultas y documentos de forma distinta.
Modalidad	Texto, imagen, documentos mixtos.	Para PDFs visuales o capturas quizá no baste texto plano.
Local o cloud	Privacidad, latencia, coste y operación.	Conecta directamente con el capítulo 06.
Evaluación	Recall@k, MRR, precisión, nDCG.	El benchmark externo orienta; tu caso decide.

Un detalle importante: si reindexas con otro modelo, los vectores antiguos y nuevos normalmente no son comparables. Cambiar de embedding model puede implicar recalcular todo el índice. Por eso conviene guardar `embedding_model`, `embedding_version`, `dimension`, `normalization`, `created_at` y `source_hash` junto a cada vector.

Cómo trabajar con embeddings sin romper producción

Trabajar con embeddings no es solo llamar una API y guardar un array. Es construir una cadena reproducible: preparar texto, generar vectores, versionarlos, guardarlos, consultarlos y evaluar si siguen sirviendo cuando cambian documentos, modelos o permisos.

TAREA	CÓMO HACERLO	QUÉ COMPROBAR
Preparar entrada	Añadir título, sección, ruta y texto limpio.	Que el fragmento sea entendible fuera del documento original.
Generar por lotes	Enviar batches razonables y controlar reintentos.	Coste, rate limits, errores parciales y orden de resultados.
Versionar	Guardar modelo, dimensión, normalización y hash de fuente.	Poder saber qué vector corresponde a qué texto exacto.
Normalizar	Reescalar si usarás coseno como producto punto.	No mezclar vectores normalizados y sin normalizar.
Guardar metadata	Curso, cliente, permiso, fecha, idioma, tipo de documento.	Poder filtrar antes o después de recuperar.
Reindexar	Planificar jobs idempotentes y reanudables.	No duplicar vectores ni mezclar versiones.
Evaluar	Mantener consultas reales con positivos y hard negatives.	Que cambios de modelo o dimensión no degraden el ranking.
Monitorizar	Medir latencia, top-k vacío, drift y feedback.	Detectar documentos obsoletos o consultas nuevas.

Una regla sencilla: el texto que guardas junto al vector debe ser suficiente para explicar por qué salió ese resultado. Si solo guardas el vector y un identificador, depurar será una tortura tranquila.

Búsqueda semántica no es RAG

Búsqueda semántica recupera candidatos. RAG usa candidatos para construir una respuesta generada. Esta diferencia parece pequeña, pero cambia cómo evalúas.

SISTEMA	SALIDA	QUÉ EVALÚAS
Búsqueda semántica	Lista de documentos o fragmentos.	Si el resultado correcto aparece arriba.
RAG	Respuesta generada con contexto.	Si la respuesta está fundamentada en los fragmentos correctos.
Recomendador	Elementos parecidos o útiles.	Si el usuario acepta, compra, lee o resuelve.
Clasificación por similitud	Etiqueta más cercana.	Si la etiqueta elegida es correcta.

Si el retrieval falla, el generador no puede arreglarlo de forma fiable. Puede escribir una respuesta bonita con evidencia equivocada. Por eso los próximos capítulos separan bases vectoriales, RAG y evaluación de RAG.

Medir si recupera bien

Un buscador semántico debe evaluarse con consultas y respuestas esperadas. No hace falta empezar con un benchmark gigante: puedes crear 30 consultas reales, marcar qué documento debería aparecer y medir. Las métricas son las estándar de la recuperación de información.¹⁹

Recall@k:

$$\text{Recall@k} = \frac{\text{consultas con al menos un resultado correcto en top-k}}{\text{total de consultas}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
k	Número de resultados que miras.	3.
Top-k	Primeros k documentos devueltos.	Los tres primeros resultados.
Resultado correcto	Documento marcado como relevante.	El artículo que resuelve el problema.

MRR (Mean Reciprocal Rank), una métrica consolidada en las campañas de evaluación TREC, mide en qué posición aparece el primer resultado correcto:²⁰

$$\text{MRR} = \frac{1}{Q} \sum_{j=1}^Q \frac{1}{\text{rank}_j}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Q	Número de consultas evaluadas.	30.
rank_j	Posición del primer resultado correcto para la consulta j .	1 si sale primero, 3 si sale tercero.
$1/\text{rank}_j$	Penalización por aparecer más abajo.	1, 0,5, 0,333...

Si el documento correcto aparece siempre en posición 8 y tú solo pasas top-3 al LLM, tu RAG no verá la evidencia aunque “el buscador la tenía”. Ese detalle es muy de ingeniería y muy poco de demo.

Evaluar embeddings, no solo el buscador

Evaluar embeddings no es preguntar “¿me gusta el primer resultado?”. Hay que separar varias preguntas:

PREGUNTA	MÉTRICA ÚTIL	QUÉ DETECTA
¿Aparece algún documento correcto entre los primeros?	Recall@k	Si el sistema encuentra evidencia suficiente.
¿Aparece arriba o enterrado?	MRR	Si el primer resultado útil llega pronto.
¿Ordena bien varios relevantes?	nDCG@k	Si documentos muy relevantes suben más que los medianos.
¿Se degrada al reducir dimensión?	Curva dimensión-métrica	Si puedes ahorrar memoria sin perder calidad.
¿Funciona en todos los grupos?	Métricas por idioma, dominio, tipo de consulta.	Si el promedio oculta fallos por segmento.
¿Distingue parecidos peligrosos?	Hard negatives	Si confunde textos casi iguales pero incorrectos.

BEIR y MTEB existen porque un embedding puede ir bien en una tarea y flojo en otra.²¹ Para un proyecto real, el benchmark externo sirve para elegir candidatos, pero tu evaluación interna decide.

nDCG@k se usa cuando no todos los documentos relevantes valen lo mismo:

$$DCG@k = \sum_{i=1}^k \frac{2^{rel_i} - 1}{\log_2(i + 1)}$$

$$nDCG@k = \frac{DCG@k}{IDCG@k}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
rel_i	Relevancia del resultado en posición i .	2 si responde, 1 si ayuda, 0 si no sirve.
$DCG@k$	Ganancia acumulada con descuento por posición.	Premia relevancia arriba.
$IDCG@k$	DCG ideal si el ranking fuera perfecto.	Mejor orden posible para esa consulta.
$nDCG@k$	DCG normalizado entre 0 y 1.	1 significa orden ideal.

Una evaluación seria de embeddings debería guardar:

CAMPO	EJEMPLO	POR QUÉ
<code>query_id</code>	<code>q-014</code>	Permite repetir y auditar resultados.
<code>query_text</code>	<code>"no puedo entrar al campus"</code>	La consulta real que hizo una persona o un caso diseñado.
<code>positive_ids</code>	<code>["doc-01"]</code>	Documentos que deben aparecer.
<code>graded_relevance</code>	<code>{"doc-01": 2, "doc-05": 1}</code>	Diferencia evidencia principal de apoyo parcial.
<code>hard_negatives</code>	<code>["doc-03"]</code>	Documentos parecidos que no responden.
<code>filters</code>	<code>{"curso": "2026"}</code>	Condiciones de producto o permisos.
<code>embedding_model</code>	<code>all-MiniLM-L6-v2</code>	La evaluación depende del modelo.
<code>dimension</code>	<code>384</code>	Cambiar dimensión puede cambiar ranking.
<code>top_k</code>	<code>3, 5, 10</code>	Define qué ve usuario o LLM.

La parte más útil suele ser mirar errores, no solo el número final. Si una consulta falla porque el documento está mal troceado, el embedding no era el problema. Si falla porque hay dos documentos casi idénticos y uno está obsoleto, necesitas metadata y filtros. Si falla solo al

bajar de 384 a 64 dimensiones, quizá encuentraste el límite de compresión de tu caso.

En el día a día

En una universidad, embeddings pueden servir para que una persona encuentre normativa aunque no conozca el nombre exacto del trámite. En soporte interno, pueden agrupar tickets parecidos y detectar respuestas repetidas. En producto, pueden recomendar documentación relacionada. En una base de conocimiento, pueden recuperar fragmentos para que un LLM responda con contexto.

La parte delicada es que una búsqueda semántica buena no depende solo del modelo. Depende de cómo partes documentos, qué metadata guardas, si filtras por permisos, si separas versiones antiguas, si reordenas resultados y si mides con preguntas reales.

Un caso cercano: tienes 800 artículos de ayuda. Si alguien pregunta “me han bloqueado la cuenta”, un buscador por palabra puede priorizar artículos que contienen “bloqueado”. Un embedding puede recuperar “Restablecer acceso tras demasiados intentos”. Pero si el artículo correcto está obsoleto y falta metadata de versión, el sistema seguirá pareciendo inteligente mientras devuelve una respuesta mala.

Por qué debería importarte

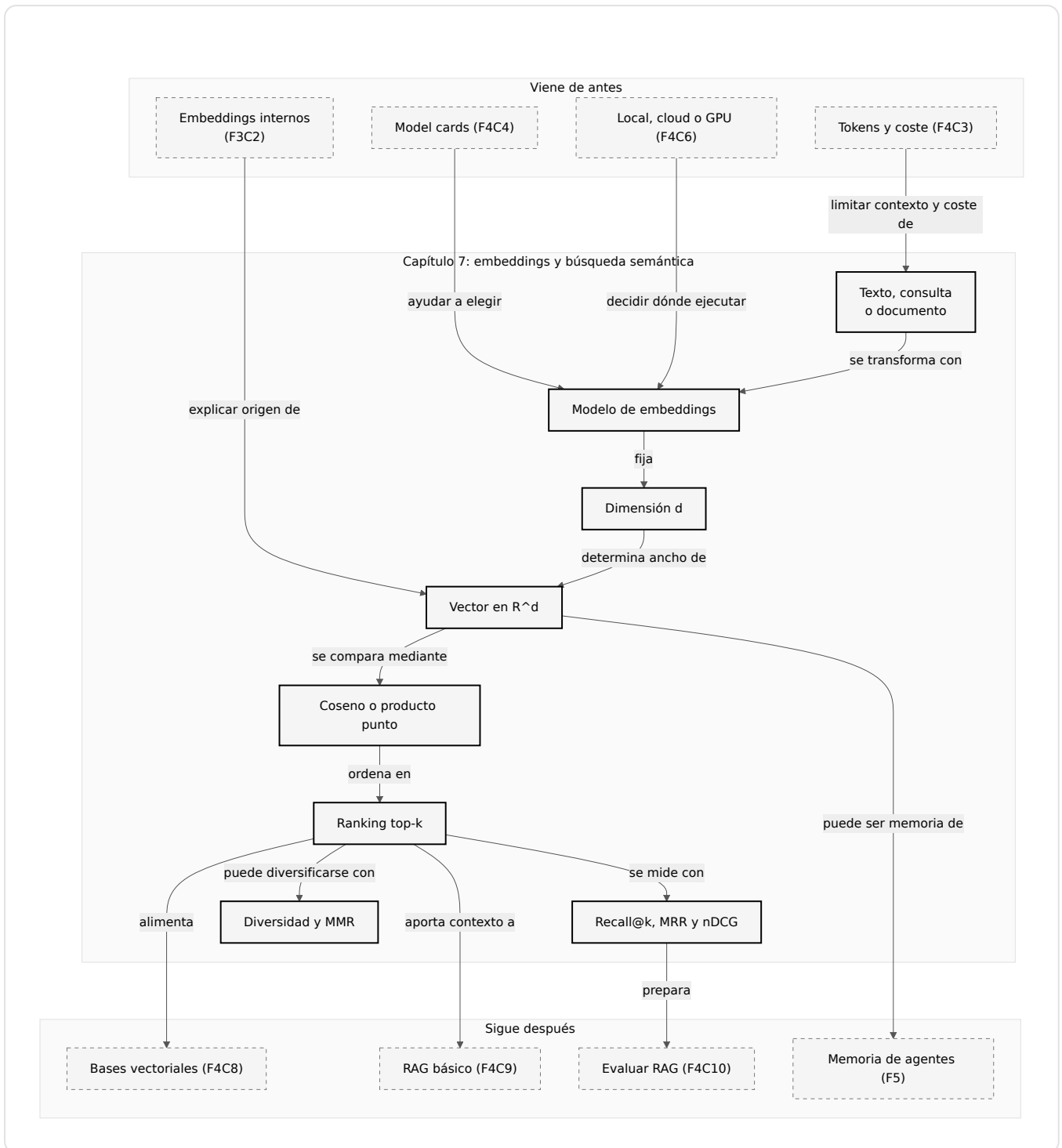
Embeddings son la puerta de entrada a casi todo lo que se vende como “IA con tus datos”. Si no entiendes esta pieza, no sabes si tu RAG falla por el modelo generativo, por el chunking, por la base vectorial, por la métrica o por la evaluación.

También importan por coste. Embeddings se calculan al indexar, se guardan durante meses, se consultan muchas veces y ocupan memoria. Una mala decisión de dimensión, chunking o modelo puede multiplicar almacenamiento y latencia sin mejorar recuperación.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Pensar que más similitud siempre significa mejor respuesta	La similitud mide cercanía aproximada, no utilidad ni vigencia.	Mirar documento, fecha, permisos y pregunta concreta.
Indexar documentos enormes	Recuperas un bloque parecido pero poco accionable.	Probar secciones o chunks con título y metadata.
Cambiar modelo sin reindexar	Los vectores nuevos pueden no ser compatibles con los antiguos.	Versionar modelo, dimensión, normalización y fecha.
Elegir dimensión por tamaño aparente	3072 dimensiones no garantizan mejor producto que 768 en tu corpus.	Comparar dimensión contra Recall@k, MRR, nDCG, coste y latencia.
Evaluar con tres consultas bonitas	El sistema parece bueno solo porque las pruebas eran fáciles.	Crear un set de consultas reales con respuestas esperadas.
Olvidar filtros	Un resultado semánticamente cercano puede ser de otro curso, cliente o versión.	Combinar similitud con metadata y permisos.
Subir top-k sin pensar	Más resultados pueden meter ruido en el LLM y subir coste.	Medir recall@k y calidad de respuesta final.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Embedding	Vector que representa una entrada para compararla con otras.
Búsqueda semántica	Búsqueda que recupera por cercanía aproximada de significado.
Similitud coseno	Medida de alineación entre dos vectores.
Dimensión de embedding	Número de componentes numéricos que tiene cada vector.
Top-k	Primeros k resultados según una puntuación.
Vector normalizado	Vector reescalado para tener norma 1.
ANN	Búsqueda aproximada de vecinos cercanos.
HNSW	Índice por grafo usado para búsqueda vectorial aproximada.
FAISS	Biblioteca de Meta para búsqueda y clustering de vectores.
Recall@k	Métrica que mira si aparece un resultado correcto entre los k primeros.
MRR	Métrica que premia que el primer resultado correcto aparezca arriba.
nDCG@k	Métrica que evalúa orden y grados de relevancia en los primeros k resultados.
Hard negative	Documento parecido pero incorrecto que prueba si el embedding discrimina bien.
MMR	Técnica para equilibrar relevancia y diversidad.

Antes de pasar página

- ☐ ¿Puedo explicar qué es un embedding sin decir “significado puro”? (Si no, vuelve a «Qué sí es: una coordenada útil».)
- ☐ ¿Puedo explicar por qué la dimensión afecta memoria, coste y latencia? (Si no, vuelve a «Qué significa la dimensión».)
- ☐ ¿Puedo calcular un coseno sencillo entre dos vectores? (Si no, vuelve a «La métrica que decide el ranking».)

- ☐ ¿Sé por qué normalizar vectores puede convertir coseno en producto punto? (Si no, vuelve a «La métrica que decide el ranking».)
- ☐ ¿Sé distinguir búsqueda semántica de RAG? (Si no, vuelve a «Búsqueda semántica no es RAG».)
- ☐ ¿Sé qué campos versionar junto a un vector? (Si no, vuelve a «Cómo trabajar con embeddings sin romper producción».)
- ☐ ¿Sé por qué cambiar de modelo obliga normalmente a reindexar? (Si no, vuelve a «Cómo trabajar con embeddings sin romper producción».)
- ☐ ¿Sé medir Recall@k, MRR y nDCG con consultas reales? (Si no, vuelve a «Medir si recupera bien».)
- ☐ ¿Sé comparar varias dimensiones antes de pagar más almacenamiento? (Si no, vuelve a «Evaluar embeddings, no solo el buscador».)
- ☐ ¿Sé cuándo usar búsqueda exacta y cuándo mirar ANN? (Si no, vuelve a «Exacto, aproximado y lo que cuesta».)
- ☐ ¿Sé por qué filtros y metadata son tan importantes como la similitud? (Si no, vuelve a «Cómo trabajar con embeddings sin romper producción».)

En resumen

IDEA FUERZA	DETALLE
Un embedding es una coordenada útil.	Convierte texto u otros objetos en vectores comparables, no en verdad garantizada.
La dimensión tiene coste de ingeniería.	Más dimensiones implican más memoria, cómputo, latencia e índice; solo compensan si mejoran métricas reales.
La búsqueda semántica es ranking.	Consulta y documentos se vectorizan, se comparan y se ordenan por una métrica.
La unidad indexada decide mucho.	Documento, sección, párrafo o chunk cambian la calidad del resultado.
La escala obliga a elegir índice.	Exacto es simple; ANN reduce latencia a cambio de aproximación.
Sin evaluación solo hay intuición.	Recall@k, MRR, nDCG y hard negatives separan demo bonita de sistema útil.

Para saber más

Cohere. (2026). *Introduction to Embeddings at Cohere*.

<https://docs.cohere.com/v2/docs/embeddings>

Google. (2026). *Gemini API: Embeddings*. <https://ai.google.dev/gemini-api/docs/embeddings>

Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-Scale Similarity Search with GPUs*. *IEEE Transactions on Big Data*, 7(3), 535-547. <https://doi.org/10.1109/TBDATA.2019.2921572>

Kusupati, A. y otros (2022). *Matryoshka Representation Learning*.

<https://arxiv.org/abs/2205.13147>

Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs*. *IEEE TPAMI*, 42(4), 824-836.

<https://doi.org/10.1109/TPAMI.2018.2889473>

Muennighoff, N., Tazi, N., Magne, L. y Reimers, N. (2023). *MTEB: Massive Text Embedding Benchmark*. <https://arxiv.org/abs/2210.07316>

OpenAI. (2026). *text-embedding-3-large*. <https://developers.openai.com/api/docs/models/text-embedding-3-large>

OpenAI. (2026). *Vector embeddings*. <https://platform.openai.com/docs/guides/embeddings>

Reimers, N. y Gurevych, I. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. *Proceedings of EMNLP*, 3982-3992. <https://doi.org/10.18653/v1/D19-1410>

Sentence Transformers. (2026). *Semantic Search*.

<https://sbert.net/examples/applications/semantic-search/README.html>

Thakur, N., Reimers, N., Rücklé, A., Srivastava, A. y Gurevych, I. (2021). *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models*.

NeurIPS Datasets and Benchmarks. <https://arxiv.org/abs/2104.08663>

Voyage AI. (2026). *Text Embeddings*. <https://docs.voyageai.com/docs/embeddings>

Notas

1. OpenAI. (2026). *Vector embeddings*. <https://platform.openai.com/docs/guides/embeddings>. Consultado el 25 de mayo de 2026. ↵

2. OpenAI. (2026). *text-embedding-3-large*. <https://developers.openai.com/api/docs/models/text-embedding-3-large>. Consultado el 25 de mayo de 2026. ↵

3. Google. (2026). *Gemini API: Embeddings*. <https://ai.google.dev/gemini-api/docs/embeddings>. Consultado el 25 de mayo de 2026. ↵
4. Cohere. (2026). *Introduction to Embeddings at Cohere*. <https://docs.cohere.com/v2/docs/embeddings>. Consultado el 25 de mayo de 2026. ↵
5. Voyage AI. (2026). *Text Embeddings*. <https://docs.voyageai.com/docs/embeddings>. Consultado el 25 de mayo de 2026. ↵
6. Sentence Transformers. (2026). *Semantic Search*. <https://sbert.net/examples/applications/semantic-search/README.html>. Consultado el 25 de mayo de 2026. ↵
7. Reimers, N. y Gurevych, I. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. Proceedings of EMNLP, 3982-3992. <https://doi.org/10.18653/v1/D19-1410>. ↵
8. Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-Scale Similarity Search with GPUs*. *IEEE Transactions on Big Data*, 7(3), 535-547. <https://doi.org/10.1109/TBDATA.2019.2921572>. ↵
9. Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs*. *IEEE TPAMI*, 42(4), 824-836. <https://doi.org/10.1109/TPAMI.2018.2889473>. ↵
10. Thakur, N., Reimers, N., Rücklé, A., Srivastava, A. y Gurevych, I. (2021). *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models*. NeurIPS Datasets and Benchmarks. <https://arxiv.org/abs/2104.08663>. ↵
11. Tomas Mikolov y otros (2013). *Efficient Estimation of Word Representations in Vector Space*. <https://arxiv.org/abs/1301.3781>. Nils Reimers e Iryna Gurevych. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. EMNLP. <https://arxiv.org/abs/1908.10084>. ↵
12. Aaron van den Oord, Yazhe Li y Oriol Vinyals. (2018). *Representation Learning with Contrastive Predictive Coding*. <https://arxiv.org/abs/1807.03748>. La aplicación a recuperación densa con negativos en el lote y negativos difíciles se formaliza en Vladimir Karpukhin y otros (2020). *Dense Passage Retrieval for Open-Domain Question Answering*. EMNLP. <https://arxiv.org/abs/2004.04417>. ↵
13. Kusupati, A. y otros (2022). *Matryoshka Representation Learning*. <https://arxiv.org/abs/2205.13147>. El trabajo propone aprender representaciones que funcionan a varias longitudes anidadas. ↵
14. Cohere (2026), documentación de embeddings citada en la tabla de estado del arte. ↵
15. Aditya Kusupati y otros (2022). *Matryoshka Representation Learning*. NeurIPS. <https://arxiv.org/abs/2205.13147>. ↵
16. Gerard Salton, A. Wong y C. S. Yang. (1975). *A Vector Space Model for Automatic Indexing*. *Communications of the ACM*, 18(11), 613-620. <https://doi.org/10.1145/361219.361220>. ↵

- .7. Jeff Johnson, Matthijs Douze y Hervé Jégou. (2019). *Billion-scale Similarity Search with GPUs*. IEEE Transactions on Big Data. <https://arxiv.org/abs/1702.08734>. ↵
- .8. Nils Reimers e Iryna Gurevych. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. EMNLP. <https://arxiv.org/abs/1908.10084>. Para reordenación con cross-encoders: Rodrigo Nogueira y Kyunghyun Cho. (2019). *Passage Re-ranking with BERT*. <https://arxiv.org/abs/1901.04085>. ↵
- .9. Christopher D. Manning, Prabhakar Raghavan e Hinrich Schütze. (2008). *Introduction to Information Retrieval*. Cambridge University Press. Capítulo 8 (evaluación en recuperación de información). <https://nlp.stanford.edu/IR-book/>. ↵
- !0. Ellen M. Voorhees. (2002). *The Philosophy of Information Retrieval Evaluation*. En Evaluation of Cross-Language Information Retrieval Systems (CLEF 2001), LNCS 2406, 355-370. https://doi.org/10.1007/3-540-45691-0_34. ↵
- !1. Thakur y otros (2021) proponen BEIR para evaluar recuperación en tareas heterogéneas. Muennighoff, N., Tazi, N., Magne, L. y Reimers, N. (2023). *MTEB: Massive Text Embedding Benchmark*. <https://arxiv.org/abs/2210.07316>. MTEB compara embeddings en clasificación, clustering, retrieval, reranking, similitud semántica y otras tareas. ↵

Capítulo 08: Bases vectoriales, filtros y búsqueda híbrida

Cuando el vector deja de ser el problema

En el [capítulo 07](#) construimos la pieza básica: convertir textos en vectores, compararlos y ordenar resultados. Eso ya permite hacer una búsqueda semántica pequeña. Pero en cuanto el sistema deja de ser una demo, aparece otra pregunta: dónde viven esos vectores, cómo se filtran, cómo se actualizan, cómo se borran y cómo sabemos que el índice no está devolviendo resultados bonitos pero equivocados.

Imagina una universidad con miles de documentos internos. Hay normativa de 2024, 2025 y 2026; manuales para estudiantes y profesorado; documentos públicos y documentos solo visibles para equipos concretos. La consulta "no puedo entrar a Moodle con doble factor" no puede devolver cualquier texto parecido. Debe devolver documentos vigentes, del curso correcto y visibles para la persona que pregunta.

Una base vectorial no es "una carpeta de embeddings". Es el lugar donde se cruzan similitud, filtros, permisos, versiones, índices, latencia, borrado y evaluación. Si esta pieza está mal diseñada, el RAG del [capítulo 09](#) heredará el problema aunque el modelo generativo sea excelente.

Estado del arte con fecha de corte

Fecha de corte: 25 de mayo de 2026.

Fuentes consultadas ese día: documentación oficial de Qdrant, pgvector, Weaviate, Milvus y Pinecone; y trabajos académicos sobre FAISS, HNSW, cuantización de producto, BM25 y fusión de rankings.

Lo estable es la arquitectura: guardar vectores con identificadores, texto y metadata; construir índices; filtrar candidatos; combinar señales densas y léxicas; medir recall y latencia. Lo cambiante son APIs concretas, parámetros, límites por plan, soporte de filtros, algoritmos de índice, modelos integrados y costes de almacenamiento.

FUENTE	QUÉ APORTA	CÓMO USARLA
FAISS. 1	Muestra técnicas de búsqueda eficiente de vectores a gran escala.	Para entender por qué no basta hacer producto punto contra todo cuando crece el corpus.
HNSW. 2	Formaliza un índice por grafo usado por muchas bases vectoriales.	Para entender el intercambio entre memoria, construcción, rapidez y recall.
Qdrant. 3	Explica que índice vectorial e índice de payload resuelven partes distintas del problema.	Para no confundir "tengo HNSW" con "mis filtros ya van bien".
pgvector. 4	Integra búsqueda vectorial dentro de PostgreSQL con operadores de distancia e índices HNSW e IVFFlat.	Para proyectos donde transacciones, SQL, joins y vectores conviven en la misma base.
Weaviate. 5	Documenta búsqueda híbrida que combina vector y BM25F mediante fusión configurable.	Para ver el patrón denso + léxico sin construir todo a mano.
Milvus. 6	Distingue filtrado estándar e iterativo en búsqueda vectorial con metadata.	Para razonar sobre filtros complejos y latencia.
Pinecone. 7	Compara usar un índice híbrido único frente a índices densos y dispersos separados.	Para entender que "híbrido" también es una decisión de arquitectura.

Qué no es una base vectorial

Una base vectorial no arregla embeddings malos. Si el modelo coloca cerca documentos que no deberían estarlo, el índice solo acelerará ese error. Tampoco arregla un mal troceado: si guardas párrafos sin título, sin sección y sin fecha, recuperarás fragmentos pobres con mucha rapidez.

Tampoco es una base de conocimiento completa. El vector no sustituye al texto original, a las citas, a las reglas de acceso, al historial de versiones ni al sistema que decide si un documento está vigente. La base vectorial guarda una representación para buscar; la verdad documental sigue viviendo en el contenido y en la metadata.

Y no es siempre la herramienta adecuada. Para cien documentos, una búsqueda exacta en memoria puede bastar. Para datos relacionales con filtros complejos, PostgreSQL con `pgvector` puede ser suficiente. Para millones de fragmentos, alta concurrencia, filtros frecuentes o múltiples señales de ranking, conviene pensar en una base vectorial dedicada o en un buscador que combine índice invertido y vectores.

Qué sí es: un contrato de recuperación

Una base vectorial no guarda solo vectores: guarda **registros**. Cada registro r_i es una tupla con identificador, vector, texto, metadata y versión, y el contrato de recuperación se define sobre esa estructura, no sobre el vector aislado:

$$r_i = (id_i, v_i, texto_i, m_i, version_i)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
r_i	Registro número i .	Fragmento de una normativa.
id_i	Identificador estable.	normativa-2026#sec-04 .
v_i	Vector del fragmento.	768 números float32 .
$texto_i$	Texto recuperable o puntero al texto.	Párrafo que se pasará al RAG.
m_i	Metadata o payload.	curso=2026 , rol=estudiante , vigente=true .
$version_i$	Versión de embedding y documento.	embed-v3-large@2026-05-25 .

La búsqueda con filtro se expresa así:

$$\text{TopK}(q, C, F, k) = \{r_{(1)}, \dots, r_{(k)}\}$$

$$r_{(j)} \in C, \quad F(m_{(j)}) = 1, \quad s(q, v_{(1)}) \geq \dots \geq s(q, v_{(k)})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q	Vector de la consulta.	Embedding de "acceso a Moodle".
C	Colección donde buscamos.	Fragmentos de documentación interna.
F	Función de filtro sobre metadata.	curso == 2026 and vigente == true .
$m_{(j)}$	Metadata del resultado en posición j .	Curso, idioma, rol, fuente.
$s(q, v)$	Puntuación de similitud.	Coseno o producto punto.
k	Número de resultados devueltos.	8 fragmentos para un RAG.

Esta fórmula tiene una lección importante: el filtro no es decoración posterior. Forma parte de lo que significa "resultado válido". Si el sistema encuentra el fragmento más parecido del mundo pero no cumple `F` , ese resultado no debería existir para la consulta.

El coste real: vectores, índices y payload

En el capítulo anterior calculamos el coste bruto de guardar vectores: $N \cdot d \cdot b$. En una base vectorial real, esa es solo la primera línea de la factura de memoria; el total suma índice, payload y réplicas:

$$M_{total} \approx N \cdot d \cdot b + M_{indice} + M_{payload} + M_{réplicas}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
N	Número de registros vectoriales.	10 millones de chunks.
d	Dimensión del vector.	768 dimensiones.
b	Bytes por componente.	4 bytes en <code>float32</code> , 2 en <code>float16</code> .
M_{indice}	Memoria del índice ANN.	Grafo HNSW o listas IVF.
$M_{payload}$	Metadata, texto corto, ids y estructuras auxiliares.	Fechas, permisos, fuente, idioma.
$M_{réplicas}$	Copias por disponibilidad o rendimiento.	Dos réplicas duplican parte del coste.

Con 10 millones de vectores de 768 dimensiones en `float32` , solo el bloque vectorial ocupa alrededor de 30,72 GB. Eso no incluye índice, payload, logs, réplicas, snapshots ni espacio temporal para reconstruir índices. Por eso la dimensión del capítulo 07 y la operación del capítulo 06 vuelven aquí con fuerza.

La factura de memoria no es solo los vectores

Colección de 10M vectores de 768 dims en float32

factura real $\approx$ 2-3x el bloque de vectores



El índice y las réplicas suelen igualar o superar el bloque de vectores.

Tamaños de índice, payload y réplicas ilustrativos; presupuesta la colección entera, no solo los vectores.

IA para gente curiosa / Facsímil 04 / Capítulo 08 / 686f6c61

La selectividad del filtro también importa:

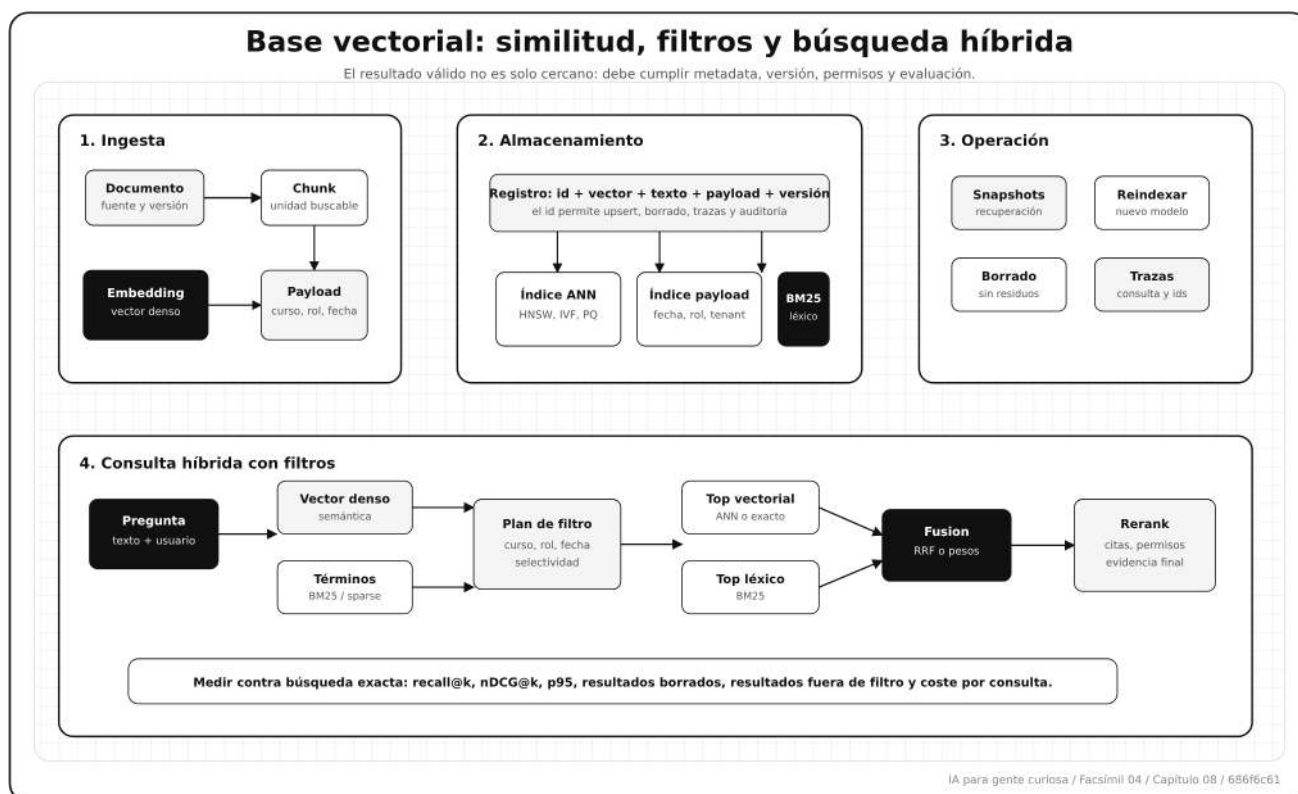
$$\sigma(F) = \frac{|\{r_i \in C : F(m_i) = 1\}|}{|C|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\sigma(F)$	Fracción de la colección que pasa el filtro.	0,02 si quedan 2 de cada 100.
\$	C	\$
\$	{...}	\$

Un filtro con $\sigma = 0,9$ apenas reduce el problema. Uno con $\sigma = 0,001$ puede romper supuestos del índice aproximado si no está bien planificado. Las bases vectoriales serias dedican mucha ingeniería a combinar índice vectorial e índice de metadata porque las dos piezas tiran en direcciones distintas.

Cómo funciona por dentro

Una base vectorial tiene dos rutas principales: ingesta y consulta. En ingesta recibe texto, genera o recibe embeddings, valida el esquema, guarda payload y actualiza índices. En consulta recibe una pregunta, genera el vector de consulta, aplica filtros, busca candidatos, fusiona señales si hay búsqueda híbrida y devuelve resultados con puntuaciones y metadata.



La imagen resume el punto central: una consulta real atraviesa dos mundos. Por un lado está la cercanía semántica; por otro, el contrato operativo que decide si ese resultado se puede usar. El producto final no debería aceptar un resultado solo porque su vector está cerca.

Índices: exacto, HNSW, IVFFlat y compresión

La búsqueda exacta compara la consulta con todos los vectores. Es fácil de razonar y sirve como referencia de calidad, pero su coste crece con $N \cdot d$. A partir de cierto tamaño, necesitamos índices aproximados de vecinos cercanos.

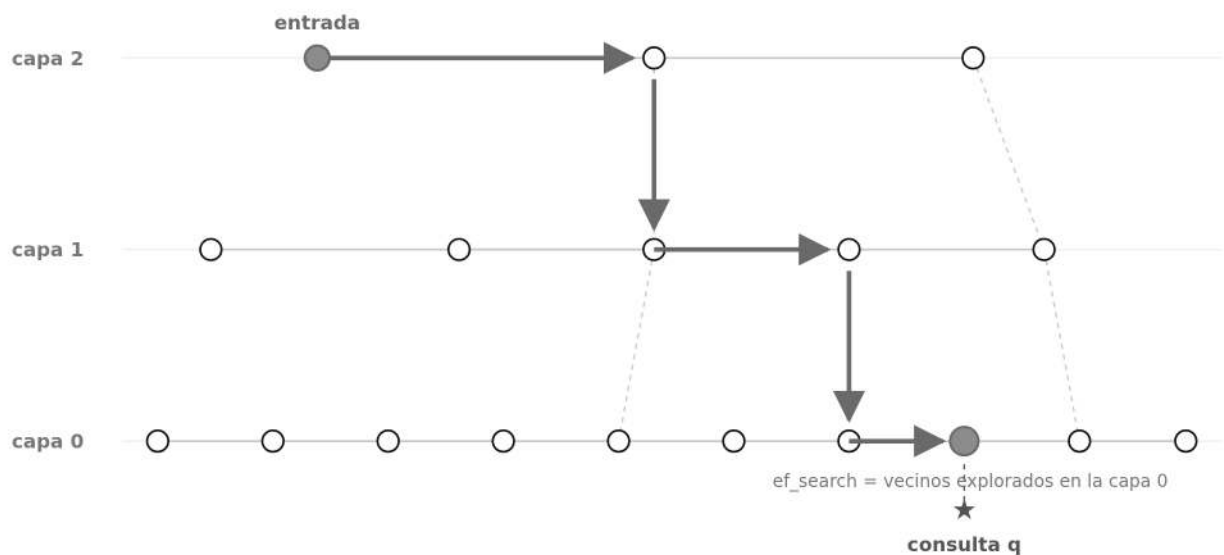
OPCIÓN	IDEA	PARÁMETROS TÍPICOS	QUÉ SE MIDE
Exacta	Comparar contra todos los vectores que pasan el filtro.	Sin índice ANN.	Calidad máxima, latencia base.
HNSW	Navegar un grafo de vecinos por capas.	<code>M</code> , <code>ef_construction</code> , <code>ef_search</code> .	Recall frente a memoria y p95.
IVFFlat	Dividir vectores en listas y buscar solo algunas.	<code>lists</code> , <code>probes</code> .	Recall frente a velocidad y coste de build.
PQ	Comprimir vectores por subespacios.	Número de subcuantizadores y bits.	Ahorro de memoria frente a pérdida de precisión.

HNSW suele dar buen equilibrio de recall y latencia, pero no es gratis: guarda conexiones entre vectores y consume memoria adicional.⁸ IVFFlat puede construir más rápido y ocupar menos, pero requiere elegir listas y probes con cuidado; la documentación de pgvector lo explica como un intercambio entre rendimiento y recall.⁹ La cuantización de producto reduce memoria representando subespacios con códigos compactos, pero introduce aproximación adicional.¹⁰

Conviene tener una imagen mental de HNSW, porque es el índice más usado. Es un grafo navegable de pequeño mundo organizado en **capas**: arriba hay pocos nodos con saltos largos, y abajo están todos los vectores con conexiones cortas. La búsqueda entra por la capa superior, se acerca a la consulta dando saltos grandes y baja de capa en capa afinando, hasta hacer una búsqueda fina en la capa 0. Ese descenso es lo que la hace rápida; `ef_search` controla cuántos vecinos explora al final.

HNSW: un grafo navegable por capas

La búsqueda entra por arriba y baja hacia la consulta; ef_search controla cuánto explora abajo



IA para gente curiosa / Facsímil 04 / Capítulo 08 / 686f6c61

Un criterio práctico: conserva siempre un modo de evaluación exacta, aunque sea sobre una muestra. Si no puedes comparar el índice aproximado contra el ranking exacto, no sabes si ganar latencia te está costando evidencias importantes.

La métrica que de verdad mide un índice aproximado

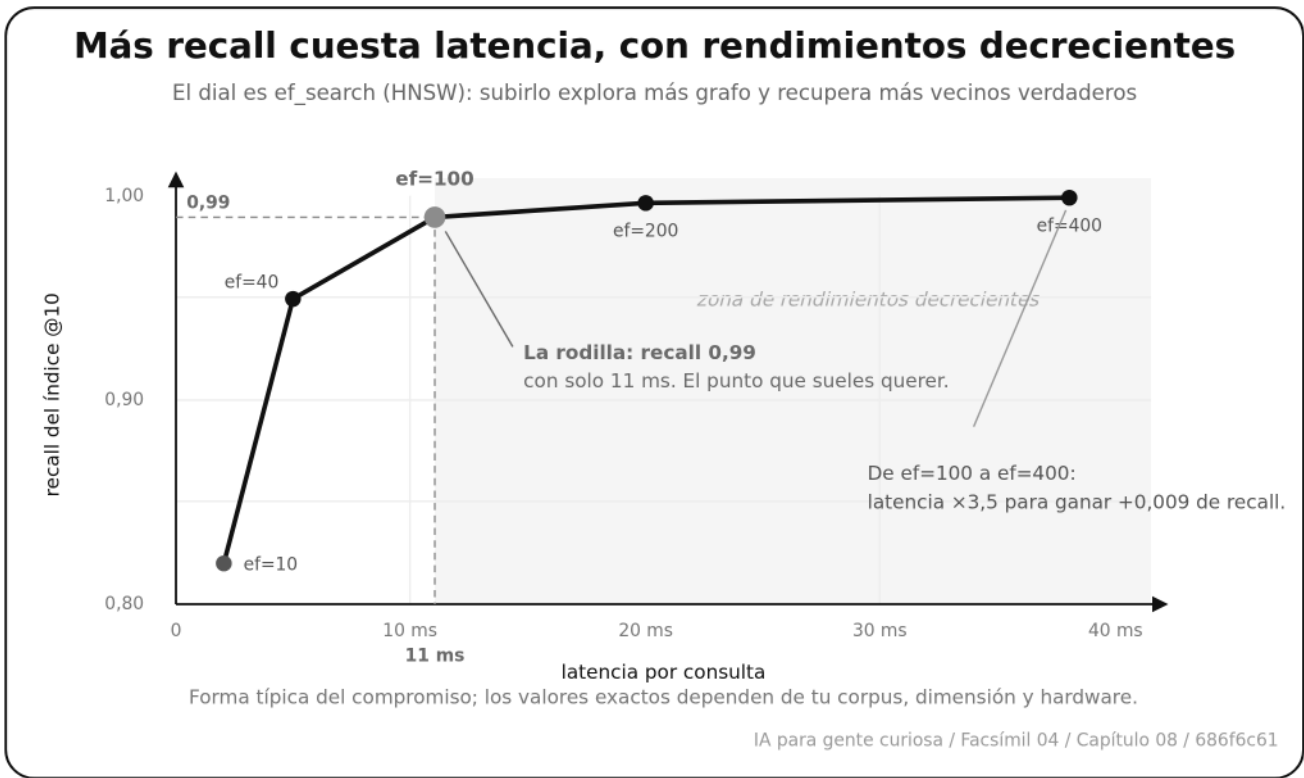
«Aproximado» tiene un número. El **recall del índice ANN** mide qué fracción de los verdaderos k vecinos más cercanos (los que daría la búsqueda exacta) recupera el índice aproximado para la misma consulta. Es la métrica estándar con la que se comparan los algoritmos de vecinos aproximados:¹¹

$$\text{Recall}_{\text{ANN}@k} = \frac{|R_{\text{ANN}}(q, k) \cap R_{\text{exacto}}(q, k)|}{k}$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
$R_{\text{ANN}}(q, k)$	Los k resultados que devuelve el índice aproximado.	Top-10 de HNSW.
$R_{\text{exacto}}(q, k)$	Los k verdaderos vecinos más cercanos (fuerza bruta).	Top-10 exacto sobre la misma colección.
$\cap$	Intersección: cuántos coinciden.	9 de 10.

No hay que confundir este recall con el **recall de recuperación** del capítulo 07: aquel mide si encuentras el documento relevante para una persona; este mide si el índice rápido encuentra lo mismo que el lento. Un índice puede tener `Recall_ANN@10 = 0,9` (pierde uno de cada diez vecinos verdaderos) y aun así servir bien si esos vecinos perdidos no eran los relevantes, o fatal si lo eran. Por eso se miden los dos.

El recall del índice no es fijo: es un dial. En HNSW lo gobierna `ef_search` ; subirlo explora más grafo, sube el recall y cuesta latencia, con rendimientos decrecientes. La forma de ese compromiso es la que de verdad eliges al operar una base vectorial:



Comprimir para que quepa: int8, binario y rerank

La product quantization de la tabla no es la única forma de pagar menos memoria. A escala, lo más directo es bajar la precisión de cada componente del vector, y la diferencia es brutal:

REPRESENTACIÓN	BYTES POR COMPONENTE	10M VECTORES DE 768 DIMS	FACTOR
float32	4	30,72 GB	x1
int8 (cuantización escalar)	1	7,68 GB	x4
Binario (1 bit por dimensión)	0,125	0,96 GB	x32

La cuantización escalar a `int8` mapea cada número real a un entero de 8 bits y suele perder muy poca calidad. La **cuantización binaria** lleva cada dimensión a un solo bit (el signo), permite comparar vectores con distancia de Hamming (operaciones sobre bits, baratísimas) y reduce la memoria 32 veces, a costa de una pérdida de precisión mayor.

El patrón de ingeniería que recupera esa precisión es el mismo de dos fases que vimos con los rerankers: **buscar en comprimido, reordenar en preciso**. Recuperas un top-k amplio (por ejemplo 200) con vectores binarios, rapidísimo y con poca memoria, y reordenas esos 200 candidatos con los vectores `float32` (o `int8`) originales para recuperar el orden fino. Así pagas memoria de índice barata para el grueso y precisión completa solo donde decide el ranking. Como siempre, la regla es medir el `Recall_ANN@k` de la versión comprimida frente a la exacta antes de comprometerte: si binario mantiene tu recall de recuperación, el ahorro de 32x es difícil de rechazar; si lo hunde, `int8` suele ser el término medio sensato.

Filtros: dónde se gana o se rompe la recuperación

Los filtros parecen sencillos hasta que crece el corpus. Filtrar por `curso=2026` es fácil; filtrar por `tenant`, `rol`, `vigente`, `idioma`, `producto`, `region`, `tipo_documento` y `fecha` ya obliga a planificar.

Hay tres patrones frecuentes:

PATRÓN	CÓMO FUNCIONA	RIESGO
Filtrar antes	Primero reduce candidatos por metadata y luego busca vectores.	Si queda muy poco, el índice ANN puede no aportar mucho.
Buscar antes	Primero recupera muchos vecinos y luego descarta por metadata.	Puede perder resultados válidos si estaban fuera del primer lote.
Filtrado integrado	El índice combina metadata y navegación vectorial.	Requiere crear buenos índices de payload y entender selectividad.

Qdrant lo dice de forma muy clara: el índice vectorial acelera la búsqueda vectorial y los índices de payload aceleran filtros; hacen trabajos distintos.¹² Milvus también separa filtrado estándar e iterativo porque filtros complejos pueden cambiar mucho la latencia.¹³

Para entenderlo, piensa en una biblioteca. Si buscas "reglamento de prácticas" en toda la biblioteca y luego tiras los libros antiguos, quizá no veas el reglamento correcto porque no entró en el primer top-k. Si primero entras en la estantería "2026" y después buscas por significado, reduces ruido. Pero si la estantería tiene solo tres documentos, un recorrido exacto puede ser mejor que un índice sofisticado.

Por qué un filtro muy selectivo rompe HNSW

La frase del capítulo anterior («un filtro con $\sigma = 0,001$ puede romper supuestos del índice») tiene un mecanismo concreto, y conviene entenderlo. HNSW busca **navegando un grafo**: salta de vecino en vecino acercándose a la consulta. Ese recorrido funciona porque el grafo está bien conectado. Cuando aplicas un filtro muy selectivo *durante* la navegación (filtrado integrado), la mayoría de los nodos por los que el algoritmo querría pasar quedan descartados, el camino se corta y la búsqueda se queda atascada en una región sin poder llegar a los verdaderos vecinos que sí cumplen el filtro. El resultado es un **precipicio de recall**: por debajo de cierta selectividad, el recall del índice cae en picado aunque la respuesta correcta exista en la colección.

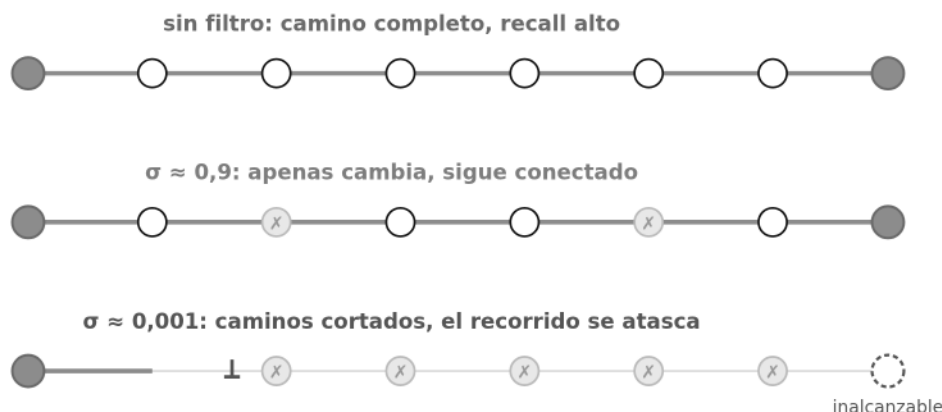
Por eso la selectividad del filtro decide la estrategia, no es un detalle:

SELECTIVIDAD σ	QUÉ PASA	ESTRATEGIA RAZONABLE
Alta ($\sigma \approx 0,9$)	El filtro apenas reduce; el grafo casi no cambia.	Filtrado integrado; el ANN funciona bien.
Media ($\sigma \approx 0,1$)	Zona delicada; el recall puede empezar a sufrir.	Medir recall con y sin filtro; ajustar <code>ef_search</code> .
Baja ($\sigma \approx 0,001$)	Quedan tan pocos candidatos que el grafo se desconecta.	Pre-filtrar y hacer búsqueda exacta sobre el subconjunto: con pocos vectores, la fuerza bruta gana al grafo.

La regla de ingeniería: no elijas pre, post o integrado por defecto; elígelo según la selectividad esperada de tus filtros, y mide el `Recall_ANN@k` **con el filtro puesto**, no solo sin él.

Un filtro muy selectivo desconecta el grafo

Si filtras durante la navegación, los nodos descartados cortan los caminos hacia la consulta



Por debajo de cierta selectividad el recall cae en picado: pre-filtra y haz búsqueda exacta sobre el subconjunto.

IA para gente curiosa / Facsímil 04 / Capítulo 08 / 686f6c61

Búsqueda híbrida: cuando exactitud léxica y semántica se necesitan

Los embeddings son fuertes con sinónimos e intención. BM25 es fuerte con palabras raras, siglas, códigos, nombres propios y errores donde la palabra exacta importa. La búsqueda híbrida combina ambas señales.

BM25, simplificado, puntúa una consulta Q sobre un documento D así:

$$\text{BM25}(D, Q) = \sum_{t \in Q} \text{IDF}(t) \frac{f(t, D)(k_1 + 1)}{f(t, D) + k_1(1 - b + b \frac{|D|}{\text{avgdl}})}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
t	Término de la consulta.	MFA , Moodle , matrícula .
$f(t, D)$	Veces que aparece el término en el documento.	3 apariciones de Moodle .
$\text{IDF}(t)$	Peso de rareza del término en la colección.	MFA pesa más que el .
$\$$	D	$\$$
avgdl	Longitud media de documentos.	220 tokens.
k_1, b	Parámetros de saturación y longitud.	Valores habituales: $k_1 = 1,2$, $b = 0,75$.

BM25 viene de la familia probabilística de recuperación de información y sigue siendo una base fuerte para búsqueda léxica.¹⁴ La parte densa recupera significado; la parte léxica protege términos que no deberían diluirse.

Una forma sencilla de fusionar rankings es RRF:

$$\text{RRF}(d) = \sum_{j=1}^S \frac{1}{k_0 + \text{rank}_j(d)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
d	Documento candidato.	doc-01 .
S	Número de sistemas que devuelven ranking.	Vectorial y BM25: $S = 2$.
$\text{rank}_j(d)$	Posicion del documento en el sistema j .	1 en BM25, 5 en vectorial.
k_0	Constante que suaviza el peso de la posición.	60 es un valor comun en RRF.

RRF funciona bien porque no exige que las puntuaciones de BM25 y embeddings estén en la misma escala.¹⁵ Eso es práctico: un coseno de 0,72 y un BM25 de 11,4 no son directamente comparables, pero sus posiciones en rankings sí pueden combinarse.

CONSULTA	VECTOR DENSO AYUDA	BM25 AYUDA	HIBRIDO EVITA
"no puedo entrar al campus"	Encuentra "restablecer acceso a Moodle".	Poco, si no comparte palabras.	Quedarse solo con sinónimos.
"error SAML 403 Moodle"	Puede entender "login".	Protege SAML y 403 .	Perder códigos exactos.
"matrícula 2026 septiembre"	Relaciona matrícula con calendario.	Protege 2026 y septiembre .	Devolver normativa antigua.
"API pagos webhook reintentos"	Relaciona integración y eventos.	Protege webhook .	Mezclar artículos de producto.

Weaviate expone búsqueda híbrida como combinacion de resultados vectoriales y BM25F por fusión configurable.¹⁶ Pinecone documenta dos caminos: índice híbrido único o índices densos y dispersos separados, cada uno con sus ventajas operativas.¹⁷ La idea pedagógica es la misma: no hay que elegir religión entre vector y palabras exactas; hay que medir cuál combina mejor en tu corpus.

Más allá de un vector por documento: interacción tardía

Hasta aquí hemos asumido **un vector por documento**: el bi-encoder comprime todo el texto en un único punto del espacio. Es eficiente, pero pierde detalle, porque un párrafo largo no cabe bien en una sola coordenada. Hay un paradigma alternativo que conviene conocer: la **interacción tardía** (late interaction), popularizada por ColBERT.¹⁸

En lugar de un vector por documento, ColBERT guarda **un vector por token** y puntúa con MaxSim: para cada token de la consulta busca el token del documento más parecido y suma esas máximas similitudes.

$$s(q, d) = \sum_{i \in q} \max_{j \in d} \text{sim}(q_i, d_j)$$

SÍMBOLO	SIGNIFICADO
q_i	Vector del token i de la consulta.
d_j	Vector del token j del documento.
$\max_{j \in d}$	Para cada token de la consulta, su mejor coincidencia en el documento.
$\sum_{i \in q}$	Suma de esas coincidencias sobre todos los tokens de la consulta.

La ventaja es precisión: al comparar token a token, capta coincidencias finas que un solo vector promediaría y perdería. El coste es memoria e índice: guardar decenas de vectores por documento multiplica el almacenamiento y exige un índice especializado, por lo que en la práctica suele usarse como **reordenador** sobre los candidatos de un bi-encoder, igual que el cross-encoder del capítulo 07, pero más barato de servir. La decisión es la de siempre: la interacción tardía compensa cuando tu evaluación muestra que el bi-encoder pierde matices que importan; si no, su coste de almacenamiento no se justifica.

MaxSim: cada token busca su mejor pareja

Similitud entre cada token de la consulta (filas) y cada token del documento (columnas)

	d ₁	d ₂	d ₃	d ₄	máx
q ₁	0,2	0,9	0,1	0,3	0,9
q ₂	0,1	0,2	0,8	0,1	0,8
q ₃	0,7	0,1	0,2	0,3	0,7

puntuación = suma de los máximos = 0,9 + 0,8 + 0,7 = 2,4

IA para gente curiosa / Facsímil 04 / Capítulo 08 / 686f6c61

Diseñar el esquema de una colección

Antes de indexar, conviene escribir el contrato. Una colección debería responder estas preguntas:

DECISIÓN	PREGUNTA TÉCNICA	MALA SEÑAL
Identificador	Qué id estable permite reindexar sin duplicar?	IDs aleatorios sin relación con fuente y sección.
Texto	Guardamos texto completo o puntero?	Resultados sin cita recuperable.
Metadata	Qué campos se filtran de verdad?	Guardar metadata bonita que nunca se indexa.
Versión	Qué modelo, dimensión y fecha generó el vector?	Mezclar embeddings incompatibles.
Permisos	El filtro de acceso vive en la consulta?	Filtrar después de mostrar candidatos.
Vigencia	Cómo caduca o se reemplaza un documento?	Resultados de años anteriores en top-k.
Borrado	Qué significa borrar: vector, payload, texto y cache?	Quedan fragmentos recuperables por accidente.

Si usas PostgreSQL con `pgvector`, el esquema puede ser explícito:

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE chunks (
  id text PRIMARY KEY,
  source_id text NOT NULL,
  chunk_text text NOT NULL,
  embedding vector(768) NOT NULL,
  curso integer NOT NULL,
  rol text NOT NULL,
  vigente boolean NOT NULL,
  embedding_model text NOT NULL,
  indexed_at timestamptz NOT NULL DEFAULT now()
);

CREATE INDEX chunks_embedding_hnsw
ON chunks USING hnsw (embedding vector_cosine_ops);

CREATE INDEX chunks_metadata
ON chunks (curso, rol, vigente);

SELECT id, chunk_text
FROM chunks
WHERE curso = 2026
  AND rol IN ('estudiante', 'publico')
  AND vigente = true
ORDER BY embedding <=> $1
LIMIT 8;
```

El detalle importante no es memorizar esta sintaxis. El detalle es que el campo vectorial, los filtros y la versión viven juntos. Si cambias el modelo de embedding o la dimensión, no estás "actualizando una columna"; estás cambiando el espacio de búsqueda.

Cómo trabajar con bases vectoriales con criterio

Una base vectorial en producción necesita disciplina operativa. La parte difícil no es insertar el primer vector; es mantener el sistema correcto cuando cambian documentos, permisos, modelos y volumen.

PRÁCTICA	QUÉ HACES	POR QUÉ IMPORTA
IDs deterministas	Derivas el id de fuente, sección y versión.	Permite <code>upsert</code> idempotente y evita duplicados.
Versionado de embeddings	Guardas modelo, dimensión, normalización y fecha.	Permite reindexar y comparar variantes.
Doble índice temporal	Construyes el nuevo índice junto al anterior.	Evita cortar servicio mientras migras.
Borrado verificable	Compruebas que ids borrados no vuelven en top-k.	Evita respuestas con contenido retirado.
Filtros obligatorios	El backend añade filtros de permisos siempre.	El cliente no decide qué puede ver.
Evaluación continua	Mides recall, p95, coste y errores por segmento.	Detecta degradación antes que usuarios y usuarias.
Trazas de retrieval	Guardas consulta, filtros, ids, scores y versión.	Permite explicar por qué se recuperó algo.

Una buena pregunta de ingeniería: si mañana cambiamos de `all-MiniLM-L6-v2` a otro modelo de 1024 dimensiones, ¿qué pasos exactos hay que hacer? Si la respuesta no incluye reindexado, evaluación, cambio de versión y plan de retirada del índice anterior, falta diseño.

Frescura: por qué borrar es más difícil que insertar

Un índice no es una foto: cambia cada día. Insertar vectores nuevos es relativamente fácil, porque HNSW admite añadir nodos al grafo de forma incremental. **Borrar es el problema.** En un grafo HNSW, los demás nodos pueden tener aristas que apuntan al vector que quieres eliminar; quitarlo de golpe dañaría caminos de navegación. Por eso la mayoría de los sistemas no borran de verdad: marcan el vector con un **tombstone** (una lápida) y lo excluyen de los resultados, pero sigue ocupando memoria y participando en el grafo. Cuando se acumulan muchos tombstones, el índice se degrada (más memoria, peor recall) y hay que **reconstruirlo** periódicamente.

Las consecuencias prácticas: un documento «retirado» puede seguir vivo en el índice si tu flujo de borrado solo lo quita de la base de datos pero no del índice vectorial; actualizar un documento es en realidad borrar más insertar (con re-embedding si cambió el texto); y la consistencia entre tu fuente de verdad y el índice es eventual, no inmediata. Por eso el «borrado verificable» de la tabla anterior no es paranoia: es la única forma de saber que el contenido retirado no vuelve en un top-k.

Cuando no cabe en una máquina: sharding e índice en disco

A escala de cientos de millones o miles de millones de vectores, el índice deja de caber en la RAM de un solo servidor, y aparecen dos estrategias. La primera es el **sharding**: partir la colección entre varios nodos, lanzar la consulta a todos en paralelo y fusionar sus resultados (un patrón *scatter-gather*), con réplicas para disponibilidad y para absorber más consultas por segundo. La segunda es mover el índice al **disco**: sistemas como DiskANN guardan el grafo en SSD y usan la RAM solo para la parte caliente de la navegación, lo que permite servir miles de millones de vectores desde una máquina a cambio de algo más de latencia.¹⁹ Para la mayoría de proyectos esto es prematuro (con millones de vectores, una sola máquina sobra), pero conviene saber que el techo no es la API de embeddings, sino la memoria del índice, y que existe un plan cuando se alcanza.

Evaluar una base vectorial

Aquí evaluamos dos capas: la calidad de recuperación y la calidad operativa. La primera pregunta es "encuentro lo correcto?". La segunda es "lo encuentro dentro del contrato de producto?".

MÉTRICA	QUÉ MIDE	CÓMO SE CALCULA
Recall ANN@k	Cuánto se parece el índice aproximado al exacto.	Resultados ANN frente a búsqueda exacta.
Recall con filtro@k	Si aparecen documentos correctos cumpliendo metadata.	Casos con filtros obligatorios.
nDCG@k	Si los documentos más útiles suben arriba.	Relevancia graduada por posición.
p50, p95, p99	Latencia normal y de cola.	Tiempos por consulta real.
Tasa de resultados retirados	Cuántos resultados ya no deberían aparecer.	Hits con <code>vigente=false</code> o versión antigua.
Cobertura de permisos	Si cada consulta aplica el filtro correcto.	Trazas con usuario, rol y condición.
Coste por consulta	CPU, memoria, GPU o precio cloud.	Coste mensual dividido por consultas útiles.

Una prueba mínima compara tres rankings para las mismas consultas: exacto filtrado, ANN filtrado e híbrido filtrado. Si el ANN pierde documentos que el exacto encuentra, ajustas parámetros o cambias índice. Si el híbrido mejora consultas con siglas pero empeora consultas naturales, ajustas fusión o decides cuándo activarlo.

En el día a día

En una universidad, una base vectorial es lo que permite que el buscador de la intranet encuentre la normativa correcta aunque la persona no use las palabras exactas, y que solo le muestre la versión vigente de su curso. En soporte interno, es donde viven los miles de fragmentos de documentación que un asistente recupera antes de responder, con filtros por producto, idioma y permisos. En cualquier RAG de producción, la base vectorial es la pieza que decide qué evidencia llega al modelo: si recupera mal, el resto da igual.

La parte delicada es que casi nunca falla el coseno. Falla el filtro que no se aplicó, la versión antigua que nadie retiró, el documento que el índice aproximado no encontró porque alguien subió `ef_search` demasiado bajo, o el término exacto (`SAML` , `2026` , un número de factura) que la búsqueda solo densa diluyó. Por eso una base vectorial sería es tanto un problema de recuperación como de gobernanza de datos.

Por qué debería importarte

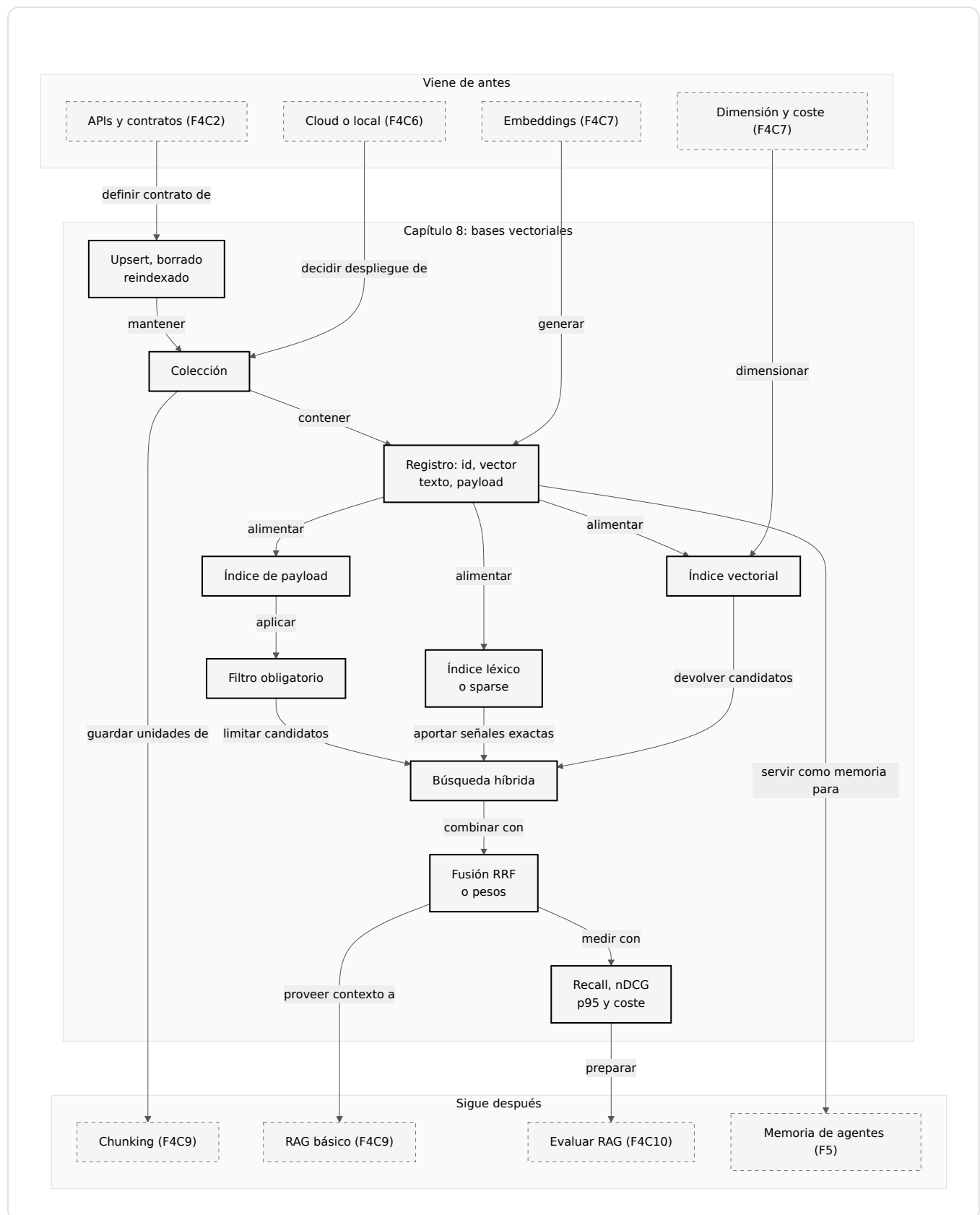
Porque es donde un RAG se rompe en silencio. Un modelo generativo excelente con una base vectorial mal diseñada produce respuestas seguras y equivocadas, y desde fuera parece un problema del modelo. Si entiendes esta pieza, sabes diagnosticar: si el fallo es de recuperación (no apareció el documento), de filtro (apareció uno que no debía), de índice (el ANN perdió un vecino), de fusión (lo léxico tapó lo semántico) o de versión (contenido retirado que sigue vivo).

También importa por coste y por escala. Diez millones de vectores no son «llamar a una API»: son decenas de gigas de memoria, índices que tardan en construirse, réplicas, copias de seguridad y una factura que crece con la dimensión y con cada filtro mal planificado. Elegir índice, cuantización y estrategia de filtrado con criterio es lo que separa un piloto que funciona en una demo de un sistema que aguanta usuarios reales.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Creer que vector DB equivale a RAG	La base recupera candidatos; no decide chunking, citas, abstención ni respuesta final.	Separar retrieval, contexto y generación.
Filtrar después de recuperar poco	Si pides top-10 global y luego descartas por permisos, puedes quedarte sin el resultado correcto.	Aplicar filtros como parte del plan de búsqueda.
No versionar embeddings	Mezclar modelos o dimensiones hace que las distancias dejen de significar lo mismo.	Guardar <code>embedding_model</code> , dimensión y fecha en cada registro.
Olvidar términos exactos	Siglas, códigos, IDs y nombres propios pueden perderse en búsqueda solo densa.	Probar búsqueda híbrida con BM25 o sparse vectors.
Medir solo latencia media	p95 o p99 pueden ser malos aunque la media parezca aceptable.	Medir percentiles y separar consultas con filtros complejos.
No probar borrados	Un índice puede seguir devolviendo contenido retirado si el flujo de borrado falla.	Crear tests de ids retirados y verificar que no aparecen.
Elegir herramienta por moda	Cada base cambia filtros, operación, costes, backup y SQL disponible.	Comparar con una matriz de requisitos reales.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Base vectorial	Sistema que guarda vectores y permite recuperarlos por similitud.
Colección	Conjunto de registros con el mismo contrato de vector y metadata.
Payload	Metadata asociada al vector, usada para filtrar y explicar resultados.
Índice vectorial	Estructura que acelera la búsqueda de vecinos cercanos.
HNSW	Índice por grafo navegable para búsqueda aproximada.
IVFFlat	Índice que divide el espacio en listas y busca solo algunas.
Filtro	Condición que limita qué registros pueden entrar en el ranking.
BM25	Ranking léxico basado en frecuencia, rareza y longitud de documento.
Búsqueda híbrida	Combinación de ranking vectorial y ranking léxico.
RRF	Fusión de rankings basada en posiciones.
Upsert	Insertión o actualización idempotente de un registro.

Antes de pasar página

- ☐ ¿Puedo explicar por qué una base vectorial no arregla embeddings malos? (Si no, vuelve a «Qué no es una base vectorial».)
- ☐ ¿Sé qué debe guardar un registro vectorial además del vector? (Si no, vuelve a «Qué sí es: un contrato de recuperación».)
- ☐ ¿Puedo calcular el coste bruto de memoria para N , d y bytes por componente? (Si no, vuelve a «El coste real: vectores, índices y payload».)
- ☐ ¿Sé explicar qué mide la selectividad de un filtro? (Si no, vuelve a «El coste real: vectores, índices y payload».)
- ☐ ¿Sé distinguir búsqueda exacta, HNSW, IVFFlat y PQ? (Si no, vuelve a «Índices: exacto, HNSW, IVFFlat y compresión».)
- ☐ ¿Sé por qué filtrar después de recuperar poco puede fallar? (Si no, vuelve a «Filtros: dónde se gana o se rompe la recuperación».)

- ☐ ¿Sé cuándo BM25 aporta algo que el embedding puede perder? (Si no, vuelve a «Búsqueda híbrida: cuando exactitud léxica y semántica se necesitan».)
- ☐ ¿Puedo explicar RRF sin comparar puntuaciones incompatibles? (Si no, vuelve a «Búsqueda híbrida: cuando exactitud léxica y semántica se necesitan».)
- ☐ ¿Sé qué campos versionar para reindexar con seguridad? (Si no, vuelve a «Diseñar el esquema de una colección».)
- ☐ ¿Sé qué métricas mirar además de recall: p95, coste y resultados retirados? (Si no, vuelve a «Evaluar una base vectorial».)

En resumen

IDEA FUERZA	DETALLE
Una base vectorial es un contrato operativo.	Guarda vectores, texto, ids, metadata, versiones e índices.
El filtro forma parte de la respuesta correcta.	Un resultado cercano pero fuera de curso, rol o vigencia no es válido.
El índice aproximado debe compararse con exacto.	Sin baseline exacto, no sabes cuánto recall pierdes por ganar latencia.
La búsqueda híbrida une dos señales.	Embeddings capturan significado; BM25 protege términos exactos.
Operar importa tanto como buscar.	Upsert, borrado, snapshots, reindexado y trazas deciden si el sistema aguanta.
La evaluación debe incluir producto.	Recall, nDCG, filtros, p95, coste y documentos retirados cuentan juntos.

Para saber más

Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). *Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods*. SIGIR, 758-759.

<https://doi.org/10.1145/1571941.1572114>

Jégou, H., Douze, M. y Schmid, C. (2011). *Product Quantization for Nearest Neighbor Search*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117-128.

<https://doi.org/10.1109/TPAMI.2010.57>

Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-Scale Similarity Search with GPUs*. *IEEE Transactions on Big Data*, 7(3), 535-547. <https://doi.org/10.1109/TBDATA.2019.2921572>

Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs*. *IEEE TPAMI*, 42(4), 824-836. <https://doi.org/10.1109/TPAMI.2018.2889473>

Milvus. (2026). *Filtered Search*. <https://milvus.io/docs/filtered-search.md>

pgvector. (2026). *pgvector: Open-source vector similarity search for Postgres*. <https://github.com/pgvector/pgvector>

Pinecone. (2026). *Hybrid search*. <https://docs.pinecone.io/docs/hybrid-search-and-sparse-vectors>

Qdrant. (2026). *Indexing*. <https://qdrant.tech/documentation/manage-data/indexing/>

Robertson, S. y Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond*. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. <https://doi.org/10.1561/15000000019>

Weaviate. (2026). *Hybrid search*. <https://docs.weaviate.io/weaviate/search/hybrid>

Notas

1. Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-Scale Similarity Search with GPUs*. *IEEE Transactions on Big Data*, 7(3), 535-547. <https://doi.org/10.1109/TBDATA.2019.2921572>. ↵
2. Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs*. *IEEE TPAMI*, 42(4), 824-836. <https://doi.org/10.1109/TPAMI.2018.2889473>. ↵
3. Qdrant. (2026). *Indexing*. <https://qdrant.tech/documentation/manage-data/indexing/>. Consultado el 25 de mayo de 2026. ↵
4. pgvector. (2026). *pgvector: Open-source vector similarity search for Postgres*. <https://github.com/pgvector/pgvector>. Consultado el 25 de mayo de 2026. ↵
5. Weaviate. (2026). *Hybrid search*. <https://docs.weaviate.io/weaviate/search/hybrid>. Consultado el 25 de mayo de 2026. ↵
6. Milvus. (2026). *Filtered Search*. <https://milvus.io/docs/filtered-search.md>. Consultado el 25 de mayo de 2026. ↵
7. Pinecone. (2026). *Hybrid search*. <https://docs.pinecone.io/docs/hybrid-search-and-sparse-vectors>. Consultado el 25 de mayo de 2026. ↵

8. Malkov y Yashunin, 2020. ↵
9. pgvector, 2026. ↵
10. Jégou, H., Douze, M. y Schmid, C. (2011). *Product Quantization for Nearest Neighbor Search*. *IEEE TPAMI*, 33(1), 117-128. <https://doi.org/10.1109/TPAMI.2010.57>. ↵
11. Martin Aumüller, Erik Bernhardsson y Alexander Faithfull. (2020). *ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms*. *Information Systems*, 87. <https://doi.org/10.1016/j.is.2019.02.006>. ↵
12. Qdrant, 2026. ↵
13. Milvus, 2026. ↵
14. Robertson, S. y Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond*. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. <https://doi.org/10.1561/15000000019>. ↵
15. Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). *Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods*. *SIGIR*, 758-759. <https://doi.org/10.1145/1571941.1572114>. ↵
16. Weaviate, 2026. ↵
17. Pinecone, 2026. ↵
18. Omar Khattab y Matei Zaharia. (2020). *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT*. *SIGIR*. <https://arxiv.org/abs/2004.12832>. ↵
19. Suhas Jayaram Subramanya y otros (2019). *DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node*. *NeurIPS 2019*. Implementación de referencia: <https://github.com/microsoft/DiskANN>. ↵

Capítulo 09: RAG básico: chunking, retrieval, citas y abstención

El momento en que buscar ya no basta

En el capítulo 08 aprendimos a guardar fragmentos, buscarlos con vectores, filtrarlos por metadata y combinar señal densa con BM25. Eso devuelve candidatos. Un RAG hace algo más delicado: convierte esos candidatos en una respuesta útil, citada y capaz de decir “no tengo evidencia suficiente”.

Imagina un asistente para alumnado. La persona pregunta: “¿puedo ampliar matrícula en septiembre si tengo pagos pendientes?”. El sistema no debería inventar una política general. Debe recuperar normativa vigente, decidir qué fragmentos entran en contexto, responder solo con lo que esos fragmentos sostienen y enseñar las fuentes.

RAG significa *retrieval-augmented generation*: generación aumentada por recuperación. La idea fue formulada como una forma de combinar modelos generativos con memoria no paramétrica recuperada desde un índice externo.¹ En producto, RAG no es una palabra bonita para “chat con PDFs”. Es una arquitectura para que el modelo responda con información que no tiene por entrenamiento, que cambia con el tiempo o que pertenece a una organización concreta.

Cuando decimos “memoria no paramétrica” estamos diciendo algo muy concreto: el conocimiento no está guardado dentro de los pesos del modelo. Está fuera, en documentos, tablas, índices, bases de datos o sistemas que podemos actualizar sin volver a entrenar el modelo.

Estado del arte con fecha de corte

Fecha de corte: 25 de mayo de 2026.

Fuentes consultadas ese día: documentación oficial de OpenAI File Search, LangChain, LlamaIndex, Haystack, Google Vertex AI RAG Engine, Azure AI Search, Pinecone, Weaviate, Qdrant y pgvector; y trabajos académicos sobre RAG, BM25 y fusión de rankings.

Lo estable es el patrón: ingestión, partición, indexado, recuperación, construcción de contexto, generación, citas y evaluación. Lo cambiante son proveedores, límites, APIs, modelos de embeddings, formatos soportados, coste, residencia de datos, rerankers y herramientas gestionadas.

FUENTE	QUÉ APORTA	CÓMO USARLA
RAG original. ²	Formaliza el uso de documentos recuperados como memoria externa para generación.	Para entender que RAG no es solo “meter documentos en un prompt”.
OpenAI File Search. ³	Herramienta gestionada de la Responses API para buscar archivos en vector stores y devolver contexto al modelo.	Para montar rápido un RAG alojado sin programar todo el retrieval.
LangChain Retrieval. ⁴	Describe loaders, splitters, embeddings, vector stores, retrievers y arquitecturas 2-step, agentic e híbridas.	Para componer aplicaciones RAG con piezas intercambiables.
LlamaIndex RAG. ⁵	Ordena RAG en loading, indexing, storing, querying y evaluation; introduce Documents, Nodes y retrievers.	Para proyectos centrados en ingestión de datos y gestión de índices.
Haystack pipelines. ⁶	Modela RAG como grafo de componentes con ramas, validación y flujos de indexado/consulta.	Para equipos que quieren pipelines explícitos y desplegables.
Vertex AI RAG Engine. ⁷	Servicio gestionado de RAG dentro de Vertex AI.	Para entornos Google Cloud donde operación, permisos y plataforma pesan.
Azure AI Search. ⁸	Combina búsqueda tradicional, vectorial, semántica, filtros e integración con escenarios generativos.	Para organizaciones ya montadas sobre Azure y Microsoft Learn.
Pinecone y Weaviate. ⁹	Bases vectoriales gestionadas con patrones RAG, búsqueda semántica, filtros y opciones híbridas.	Para delegar índice y escalado sin entregar el diseño del sistema.
BM25 y RRF. ¹⁰	Dan bases sólidas para retrieval léxico y fusión de rankings.	Para no construir RAG solo con embeddings.

Qué no es RAG

RAG no es subir un PDF a un chat y confiar. Si el PDF se trocea mal, si la búsqueda trae el fragmento equivocado o si el prompt permite responder sin evidencia, el sistema puede fallar con mucha seguridad aparente.

Tampoco es fine-tuning. Ajustar un modelo puede enseñar formato, tono o una tarea repetida; RAG aporta contexto externo en tiempo de consulta. Si la información cambia cada semana, suele ser mejor actualizar documentos e índice que reentrenar.

RAG tampoco elimina la necesidad de permisos. Recuperar un fragmento privado y luego pedir al modelo que “no lo mencione” es una mala frontera. El filtrado debe ocurrir antes de que el texto entre en contexto.

RAG, memoria y entrenamiento no son lo mismo

Esta diferencia es fundamental. Si la mezclamos, acabamos usando RAG para lo que pide entrenamiento, fine-tuning para lo que pide documentos, o “memoria” para cosas que deberían ser permisos, trazas o contexto recuperado.

El modelo base tiene conocimiento en sus pesos. Eso viene del entrenamiento: grandes cantidades de datos, mucho cálculo y una actualización difícil de repetir para una aplicación pequeña. Cuando haces fine-tuning o LoRA, sigues tocando comportamiento aprendido: formato, estilo, patrones de respuesta, especialización en una tarea repetida. RAG, en cambio, no cambia los pesos. Recupera evidencia externa en el momento de la pregunta.

MECANISMO	DÓNDE VIVE LA INFORMACIÓN	CUÁNDO CAMBIA	PARA QUÉ SIRVE
Entrenamiento base	En los pesos del modelo.	Antes de que tú uses el modelo.	Capacidades generales: lenguaje, razonamiento, código, patrones del mundo.
Fine-tuning / LoRA	En pesos ajustados o adaptadores.	Cuando entrenas con ejemplos.	Formato, tono, clasificación estable, tareas repetidas y medibles.
RAG	En un corpus externo recuperable.	Cuando cambian documentos o índices.	Conocimiento privado, vigente, citable o cambiante.
Memoria de conversación	En mensajes previos o resúmenes guardados.	Durante una sesión o entre sesiones si se persiste.	Preferencias, contexto personal y continuidad conversacional.
Caché	En una capa técnica de reutilización.	Mientras sea válida.	Ahorrar coste o latencia; no añade conocimiento nuevo.
Tool	En un sistema externo consultado o ejecutado.	En tiempo real.	Calcular, buscar estado vivo, escribir en sistemas o consultar APIs.

Ejemplo sencillo: si una universidad cambia la normativa de matrícula, no quieres reentrenar un modelo. Quieres actualizar el documento, reindexar y que las respuestas citen la normativa nueva. Si, en cambio, el problema es que el modelo siempre devuelve un JSON con campos mal nombrados, RAG no lo arregla por sí solo: quizá necesitas mejor contrato de salida, ejemplos, validación o ajuste.

La memoria conversacional tampoco equivale a RAG. Si el usuario dice “soy estudiante de segundo”, eso puede vivir como memoria o como contexto de sesión. Pero si pregunta “¿qué dice la normativa vigente sobre ampliación?”, eso debe salir del corpus, con fecha y cita. La memoria puede ayudar a formular la consulta; no debe sustituir la fuente.

Por qué usar un RAG

Usamos RAG cuando responder bien exige recuperar evidencia externa. No es una moda de arquitectura; es una respuesta a una limitación práctica: los modelos no traen dentro todos tus documentos, no conocen todos los cambios recientes y no pueden demostrar por sí solos de dónde sale una afirmación.

MOTIVO	QUÉ PROBLEMA RESUELVE	SEÑAL DE QUE RAG ENCAJA
Información cambiante	El conocimiento se actualiza sin tocar pesos.	Normativas, catálogos, precios internos, políticas y manuales vivos.
Conocimiento privado	El modelo no vio tus documentos durante entrenamiento.	Intranets, tickets, expedientes, documentación de producto.
Citas y auditoría	La respuesta puede revisarse contra fuentes.	Necesitas enseñar página, sección, fecha o documento.
Permisos	Cada persona puede recuperar solo lo que le corresponde.	Hay roles, grupos, tenants, cursos, clientes o áreas.
Coste de actualización	Reindexar suele ser más barato que reentrenar.	El corpus cambia más que el comportamiento deseado.
Especialización ligera	El modelo general se apoya en contexto de dominio.	La tarea pide lenguaje natural más conocimiento concreto.
Depuración	Puedes ver qué documentos entraron en la respuesta.	Necesitas saber si falló búsqueda, contexto o generación.

También hay casos donde RAG no es la primera respuesta.

SITUACIÓN	MEJOR PRIMERA HERRAMIENTA
El modelo no sigue el formato.	Prompt, ejemplos, salida estructurada o fine-tuning.
La respuesta depende de cálculo exacto.	Tool o código, no solo documentos.
La información vive en una base de datos transaccional.	SQL, API o tool; quizá luego RAG para explicar resultados.
La tarea es siempre igual y estable.	Fine-tuning/LoRA puede ser más eficiente.
El corpus está desordenado o sin dueño.	Gobernanza documental antes de RAG.

Qué sí es: una cadena verificable

Un RAG mínimo tiene dos procesos separados. El primero prepara el conocimiento; el segundo responde una pregunta.

PROCESO	OCURRE CUÁNDO	QUÉ PRODUCE
Indexación	Antes de la consulta, cuando entran o cambian documentos.	Chunks con embeddings, texto, metadata y versión.
Consulta	Cuando una persona pregunta.	Respuesta con citas, o abstención si no hay evidencia suficiente.

Esta es la formulación *retrieve-then-generate* que introdujo el trabajo fundacional de RAG: primero se recupera evidencia relevante con un recuperador y después un generador la condiciona para producir la respuesta.¹¹ En tres pasos:

$$q = f_{\theta}(x)$$

$$R_k = \text{TopK}(q, C, F, k)$$

$$y = g_{\phi}(x, \text{Contexto}(R_k))$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Pregunta original.	“¿puedo ampliar matrícula en septiembre?”.
f_{θ}	Modelo de embeddings.	Convierte la pregunta en vector.
q	Vector de la pregunta.	768 números.
C	Colección de chunks indexados.	Normativa, FAQs y manuales.
F	Filtros obligatorios.	curso=2026 , vigente=true , rol=estudiante .
R_k	Fragmentos recuperados.	Top 6 chunks después de filtros.
g_{ϕ}	Modelo generativo.	LLM que redacta la respuesta.
y	Respuesta final.	Texto con citas o abstención.

La parte peligrosa está en $\text{Contexto}(R_k)$. No todos los chunks recuperados deben entrar en el prompt. Hay que ordenar, recortar, quitar duplicados, proteger permisos, preservar citas y dejar hueco para instrucciones y respuesta.

Términos que no podemos dar por sabidos

La jerga de RAG engaña porque muchas palabras parecen pequeñas y en realidad esconden decisiones de sistema. “Chunk”, “metadata” o “top-k” no son etiquetas académicas: son puntos donde puedes ganar o perder calidad, coste, privacidad y capacidad de depuración.

La primera familia de términos aparece antes de preguntar nada. Es la parte de preparación del conocimiento.

TÉRMINO	QUÉ SIGNIFICA DE VERDAD	EN EL EJEMPLO DE MATRÍCULA
Corpus	Conjunto de fuentes que el sistema tiene permiso para consultar. No es “todo lo que existe”; es lo que has decidido meter en el sistema.	Normativa 2026, FAQ de secretaría, calendario académico y manual de trámites.
Fuente	Documento o sistema original del que sale la información. Puede ser un PDF, una web, una tabla o una base de datos.	<code>normativa_matricula_2026.pdf</code> .
Documento	Representación interna de una fuente. Suele incluir texto, título, fecha, URL, propietario y versión.	La normativa convertida a texto con su fecha de publicación.
Parser	Programa que extrae texto y estructura. Si el parser lee mal una tabla, el RAG recuperará texto defectuoso.	Sacar de un PDF el artículo, el título y los apartados en orden correcto.
OCR	Reconocimiento óptico de caracteres. Convierte una imagen o escaneo en texto.	Una normativa escaneada que no permite seleccionar texto.
Chunk	Fragmento recuperable. Debe ser suficientemente pequeño para buscar bien y suficientemente completo para citarlo.	Un artículo completo sobre ampliación de matrícula.
Token	Unidad interna aproximada de texto que usan modelos y muchos contadores de coste. No siempre coincide con palabra.	“matrícula” puede ocupar más de un token según el tokenizador.
Solape	Parte repetida entre dos chunks consecutivos para no cortar ideas.	Repetir el encabezado y la frase anterior cuando un artículo pasa de una ventana a otra.
Metadato	Datos sobre el chunk, no necesariamente contenido para responder. Sirve para filtrar, citar y operar.	<code>curso=2026 , vigente=true , seccion=matricula , pagina=12</code> .
ACL	Reglas de acceso. Indican quién puede recuperar un chunk antes de que llegue al modelo.	Alumnado ve normativa pública; personal interno ve notas de gestión.
Hash	Huella calculada del texto. Si el documento cambia, cambia el hash.	Detectar que se subió una normativa nueva aunque mantenga el mismo nombre de archivo.
Versión de corpus	Identificador de qué conjunto de documentos e índices se usó. Es clave para reproducir una respuesta.	<code>matricula-2026-v3</code> , usado el 25 de mayo de 2026.

La segunda familia aparece cuando alguien pregunta. Aquí conviene separar “buscar parecido” de “encontrar evidencia”.

TÉRMINO	QUÉ SIGNIFICA DE VERDAD	QUÉ DECISIÓN EXIGE
Query	Consulta que entra al retrieval. Puede ser la pregunta original o una versión reescrita.	Decidir si buscas literalmente “pagos pendientes” o reformulas con sinónimos.
Query rewrite	Reescritura de la pregunta para recuperar mejor. Es útil, pero debe registrarse.	Convertir “¿puedo ampliar?” en “ampliación de matrícula septiembre pagos pendientes”.
Embedding	Vector numérico que representa un texto para comparar significado aproximado.	Elegir modelo, dimensión, coste y cuándo recalcular vectores.
Dimensión	Número de coordenadas del embedding. Más dimensión no garantiza mejor resultado; cambia memoria, coste e índice.	Un embedding de 768 dimensiones ocupa menos que uno de 3.072, pero puede rendir distinto.
Índice	Estructura que permite buscar rápido. Sin índice, compararías contra todo el corpus en cada pregunta.	Crear índice vectorial, índice léxico o ambos.
Vector store	Almacén que guarda vectores, texto y metadata para buscar por similitud y filtros.	Qdrant, pgvector, Pinecone, Weaviate o File Search.
FTS	<i>Full-text search</i> : búsqueda textual clásica por palabras, operadores y relevancia.	Encontrar “artículo 14” o “pagos pendientes” aunque el embedding no lo priorice.
BM25	Fórmula de ranking léxico. Premia términos relevantes y penaliza documentos donde una palabra aparece por aparecer.	Si la pregunta dice “septiembre”, BM25 ayuda a subir chunks que contienen esa palabra exacta.
ANN	<i>Approximate nearest neighbors</i> . Técnica para buscar vectores parecidos sin comparar todos contra todos.	Acelerar búsqueda en miles o millones de chunks aceptando una aproximación controlada.
Top-k	Número de candidatos devueltos. <code>k=5</code> trae cinco chunks antes de rerank o contexto.	Si <code>k</code> es bajo, quizá pierdes evidencia; si es alto, sube ruido y coste.
Score	Puntuación de similitud o relevancia. No siempre es comparable entre métodos.	No comparar sin más un coseno <code>0,78</code> con un BM25 <code>12,4</code> .
RRF	Fusión por posiciones. Combina rankings usando el puesto de cada documento, no su score bruto.	Mezclar búsqueda vectorial y BM25 sin calibrar escalas distintas.
Reranker	Modelo o regla que reordena candidatos después de una primera búsqueda rápida.	Pasar de 50 candidatos baratos a 6 fragmentos buenos para el prompt.
Filtro	Restricción obligatoria antes o durante la búsqueda. No es una preferencia.	<code>vigente=true</code> , <code>curso=2026</code> , <code>rol=estudiante</code> .

La tercera familia aparece cuando ya hay candidatos y el sistema debe responder. Aquí es donde un RAG deja de ser buscador y se convierte en producto.

TÉRMINO	QUÉ SIGNIFICA DE VERDAD	QUÉ SE REvisa EN PRODUCCIÓN
Context builder	Pieza que decide qué chunks entran en el prompt y con qué formato.	Deduplicación, orden, citas, presupuesto y prioridad de fuentes.
Presupuesto de contexto	Límite de tokens disponible para instrucciones, pregunta, evidencia y respuesta.	No gastar 90% del contexto en fragmentos repetidos.
Cita	Enlace entre una afirmación y el fragmento que la sostiene.	Que [F1] apunte a source_id , página, sección y hash.
Grounding	Grado en que la respuesta está apoyada en la evidencia recuperada.	Si la frase importante se puede subrayar en una fuente.
Abstención	Respuesta correcta cuando falta evidencia suficiente.	Decir qué dato falta en vez de rellenarlo con probabilidad.
Umbral τ	Valor mínimo de soporte para responder. No se elige a ojo; se calibra con evaluación.	Responder si soporte $\geq 0,65$ y abstenerse si queda por debajo.
Traza	Registro completo de una consulta. Sin traza, no sabes dónde falló.	Pregunta, filtros, top-k, scores, prompt, respuesta, modelo y coste.
Recall@k	Métrica: si la evidencia necesaria aparece entre los k primeros resultados.	Si la respuesta correcta necesitaba un artículo y no aparece en top 10, falló retrieval.
nDCG	Métrica que premia que los resultados más útiles aparezcan arriba.	No basta con traer el chunk correcto; conviene que llegue en primeras posiciones.

Una forma sencilla de recordarlo: el corpus decide **qué puede saber** el sistema; el retrieval decide **qué encuentra**; el context builder decide **qué lee** el modelo; y la abstención decide **qué no debe fingir**.

Elementos importantes de un RAG

Un RAG no es una sola pieza. Es una cadena. Si una parte falla, el resultado final puede parecer correcto y estar mal apoyado. Por eso conviene nombrar los elementos con precisión.

ELEMENTO	PREGUNTA QUE RESPONDE	ERROR TÍPICO
Corpus	¿Qué fuentes entran y cuáles quedan fuera?	Indexar documentos sin vigencia, duplicados o sin propietario.
Ingesta	¿Cómo entran los documentos al sistema?	Subir archivos manualmente sin versiones ni borrado.
Parsing	¿El texto extraído conserva estructura?	Perder tablas, encabezados, notas o páginas.
Chunking	¿Cuál es la unidad recuperable?	Cortar ideas por tamaño fijo sin respetar secciones.
Metadata	¿Cómo filtro, cito y opero cada chunk?	Guardar solo texto y no poder filtrar por fecha, rol o fuente.
Embeddings	¿Cómo busco por significado aproximado?	Usar un modelo sin evaluar idioma, dominio, dimensión y coste.
Índice léxico	¿Cómo busco palabras exactas?	Confiar solo en vectores y perder códigos, fechas o términos raros.
Retrieval híbrido	¿Cómo combino significado, palabras y filtros?	Mezclar scores incompatibles sin fusión ni trazas.
Reranking	¿Cómo reordeno candidatos prometedores?	Meter al prompt los primeros resultados sin segunda revisión.
Context builder	¿Qué evidencia entra al prompt?	Pasar demasiados chunks, repetidos o sin citas.
Generación	¿Cómo redacta el modelo con restricciones?	Permitir respuesta sin evidencia o sin formato de cita.
Abstención	¿Cuándo no se responde?	Contestar siempre aunque el corpus no contenga la respuesta.
Evaluación	¿Cómo sé si mejora?	Medir solo “me gusta” y no recall, groundedness, coste o latencia.
Observabilidad	¿Cómo depuro cada consulta?	No guardar query, filtros, ranking, contexto y respuesta.

El orden importa. No tiene sentido discutir modelos grandes si el parser rompe tablas. No tiene sentido ajustar el prompt si el retrieval no trae el artículo correcto. No tiene sentido comprar una base vectorial si nadie sabe qué documentos están vigentes.

Dimensiones en un RAG

En RAG, “dimensión” puede significar dos cosas. La primera es matemática: la dimensión del embedding. La segunda es de diseño: las dimensiones que debes controlar para que el sistema funcione.

La dimensión matemática es el número de coordenadas del vector. Si un chunk se convierte en un embedding de d dimensiones, queda así:

$$e(c) = [e_1, e_2, e_3, \dots, e_d]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
c	Chunk que queremos representar.	Artículo sobre ampliación de matrícula.
$e(c)$	Embedding del chunk.	Vector guardado en el índice.
d	Número de dimensiones.	768, 1.024, 1.536 o 3.072, según modelo.
e_i	Valor de una coordenada.	Un número flotante aprendido por el modelo.

Estas dimensiones no son columnas humanas como “matrícula”, “pago” o “septiembre”. Son coordenadas aprendidas. El significado está distribuido por muchas posiciones a la vez. Por eso no se interpreta una dimensión aislada como si fuera una etiqueta; se compara el vector completo con otros vectores.

La similitud suele medirse con el coseno, la medida clásica del modelo de espacio vectorial de la recuperación de información:¹²

$$\text{sim}(a, b) = \frac{a \cdot b}{\|a\| \|b\|}$$

IDEA	QUÉ IMPLICA
Más dimensiones no significa automáticamente mejor RAG.	Puede mejorar representación, pero también memoria, latencia y coste.
No mezcles modelos de embeddings en el mismo índice sin control.	Dos modelos pueden tener dimensiones y geometrías incompatibles.
Si cambias de modelo de embeddings, normalmente reindexas.	Los vectores antiguos ya no viven en el mismo espacio.
La dimensión afecta almacenamiento.	Más coordenadas por chunk implica más bytes y más trabajo para el índice.
La dimensión afecta recuperación, no generación directamente.	Ayuda a encontrar evidencia; no hace que el LLM razone mejor por sí sola.

El coste bruto de guardar vectores puede aproximarse así:

$$\text{bytes} \approx N \times d \times b$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
N	Número de chunks.	100.000 chunks.
d	Dimensiones por embedding.	1.536 dimensiones.
b	Bytes por valor.	4 bytes en float32, 2 bytes en float16.

Con 100.000 chunks, 1.536 dimensiones y float32, solo los vectores ocupan aproximadamente 614 MB antes de contar metadata, texto, índices auxiliares y réplicas. Esta cuenta no decide la arquitectura, pero te obliga a pensar como ingeniero: dimensión, volumen, tipo numérico, latencia y presupuesto van juntos.

La segunda lectura de “dimensiones” es de diseño. Un RAG se optimiza mirando varias dimensiones a la vez:

DIMENSIÓN DE DISEÑO	PREGUNTA
Calidad del corpus	¿Los documentos son correctos, vigentes y no duplicados?
Recuperación	¿La evidencia necesaria aparece en top-k?
Contexto	¿El prompt recibe lo justo, ordenado y citado?
Generación	¿El modelo responde con contrato, citas y abstención?
Evaluación	¿Sabemos medir si mejora o empeora?
Operación	¿Podemos actualizar, borrar, auditar y controlar coste?

Chunking: partir sin romper el significado

Un chunk es la unidad que el sistema puede recuperar. Si es demasiado pequeño, pierde contexto. Si es demasiado grande, arrastra ruido y ocupa mucho prompt. La unidad correcta depende del documento y de la pregunta.

Partir un documento en ventanas con solape es una ventana deslizante, y el número de chunks sale de una cuenta combinatoria exacta. Si el documento tiene L tokens, la ventana mide w y el solape es o , cada ventana avanza $w - o$ tokens, así que el número de chunks es:

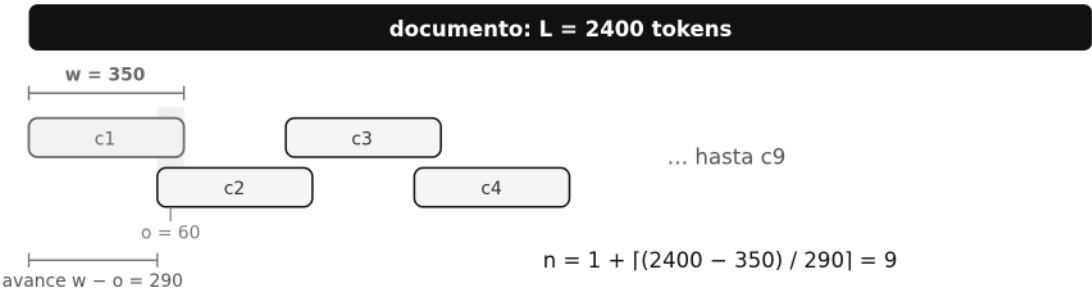
$$n = 1 + \left\lceil \frac{\max(0, L - w)}{w - o} \right\rceil$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Longitud del documento.	2.400 tokens.
w	Tamaño de chunk.	350 tokens.
o	Solape entre chunks.	60 tokens.
$w - o$	Avance real de cada ventana.	290 tokens.
n	Número de chunks.	9 chunks.

El solape evita cortar una idea justo en la frontera. Pero el solape también duplica texto, embeddings y coste. Si todo solapa demasiado, el índice se llena de fragmentos casi iguales y el top-k pierde diversidad.

Chunking con solape: una ventana deslizante

Ventanas de w tokens que avanzan $w - o$; el solape o evita cortar una idea



IA para gente curiosa / Facsímil 04 / Capítulo 09 / 686f6c61

TIPO DE DOCUMENTO	CHUNK INICIAL RAZONABLE	QUÉ CUIDAR
FAQ corta	Una pregunta-respuesta por chunk.	Mantener la pregunta original en el texto.
Normativa	Artículo, sección o bloque con título.	Guardar fecha, versión, capítulo y vigencia.
Manual técnico	Sección con pasos completos.	No separar requisito, comando y salida esperada.
Contrato o política	Cláusula completa con encabezado.	Preservar definiciones y excepciones.
Código	Función, clase o bloque lógico.	Mantener ruta, lenguaje y dependencias cercanas.

Un buen chunk no es “350 tokens”. Un buen chunk es una pieza que, leída sola, todavía puede sostener una respuesta concreta.

Un buen chunk se sostiene solo

No es cuestión de tamaño: leído aislado, todavía debe poder responder y citarse

MAL · cortar por tamaño, en mitad de una idea

«...la ampliación de matrícula se con-

-cede si el pago está al día...»

Ningún trozo responde solo y la cita no sostiene la frase.

BIEN · unidad completa con encabezado y vigencia

Art. 14 · Ampliación de matrícula (curso 2026): se concede si el pago está al día. Plazo de solicitud: septiembre.

Leído solo ya responde y se puede citar: [norm-2026 · Art. 14].

IA para gente curiosa / Facsímil 04 / Capítulo 09 / 686f6c61

Más allá de la ventana fija

La ventana deslizante con solape es el punto de partida, pero tiene un defecto: corta donde toca por tamaño, no por sentido. Hay dos estrategias que un ingeniero debería conocer porque resuelven la tensión entre «recuperar bien» (trozos pequeños y específicos) y «dar buen contexto» (trozos grandes y completos):

ESTRATEGIA	CÓMO FUNCIONA	QUÉ RESUELVE
Chunking fijo con solape	Ventanas de w tokens con solape o .	Simple y predecible; el punto de partida.
Chunking semántico	Corta donde cambia el tema (por ejemplo, cuando la similitud entre frases consecutivas baja de un umbral).	Evita partir una idea por la mitad por culpa del tamaño.
Parent-document (<i>small-to-big</i>)	Indexas y recuperas trozos pequeños y específicos, pero al LLM le entregas el bloque «padre» (la sección completa) del que salió el trozo.	Combina la precisión de búsqueda de lo pequeño con el contexto de lo grande.

La regla no cambia: la unidad de recuperación y la unidad de contexto no tienen por qué ser la misma, y separarlas suele mejorar tanto el recall como la calidad de la respuesta. Como siempre, se decide midiendo `Recall@k` y la calidad final en tu corpus, no por moda.

Soluciones de terceros: qué comprar, qué montar y qué no delegar

Hay varias formas de montar RAG. La decisión no es “framework sí o no”. La decisión es qué parte quieres delegar y qué parte necesitas controlar.

FAMILIA	EJEMPLOS	TE QUITA TRABAJO EN	TE DEJA RESPONSABLE DE
RAG gestionado por proveedor	OpenAI File Search, Vertex AI RAG Engine.	Vector store, búsqueda, integración con modelo, parte de la operación.	Calidad de documentos, permisos, evaluación, costes y trazabilidad.
Framework de orquestación	LangChain, LlamaIndex, Haystack.	Loaders, splitters, retrievers, pipelines, integraciones.	Diseño del flujo, selección de componentes y despliegue.
Base vectorial / buscador	Qdrant, pgvector, Pinecone, Weaviate, Milvus, Elasticsearch/OpenSearch.	Almacenamiento, índice, filtros, latencia de búsqueda.	Chunking, generación, citas, abstención y evaluación final.
Parsing y preparación	Extractores PDF, OCR, conversores HTML, pipelines ETL.	Sacar texto de documentos difíciles.	Validar tablas, orden de lectura, duplicados y metadatos.
Observabilidad y evaluación	LangSmith, evaluadores propios, trazas internas.	Registro, comparación y análisis de runs.	Definir qué significa “respuesta correcta” en tu dominio.

OpenAI File Search es útil si quieres empezar rápido con archivos y vector stores gestionados dentro de la Responses API.¹³ LangChain encaja cuando quieres componer loaders, splitters, retrievers, vector stores y varios estilos de RAG.¹⁴ LlamaIndex brilla cuando el centro del problema es ingestión, nodos, índices y consulta sobre datos propios.¹⁵ Haystack es especialmente claro si quieres pensar en pipelines como grafos de componentes conectados y validables.¹⁶

La regla práctica: compra o usa framework para acelerar, pero no delegues el criterio. Ninguna herramienta sabe por defecto qué documentos están vigentes, qué permiso tiene cada persona, qué cita es suficiente o cuándo conviene abstenerse.

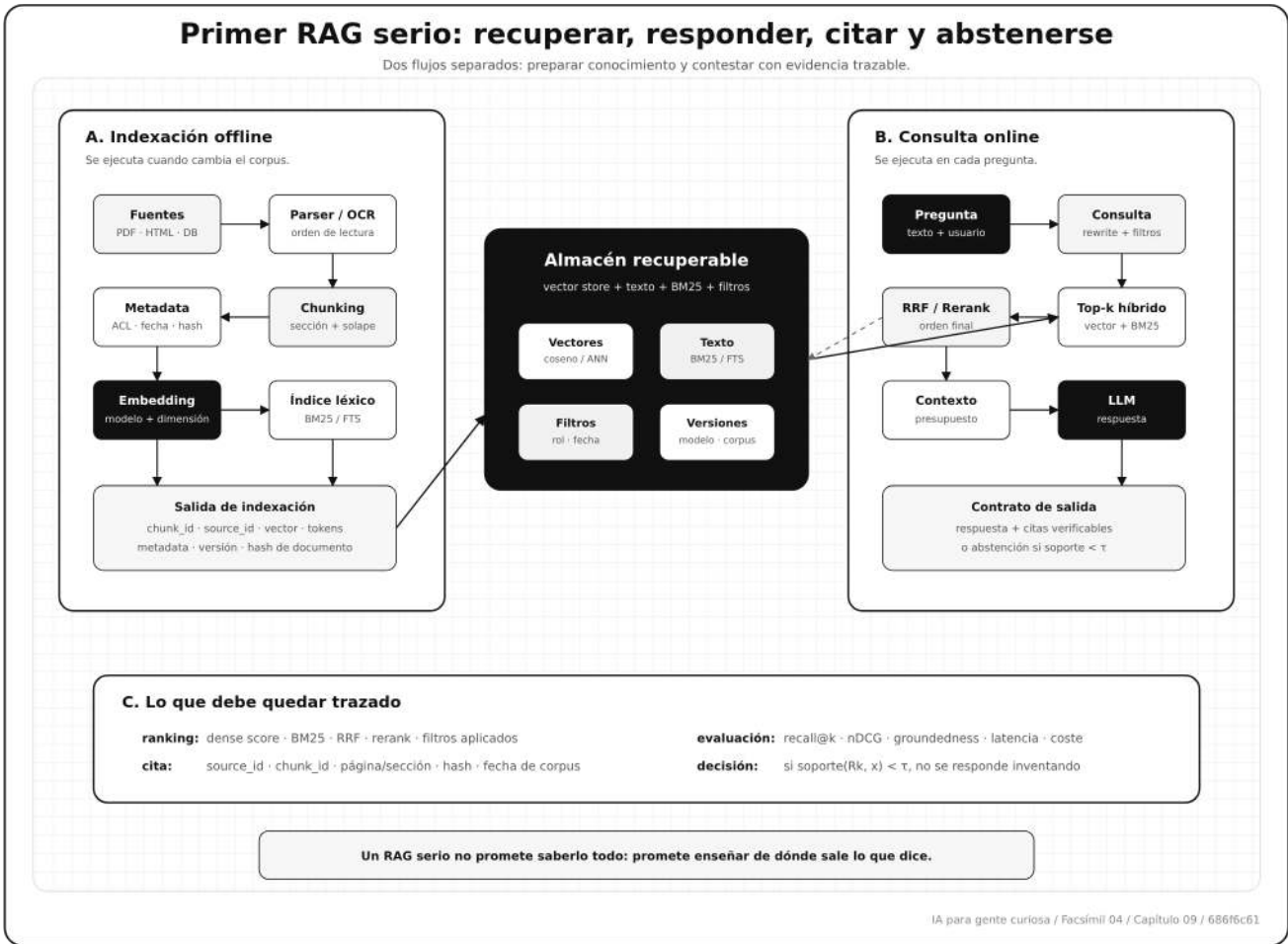
Antes de elegir una solución de terceros, conviene escribir una pequeña ADR técnica. No hace falta una novela; hace falta que nadie confunda “funciona en demo” con “lo podemos operar”.

PREGUNTA DE INGENIERÍA	POR QUÉ IMPORTA	SEÑAL DE BUENA RESPUESTA
¿Cómo actualiza y borra documentos?	RAG falla mucho cuando el índice conserva versiones antiguas.	Hay <code>upsert</code> , borrado por documento, reindexado y versión de corpus.
¿Dónde aplico permisos?	El texto no autorizado no debe entrar al contexto.	Filtros por usuario, grupo, tenant, vigencia y clasificación antes del top-k.
¿Puedo combinar vector, BM25 y filtros?	Muchas preguntas reales mezclan significado, palabras exactas y metadata.	Retrieval híbrido con scores visibles y filtros que no rompen latencia.
¿Qué devuelve como cita?	Sin <code>source_id</code> , página, sección o hash, revisar una respuesta es difícil.	Cada fragmento trae identificador estable, título, fecha y localización.
¿Puedo ver trazas?	Si no ves chunks, scores y prompt, no puedes depurar.	Logs por query, ranking, contexto final, coste y respuesta.
¿Puedo cambiar de modelo?	El embedding de hoy puede no ser el de mañana.	Índices versionados por modelo y dimensión; migración repetible.
¿Qué pasa con tablas e imágenes?	Mucha documentación útil no es texto plano.	Parsing verificable, OCR cuando toca y preservación de estructura.
¿Cómo se evalúa?	Una demo bonita no mide recall ni groundedness.	Set de preguntas, respuestas esperadas, citas esperadas y regresión automática.
¿Cómo salgo de la herramienta?	El bloqueo aparece cuando tus datos solo viven en su formato.	Exportación de chunks, metadata, vectores y trazas.

Una ruta razonable para empezar es esta:

CONTEXTO DEL EQUIPO	ruta inicial	CUÁNDO CAMBIAR
Quieres validar una idea en días.	File Search o RAG gestionado equivalente.	Cuando necesites permisos complejos, índices propios o trazas más finas.
Tienes app Python/TypeScript y datos variados.	LangChain, LlamaIndex o Haystack con Qdrant, pgvector, Pinecone o Weaviate.	Cuando el framework esconda demasiado o el flujo ya sea estable.
Ya estás en Google Cloud o Azure.	Vertex AI RAG Engine o Azure AI Search.	Cuando residencia, permisos, facturación y operación sean prioridad.
Necesitas control total y bajo coste.	pgvector/Qdrant autogestionado, pipeline propio y evaluación propia.	Cuando el volumen o el equipo pidan servicio gestionado.

Arquitectura mínima de un primer RAG



El diagrama separa lo que suele mezclarse. Indexar es preparar el material. Consultar es decidir qué evidencia entra. Responder es redactar, citar y abstenerse si el material no basta.

Cómo montar un primer RAG de verdad

Un primer RAG serio no empieza por el modelo. Empieza por el contrato de respuesta.

PASO	DECISIÓN	SALIDA VERIFICABLE
1. Elegir corpus	Qué documentos entran y cuáles no.	Lista de fuentes, versiones y propietarios.
2. Extraer texto	Cómo leer PDF, HTML, Markdown o base de datos.	Texto limpio con orden de lectura revisado.
3. Partir en chunks	Unidad recuperable y citable.	Chunks con <code>source_id</code> , sección, fecha y hash.
4. Indexar	Embeddings, BM25, filtros y vector store.	Índice versionado y reproducible.
5. Recuperar	Top-k, filtros y búsqueda híbrida.	Lista de chunks con scores y metadata.
6. Construir contexto	Qué entra al prompt y con qué formato.	Contexto numerado con fuentes.
7. Generar	Instrucciones para responder con evidencia.	Respuesta con citas o abstención.
8. Registrar	Guardar query, chunks, respuesta y versión.	Traza para depurar y evaluar.

Un prompt mínimo de RAG debe separar instrucciones y contexto. El modelo no debe tratar los documentos como órdenes, sino como material de consulta:

```
Responde usando solo el contexto incluido.
Si el contexto no contiene evidencia suficiente, responde:
"No tengo evidencia suficiente en las fuentes disponibles."

Pregunta:
{pregunta}

Contexto:
[F1] {fragmento_1}
[F2] {fragmento_2}

Formato:
- Respuesta breve.
- Citas entre corchetes, por ejemplo [F1].
- Si hay duda, explica qué dato falta.
```

La cita no es decoración. Es una interfaz de confianza: permite revisar si la frase que el modelo escribió está realmente en las fuentes.

Citas y abstención

Responder o abstenerse es una regla de umbral: la versión para RAG de la **opción de rechazo** clásica en clasificación, donde el sistema responde solo cuando su confianza supera un umbral y se abstiene si no.¹⁷

$$\text{responder}(x) = \begin{cases} g_\phi(x, R_k), & \text{si } \text{soporte}(R_k, x) \geq \tau \\ \text{abstenerse}, & \text{si } \text{soporte}(R_k, x) < \tau \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Pregunta.	"¿Puedo ampliar matrícula con pagos pendientes?".
R_k	Fragmentos recuperados.	Seis chunks tras filtros.
$\text{soporte}(R_k, x)$	Evidencia disponible para responder.	0,82 si hay fragmentos claros.
τ	Umbral mínimo para responder.	0,65 en una primera prueba.
g_ϕ	Modelo que redacta.	LLM con prompt de citas.

Ese soporte puede empezar siendo una regla: top score mínimo, al menos una fuente vigente y presencia de términos críticos. Más adelante puede ser un evaluador aprendido o un evaluador con rúbrica. Lo importante es que la abstención no sea una vergüenza; es una conducta correcta cuando falta evidencia.

SITUACIÓN	QUÉ DEBE HACER EL RAG	POR QUÉ
Hay una fuente vigente y clara.	Responder y citar.	La evidencia sostiene la respuesta.
Hay fuentes parecidas pero de años distintos.	Responder solo si el filtro de vigencia lo resuelve.	El parecido semántico no basta.
Hay dos fuentes que se contradicen.	Mostrar conflicto o abstenerse.	Elegir una en silencio rompe confianza.
No aparece evidencia directa.	Abstenerse y decir qué falta.	Mejor que rellenar huecos con probabilidad.
La pregunta pide acción externa.	Recuperar contexto, pero delegar la acción a una tool.	RAG informa; no ejecuta trámites por sí mismo.

Cómo optimizar bien un RAG

Optimizar RAG no significa tocar un parámetro al azar hasta que una demo parezca mejor. Significa separar dónde falla el sistema y medir cada etapa. Si una respuesta sale mal, la causa puede estar en el corpus, el parser, el chunking, el embedding, el índice, el reranker, el contexto, el prompt o la ausencia de una regla de abstención.

La primera regla es crear un pequeño conjunto de evaluación antes de optimizar. No hace falta empezar con mil preguntas. Puedes empezar con 30 o 50 preguntas reales, cada una con la fuente esperada y una explicación de qué debería responder el sistema. Sin ese conjunto, cada cambio se evalúa con intuición y la intuición se cansa rápido.

CAPA	QUÉ OPTIMIZAR	CÓMO MEDIRLO
Corpus	Fuentes vigentes, sin duplicados y con propietario.	Porcentaje de documentos con fecha, versión, dueño y estado.
Parsing	Texto correcto, tablas preservadas, orden de lectura.	Revisión manual de muestras y errores por tipo de documento.
Chunking	Unidad completa, citable y no demasiado ruidosa.	Recall@k por tamaño de chunk y tasa de chunks duplicados.
Metadata	Filtros útiles y citas revisables.	Porcentaje de chunks con <code>source_id</code> , página, sección, fecha y ACL.
Embeddings	Modelo adecuado a idioma, dominio y coste.	Recall@k, latencia, coste por millón de chunks y memoria.
Retrieval híbrido	Combinar significado, palabras exactas y filtros.	Comparar vector solo, BM25 solo, híbrido y RRF.
Reranking	Subir la evidencia buena antes del prompt.	nDCG@k, MRR y coste añadido por consulta.
Context builder	Meter evidencia suficiente sin ruido.	Precisión de citas, tokens de contexto y duplicados.
Prompt	Responder con contrato, citas y abstención.	Groundedness, formato válido y tasa de abstención correcta.
Operación	Actualizar, borrar, trazar y controlar coste.	Tiempo de reindexado, errores, latencia p95 y coste por respuesta.

Una receta práctica para optimizar sería:

1. Congela un corpus pequeño y versionado.
2. Escribe preguntas de evaluación con sus fuentes esperadas.
3. Mide retrieval antes de mirar la respuesta del LLM.
4. Ajusta chunking y metadata hasta que la evidencia aparezca arriba.
5. Añade BM25 o búsqueda híbrida si pierdes términos exactos.
6. Añade reranker si recuperas bien pero ordenas mal.
7. Recorta contexto y deduplica antes de tocar el prompt.
8. Obliga a citar y abstenerse con un contrato de salida.
9. Guarda trazas de cada consulta para comparar versiones.
10. Cambia una cosa cada vez; si cambias cinco, no sabes qué funcionó.

Hay una trampa frecuente: intentar arreglar con prompt lo que es un fallo de retrieval. Si el chunk correcto no entra en contexto, el modelo no puede citarlo. Otra trampa es subir `top-k` sin control. Traer más chunks puede mejorar recall, pero también mete ruido, sube coste y aumenta la probabilidad de mezclar fuentes.

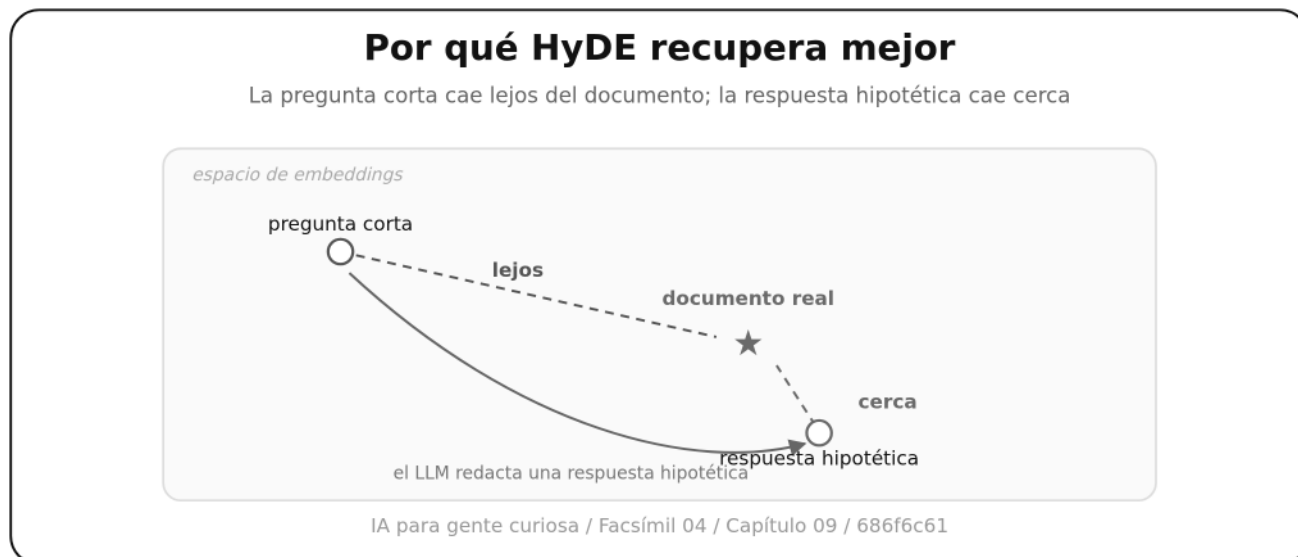
SÍNTOMA	DIAGNÓSTICO PROBABLE	PRIMER AJUSTE
La respuesta suena bien pero no cita la fuente correcta.	Retrieval o context builder fallan.	Revisar top-k, filtros, metadata y orden de contexto.
Recupera documentos antiguos.	Falta filtro de vigencia o borrado.	Añadir <code>vigente=true</code> , versión de corpus y política de retirada.
Pierde códigos, nombres propios o fechas.	Vector solo no basta.	Añadir BM25/FTS y fusión RRF.
Recupera muchos chunks parecidos.	Solape excesivo o duplicados.	Deduplicar por hash, sección o similitud entre chunks.
Responde cuando no sabe.	Falta abstención o umbral.	Definir soporte mínimo y respuesta de insuficiencia.
Es lento.	Índice, reranker o contexto demasiado grandes.	Medir p95, reducir candidatos, cachear embeddings y ajustar ANN.

Mejorar la consulta antes de buscar

Un fallo silencioso de retrieval es que la pregunta y el documento correcto no comparten palabras ni forma. «¿Me devuelven el dinero si me borro?» y «Reembolso por baja de matrícula: artículo 14» hablan de lo mismo con vocabularios distintos. Tres técnicas atacan esto **transformando la consulta** antes de recuperar:

TÉCNICA	QUÉ HACE	CUÁNDO AYUDA
Multi-query	El LLM genera varias reformulaciones de la pregunta, se recupera con todas y se unen los resultados.	Preguntas ambiguas o con varias formas de expresarse.
HyDE	El LLM redacta una respuesta hipotética a la pregunta y se busca con el embedding de <i>esa respuesta</i> , que se parece más al documento real que la pregunta corta.	Cuando la pregunta es breve y el corpus es denso.
Descomposición	Una pregunta compuesta se parte en sub-preguntas que se recuperan por separado.	«¿Puedo ampliar matrícula y aplazar el pago a la vez?».

HyDE (Hypothetical Document Embeddings) es la más citada: genera un documento hipotético y busca con su vector, partiendo de que una respuesta redactada cae más cerca del documento real en el espacio que la pregunta.¹⁸ El coste es una llamada extra al LLM por consulta; conviene medir si el recall que ganas compensa esa latencia.

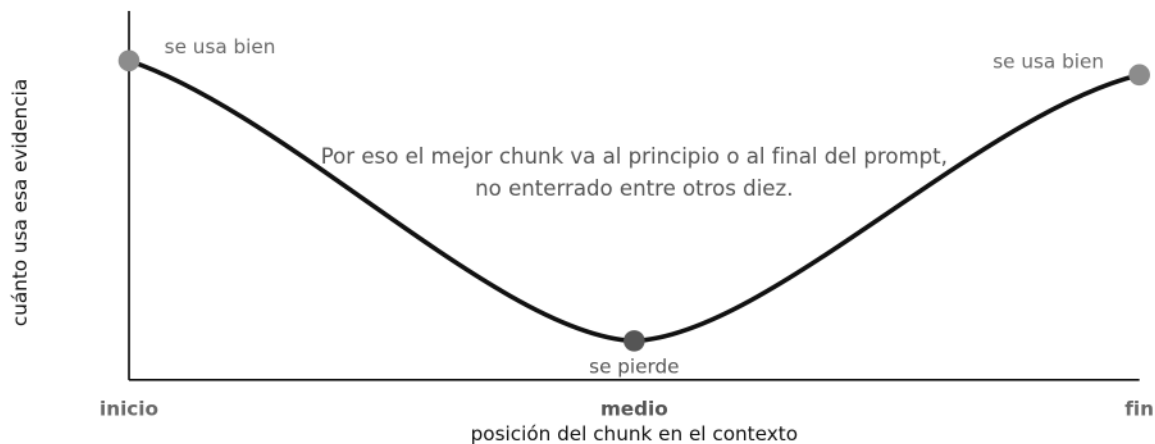


El orden importa: *lost in the middle*

Una vez recuperados y reordenados los chunks, falta una decisión que se olvida: **en qué orden van dentro del prompt**. No es indiferente. Los modelos de lenguaje usan mejor la evidencia que aparece al **principio** y al **final** del contexto, y peor la que cae en la **mitad**, un efecto medido y bautizado como *lost in the middle*.¹⁹ La consecuencia práctica es directa: coloca el chunk más relevante al principio o al final del contexto, no enterrado en medio de otros diez; y resiste la tentación de subir `top-k` «por si acaso», porque meter más evidencia mediocre empuja la buena hacia la zona donde el modelo la aprovecha peor.

Lost in the middle

El modelo aprovecha mejor la evidencia del principio y del final que la de la mitad



IA para gente curiosa / Facsímil 04 / Capítulo 09 / 686f6c61

El pipeline de un RAG, etapa a etapa

Recuperar barato, reordenar caro, ensamblar con criterio, generar con contrato



Cada flecha es un punto de medida: mide retrieval antes de mirar la respuesta del LLM.

IA para gente curiosa / Facsímil 04 / Capítulo 09 / 686f6c61

¿Solo texto? Qué más puede entrar en un RAG

Texto es el punto de partida porque el LLM consume tokens y porque muchos documentos terminan convertidos a texto. Pero RAG no tiene por qué limitarse a texto plano. Lo importante es convertir cada fuente en una representación recuperable, filtrable y citable.

TIPO DE INFORMACIÓN	CÓMO ENTRA AL RAG	QUÉ HAY QUE CUIDAR
PDF y documentos	Texto extraído, páginas, títulos y chunks.	Orden de lectura, notas, tablas, encabezados y versión.
Tablas	Filas, columnas, celdas relevantes o resumen estructurado.	No perder unidades, claves, fechas ni relación fila-columna.
Bases de datos	Resultados de SQL o vistas preparadas.	Permisos, frescura, consultas reproducibles y explicación del resultado.
Imágenes	OCR, descripciones, embeddings multimodales o regiones anotadas.	Distinguir texto visible, objetos, gráficos y metadatos.
Audio	Transcripción, marcas de tiempo y hablantes.	Errores de transcripción, idioma, ruido y citas por minuto/segundo.
Vídeo	Transcripción, fotogramas clave, escenas y marcas temporales.	Recuperar el momento exacto, no solo un resumen genérico.
Código	Funciones, clases, rutas, tests y documentación cercana.	Mantener dependencias, imports, versión y lenguaje.
Logs y tickets	Eventos normalizados, campos, tiempos y etiquetas.	Ruido, duplicados, retención y datos sensibles.
Grafos u ontologías	Nodos, relaciones, triples y consultas de grafo.	No convertir relaciones precisas en texto ambiguo.
Resultados de tools	Salidas de APIs, cálculos o búsquedas vivas.	Separar evidencia recuperada de acciones ejecutadas.

Cuando una fuente no es texto, tienes dos estrategias. La primera es traducirla a texto fiel: OCR, transcripción, descripción de imagen, explicación de tabla. La segunda es usar embeddings específicos: embeddings de imagen, multimodales, de audio o de código. En sistemas reales se mezclan ambas: una imagen puede tener OCR, una descripción y un vector multimodal; una tabla puede tener filas indexadas y, además, una consulta SQL cuando hace falta precisión.

La pregunta de ingeniería no es “¿puedo meterlo en RAG?”, sino “¿puedo recuperar la parte correcta, respetar permisos, citarla y comprobarla?”. Si no puedes citar una celda, una página, una región de imagen, un minuto de audio o una fila de base de datos, todavía no tienes una evidencia robusta.

Ruta rápida con una solución gestionada

Si quieres montar algo rápido sobre archivos, una opción es usar File Search con vector stores gestionados. La idea es: crear vector store, subir archivos y dejar que la herramienta de búsqueda recupere contexto para la llamada al modelo.²⁰

```
from openai import OpenAI

client = OpenAI()

store = client.vector_stores.create(name="normativa-universidad")

client.vector_stores.files.upload_and_poll(
    vector_store_id=store.id,
    file=open("normativa_matricula_2026.pdf", "rb"),
)

respuesta = client.responses.create(
    model="gpt-4.1-mini",
    input=(
        "¿Puedo ampliar matrícula en septiembre "
        "si tengo pagos pendientes? Cita las fuentes."
    ),
    tools=[
        {
            "type": "file_search",
            "vector_store_ids": [store.id],
            "max_num_results": 6,
        }
    ],
)

print(respuesta.output_text)
```

Esto sirve para prototipar o para casos donde te compensa delegar infraestructura. Pero incluso aquí quedan decisiones tuyas: qué archivos subes, cómo versionas, cómo retiras documentos antiguos, qué permisos aplicas, cómo revisas citas y cómo evalúas si la respuesta es correcta.

En el día a día

En una universidad, un RAG es lo que permite que un asistente responda «sí puedes ampliar matrícula en septiembre, según el artículo 12 de la normativa vigente» en vez de inventarse una fecha. En soporte interno, recupera el procedimiento correcto antes de redactar la respuesta, con la versión y el permiso adecuados. En un producto, es la diferencia entre un chatbot que suena bien y uno que cita la fuente y se calla cuando no la tiene.

La parte delicada es que un RAG bueno no se nota por el modelo generativo, sino por todo lo de alrededor: cómo troceas, qué metadata guardas, cómo filtras por permisos y vigencia, cuántos fragmentos pasas al prompt, cómo construyes las citas y cuándo decides abstenerte. Un RAG es, sobre todo, una cadena de decisiones de ingeniería sobre la evidencia.

Por qué debería importarte

Porque es el patrón con el que se construye la mayoría de lo que se vende como «IA con tus datos», y es donde más fácil es engañarse. Un modelo excelente con un retrieval malo produce respuestas seguras y falsas, y el fallo parece del modelo cuando casi siempre está en la evidencia: el chunk que cortó una excepción, el filtro de vigencia que faltó, el top-k que metió ruido, la cita que el modelo inventó.

Si entiendes la cadena, sabes diagnosticar dónde falla y exigir lo que distingue un RAG serio de una demo: contrato de evidencia, citas verificables, abstención cuando falta soporte y trazas por etapa. Esa es la diferencia entre responder bonito y responder de forma que alguien pueda comprobar.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Llamar RAG a cualquier chat con documentos	Puede no haber filtros, citas, trazas ni abstención.	Exigir contrato de evidencia desde el principio.
Trocear por tamaño sin mirar estructura	Cortas definiciones, excepciones o pasos completos.	Partir por secciones, títulos y unidades citables.
Meter demasiado contexto	El modelo recibe ruido y puede mezclar fuentes.	Medir top-k, deduplicar y respetar presupuesto.
Confiar en la cita generada sin validarla	El modelo puede citar un fragmento que no sostiene la frase.	Construir citas desde ids recuperados, no desde memoria del modelo.
No abstenerse nunca	El sistema responde incluso cuando no hay evidencia.	Definir umbrales y respuesta estándar de insuficiencia.
Olvidar permisos en retrieval	El texto ya entró al contexto aunque luego no lo muestres.	Aplicar filtros antes de recuperar y antes de construir contexto.
Evaluar solo la respuesta final	No sabes si falló retrieval, chunking o generación.	Guardar trazas por etapa; el capítulo 10 entra ahí.

Cómo encaja todo

Vocabulario aprendido

El vocabulario de este capítulo no conviene memorizarlo como lista. Conviene leerlo como

piezas de una máquina: cada termino responde a una pregunta concreta.

TÉRMINO	RESPONDE A	DEFINICIÓN ÚTIL
Corpus	¿Qué puede consultar el sistema?	Conjunto de fuentes aceptadas para el RAG, con permisos, versión y propietario.
Fuente	¿De dónde salió la evidencia?	Documento, web, tabla o base de datos original.
Parser	¿Cómo convierto la fuente en texto usable?	Pieza que extrae texto, títulos, tablas y orden de lectura.
OCR	¿Qué hago si el documento es una imagen?	Técnica que convierte escaneos o imágenes en texto recuperable.
Chunk	¿Cuál es la unidad mínima recuperable?	Fragmento que puede buscarse, meterse en contexto y citarse.
Metadata	¿Cómo filtro y explico un chunk?	Datos como curso, vigencia, sección, página, rol, hash y fecha.
ACL	¿Quién puede recuperar cada fragmento?	Regla de acceso aplicada antes de construir contexto.
Hash	¿Cómo sé si cambió una fuente?	Huella calculada del texto o archivo para detectar cambios.
Embedding	¿Cómo comparo significado aproximado?	Vector numérico que representa una pregunta o fragmento.
Vector store	¿Dónde guardo vectores y metadata?	Almacén preparado para buscar por similitud y filtros.
FTS	¿Cómo busco palabras exactas?	Búsqueda textual clásica sobre términos, frases y campos.
BM25	¿Cómo ordeno resultados por relevancia léxica?	Ranking que combina frecuencia de términos y rareza informativa.
ANN	¿Cómo busco vectores a escala?	Búsqueda aproximada de vecinos cercanos para no comparar contra todo.
Top-k	¿Cuántos candidatos saco?	Número de resultados que pasan a rerank, contexto o evaluación.
RRF	¿Cómo mezclo rankings distintos?	Fusión por posiciones; útil para combinar BM25 y embeddings.
Reranker	¿Cómo ordeno mejor candidatos ya encontrados?	Modelo o regla más lenta que reevalúa resultados prometedores.
Context builder	¿Qué lee finalmente el modelo?	Pieza que selecciona, ordena, recorta y etiqueta evidencia.

TÉRMINO	RESPONDE A	DEFINICIÓN ÚTIL
Cita	¿Qué fuente sostiene esta frase?	Referencia trazable a chunk, documento, página, sección y versión.
Grounding	¿La respuesta está apoyada en evidencia?	Grado en que cada afirmación importante sale de los chunks recuperados.
Abstención	¿Qué hago si no hay evidencia suficiente?	Responder que falta soporte en vez de inventar.
Traza	¿Cómo depuro una respuesta?	Registro de pregunta, filtros, rankings, contexto, modelo, coste y salida.
File Search	¿Qué delego si uso una solución gestionada?	Herramienta alojada para subir archivos, buscar en vector store y pasar contexto al modelo.

Antes de pasar página

- ☐ ¿Puedo explicar la diferencia entre base vectorial y RAG? (Si no, vuelve a «Qué no es RAG».)
- ☐ ¿Sé distinguir RAG, memoria conversacional, caché, tool y entrenamiento? (Si no, vuelve a «RAG, memoria y entrenamiento no son lo mismo».)
- ☐ ¿Sé justificar por qué usar RAG en vez de fine-tuning? (Si no, vuelve a «Por qué usar un RAG».)
- ☐ ¿Sé separar indexación y consulta? (Si no, vuelve a «Qué sí es: una cadena verificable».)
- ☐ ¿Sé qué metadata mínima debe llevar un chunk citable? (Si no, vuelve a «Elementos importantes de un RAG».)
- ☐ ¿Sé explicar corpus, parser, OCR, ACL, hash, top-k y reranker sin esconderme en siglas? (Si no, vuelve a «Términos que no podemos dar por sabidos».)
- ☐ ¿Sé explicar qué significa la dimensión de un embedding y cómo afecta a memoria, coste e índice? (Si no, vuelve a «Dimensiones en un RAG».)
- ☐ ¿Puedo calcular cómo cambia el número de chunks con tamaño y solape? (Si no, vuelve a «Chunking: partir sin romper el significado».)
- ☐ ¿Sé optimizar retrieval antes de tocar el prompt? (Si no, vuelve a «Cómo optimizar bien un RAG».)
- ☐ ¿Sé qué añadiría si el RAG debe trabajar con tablas, imágenes, audio, vídeo, código o bases de datos? (Si no, vuelve a «¿Solo texto? Qué más puede entrar en un RAG».)

- ☐ ¿Sé cuándo usar una solución gestionada y cuándo montar componentes? (Si no, vuelve a «Soluciones de terceros: qué comprar, qué montar y qué no delegar».)
- ☐ ¿Sé construir un prompt que separe instrucciones, pregunta y contexto? (Si no, vuelve a «Arquitectura mínima de un primer RAG».)
- ☐ ¿Sé por qué la abstención es una salida correcta? (Si no, vuelve a «Citas y abstención».)
- ☐ ¿Puedo explicar qué parte cambiaría para pasar del ejemplo local a un LLM real? (Si no, vuelve a «Cómo montar un primer RAG de verdad».)
- ☐ ¿Sé qué trazas guardar para evaluar el sistema en el capítulo 10?

En resumen

IDEA FUERZA	DETALLE
RAG une recuperación y generación.	El modelo responde con contexto externo recuperado en tiempo de consulta.
RAG no es memoria ni entrenamiento.	No cambia pesos; recupera evidencia externa y actualizable.
Usamos RAG cuando necesitamos fuentes.	Información cambiante, privada, citable o filtrada por permisos.
Las dimensiones importan.	Afectan representación, almacenamiento, latencia, coste y reindexado.
El chunk es la unidad de confianza.	Si no puedes citarlo, no deberías usarlo como evidencia.
Las soluciones de terceros aceleran, no deciden.	Delegan infraestructura o composición, pero no sustituyen evaluación y permisos.
Optimizar exige medir por capas.	Corpus, parsing, chunking, retrieval, rerank, contexto y generación fallan de formas distintas.
RAG no es solo texto plano.	Puede incorporar tablas, imágenes, audio, vídeo, código, grafos, bases de datos y tools si son recuperables y citables.
Citar exige diseño.	La cita debe apuntar a una fuente concreta y vigente.
Abstenerse es parte del producto.	Cuando falta evidencia, responder menos es responder mejor.
El primer RAG debe dejar trazas.	Query, filtros, chunks, scores, prompt y respuesta permiten depurar.

Para saber más

Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). *Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods*. SIGIR, 758-759.

<https://doi.org/10.1145/1571941.1572114>

deepset. (2026). *Pipelines*. <https://docs.haystack.deepset.ai/docs/pipelines>

Google Cloud. (2026). *Vertex AI RAG Engine overview*. <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/rag-overview>

LangChain. (2026). *Retrieval*. <https://docs.langchain.com/oss/python/langchain/retrieval>

Lewis, P. y otros (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. NeurIPS 33, 9459-9474.

<https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>

LlamaIndex. (2026). *Introduction to RAG*.

<https://docs.llamaindex.ai/en/stable/understanding/rag/>

OpenAI. (2026). *File search*. <https://platform.openai.com/docs/guides/tools-file-search/>

Qdrant. (2026). *Indexing*. <https://qdrant.tech/documentation/manage-data/indexing/>

Robertson, S. y Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond*. *Foundations and Trends in Information Retrieval*, 3(4), 333-389.

<https://doi.org/10.1561/15000000019>

Notas

1. Lewis, P. y otros (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. NeurIPS 33, 9459-9474.
<https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>.

↩

2. Lewis y otros, 2020. ↩

3. OpenAI. (2026). *File search*. <https://platform.openai.com/docs/guides/tools-file-search/>. Consultado el 25 de mayo de 2026. ↩

4. LangChain. (2026). *Retrieval*. <https://docs.langchain.com/oss/python/langchain/retrieval>. Consultado el 25 de mayo de 2026. ↩

5. LlamaIndex. (2026). *Introduction to RAG*.

<https://docs.llamaindex.ai/en/stable/understanding/rag/>. Consultado el 25 de mayo de 2026. ↩

6. deepset. (2026). *Pipelines*. <https://docs.haystack.deepset.ai/docs/pipelines>. Consultado el 25 de mayo de 2026. ↵
7. Google Cloud. (2026). *Vertex AI RAG Engine overview*. <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/rag-overview>. Consultado el 25 de mayo de 2026. ↵
8. Microsoft. (2026). *Retrieval-augmented generation in Azure AI Search*. <https://learn.microsoft.com/en-us/azure/search/retrieval-augmented-generation-overview>. Consultado el 25 de mayo de 2026. ↵
9. Pinecone. (2026). *Build a RAG chatbot*. <https://docs.pinecone.io/guides/get-started/build-a-rag-chatbot>. Consultado el 25 de mayo de 2026. Weaviate. (2026). *Retrieval Augmented Generation (RAG)*. <https://docs.weaviate.io/weaviate/search/generative>. Consultado el 25 de mayo de 2026. ↵
10. Robertson, S. y Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond*. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. <https://doi.org/10.1561/15000000019>. Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). *Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods*. *SIGIR*, 758-759. <https://doi.org/10.1145/1571941.1572114>. ↵
1. Patrick Lewis y otros (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. *NeurIPS*. <https://arxiv.org/abs/2005.11401>. ↵
2. Gerard Salton, A. Wong y C. S. Yang. (1975). *A Vector Space Model for Automatic Indexing*. *Communications of the ACM*, 18(11), 613-620. <https://doi.org/10.1145/361219.361220>. ↵
3. OpenAI, 2026. ↵
4. LangChain, 2026. ↵
5. LlamaIndex, 2026. ↵
6. deepset, 2026. ↵
7. C. K. Chow. (1970). *On Optimum Recognition Error and Reject Tradeoff*. *IEEE Transactions on Information Theory*, 16(1), 41-46. <https://doi.org/10.1109/TIT.1970.1054406>. ↵
8. Luyu Gao y otros (2022). *Precise Zero-Shot Dense Retrieval without Relevance Labels*. <https://arxiv.org/abs/2212.10496>. ↵
9. Nelson F. Liu y otros (2023). *Lost in the Middle: How Language Models Use Long Contexts*. *TACL 2024*. <https://arxiv.org/abs/2307.03172>. ↵
10. OpenAI, 2026. ↵

Capítulo 10: Evaluar RAG: retrieval, groundedness y abstención

La demo no cuenta como evaluación

En el capítulo 09 montamos el primer RAG serio: corpus, chunks, embeddings, retrieval, contexto, citas y abstención. Ahora viene la parte que separa una demo de un sistema profesional: demostrar que mejora, saber dónde falla y decidir cuándo no debe responder.

Un RAG puede fallar de formas muy distintas. Puede no recuperar el documento correcto. Puede recuperarlo y no meterlo en el contexto final. Puede meterlo y redactar algo que no está sostenido. Puede citar una fuente que no dice eso. Puede contestar cuando debería abstenerse. Si solo miras la respuesta final, llegas tarde: ves el síntoma, pero no el órgano que falló.

Evaluar RAG es medir la cadena completa:

1. ¿La pregunta está en un conjunto de evaluación representativo?
2. ¿El corpus contiene la evidencia necesaria?
3. ¿El retrieval trae esa evidencia entre los primeros resultados?
4. ¿El context builder la mete en el prompt sin ahogarla en ruido?
5. ¿La respuesta está apoyada por el contexto?
6. ¿Las citas apuntan a fragmentos que sostienen lo dicho?
7. ¿El sistema se abstiene cuando no hay evidencia?
8. ¿El coste, la latencia y la operación siguen siendo aceptables?

La evaluación no es una fase final. Es una pieza del producto.

Estado del arte con fecha de corte

Fecha de corte: 25 de mayo de 2026.

Fuentes consultadas ese día: documentación de Ragas, TruLens, LangSmith, LlamaIndex, Phoenix y OpenAI Graders; y referencias académicas sobre RAG, retrieval y benchmarks como BEIR/MTEB.

RAGAS propuso evaluar RAG separando componentes como recuperación, relevancia y fidelidad de la respuesta.¹ La documentación actual de Ragas organiza métricas de RAG como context precision, context recall, response relevancy, faithfulness y métricas multimodales.²

TruLens populariza una tríada muy práctica: relevancia del contexto, groundedness y relevancia de la respuesta.³ LangSmith, LlamaIndex y Phoenix empujan una idea parecida desde producto: datasets, experimentos, trazas, evaluadores y comparación entre versiones.⁴

OpenAI Graders documenta la idea de evaluadores configurables con criterios, escalas y umbrales, incluyendo verificaciones de texto, similitud, evaluadores de modelo y ejecución de código.⁵ La lección importante para nuestro facsímil no es “usa esta herramienta”, sino “define la rúbrica y conserva la traza”.

FAMILIA	QUÉ MIDE	CUÁNDO USARLA
Métricas clásicas de retrieval	Si los chunks esperados aparecen y en qué posición.	Antes de mirar la respuesta del LLM.
Métricas de contexto	Si el contexto final es útil, completo y no ruidoso.	Cuando el retrieval trae candidatos pero la respuesta falla.
Métricas de groundedness	Si las afirmaciones están sostenidas por el contexto.	Para respuestas citadas, informes y asistentes de documentación.
Métricas de abstención	Si responde cuando debe y se calla cuando toca.	En dominios donde inventar cuesta confianza.
Métricas de operación	Latencia, coste, errores, cobertura y deriva.	Cuando el RAG ya vive en una aplicación.

Qué significa evaluar por capas

Un RAG se evalúa por capas porque cada capa tiene una pregunta distinta. Si solo mides “respuesta correcta”, no sabes si mejorar embeddings, chunking, prompt, reranker o corpus.

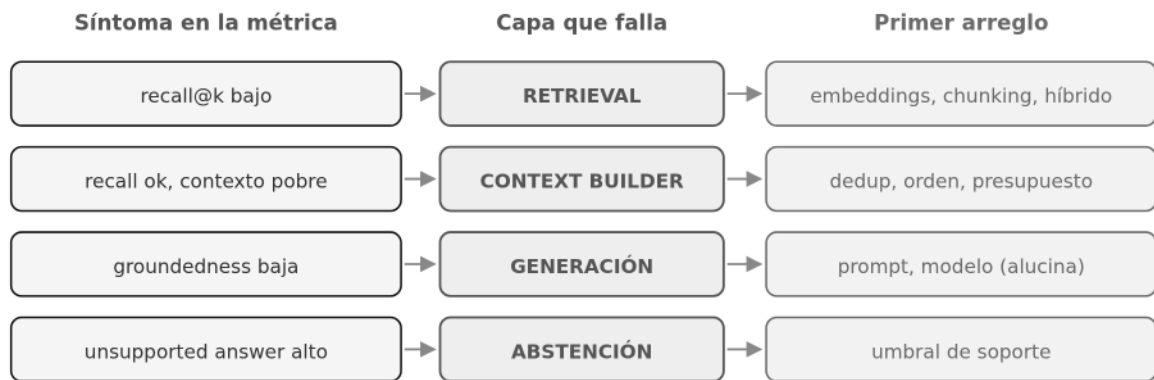
CAPA	PREGUNTA	EVIDENCIA QUE NECESITAS
Corpus	¿Existe la fuente correcta?	Documento vigente, versión, propietario y fecha.
Parsing	¿El texto extraído es fiel?	Comparación contra PDF, HTML, tabla o fuente original.
Chunking	¿La unidad recuperable sostiene una respuesta?	Chunks con título, sección, página y hash.
Retrieval	¿Aparece la evidencia en top-k?	Ranking de chunks y qrels.
Reranking	¿Sube la evidencia buena?	Ranking antes/después y relevancia graduada.
Context builder	¿El modelo recibe lo necesario y no demasiado ruido?	Contexto final enviado al LLM.
Generación	¿La respuesta contesta con contrato?	Respuesta, citas, formato y abstención.
Groundedness	¿Cada afirmación está sostenida?	Claims separados y evidencia por claim.
Operación	¿El sistema aguanta uso real?	Trazas, latencia, coste, errores y feedback.

La regla de ingeniería es sencilla: primero evalúa retrieval sin LLM; luego evalúa generación con contexto fijo; después evalúa el sistema completo. Si mezclas todo desde el principio, cada fallo parece misterioso.

Medir por capas tiene un beneficio inmediato: convierte un síntoma en un diagnóstico. Cada métrica que baja apunta a una capa concreta y a un primer arreglo, en vez de obligarte a adivinar.

De síntoma a capa a arreglo

Medir por capas convierte un fallo difuso en un diagnóstico concreto



Por eso se mide cada capa por separado: el número te dice dónde mirar.

IA para gente curiosa / Facsímil 04 / Capítulo 10 / 686f6c61

Dataset de evaluación: el corazón del sistema

Un dataset de evaluación no es un conjunto de preguntas bonitas. Es un contrato de verdad para comparar versiones. Debe contener preguntas que representen el uso real, preguntas que el sistema debe responder, preguntas que debe rechazar por falta de evidencia y preguntas donde los filtros importan.

Un ejemplo de fila mínima:

CAMPO	PARA QUÉ SIRVE
id	Identificador estable de la pregunta.
pregunta	Lo que una persona o sistema preguntaría.
answerable	Si el corpus contiene evidencia suficiente.
gold_chunks	Chunks esperados o aceptables.
gold_answer	Respuesta de referencia, si existe.
gold_citations	Citas esperadas.
filtros	Curso, rol, vigencia, idioma, cliente o tenant.
tipo	Single-hop, multi-hop, tabla, código, imagen, temporal, etc.
dificultad	Fácil, media, difícil, o escala propia.
criterio	Qué debe ocurrir para considerar la respuesta válida.

No todas las preguntas necesitan una respuesta literal de referencia. Para retrieval basta con qrels: juicios de relevancia entre pregunta y chunks. Para groundedness necesitas contexto y respuesta. Para abstención necesitas casos sin evidencia suficiente.

TIPO DE PREGUNTA	EJEMPLO	QUÉ PRUEBA
Directa	“¿Cuándo se abre la ampliación de matrícula?”	Retrieval simple y cita directa.
Con filtro	“¿Qué aplica al curso 2026?”	Metadata y vigencia.
Multi-hop	“¿Puedo ampliar si tengo pagos pendientes y cómo se solicita?”	Recuperar más de un fragmento.
Tabla	“¿Qué plazo corresponde a segunda matrícula?”	Parsing y estructura tabular.
No responsable	“¿Cuál es el horario de cafetería?” si no está en corpus.	Abstención.
Contradicción documental	Dos fuentes con fechas distintas.	Vigencia, prioridad y explicación.
Texto largo	Preguntas que requieren contexto distribuido.	Recall, deduplicación y presupuesto.

Ejemplos de datasets de evaluación que puedes construir en un proyecto real:

DATASET	QUÉ CONTIENE	POR QUÉ MERECE EXISTIR
FAQ real	Preguntas frecuentes, respuesta esperada y fuente exacta.	Mide si el RAG resuelve lo que más se pregunta.
Normativa vigente	Preguntas con <code>curso</code> , <code>fecha</code> , <code>rol</code> y chunks esperados.	Mide filtros y prioridad documental.
Casos sin evidencia	Preguntas plausibles cuya respuesta no está en el corpus.	Mide abstención correcta.
Multi-hop	Preguntas que exigen dos o más fuentes.	Mide si el contexto compone evidencia sin mezclar.
Tablas	Preguntas sobre filas, columnas, importes, plazos o unidades.	Mide parsing y recuperación estructurada.
Citas difíciles	Respuestas donde una cita parcial no basta.	Mide si la cita sostiene la afirmación completa.
Regresión	Casos que ya fallaron en producción y se corrigieron.	Evita reintroducir errores.
Segmentos críticos	Preguntas por idioma, área, producto, cliente o perfil.	Evita que la media esconda fallos locales.

Si vienes del facsimilar 3, capítulo 06, la diferencia es esta: datasets como FLAN, Dolly-15K, HH-RLHF o LAION-5B pueden entrenar o adaptar modelos; este dataset de evaluación no debería entrenar el sistema que estás midiendo. Su trabajo es ponerle un espejo fiable.

El dataset debe crecer desde producción. Las primeras 30 preguntas sirven para empezar. Las siguientes 300 salen de dudas reales, tickets, búsquedas sin respuesta, feedback de usuarios y revisiones de profesorado o equipo de dominio.

Generar el dataset: a mano y sintético

Construir 300 preguntas a mano es lento, así que existe un atajo: **generación sintética**. Un LLM lee cada chunk del corpus y propone preguntas que ese chunk podría responder, junto con el chunk de oro asociado. Es lo que automatiza RAGAS con su generación de conjuntos de prueba, capaz de producir preguntas de distinta dificultad y tipo (directas, multi-hop, de razonamiento) a partir de tus propios documentos.⁶

Dos cautelas de ingeniería, porque lo sintético tiene trampa:

- **Revísalo a mano antes de usarlo.** Una pregunta sintética puede ser trivial («¿qué dice el artículo 14?» copiando el texto), ambigua o estar mal emparejada con su oro. Un dataset sin revisar mide la capacidad del generador, no la de tu RAG.

- **Lo sintético no sustituye lo real.** Las preguntas que de verdad importan (las raras, las mal formuladas, las que nadie anticipó) salen de producción, no de un generador. Usa lo sintético para arrancar y para cubrir tipos; usa lo real para decidir.

Y un caso que el generador casi nunca produce solo y tienes que añadir tú: las **preguntas no respondibles**, sin evidencia en el corpus, que son las que prueban la abstención. Sin ellas, tu dataset premia a un sistema que lo contesta todo.

Qrels y relevancia graduada

Un qrel es un juicio de relevancia. Dice que una pregunta necesita tal documento o tal chunk. Puede ser binario o graduado.

RELEVANCIA	SIGNIFICADO
0	No ayuda a responder.
1	Relacionado, pero insuficiente.
2	Útil para parte de la respuesta.
3	Evidencia central.

Ejemplo:

PREGUNTA	CHUNK	RELEVANCIA
q1 ampliación con pagos pendientes	norm-2026#c1	3
q1 ampliación con pagos pendientes	faq-campus#c1	0
q2 acceso al campus	faq-campus#c1	3
q3 horario de cafetería	ningún chunk	no respondible

Esta tabla permite evaluar retrieval sin llamar al LLM. Eso ahorra coste y te dice si la base del sistema funciona.

Métricas de retrieval

Las métricas de retrieval responden a una pregunta: ¿la evidencia correcta aparece en el ranking? No hay que inventarlas: son las métricas estándar de la recuperación de información, con décadas de uso en las campañas TREC, y `nDCG` añade el descuento por posición para relevancia graduada.⁷

Sea G_q el conjunto de chunks relevantes para una pregunta q , y sea $R_k(q)$ la lista de los k primeros chunks recuperados.

$$\text{Precision@k}(q) = \frac{|R_k(q) \cap G_q|}{k}$$

$$\text{Recall@k}(q) = \frac{|R_k(q) \cap G_q|}{|G_q|}$$

$$\text{Hit@k}(q) = \begin{cases} 1, & \text{si } R_k(q) \cap G_q \neq \emptyset \\ 0, & \text{si } R_k(q) \cap G_q = \emptyset \end{cases}$$

MÉTRICA	QUÉ TE DICE	CAUTION
Precision@k	De lo que recuperas, cuánto sirve.	Puede ser baja si necesitas traer contexto amplio.
Recall@k	De lo que necesitabas, cuánto aparece.	Puede subir trayendo demasiado ruido.
Hit@k	Si aparece al menos una evidencia.	No mide si aparece toda la evidencia.
MRR	Qué tan pronto aparece la primera evidencia.	No basta para preguntas multi-hop.
nDCG@k	Si lo más relevante aparece arriba.	Requiere relevancia graduada.

MRR se calcula con la posición de la primera evidencia relevante:

$$\text{RR}(q) = \frac{1}{\text{rank}_q}$$

Si el primer chunk relevante aparece en posición 1, RR vale 1. Si aparece en posición 5, vale 0,2. Si no aparece, vale 0. El MRR es la media de RR en muchas preguntas.

nDCG usa relevancia graduada:

$$\text{DCG@k} = \sum_{i=1}^k \frac{2^{rel_i} - 1}{\log_2(i + 1)}$$

$$\text{nDCG@k} = \frac{\text{DCG@k}}{\text{IDCG@k}}$$

SÍMBOLO	SIGNIFICADO
rel_i	Relevancia del resultado en posición i .
DCG	Ganancia descontada por posición.
$IDCG$	DCG ideal si los mejores resultados estuvieran arriba.

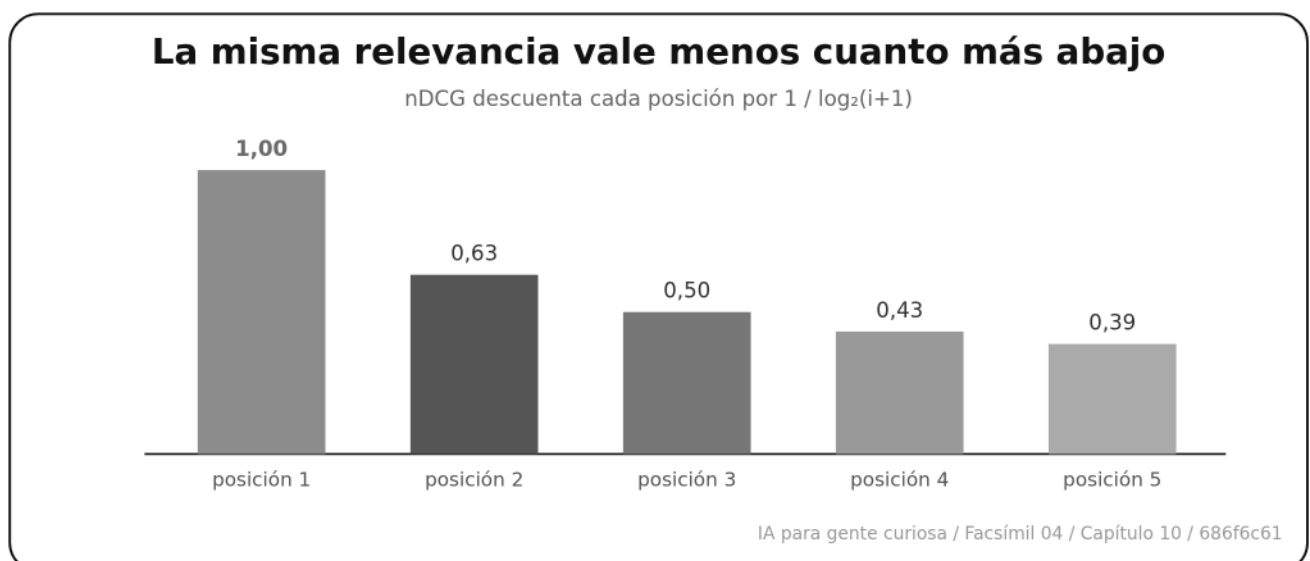
Veámoslo con números, porque es la métrica que más confunde. Recuperas 3 chunks y un humano les da relevancia graduada (0 = no sirve, 1 = ayuda, 2 = relevante, 3 = clave). El ranking real trae las relevancias [2, 0, 3]: un chunk relevante arriba, ruido en medio y el chunk clave enterrado en la posición 3.

$$\text{DCG@3} = \frac{2^2 - 1}{\log_2 2} + \frac{2^0 - 1}{\log_2 3} + \frac{2^3 - 1}{\log_2 4} = 3 + 0 + 3,5 = 6,5$$

El ranking ideal pondría el clave primero, [3, 2, 0], de modo que $\text{IDCG@3} = 7 + 1,893 + 0 = 8,893$. Normalizando:

$$\text{nDCG@3} = \frac{6,5}{8,893} = 0,73$$

La lectura de ingeniería: 0,73 no es desastroso, pero toda la penalización viene de tener el chunk clave en la posición 3 en lugar de la 1. Eso es lo que mide nDCG y no las métricas binarias: **enterrar lo bueno cuesta**, y cuesta porque cada posición descuenta por $1/\log_2(i+1)$, un factor que cae rápido.



Precision@k y recall@k vienen de la tradición de recuperación de información. BEIR y MTEB son recordatorios útiles: un retriever puede brillar en un benchmark y fallar en tu dominio.⁸ Por eso el benchmark público orienta, pero el dataset interno decide.

Métricas de contexto

El retrieval devuelve candidatos. El contexto final es lo que realmente lee el modelo. Entre una cosa y otra puede haber deduplicación, recorte, reordenación, filtros, prioridad de fuentes y presupuesto de tokens.

Ragas llama context precision y context recall a dos ideas útiles.⁹ Las traduzco de forma operativa:

MÉTRICA	PREGUNTA
Context precision	¿El contexto incluido es útil o está lleno de ruido?
Context recall	¿El contexto contiene toda la evidencia necesaria?

Estas son las métricas de contexto que formaliza el marco RAGAS para evaluar RAG de forma automática; aquí las usamos en su versión más simple:¹⁰

$$\text{ContextPrecision} = \frac{\text{chunks útiles en contexto}}{\text{chunks en contexto}}$$

$$\text{ContextRecall} = \frac{\text{evidencias esperadas presentes}}{\text{evidencias esperadas}}$$

La diferencia con retrieval es sutil pero importante. Retrieval@k mide el ranking bruto. Context precision/recall mide el paquete de evidencia que llegó al prompt.

FALLO	RETRIEVAL	CONTEXTO
El chunk bueno no aparece en top 20.	Falló retrieval.	No tiene oportunidad.
El chunk bueno aparece en top 5 pero se recorta.	Retrieval bien.	Falló context builder.
Hay cinco chunks repetidos.	Retrieval dudoso.	Falló deduplicación.
Entra una fuente antigua y otra vigente.	Filtros dudosos.	Falló prioridad o explicación.

Groundedness, faithfulness y citas

Groundedness significa que la respuesta está apoyada por el contexto recuperado. Faithfulness suele usarse de forma cercana: la respuesta no añade hechos que no se desprenden del contexto. TruLens lo conecta con la tríada: contexto relevante, respuesta apoyada en el contexto y respuesta relevante para la pregunta.¹¹

La forma práctica de evaluarlo es separar la respuesta en afirmaciones.

RESPUESTA	CLAIMS
“La ampliación se abre en septiembre y no puede haber pagos vencidos.”	1. La ampliación se abre en septiembre. 2. No puede haber pagos vencidos.

Cada claim se evalúa contra el contexto:

CLAIM	EVIDENCIA	RESULTADO
La ampliación se abre en septiembre.	Chunk norm-2026#c1 .	Sostenido.
No puede haber pagos vencidos.	Chunk norm-2026#c1 .	Sostenido.
Se aprueba automáticamente.	No aparece en contexto.	No sostenido.

Esta es la idea de la faithfulness de RAGAS: descomponer la respuesta en afirmaciones y medir qué fracción se sostiene en el contexto recuperado.¹²

$$\text{Groundedness} = \frac{\text{claims sostenidos}}{\text{claims totales}}$$

La cita añade otra capa. No basta con que la respuesta sea cierta: debe citar el fragmento correcto.

$$\text{CitationPrecision} = \frac{\text{citas válidas usadas}}{\text{citas usadas}}$$

$$\text{CitationRecall} = \frac{\text{evidencias citadas}}{\text{evidencias necesarias}}$$

CASO	GROUNDNESS	CITAS	DIAGNÓSTICO
Respuesta correcta y cita correcta.	Alta.	Alta.	Bien.
Respuesta correcta sin cita.	Alta.	Baja.	Falta trazabilidad.
Respuesta correcta con cita equivocada.	Puede parecer alta.	Baja.	Interfaz de confianza rota.
Respuesta inventada con cita real.	Baja.	Engaños a.	El chunk citado no sostiene la frase.

Un evaluador LLM puede ayudar a evaluar groundedness, pero no es oráculo. Debe recibir rúbrica clara, contexto, respuesta y, si existe, respuesta de referencia. Sus resultados deben compararse con revisión humana en una muestra. OpenAI Graders documenta distintos tipos de evaluadores y la idea de devolver una puntuación numérica contra criterios.¹³

Estar fundamentado no es estar en lo cierto

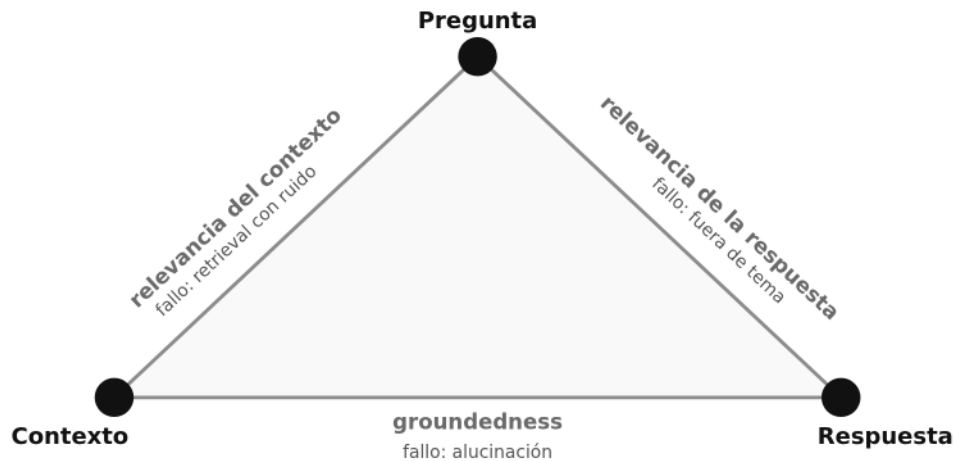
Cuidado con un atajo peligroso: confundir groundedness con corrección. Una respuesta puede apoyarse en el chunk correcto y, aun así, ser falsa, porque lo leyó mal, mezcló dos cláusulas o invirtió una condición («se concede» en vez de «no se concede»). Por eso un RAG serio mide tres cosas distintas, que son la **tríada de evaluación** que populariza TruLens:¹⁴

PREGUNTA	QUÉ COMPARA	FALLO QUE DETECTA
Relevancia del contexto	Pregunta ↔ contexto recuperado	El retrieval trajo ruido.
Groundedness	Contexto ↔ respuesta	La respuesta alucina más allá del contexto.
Relevancia de la respuesta	Pregunta ↔ respuesta	La respuesta se va por las ramas.

Y, por encima de las tres, la **corrección** (*answer correctness*): ¿la respuesta es verdad? Eso se compara contra una respuesta de oro, no solo contra el contexto. Un sistema puede tener groundedness alta y corrección baja a la vez; medir solo lo primero da una falsa sensación de seguridad.

La tríada de evaluación de un RAG

Tres relaciones entre pregunta, contexto y respuesta; cada lado, un fallo distinto



Más, aparte: ¿es verdad? (corrección), que se compara contra una respuesta de oro. IA para gente curiosa / Facsimil 04 / Capítulo 10 / 686f6c61

Evaluar abstención

Abstenerse no significa que el sistema sea torpe. En RAG, abstenerse puede ser la respuesta correcta. Hay preguntas que el corpus no cubre, fuentes contradictorias, permisos insuficientes o evidencia demasiado débil.

La decisión es una regla de umbral, la **opción de rechazo** clásica de la clasificación: responder solo cuando la confianza supera un umbral, y abstenerse si no.¹⁵

$$d(q) = \begin{cases} \text{responder,} & \text{si } s(q) \geq \tau \\ \text{abstenerse,} & \text{si } s(q) < \tau \end{cases}$$

SÍMBOLO	SIGNIFICADO
q	Pregunta.
$s(q)$	Soporte estimado: evidencia, scores, citas, groundedness.
τ	Umbral mínimo para responder.
$d(q)$	Decisión final.

La matriz de abstención:

REALIDAD	EL SISTEMA RESPONDE	EL SISTEMA SE ABSTIENE
Hay evidencia suficiente.	Respuesta evaluable.	Abstención innecesaria.
No hay evidencia suficiente.	Respuesta no sostenida.	Abstención correcta.

Métricas útiles:

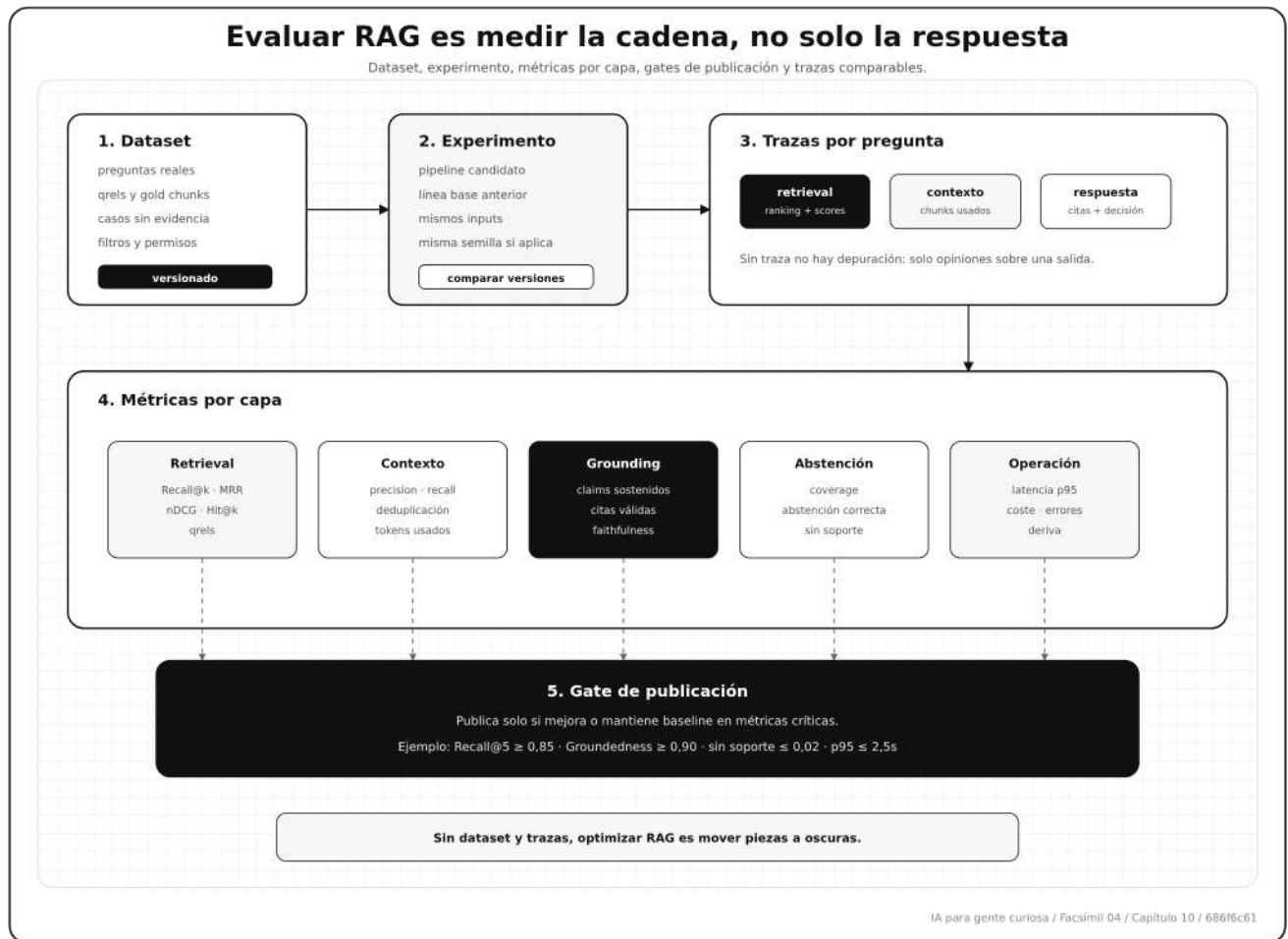
$$\text{Coverage} = \frac{\text{preguntas respondidas}}{\text{preguntas totales}}$$

$$\text{CorrectAbstentionRate} = \frac{\text{abstenciones correctas}}{\text{preguntas no respondibles}}$$

$$\text{UnsupportedAnswerRate} = \frac{\text{respuestas sin soporte}}{\text{preguntas no respondibles}}$$

Subir cobertura no siempre es bueno. Si responde más a costa de inventar más, el sistema empeora. El umbral debe calibrarse con curvas: cuánto ganas en cobertura y cuánto pierdes en precisión de respuesta.

Arquitectura de evaluación



El diagrama tiene una idea central: cada experimento debe producir trazas comparables. Si cambias embeddings, chunking o prompt, no basta con ver una respuesta bonita. Comparas contra baseline y miras qué capa se movió.

Las métricas tienen incertidumbre

Antes de comparar dos versiones, una advertencia que se ignora demasiado: **una métrica medida sobre un dataset es una estimación, no un valor exacto**. Cada métrica es una proporción sobre una muestra de n preguntas, así que tiene un intervalo de confianza, como vimos en el capítulo 1. Para una tasa observada $\hat{p}$ sobre n preguntas, el intervalo al 95 % por la aproximación normal (intervalo de Wald) es:¹⁶

$$\hat{p} \pm 1,96 \frac{\hat{p}(1 - \hat{p})}{n}$$

Con $n = 40$ preguntas y un groundedness de 0,82, el margen es de $\pm 0,12$: dos versiones que difieren en dos o tres puntos están **dentro del ruido**, y declarar ganadora a una es engañarse. Dos consecuencias prácticas. Primera: reporta siempre n y el intervalo, no solo la media. Segunda: para comparar dos versiones sobre las **mismas** preguntas, usa una prueba pareada (McNemar sobre aciertos por consulta, o un bootstrap), que detecta diferencias reales con menos muestra que comparar dos medias sueltas. Un gate honesto exige no solo superar el umbral, sino superarlo por más que el margen de error.

Gates: decidir si una versión publica

Un gate es una regla de publicación. Evita que un cambio que mejora una métrica rompa otra más importante.

MÉTRICA	UMBRAL EJEMPLO	QUÉ PROTEGE
Recall@5	$\geq 0,85$	Que la evidencia aparezca.
nDCG@5	$\geq 0,80$	Que aparezca arriba.
Groundedness	$\geq 0,90$	Que no añada afirmaciones sin soporte.
Citation precision	$\geq 0,95$	Que las citas sean revisables.
Correct abstention	$\geq 0,85$	Que no responda fuera del corpus.
Unsupported answer rate	$\leq 0,02$	Que no conteste sin evidencia.
Latencia p95	$\leq 2,5s$	Que sea usable.
Coste por respuesta	$\leq presupuesto$	Que sea sostenible.

Los umbrales no salen de una tabla universal. Salen del dominio. Un asistente de lectura puede aceptar más incertidumbre que un asistente que orienta trámites administrativos. Lo importante es que el equipo escriba el umbral antes de mirar si su cambio favorito pasa.

Evaladores LLM y rúbricas

Los evaluadores LLM son útiles para escalar revisión, pero hay que usarlos con cuidado. Un evaluador no sustituye una rúbrica: ejecuta una rúbrica.

Una rúbrica mínima para groundedness:

Evalúa si la respuesta está sostenida por el contexto.

Entrada:

- Pregunta del usuario.
- Contexto recuperado con ids de chunk.
- Respuesta generada.

Devuelve JSON:

```
{
  "score": 0.0 a 1.0,
  "claims_no_sostenidos": [...],
  "citas_invalidas": [...],
  "decision": "pasa" | "revisar" | "falla"
}
```

Criterio:

- 1.0: todas las afirmaciones relevantes están sostenidas.
- 0.5: la respuesta mezcla evidencia con inferencias no citadas.
- 0.0: la respuesta contradice o inventa respecto al contexto.

Buenas prácticas:

PRÁCTICA	MOTIVO
Separar evaluación de retrieval y generación.	Un evaluador de respuesta no descubre por sí solo si faltó un chunk.
Guardar entradas del evaluador.	Sin prompt, contexto y respuesta no puedes auditar el score.
Usar muestras revisadas por personas.	El evaluador debe calibrarse contra criterio humano.
Evaluar con varios tipos de pregunta.	Una métrica media puede esconder fallos en tablas, fechas o multi-hop.
Repetir evaluaciones críticas.	Algunos evaluadores tienen variabilidad; mide estabilidad.
No entrenar el sistema para complacer al evaluador.	El objetivo es utilidad verificable, no ganar una métrica estrecha.

Hay una razón de fondo para no tratar al juez como oráculo: tiene **sesgos medidos**. Los evaluadores LLM favorecen la primera respuesta que ven (sesgo de posición), las respuestas más largas (sesgo de verbosidad) y el estilo del propio modelo (autopreferencia).¹⁷ Por eso conviene barajar el orden, controlar la longitud y, sobre todo, **calibrar el juez contra etiquetas humanas** en una muestra antes de fiarte de él a escala. Cuando se usa bien, el

patrón estándar es G-Eval: dar al evaluador una rúbrica explícita y pedirle una cadena de razonamiento antes del score, lo que correlaciona mejor con el juicio humano que pedir una nota a secas.¹⁸

Herramientas de evaluación que ya existen

No hace falta escribir toda la evaluación desde cero. Hay un ecosistema maduro de herramientas, y conviene conocerlo para elegir en vez de reinventar. La distinción clave es **abierta** (una librería que corres tú, normalmente integrable en tu CI) frente a **gestionada** (un servicio que aloja datasets, trazas y paneles).

HERRAMIENTA	TIPO	FUERTE EN	CUÁNDO ELEGIRLA
RAGAS	Abierta	Métricas de RAG (context precision/recall, faithfulness, answer relevance) sin necesidad de respuesta de oro; generación de datasets sintéticos. ¹⁹	Quieres métricas de RAG estándar rápido y un primer dataset.
TruLens	Abierta	La tríada de RAG y feedback functions sobre trazas. ²⁰	Quieres instrumentar groundedness y relevancia sobre tu app.
DeepEval	Abierta	Evals como tests unitarios (estilo <code>pytest</code>), con asserts sobre métricas.	Quieres que la evaluación falle el build como un test más.
Promptfoo	Abierta	Comparación lado a lado de prompts/modelos y evals en CI. ²¹	Comparas versiones y quieres una matriz reproducible.
Arize Phoenix	Abierta	Observabilidad y trazas de LLM/RAG, análisis de fallos.	Quieres ver y depurar trazas, no solo números.
LangSmith	Gestionada	Datasets, trazas, experimentos y evaluadores integrados con LangChain. ²²	Ya usas LangChain y quieres datasets y CI gestionados.
OpenAI Evals / Graders	Mixta	Graders sobre la API y framework de evals reproducibles. ²³	Trabajas sobre la API de OpenAI y quieres graders nativos.

La regla de ingeniería es la de siempre: la herramienta acelera, pero **no decide por ti**. Sea cual sea, sigues necesitando un dataset que represente tu caso, métricas por capa y un gate honesto. Una herramienta que da un número bonito sin un dataset bueno detrás no es evaluación, es decoración.

Evaluación offline, online y sombra

Hay tres modos de evaluación que se complementan.

MOD O	QUÉ HACE	CUÁNDO USARLO
Offline	Ejecuta un dataset fijo contra una versión del RAG.	Antes de publicar cambios.
Sombra	Ejecuta una versión candidata con tráfico real sin mostrarla.	Para medir deriva y casos reales sin afectar a usuarios.
Online	Mide interacción real, feedback, coste, latencia y errores.	Cuando el sistema está en uso.

Offline te da repetibilidad. Sombra te da realidad sin exposición directa. Online te da señales de producto. Ninguna sustituye a las otras.

En el día a día

En el día a día, evaluar un RAG es lo que convierte «creo que funciona» en «sé que funciona, y aquí está la prueba». Es el dataset de 50 preguntas con su fuente esperada que corres antes de publicar cada cambio; es la métrica que distingue si una respuesta mala fue culpa del retrieval, del contexto o de la generación; es el número que enseñas cuando alguien pregunta «¿y esto cuánto acierta?». Sin evaluación, cada cambio en el chunking, el modelo o el prompt es una apuesta a ciegas.

La parte delicada es que evaluar bien cuesta disciplina: hay que construir el dataset con casos reales y no respondibles, medir por capas en vez de mirar solo la respuesta final, y resistir la tentación de quedarse con la media cuando el promedio alto esconde fallos graves por tipo de pregunta.

Por qué debería importarte

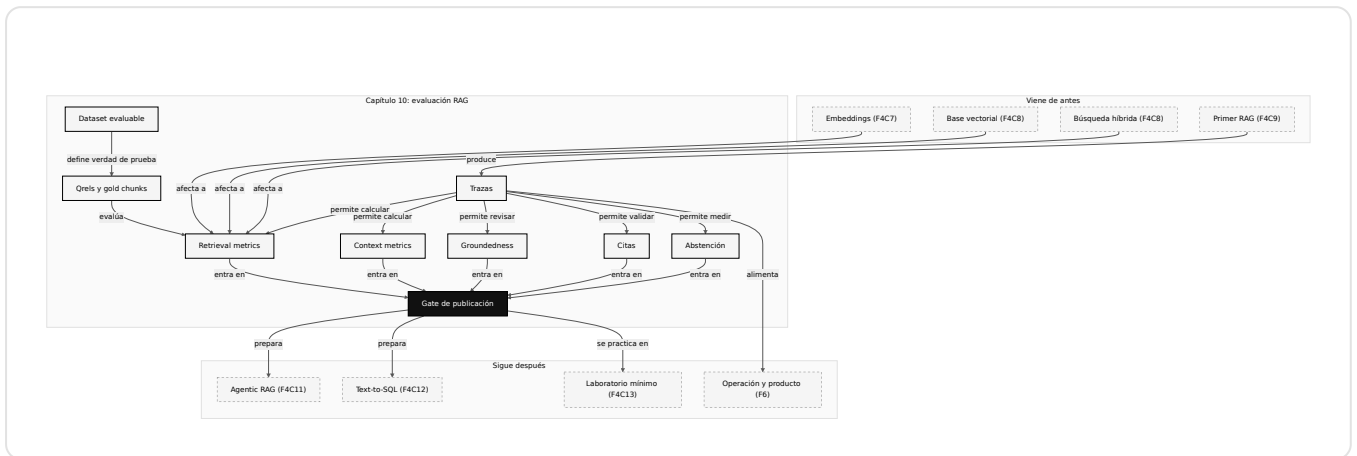
Porque es lo único que separa un RAG que mejora de uno que solo cambia. Sin un conjunto de evaluación, optimizar es mover parámetros y mirar tres ejemplos bonitos; con él, cada decisión (subir top-k, cambiar de embeddings, añadir un reranker) se valida con evidencia y se puede defender. La evaluación es también la condición para tener *gates*: reglas que deciden si una versión publica o no, en lugar de fiarlo todo a la intuición de quien la prueba.

Y porque medir por capas es lo que te hace eficiente diagnosticando. Cuando una respuesta sale mal, la evaluación te dice dónde mirar (retrieval, contexto, generación o abstención) en vez de obligarte a adivinar. Eso es la diferencia entre arreglar la causa y parchear el síntoma.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Evaluar solo la respuesta final	No sabes si falló retrieval, contexto o generación.	Guardar trazas y medir por capas.
Usar tres preguntas elegidas a mano	La demo se adapta a tus expectativas.	Crear dataset con casos reales y no respondibles.
Optimizar recall subiendo top-k sin límite	Traes más evidencia, pero también más ruido y coste.	Medir recall, precision, nDCG y tokens de contexto.
Confiar ciegamente en un evaluador LLM	El evaluador también se equivoca y depende de la rúbrica.	Calibrarlo con revisión humana y guardar entradas.
No medir abstención	El sistema aprende a contestarlo todo.	Incluir preguntas sin evidencia y umbrales.
No versionar corpus e índice	No puedes reproducir por qué respondió algo.	Guardar versión de corpus, embeddings, chunks y prompt.
Mezclar cambios	Si mejoras o empeoras, no sabes por qué.	Cambiar una variable por experimento.
Mirar solo medias	Un promedio alto oculta fallos graves por tipo de pregunta.	Reportar métricas por segmento: tabla, multi-hop, fecha, rol.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	RESPONDE A	DEFINICIÓN ÚTIL
Evaluación offline	¿Mejora antes de publicar?	Prueba repetible sobre dataset fijo.
Evaluación online	¿Qué ocurre en uso real?	Medición con tráfico, feedback, coste y latencia.
Evaluación sombra	¿Cómo pruebo sin mostrar al usuario?	Ejecutar versión candidata en paralelo y registrar resultados.
Qrel	¿Qué chunk debería recuperar?	Juicio de relevancia pregunta-documento.
Precision@k	¿Cuánto ruido hay en top-k?	Proporción de resultados relevantes entre los k primeros.
Recall@k	¿Apareció la evidencia esperada?	Proporción de evidencias recuperadas.
Hit@k	¿Aparece al menos una evidencia?	Indicador binario de recuperación suficiente mínima.
MRR	¿Cuán pronto aparece la primera evidencia?	Media del inverso de la primera posición relevante.
nDCG	¿Lo más útil aparece arriba?	Métrica con relevancia graduada y descuento por posición.
Context precision	¿El contexto final está limpio?	Proporción de chunks útiles dentro del contexto usado.
Context recall	¿El contexto contiene lo necesario?	Proporción de evidencia esperada incluida en el prompt.
Groundedness	¿La respuesta se apoya en contexto?	Claims sostenidos por evidencia recuperada.
Citation precision	¿Las citas usadas son válidas?	Citas que apuntan a evidencia real entre citas usadas.
Citation recall	¿Cité toda la evidencia necesaria?	Evidencias necesarias citadas entre evidencias esperadas.
Coverage	¿Cuánto responde el sistema?	Porcentaje de preguntas no abstendidas.
Gate	¿Publicamos esta versión?	Regla de aceptación con umbrales técnicos y de producto.

TÉRMINO	RESPONDE A	DEFINICIÓN ÚTIL
Evaluador LLM	¿Quién puntúa respuestas abiertas?	Modelo evaluador guiado por una rúbrica auditable.

Antes de pasar página

- ☐ ¿Sé explicar por qué evaluar una respuesta final no basta? (Si no, vuelve a «Qué significa evaluar por capas».)
- ☐ ¿Sé construir una fila mínima de dataset de evaluación? (Si no, vuelve a «Dataset de evaluación: el corazón del sistema».)
- ☐ ¿Sé qué es un qrel y cuándo usar relevancia graduada? (Si no, vuelve a «Qrels y relevancia graduada».)
- ☐ ¿Sé calcular precision@k, recall@k, hit@k, MRR y nDCG? (Si no, vuelve a «Métricas de retrieval».)
- ☐ ¿Sé separar retrieval metrics de context metrics? (Si no, vuelve a «Métricas de contexto».)
- ☐ ¿Sé evaluar groundedness separando claims y evidencia? (Si no, vuelve a «Groundedness, faithfulness y citas».)
- ☐ ¿Sé medir precisión y recall de citas? (Si no, vuelve a «Groundedness, faithfulness y citas».)
- ☐ ¿Sé construir una matriz de abstención? (Si no, vuelve a «Evaluar abstención».)
- ☐ ¿Sé definir gates de publicación con umbrales explícitos? (Si no, vuelve a «Gates: decidir si una versión publica».)
- ☐ ¿Sé usar un evaluador LLM sin tratarlo como oráculo? (Si no, vuelve a «Evaluadores LLM y rúbricas».)
- ☐ ¿Sé guardar trazas suficientes para comparar versiones? (Si no, vuelve a «Arquitectura de evaluación».)

En resumen

IDEA FUERZA	DETALLE
Evaluar RAG es evaluar una cadena.	Corpus, parsing, retrieval, contexto, generación, citas y abstención.
Retrieval se mide antes del LLM.	Si no recuperas la evidencia, la generación no puede arreglarlo.
Groundedness exige claims.	No basta con una sensación global de respuesta correcta.
Citar también se evalúa.	Una cita debe sostener la frase que acompaña.
Abstenerse puede ser correcto.	Coverage alto con respuestas sin soporte es mala señal.
Un evaluador LLM necesita rúbrica.	La herramienta puntúa; el criterio lo diseña el equipo.
Un gate protege producto.	Publicas si la versión candidata supera umbrales críticos.
Sin trazas no hay aprendizaje.	Cada consulta debe dejar ranking, contexto, respuesta, citas y costes.

Para saber más

Arize Phoenix. (2026). *Evaluate RAG*.

<https://arize.com/docs/phoenix/cookbook/evaluation/evaluate-rag>

Arize Phoenix. (2026). *Evaluation concepts*.

<https://arize.com/docs/phoenix/evaluation/concepts-evs/evaluation>

Es, S., James, J., Espinosa-Anke, L. y Schockaert, S. (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. <https://arxiv.org/abs/2309.15217>

LangChain. (2026). *Evaluate a RAG application*.

<https://docs.langchain.com/langsmith/evaluate-rag-tutorial>

LlamaIndex. (2026). *Evaluation modules*.

https://developers.llamaindex.ai/python/framework/module_guides/evaluating/modules/

OpenAI. (2026). *Graders*. <https://developers.openai.com/api/docs/guides/graders>

Ragas. (2026). *List of available metrics*.

https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/

Thakur, N. y otros (2021). *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models*. <https://arxiv.org/abs/2104.08663>

Notas

1. Es, S., James, J., Espinosa-Anke, L. y Schockaert, S. (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. <https://arxiv.org/abs/2309.15217>. ↵
2. Ragas. (2026). *List of available metrics*. https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/. Consultado el 25 de mayo de 2026. ↵
3. TruLens. (2026). *RAG Triad*. https://www.trulens.org/getting_started/core_concepts/rag_triad/. Consultado el 25 de mayo de 2026. ↵
4. LangChain. (2026). *Evaluate a RAG application*. <https://docs.langchain.com/langsmith/evaluate-rag-tutorial>. LlamaIndex. (2026). *Evaluation modules*. https://developers.llamaindex.ai/python/framework/module_guides/evaluating/modules/. Arize Phoenix. (2026). *Evaluate RAG*. <https://arize.com/docs/phoenix/cookbook/evaluation/evaluate-rag>. Consultado el 25 de mayo de 2026. ↵
5. OpenAI. (2026). *Graders*. <https://developers.openai.com/api/docs/guides/graders>. Consultado el 25 de mayo de 2026. ↵
6. Shahul Es y otros (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. <https://arxiv.org/abs/2309.15217>. Documentación de generación de datasets: <https://docs.ragas.io>. ↵
7. Christopher D. Manning, Prabhakar Raghavan e Hinrich Schütze. (2008). *Introduction to Information Retrieval*. Cambridge University Press, cap. 8. <https://nlp.stanford.edu/IR-book/>. Para nDCG: Kalervo Järvelin y Jaana Kekäläinen. (2002). *Cumulated Gain-based Evaluation of IR Techniques*. ACM TOIS, 20(4), 422-446. <https://doi.org/10.1145/582415.582418>. ↵
8. Thakur, N. y otros (2021). *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models*. <https://arxiv.org/abs/2104.08663>. Muennighoff, N. y otros (2023). *MTEB: Massive Text Embedding Benchmark*. <https://arxiv.org/abs/2210.07316>. ↵
9. Ragas, 2026. ↵
10. Shahul Es y otros (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. <https://arxiv.org/abs/2309.15217>. ↵
11. TruLens, 2026. ↵
12. Shahul Es y otros (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. <https://arxiv.org/abs/2309.15217>. ↵

- .3. OpenAI, 2026. ↵
- .4. TruLens. (2026). *RAG Triad*. https://www.trulens.org/getting_started/core_concepts/rag_triad/. ↵
- .5. C. K. Chow. (1970). *On Optimum Recognition Error and Reject Tradeoff*. IEEE Transactions on Information Theory, 16(1), 41-46. <https://doi.org/10.1109/TIT.1970.1054406>. ↵
- .6. El intervalo de Wald usa la aproximación normal de la binomial; para n pequeño conviene el intervalo de Wilson. Lawrence D. Brown, T. Tony Cai y Anirban DasGupta. (2001). *Interval Estimation for a Binomial Proportion*. Statistical Science, 16(2), 101-133. <https://doi.org/10.1214/ss/1009213286>. ↵
- .7. Lianmin Zheng y otros (2023). *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. NeurIPS Datasets and Benchmarks. <https://arxiv.org/abs/2306.05685>. ↵
- .8. Yang Liu y otros (2023). *G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment*. EMNLP. <https://arxiv.org/abs/2303.16634>. ↵
- .9. Shahul Es y otros (2023). *RAGAS*. <https://arxiv.org/abs/2309.15217>. Documentación: <https://docs.ragas.io>. ↵
- .10. TruLens. (2026). *RAG Triad*. <https://www.trulens.org>. ↵
- .11. promptfoo. (2026). *LLM evals & red teaming*. <https://www.promptfoo.dev>. ↵
- .12. LangChain. (2026). *Evaluate a RAG application*. <https://docs.smith.langchain.com>. ↵
- .13. OpenAI. (2026). *Graders*. <https://developers.openai.com/api/docs/guides/graders>. ↵

Capítulo 11: Agentic RAG y GraphRAG: cuándo complicar

Cuando un RAG fijo se queda corto

En el [capítulo 09](#) construimos el RAG básico: partir una colección, buscar fragmentos, pasarlos al modelo y responder con citas. En el [capítulo 10](#) aprendimos a medir si la evidencia aparece, si el contexto sostiene la respuesta y si el sistema sabe abstenerse.

Ese patrón funciona muy bien para preguntas directas: “¿qué dice esta normativa sobre X?”, “¿cuál es el plazo?”, “¿dónde se configura este parámetro?”. Pero en proyectos reales aparecen preguntas menos limpias:

- “Compara los requisitos de matrícula con los requisitos de beca y dime dónde se contradicen.”
- “¿Qué temas se repiten en todas las quejas del alumnado este curso?”
- “¿Qué documentos explican por qué cambió esta política?”
- “Busca primero en la normativa; si no basta, mira FAQ, calendario y expediente.”
- “Esta respuesta cita una fuente floja; revisa si hay otra fuente mejor.”

Aquí ya no basta con una única búsqueda top-k. El sistema necesita planificar, reescribir consultas, dividir la pregunta, escoger fuentes, comprobar si lo recuperado basta, volver a buscar si no basta y quizá consultar un grafo de relaciones. Eso es lo que solemos llamar **Agentic RAG** y **GraphRAG**. La pregunta importante no es “¿puedo hacerlo?”, sino “¿merece la pena pagar la complejidad?”.

Estado del arte con fecha de corte

Fecha de corte: 26 de mayo de 2026.

Fuentes consultadas ese día: documentación oficial de LangChain sobre arquitecturas RAG, documentación de LlamaIndex sobre estrategias agentic, documentación de Microsoft GraphRAG, papers de ReAct, Toolformer, HyDE, Self-RAG, Corrective RAG, RAPTOR y GraphRAG, y el capítulo anterior de evaluación del propio facsímil.

El patrón base sigue siendo RAG: recuperar información externa en tiempo de consulta para responder con contexto específico.¹ Lo que cambia es quién decide los pasos. LangChain distingue entre RAG de dos pasos, Agentic RAG e híbridos con validación; en su documentación, el RAG de dos pasos es más predecible y el agentic gana flexibilidad a costa

de latencia variable.² LlamaIndex describe estrategias agentic como routing, query transformations, sub-question query engines y agentes de datos sobre motores RAG existentes.³

La parte “agentic” bebe de trabajos como ReAct, que intercalan razonamiento y acciones para consultar fuentes externas durante la resolución.⁴ Toolformer exploró cómo modelos de lenguaje pueden aprender a usar herramientas externas mediante ejemplos auto-supervisados.⁵ En retrieval avanzado aparecen técnicas como HyDE, que genera un documento hipotético para buscar textos reales cercanos; Self-RAG, que introduce recuperación y crítica/reflexión; Corrective RAG, que evalúa la calidad de documentos recuperados y activa acciones correctivas si la evidencia es débil; y Adaptive RAG, que clasifica la complejidad de cada pregunta para decidir cuánta recuperación aplicar, desde ninguna hasta varios pasos.⁶

GraphRAG se volvió relevante porque muchas preguntas no piden “el chunk más parecido”, sino entender relaciones o patrones globales del corpus. El paper de Microsoft GraphRAG plantea un índice basado en grafo de entidades y resúmenes de comunidades para responder preguntas de comprensión global sobre colecciones privadas.⁷ La documentación de GraphRAG separa local search, global search, DRIFT search, basic search y question generation.⁸

Qué no es complicar bien

Complicar un RAG no significa meter un agente delante de todo. Si el sistema siempre hace una pregunta directa sobre un documento vigente, un RAG de dos pasos puede ser mejor: menos latencia, menos coste, menos puntos de fallo y evaluación más sencilla.

Tampoco significa que el modelo “piense libremente” hasta encontrar la respuesta. En ingeniería, un Agentic RAG serio tiene herramientas permitidas, límites de pasos, trazas, umbrales de evidencia y reglas de salida. Si no puedes reconstruir qué buscó, qué encontró, qué descartó y por qué respondió, no has ganado inteligencia: has perdido depuración.

GraphRAG tampoco es “usar una base de grafos porque suena potente”. Un grafo merece la pena cuando las relaciones importan: entidades, dependencias, comunidades, jerarquías, trazabilidad entre documentos, patrones globales o preguntas que no se resuelven con un párrafo aislado. Si tu corpus son veinte FAQs cortas, GraphRAG puede ser una mudanza para cruzar la calle.

Qué sí es Agentic RAG

Agentic RAG es un RAG donde el sistema puede decidir pasos intermedios antes de responder. La palabra clave no es “autonomía”; la palabra clave es **control del flujo**.

Un RAG fijo hace esto:

```
pregunta -> retrieval -> contexto -> respuesta
```

Un Agentic RAG puede hacer esto:

```
pregunta -> diagnosticar  
          -> elegir herramienta  
          -> buscar  
          -> evaluar evidencia  
          -> responder o abstenerse
```

Técnicamente, un agente es un bucle de estado, acción y transición: el modelo observa un estado, elige una acción (buscar, leer, validar, responder) y produce un nuevo estado. Es la estructura del proceso de decisión secuencial, y la base del patrón ReAct, que intercala razonamiento y acciones de búsqueda.⁹

$$s_t = (x, H_t, E_t, B_t)$$

$$a_t = \pi_{\theta}(s_t)$$

$$s_{t+1} = \text{step}(s_t, a_t)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Pregunta original.	“Compara normativa y FAQ sobre pagos pendientes”.
H_t	Historial de pasos hasta el momento.	Búsqueda en normativa, lectura de FAQ, validación.
E_t	Evidencia acumulada.	Chunks, citas, resultados SQL o relaciones de grafo.
B_t	Presupuesto restante.	Máximo 4 pasos, 2 búsquedas y 1 consulta externa.
s_t	Estado del flujo en el paso t .	Lo que el sistema sabe y puede hacer ahora.
a_t	Acción elegida.	buscar_normativa , consultar_grafo , responder .
π_θ	Política de decisión del modelo o del router.	Decide el siguiente paso.
step	Ejecución controlada de una acción.	Llama a una herramienta y actualiza la traza.

La lista de acciones no debería ser infinita. En un sistema real se define algo así:

ACCIÓN	QUÉ HACE	CUÁNDO TIENE SENTIDO	QUÉ REGISTRA
buscar_texto	Busca chunks por consulta.	Pregunta directa o evidencia textual.	Query, filtros, top-k, scores y chunks.
buscar_hibrido	Combina vector, BM25 y filtros.	Hay términos exactos y significado aproximado.	Rankings de cada señal y fusión.
descomponer	Divide una pregunta en subpreguntas.	La respuesta depende de varias fuentes.	Subpreguntas y razón de cada una.
router	Elige corpus, índice o herramienta.	Hay normativa, FAQ, SQL, tickets o grafo.	Opción elegida y alternativas descartadas.
evaluar_evidencia	Mide si lo recuperado basta.	Antes de redactar o cuando hay duda.	Soporte, citas candidatas y huecos.
consultar_grafo	Busca relaciones entre entidades.	Importan dependencias, comunidades o vínculos.	Nodos, aristas, fuente y camino usado.
consultar_tabla	Consulta datos estructurados.	Fechas, importes, estados o conteos exactos.	Consulta, resultado y validación.
responder	Redacta con citas.	Evidencia suficiente.	Claims, citas y decisión final.
abstenerse	No responde por falta de soporte.	Evidencia insuficiente o permisos insuficientes.	Qué faltó y qué fuente sería necesaria.

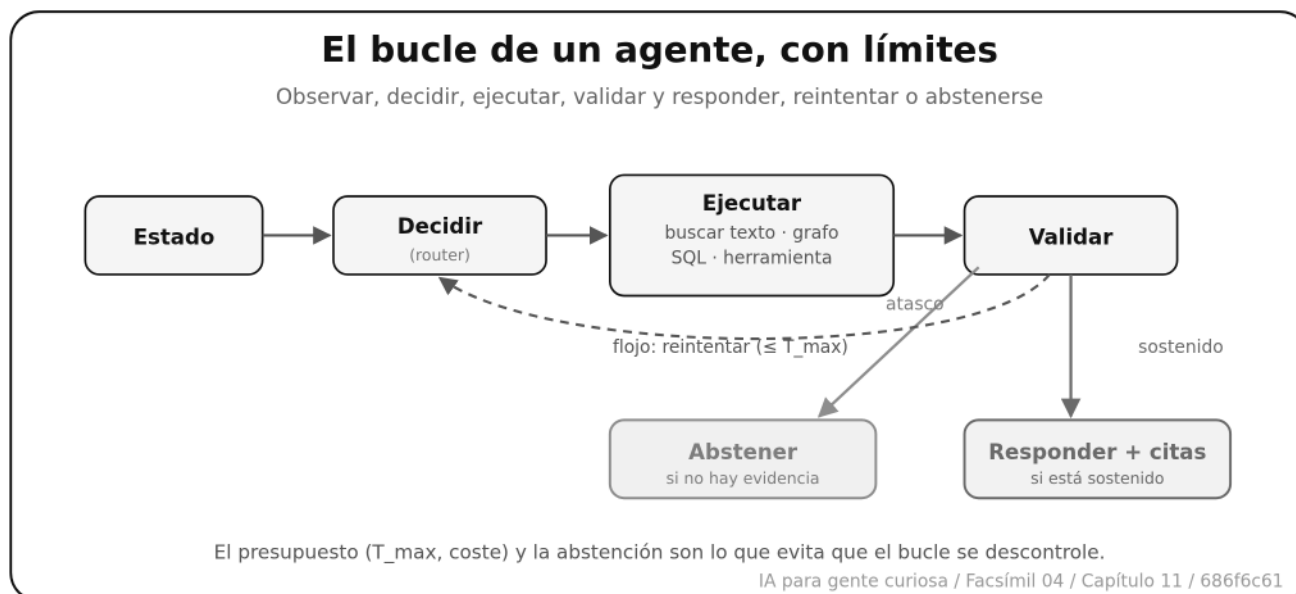
El coste de un agente es la suma del coste de cada paso, y por eso un agente sin tope de pasos es un agente sin tope de factura:

$$C_{\text{total}} = \sum_{t=1}^T c(a_t)$$

$$T \leq T_{\text{max}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
C_{total}	Coste total de la ejecución.	Tokens, llamadas, latencia o euros.
$c(a_t)$	Coste de la acción a_t .	Una búsqueda cuesta poco; una llamada LLM cuesta más.
T	Número de pasos ejecutados.	3 pasos.
T_{max}	Límite máximo permitido.	5 pasos por pregunta.

La frase que debería quedarse en la cabeza: **Agentic RAG no es “hacer más cosas”; es decidir si hace falta hacerlas y dejar rastro de cada decisión.**



El porqué de los bucles: planificación y reflexión

¿Por qué un bucle mejora a una sola pasada? Porque permite **planificar y corregir**. ReAct fue el patrón base (razonar y actuar intercalados), pero la investigación añadió dos ideas que conviene conocer. **Reflexion** hace que el agente, tras un fallo, escriba en lenguaje natural qué salió mal y lo use como memoria para el siguiente intento, en vez de repetir el error.¹⁰ **Tree-of-Thoughts** explora varias ramas de razonamiento y se queda con la mejor, en lugar de comprometerse con la primera.¹¹

La lección de ingeniería no es «usa siempre el más sofisticado». Es la contraria: estas técnicas pagan en tareas de varios pasos con verificación clara (resolver, comprobar, reintentar), y son **puro derroche** en la mayoría de preguntas de un RAG, que se resuelven en una pasada. Conocerlas sirve para saber qué tienes disponible cuando una tarea de verdad lo necesita, no para meterlas por defecto.

Tipos de RAG avanzado y casos de uso

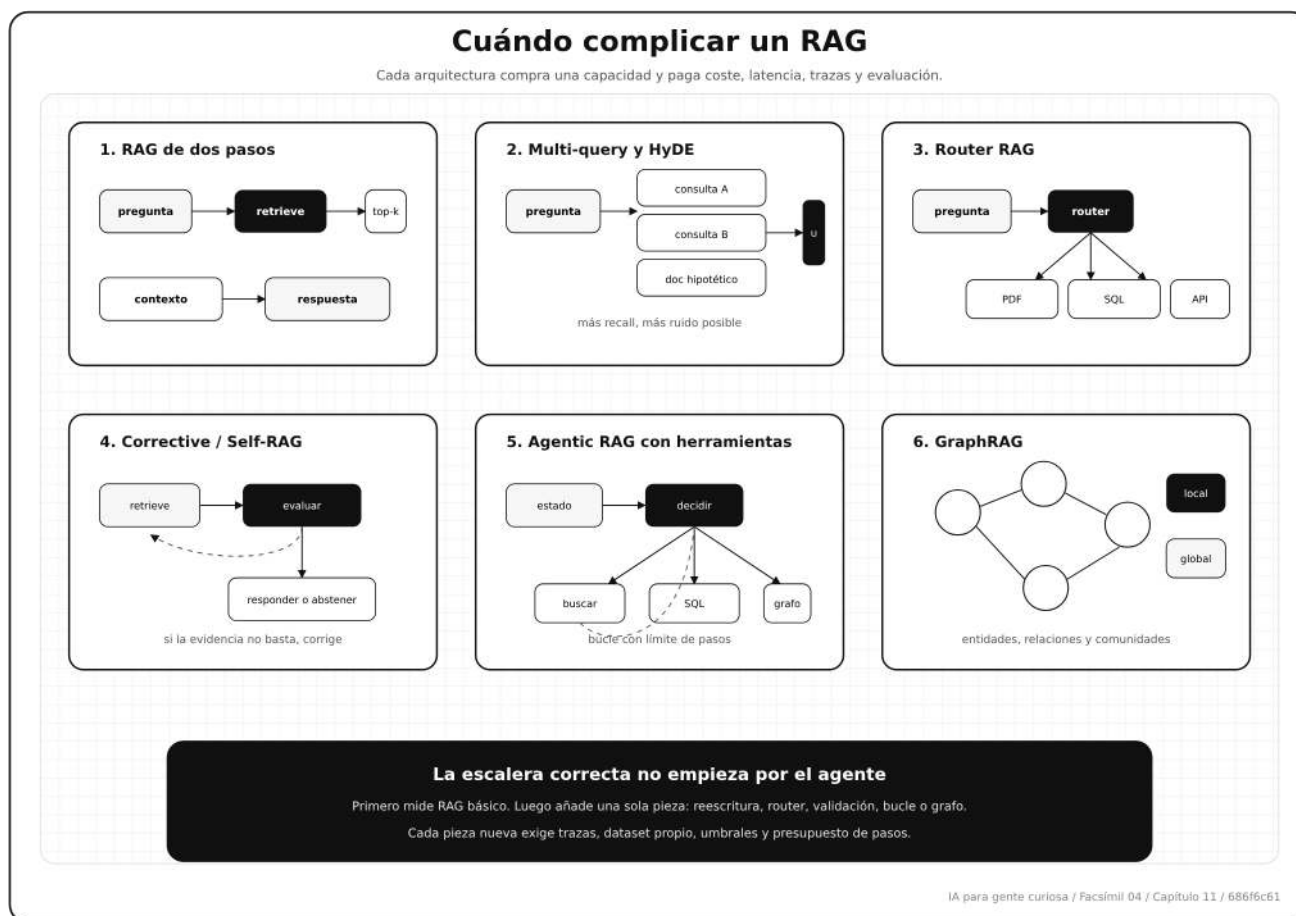
Antes de elegir una arquitectura, conviene ponerle nombre a cada patrón. Muchos problemas se arreglan con una pieza pequeña: reescritura, router o validación. No todos necesitan un bucle completo.

TIPO	QUÉ SIGNIFICA	CASO DONDE ENCAJA	NO SIRVE PARA
RAG de dos pasos	Siempre recupera primero y genera después.	FAQ, documentación técnica, normativa directa.	Preguntas que requieren decidir varias rutas.
Multi-query retrieval	Genera varias consultas para una misma pregunta y une resultados.	“doble factor”, “2FA”, “MFA” y “autenticación” pueden aparecer en documentos distintos.	Fuentes con permisos complejos si no filtras antes.
HyDE	Genera un texto hipotético que parece responder y busca documentos reales similares.	Consulta vaga sin etiquetas de relevancia ni ejemplos de entrenamiento.	Dominios donde el texto hipotético puede desviar hacia detalles falsos.
Query decomposition	Divide una pregunta compuesta en subpreguntas.	Comparar beca, matrícula y pagos pendientes.	Preguntas simples donde añade latencia sin necesidad.
Router RAG	Elige corpus, índice, herramienta o flujo.	Normativa en PDFs, estado vivo en SQL y manuales en Markdown.	Sistemas sin fuentes diferenciadas.
Hybrid RAG con validación	Recupera, evalúa evidencia, corrige o regenera si hace falta.	Dominios donde citar mal rompe confianza.	Prototipos donde aún no hay dataset de evaluación.
Corrective RAG	Evalúa la calidad de los documentos recuperados y activa otra búsqueda si no basta.	Corpus incompleto, preguntas ambiguas o retrieval frágil.	Sistemas sin fuente alternativa ni umbral definido.
Self-RAG	Decide cuándo recuperar y critica pasajes/respuestas mediante señales internas o entrenadas.	Respuestas largas donde no siempre hace falta recuperar.	Integraciones donde necesitas flujo totalmente determinista.
Adaptive RAG	Clasifica la complejidad de la pregunta y elige la estrategia, desde no recuperar hasta varios pasos.	Sistemas con mezcla de preguntas simples y complejas, donde aplicar siempre lo más caro sale mal.	Cargas homogéneas donde una sola estrategia ya basta.
RAPTOR	Construye árboles de resúmenes para recuperar a varios niveles de abstracción.	Manuales largos, informes extensos, libros internos.	Corpus pequeño con respuestas puntuales.
GraphRAG local	Usa grafo para preguntas sobre entidades y relaciones concretas.	“¿Qué políticas dependen de este requisito?”	Preguntas puramente textuales sin relaciones útiles.

TIPO	QUÉ SIGNIFICA	CASO DONDE ENCAJA	NO SIRVE PARA
GraphRAG global	Usa resúmenes de comunidades para preguntas sobre el corpus completo.	“¿Qué patrones aparecen en todas las incidencias?”	Preguntas que solo piden una fecha exacta.
RAG con herramientas	El modelo consulta búsqueda, SQL, APIs o grafo según necesidad.	“Busca la política y comprueba si mi expediente cumple.”	Saltarse permisos o validar por intuición.

La clave práctica: cada fila añade una nueva promesa y una nueva deuda. Multi-query promete más cobertura y paga más búsquedas. Router promete elegir mejor la fuente y paga clasificación. GraphRAG promete visión relacional y paga extracción, grafo, resúmenes y evaluación nueva.

Arquitecturas, una por una



El diagrama enseña la escalera de complejidad. Si una pregunta se resuelve con un RAG de dos pasos, no necesitas un bucle. Si falla por vocabulario, quizá basta multi-query. Si falla por elegir mal la fuente, quizá basta router. Si falla porque el corpus completo tiene patrones, GraphRAG puede tener sentido.

Recuperar bajo demanda y corregir: Self-RAG y CRAG

Dos arquitecturas merecen mirarse de cerca porque atacan dos errores opuestos del RAG fijo: recuperar siempre (aunque no haga falta) y confiar en lo recuperado (aunque sea malo).

Self-RAG entrena al modelo para que **decida cuándo recuperar** y para que critique su propio trabajo con tokens de reflexión: emite señales que indican si un pasaje es relevante, si la respuesta está sostenida por él y si responde a la pregunta.¹² El valor para ingeniería: no recupera para preguntas que el modelo ya sabe (ahorra coste y ruido) y se autoevalúa antes de responder.

Corrective RAG (CRAG) añade un **evaluador de recuperación**: puntúa la calidad de los chunks recuperados y, si es baja, descompone, reescribe o busca en una fuente alternativa (por ejemplo, la web) en lugar de responder con evidencia floja.¹³ El valor: convierte un retrieval frágil en una respuesta robusta sin esperar a que falle en producción.

La idea común, y la lección del capítulo: **recuperar no es siempre, y confiar no es gratis**. Ambas técnicas son baratas conceptualmente (una decisión y una validación) comparadas con saltar directamente a un agente con bucle completo.

GraphRAG: qué cambia cuando aparece un grafo

Un grafo representa entidades y relaciones. En vez de tratar el corpus solo como chunks sueltos, intentamos extraer una estructura:

$$G = (V, E)$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
G	Grafo de conocimiento extraído o curado.	Grafo de normativa, trámites y requisitos.
V	Conjunto de nodos o entidades.	“Ampliación de matrícula”, “pagos pendientes”, “beca general”.
E	Conjunto de aristas o relaciones.	“requiere”, “contradice”, “se aplica a”, “depende de”.

Un grafo de conocimiento no es solo nodos y aristas: cada arista (una relación entre dos entidades) debería guardar su procedencia para ser auditable, como insiste el trabajo de GraphRAG.¹⁴ La estructura de una arista es:

$$e = (v_i, r, v_j, fuente, confianza)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
v_i	Entidad origen.	“Ampliación de matrícula”.
r	Relación.	“requiere”.
v_j	Entidad destino.	“no tener pagos pendientes vencidos”.
fuelle	Documento o chunk que sostiene la relación.	norm-2026#art-14 .
confianza	Señal de extracción o validación.	0,82.

La diferencia con RAG básico es que GraphRAG puede contestar usando caminos, vecindarios o comunidades:

MODO	QUÉ BUSCA	PREGUNTA TÍPICA
Local search	Entidades y relaciones cercanas a una entidad.	“¿Qué requisitos dependen de pagos pendientes?”
Global search	Resúmenes de comunidades del grafo.	“¿Qué patrones aparecen en las incidencias del curso?”
DRIFT search	Combina señal comunitaria con seguimiento local más amplio.	“Explora este tema y saca líneas de investigación.”
Question generation	Propone preguntas siguientes para investigar el corpus.	“¿Qué debería revisar ahora?”

La documentación de Microsoft GraphRAG lo expresa así: local search combina datos del grafo con chunks originales; global search busca sobre community reports en estilo map-reduce; DRIFT usa información de comunidades para ampliar el punto de partida local.¹⁵

GraphRAG encaja cuando la pregunta no vive en un solo fragmento:

CASO CERCANO	POR QUÉ GRAPHRAG AYUDA
Analizar miles de tickets de soporte.	Las relaciones entre producto, síntoma, versión y solución revelan comunidades.
Revisar normativa dispersa.	Las dependencias entre requisitos importan tanto como cada párrafo.
Explorar literatura académica.	Autores, métodos, datasets y resultados forman un grafo natural.
Mapear incidencias de producto.	Puede mostrar temas recurrentes y relaciones entre módulos.
Entender una organización documental.	Las entidades conectan documentos que no comparten las mismas palabras.

Pero también tiene costes:

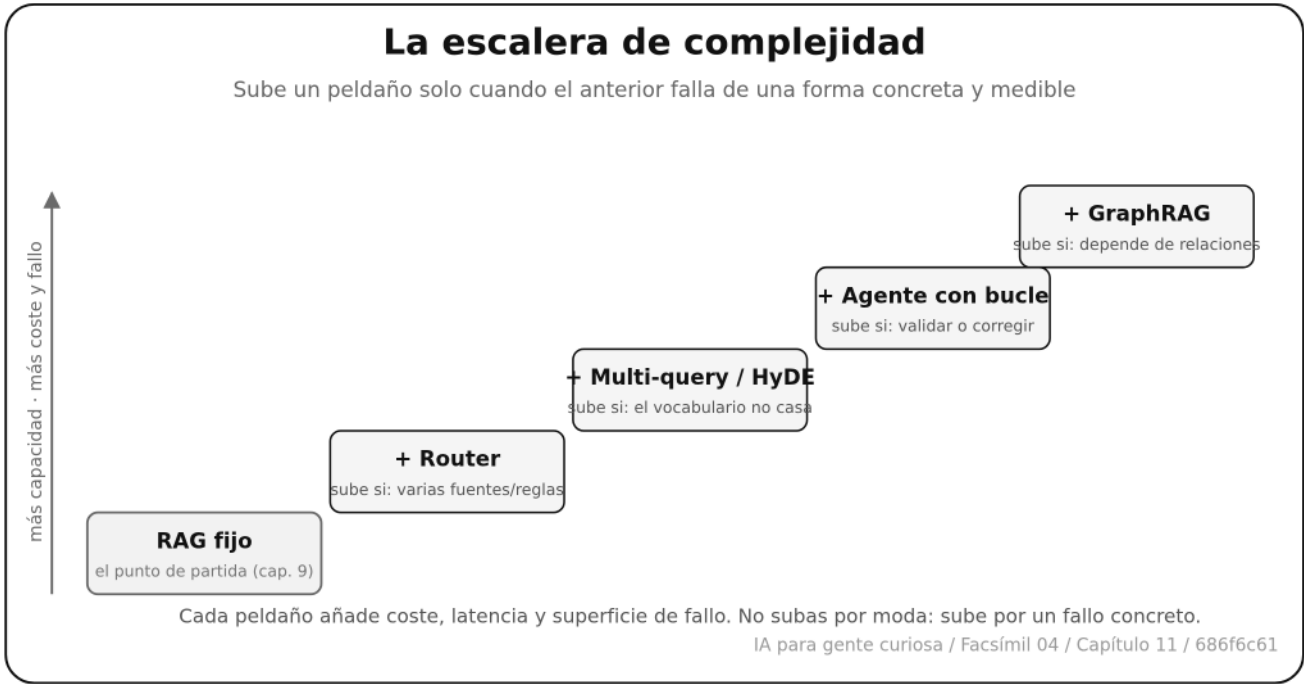
COSTE	QUÉ IMPLICA
Extracción	El modelo debe detectar entidades y relaciones con calidad suficiente.
Normalización	“doble factor”, “2FA” y “MFA” quizá son la misma entidad.
Actualización	Si cambian documentos, hay que actualizar grafo, resúmenes e índices.
Evaluación	Ya no basta medir top-k; hay que medir caminos, relaciones y resúmenes.
Explicabilidad	Una respuesta global debe enseñar qué comunidades o fuentes la sostienen.

Cómo elegir sin montar una catedral

La pregunta de arquitectura se puede formular como diagnóstico:

SÍNTOMA	PRIMERA MEJORA RAZONABLE	SI NO BASTA
No aparece el documento correcto por vocabulario.	Multi-query o HyDE.	Entrenar o cambiar embeddings; añadir BM25/fusión.
La pregunta mezcla varias cosas.	Query decomposition.	Agentic RAG con subpreguntas y validación.
Hay varias fuentes con reglas distintas.	Router.	Herramientas especializadas por fuente.
El sistema trae contexto flojo.	Retrieval validation y reranker.	Corrective RAG o búsqueda alternativa.
El corpus es largo y jerárquico.	Resúmenes por sección.	RAPTOR o índice jerárquico.
La respuesta depende de relaciones.	Grafo de entidades.	GraphRAG local.
La pregunta pide patrones del corpus entero.	Resúmenes agregados.	GraphRAG global.
Hace falta estado exacto.	Tool o SQL.	Capítulo 12: Text-to-SQL y herramientas de datos.

Regla práctica: añade una sola pieza por experimento. Si incorporas router, multi-query, GraphRAG y validación a la vez, cuando mejore o empeore no sabrás por qué.



Antes de complicar: la lista de comprobación

La escalera solo sirve si te resistes a subirla por moda. Antes de añadir un peldaño (router, agente, GraphRAG), pasa esta lista. Si falta una sola casilla, todavía no toca:

- ☐ **El RAG simple falla, y sé exactamente en qué consulta.** Tengo el caso reproducido, no una sensación.
- ☐ **El fallo es de la pieza que voy a añadir.** Si falla por retrieval básico, un grafo no lo arregla.
- ☐ **Tengo presupuesto.** Hay un tope de pasos ($T_{\max}$), de coste y de latencia, y una condición de parada.
- ☐ **Guardo traza por paso.** Si la nueva pieza falla, podré ver qué decidió y por qué.
- ☐ **Puedo medir la mejora.** Tengo un baseline y un dataset para hacer un *ablation* (añadir solo esta pieza y comparar).
- ☐ **La mejora compensa la factura.** El ejemplo de coste y latencia de este capítulo dice que sí.

La regla detrás de la lista: complicar es una decisión de ingeniería con coste, no un gesto de sofisticación. Un sistema simple que funciona y se puede medir vale más que uno complejo que nadie sabe por qué responde lo que responde.

Evaluar RAG avanzado

Todo lo que complicas debe medirse. Un Agentic RAG no se evalúa solo por respuesta final: se evalúa por ruta.

CAPA	MÉTRICA O REVISIÓN	PREGUNTA
Router	Accuracy de ruta o matriz de confusión.	¿Eligió la fuente correcta?
Multi-query	Recall@k por consulta y unión final.	¿Las variantes trajeron evidencia nueva o solo ruido?
Decompositio n	Subpreguntas necesarias y suficientes.	¿Dividió bien el problema?
Corrective RAG	Tasa de corrección útil.	¿Volvió a buscar cuando debía?
Agentic loop	Pasos, coste, latencia y salida.	¿Gastó pasos con sentido?
Graph local	Nodos/aristas correctos y fuentes.	¿El camino usado está sostenido?
Graph global	Cobertura, diversidad y trazabilidad.	¿El resumen global representa el corpus?
Respuesta	Groundedness, citas y abstención.	¿Lo dicho está sostenido?

La decisión de adoptar (o no) una arquitectura más compleja se puede escribir como un gate: solo complicas si ganas calidad por encima de un umbral sin romper los presupuestos de coste y tiempo. Un gate mínimo:

$$G \geq \tau_g$$

$$C \leq C_{\max}$$

$$T \leq T_{\max}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
G	Groundedness o soporte mínimo.	0,92.
τ_g	Umbral de soporte.	0,90.
C	Coste total de la ejecución.	0,012 euros o 8.000 tokens.
$C_{\max}$	Coste máximo aceptable.	0,02 euros.
T	Pasos realizados.	4.
$T_{\max}$	Máximo de pasos.	5.

Para GraphRAG hay una evaluación adicional: no basta con que la respuesta suene bien. Hay que revisar si las entidades, relaciones y comunidades usadas son correctas. Un resumen global puede ser fluido y aun así esconder que una comunidad importante no entró en el mapa.

Casos cercanos

Secretaría académica. Preguntan: “¿Puedo ampliar matrícula si tengo una beca pendiente y pagos vencidos?”. Un RAG básico quizá encuentra solo la normativa de ampliación. Query decomposition separa “ampliación”, “beca pendiente” y “pagos vencidos”. El router decide si mirar normativa, becas y calendario. La respuesta final debe citar cada pieza.

Equipo de soporte técnico. Preguntan: “¿Qué problemas se repiten desde la última versión?”. No quieres una respuesta sobre un ticket concreto. Quieres agrupar incidencias por producto, síntoma, versión y solución. Aquí GraphRAG global o resúmenes jerárquicos pueden mostrar patrones que un top-k no ve.

Documentación de ingeniería. Preguntan: “¿Cómo configuro el conector si uso PostgreSQL y despliegue local?”. El sistema puede necesitar buscar en documentación, consultar ejemplos de configuración y revisar límites de versión. Agentic RAG con herramientas de búsqueda acotadas puede ser útil, siempre que cite y limite dominios.

Compliance documental. Preguntan: “¿Qué políticas dependen de este requisito?”. GraphRAG local encaja porque lo importante es el vecindario de una entidad: requisito, políticas, controles, evidencias y documentos fuente.

Producto con datos vivos. Preguntan: “¿Qué clientes están afectados y qué documento explica la política?”. Aquí no basta RAG. Necesitas tool o SQL para estado vivo y RAG para explicar la política. Este puente prepara el [capítulo 12](#).

Para entenderlo sin perderse

Una forma sencilla de explicarlo: un RAG básico se parece a preguntar a alguien que tiene una estantería y te trae tres páginas parecidas a tu pregunta. Si esas páginas contienen la respuesta, perfecto. Si la pregunta exige comparar, comprobar vigencia, mirar otra fuente o seguir una relación entre documentos, la persona necesita una libreta de trabajo: primero mira dónde buscar, luego consulta, después comprueba si lo encontrado basta y solo entonces responde.

En esa metáfora:

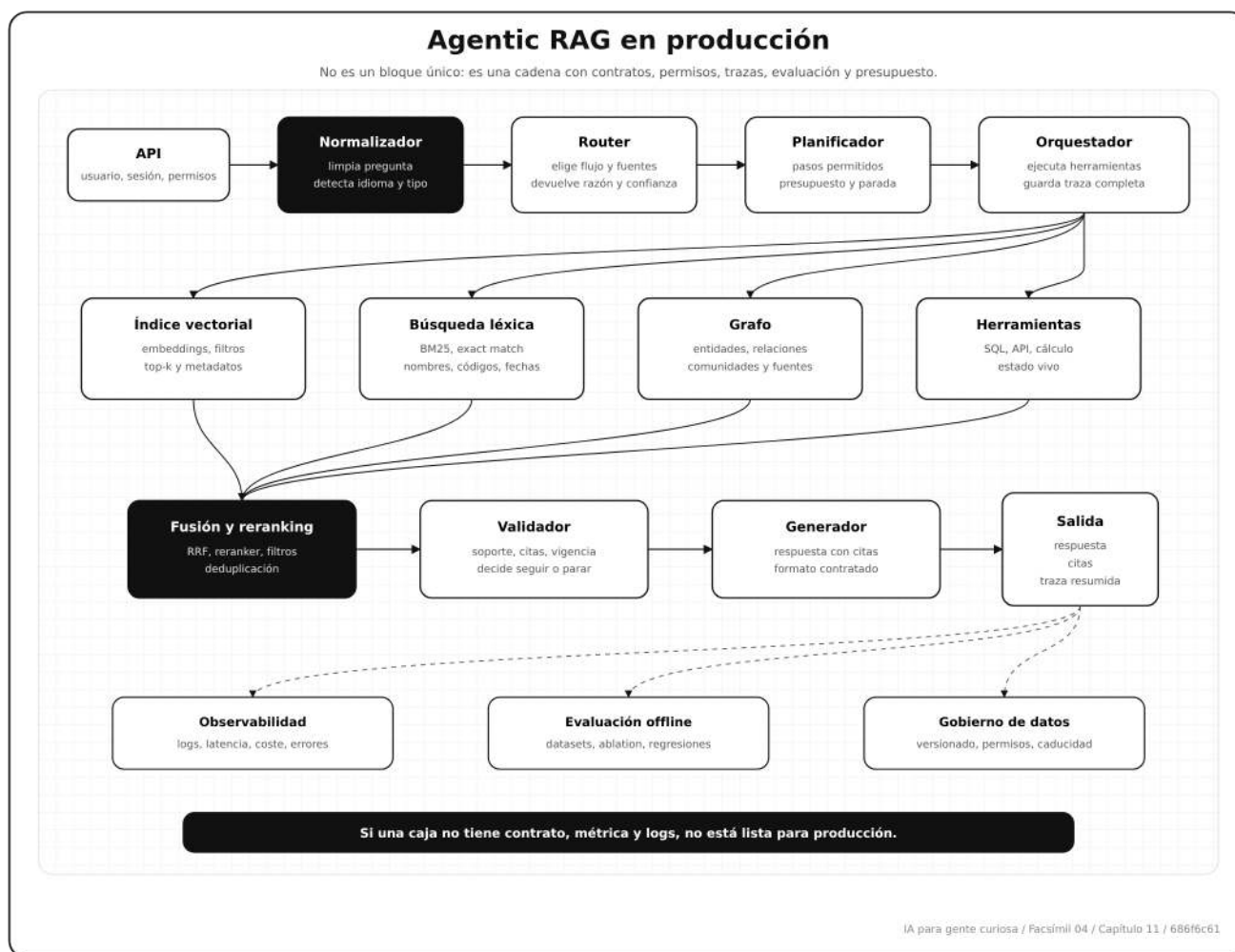
PIEZA	TRADUCCIÓN MENTAL	QUÉ DEBERÍA QUEDAR CLARO
RAG básico	Traer páginas parecidas.	Sirve si la pregunta vive en uno o varios fragmentos cercanos.
Router	Decidir en qué estantería mirar.	No es una caja negra: clasifica la pregunta y elige fuente.
Multi-query	Preguntar lo mismo con varias palabras.	Mejora cobertura cuando el vocabulario cambia.
Descomposición	Separar una pregunta grande en preguntas pequeñas.	Ayuda si hay que comparar o juntar varias condiciones.
Validador	Comprobar si las páginas sostienen la respuesta.	Reduce respuestas bonitas pero poco justificadas.
Traza	Libreta de lo que hizo el sistema.	Permite depurar, auditar y evaluar.
Grafo	Mapa de cosas conectadas.	Ayuda cuando la relación importa tanto como el texto.
Comunidad	Barrio dentro del grafo.	Grupo de entidades que aparecen conectadas muchas veces.

El error habitual es imaginar que Agentic RAG “razona más”. En producción me interesa una definición menos romántica: **Agentic RAG toma decisiones explícitas entre pasos permitidos y deja evidencia de esas decisiones**. Si no hay pasos permitidos, presupuesto y traza, no tienes una arquitectura: tienes una conversación difícil de depurar.

Cómo lo montaría en un sistema real

Si tuviera que construir esto en una empresa, no empezaría por GraphRAG ni por un agente completo. Empezaría por una pregunta incómoda: “¿qué fallo real quiero corregir?”. Si el fallo es vocabulario, multi-query. Si el fallo es escoger mal la fuente, router. Si el fallo es no saber si la evidencia basta, validador. Si el fallo es entender relaciones entre documentos, grafo.

La arquitectura mínima sería tendría estas capas:



Cada caja de la figura debería poder probarse por separado. Un ingeniero no debería preguntar “¿funciona el agente?”, sino cosas más concretas:

- ¿El router elige bien entre normativa, FAQ, SQL, tickets y grafo?
- ¿El retriever trae evidencia suficiente con filtros de permisos?
- ¿La fusión elimina duplicados y mantiene fuentes distintas?
- ¿El validador detecta falta de soporte, citas flojas o documentos caducados?
- ¿El generador respeta el formato de salida y no inventa campos?
- ¿La traza permite reproducir la respuesta cinco días después?

Herramientas y orquestadores que ya existen

No tienes que escribir el bucle del agente ni el pipeline del grafo a mano. Hay orquestadores maduros, y conocerlos evita reinventar lo difícil (el control de flujo, las trazas, los reintentos). La elección depende de cuánto control quieras y de si tu problema es agentic, de grafo o de pipeline.

HERRAMIENTA	PARA QUÉ	CUÁNDO ELEGIRLA
LangGraph	Modela el agente como un grafo de estados explícito (nodos, aristas, ciclos con condición).	Quieres control fino del bucle, los reintentos y la parada, con trazas.
LlamaIndex (Workflows / agents)	Workflows por eventos y agentes de RAG sobre tus índices.	Ya indexas con LlamaIndex y quieres pasos agentic encima.
Microsoft GraphRAG	Pipeline completo de GraphRAG: extracción de entidades, comunidades (Leiden) y resúmenes.	Vas a montar GraphRAG y no quieres reimplementar la indexación del grafo.
DSPy	Programa el sistema como módulos y optimiza sus prompts contra una métrica.	Quieres dejar de ajustar prompts a mano y optimizar contra tu eval.
Haystack	Pipelines de RAG componibles (retrievers, rankers, generadores) listos para producción.	Quieres un pipeline modular y estable más que un agente libre.

La regla, otra vez: la herramienta da el andamiaje, pero el contrato de cada pieza (entrada, salida, errores, timeout, permisos, trazas) y el presupuesto los pones tú. Un orquestador no te exime de saber qué hace cada paso y por qué.

Contratos de herramientas

Una herramienta en un sistema agentic no debería ser “una función que el modelo puede llamar”. Debe ser un contrato. El contrato dice qué recibe, qué devuelve, qué permisos exige, cuánto tarda, cómo falla y cómo se audita.

Ejemplo de contrato de una herramienta de búsqueda documental:

```
{
  "name": "buscar_normativa",
  "input": {
    "query": "string",
    "filters": {
      "curso": "2026",
      "estado": "vigente"
    },
    "top_k": 8
  },
  "output": {
    "results": [
      {
        "chunk_id": "norm-2026#art-14",
        "score": 0.84,
        "source": "Normativa 2026",
        "valid_from": "2026-01-01",
        "text": "fragmento recuperado"
      }
    ]
  },
  "errors": [
    "permission_denied",
    "source_unavailable",
    "low_recall"
  ],
  "timeout_ms": 1200,
  "audit": true
}
```

Lo importante no es el JSON, sino la disciplina:

CAMPO	QUÉ APORTA	QUÉ SE ROMPE SI FALTA
name	Identidad estable de la herramienta.	No puedes comparar ejecuciones ni versionar cambios.
input	Qué puede pedir el sistema.	El modelo puede mandar consultas vagas o imposibles.
filters	Permisos, vigencia, idioma, cliente o corpus.	Puedes mezclar documentos que no deberían mezclarse.
top_k	Límite de resultados.	Coste y contexto crecen sin control.
score	Señal de recuperación.	No sabes por qué entró un fragmento.
source	Procedencia legible.	No puedes citar ni revisar.
valid_from	Vigencia temporal.	Respuestas correctas pueden quedar obsoletas.
errors	Fallos esperados.	El sistema trata un fallo como si fuera ausencia de información.
timeout_ms	Límite de latencia.	Una herramienta lenta secuestra toda la respuesta.
audit	Obligación de guardar traza.	No puedes reproducir ni explicar decisiones.

Un contrato de herramienta también debe declarar qué hacer cuando no basta:

RESULTADO	DECISIÓN CORRECTA
permission_denied	No buscar atajos; responder que no hay permiso suficiente.
source_unavailable	Degradar con aviso o pedir reintento, según criticidad.
low_recall	Reescribir consulta, usar búsqueda híbrida o abstenerse.
empty_result	Distinguir “no existe” de “no he podido encontrarlo”.
conflicting_sources	Priorizar vigencia, autoridad y citar el conflicto.

GraphRAG por dentro

GraphRAG no empieza en la query; empieza en la ingesta. Antes de responder hay que convertir documentos en entidades, relaciones, comunidades y resúmenes. Ese proceso tiene mucha ingeniería escondida.

Una tubería razonable:

PASO	QUÉ HACE	RIESGO PRINCIPAL	CONTROL
Ingesta	Lee PDFs, HTML, Markdown, tickets, tablas o transcripciones.	Perder estructura del documento.	Guardar fuente, página, sección y fecha.
Chunking	Parte documentos en unidades recuperables.	Cortar relaciones importantes.	Chunks con solape y metadatos ricos.
Extracción	Detecta entidades y relaciones.	Extraer nombres distintos para lo mismo.	Diccionario, revisión y normalización.
Canonicalización	Une variantes de una entidad.	Mezclar entidades parecidas pero distintas.	Alias, reglas y confianza.
Aristas	Crea relaciones con fuente.	Relación sin prueba documental.	Toda arista guarda chunk, frase y score.
Comunidades	Agrupar zonas densas del grafo.	Comunidades demasiado grandes o pequeñas.	Medir modularidad y revisar muestras.
Resúmenes	Resume comunidades.	Perder excepciones importantes.	Citar nodos y documentos representativos.
Índices	Indexa texto, nodos, aristas y resúmenes.	Recuperar solo una vista parcial.	Búsqueda híbrida y evaluación por tipo.
Actualización	Reprocesa cambios del corpus.	Grafo viejo con documentos nuevos.	Versionado, diffs y caducidad.

Se puede escribir de forma compacta:

$$V = \text{canon}(\text{entidades}(D))$$

$$E = \{(v_i, r, v_j, \text{fuente}, \text{confianza})\}$$

$$R_c = \text{resumen}(C_c, \text{fuentes}_c)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
D	Documentos de entrada.	Normativa, FAQ, tickets y manuales.
V	Entidades normalizadas.	“2FA” y “doble factor” como la misma entidad.
E	Relaciones con fuente.	“ampliación requiere no tener pagos vencidos”.
C_c	Comunidad del grafo.	Trámites de matrícula y pagos.
R_c	Resumen de comunidad.	“Los bloqueos se concentran en pagos y plazos”.

La parte difícil no es dibujar nodos. La parte difícil es saber si el nodo es correcto, si dos nodos son la misma cosa, si la relación tiene fuente, si el resumen no borra excepciones y si el grafo se actualiza cuando cambia el corpus.

Comunidades y consulta: local frente a global

¿De dónde salen esos «resúmenes de comunidad»? Una vez construido el grafo, GraphRAG detecta **comunidades**: grupos de entidades más conectadas entre sí que con el resto, usando un algoritmo de detección de comunidades. El estándar es **Leiden**, que mejora al clásico Louvain garantizando que las comunidades queden bien conectadas.¹⁶ Luego un LLM resume cada comunidad. Eso permite responder preguntas que ningún chunk suelto contiene, porque la respuesta vive en la *estructura*, no en el texto.

Esa construcción habilita dos modos de consulta que conviene no confundir:

MOD O	QUÉ HACE	PREGUNTA TÍPICA
Local	Parte de una entidad concreta y recorre sus vecinos en el grafo.	«¿Qué requisitos dependen de la beca?»
Glob al	Agrega los resúmenes de comunidad para responder sobre el corpus entero.	«¿Qué patrones de bloqueo se repiten en toda la normativa?»

La regla de ingeniería: el modo **local** es para preguntas sobre relaciones de una entidad; el **global**, para preguntas de síntesis sobre todo el corpus, que es justo donde un RAG por chunks fracasa porque la respuesta no está escrita en ningún sitio, hay que componerla.

Coste, latencia y presupuesto

Un RAG avanzado puede mejorar respuestas y empeorar producto si duplica latencia sin medirlo. Por eso hay que convertir la arquitectura en números.

Si generas q consultas, traes k fragmentos por consulta y cada fragmento tiene L tokens de media:

$$N_{\text{ctx}} \approx q \cdot k \cdot L$$

Si además usas reranker y un generador:

$$T_{\text{total}} = T_{\text{router}} + \max_i(T_{\text{retrieval},i}) + T_{\text{rerank}} + T_{\text{validación}} + T_{\text{generación}}$$

Y el coste de una respuesta puede aproximarse así:

$$C_{\text{respuesta}} = C_{\text{emb}} + C_{\text{retrieval}} + C_{\text{rerank}} + C_{\text{LLM}}$$

NÚMERO	QUÉ SIGNIFICA	QUÉ MIRAR EN PRODUCCIÓN
q	Número de consultas generadas.	Si sube, suben recall, coste y ruido.
k	Fragmentos recuperados por consulta.	Un top-k alto puede tapar la evidencia buena.
L	Tokens por fragmento.	Chunks largos llenan contexto rápido.
T_{total}	Latencia total.	Medir P50, P95 y timeouts, no solo media.
$C_{\text{respuesta}}$	Coste por respuesta.	Dividir por respuesta útil, no por llamada.
C_{index}	Coste de indexar.	En GraphRAG puede ser alto antes de la primera query.

Para GraphRAG hay otro coste:

$$C_{\text{graph-index}} = C_{\text{extracción}} + C_{\text{normalización}} + C_{\text{comunidades}} + C_{\text{resúmenes}}$$

Donde $C_{\text{extracción}}$ es el coste de que un LLM extraiga entidades y relaciones de cada documento, $C_{\text{normalización}}$ el de unificar entidades duplicadas, $C_{\text{comunidades}}$ el de detectarlas (Leiden) y $C_{\text{resúmenes}}$ el de resumir cada una con un LLM. Este coste se paga al construir o actualizar el índice, no solo al responder. Por eso GraphRAG puede ser brillante en colecciones estables y caro en corpus que cambian cada hora.

Pongámoslo en números para que las cuentas dejen de ser abstractas. Una consulta agentic que genera $q = 3$ subconsultas, trae $k = 5$ fragmentos por consulta de $L = 300$ tokens cada uno, con reranker, validación y un generador:

COMPONENTE	CUENTA	RESULTADO
Contexto	$N_{\text{ctx}} = 3 \cdot 5 \cdot 300$	4 500 tokens
Latencia	0,3 + 0,4 + 0,5 + 0,3 + 1,5 (router, retrieval, rerank, validación, generación)	3,0 s
Coste	embeddings + 4 700 tokens de entrada + 400 de salida, a tarifas ilustrativas	$\approx 0,013$ por respuesta

Compáralo con un RAG fijo del capítulo 9 (una consulta, sin rerank ni validación): la versión agentic multiplica por tres el contexto y la latencia, y dobla el coste. Esa es la factura de complicar, y solo se justifica si la calidad sube lo suficiente como para pagarla. La cuenta, no la intuición, es lo que te dice si el peldaño merece la pena.

Los fallos propios de un agente

Dar autonomía a un sistema añade modos de fallo que un RAG fijo no tiene, y que un ingeniero debe acotar **por diseño**, no esperar a que aparezcan en producción:

FALLO	QUÉ OCURRE	CÓMO SE ACOTA
Bucle infinito	El agente repite pasos sin converger.	Tope de pasos T_{max} y condición de parada explícita.
Deriva del objetivo	A los pocos pasos responde a una pregunta distinta de la original.	Re-ancorar la pregunta original x en cada paso del estado.
Sobre-recuperación	Trae más y más contexto «por si acaso».	Presupuesto de contexto y de coste por respuesta.
Atasco sin evidencia	No encuentra nada útil y aun así insiste.	Abstención cuando el soporte no supera el umbral.
Coste descontrolado	Cada reintento multiplica llamadas al LLM.	<i>Circuit breaker</i> : corta y degrada a una ruta simple.

La regla que resume todo el capítulo: **un agente sin límites es una factura sin límites**. La autonomía es útil solo cuando va acompañada de un presupuesto (T_{max} , coste, tiempo) y de una salida de emergencia que devuelva el control a una ruta simple cuando algo se

descontrola.

Evaluación para ingeniería

La evaluación de RAG avanzado tiene que separar piezas. Si solo miras la respuesta final, no sabes si mejoró el retriever, el router, el grafo o simplemente hubo suerte en una muestra.

Un plan de evaluación serio:

PRUEBA	QUÉ COMPARA	PREGUNTA QUE RESPONDE
Baseline	RAG básico contra sistema nuevo.	¿Complicar mejora algo medible?
Ablation	Quitar una pieza cada vez.	¿Qué aporta router, multi-query, grafo o validador?
Router accuracy	Ruta esperada contra ruta elegida.	¿Va a la fuente correcta?
Recall@k	Evidencia esperada dentro del top-k.	¿Recupera lo necesario?
MRR / nDCG	Orden de documentos relevantes.	¿Lo bueno aparece arriba?
Node precision	Nodos correctos del grafo.	¿Las entidades recuperadas son válidas?
Edge precision	Relaciones correctas del grafo.	¿Las aristas están sostenidas por fuentes?
Citation support	Claims con cita suficiente.	¿Cada afirmación importante está soportada?
Abstention rate	Casos donde no responde.	¿Se abstiene cuando falta evidencia?
Latencia P95	Tiempo para el 95% de consultas.	¿El producto aguanta en uso real?
Coste por respuesta útil	Coste dividido por respuestas aceptables.	¿La mejora compensa?

El ablation test es especialmente sano:

VARIANTE	QUÉ ACTIVA	QUÉ ESPERAS APRENDER
A	RAG básico.	Línea base.
B	A + búsqueda híbrida.	Si el problema era vocabulario exacto.
C	B + multi-query.	Si faltaba cobertura semántica.
D	C + router.	Si elegir fuente aporta mejora.
E	D + validador.	Si reduce respuestas sin soporte.
F	E + GraphRAG.	Si las relaciones añaden valor real.

Si F gana solo un 1% pero dobla latencia y coste, quizá no merece producción. Si F gana mucho en preguntas relacionales y pierde en preguntas simples, el router debe activar GraphRAG solo cuando toque.

Caso completo de diseño

Supongamos un sistema con 50.000 documentos internos: normativa, manuales, tickets, FAQs, actas y algunas tablas con estado vivo. El objetivo es responder a equipos internos con citas, permisos y trazas.

Yo lo diseñaría por fases:

FASE	QUÉ HARÍA	CRITERIO PARA AVANZAR
1. Corpus	Inventario de fuentes, permisos, vigencia y formatos.	Saber qué puede ver cada usuario y qué fuente manda.
2. Baseline	RAG básico con chunking, embeddings y búsqueda híbrida.	Dataset de 100-300 preguntas reales con respuestas esperadas.
3. Evaluación	Medir recall@k, soporte de citas, abstención y latencia.	Detectar el fallo dominante.
4. Router	Separar normativa, FAQ, tickets, manuales, SQL y grafo.	Accuracy de ruta suficiente y trazas legibles.
5. Validación	Añadir groundedness, vigencia y conflicto entre fuentes.	Menos respuestas sin soporte.
6. Grafo local	Entidades y relaciones para normativa y dependencias.	Mejora clara en preguntas relacionales.
7. GraphRAG global	Comunidades para tickets, actas e incidencias.	Mejora clara en preguntas de patrones.
8. Producción	Observabilidad, costes, permisos, caché y regresiones.	Alertas y evaluación continua antes de ampliar uso.

La decisión final no sería “usar GraphRAG sí o no”. Sería una política:

TIPO DE PREGUNTA	FLUJO RECOMENDADO
“¿Dónde dice X?”	RAG básico con búsqueda híbrida y citas.
“¿Cuál es el estado actual de X?”	Tool o SQL, después explicación con RAG.
“Compara X e Y”	Descomposición, router y validación.
“¿Qué depende de este requisito?”	GraphRAG local.
“¿Qué patrones aparecen en todo el corpus?”	GraphRAG global.
“No hay evidencia suficiente”	Abstención con fuente necesaria.

Este diseño también ayuda a una persona curiosa: no hay una herramienta universal. Hay una escalera. Cada peldaño se sube cuando el anterior falla de una forma concreta.

En el día a día

En el día a día, lo agentic y GraphRAG aparecen cuando una pregunta deja de tener una sola respuesta en un solo fragmento. «¿Puedo ampliar matrícula si tengo una beca pendiente y un pago vencido?» no se resuelve con un chunk: hay que recuperar dos normas, ver cómo se relacionan y componer. Ahí un router que descompone la pregunta, o un grafo que conoce qué requisito bloquea a cuál, gana al RAG fijo. Pero la mayoría de las preguntas de un día normal («¿cuándo abre la ampliación?») se resuelven con el RAG básico del capítulo 9, y meterles un agente solo añade latencia, coste y formas nuevas de fallar.

La parte delicada, y el corazón de este capítulo, es saber **cuándo no complicar**. Cada peldaño de la escalera (router, multi-query, agente con bucle, GraphRAG) se sube cuando el anterior falla de una forma concreta y medible, no porque suene moderno. Un equipo que monta GraphRAG sin tener preguntas que dependan de relaciones acaba manteniendo una catedral para resolver lo que un RAG simple ya hacía.

Por qué debería importarte

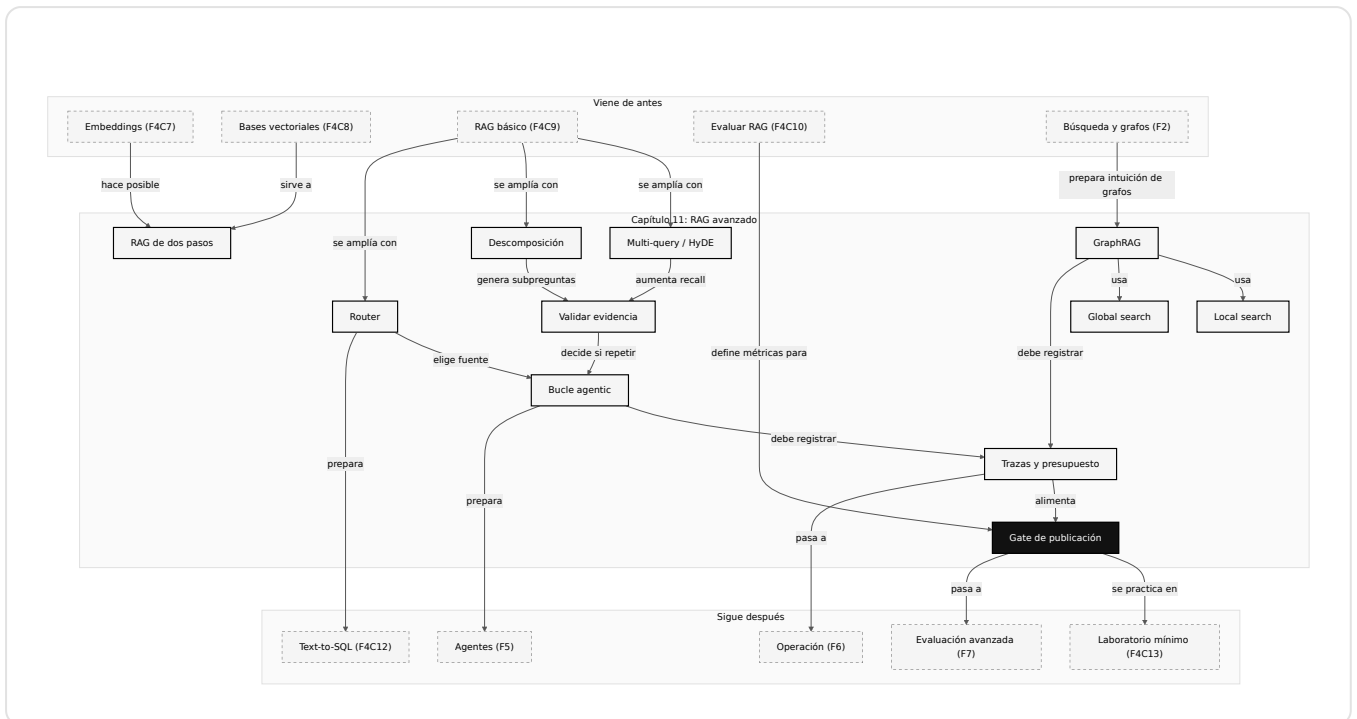
Porque la complejidad mal puesta es la forma más cara de empeorar un sistema. Un agente con bucles sin presupuesto es una factura impredecible; un grafo sin relaciones útiles es mantenimiento permanente sin mejora; una arquitectura con cinco piezas nuevas a la vez es imposible de depurar cuando falla. Saber cuándo escalar, y cuándo resistirse, es lo que separa a un ingeniero de alguien que colecciona arquitecturas.

Y porque cada peldaño añade superficie de fallo que hay que medir. Lo agentic no se evalúa solo por la respuesta final: hay que evaluar el router, las subpreguntas, los nodos y aristas del grafo, los pasos del bucle y el coste por respuesta útil. Si añades complejidad sin añadir medición, no has mejorado el sistema: solo has perdido la capacidad de saber si funciona.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Llamar agente a cualquier if	Un router determinista puede ser suficiente; no todo flujo condicional es un agente.	Nombrar la pieza exacta: router, validador, descomposición o bucle.
Añadir bucles sin presupuesto	La latencia y el coste se vuelven impredecibles.	Definir $T_{\max}$, coste máximo y condición de parada.
Usar GraphRAG sin relaciones útiles	Si no hay entidades ni vínculos relevantes, el grafo añade mantenimiento sin mejorar respuestas.	Probar primero si las preguntas fallan por relaciones, no por retrieval básico.
No guardar la ruta de decisión	Si falla, no sabes qué fuente eligió ni por qué.	Guardar traza: ruta, consultas, resultados, scores, citas y validaciones.
Medir solo groundedness final	Puede responder bien por casualidad aunque haya elegido mal el camino.	Evaluar router, subpreguntas, nodos, aristas, pasos y respuesta.
Mezclar todas las mejoras a la vez	No puedes atribuir la mejora ni depurar el fallo.	Añadir una pieza por experimento y comparar contra baseline.
No definir contratos de herramientas	Cada integración acaba devolviendo lo que quiere y fallando de forma distinta.	Declarar input, output, errores, timeout, permisos y trazas.
Ignorar el coste de indexar GraphRAG	El grafo también cuesta antes de contestar la primera pregunta.	Separar coste de indexación y coste por query.
Mirar solo la media de latencia	La media puede ocultar consultas lentas que rompen la experiencia.	Medir P50, P95, timeouts y coste por respuesta útil.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Agentic RAG	RAG donde un modelo o router decide pasos de búsqueda, herramientas y validación antes de responder.
RAG de dos pasos	Flujo fijo: recuperar contexto y generar respuesta.
Multi-query retrieval	Varias consultas para cubrir vocabularios distintos de una misma necesidad.
HyDE	Generar un documento hipotético para buscar documentos reales parecidos.
Query decomposition	Dividir una pregunta grande en subpreguntas recuperables.
Router	Pieza que decide qué corpus, índice, herramienta o flujo usar.
Corrective RAG	RAG que evalúa si lo recuperado basta y corrige si no basta.
Self-RAG	Enfoque donde la recuperación y la crítica forman parte del comportamiento del modelo.
RAPTOR	Recuperación con árbol de resúmenes a distintos niveles de abstracción.
GraphRAG	RAG basado en grafo de entidades, relaciones y resúmenes.
Local search	Búsqueda centrada en entidades o relaciones concretas del grafo.
Global search	Búsqueda sobre resúmenes de comunidades para preguntas del corpus completo.
Community summary	Resumen de un grupo de nodos relacionados dentro del grafo.
Presupuesto de pasos	Límite de acciones, llamadas, coste o latencia permitido.
Contrato de herramienta	Especificación de entradas, salidas, errores, permisos, timeout y auditoría de una herramienta.
Traza	Registro reproducible de ruta, consultas, resultados, decisiones, costes y citas usadas.
Canonicalización	Unión controlada de variantes que representan la misma entidad.
Ablation test	Prueba donde se quita una pieza del sistema para medir qué aporta realmente.

TÉRMINO	DEFINICIÓN
Latencia P95	Tiempo por debajo del cual responde el 95% de las consultas.
Edge precision	Proporción de relaciones del grafo que son correctas y están sostenidas por fuentes.

Antes de pasar página

- ☐ ¿Puedo explicar cuándo basta un RAG de dos pasos? (Si no, vuelve a «Qué sí es Agentic RAG».)
- ☐ ¿Sé distinguir multi-query, HyDE y query decomposition? (Si no, vuelve a «Arquitecturas, una por una».)
- ☐ ¿Puedo explicar qué hace un router y qué debe registrar? (Si no, vuelve a «Arquitecturas, una por una».)
- ☐ ¿Sé por qué Corrective RAG y Self-RAG no son “más complejidad por presumir”, sino validación y decisión de recuperación? (Si no, vuelve a «Arquitecturas, una por una».)
- ☐ ¿Puedo escribir qué es un grafo $G = (V, E)$ y qué significa una arista con fuente? (Si no, vuelve a «GraphRAG: qué cambia cuando aparece un grafo».)
- ☐ ¿Sé cuándo GraphRAG local encaja mejor que GraphRAG global? (Si no, vuelve a «GraphRAG: qué cambia cuando aparece un grafo».)
- ☐ ¿Puedo explicar cómo se construye GraphRAG: entidades, relaciones, comunidades y resúmenes? (Si no, vuelve a «GraphRAG por dentro».)
- ☐ ¿Sé diseñar un contrato mínimo para una herramienta de búsqueda? (Si no, vuelve a «Contratos de herramientas».)
- ☐ ¿Puedo definir un presupuesto de pasos, coste y latencia para un Agentic RAG? (Si no, vuelve a «Coste, latencia y presupuesto».)
- ☐ ¿Sé calcular por qué multi-query aumenta contexto, coste y ruido? (Si no, vuelve a «Coste, latencia y presupuesto».)
- ☐ ¿Sé qué métricas miraría además de la respuesta final? (Si no, vuelve a «Evaluación para ingeniería».)
- ☐ ¿Puedo plantear un ablation test para saber qué pieza aporta mejora? (Si no, vuelve a «Evaluación para ingeniería».)
- ☐ ¿Sé qué mirar en la traza de cada pregunta para saber si el sistema acertó? (Si no, repasa la sección correspondiente.)

En resumen

IDEA FUERZA	DETALLE
No empieces por Agentic RAG.	Empieza por RAG básico evaluado; complica solo cuando sabes qué falla.
Agentic RAG decide pasos.	Puede reescribir, dividir, enrutar, consultar herramientas, validar y volver a buscar.
GraphRAG usa relaciones.	Sirve cuando importan entidades, dependencias, comunidades o preguntas globales del corpus.
Producción exige contratos.	Cada herramienta necesita entradas, salidas, errores, permisos, timeout y trazas.
GraphRAG cuesta antes de responder.	Extraer entidades, normalizar, crear comunidades y resumir también forman parte del presupuesto.
Cada pieza nueva exige evaluación propia.	Router, descomposición, grafo, bucle y respuesta final se miden por separado.
El ablation test evita autoengaños.	Compara RAG básico contra mejoras incrementales para saber qué aporta cada pieza.
El coste no es solo dinero.	También pagas latencia, trazabilidad, mantenimiento, permisos y complejidad de depuración.

Para saber más

Asai, A., Wu, Z., Wang, Y., Sil, A. y Hajishirzi, H. (2023). *Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection*. [arXiv](#)

Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods. *Proceedings of SIGIR*, 758-759. [DOI](#)

Edge, D. y otros (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. [arXiv](#)

Gao, L., Ma, X., Lin, J. y Callan, J. (2022). *Precise Zero-Shot Dense Retrieval without Relevance Labels*. [arXiv](#)

LangChain. (2026). *Retrieval*. [Documentación oficial](#)

Lewis, P. y otros (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474. [NeurIPS](#)

LlamaIndex. (2026). *Agentic strategies*. [Documentación oficial](#)

Microsoft. (2026). *GraphRAG Query Engine overview*. [Documentación oficial](#)

OpenAI. (2026). *Graders*. [Documentación oficial](#)

Sarathi, P. y otros (2024). *RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval*. [arXiv](#)

Schick, T. y otros (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools*. [DOI](#)

Yan, S.-Q., Gu, J.-C., Zhu, Y. y Ling, Z.-H. (2024). *Corrective Retrieval Augmented Generation*. [arXiv](#)

Yao, S. y otros (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. *International Conference on Learning Representations*. [arXiv](#)

Notas

1. Lewis, P. y otros (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. *Advances in Neural Information Processing Systems 33*, 9459-9474. [NeurIPS](#). ↵
2. LangChain. (2026). *Retrieval*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. La página compara 2-step RAG, Agentic RAG e Hybrid RAG, y explica que un agente puede decidir cuándo y cómo recuperar mediante herramientas. ↵
3. LlamaIndex. (2026). *Agentic strategies*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
4. Yao, S. y otros (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. *International Conference on Learning Representations*. [arXiv](#). ↵
5. Schick, T. y otros (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools*. [DOI](#). ↵
6. Gao, L., Ma, X., Lin, J. y Callan, J. (2022). *Precise Zero-Shot Dense Retrieval without Relevance Labels*. [arXiv](#). Asai, A., Wu, Z., Wang, Y., Sil, A. y Hajishirzi, H. (2023). *Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection*. [arXiv](#). Yan, S.-Q., Gu, J.-C., Zhu, Y. y Ling, Z.-H. (2024). *Corrective Retrieval Augmented Generation*. [arXiv](#). Jeong, S., Baek, J., Cho, S., Hwang, S. J. y Park, J. C. (2024). *Adaptive-RAG: Learning to Adapt Retrieval-Augmented Large Language Models through Question Complexity*. [arXiv](#). ↵
7. Edge, D. y otros (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. [arXiv](#). ↵

8. Microsoft. (2026). *GraphRAG Query Engine overview*. Documentación oficial. Consultado el 26 de mayo de 2026. ↵
9. Shunyu Yao y otros (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR. <https://arxiv.org/abs/2210.03629>. La formulación de estado-acción-transición es la de los procesos de decisión secuencial: Richard S. Sutton y Andrew G. Barto. (2018). *Reinforcement Learning: An Introduction* (2.^a ed.). MIT Press. ↵
- .0. Noah Shinn y otros (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. NeurIPS. <https://arxiv.org/abs/2303.11366>. ↵
- .1. Shunyu Yao y otros (2023). *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. NeurIPS. <https://arxiv.org/abs/2305.10601>. ↵
- .2. Akari Asai y otros (2023). *Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection*. ICLR 2024. <https://arxiv.org/abs/2310.11511>. ↵
- .3. Shi-Qi Yan y otros (2024). *Corrective Retrieval Augmented Generation*. <https://arxiv.org/abs/2401.15884>. ↵
- .4. Darren Edge y otros (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. <https://arxiv.org/abs/2404.16130>. ↵
- .5. Microsoft, 2026. ↵
- .6. Vincent A. Traag, Ludo Waltman y Nees Jan van Eck. (2019). *From Louvain to Leiden: guaranteeing well-connected communities*. Scientific Reports, 9, 5233. <https://doi.org/10.1038/s41598-019-41695-z>. ↵

Capítulo 12: Text-to-SQL y herramientas de datos

La pregunta que no está en un documento

En el [capítulo 11](#) vimos que un sistema puede decidir consultar distintas fuentes antes de responder. Algunas fuentes son textos: normativas, manuales, tickets, actas. Otras no son textos en sentido estricto: tablas, bases de datos, métricas, historiales, estados de pedidos, matrículas, pagos, sensores o registros de producto.

Ahí aparece una frontera importante. Un RAG puede explicar “qué dice la política de matrícula”. Pero si preguntas “¿cuántos alumnos tienen pago pendiente y beca en revisión?”, no quieres que el modelo imagine un número a partir de párrafos. Quieres que consulte datos.

Text-to-SQL nace de esa necesidad: convertir una pregunta humana en una consulta SQL controlada. No es “hablar con la base de datos como si fuera una persona”. Es construir un puente auditado entre intención, esquema, permisos, consulta, ejecución, resultado y explicación.

Estado del arte con fecha de corte

Fecha de corte: 26 de mayo de 2026.

Fuentes consultadas ese día: papers de Spider, RAT-SQL, PICARD, BIRD y Spider 2.0; documentación oficial de LangChain SQL Agent, LlamaIndex NL SQL, OpenAI Function Calling y Structured Outputs; documentación de SQLGlot, DuckDB y repositorio público de Vanna.

Text-to-SQL se estudia desde hace años como una tarea de *semantic parsing*: traducir lenguaje natural a una representación formal ejecutable. Spider marcó un salto porque propuso preguntas y consultas complejas sobre 200 bases de datos, con esquemas distintos entre entrenamiento y prueba; su objetivo era medir generalización a bases nuevas, no memorizar una sola tabla.¹

Después se vio que una parte crítica no era solo generar SQL, sino entender el esquema. RAT-SQL trabajó explícitamente el *schema linking*: conectar palabras de la pregunta con tablas, columnas y relaciones del esquema mediante atención consciente de relaciones.² PICARD atacó otro problema: incluso un modelo capaz puede producir SQL inválido. Su propuesta restringe la decodificación con parsing incremental para rechazar tokens que romperían la sintaxis formal.³

Los benchmarks más recientes empujaron la tarea hacia condiciones más reales. BIRD introdujo bases de datos grandes, valores sucios, conocimiento externo y eficiencia de consulta; reportó 12.751 pares pregunta-SQL sobre 95 bases de datos y 33,4 GB.⁴ Spider 2.0 fue más allá: problemas de flujo empresarial, más de 1.000 columnas en algunas bases, varios dialectos como BigQuery y Snowflake, metadatos extensos y tareas que pueden requerir múltiples consultas.⁵

En herramientas prácticas, LangChain documenta un flujo de SQL agent que lista tablas, decide cuáles son relevantes, inspecciona esquemas, genera consulta, revisa errores comunes, ejecuta y formula respuesta.⁶ LlamaIndex ofrece componentes NL SQL y advierte que ejecutar consultas generadas requiere especial cuidado con permisos y entorno.⁷ OpenAI documenta function calling como forma de describir herramientas mediante esquemas y recibir argumentos estructurados; Structured Outputs permite exigir que una salida siga un esquema JSON compatible.⁸

Qué no es Text-to-SQL

Text-to-SQL no es “darle acceso libre a la base de datos al modelo”. El modelo no debería decidir por su cuenta qué puede leer, cuántas filas puede sacar, qué tablas puede cruzar o si una consulta es aceptable. Eso lo decide el sistema.

Tampoco es un reemplazo completo de un equipo de datos. Muchas preguntas parecen simples y esconden definiciones de negocio: “cliente activo”, “ingreso neto”, “alumno matriculado”, “churn”, “pedido completado”. Si esas definiciones no viven en una capa semántica o en documentación recuperable, el modelo puede generar una consulta válida que responde a otra pregunta.

Y no es solo una tarea de SQL. En producción intervienen permisos, catálogo, metadatos, documentación de columnas, dialecto, coste, timeouts, límites de filas, validación, trazas y evaluación. La consulta es una pieza. El sistema es todo lo que evita que esa pieza se use mal.

Qué sí es

Text-to-SQL es una tubería controlada:

```
pregunta -> intención -> esquema relevante -> SQL candidato
          -> validación -> ejecución limitada -> resultado
          -> explicación con trazas
```

La unidad de trabajo no es “una query”. La unidad de trabajo es una **solicitud de datos** con contexto:

PIEZA	QUÉ CONTIENE	EJEMPLO
Pregunta	Lo que pide la persona.	“Ingresos por campus en marzo”.
Usuario	Identidad, rol y permisos.	analista_matricula , campus permitido.
Dominio	Área de datos.	Matrícula, becas, pagos, soporte.
Esquema	Tablas, columnas, claves y tipos.	pagos , alumnos , campus .
Semántica	Definiciones de negocio.	“ingreso neto = importe - devoluciones”.
SQL	Consulta candidata.	SELECT campus, SUM(...)
Validación	Reglas antes de ejecutar.	Solo SELECT , LIMIT , timeout.
Resultado	Filas devueltas.	Tabla agregada.
Explicación	Resumen humano.	“Campus Norte concentra el 42%”.
Traza	Registro completo.	Tablas usadas, SQL, coste, tiempo.

Para entenderlo bien, pensemos en una pregunta sencilla:

“¿Cuáles son los tres campus con más pagos pendientes?”

El sistema no debería saltar directamente a escribir SQL. Primero debe saber qué significa “pagos pendientes”, dónde vive “campus”, si la persona puede ver todos los campus, si hay que excluir pagos anulados, si la moneda importa y si “tres” implica ordenar de mayor a menor.

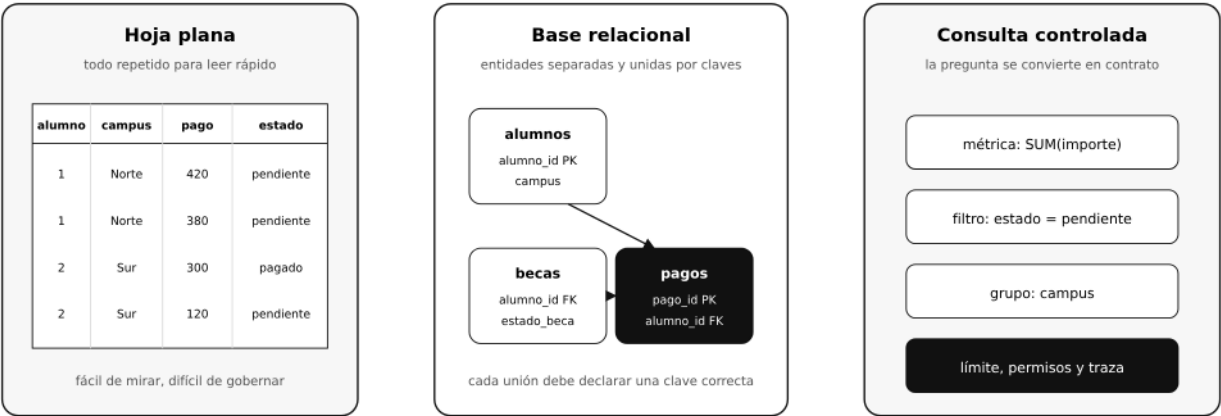
SQL desde cero, pero sin rebajarlo

SQL es el lenguaje clásico para consultar datos relacionales. Una base relacional organiza información en tablas. Cada tabla tiene filas y columnas. Una fila representa una entidad o evento; una columna representa un atributo.

Una base de datos no es un Excel grande. En una hoja plana es habitual repetir datos para que todo quepa en una misma vista. En una base relacional se separan entidades y eventos para que cada cosa tenga un lugar claro: alumnos por un lado, pagos por otro, becas por otro. Después una consulta une lo que necesita mediante claves.

SQL pregunta sobre relaciones, no sobre una hoja gigante

El modelo debe entender entidades, claves, filtros y métricas antes de escribir una consulta.



El modelo no debería “saber el número”: debería construir una consulta que otro componente pueda revisar.

IA para gente curiosa / Facsímil 04 / Capítulo 12 / 686f6c61

Ejemplo mínimo:

TABLA	PAGOS	SIGNIFICADO
	pago_id	identificador del pago.
	alumno_id	alumno asociado.
	campus	campus administrativo.
	estado	pagado , pendiente , devuelto .
	importe	cantidad.
	fecha	fecha del movimiento.

Una consulta básica:

```
SELECT campus, SUM(importe) AS total_pendiente
FROM pagos
WHERE estado = 'pendiente'
GROUP BY campus
ORDER BY total_pendiente DESC
LIMIT 3;
```

Esto se lee así:

CLÁUSULA	QUÉ HACE	EN CASTELLANO
SELECT	Elige columnas o cálculos.	Quiero campus y suma de importe.
FROM	Indica la tabla base.	Usa la tabla de pagos.
WHERE	Filtra filas antes de agrupar.	Solo pagos pendientes.
GROUP BY	Agrupar filas por una dimensión.	Junta pagos por campus.
ORDER BY	Ordena resultados.	Primero los importes mayores.
LIMIT	Limita filas devueltas.	Dame solo tres.

La parte que más se suele subestimar es el `JOIN` : cruzar tablas. Si `pagos` tiene `alumno_id` , pero la titulación vive en `alumnos` , necesitamos unir:

```
SELECT a.titulacion, SUM(p.importe) AS total_pendiente
FROM pagos AS p
JOIN alumnos AS a ON a.alumno_id = p.alumno_id
WHERE p.estado = 'pendiente'
GROUP BY a.titulacion
ORDER BY total_pendiente DESC
LIMIT 5;
```

Un `JOIN` no es decoración. Es una afirmación sobre relación entre tablas. Si la clave es incorrecta, el resultado puede ser válido en SQL y falso en negocio.

Errores SQL que cambian la historia

Para un ingeniero, el peligro no está solo en que el modelo genere SQL inválido. Ese fallo es fácil de detectar. El peligro serio es que genere SQL válido, rápido y aparentemente razonable, pero calcule otra cosa.

ERROR	SQL QUE SUELE APARECER	QUÉ ROMPE	CÓMO PENSARLO
Duplicar filas con un JOIN	Unir <code>alumnos</code> con varias filas de <code>pagos</code> .	<code>COUNT(*)</code> sube porque un alumno aparece varias veces.	Cuenta entidades con <code>COUNT(DISTINCT alumno_id)</code> .
Confundir evento y entidad	Contar pagos como si fueran alumnos.	Mide movimientos, no personas.	Pregunta si cada fila es “cosa” o “suceso”.
Ignorar NULL	<code>AVG(importe)</code> sin revisar ausentes.	Algunos cálculos excluyen nulos sin avisar.	Decide si nulo significa desconocido, cero o no aplica.
Filtrar tarde	Usar <code>HAVING</code> cuando tocaba <code>WHERE</code> .	Agrupar datos que no deberían entrar.	<code>WHERE</code> filtra filas; <code>HAVING</code> filtra grupos.
Fechas mal cerradas	<code>fecha <= '2026-03-31'</code> .	Puede perder horas del último día.	Usa rangos semiabiertos: <code>>= inicio</code> y <code>< fin</code> .
Moneda mezclada	<code>SUM(importe)</code> sin moneda.	Suma euros, dólares o créditos como si fueran iguales.	Agrupar o convertir antes de sumar.
Estado de negocio incompleto	<code>estado != 'pagado'</code> .	Incluye anulados, devueltos o pruebas.	Define catálogo permitido, no solo excluido.

Miremos el fallo de duplicación, que aparece mucho en sistemas Text-to-SQL. Si una tabla `alumnos` tiene una fila por alumno y `pagos` tiene varias filas por alumno, este SQL cuenta pagos, no alumnos:

```
SELECT COUNT(*) AS alumnos_con_pago
FROM alumnos AS a
JOIN pagos AS p ON p.alumno_id = a.alumno_id
WHERE p.estado = 'pendiente';
```

Si un alumno tiene tres pagos pendientes, aparece tres veces. La consulta puede ejecutarse sin quejarse y devolver un número bonito, pero el significado es otro. Para contar alumnos únicos:

```
SELECT COUNT(DISTINCT a.alumno_id) AS alumnos_con_pago_pendiente
FROM alumnos AS a
JOIN pagos AS p ON p.alumno_id = a.alumno_id
WHERE p.estado = 'pendiente';
```

La diferencia entre `COUNT(*)` y `COUNT(DISTINCT ...)` no es un detalle académico. Es la diferencia entre contar filas y contar entidades. Cuando una persona pregunta “cuántos alumnos”, normalmente quiere entidades. Cuando pregunta “cuántos pagos”, quiere eventos.

También importan las fechas. Si `fecha` incluye hora, esta condición parece correcta pero puede dejar fuera movimientos del 31 de marzo por la tarde:

```
WHERE fecha >= '2026-03-01'
AND fecha <= '2026-03-31'
```

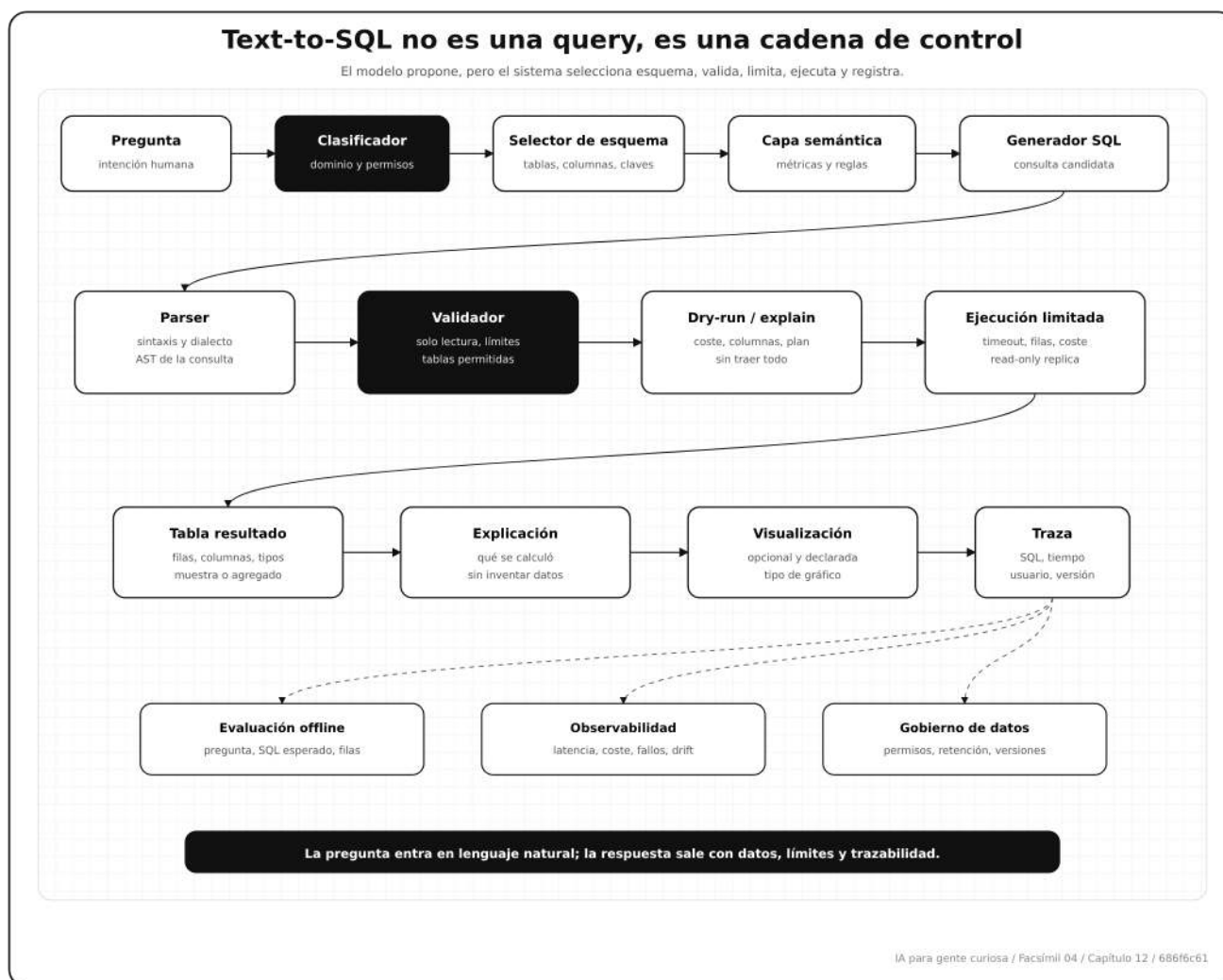
El patrón más robusto para intervalos suele ser:

```
WHERE fecha >= '2026-03-01'
AND fecha < '2026-04-01'
```

Un buen sistema Text-to-SQL no solo valida sintaxis. También mira estas trampas: cardinalidad de las tablas, tipo de métrica, nulos, moneda, rango temporal, catálogo de estados y claves de unión.

El mecanismo paso a paso

Text-to-SQL funciona bien cuando se separan tareas. Un modelo que recibe “todas las tablas, toda la documentación y genera SQL” puede acertar en una demo pequeña, pero se vuelve frágil con esquemas grandes.



Un sistema así no depende de una única llamada al modelo. Depende de contratos entre piezas. La persona pregunta; el clasificador decide si la pregunta es de datos o de documentos; el selector reduce el esquema; la capa semántica aporta definiciones; el generador propone SQL; el validador inspecciona; la base ejecuta con límites; la respuesta explica y registra.

Corregir el error en vez de fallar: el bucle de autocorrección

Aunque el modelo sea bueno, fallará: una columna mal escrita, un `JOIN` que no existe, una función de un dialecto equivocado. La diferencia entre un prototipo y un sistema serio es qué pasa entonces. La pieza que lo distingue es el **bucle de autocorrección**: cuando el validador rechaza el SQL o la base de datos devuelve un error, ese error se le **devuelve al modelo** junto con el SQL fallido para que lo arregle, en lugar de propagar el fallo al usuario.

El mensaje de error de una base de datos es muy informativo (`no such column: paid_at` , `ambiguous column name: campus`), y el modelo suele corregir a la primera o segunda con esa pista. Pero hay dos reglas de ingeniería innegociables:

- **Tope de reintentos.** Sin un máximo (dos o tres), un SQL que el modelo no sabe arreglar genera un bucle de llamadas y factura sin fin. Igual que el agente del capítulo 11: sin límite, no hay sistema.
- **Solo se reintenta lo seguro.** Se reintenta sobre errores de *sintaxis o esquema*, validando de nuevo cada candidato antes de ejecutar. Nunca se reintenta «a ciegas» ejecutando una y otra vez contra la base de datos de producción.

Cuando los reintentos se agotan, la respuesta correcta no es inventar: es **abstenerse** y, si procede, pedir aclaración. Un Text-to-SQL que se rinde con un «no he podido construir una consulta fiable para esto» es mejor que uno que devuelve un número que nadie puede defender.

El problema real: esquema, semántica y valores

Cuando un humano experto escribe SQL, no solo recuerda sintaxis. Recuerda el significado de cada tabla, qué columnas son fiables, qué claves se unen, qué estados se excluyen, qué fechas mandan y qué métricas no se calculan directamente.

Text-to-SQL falla por tres razones principales:

FOCO	QUÉ FALLA	EJEMPLO
Esquema	El modelo elige tabla o columna incorrecta.	Usa <code>created_at</code> en vez de <code>paid_at</code> .
Semántica	La consulta no respeta definición de negocio.	Cuenta alumnos anulados como activos.
Valores	No sabe cómo están escritos los datos reales.	Busca <code>pendiente</code> pero la tabla usa <code>PENDING</code> .

El *schema linking* conecta pregunta y esquema. Si la pregunta dice “campus con pagos pendientes”, el sistema debe vincular:

PALABRA DE LA PREGUNTA	CANDIDATO EN DATOS	POR QUÉ
campus	<code>alumnos.campus</code> o <code>pagos.campus</code>	Dimensión de agrupación.
pagos	tabla <code>pagos</code>	Hecho económico.
pendientes	<code>pagos.estado = 'pendiente'</code>	Filtro.
tres	<code>LIMIT 3</code>	Tamaño de salida.
más	<code>ORDER BY total DESC</code>	Orden descendente.

En bases pequeñas se puede meter todo el esquema en el prompt. En bases grandes no. Hay que recuperar el esquema como se recuperan documentos: por dominio, por nombres, por descripciones, por consultas de ejemplo y por permisos.

Recuperar el subesquema relevante

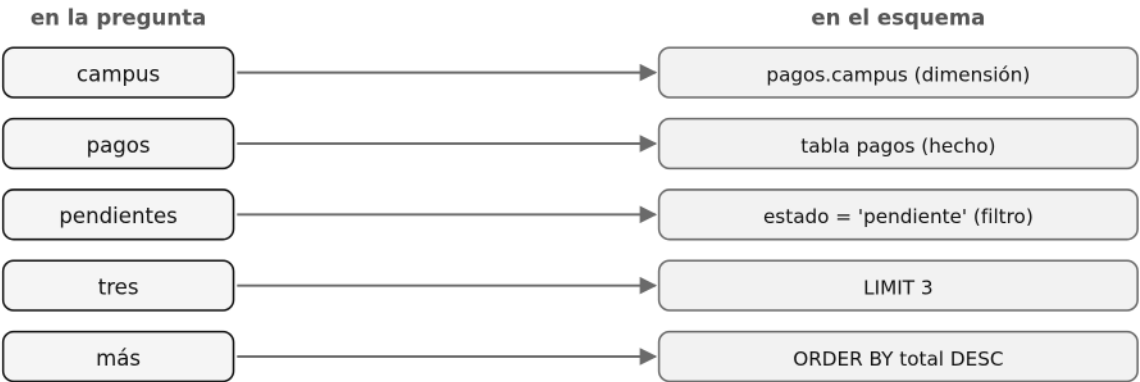
Conviene insistir en esto porque es el cuello de botella real: **la mayoría de los fallos de Text-to-SQL no son de sintaxis SQL, son de schema linking**. El modelo escribe SQL válido, pero sobre la tabla o la columna equivocada. Por eso, con un esquema de cientos de tablas, el trabajo de ingeniería no es «mejorar el prompt», sino **recuperar el subesquema correcto** antes de generar, igual que un RAG recupera los chunks correctos. Las señales que ayudan a ese retrieval del esquema:

SEÑAL	QUÉ APORTA
Nombres de tablas y columnas	El emparejamiento más directo, pero traicionero: <code>paid_at</code> frente a <code>created_at</code> .
Descripciones y comentarios	Lo que el nombre no dice; es donde vive la semántica de negocio.
Valores de ejemplo	Resuelve el problema de los valores: si la columna <code>estado</code> contiene <code>PENDING</code> , el modelo no inventará <code>pendiente</code> .
Claves y relaciones	Qué tablas se unen y por qué columna, para no inventar joins.
Consultas anteriores	Ejemplos parecidos que ya funcionaron (few-shot dirigido).

Esta dificultad es precisamente lo que miden los benchmarks académicos de Text-to-SQL, que evalúan sobre esquemas que el modelo no ha visto antes.⁹ La regla de ingeniería: trata el esquema como un corpus que se recupera, no como un texto fijo que se pega; y mide el linking por separado, porque es donde se gana o se pierde la batalla.

Schema linking: pregunta → esquema

«los tres campus con más pagos pendientes» se enlaza, pieza a pieza, con el esquema



Trampa de valores: si la columna guarda PENDING, «pendiente» no casa. · Trampa de esquema: la fecha que manda es paid_at, no created_at.

IA para gente curiosa / Facsímil 04 / Capítulo 12 / 686f6c61

La capa semántica como contrato

La capa semántica es el lugar donde una organización deja de discutir cada vez qué significa una métrica. No es una frase bonita en un prompt. Es un contrato versionado entre negocio, ingeniería y análisis.

Una definición mínima de métrica debería responder:

PREGUNTA	EJEMPLO PARA INGRESO_NETO
¿Qué entidad mide?	Pagos confirmados.
¿Qué columna numérica usa?	pagos.importe .
¿Qué estados entran?	Solo pagado .
¿Qué estados salen?	pendiente , anulado , devuelto , prueba .
¿Qué fecha manda?	fecha_pago , no fecha_creacion .
¿Qué dimensiones permite?	campus , titulacion , mes .
¿Quién puede verla?	Roles de análisis y dirección académica.
¿Cómo se prueba?	Casos con resultado esperado.

Esa misma definición puede expresarse de muchas formas: YAML, dbt metrics, una tabla de metadatos, una API interna o una vista SQL. Lo importante es que el modelo no invente la definición cada vez.

```
metric: ingreso_netos
entity: pagos
expression: SUM(importe)
filters:
  - estado = 'pagado'
time_dimension: fecha_pago
allowed_dimensions:
  - campus
  - titulacion
  - mes
blocked_columns:
  - dni
  - email_personal
owner: equipo_datos_matricula
```

Con esa capa, una pregunta como “ingresos por campus en marzo” no empieza desde cero. El sistema sabe que “ingresos” apunta a `ingreso_netos`, que marzo debe filtrarse con `fecha_pago`, que el grupo permitido es `campus` y que algunas columnas ni siquiera deben entrar en el contexto.

Para una persona curiosa, la idea es esta: una base de datos guarda datos, pero no siempre guarda significado. La capa semántica añade significado compartido. Para un ingeniero, añade algo igual de importante: reduce libertad donde la libertad produce errores.

Herramientas de datos: no todo es Text-to-SQL

Text-to-SQL es una herramienta, pero no la única. A veces conviene no dejar que el modelo genere SQL libre, sino exponer operaciones más estrechas.

TIPO DE HERRAMIENTA	QUÉ HACE	CUÁNDO USARLA	EJEMPLO
Text-to-SQL libre	Genera consultas nuevas.	Análisis exploratorio con validación fuerte.	"Agrupa pagos por campus y mes".
Plantilla parametrizada	Rellena parámetros de una consulta fija.	Métricas críticas y repetibles.	campus , fecha_inicio , fecha_fin .
Stored procedure	Llama a una función definida en la base.	Regla compleja y estable.	calcular_morosidad(campus, mes) .
Semantic layer	Consulta métricas y dimensiones declaradas.	BI, reporting y definiciones de negocio.	ingreso_neto por campus .
DataFrame tool	Opera sobre tablas en memoria.	Exploración local, CSV, notebooks.	Pandas, Polars, DuckDB.
Chart tool	Convierte resultados en gráfico.	Cuando la salida natural es visual.	Barras por campus.
Data quality tool	Comprueba nulos, duplicados o rangos.	Antes de confiar en una respuesta.	"¿Hay importes negativos?".

La pregunta de arquitectura no es “¿puedo generar SQL?”. La pregunta buena es “¿qué superficie de datos quiero exponer?”. Cuanto más abierta sea la herramienta, más validación necesita.

Contrato de una herramienta SQL

Una herramienta SQL no debería aceptar una cadena cualquiera. Debería tener un contrato que obligue a declarar intención, dominio, límites y formato de respuesta.


```
{
  "name": "consultar_datos",
  "input": {
    "question": "Pagos pendientes por campus en marzo",
    "domain": "matricula",
    "sql": "SELECT campus, SUM(importe) AS total FROM pagos ...",
    "dialect": "sqlite",
    "max_rows": 50,
    "timeout_ms": 1500,
    "purpose": "analisis_agregado"
  },
  "output": {
    "columns": ["campus", "total"],
    "rows": [["Norte", 19320.0]],
    "row_count": 1,
    "elapsed_ms": 24,
    "trace_id": "f4c12-001"
  },
  "errors": [
    "tabla_no_permitida",
    "consulta_no_select",
    "demasiadas_filas",
    "timeout",
    "sql_invalido"
  ]
}
```

Cada campo tiene una razón:

CAMPO	QUÉ CONTROLA	POR QUÉ IMPORTA
domain	Área funcional.	Evita mezclar tablas sin contexto.
sql	Consulta candidata.	Debe poder validarse y registrarse.
dialect	Motor SQL esperado.	LIMIT , fechas y funciones cambian entre motores.
max_rows	Filas máximas.	Protege coste y evita respuestas inmanejables.
timeout_ms	Tiempo máximo.	Una mala consulta no debe bloquear el sistema.
purpose	Uso declarado.	No es igual explorar que cerrar un informe.
trace_id	Identificador de ejecución.	Permite reproducir y auditar.

La herramienta puede estar detrás de OpenAI Function Calling, de un agente de LangChain, de LlamaIndex, de una API propia o de un servicio interno. El principio no cambia: el modelo propone argumentos estructurados, pero el servidor valida y ejecuta.

Permisos, datos sensibles y trazas

Una herramienta de datos tiene tres identidades distintas y conviene no mezclarlas:

IDENTIDAD	QUÉ REPRESENTA	ERROR TÍPICO
Persona usuaria	Quien hace la pregunta.	Darle acceso por el rol técnico del servidor.
Modelo	Quien propone la consulta.	Tratar su SQL como decisión autorizada.
Servicio de datos	Quien ejecuta.	Conectar con permisos demasiado amplios.

El modelo no debería tener permisos. Quien tiene permisos es el servicio que recibe una petición, revisa el rol de la persona, aplica reglas y ejecuta una consulta limitada. Esa separación evita que una respuesta dependa de una frase del prompt.

En una arquitectura seria, pondría estos controles:

CONTROL	QUÉ PROTEGE	EJEMPLO CONCRETO
Conexión read-only	Evita modificar datos.	Usuario SQL sin <code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code> ni DDL.
Réplica de lectura	Aísla producción.	Consultar una réplica o almacén analítico.
Row-level security	Limita filas por rol.	Un campus solo ve sus alumnos.
Column allowlist	Limita columnas visibles.	Exponer <code>campus</code> , no <code>dni</code> ni <code>email_personal</code> .
Query timeout	Evita consultas largas.	Cortar a 1.500 ms en exploración.
Row limit	Controla volumen de salida.	Máximo 100 filas por respuesta.
Redacción de logs	Evita guardar datos innecesarios.	Registrar SQL y hash de usuario, no tabla completa.
Trazabilidad	Permite reproducir.	<code>trace_id</code> , versión de esquema, modelo y validador.

La traza no es vigilancia decorativa. Es el expediente técnico de la respuesta. Si alguien pregunta “¿de dónde salió este número?”, necesitas poder reconstruir pregunta, usuario, rol, tablas candidatas, SQL generado, reglas aplicadas, resultado, latencia y versión del sistema.

Validación antes de ejecutar

La validación no es un adorno. Es el corazón del sistema. Un validador mínimo debería revisar:

CONTROL	QUÉ COMPRUEBA	EJEMPLO DE REGLA
Tipo de consulta	Solo lectura.	Aceptar únicamente <code>SELECT</code> o <code>WITH ... SELECT</code> .
Tablas permitidas	Superficie acotada.	<code>pagos</code> , <code>alumnos</code> , <code>campus</code> .
Columnas permitidas	Evitar columnas fuera de contrato.	No exponer <code>dni</code> si no hace falta.
Límite de filas	Salida manejable.	Añadir o exigir <code>LIMIT 100</code> .
Timeout	Coste temporal.	Cortar a 1,5 segundos.
Dialecto	Sintaxis correcta.	SQLite no es BigQuery.
Agregación	Preguntas agregadas devuelven agregados.	<code>SUM</code> , <code>COUNT</code> , <code>AVG</code> .
Filtros de usuario	Permisos por rol.	Campus permitido por sesión.
Explicación	Respuesta basada en resultado.	No resumir columnas que no salieron.

Herramientas como SQLGlot ayudan a parsear, inspeccionar y transpilar SQL entre dialectos.¹⁰ Motores embebidos como DuckDB son útiles para análisis local, CSV, Parquet o notebooks, porque permiten ejecutar SQL desde Python sin levantar un servidor externo.¹¹

Antes de ejecutar, una puerta de validación

Ejecutar antes de validar es jugársela: la validación es el corazón, no un adorno



Si una puerta falla, se rechaza la consulta: la base de datos no sabe qué pretendía el usuario.

IA para gente curiosa / Facsímil 04 / Capítulo 12 / 686f6c61

Query plan: cuando una consulta correcta no cabe

Una consulta puede ser correcta y aun así no ser aceptable. Si tarda treinta segundos, bloquea recursos o escanea una tabla enorme para devolver tres filas, el problema no es solo del modelo: es del sistema que no miró el plan.

El `query plan` es la explicación interna que calcula el motor antes de ejecutar. Según el motor puede verse con `EXPLAIN`, `EXPLAIN ANALYZE`, perfiles de consulta o `dry-run`. Los nombres cambian, pero la pregunta es la misma: ¿qué tendrá que hacer la base para responder?

```
EXPLAIN
SELECT campus, SUM(importe) AS total_pendiente
FROM pagos
WHERE estado = 'pendiente'
GROUP BY campus
ORDER BY total_pendiente DESC
LIMIT 3;
```

Al leer un plan, no hace falta entender todo desde el primer día. Empieza por estas señales:

SEÑAL	QUÉ SIGNIFICA	POR QUÉ IMPORTA
Full scan	Recorrer muchas filas de una tabla.	Puede ser normal en tablas pequeñas, caro en tablas grandes.
Index scan	Localizar filas con un índice.	Suele ayudar si el filtro es selectivo.
Cardinalidad estimada	Filas que el motor cree que pasarán.	Si estima mal, elige planes pobres.
Join strategy	Forma de unir tablas.	Nested loop, hash join o merge join tienen costes distintos.
Sort	Ordenación intermedia.	<code>ORDER BY</code> sobre muchas filas puede ser caro.
Temporary spill	Datos intermedios fuera de memoria.	Señal de presión de memoria o consulta pesada.

Un índice no resuelve todo. Acelera ciertas búsquedas a cambio de ocupar espacio y complicar escrituras. Si filtras mucho por `estado` y `fecha`, un índice puede ayudar:

```
CREATE INDEX idx_pagos_estado_fecha
ON pagos (estado, fecha);
```

Pero si casi todos los pagos están en `estado = 'pendiente'`, ese índice quizá no aporta mucho, porque el filtro no reduce bastante. Esta idea se llama selectividad: un filtro útil descarta muchas filas.

Para Text-to-SQL, el plan sirve como control previo:

PREGUNTA TÉCNICA	DECISIÓN DEL SISTEMA
¿Escanea más filas de las permitidas?	Pedir aclaración, añadir filtro o bloquear.
¿Usa tablas fuera del dominio?	Rechazar y pedir reformulación.
¿Ordena millones de filas sin agregación previa?	Proponer una consulta agregada.
¿No hay índice para el filtro principal?	Avisar de latencia o derivar a informe offline.

Dialecto SQL: el mismo pedido cambia por motor

SQL tiene una gramática común, pero cada motor añade funciones, tipos y límites. Un sistema que genera SQL debe saber para qué motor escribe. SQLite no es PostgreSQL, PostgreSQL no es BigQuery, BigQuery no es Snowflake.

NECESIDAD	SQLITE	POSTGRESQL	BIGQUERY	SNOWFLAKE
Limitar filas	<code>LIMIT 10</code>	<code>LIMIT 10</code>	<code>LIMIT 10</code>	<code>LIMIT 10</code>
Mes de una fecha	<code>strftime('%Y-%m', fecha)</code>	<code>date_trunc('month', fecha)</code>	<code>DATE_TRUNC(fecha, MONTH)</code>	<code>DATE_TRUNC('MONTH', fecha)</code>
Concatenar texto	<code>`a`</code>		<code>b`</code>	<code>`a`</code>
Fecha actual	<code>date('now')</code>	<code>CURRENT_DATE</code>	<code>CURRENT_DATE()</code>	<code>CURRENT_DATE()</code>
Muestra aproximada	Limitado	<code>TABLESAMPLE</code>	<code>TABLESAMPLE SYSTEM</code>	<code>SAMPLE</code>

No hace falta memorizar todos los dialectos. Lo importante es no mezclar. Si el contrato dice `dialect: "sqlite"`, el generador, el parser, los ejemplos y el validador deben hablar SQLite. Si el almacén real es BigQuery, conviene validar con BigQuery o con un parser que entienda sus particularidades.

Esta es una razón más para no pegar ejemplos al azar en el prompt. Un ejemplo de PostgreSQL puede enseñar al modelo una función que luego falla en BigQuery. Los ejemplos son datos de entrenamiento local para la consulta que estás a punto de generar; si están mal elegidos, orientan mal.

Cómo elegir arquitectura

La solución correcta depende del riesgo, del tamaño del esquema y de lo repetible que sea la pregunta.

SITUACIÓN	ARQUITECTURA RECOMENDADA	POR QUÉ
Métrica crítica y estable.	Plantilla parametrizada.	Menos libertad, más confianza.
Exploración interna con datos agregados.	Text-to-SQL con validación y trazas.	Permite flexibilidad controlada.
BI con definiciones compartidas.	Semantic layer + modelo.	Las métricas viven fuera del prompt.
CSV local o notebook.	DuckDB/DataFrame tool.	Iteración rápida y entorno cerrado.
Esquema enorme.	Retrieval de esquema + examples RAG.	No cabe todo el catálogo.
Varias bases y documentos.	Router + herramientas especializadas.	No todo debe ir por SQL.
Preguntas repetidas de negocio.	Stored procedures o vistas.	La lógica queda versionada.

Mi regla práctica: si la pregunta puede romper un informe importante, no empieces con SQL libre. Empieza con métrica declarada, plantilla o vista. Usa Text-to-SQL libre para exploración, no para convertir cada pregunta de negocio en una consulta nueva sin revisión.

Coste, latencia y contexto

Text-to-SQL parece barato porque la salida es corta. Pero el contexto puede crecer muchísimo: documentación de tablas, columnas, ejemplos, métricas, permisos, dialecto y trazas anteriores.

El contexto que entra al modelo es la suma de todo lo que tiene que ver para escribir SQL correcto, y crece rápido:

$$T_{ctx} = T_{pregunta} + T_{schema} + T_{docs} + T_{ejemplos} + T_{politicas}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{ctx}	Tokens totales de contexto.	6.400 tokens.
T_{pregunta}	Pregunta del usuario.	30 tokens.
T_{schema}	Tablas, columnas y claves incluidas.	2.500 tokens.
T_{docs}	Documentación de negocio.	1.200 tokens.
T_{ejemplos}	Consultas parecidas.	1.800 tokens.
$T_{\text{políticas}}$	Permisos y reglas.	870 tokens.

La latencia total tampoco es solo la generación del LLM: es la suma de toda la cadena, desde enrutar la pregunta hasta redactar la respuesta:

$$L_{\text{total}} = L_{\text{router}} + L_{\text{schema}} + L_{\text{LLM}} + L_{\text{validacion}} + L_{\text{db}} + L_{\text{resumen}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
L_{schema}	Tiempo de recuperar esquema relevante.	80 ms.
L_{LLM}	Tiempo de generar SQL o plan.	1.200 ms.
$L_{\text{validacion}}$	Parser, reglas y dry-run.	90 ms.
L_{db}	Ejecución en base de datos.	300 ms.
L_{resumen}	Redacción final.	700 ms.

Y el coste por pregunta suma todo lo que se paga, no solo los tokens del modelo:

$$C = C_{\text{modelo}} + C_{\text{db}} + C_{\text{observabilidad}} + C_{\text{mantenimiento}}$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
C_{modelo}	Tokens de entrada y salida del LLM.	Factura del proveedor por consulta.
C_{db}	Cómputo de ejecutar la consulta en la base de datos.	Warehouse que cobra por consulta o por cómputo escaneado.
$C_{\text{observabilidad}}$	Trazas, logs y métricas de cada pregunta.	Almacenamiento y herramienta de observabilidad.
$C_{\text{mantenimiento}}$	Trabajo humano de mantener el sistema.	Documentar columnas, versionar métricas, revisar fallos.

La parte invisible suele ser $C_{\text{mantenimiento}}$: documentar columnas, versionar métricas, revisar consultas fallidas, actualizar ejemplos y controlar cambios de esquema. Un Text-to-SQL no se paga una vez; se paga cada vez que el esquema cambia.

Juntando las tres cuentas en una pregunta concreta, con los valores de los ejemplos de arriba:

MAGNITUD	CUENTA	RESULTADO
Contexto	$30 + 2500 + 1200 + 1800 + 870$	6 400 tokens
Latencia	$20 + 80 + 1200 + 90 + 300 + 700$ ms (router, esquema, LLM, validación, db, resumen)	$\approx 2,4$ s
Coste del modelo	6 400 tokens de entrada + 150 de salida, a tarifas ilustrativas de 2,00/8,00 por millón	$\approx 0,014$ por pregunta

La lectura de ingeniería: el coste del modelo (0,014) parece ridículo, pero **el contexto es el 98 % de los tokens**, y de él la mayor parte es el esquema y los ejemplos. Por eso la palanca real de coste y latencia no es el modelo más barato, sino **recuperar menos esquema** (volvemos al schema linking): cada tabla que no metes en el prompt baja las tres cuentas a la vez.

Evaluar Text-to-SQL

La evaluación clásica compara SQL generado contra SQL esperado. Eso ayuda, pero no basta. Dos consultas distintas pueden devolver el mismo resultado; dos consultas parecidas pueden divergir en casos frontera.

MÉTRICA	QUÉ MIDE	CUIDADO
Exact match	Si el SQL coincide con el esperado.	Penaliza consultas equivalentes escritas distinto.
Execution accuracy	Si produce el resultado correcto.	Puede acertar por casualidad en datos pequeños.
Result-set match	Si filas y columnas coinciden.	Hay que controlar orden, tipos y redondeo.
Component match	Si <code>SELECT</code> , <code>WHERE</code> , <code>JOIN</code> , <code>GROUP BY</code> están bien.	Útil para depurar.
Schema-link accuracy	Si eligió tablas y columnas correctas.	Necesita anotación o revisión.
Permission pass rate	Si respeta permisos y límites.	No basta medir precisión.
Query cost	Coste estimado o real de ejecución.	Una query correcta puede ser inviable.
Latencia P95	Tiempo para el 95% de consultas.	La media oculta colas lentas.
Clarification rate	Cuándo pide aclaración.	Preguntar puede ser mejor que inventar.

Estas métricas no son invento nuestro: vienen de los benchmarks académicos de Text-to-SQL. **Spider** estandarizó la evaluación sobre esquemas no vistos y popularizó dos medidas: el *exact match* (componente a componente) y la *execution accuracy* (mismas filas que el SQL de oro).¹² **BIRD** dio un paso hacia el mundo real: bases sucias y grandes, valores con ruido, y una métrica de *valid efficiency score* que penaliza consultas correctas pero carísimas.¹³ La lección que se traslada a tu sistema: **execution accuracy importa más que exact match** (lo que cuenta es la cifra correcta, no que el SQL sea idéntico), pero ninguna métrica pública sustituye a un dataset con tu esquema, tus valores y tus permisos.

Un dataset propio debería tener:

CAMPO	EJEMPLO
question	"Pagos pendientes por campus en marzo".
user_role	analista_matricula .
allowed_tables	pagos , alumnos , campus .
gold_sql	Consulta esperada o plantilla.
expected_result	Filas esperadas.
must_not_use	Columnas que no proceden.
notes	Definición de negocio relevante.

En producción mediría, como mínimo:

CAPA	PREGUNTA DE EVALUACIÓN
Clasificador	¿Detectó que era una pregunta de datos?
Selector de esquema	¿Incluyó las tablas necesarias y excluyó ruido?
Generador	¿Produjo SQL válido para el dialecto?
Validador	¿Bloqueó lo que debía bloquear?
Ejecución	¿Devolvió resultado correcto dentro de límite?
Resumen	¿Explicó solo lo que aparece en la tabla?
Trazas	¿Puedo reproducir la respuesta?

Una forma práctica de empezar es construir un pequeño harness de evaluación. No tiene que ser perfecto. Tiene que hacer visible cuándo el sistema mejora o empeora.

```
{
  "id": "matricula-001",
  "question": "Alumnos con pago pendiente por campus en marzo",
  "role": "analista_matricula",
  "dialect": "sqlite",
  "allowed_tables": ["pagos", "alumnos"],
  "expected_sql_patterns": ["GROUP BY campus", "estado = 'pendiente'"],
  "expected_result": [
    {"campus": "Norte", "alumnos": 2},
    {"campus": "Centro", "alumnos": 1}
  ],
  "max_latency_ms": 1500,
  "max_rows": 20,
  "must_not_use": ["dni", "email_personal"],
  "review_note": "Debe contar alumnos únicos, no pagos."
}
```

En ese ejemplo no basta con que aparezca `GROUP BY campus`. El caso dice explícitamente que hay que contar alumnos únicos. Esa nota evita que una consulta con `COUNT(*)` pase por casualidad cuando los datos de prueba son pequeños.

Un harness útil debería guardar cuatro salidas:

SALIDA	PARA QUÉ SIRVE
SQL generado	Revisar estructura y dialecto.
Resultado devuelto	Comparar con la tabla esperada.
Razón de validación	Saber si el sistema aceptó o bloqueó con criterio.
Traza	Reproducir el caso con misma versión de esquema y modelo.

Y debería separar tipos de error. No es igual fallar por sintaxis que fallar por semántica:

TIPO DE FALLO	EJEMPLO	QUÉ ARREGLAR
Sintaxis	Función inexistente para el dialecto.	Ejemplos y parser.
Schema linking	Usa <code>created_at</code> en lugar de <code>paid_at</code> .	Descripciones y selector de esquema.
Semántica	Cuenta pagos cuando debía contar alumnos.	Capa semántica y casos de prueba.
Permisos	Incluye columna no permitida.	Validador y allowlist.
Coste	Query correcta pero pesada.	Plan, índices, límites o vista agregada.
Explicación	Resume algo que no está en el resultado.	Contrato de respuesta y groundedness.

Cuando conviene pedir aclaración

Una buena herramienta de datos no responde siempre. A veces pregunta.

PREGUNTA ORIGINAL	QUÉ FALTA	MEJOR RESPUESTA DEL SISTEMA
"Ventas del mes".	Mes, definición de venta, región.	"¿Qué mes y qué métrica de ventas quieres usar?"
"Alumnos activos".	Definición de activo.	"Puedo usar matrícula vigente o acceso reciente. ¿Cuál prefieres?"
"Top clientes".	Top por ingresos, pedidos o margen.	"¿Quieres ordenar por ingresos, margen o número de pedidos?"
"Comparar campus".	Periodo y métrica.	"Dime periodo y métrica principal."

Pedir aclaración no es fallar. En datos, muchas respuestas incorrectas nacen de contestar demasiado rápido.

Caso completo: de pregunta a respuesta trazable

Tomemos una pregunta realista:

“Dame los tres campus con más alumnos con pagos pendientes en marzo.”

Parece una pregunta simple. No lo es. Hay una entidad, alumnos; un evento, pagos; una condición, pendientes; una ventana temporal, marzo; una agrupación, campus; un ranking, los tres primeros.

El sistema debería recorrer algo parecido a esto:

PASO	DECISIÓN	RESULTADO INTERMEDIO
1. Intención	Es una pregunta de datos agregados.	Ruta a herramienta SQL.
2. Dominio	Matrícula y pagos.	No consulta documentos generales.
3. Usuario	Rol <code>analista_matricula</code> .	Puede ver agregados por campus.
4. Esquema	Necesita <code>pagos</code> y quizá <code>alumnos</code> .	No carga todo el catálogo.
5. Semántica	“Alumnos con pagos pendientes” cuenta alumnos únicos.	Métrica: <code>COUNT(DISTINCT alumno_id)</code> .
6. Fecha	Marzo se interpreta como rango semiabierto.	<code>>= '2026-03-01'</code> y <code>< '2026-04-01'</code> .
7. SQL candidato	Genera consulta agregada.	<code>GROUP BY campus , ORDER BY , LIMIT 3</code> .
8. Validación	Solo lectura, tablas permitidas, límite.	Acepta o pide corrección.
9. Plan	Estima filas y coste.	Bloquea si escanea demasiado.
10. Ejecución	Ejecuta en réplica de lectura.	Devuelve tabla pequeña.
11. Respuesta	Explica solo la tabla.	No inventa causas.
12. Traza	Registra expediente.	Permite reproducir.

Un SQL razonable podría ser:

```

SELECT
  p.campus,
  COUNT(DISTINCT p.alumno_id) AS alumnos_con_pago_pendiente
FROM pagos AS p
WHERE p.estado = 'pendiente'
  AND p.fecha >= '2026-03-01'
  AND p.fecha < '2026-04-01'
GROUP BY p.campus
ORDER BY alumnos_con_pago_pendiente DESC
LIMIT 3;

```

Fíjate en lo que no hace:

NO HACE	POR QUÉ
No usa <code>COUNT(*)</code> .	Contaría pagos, no alumnos.
No usa <code>estado != 'pagado'</code> .	Metería estados no definidos.
No pide columnas personales.	La pregunta solo necesita agregados.
No devuelve filas individuales.	La salida pedida es un ranking.
No explica causas.	La consulta no investigó causas, solo conteos.

La respuesta humana debería sonar así:

“Con la definición de pagos pendientes como `estado = 'pendiente'` y tomando marzo como `[2026-03-01, 2026-04-01)`, los tres campus con más alumnos únicos con pagos pendientes son Norte, Centro y Sur. La consulta usa datos agregados y no incluye información personal.”

Y la traza técnica podría guardar:

```
{
  "trace_id": "f4c12-demo-023",
  "question": "Dame los tres campus con más alumnos con pagos pendientes en marzo",
  "route": "sql_tool",
  "role": "analista_matricula",
  "dialect": "sqlite",
  "tables": ["pagos"],
  "metric": "count_distinct_alumno_id",
  "validated": true,
  "checks": ["read_only", "allowed_tables", "row_limit", "date_range"],
  "row_count": 3,
  "elapsed_ms": 31
}
```

Esta traza no es para enseñarla entera a la persona que pregunta. Es para que el equipo pueda auditar, depurar y mejorar el sistema.

Soluciones de terceros y piezas habituales

Hay herramientas ya hechas, pero conviene saber qué problema resuelve cada una. No todas sustituyen una arquitectura propia.

PIEZA	QUÉ APORTA	QUÉ REVISARÍA ANTES DE USARLA
LangChain SQL Agent	Flujo agentic para inspeccionar esquema, generar, revisar y ejecutar SQL.	Permisos de conexión, trazas, límites y revisión humana.
LlamaIndex NL SQL	Query engines para lenguaje natural sobre tablas SQL.	Qué esquema entra, cómo controla ejecución y cómo registra fuentes.
Vanna	Enfoque natural language -> SQL -> respuestas con permisos y componentes de UI.	El repositorio público aparece archivado desde marzo de 2026; revisaría mantenimiento y versión usada. ¹⁴
SQLGlot	Parser, AST y transpiler SQL.	Cobertura del dialecto y reglas propias de validación.
DuckDB	Motor local para análisis, CSV y Parquet desde Python.	Memoria, tamaño de datos y diferencias frente al motor de producción.
dbt / capa semántica	Métricas y transformaciones versionadas.	Quién mantiene definiciones y cómo se exponen al modelo.
BI tradicional	Dashboards y métricas curadas.	Qué preguntas quedan fuera del dashboard.

Una buena arquitectura puede mezclar varias: LangChain o LlamaIndex para orquestar, SQLGlot para validar, DuckDB para prototipos locales, una capa semántica para métricas y una API propia para permisos.

En el día a día

En el día a día, Text-to-SQL aparece cuando la respuesta no está escrita en ningún documento, sino que hay que **calcularla** sobre datos vivos: «¿cuántos alumnos con pagos pendientes hay en el campus Norte este mes?». Ningún chunk contiene ese número; sale de una consulta. Es lo que permite que una persona de negocio pregunte en lenguaje natural y obtenga una cifra de la base de datos sin saber SQL. En un producto, es el puente entre el RAG (que recupera texto) y el estado exacto del sistema (que vive en tablas).

La parte delicada es que aquí un error no se nota como un texto raro: se nota como un **número equivocado que parece correcto**. Una consulta puede ejecutar sin error y calcular otra cosa (contar pagos en vez de alumnos, sumar una columna que no era), y nadie lo ve hasta que alguien toma una decisión con ese dato. Por eso Text-to-SQL es, sobre todo, un problema de validación y de semántica, no de generar SQL bonito.

Por qué debería importarte

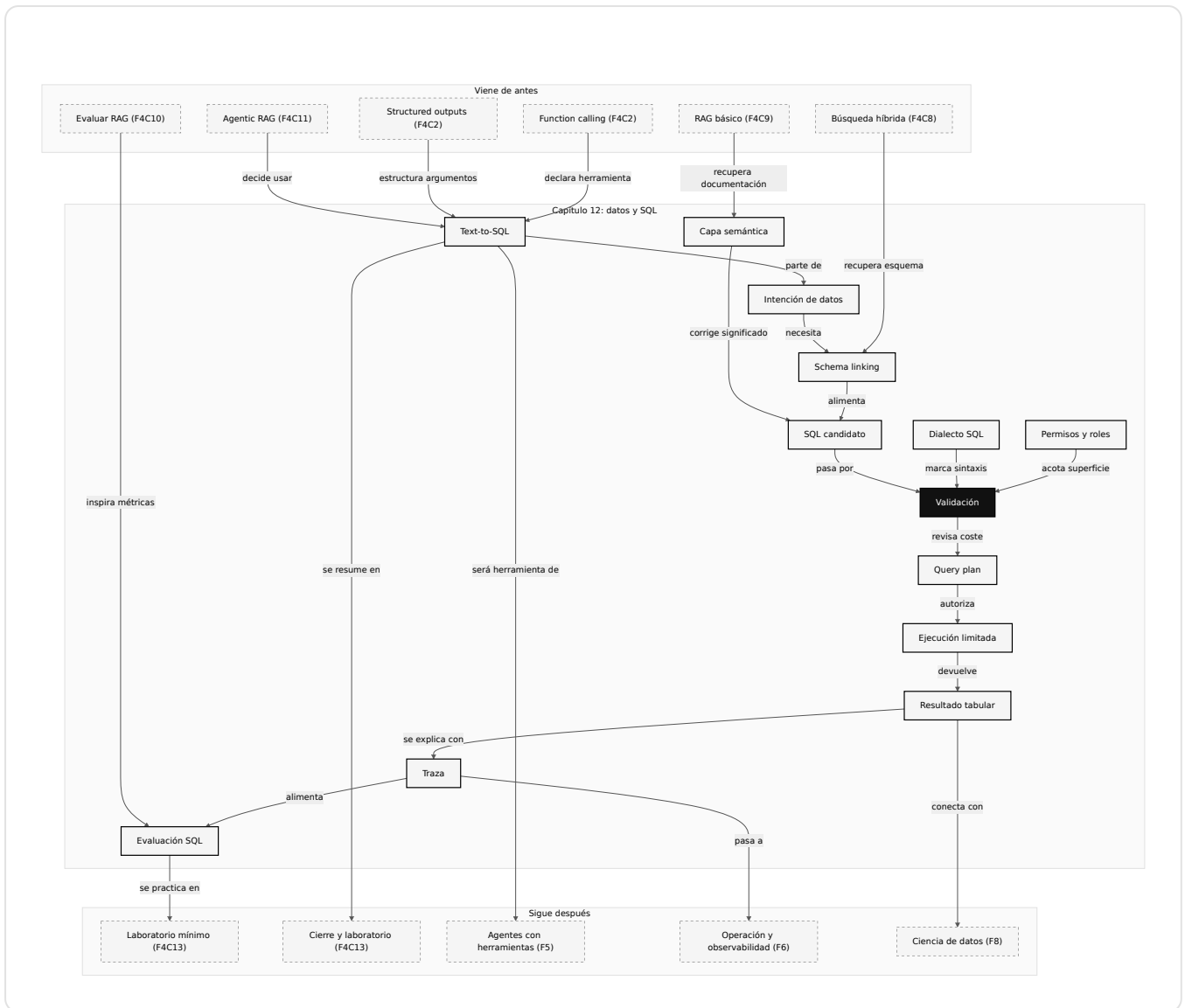
Porque es donde la IA toca los datos de verdad, y donde un fallo cuesta más: una respuesta de RAG mal fundamentada es un texto dudoso; un número mal calculado es una decisión mal tomada. Si entiendes la cadena (esquema, semántica, validación, permisos, ejecución y traza), sabes por qué un sistema responde un número y puedes defenderlo o corregirlo. Sin esa cadena, tienes un generador de SQL que a veces acierta y nunca sabes cuándo.

También importa porque conecta dos mundos que la mayoría de la gente mantiene separados: el lenguaje natural y el modelo de datos. Saber traducir «ingresos del trimestre» a la métrica exacta, con su definición versionada y sus permisos, es una habilidad de ingeniería que no caduca, use el motor que use por debajo.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir SQL válido con respuesta correcta	Una consulta puede ejecutar y aun así calcular otra cosa.	Evaluar resultado, tablas usadas y definición de negocio.
Meter todo el esquema en el prompt	Con esquemas grandes sube el coste y entra ruido.	Recuperar solo tablas, columnas y ejemplos relevantes.
Olvidar la capa semántica	“Ingresos”, “activo” o “pendiente” no significan lo mismo en cada empresa.	Declarar métricas y definiciones fuera del prompt.
Contar filas cuando quería entidades	<code>COUNT(*)</code> puede contar pagos, no alumnos.	Revisar cardinalidad y usar <code>COUNT(DISTINCT ...)</code> cuando proceda.
Ignorar el plan de consulta	Una query correcta puede ser demasiado cara.	Revisar <code>EXPLAIN</code> , cardinalidad, índices y límites.
Mezclar dialectos	Una función válida en un motor puede fallar en otro.	Pasar <code>dialect</code> en el contrato y validar contra ese motor.
Ejecutar antes de validar	La base de datos no sabe qué pretendía el usuario.	Parsear, limitar, revisar permisos y hacer dry-run.
Medir solo exact match	SQL distinto puede producir el mismo resultado correcto.	Combinar execution accuracy, result-set match y revisión por componentes.
No registrar trazas	No puedes explicar por qué salió un número.	Guardar pregunta, esquema, SQL, resultados, latencia y versión.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Text-to-SQL	Traducción controlada de una pregunta humana a una consulta SQL ejecutable.
SQL	Lenguaje declarativo para consultar y transformar datos relacionales.
Tabla	Conjunto de filas y columnas.
Fila	Registro individual dentro de una tabla.
Columna	Atributo de una tabla, como fecha, estado o importe.
Clave primaria	Columna o conjunto de columnas que identifica una fila única.
Clave foránea	Columna que conecta una tabla con otra.
JOIN	Unión entre tablas mediante una relación, normalmente una clave.
Cardinalidad	Número aproximado de filas que participan en una operación.
Índice	Estructura que ayuda a encontrar filas sin recorrer toda la tabla.
Schema linking	Vincular palabras de la pregunta con tablas, columnas, claves y valores.
Dialecto SQL	Variante de SQL de un motor concreto.
Semantic layer	Capa de métricas y reglas de negocio compartidas.
Dry-run	Comprobación previa de una consulta antes de ejecutarla plenamente.
Query plan	Plan interno de ejecución calculado por la base de datos.
Read-only	Conexión que solo permite lectura.
Row-level security	Regla que limita qué filas puede ver cada rol o persona.
Execution accuracy	Métrica que evalúa si el resultado producido por la consulta es correcto.
Result-set match	Comparación entre resultado esperado y resultado devuelto.
Row limit	Límite máximo de filas que puede devolver una consulta.
Traza SQL	Registro de pregunta, SQL, usuario, tablas, tiempo, resultado y errores.

Antes de pasar página

- ☐ ¿Puedo explicar por qué Text-to-SQL no es acceso libre a una base de datos? (Si no, vuelve a «Qué no es Text-to-SQL».)
- ☐ ¿Sé leer una consulta con `SELECT`, `FROM`, `WHERE`, `JOIN`, `GROUP BY`, `ORDER BY` y `LIMIT`? (Si no, vuelve a «SQL desde cero, pero sin rebajarlo».)
- ☐ ¿Puedo explicar qué es schema linking con un ejemplo? (Si no, vuelve a «El problema real: esquema, semántica y valores».)
- ☐ ¿Sé distinguir entidad, evento, clave primaria y clave foránea? (Si no, vuelve a «SQL desde cero, pero sin rebajarlo».)
- ☐ ¿Puedo detectar cuándo un `JOIN` duplica filas? (Si no, vuelve a «Errores SQL que cambian la historia».)
- ☐ ¿Sé por qué `COUNT(*)` y `COUNT(DISTINCT ...)` no responden a lo mismo? (Si no, vuelve a «Errores SQL que cambian la historia».)
- ☐ ¿Sé distinguir SQL libre, plantilla parametrizada, stored procedure y semantic layer? (Si no, vuelve a «Herramientas de datos: no todo es Text-to-SQL».)
- ☐ ¿Puedo diseñar un contrato mínimo para una herramienta SQL? (Si no, vuelve a «Contrato de una herramienta SQL».)
- ☐ ¿Sé qué validaciones haría antes de ejecutar una consulta? (Si no, vuelve a «Validación antes de ejecutar».)
- ☐ ¿Sé leer las señales básicas de un query plan? (Si no, vuelve a «Query plan: cuando una consulta correcta no cabe».)
- ☐ ¿Sé por qué el dialecto SQL debe viajar en el contrato? (Si no, vuelve a «Dialecto SQL: el mismo pedido cambia por motor».)
- ☐ ¿Puedo calcular por qué el contexto crece con esquema, documentación y ejemplos? (Si no, vuelve a «Coste, latencia y contexto».)
- ☐ ¿Sé diferenciar exact match, execution accuracy y result-set match? (Si no, vuelve a «Evaluar Text-to-SQL».)
- ☐ ¿Sé qué revisar en la traza de una consulta Text-to-SQL para confiar en el resultado? (Si no, repasa la sección correspondiente.)

En resumen

IDEA FUERZA	DETALLE
Text-to-SQL no es una query suelta.	Es una cadena con intención, esquema, SQL, validación, ejecución, resultado y traza.
El esquema manda.	Si el sistema no entiende tablas, columnas, claves y valores, generará consultas frágiles.
La semántica evita números falsos.	Las métricas de negocio deben estar definidas fuera del prompt.
La cardinalidad cambia respuestas.	Contar filas, eventos o entidades no es lo mismo.
El plan importa.	Una consulta correcta puede ser demasiado cara para ejecutarse en vivo.
La validación es obligatoria.	Solo lectura, límites, permisos, dialecto, timeout y trazas antes de ejecutar.
La evaluación mira resultados.	Exact match ayuda, pero execution accuracy y result-set match son centrales.

Para saber más

DuckDB. (2026). *Python API*. [Documentación oficial](#)

LangChain. (2026). *Build a SQL agent*. [Documentación oficial](#)

Lei, F. y otros (2024). *Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows*. [arXiv](#)

Li, J. y otros (2023). *Can LLM Already Serve as A Database Interface? A BIG Bench for Large-Scale Database Grounded Text-to-SQLs*. [arXiv](#)

LlamaIndex. (2026). *NL SQL table query engine*. [Documentación oficial](#)

OpenAI. (2026). *Function calling*. [Documentación oficial](#)

OpenAI. (2026). *Structured model outputs*. [Documentación oficial](#)

Scholak, T., Schucher, N. y Bahdanau, D. (2021). *PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models*. *Proceedings of EMNLP*, 9895-9901. [DOI](#)

SQLGlott. (2026). *Python SQL parser and transpiler*. [Documentación](#)

Vanna AI. (2026). *Vanna 2.0: Turn Questions into Data Insights*. [GitHub](#)

Wang, B., Shin, R., Liu, X., Polozov, O. y Richardson, M. (2020). *RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers*. *Proceedings of ACL*, 7567-7578. [DOI](#)

Yu, T. y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. *Proceedings of EMNLP*, 3911-3921. [ACL Anthology](#)

Notas

1. Yu, T. y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. *Proceedings of EMNLP*, 3911-3921. [ACL Anthology](#). ↵
2. Wang, B., Shin, R., Liu, X., Polozov, O. y Richardson, M. (2020). *RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers*. *Proceedings of ACL*, 7567-7578. [DOI](#). ↵
3. Scholak, T., Schucher, N. y Bahdanau, D. (2021). *PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models*. *Proceedings of EMNLP*, 9895-9901. [DOI](#). ↵
4. Li, J. y otros (2023). *Can LLM Already Serve as A Database Interface? A Blg Bench for Large-Scale Database Grounded Text-to-SQLs*. [arXiv](#). ↵
5. Lei, F. y otros (2024). *Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows*. [arXiv](#). ↵
6. LangChain. (2026). *Build a SQL agent*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
7. LlamaIndex. (2026). *NL SQL table query engine*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
8. OpenAI. (2026). *Function calling*. [Documentación oficial](#). OpenAI. (2026). *Structured model outputs*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
9. Tao Yu y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. EMNLP. <https://arxiv.org/abs/1809.08887>. ↵
0. SQLGlott. (2026). *Python SQL parser and transpiler*. [Documentación](#). Consultado el 26 de mayo de 2026. ↵
1. DuckDB. (2026). *Python API*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵

- .2. Tao Yu y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. EMNLP. <https://arxiv.org/abs/1809.08887>. ↵
- .3. Jinyang Li y otros (2023). *Can LLM Already Serve as a Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs (BIRD)*. NeurIPS. <https://arxiv.org/abs/2305.03111>. ↵
- .4. Vanna AI. (2026). *Vanna 2.0: Turn Questions into Data Insights*. [GitHub](#). Consultado el 26 de mayo de 2026. El repositorio público consultado aparece como archivado el 29 de marzo de 2026. ↵

Capítulo 13: Lo que deberías saber: la caja de herramientas

Entrando en el cierre

Este facsímil empezó con una pregunta muy práctica: cuando un sistema de IA no hace lo que necesitas, ¿qué cambias exactamente?

Podrías cambiar el prompt. Podrías exigir una salida estructurada. Podrías añadir documentos con RAG. Podrías conectar una herramienta. Podrías elegir otro modelo. Podrías servirlo en local. Podrías usar embeddings, bases vectoriales, búsqueda híbrida, evaluación, GraphRAG o Text-to-SQL. La caja ya no está vacía.

Pero tener herramientas no significa tener criterio. Un equipo puede complicar una solución hasta que nadie la entiende. También puede quedarse corto y llamar “IA” a una demo que no soporta coste, permisos, datos reales ni evaluación. Este capítulo es una revisión activa: no busca que recuerdes nombres, sino que puedas justificar decisiones.

Si has entendido el facsímil, deberías poder mirar un caso y decir con calma: qué problema hay, qué intervención toca, qué contrato hace falta, qué métrica lo prueba, qué coste introduce y qué pasaría si mañana cambia el dato, el modelo o el usuario.

Este capítulo es la **brújula** del cierre: una revisión activa de todo lo construido, capítulo a capítulo, para poder justificar decisiones en vez de recordar nombres. Si el recap es una conversación técnica, su valor está en sostener cada afirmación con criterio: qué intervención toca, qué contrato hace falta y qué métrica lo prueba.

Fecha de corte y alcance

Fecha de corte: 26 de mayo de 2026.

Alcance: este cierre resume mecanismos estables del facsímil y referencias ya trabajadas: function calling, salidas estructuradas, prompt caching, model cards, LoRA, QLoRA, RAG, búsqueda vectorial, búsqueda híbrida, evaluación RAG, GraphRAG y Text-to-SQL.

Las herramientas concretas cambiarán. Los nombres de APIs, modelos, precios, runtimes y proveedores se moverán. El criterio que queremos conservar es más estable: separar entrada, contexto, herramienta, pesos, despliegue, evaluación y trazas.

Qué debería llevarse cada perfil

Este facsímil está escrito para gente curiosa con o sin perfil técnico. Eso no significa que todo el mundo tenga que llevarse lo mismo. Significa que cada persona debería poder salir con una forma más precisa de mirar un sistema de IA.

PERFIL	DEBERÍA PODER HACER AL TERMINAR ESTE FACSIMIL	PREGUNTA QUE YA NO DEBERÍA ACEPTAR SIN MÁS
Ingeniería	Diseñar una arquitectura mínima con contratos, permisos, trazas y evaluación.	“¿Y si le ponemos un agente?”
Datos	Separar pregunta documental, búsqueda semántica, métrica, SQL y validación de resultados.	“¿El modelo ya sabe los datos?”
Producto	Decidir si una función merece prompt, RAG, tool, ajuste, modelo local o nada de IA.	“¿Podemos meter IA aquí?”
Docencia	Explicar cada herramienta con un ejemplo pequeño, una limitación y una prueba.	“¿Esto funciona porque lo he visto en una demo?”
Persona curiosa	Preguntar qué entra, qué sale, qué se mide, qué cuesta y qué pasa cuando no hay evidencia.	“¿La IA lo ha dicho, entonces será verdad?”

El punto común es el criterio. La persona técnica lo aplicará escribiendo código, definiendo contratos o levantando infraestructura. La persona no técnica lo aplicará haciendo mejores preguntas, detectando promesas débiles y pidiendo pruebas antes de confiar.

La brújula: qué quieres cambiar

La pregunta más importante del facsímil no es “¿qué herramienta está de moda?”. Es esta:

¿Dónde está el cuello del problema?

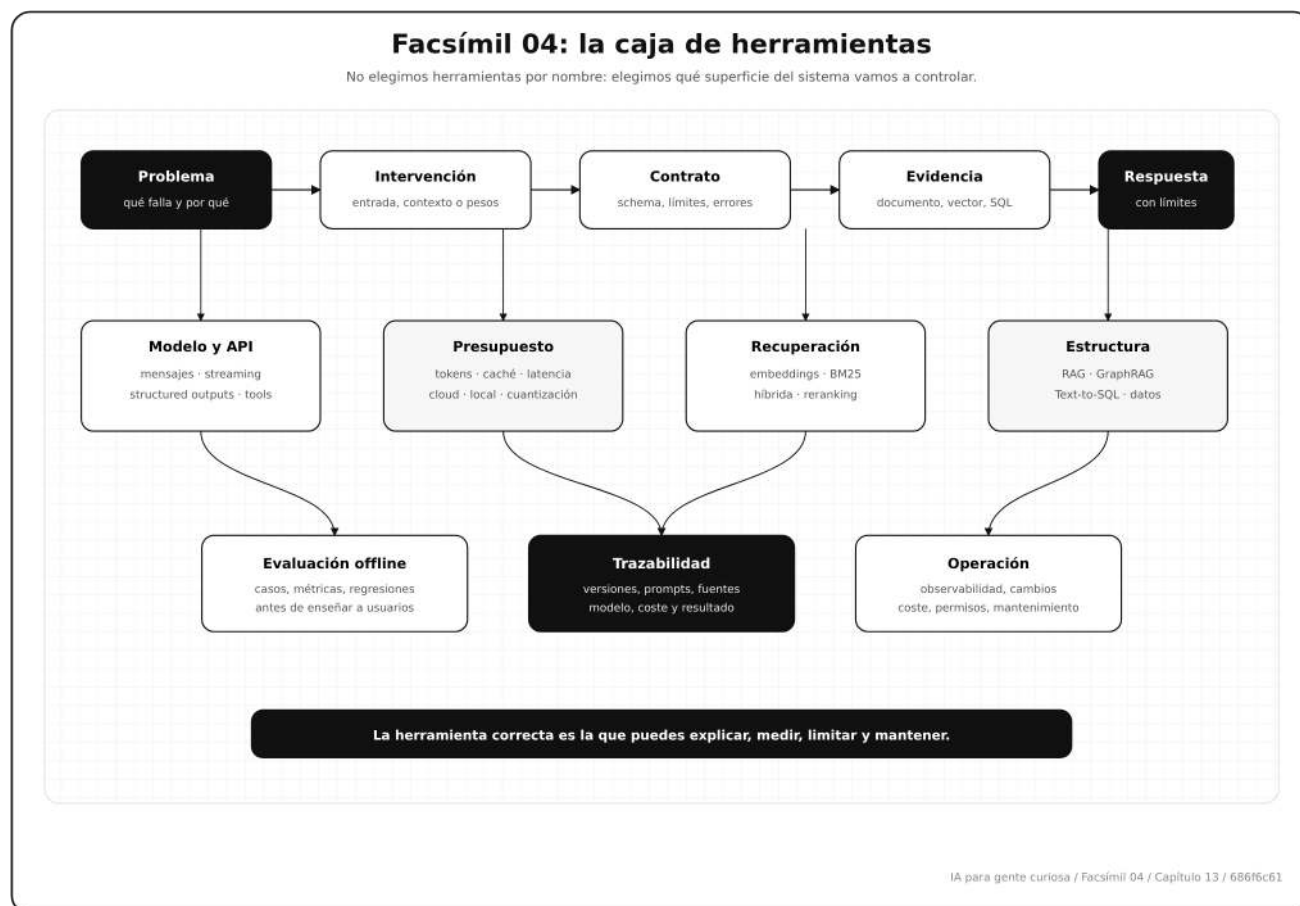
Cada intervención cambia una parte distinta del sistema. Si confundes la parte, puedes trabajar mucho y mejorar poco.

SI EL PROBLEMA ESTÁ EN...	INTERVENCIÓN NATURAL	QUÉ CAMBIA	QUÉ NO ARREGLA
Instrucción confusa	Prompt, ejemplos, formato	La entrada	Datos que el modelo no tiene.
Salida frágil	JSON schema, parser, validador	El contrato de salida	La calidad del contenido.
Conocimiento cambiante	RAG o herramienta de consulta	El contexto	El razonamiento sobre datos mal definidos.
Estado real	Tool/API/base de datos	La capacidad de actuar o consultar	Permisos, auditoría o validación.
Estilo repetido	Fine-tuning o LoRA	Parte del comportamiento	Información viva.
Coste o latencia	Modelo menor, caché, local, cuantización	Infraestructura y presupuesto	Preguntas mal formuladas.
Recuperación pobre	Embeddings, BM25, híbrida, reranking	Evidencia encontrada	Respuestas sin verificación.
Preguntas relacionales	GraphRAG o Text-to-SQL	Estructura consultable	Ambigüedad de negocio.

OpenAI documenta function calling como forma de describir herramientas con esquemas y recibir argumentos estructurados.¹ Las salidas estructuradas permiten exigir una forma de respuesta compatible con un esquema.² Esos mecanismos no hacen que el modelo “sepa más”; hacen que el sistema sea más gobernable.

La primera habilidad del facsímil es elegir la intervención pequeña que toca. La segunda es saber cuándo esa intervención ya no basta.

El mapa completo de la caja



El mapa tiene una idea central: las herramientas no están alineadas por glamour, sino por responsabilidad. Si una pieza no tiene contrato, métrica ni traza, todavía no es una pieza de ingeniería; es una promesa.

1. Elegir la intervención correcta

En el [capítulo 01](#) aprendimos a no empezar por la herramienta. Antes de decir RAG, fine-tuning, modelo local o agente, hay que diagnosticar el tipo de fallo.

Para recordar. Prompt y ejemplos cambian la entrada. Structured outputs cambian el contrato. RAG cambia el contexto. Una tool cambia la capacidad de consultar o actuar. Fine-tuning y LoRA cambian parte del comportamiento aprendido.³

Caso cercano. Una universidad quiere un asistente que responda dudas de matrícula. Si falla el tono, quizá basta con ejemplos. Si falla porque desconoce normas nuevas, RAG. Si debe consultar expediente, tool. Si debe devolver JSON para abrir un ticket, structured outputs. Si siempre corrige con una rúbrica propia, puede aparecer un ajuste.

Vuelve al capítulo 01 si: no puedes rellenar una tabla de “cambia / sirve cuando / no sirve para” sin mirar apuntes.

2. APIs: mensajes, eventos y contratos

En el capítulo 02 dejamos de tratar la API como una caja negra. Una llamada moderna tiene mensajes, roles, instrucciones, entradas multimodales, parámetros, streaming, tools, esquemas de salida y manejo de errores.

Para recordar. Una API buena no es solo “envío prompt y recibo texto”. Es un contrato entre aplicación y modelo. La aplicación decide qué datos manda, qué formato exige, qué herramientas declara, cómo valida y qué hace si la respuesta no cumple.

PIEZA	PREGUNTA QUE RESPONDE	ERROR TÍPICO
Mensajes	¿Quién dice qué y con qué prioridad?	Mezclar instrucciones de sistema con texto de usuario.
Parámetros	¿Cuánta variación permito?	Tocar temperatura sin evaluación.
Streaming	¿Cómo llega la respuesta?	No diseñar cancelación ni estados parciales.
Tools	¿Qué puede consultar el sistema?	Dejar que el modelo decida permisos.
Structured outputs	¿Qué forma debe tener la salida?	Parsear texto libre con expresiones frágiles.

Caso cercano. Un asistente administrativo clasifica correos y crea tareas. No debería “redactar algo parecido a JSON”. Debería devolver campos obligatorios, categorías permitidas, confianza y razón breve. La aplicación valida antes de crear nada.

Vuelve al capítulo 02 si: no puedes diseñar una llamada completa con entrada, salida estructurada, streaming opcional, tool declarada y errores previstos.

3. Tokens, contexto y caché

En el capítulo 03 aprendimos que el contexto no es gratis. Cada tabla, documento, ejemplo, historial y política consume tokens. El coste no vive solo en la salida.

Para recordar. El contexto es memoria temporal, no memoria perfecta. La caché puede reducir coste si reutilizas prefijos estables, pero no arregla un prompt lleno de ruido. La documentación de prompt caching trata precisamente esta idea: reutilizar segmentos repetidos para optimizar coste y latencia cuando el patrón lo permite.⁴

Identidad de coste (no es un teorema). El coste de un turno es, sencillamente, la suma de sus partes. No predice un número exacto: sirve para ver dónde se va el dinero y qué recortar. Es la misma lógica del coste total de propiedad (TCO): un coste real no es solo el precio visible, sino la suma de todos los componentes que hay que sostener.⁵

$$C_{\text{turno}} = C_{\text{input}} + C_{\text{output}} + C_{\text{herramientas}} + C_{\text{observabilidad}}$$

SÍMBOLO	QUÉ SIGNIFICA	QUÉ REVISAR
C_{input}	Coste de contexto enviado.	Esquema, docs, ejemplos, historial.
C_{output}	Coste de tokens generados.	Respuestas largas, razonamiento, formato.
$C_{\text{herramientas}}$	Consultas externas.	Bases, vector stores, APIs.
$C_{\text{observabilidad}}$	Trazas y evaluación.	Logs, métricas, almacenamiento.

Caso cercano. Meter todo el manual de 300 páginas en cada llamada puede funcionar en una demo y fracasar en coste. Un sistema serio recupera lo relevante, cachea instrucciones estables y mide si el contexto añadido mejora la respuesta.

Vuelve al capítulo 03 si: no puedes explicar por qué más contexto puede empeorar una respuesta.

4. Model cards y elección de modelos

En el [capítulo 04](#) aprendimos a leer una ficha de modelo como un expediente técnico. Nombre, licencia, parámetros, contexto, precisión, benchmarks, proveedor, plantilla de chat y formato de pesos no son adornos.

Para recordar. Una model card sirve para comparar promesas con restricciones. El trabajo original sobre model cards propuso documentar uso previsto, factores relevantes, métricas, datos de evaluación y consideraciones éticas de los modelos.⁶

TÉRMINO	QUÉ APORTA REALMENTE	PREGUNTA SANA
Parámetros totales	Capacidad almacenada aproximada.	¿Cuántos se activan por token?
Context length	Ventana máxima declarada.	¿Con qué coste y calidad real?
Tensor type	Precisión y formatos presentes.	¿Qué usa inferencia de verdad?
License	Condiciones de uso.	¿Permite mi caso concreto?
Benchmarks	Señales comparativas.	¿Se parecen a mi tarea?
Chat template	Contrato de entrada.	¿Estoy enviando mensajes como toca?

Caso cercano. Un modelo con ventana de contexto enorme puede parecer perfecto para contratos. Pero si recupera mal detalles de la mitad del documento, tarda demasiado o cuesta diez veces más, el número de contexto no resuelve el producto.

Vuelve al capítulo 04 si: no puedes explicar una model card sin convertirla en un catálogo de siglas.

5. Local, cuantización y dependencia operativa

En los capítulos 05 y 06 bajamos del modelo abstracto a la máquina real: memoria, VRAM, cuantización, runtimes, cloud, local, alquiler de GPUs, latencia, privacidad, coste y mantenimiento.

Para recordar. Local no significa gratis. Cloud no significa simple. Un modelo local necesita formato, runtime, memoria, CPU/GPU, plantilla, servidor, límites, observabilidad y actualización. Un modelo cloud necesita contrato de datos, precios, latencia, región, cuotas, cambios de versión y fallback.

DECISIÓN	GANAS	PAGAS
API cloud	Facilidad, escalado, modelos potentes.	Coste variable, dependencia, políticas de datos.
Local en portátil	Control y aprendizaje.	Menos potencia, setup, rendimiento limitado.
Servidor local	Control operativo.	GPU, mantenimiento, scheduling, monitorización.
GPU alquilada	Capacidad temporal.	Gestión de imágenes, datos, apagado y coste/hora.
Cuantización	Menos memoria y coste.	Posible pérdida de calidad y estabilidad.

Caso cercano. Si una herramienta interna procesa expedientes sensibles y tiene pocos usuarios, local puede ser razonable. Si atiende miles de solicitudes variables y necesita el mejor modelo disponible, cloud puede ganar. Si la carga es estable y grande, quizá compensa un servidor propio. No hay respuesta universal: hay presupuesto y restricciones.

Vuelve a los capítulos 05 y 06 si: no puedes estimar memoria, latencia y coste antes de instalar nada.

6. Embeddings y búsqueda semántica

En el capítulo 07 convertimos texto en vectores. Un embedding no es una etiqueta. Es una posición en un espacio de dimensiones donde podemos medir cercanía.

Para recordar. La dimensión de un embedding no es “una idea humana”, sino un eje numérico aprendido. El significado aparece por la combinación de muchos ejes. Comparar embeddings permite buscar por parecido semántico, no solo por palabras exactas.

La similitud coseno sigue siendo la fórmula mental básica. No es un invento reciente: es la medida del modelo de espacio vectorial, propuesto para recuperación de información en los años setenta, donde documentos y consultas se representan como vectores y su relevancia se estima por el ángulo entre ellos.⁷

$$\text{sim}(q, d) = \frac{q \cdot d}{\|q\| \|d\|}$$

SÍMBOLO	QUÉ SIGNIFICA	LECTURA
q	Vector de la consulta.	Lo que pregunta la persona en números.
d	Vector del documento.	Un fragmento convertido en números.
$q \cdot d$	Producto escalar.	Cuánto apuntan en dirección parecida.
$\ q\ , \ d\ $	Normas.	Tamaño de cada vector.

El coseno mide dirección, no magnitud: dos textos sobre el mismo tema apuntan al mismo sitio aunque uno sea más largo. Por eso, cuando los vectores están normalizados a norma 1, ordenar por coseno equivale a ordenar por producto escalar, y eso es justo lo que explotan los índices vectoriales para buscar rápido.

FAISS mostró cómo buscar de forma eficiente entre miles de millones de vectores en GPU.⁸ HNSW es una estructura de grafos aproximados muy usada para búsqueda vectorial eficiente.⁹

Caso cercano. “Cómo pedir vacaciones” puede recuperar “procedimiento de ausencia laboral” aunque no comparta palabras. Eso es buenísimo. Pero “parecido” no significa “suficiente”: todavía hay que validar fecha, versión, permisos y respuesta.

Vuelve al capítulo 07 si: no puedes explicar qué es una dimensión y por qué dos textos cercanos pueden no responder la misma pregunta.

7. Bases vectoriales y búsqueda híbrida

En el [capítulo 08](#) vimos que buscar no es solo guardar embeddings. Hay índices, filtros, metadatos, fragmentos, reranking y búsqueda híbrida.

Para recordar. BM25 sigue siendo fuerte para coincidencia léxica y términos exactos.¹⁰ Reciprocal Rank Fusion permite combinar rankings de distintas búsquedas sin convertir todo a una misma escala.¹¹

PIEZA	QUÉ APORTA	QUÉ PUEDE FALLAR
Chunking	Divide documentos recuperables.	Cortar contexto necesario.
Embeddings	Busca por significado.	Perder números, códigos o nombres exactos.
BM25	Busca por términos.	No captar paráfrasis.
Filtros	Acotan por permisos, fecha, tipo.	Filtrar demasiado o tarde.
Reranking	Reordena candidatos.	Añadir latencia sin mejorar.
Metadatos	Dan contexto operativo.	Estar incompletos o desactualizados.

Caso cercano. Si buscas “artículo 17.3” necesitas texto exacto. Si buscas “cómo se solicita una revisión”, necesitas semántica. Si necesitas ambas cosas, híbrida.

Vuelve al capítulo 08 si: no puedes diseñar un índice con texto, metadatos, filtros y estrategia de recuperación.

8. RAG básico: evidencia antes que respuesta

En el [capítulo 09](#) construimos RAG: recuperar información externa, pasarla al modelo y responder con evidencia. RAG no es “hacer que el modelo sepa más”. Es darle contexto verificable en el momento de responder.

Para recordar. Un RAG mínimo necesita colección, limpieza, chunking, embeddings o búsqueda léxica, recuperación, contexto, generación, citas y abstención. El trabajo original de RAG combinó recuperación con generación para tareas intensivas en conocimiento.¹²

Caso cercano. Un asistente de normativa responde mejor si cita el documento vigente. Pero si recupera un párrafo antiguo, la redacción puede ser impecable y la respuesta falsa. RAG no elimina el problema: lo hace medible.

Vuelve al capítulo 09 si: no puedes explicar la diferencia entre entrenar conocimiento y recuperar evidencia.

9. Evaluar RAG y no fiarse de la demo

En el [capítulo 10](#) dejamos claro que una demo que responde bien tres veces no es una evaluación. Hay que medir retrieval, contexto, groundedness, abstención, latencia, coste y regresiones.

Para recordar. Evaluar RAG tiene varias capas:

CAPA	PREGUNTA
Recuperación	¿Aparece la fuente correcta entre los candidatos?
Contexto	¿Lo recuperado contiene evidencia suficiente?
Generación	¿La respuesta se apoya en esa evidencia?
Abstención	¿Sabe parar cuando no hay base?
Operación	¿Cuánto cuesta y tarda en casos reales?

RAGAS popularizó métricas orientadas a RAG como faithfulness, answer relevancy, context precision y context recall.¹³

Caso cercano. Si tu sistema responde “no lo sé” ante una pregunta que sí tiene respuesta, falta recall. Si responde con seguridad sin evidencia, falta abstención o groundedness. Si acierta pero tarda doce segundos, falta operación.

Vuelve al capítulo 10 si: no puedes proponer un dataset de evaluación con preguntas, fuentes esperadas, respuestas aceptables y casos donde debe abstenerse.

10. Agentic RAG y GraphRAG: complicar con permiso

En el [capítulo 11](#) vimos cuándo un RAG fijo se queda corto: preguntas compuestas, rutas de búsqueda, fuentes múltiples, relaciones globales o necesidad de comprobar si la evidencia basta.

Para recordar. Agentic RAG añade decisiones de flujo: descomponer, elegir fuente, recuperar de nuevo, revisar evidencia, usar herramientas. GraphRAG añade estructura de entidades y relaciones para responder preguntas locales o globales sobre corpus complejos. Microsoft GraphRAG propuso construir grafos de entidades y resúmenes de comunidades para preguntas de comprensión global.¹⁴

SI NECESITAS...	PUEDE TENER SENTIDO
Comparar varias fuentes	Query decomposition o multi-query.
Buscar con distintas estrategias	Router controlado.
Entender relaciones entre entidades	GraphRAG o grafo de conocimiento.
Revisar evidencia insuficiente	Corrective RAG o validación previa.
Limitar coste	Presupuesto de pasos y trazas.

Caso cercano. “Qué departamentos aparecen conectados a quejas sobre becas y retrasos de pago” no se resuelve igual que “cuál es el plazo de becas”. La primera pide patrones y relaciones. La segunda pide recuperar una fuente concreta.

Vuelve al capítulo 11 si: no puedes explicar qué complejidad pagas cuando añades pasos agentic.

11. Text-to-SQL y herramientas de datos

En el [capítulo 12](#) cruzamos otra frontera: preguntas que no se contestan con documentos, sino con datos. Ahí no basta con recuperar párrafos. Hay que consultar tablas, esquemas, permisos y métricas.

Para recordar. Text-to-SQL traduce una pregunta humana a SQL controlado. No es acceso libre a la base de datos. Es una cadena: intención, esquema, semántica, SQL candidato, validación, plan, ejecución limitada, resultado y traza. Spider ayudó a medir Text-to-SQL con bases nuevas y consultas complejas.¹⁵

PREGUNTA	RIESGO SI LO SIMPLIFICAS
“Alumnos con pagos pendientes”	Contar pagos en vez de alumnos.
“Ingresos de marzo”	Usar fecha de creación en vez de fecha de pago.
“Top campus”	No definir si top es por alumnos, importe o incidencias.
“Datos por titulación”	Exponer columnas innecesarias.

Caso cercano. Un `JOIN` mal hecho puede duplicar filas y devolver una cifra convincente pero falsa. En Text-to-SQL, la sintaxis correcta es solo el comienzo. La semántica y la cardinalidad mandan.

Vuelve al capítulo 12 si: no puedes explicar por qué `COUNT(*)` y `COUNT(DISTINCT alumno_id)` responden preguntas distintas.

La matriz de decisión final

Esta matriz es una forma de hacer arquitectura sin posturoeo. Primero identifica el tipo de necesidad; después elige la herramienta más pequeña que cubre el caso.

NECESIDAD REAL	PRIMER INTENTO RAZONABLE	ESCALARÍA A...	SEÑAL DE QUE TE ESTÁS PASANDO
Mejor tono o formato	Prompt + ejemplos	Structured outputs	Cambiar pesos sin evaluar prompts.
JSON fiable	Schema + validador	Tool contract	Parsear texto libre.
Responder sobre documentos	RAG básico	Agentic RAG	Usar agente para una FAQ simple.
Encontrar documentos	Híbrida + filtros	Reranking	Embeddings sin metadatos.
Consultar datos vivos	Tool o Text-to-SQL	Semantic layer	Dejar SQL libre sin permisos.
Reducir coste	Modelo menor + caché	Cuantización/l ocal	Recortar contexto sin medir calidad.
Controlar despliegue	API cloud con contratos	Servidor propio	Montar GPUs sin carga estable.
Dominio repetido	Plantillas + evals	LoRA/fine-tuning	Ajustar pesos para conocimiento cambiante.
Relaciones globales	RAG + metadatos	GraphRAG	Construir grafo sin preguntas de competencia.

La matriz no decide por ti. Te obliga a decir qué estás comprando con cada capa nueva. Esa frase, “qué estoy comprando”, es una de las mejores defensas contra arquitecturas hinchadas.

Mini casos con solución

Una buena recapitulación no solo pregunta “¿lo entendiste?”. Te pone delante situaciones pequeñas y obliga a decidir. Estas son decisiones de bolsillo, pero condensan casi todo el facsímil.

CASO	DECISIÓN RAZONABLE	POR QUÉ
Una FAQ interna cambia cada mes y el asistente responde con información antigua.	RAG básico con fuentes versionadas, citas y evaluación.	El problema es conocimiento cambiante, no estilo del modelo.
Un clasificador de tickets devuelve texto libre y rompe el flujo de soporte.	Structured outputs con catálogo cerrado y validador.	El fallo está en el contrato de salida, no en saber más contenido.
Una herramienta debe decir cuántos pagos pendientes hay por campus.	Tool de datos o Text-to-SQL con permisos y métricas definidas.	La respuesta vive en tablas, no en documentos parecidos.
Un RAG recupera fragmentos relacionados pero no la norma exacta.	Búsqueda híbrida, filtros por fecha/tipo y quizá reranking.	El problema está en retrieval, no en generación.
El sistema acierta pero tarda demasiado y cuesta demasiado por consulta.	Reducir contexto, cachear prefijos, probar modelo menor o cuantización.	El cuello está en presupuesto operativo.
El equipo quiere fine-tuning para añadir una política que cambia cada semana.	No ajustar pesos; usar RAG o tool.	El conocimiento vivo debe vivir fuera del modelo.
La pregunta exige comparar documentos, entidades y patrones de varias fuentes.	Agentic RAG controlado o GraphRAG si las relaciones importan.	La complejidad se justifica si la pregunta necesita pasos o estructura.

Ahora lo mismo, pero con tres decisiones desarrolladas.

Caso 1: “Tenemos una FAQ interna que cambia cada mes”.

No empezaría por fine-tuning. Cambiar pesos para información viva suele crear deuda: cada cambio exige nuevo ajuste, evaluación y despliegue. Empezaría por RAG básico: documentos versionados, chunking razonable, búsqueda híbrida si hay códigos o artículos, citas visibles y casos donde el sistema debe abstenerse. La métrica mínima sería: ¿recupera la fuente vigente?, ¿cita bien?, ¿responde solo con esa evidencia?

Caso 2: “Queremos consultar pagos pendientes”.

No usaría RAG como primera opción. Un documento puede explicar qué es un pago pendiente, pero el número está en una base de datos. Haría una tool o Text-to-SQL

controlado: rol de usuario, tablas permitidas, métrica definida, SQL validado, límite de filas, query plan y traza. La pregunta clave sería: ¿estoy contando pagos, alumnos o expedientes?

Caso 3: “El RAG responde bonito pero cita mal”.

No tocaría temperatura ni modelo todavía. Mediría recuperación antes: recall de fuentes esperadas, precisión de contexto, frescura de documentos, metadatos, filtros y reranking. Si la evidencia correcta no entra en el contexto, el generador no puede arreglarlo de forma fiable. Si la evidencia entra pero la respuesta no se apoya en ella, entonces revisaría prompt, formato de citas y evaluación de groundedness.

La idea de estos casos no es memorizar respuestas. Es aprender el gesto mental: diagnosticar la pieza que falla antes de cambiar la herramienta.

Cuándo no usar IA

Una caja de herramientas sería también incluye la opción de no sacar ninguna herramienta de IA. Hay problemas que se resuelven mejor con una regla, una consulta, un formulario, una validación clásica o una interfaz más clara.

SITUACIÓN	MEJOR PRIMERA OPCIÓN	POR QUÉ
Categorías cerradas y reglas simples.	if , reglas o tabla de decisión.	Más barato, explicable y determinista.
Cálculo exacto.	Código o SQL.	Un modelo generativo no debería inventar aritmética.
Flujo con permisos estrictos.	API con roles y validación.	El permiso no debe depender de texto.
Información ya estructurada.	Consulta directa o dashboard.	No hace falta traducir a lenguaje natural si el informe basta.
Tarea muy sensible y sin evaluación.	Esperar, medir o rediseñar.	No se despliega lo que no se puede comprobar.
Problema mal definido.	Taller de requisitos.	La IA no arregla una pregunta que nadie entiende.

Esto no va contra la IA. Va a favor de usarla bien. A veces la decisión más profesional es decir: aquí no hace falta un modelo; hace falta una regla clara, una tabla limpia o una conversación de producto.

El cálculo que deberías hacer antes de construir

Antes de escribir código, puedes estimar si una solución tiene sentido.

Identidad de decisión (cualitativa, no es un teorema). La utilidad neta es lo que queda al restar los costes al valor que aporta. Es una disciplina mental para discutir, no una fórmula para sacar decimales. Reproduce la estructura del análisis coste-beneficio: una intervención merece la pena cuando el beneficio neto (beneficios menos costes, incluido el riesgo) es positivo frente a la alternativa.¹⁶

$$U = Q - (C_{\text{tokens}} + C_{\text{latencia}} + C_{\text{mantenimiento}} + C_{\text{riesgo}})$$

SÍMBOLO	QUÉ SIGNIFICA	CÓMO SE OBSERVA
U	Utilidad neta de la solución.	Valor práctico después de costes.
Q	Calidad útil para la tarea.	Aciertos, groundedness, satisfacción, ahorro real.
C_{tokens}	Coste de entrada y salida.	Tokens, caché, modelo elegido.
C_{latencia}	Tiempo que tarda.	TTFT, P95, tiempo total.
$C_{\text{mantenimiento}}$	Trabajo de sostenerlo.	Datos, índices, evals, versiones.
C_{riesgo}	Impacto de fallos.	Permisos, datos sensibles, decisiones críticas.

No es una fórmula para sacar decimales. Es una disciplina mental. Si una herramienta sube Q un poco pero dispara mantenimiento y latencia, quizá no compensa. Si baja coste sin destruir calidad, merece atención. Si reduce riesgo aunque cueste algo más, puede ser la decisión correcta.

El lugar donde una demo se vuelve discutible

Una demo sirve para ver una posibilidad. Un laboratorio sirve para decidir si esa posibilidad aguanta un poco de realidad.

En este facsímil hemos hablado de APIs, modelos locales, tokens, costes, embeddings, RAG, evaluación, GraphRAG y Text-to-SQL. Todo eso puede quedarse en palabras si no lo llevamos a una mesa de trabajo mínima: datos pequeños, código ejecutable, métricas claras, trazas legibles y una decisión final.

Esta segunda parte del cierre es ese puente. No vamos a montar una plataforma industrial. Vamos a construir lo justo para que otra persona pueda ejecutar, revisar, criticar y mejorar lo que hicimos. Ese es el gesto profesional: no pedir confianza; dejar evidencia.

Qué no es un laboratorio

Un laboratorio no es un notebook que funciona una vez en tu máquina y queda abandonado. Tampoco es una captura bonita de una respuesta acertada. Y no es una colección de librerías instaladas sin una pregunta clara.

Un laboratorio tampoco sustituye a producción. En producción aparecen permisos reales, usuarios, colas, coste variable, cambios de datos y mantenimiento. El laboratorio no pretende resolver todo eso. Pretende descubrir antes qué merece pasar a una fase más seria.

Si no sabes qué pregunta estás probando, qué métrica mirarás y qué harás si sale mal, todavía no tienes laboratorio. Tienes una exploración. Puede ser útil, pero no permite decidir.

Qué sí debería dejar

Un laboratorio mínimo debe dejar cinco artefactos:

ARTEFACTO	QUÉ CONTIENE	POR QUÉ IMPORTA
Dataset	Casos de prueba con respuesta o fuente esperada.	Permite repetir la evaluación.
Runner	Código que ejecuta el sistema sobre esos casos.	Evita evaluar a mano caso por caso.
Métricas	Números que resumen el comportamiento.	Permite comparar versiones.
Trazas	Pasos internos de cada ejecución.	Permite depurar por qué falló.
Decisión	Pasar, parar, cambiar o medir más.	Evita terminar con “parece que va bien”.

Los notebooks son útiles porque mezclan explicación, código y resultados. El formato Jupyter se basa en documentos JSON con celdas, salidas y metadatos, lo que permite guardar no solo código, sino también el contexto de ejecución.¹⁷ Esa flexibilidad es estupenda para aprender, pero también exige disciplina: fijar datos, ordenar celdas, limpiar salidas innecesarias y convertir lo aprendido en scripts o tests cuando el experimento empieza a importar.

En observabilidad, OpenTelemetry describe una traza como una operación formada por spans, donde cada span representa una unidad de trabajo con contexto y atributos.¹⁸ Nosotros haremos una versión casera: una lista de pasos con nombre, entrada, salida y metadatos. No será una herramienta de producción, pero enseñará la forma mental correcta.

Estado de herramientas con fecha de corte

Fecha de corte: 26 de mayo de 2026.

Fuentes consultadas ese día: documentación de Jupyter nbformat, OpenTelemetry Tracing API, OpenAI Graders, Ragas metrics, LangSmith RAG evaluation y Arize Phoenix evaluation.

OpenAI documenta graders como evaluadores usados en evals y fine-tuning, incluyendo validación de graders y ejemplos de evaluadores basados en modelos.¹⁹ Ragas organiza métricas para aplicaciones RAG, entre ellas context precision y faithfulness.²⁰ LangSmith estructura la evaluación de RAG alrededor de corrección, relevancia, groundedness y relevancia de documentos.²¹

Phoenix separa evaluación de retrieval y evaluación de respuesta, y permite trabajar con trazas para analizar qué documentos se recuperaron y cómo se generó la respuesta.²² Su documentación de evaluación distingue evaluadores deterministas y evaluadores con modelo, y los presenta como forma de detectar regresiones y comparar cambios.²³

La lección estable no depende de una marca concreta: una evaluación útil separa dataset, ejecución, métrica, trazas y decisión.

Cómo medir de verdad: métricas, jueces y comparación

Métricas como Hit@1 y MRR bastan para empezar, pero un ingeniero de IA necesita un repertorio más amplio y, sobre todo, saber cuándo cada una miente. Esta sección reúne lo que de verdad usa la evaluación seria de sistemas con recuperación y generación.

Cuando hay varios documentos relevantes: nDCG

Hit@1 y MRR suponen una sola fuente correcta. En cuanto una pregunta admite varios documentos relevantes con distinta importancia, la métrica estándar es el nDCG (*normalized discounted cumulative gain*), que premia colocar lo más relevante arriba y aplica un descuento logarítmico a las posiciones siguientes.²⁴

$$\text{DCG@k} = \sum_{i=1}^k \frac{2^{rel_i} - 1}{\log_2(i + 1)} \quad \text{nDCG@k} = \frac{\text{DCG@k}}{\text{IDCG@k}}$$

SÍMBOLO	QUÉ SIGNIFICA	LECTURA
rel_i	Relevancia del documento en la posición i	0 = irrelevante, 2 = muy relevante.
$\log_2(i + 1)$	Descuento por posición	Cuanto más abajo, menos vale.
IDCG@k	DCG del ranking ideal	El mejor orden posible.
nDCG@k	DCG normalizado al rango [0, 1]	1.0 = orden perfecto.

Tres formas de puntuar el mismo ranking

Cinco documentos recuperados. Hit@1 y MRR dan 1.0; nDCG baja porque hay relevantes mal colocados.

1 **matricula_vigente**
muy relevante (rel 2) ✓

2 **calendario_antiguo**
no relevante (rel 0)

3 **tasas_matricula**
relevante (rel 1) ✓

4 **noticia_evento**
no relevante (rel 0)

5 **requisitos_matricula**
relevante (rel 1) ✓

Hit@1 = 1.0
Solo mira el puesto 1. El primero es relevante: aprueba.

MRR = 1.0
Premia el primer acierto. Aparece en la posición 1.

nDCG@5 = 0.94
Mira todo el orden con descuento. Penaliza que los relevantes 3 y 5 queden bajo documentos irrelevantes.

La misma lista, tres veredictos: la métrica que elijas decide qué consideras «bueno».

IA para gente curiosa / Facsímil 04 / Capítulo 13 / 686f6c61

La figura muestra el mismo ranking puntuado de tres formas. Hit@1 y MRR se quedan en 1.0 porque solo miran el primer acierto; nDCG baja a 0.94 porque dos documentos relevantes quedaron por debajo de documentos irrelevantes. Elegir métrica es elegir qué fallos te importan.

Qué es «groundedness» exactamente

Usamos «groundedness» (o «faithfulness») como si fuera obvio, pero tiene una definición operativa: una respuesta está fundamentada si **cada afirmación** que contiene puede deducirse de la evidencia recuperada. RAGAS la calcula descomponiendo la respuesta en afirmaciones atómicas y midiendo qué fracción está respaldada por el contexto.²⁵

$$\text{faithfulness} = \frac{\text{afirmaciones respaldadas por el contexto}}{\text{afirmaciones totales de la respuesta}}$$

Una respuesta puede ser correcta y aun así poco fundamentada: acertó por conocimiento del modelo, no por la evidencia. En un sistema serio eso es un fallo, porque si mañana cambia el documento la respuesta deja de ser fiable sin avisar. Por eso groundedness se mide aparte de la corrección.

Cuando no hay métrica exacta: el modelo como juez

Muchas cualidades (claridad, tono, si una respuesta «responde de verdad») no tienen fórmula. La práctica habitual es usar otro modelo como evaluador: se le dan la respuesta, el contexto y una rúbrica, y devuelve una puntuación con su razón. Es la técnica de *LLM-as-judge*, que correlaciona razonablemente con el juicio humano cuando la rúbrica es clara.²⁶ El método G-Eval formalizó pedir al modelo una puntuación guiada por criterios y mostró mejor alineación con humanos que las métricas automáticas clásicas.²⁷

Pero un juez-modelo tiene sesgos conocidos que hay que controlar:

SESGO	QUÉ HACE	MITIGACIÓN
Posición	Prefiere la primera (o la segunda) opción en comparaciones.	Alternar el orden y promediar.
Verbosidad	Premia respuestas largas aunque no sean mejores.	Penalizar longitud; pedir concisión en la rúbrica.
Autopreferencia	Favorece textos de su propia familia de modelos.	Usar un juez distinto del modelo evaluado.
Formato	Se deja llevar por listas, negritas o seguridad aparente.	Rúbrica que separa fondo de forma.



La regla: el juez-modelo es útil para escalar la evaluación, no para tener la última palabra. Calíbralo contra un pequeño set anotado a mano y vuelve a revisarlo cuando cambies de modelo.

¿Mi cambio ayudó de verdad? Comparar dos variantes

Medir una variante no basta; casi siempre quieres saber si B es mejor que A. Con datasets pequeños, una diferencia de «0.80 frente a 0.85» puede ser ruido. Para datos emparejados (los mismos casos en A y B), la prueba estándar es la de McNemar, que mira solo los casos donde una variante acierta y la otra falla.²⁸ Cuando la métrica no es binaria, el bootstrap (remuestrear los casos y mirar la distribución de la diferencia) da un intervalo de confianza sin suponer una forma concreta.²⁹

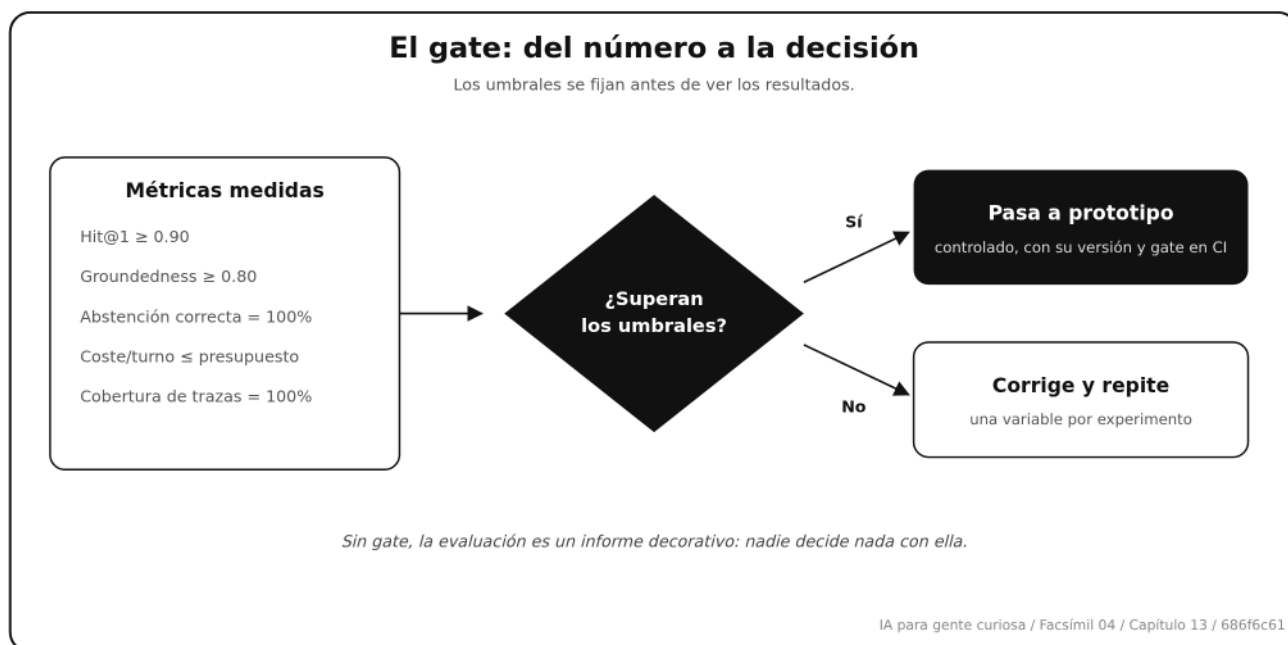
La idea práctica: no celebres una mejora sin preguntarte si sobreviviría a repetir el experimento. Una diferencia pequeña sobre pocos casos casi nunca sobrevive.

El enemigo silencioso: contaminación del set

Una evaluación solo vale si sus casos son nuevos para el sistema. Si las preguntas (o sus respuestas) estaban en los datos de entrenamiento o ya venían en el contexto, la métrica mide memoria, no capacidad. Es un problema real y creciente con modelos entrenados sobre casi toda la web.³⁰ En tu laboratorio: usa casos propios, fecha el dataset y desconfía de un 100% que llega demasiado fácil.

Del número a la decisión: el gate

Nada de lo anterior sirve si no se traduce en una decisión. Un *gate* convierte métricas en un sí o no automático: define umbrales mínimos **antes** de mirar los resultados y deja que la evaluación decida si una variante avanza o vuelve a la mesa de trabajo. Definir el umbral después de ver el número es engañarse.



Presupuesto con números: latencia y capacidad

Una última pieza que separa demo de sistema: estimar coste y latencia antes de prometer nada. La latencia de una respuesta generativa se descompone en el tiempo hasta el primer token (TTFT) y el tiempo entre tokens (TPOT):

$$L \approx \text{TTFT} + (N_{\text{out}} - 1) \cdot \text{TPOT}$$

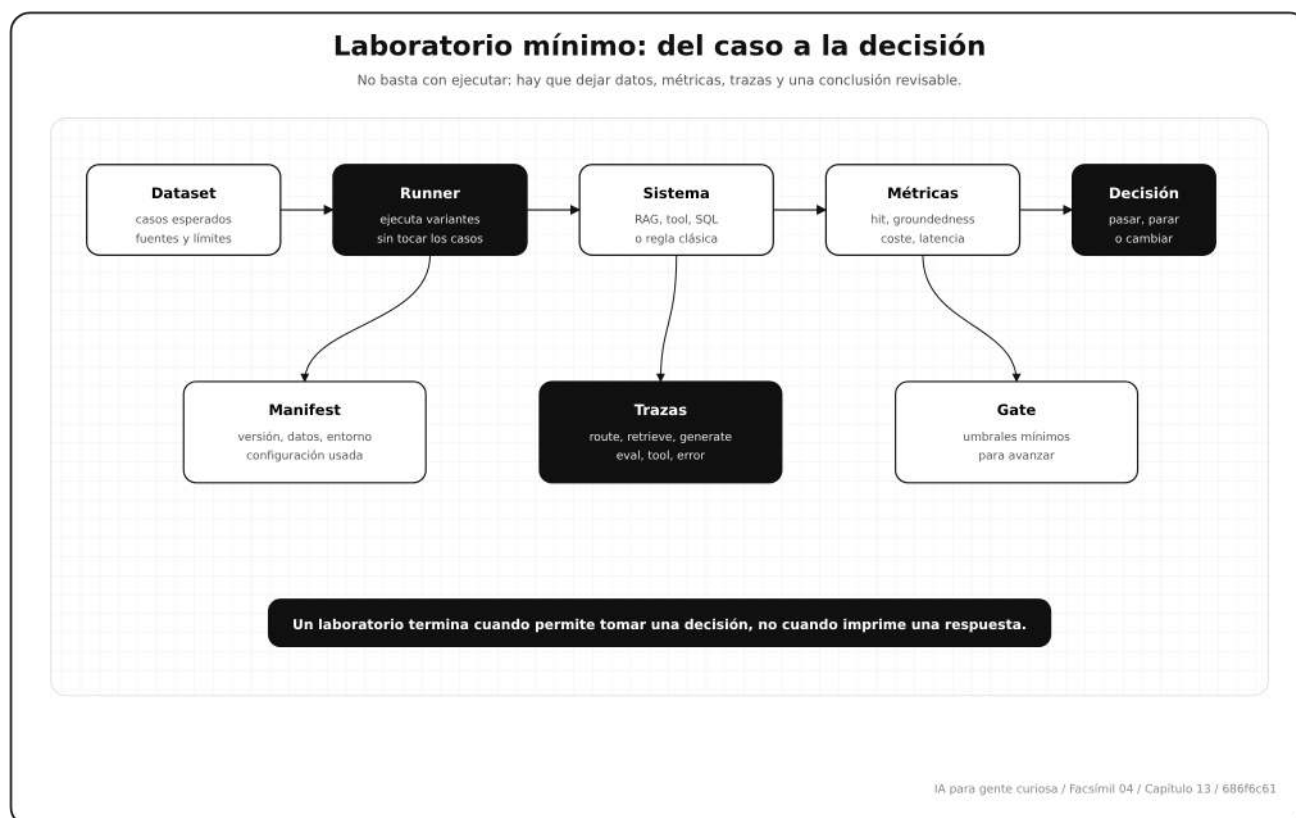
Para dimensionar capacidad sirve la ley de Little, un resultado clásico de teoría de colas: el número medio de peticiones dentro del sistema es la tasa de llegada por el tiempo medio que pasa cada una.³¹

$$L = \lambda \cdot W$$

SÍMBOLO	QUÉ SIGNIFICA	EJEMPLO
L	Peticiones simultáneas en el sistema	Concurrencia media.
λ	Tasa de llegada	5 peticiones por segundo.
W	Tiempo medio en el sistema	2 s por petición.

Con $\lambda = 5$ y $W = 2$, el sistema sostiene de media $L = 10$ peticiones a la vez. Si tu servidor no aguanta diez en vuelo, o bajas la latencia W o se forma cola y la latencia percibida sube. Estimar esto antes evita la sorpresa de «funcionaba en la demo y se cae con diez usuarios».

Anatomía de un laboratorio mínimo



La figura puede leerse de izquierda a derecha: parto de casos, ejecuto variantes, observo resultados y decido. Debajo aparecen las tres piezas que suelen faltar en las demos: manifest, trazas y gate.

En el día a día

En el trabajo real, este cierre no aparece como un examen, sino como una conversación. Alguien propone «metamos un agente», «conectemos RAG» o «esto lo resuelve un modelo». La caja de herramientas que has recorrido sirve para responder con calma: qué problema hay, qué intervención cambia algo, qué contrato hace falta, qué métrica lo prueba y qué pasa si falla.

El laboratorio es el otro lado de esa conversación. Cuando alguien afirma que «funciona», la pregunta profesional no es «¿lo has visto?», sino «¿puedo ejecutarlo, medirlo y leer la traza?». En el día a día eso se traduce en gestos pequeños: un dataset de diez casos, un runner que no toca los casos, una métrica por capa y una decisión escrita. No es burocracia; es lo que separa una demo de algo que otra persona puede sostener.

Verás el patrón en revisiones de arquitectura, en estimaciones de coste antes de construir, en la decisión de no usar IA cuando una regla basta y en la defensa de un sistema delante de alguien que pregunta «¿y si mañana cambia el dato?».

Por qué debería importarte

Porque la diferencia entre un equipo que improvisa y uno que decide está casi siempre aquí. El primero acumula demos que nadie puede reproducir; el segundo deja evidencia: contratos, trazas, evaluaciones y una conclusión defendible. Con el tiempo, el segundo avanza más rápido, no más despacio, porque no rehace lo que no entiende.

También importa para ti como persona curiosa. Saber leer un sistema (ver dónde falla, qué intervención toca, qué cuesta) es una forma de autonomía: la IA deja de ser magia y pasa a ser algo que puedes interrogar. Y cuando el siguiente facsímil hable de agentes y orquestación, esa costumbre de exigir límites, permisos y trazas será justo lo que evite que un sistema más capaz se vuelva, simplemente, más difícil de depurar.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Empezar por la herramienta	«Metamos RAG» no dice qué fallo estás resolviendo.	Escribir primero qué cambia: entrada, contexto, herramienta, pesos o despliegue.
Confundir demo con sistema	Una respuesta buena no prueba permisos, coste, latencia ni regresiones.	Exigir dataset mínimo, trazas y casos donde debe abstenerse.
Confundir notebook con laboratorio	Un notebook puede ejecutar una vez y no dejar decisión ni reproducibilidad.	Añadir dataset, métricas, manifest y conclusión.
Medir solo la respuesta final	No sabes si falló retrieval, routing, herramienta o generación.	Medir por capas y guardar spans.
Meter más contexto sin medir	Puede subir coste y ruido sin mejorar groundedness.	Medir recall, precisión de contexto y calidad de respuesta por separado.
Cambiar prompt y dataset a la vez	Si mejora, no sabes qué lo causó.	Fijar dataset y cambiar una variable por experimento.
Complicar antes de cerrar lo básico	Agentic RAG o GraphRAG pueden esconder problemas de chunking, filtros o definición.	Probar primero el flujo más simple que pueda evaluarse.
No poner gate	La evaluación queda como un informe decorativo.	Definir umbrales mínimos antes de mirar resultados.
Olvidar mantenimiento	Índices, prompts, modelos, esquemas y datos cambian.	Nombrar propietario, versión y prueba de regresión de cada pieza.
No diseñar salida de fallo	El sistema acaba respondiendo incluso cuando no sabe.	Definir abstención, aclaración y escalado antes de producción.
No contemplar la opción sin IA	Algunas tareas se resuelven mejor con reglas, SQL o una interfaz más clara.	Preguntar siempre qué gana el modelo frente a una solución clásica.

Cuadernos para practicar

A lo largo del facsímil has montado y medido sistemas con herramientas; estos cuadernos te dejan tocar las piezas sueltas. Son *notebooks* que se abren en Google Colab —gratis, en el navegador— o te puedes descargar. Cada uno, explicado paso a paso, con salidas reales, y

anclado al capítulo del que sale.

Tokens, contexto y coste: la factura de un LLM

Qué prácticas: contar tokens reales y estimar lo que cuesta procesar un volumen de trabajo. **Dónde encaja:** capítulo 3 (tokens, coste, contexto y caché). **Qué necesitas:** un navegador. Corre en CPU; sin claves.

Mides en tokens reales lo que procesas y pagas. Primero ves que un token no es una palabra ni una letra: «antidisestablishmentarianism» son 6 tokens, tres emojis son 7, y el español gasta más tokens por palabra que el inglés (y por tanto más dinero). Luego calculas el caso del día a día: clasificar diez mil tickets de soporte sale por unos **0,75 dólares** con precios de ejemplo, y ves cuánto ahorra cachear las instrucciones fijas. Por último traduces la ventana de contexto a algo tangible: 128.000 tokens son unas **176 páginas**.

La primera sorpresa de poner un LLM en producción es la factura. Aquí aprendes a estimarla antes de gastar.

► [Abrir en Google Colab](#)

↓ [Descargar el cuaderno \(.ipynb\)](#)

Salidas estructuradas: que el modelo rellene tu formulario

Qué prácticas: exigir al modelo un formato fijo y validarlo automáticamente. **Dónde encaja:** capítulo 2 (APIs de modelos: mensajes, *streaming* y salidas estructuradas). **Qué necesitas:** un navegador. Corre en CPU; sin claves (se simulan las respuestas).

Dejas de tratar la respuesta del modelo como un texto del que sacar datos a mano y le exiges un esquema: una factura con proveedor, importe positivo, fecha y concepto. Una respuesta buena se valida y se convierte en un dato con tipos —la fecha es una fecha de verdad, con la que ya puedes operar—. Una mala (el importe como texto «cuarenta», un importe negativo, un campo que falta) salta en el sitio, con un mensaje claro de qué falló, en vez de colarse en tu base de datos y reventar tres pasos después. Y montas el bucle de reintento que usan los sistemas serios: devolverle el error al modelo para que corrija.

Si vas a meter lo que dice el modelo en una base de datos o en una decisión, no puedes permitirte un «más o menos».

► [Abrir en Google Colab](#)

↓ [Descargar el cuaderno \(.ipynb\)](#)

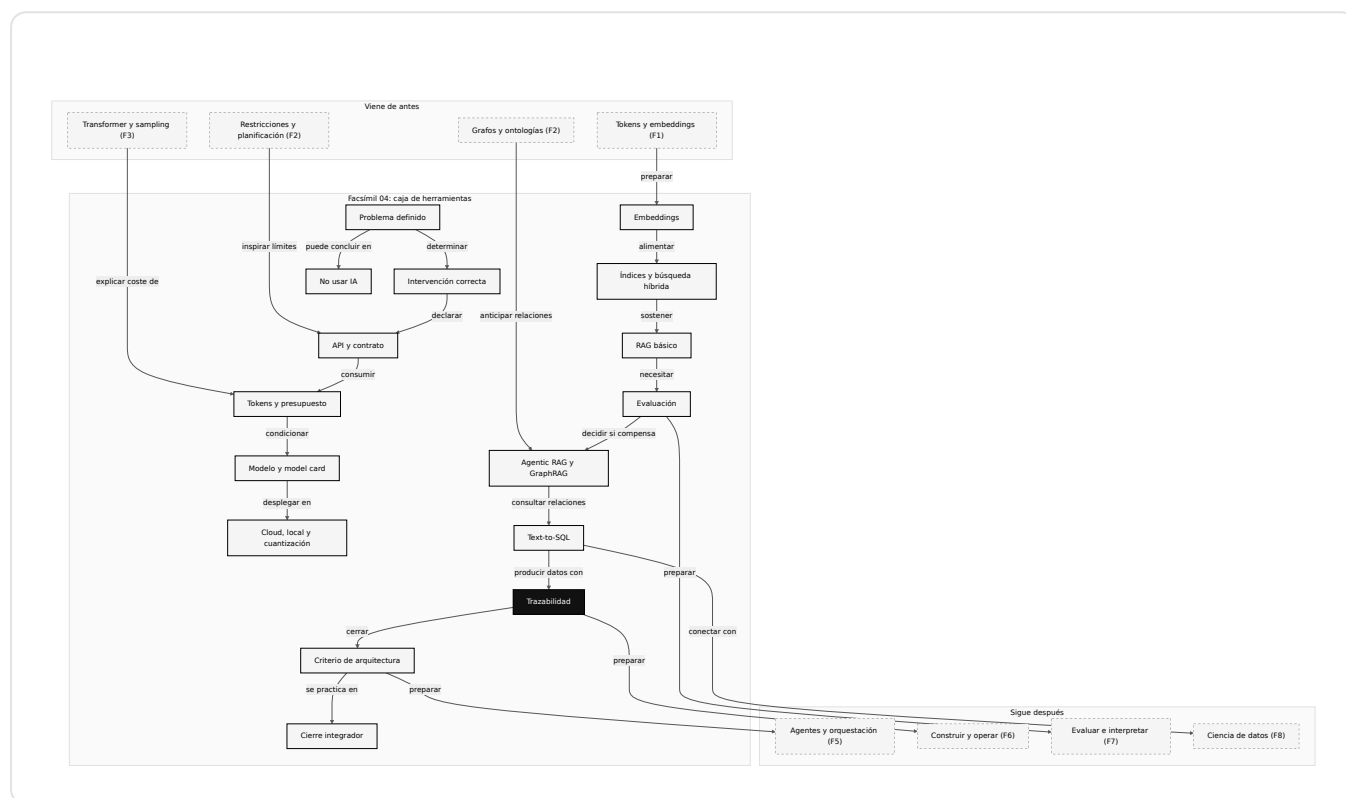
Del cierre al siguiente facsímil

El cierre natural de este facsímil no es “ya sabemos usar herramientas”. Es más preciso: ya sabemos **poner herramientas bajo control**.

El criterio de este cierre debería poder demostrarse con algo concreto: una evaluación reproducible, una traza que explique qué ocurrió, una decisión de arquitectura defendible y una salida que no dependa de buena suerte. Si una solución no deja rastro, no tiene contrato y no se puede evaluar, todavía pertenece al terreno de la demo.

El facsímil 05 dará el siguiente paso: agentes y orquestación. Ahí las herramientas dejan de ser piezas aisladas y empiezan a coordinarse. Por eso este cierre insiste tanto en límites, permisos, trazas, costes y abstención. Un agente sin esas piezas no es más capaz: solo es más difícil de depurar.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Intervención	Cambio concreto sobre entrada, contexto, herramienta, pesos o infraestructura.
Contrato	Especificación de entrada, salida, límites, errores y trazas.
Superficie de control	Lugar del sistema donde puedes limitar, validar o medir.
Presupuesto operativo	Límite de coste, tokens, latencia, memoria, pasos o consultas.
Evidencia	Información recuperada o calculada que sostiene una respuesta.
Abstención	Decisión de no responder cuando no hay base suficiente.
Trazabilidad	Capacidad de reconstruir cómo se produjo una respuesta.
Traza	Registro estructurado de los pasos que produjeron una respuesta.
Span	Paso concreto dentro de una traza.
Laboratorio	Espacio reproducible para probar una idea con casos, métricas y trazas.
Notebook	Documento ejecutable que mezcla explicación, código, salidas y metadatos.
Eval	Prueba sistemática para medir si un sistema cumple un comportamiento esperado.
Dataset de evaluación	Conjunto de casos con entradas, resultados esperados y criterios de aceptación.
Evaluación offline	Pruebas con casos preparados antes de uso real.
Evaluación online	Medición con uso real, tráfico y cambios.
Hit@k	Indica si el resultado esperado aparece entre los k primeros.
MRR	Media del inverso de la posición del primer resultado correcto.
Gate	Umbral o regla que decide si una variante puede avanzar.
Manifest	Registro de versión, datos, entorno y configuración del experimento.
Arquitectura mínima suficiente	Diseño más sencillo que resuelve el problema medido.

TÉRMINO	DEFINICIÓN
Decisión sin IA	Elección de resolver con reglas, SQL, interfaz o proceso cuando un modelo no aporta valor suficiente.

Antes de pasar página

Responde sin consultar el facsímil. Si fallas una pregunta, la pista te dice dónde volver.

- ☐ **1.** ¿Puedo elegir entre prompt, RAG, tool y ajuste explicando qué cambia cada uno? (Si no, vuelve al capítulo 01.)
- ☐ **2.** ¿Sé diseñar una llamada API con mensajes, parámetros, streaming, tool y salida estructurada? (Si no, vuelve al capítulo 02.)
- ☐ **3.** ¿Puedo estimar coste de tokens, contexto y caché antes de construir? (Si no, vuelve al capítulo 03.)
- ☐ **4.** ¿Sé leer una model card y separar marketing, métrica y restricción operativa? (Si no, vuelve al capítulo 04.)
- ☐ **5.** ¿Puedo explicar cuándo usar local, cloud, GPU alquilada o cuantización? (Si no, vuelve al capítulo 05.)
- ☐ **6.** ¿Sé qué es una dimensión de embedding y qué mide la similitud coseno? (Si no, vuelve al capítulo 06.)
- ☐ **7.** ¿Puedo diseñar búsqueda híbrida con filtros y metadatos? (Si no, vuelve al capítulo 08.)
- ☐ **8.** ¿Sé explicar por qué RAG no es memoria ni entrenamiento? (Si no, vuelve al capítulo 09.)
- ☐ **9.** ¿Puedo proponer métricas de evaluación RAG y casos donde el sistema debe abstenerse? (Si no, vuelve al capítulo 10.)
- ☐ **10.** ¿Sé cuándo Agentic RAG o GraphRAG compensan su complejidad? (Si no, vuelve al capítulo 11.)
- ☐ **11.** ¿Puedo diseñar una herramienta Text-to-SQL con permisos, validación y traza? (Si no, vuelve al capítulo 12.)
- ☐ **12.** ¿Puedo revisar una arquitectura y decir qué falta para llevarla a prototipo controlado? (Si no, vuelve a «La matriz de decisión final».)
- ☐ **13.** ¿Puedo defender cuándo no usar IA? (Si no, vuelve a «Cuándo no usar IA».)
- ☐ **14.** ¿Puedo explicar la diferencia entre demo, notebook y laboratorio? (Si no, vuelve a «Qué no es un laboratorio».)

- ☐ **15.** ¿Sé qué debe contener un dataset de evaluación mínimo y por qué importa el manifest? (Si no, vuelve a «Qué sí debería dejar».)
- ☐ **16.** ¿Puedo calcular Hit@1 y MRR con un ejemplo pequeño y leer un span? (Si no, vuelve a «Cómo medir de verdad: métricas, jueces y comparación».)
- ☐ **17.** ¿Puedo definir un gate con umbrales antes de ejecutar el experimento? (Si no, vuelve a «Anatomía de un laboratorio mínimo».)
- ☐ **18.** ¿Sé qué artefactos deja un laboratorio mínimo y por qué una traza permite auditar una decisión? (Si no, vuelve a «Anatomía de un laboratorio mínimo».)

En resumen

IDEA FUERZA	DETALLE
La caja de herramientas empieza con diagnóstico.	Antes de elegir técnica, hay que saber qué parte del sistema falla.
Un modelo útil necesita contratos alrededor.	APIs, schemas, herramientas, validadores y trazas hacen gobernable la respuesta.
La evidencia se recupera, se mide y se cita.	RAG, búsqueda híbrida, GraphRAG y Text-to-SQL son formas distintas de traer base verificable.
El coste también es arquitectura.	Tokens, contexto, caché, latencia, GPU, local y cloud cambian el diseño.
La evaluación separa demo de sistema.	Sin casos, métricas, abstención y regresiones, no sabes si mejoraste.
La complejidad tiene que ganarse el sitio.	Agentic RAG, GraphRAG o fine-tuning solo compensan cuando resuelven un fallo medido.
No usar IA también es una decisión técnica.	Si una regla, SQL o una interfaz clara resuelve mejor, esa es la herramienta correcta.
El siguiente paso es coordinar herramientas.	El facsímil 05 parte de esta base para hablar de agentes y orquestación.

Recursos para seguir: leer, construir y experimentar

Esta es la caja de herramientas: APIs, tokens, modelos locales, embeddings, bases vectoriales y RAG. Es el facsículo más práctico, así que el mejor recurso es ponerse.

Para experimentar sin apenas código. Lo has hecho en las cajas «Pruébalo en 5 minutos»: el tokenizador de OpenAI (platform.openai.com/tokenizer) para ver tokens y coste; los playgrounds (platform.openai.com , aistudio.google.com , console.anthropic.com) para salidas estructuradas y SQL; Hugging Face (huggingface.co) para leer fichas de modelos; LM Studio (lmstudio.ai) y Ollama (ollama.com) para correr modelos locales; y el Embedding Projector (projector.tensorflow.org) para la búsqueda semántica.

Para construir. Las piezas reales por tema: los SDK oficiales de OpenAI, Anthropic y Google para llamar a las APIs; LangChain o LlamaIndex para orquestrar RAG; bases vectoriales como Qdrant, Chroma o pgvector para la búsqueda; y Ollama o llama.cpp para lo local. Empieza con un runner pequeño y propio antes de montar una plataforma.

Para leer. Aquí lo importante no son los papers, sino la documentación: las guías oficiales de cada proveedor, que cada capítulo enlaza en su «Para saber más». Leerlas con un caso propio en mente es lo que convierte entender una herramienta en usarla en serio.

Para saber más

Arize Phoenix. (2026). *Evaluate RAG*. [Documentación oficial](#)

Arize Phoenix. (2026). *Evaluation concepts*. [Documentación oficial](#)

Boardman, A. E., Greenberg, D. H., Vining, A. R. y Weimer, D. L. (2018). *Cost-Benefit Analysis: Concepts and Practice* (5.^a ed.). Cambridge University Press. [DOI](#)

Brown, L. D., Cai, T. T. y DasGupta, A. (2001). *Interval Estimation for a Binomial Proportion*. *Statistical Science*, 16(2), 101-133. [DOI](#)

Chen, M. y otros (2021). *Evaluating Large Language Models Trained on Code*. [arXiv](#)

Cormack, G. V., Clarke, C. L. A. y Büttcher, S. (2009). *Reciprocal rank fusion outperforms Condorcet and individual rank learning methods*. *Proceedings of SIGIR*, 758-759. [DOI](#)

Edge, D. y otros (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. [arXiv](#)

Efron, B. y Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Chapman & Hall/CRC. [DOI](#)

Ellram, L. M. (1995). *Total cost of ownership: an analysis approach for purchasing*. *International Journal of Physical Distribution & Logistics Management*, 25(8), 4-23. [DOI](#)

Es, S. y otros (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. [arXiv](#)

Hu, E. J. y otros (2022). *LoRA: Low-Rank Adaptation of Large Language Models*. [arXiv](#)

- Järvelin, K. y Kekäläinen, J. (2002). *Cumulated gain-based evaluation of IR techniques*. *ACM Transactions on Information Systems*, 20(4), 422-446. [DOI](#)
- Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-scale similarity search with GPUs*. *IEEE Transactions on Big Data*, 7(3), 535-547. [DOI](#)
- Jupyter. (2026). *The Notebook file format*. [Documentación oficial](#)
- LangChain. (2026). *Evaluate a RAG application*. [Documentación oficial](#)
- Lewis, P. y otros (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. *Advances in Neural Information Processing Systems 33*, 9459-9474. [NeurIPS](#)
- Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . *Operations Research*, 9(3), 383-387. [DOI](#)
- Liu, Y. y otros (2023). *G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment*. *Proceedings of EMNLP 2023*. [arXiv](#)
- Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. [DOI](#)
- Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. [Disponible en línea](#)
- McNemar, Q. (1947). *Note on the sampling error of the difference between correlated proportions or percentages*. *Psychometrika*, 12(2), 153-157. [DOI](#)
- Mitchell, M. y otros (2019). *Model Cards for Model Reporting*. *Proceedings of FAT 2019*, 220-229. [DOI](#)
- OpenAI. (2026). *Function calling*. [Documentación oficial](#)
- OpenAI. (2026). *Graders*. [Documentación oficial](#)
- OpenAI. (2026). *Prompt caching*. [Documentación oficial](#)
- OpenAI. (2026). *Structured model outputs*. [Documentación oficial](#)
- OpenTelemetry. (2026). *Tracing API*. [Documentación oficial](#)
- Ragas. (2026). *List of available metrics*. [Documentación oficial](#)
- Robertson, S. y Zaragoza, H. (2009). *The probabilistic relevance framework: BM25 and beyond*. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. [DOI](#)
- Sainz, O. y otros (2023). *NLP Evaluation in Trouble: On the Need to Measure LLM Data Contamination for each Benchmark*. *Findings of EMNLP 2023*. [10776-10787. arXiv](#)

Salton, G., Wong, A. y Yang, C. S. (1975). *A vector space model for automatic indexing*. *Communications of the ACM*, 18(11), 613-620. [DOI](#)

Voorhees, E. M. (1999). *The TREC-8 Question Answering Track Report*. *Proceedings of the 8th Text REtrieval Conference (TREC-8)*, 77-82. [NIST](#)

Yu, T. y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. *Proceedings of EMNLP*, 3911-3921. [ACL Anthology](#)

Zheng, L. y otros (2023). *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. *Advances in Neural Information Processing Systems 36*. [arXiv](#)

Notas

1. OpenAI. (2026). *Function calling*. [Documentación oficial](#). Consultado el 26 de mayo de 2026.

↵

2. OpenAI. (2026). *Structured model outputs*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵

3. Hu, E. J. y otros (2022). *LoRA: Low-Rank Adaptation of Large Language Models*. *International Conference on Learning Representations*. [arXiv](#). Dettmers, T. y otros (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. *Advances in Neural Information Processing Systems 36*. [arXiv](#).

↵

4. OpenAI. (2026). *Prompt caching*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵

5. Ellram, L. M. (1995). *Total cost of ownership: an analysis approach for purchasing*. *International Journal of Physical Distribution & Logistics Management*, 25(8), 4-23. [DOI](#). ↵

6. Mitchell, M. y otros (2019). *Model Cards for Model Reporting*. *Proceedings of FAT 2019*, 220-229. [DOI](#). ↵

7. Salton, G., Wong, A. y Yang, C. S. (1975). *A vector space model for automatic indexing*. *Communications of the ACM*, 18(11), 613-620. [DOI](#). ↵

8. Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-scale similarity search with GPUs*. *IEEE Transactions on Big Data*, 7(3), 535-547. [DOI](#). ↵

9. Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. [DOI](#). ↵

10. Robertson, S. y Zaragoza, H. (2009). *The probabilistic relevance framework: BM25 and beyond*. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. [DOI](#). ↵

1. Cormack, G. V., Clarke, C. L. A. y Büttcher, S. (2009). *Reciprocal rank fusion outperforms Condorcet and individual rank learning methods*. *Proceedings of SIGIR*, 758-759. [DOI](#). ↵
2. Lewis, P. y otros (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. *Advances in Neural Information Processing Systems 33*, 9459-9474. [NeurIPS](#). ↵
3. Es, S. y otros (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. [arXiv](#). ↵
4. Edge, D. y otros (2024). *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. [arXiv](#). ↵
5. Yu, T. y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. *Proceedings of EMNLP*, 3911-3921. [ACL Anthology](#). ↵
6. Boardman, A. E., Greenberg, D. H., Vining, A. R. y Weimer, D. L. (2018). *Cost-Benefit Analysis: Concepts and Practice* (5.ª ed.). Cambridge University Press. [Enlace](#). ↵
7. Jupyter. (2026). *The Notebook file format*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
8. OpenTelemetry. (2026). *Tracing API*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
9. OpenAI. (2026). *Graders*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
10. Ragas. (2026). *List of available metrics*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
11. LangChain. (2026). *Evaluate a RAG application*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
12. Arize Phoenix. (2026). *Evaluate RAG*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
13. Arize Phoenix. (2026). *Evaluation concepts*. [Documentación oficial](#). Consultado el 26 de mayo de 2026. ↵
14. Järvelin, K. y Kekäläinen, J. (2002). *Cumulated gain-based evaluation of IR techniques*. *ACM Transactions on Information Systems*, 20(4), 422-446. [DOI](#). ↵
15. Es, S. y otros (2023). *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. [arXiv](#). ↵
16. Zheng, L. y otros (2023). *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. *Advances in Neural Information Processing Systems 36*. [arXiv](#). ↵

17. Liu, Y. y otros (2023). *G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment*. *Proceedings of EMNLP 2023*. [arXiv](#). ↵
18. McNemar, Q. (1947). *Note on the sampling error of the difference between correlated proportions or percentages*. *Psychometrika*, 12(2), 153-157. [DOI](#). ↵
19. Efron, B. y Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Chapman & Hall/CRC. [DOI](#). ↵
20. Sainz, O. y otros (2023). *NLP Evaluation in Trouble: On the Need to Measure LLM Data Contamination for each Benchmark*. *Findings of the Association for Computational Linguistics: EMNLP 2023*, 10776-10787. [arXiv](#). ↵
21. Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . *Operations Research*, 9(3), 383-387. [DOI](#). ↵
22. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, cap. 8. [Disponible en línea](#). ↵
23. Voorhees, E. M. (1999). *The TREC-8 Question Answering Track Report*. *Proceedings of the 8th Text REtrieval Conference (TREC-8)*, 77-82. [NIST](#). ↵
24. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, cap. 8 (accuracy, precision y recall). [Disponible en línea](#). ↵
25. Chen, M. y otros (2021). *Evaluating Large Language Models Trained on Code*. [arXiv](#). ↵
26. Brown, L. D., Cai, T. T. y DasGupta, A. (2001). *Interval Estimation for a Binomial Proportion*. *Statistical Science*, 16(2), 101-133. [DOI](#). ↵