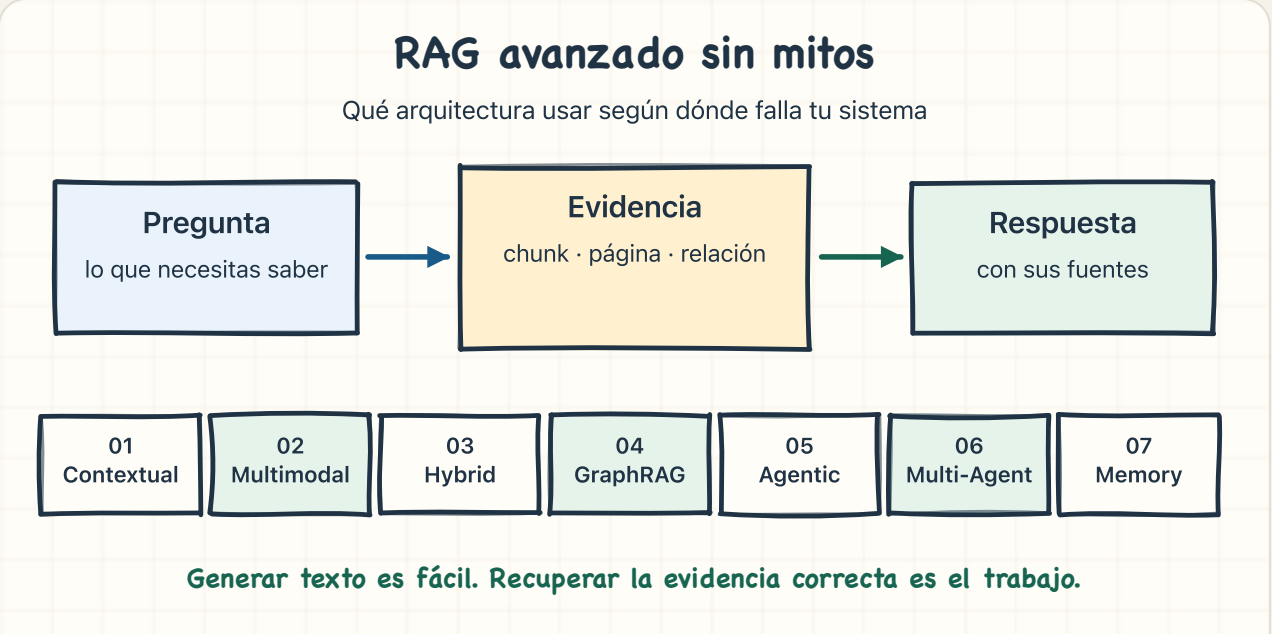
RAG avanzado sin mitos: qué arquitectura usar según dónde falla tu sistema

Creado por Manuel Herrera Mesén · 10 de noviembre de 2026

rag inteligencia-artificial llm arquitectura busqueda

Son las dos de la mañana y salta una alerta en pagos. Quien está de guardia le pregunta al asistente del equipo: "¿cómo resuelvo el timeout de conexiones en PaymentService?". La respuesta llega en segundos, bien redactada y segura de sí misma. Y sale del runbook de otro servicio.

Lo incómodo es que puede no haber ningún error de programación. El sistema hizo exactamente lo que se diseñó para hacer: buscó los fragmentos más parecidos a la pregunta y redactó con ellos. El problema estuvo antes de redactar, en lo que encontró.



Generar texto es fácil. Recuperar la evidencia correcta es el trabajo.

> **Lo que debes llevarte:** casi siempre, el problema de fondo no es generar buenas respuestas, sino recuperar evidencia correcta, suficiente y verificable.

Esa es la idea que recorre esta guía. Un RAG no mejora por sumar herramientas; mejora cuando corrige una debilidad que puedes demostrar. Las siete arquitecturas que vamos a ver no son etapas obligatorias ni alternativas excluyentes: son patrones que se combinan según la falla que tengas delante.

Tu punto de partida	Por dónde te recomiendo leer
Nunca construiste un RAG	Empieza por el caso y los fundamentos, y sigue en orden.

Tu punto de partida	Por dónde te recomiendo leer
Ya tienes uno y falla	Ve al mapa de fallas, busca tu síntoma y salta a su arquitectura.
Diseñas sistemas en producción	Lee la implementación empresarial, la evaluación y abre las notas "Para profundizar".

▼ EN ESTE ARTÍCULO

01. El caso que vamos a seguir
02. Primero: qué es RAG y cómo funciona
03. Dónde empieza a fallar Naive RAG
04. 01 · Contextual RAG: devolverle el significado al fragmento
05. 02 · Multimodal RAG: lo que no es solamente texto
06. 03 · Hybrid Search: significado y coincidencia exacta
07. 04 · GraphRAG: relaciones entre entidades
08. 05 · Agentic RAG: buscar otra vez cuando la evidencia no alcanza
09. 06 · Multi-Agent RAG: investigaciones especializadas en paralelo
10. 07 · Memory RAG: recuperar lo que se decidió antes
11. Cómo elegir una arquitectura
12. Cómo se vería en una empresa
13. Cómo evaluar un RAG de verdad
14. Mitos frecuentes
15. Cómo quedaría el asistente de guardia
16. Fuentes

El caso que vamos a seguir

Para que esto no se quede en teoría, voy a usar un solo caso de principio a fin. No es un sistema real: lo armé para esta guía. Pero cada falla que aparece es de las que te vas a encontrar en uno.

Una tienda online funciona con cuatro microservicios Java que se comunican por eventos en Kafka: pedidos (OrderService), pagos (PaymentService), inventario (InventoryService) y notificaciones (NotificationService). El equipo quiere un **asistente de guardia**: quien recibe una alerta le pregunta en lenguaje natural qué está pasando y qué revisar.

El asistente consulta cuatro fuentes:

- los **runbooks** de cada servicio, los procedimientos para operarlo y recuperarlo;
- los **ADR** (*architecture decision records*), los documentos que registran cada decisión de arquitectura y por qué se tomó;
- los **tickets** de incidentes pasados;
- y los **logs** de los servicios.

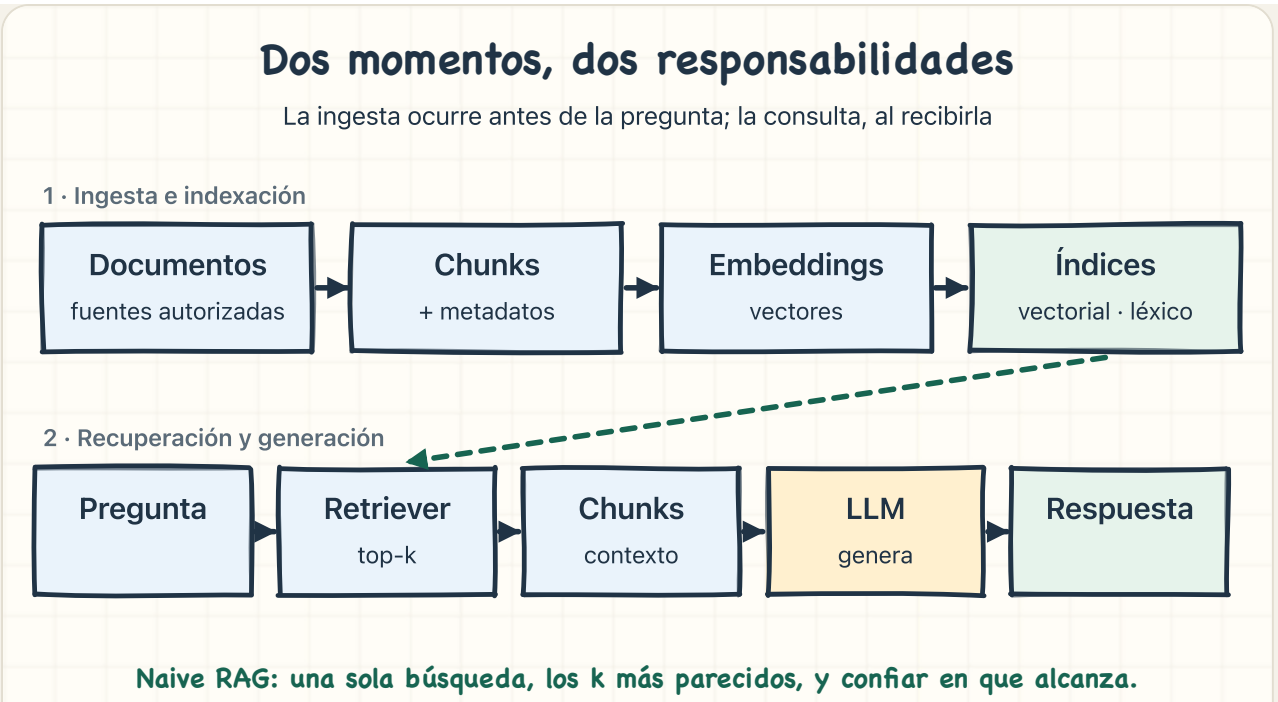
Cada capítulo empieza con una pregunta de guardia que la versión más simple responde mal, y al final armamos la versión que yo construiría.

Primero: qué es RAG y cómo funciona

Retrieval-Augmented Generation (RAG), o generación aumentada por recuperación, es una forma de construir aplicaciones en la que consultas información externa y se la das a un modelo de lenguaje antes de que responda. El modelo no aprende esa información de forma permanente: la recibe como contexto en esa llamada y no la conserva al terminar.

Todo RAG tiene dos momentos, y conviene no mezclarlos:

1. **Ingesta e indexación**, antes de que exista la pregunta: obtienes los documentos, conservas sus metadatos, extraes el texto, lo partes en fragmentos y construyes los índices.
2. **Recuperación y generación**, cuando llega la pregunta: la interpretas, consultas las fuentes autorizadas, recuperas la evidencia y el modelo construye la respuesta con ella.



La ingesta prepara lo que la consulta podrá encontrar. Si algo se pierde al indexar, ninguna consulta lo recupera.

La implementación más simple recupera los **top-k**, los k fragmentos más parecidos a la pregunta, y se los pasa al modelo. Se le suele llamar **Naive RAG**. No es un insulto: para colecciones pequeñas y preguntas simples es un buen punto de partida, y la línea base contra la que vas a medir todo lo demás. Su límite es suponer que una sola búsqueda trae todo lo necesario.

Los conceptos que vas a ver en toda la guía

Término	Qué es
LLM	Modelo de lenguaje grande (*large language model*): el modelo que lee el contexto y redacta la respuesta.
Token	La unidad en la que un modelo parte el texto; una palabra ocupa uno o varios. El costo y el tamaño del contexto se miden en tokens.
Prompt	Todo lo que el modelo recibe en una llamada: instrucciones, pregunta y evidencia.
Chunk	Un fragmento de un documento: la unidad que se indexa y se recupera.
Embedding	Un vector de números que representa el significado de un texto. Dos textos parecidos tienen vectores cercanos; la cercanía se suele medir con la similitud coseno, que va de −1 a 1.
Búsqueda semántica	Busca por significado, comparando embeddings.

Término	Qué es
Búsqueda léxica	Busca por las palabras que coinciden. **BM25** es su algoritmo más usado: puntúa por las palabras compartidas y por lo raras que son en la colección.
Base vectorial	El sistema que guarda embeddings y encuentra los más cercanos a uno dado.
Retriever	El componente que, dada una pregunta, devuelve candidatos relevantes.
Reranker	Una segunda pasada que reordena los mejores candidatos con un modelo más caro y más preciso.
Tenant	Cada cliente u organización que comparte el mismo sistema y cuyos datos deben quedar aislados de los demás.

Cómo partes el documento importa más de lo que parece

Antes de cualquier arquitectura avanzada hay una decisión que causa muchas de las fallas: el **chunking**. Un chunk demasiado grande mezcla temas y diluye su embedding; uno demasiado pequeño pierde el contexto que le da sentido. Lo habitual es cortar respetando la estructura del documento (títulos, secciones, párrafos), con algo de solapamiento entre fragmentos.

Y en cada chunk se guarda de dónde salió: documento, sección, versión y quién puede verlo. En el asistente de guardia, además, el servicio y el entorno. No existe un tamaño de chunk que funcione para todo: se elige con datos, como todo lo demás en esta guía.

> – **Para profundizar**
>
> si cambias el modelo de embeddings, tienes que reindexar todo. Los vectores de dos modelos distintos no viven en el mismo espacio y no se pueden comparar entre sí. Guarda el nombre y la versión del modelo junto al índice para que ese cambio sea una decisión explícita y no un error silencioso.

Dónde empieza a fallar Naive RAG

Cada una de estas fallas puede justificar una mejora distinta. Este mapa es el índice práctico de la guía: busca tu síntoma y salta a su arquitectura.



Empieza por el síntoma, no por la arquitectura.

Síntoma	La pregunta de guardia que lo muestra	Arquitectura
El chunk perdió su contexto	"¿Cómo resuelvo el timeout de conexiones en PaymentService?"	01 · Contextual RAG
La respuesta está en un diagrama o una tabla	"¿Qué servicios consumen OrderCreated?"	02 · Multimodal RAG
No encuentra un código o un identificador exacto	"HikariPool-1 connection timeout"	03 · Hybrid Search
Hay que conectar varias entidades	"Si OrderService deja de publicar, ¿qué se rompe?"	04 · GraphRAG
Una búsqueda no reúne la evidencia	"¿Por qué falló el pedido ORD-10482?"	05 · Agentic RAG
La investigación es amplia y paralelizable	"¿Qué causó la caída de pagos del viernes?"	06 · Multi-Agent RAG
Se pierde lo que se decidió antes	"¿Sigue vigente lo que decidimos sobre el pool?"	07 · Memory RAG

01 · Contextual RAG: devolverle el significado al fragmento

El problema: al separar un fragmento de su documento, puede perder justo la información que lo hacía útil.

Es la falla del principio de esta guía. Los cuatro runbooks tienen una sección casi idéntica sobre timeouts de la base de datos, y el chunk que la contiene dice esto:

ANTES

Chunk original

```
Si el error persiste, sube
maximumPoolSize a 20 y reinicia el
servicio.
```

DESPUÉS

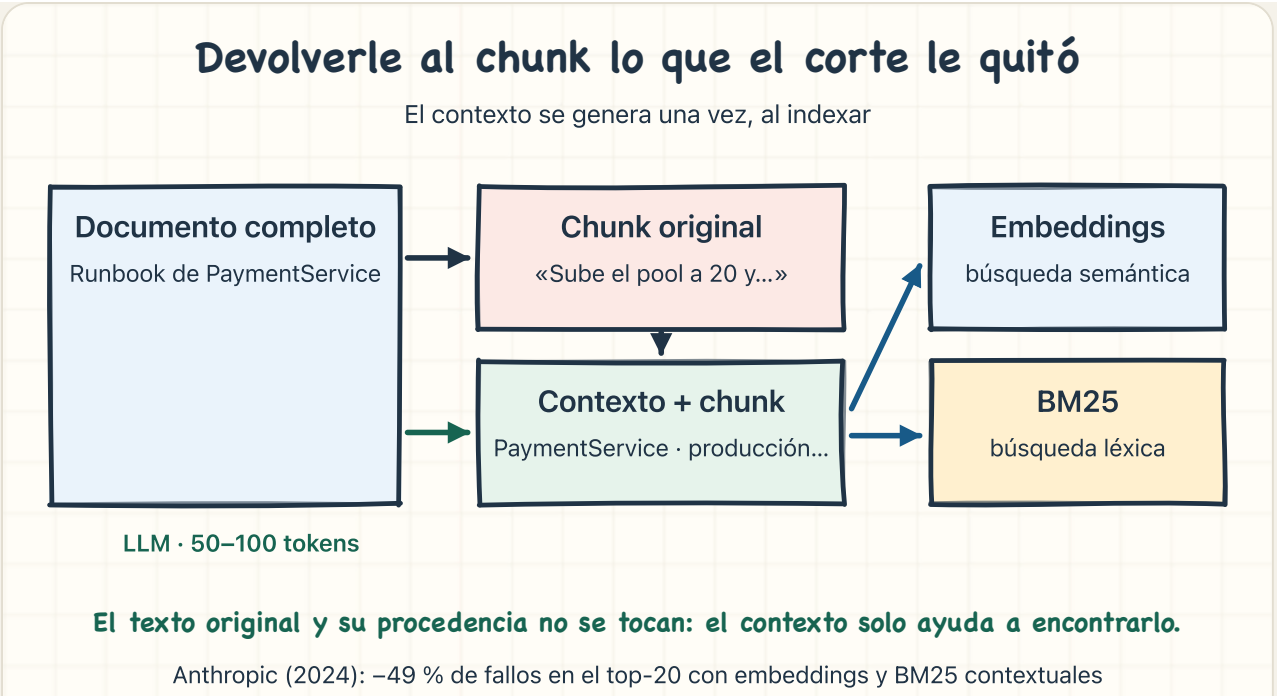
Chunk contextualizado

```
Runbook de PaymentService, producción.
Sección «Timeouts de conexión a la base
de datos».
Si el error persiste, sube
maximumPoolSize a 20 y reinicia el
servicio.
```

El chunk original no dice de qué servicio ni de qué entorno habla. Para la búsqueda, los cuatro fragmentos son igual de buenos, y el asistente puede darte la instrucción de InventoryService para un problema de pagos. El chunk contextualizado nombra el servicio, el entorno y la sección: ahora la pregunta tiene de dónde agarrarse.

Cómo funciona

Durante la ingesta, un modelo de lenguaje lee el documento completo y escribe una descripción breve para cada chunk, que se antepone al texto antes de indexarlo. Anthropic presentó la técnica con el nombre *Contextual Retrieval*: ese contexto suele ocupar entre 50 y 100 tokens, y se usa tanto para los embeddings como para el índice BM25, a los que llama **Contextual Embeddings** y **Contextual BM25**.



El contexto se genera una vez, al indexar. El texto original no se modifica.

En su experimento, la tasa de fallos de recuperación (las preguntas cuya evidencia no aparecía entre los 20 primeros resultados) era del 5,7 %. Bajó al 3,7 % con embeddings contextuales (−35 %), al 2,9 % al sumar BM25 contextual (−49 %) y al 1,9 % al agregar reranking (−67 %) (Anthropic, *Contextual Retrieval*, 2024). Son resultados sobre sus datos, no una promesa sobre los tuyos.

Cuándo usarlo	Sus límites
Manuales extensos, contratos, políticas, documentación corporativa y código, donde un fragmento aislado resulta ambiguo.	Encarece la indexación: una llamada al modelo por chunk. Si el contexto generado es incorrecto, introduce información engañosa en el índice.

Antes de pagar ese costo, mira si un metadato alcanza. Si cada chunk guarda su servicio, filtrar por `servicio = PaymentService` resuelve este caso concreto, y es más barato. Contextual RAG sirve cuando la ambigüedad no cabe en un campo.

> – Para profundizar
>
> el costo de una llamada por chunk se dispara si cada una manda el documento completo. Con **prompt caching**, que guarda en caché la parte repetida del prompt para no pagarla entera en cada llamada, el documento se envía una vez y cada chunk paga solo su parte.
>
> Anthropic estimó 1,02 USD por millón de tokens de documento con chunks de 800 tokens, documentos de 8.000, 50 tokens de instrucciones, 100 de contexto por chunk y los precios de

> Claude 3 Haiku de 2024: recalcúlalo con tus tamaños y los precios de hoy. Y cuando cambia el documento, tienes que regenerar el contexto de sus chunks, no solo sus embeddings.

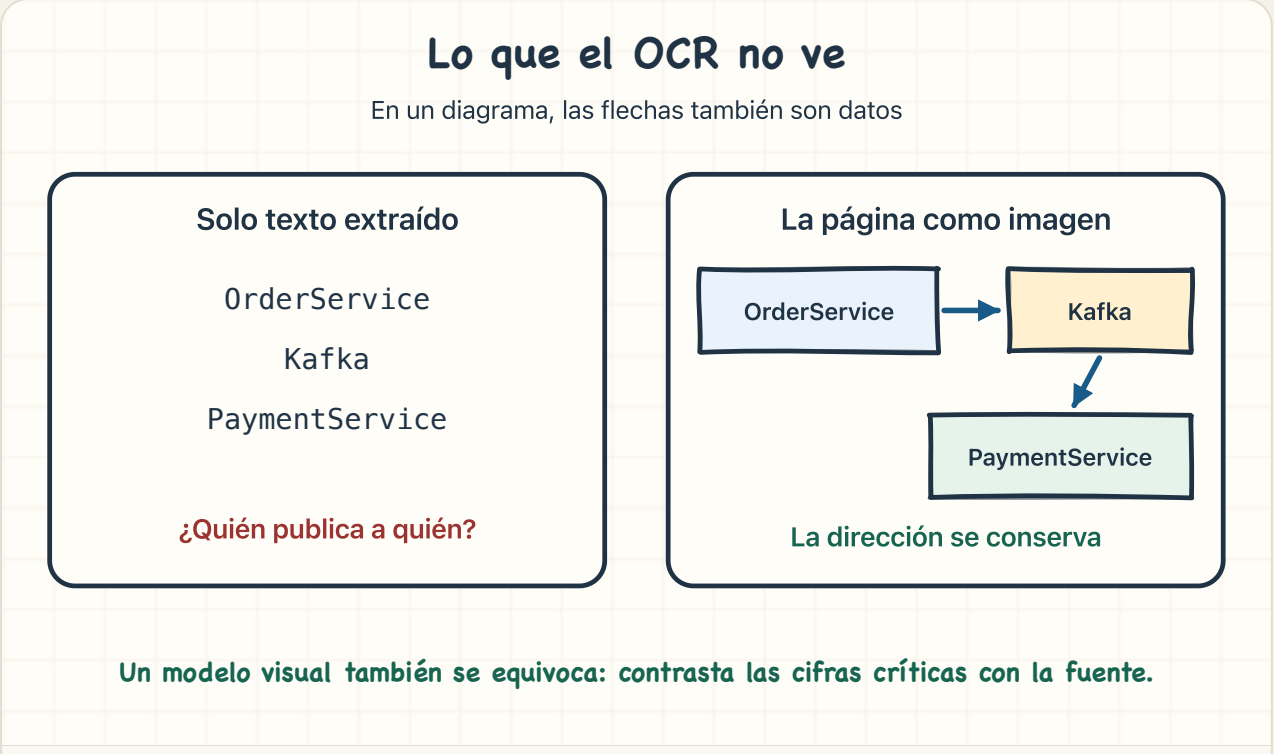
> **Lo que debes llevarte:** conserva siempre el texto original y su procedencia. El contexto generado ayuda a encontrar el fragmento; no es una nueva fuente de verdad.

---

02 · Multimodal RAG: lo que no es solamente texto

El problema: al extraer el texto de un PDF se puede destruir información que vivía en tablas, diagramas, gráficos e imágenes.

En el asistente de guardia, el diagrama de arquitectura de la tienda solo existe como imagen dentro de un ADR. Alguien pregunta: "¿qué servicios consumen OrderCreated?". El **OCR** (reconocimiento óptico de caracteres, el paso que convierte una imagen en texto) rescata los nombres de las cajas, pero la respuesta está en la dirección de las flechas, y las flechas no son texto.



Los nombres sobreviven al OCR; las relaciones dibujadas, no siempre.

Cómo funciona

Hay dos caminos, y se pueden combinar:

- **Recuperar la página como imagen.** El sistema calcula representaciones visuales de cada página, recupera las más relevantes para la pregunta y se las entrega a un modelo capaz de leer imágenes. ColPali es la referencia académica de este enfoque: indexa las páginas directamente como imagen, sin pasar por el OCR (Faysse et al., ICLR 2025).
- **Extraer mejor.** Combinar OCR, extracción estructurada de tablas, descripciones de las imágenes y embeddings multimodales, para que la estructura llegue al índice como texto con sentido.

Cuándo usarlo	Sus límites
PDFs escaneados, manuales con ilustraciones, tablas financieras, diagramas de arquitectura y todo documento donde la disposición visual importa.	Más cómputo y más almacenamiento. Un modelo visual también se equivoca al leer cifras, relaciones o estructuras.

> – **Para profundizar**
>
> ColPali no produce un vector por página, sino muchos: uno por cada parche de la imagen. La similitud se calcula con *late interaction* (el mismo mecanismo de ColBERT): cada token de la pregunta busca su parche más parecido y se suman esas coincidencias. Es más preciso que comparar un solo vector, pero ocupa bastante más espacio en el índice.

> **Lo que debes llevarte:** mirar la imagen no elimina los errores. Con datos críticos, contrasta los valores que leyó el modelo con la fuente original.

---

03 · Hybrid Search: significado y coincidencia exacta

El problema: la búsqueda semántica encuentra conceptos relacionados, pero puede pasar por alto códigos y términos exactos.

En el asistente de guardia, la persona pega el error tal como sale en el log: **`HikariPool-1 connection timeout`**. HikariCP es el pool de conexiones que Spring Boot usa por defecto, y **`HikariPool-1`** es el nombre que le da al primero. La búsqueda semántica trae documentación general sobre pools de conexiones; la léxica encuentra el ticket donde ese mensaje exacto ya apareció. Cada una ve una parte.

Cómo funciona

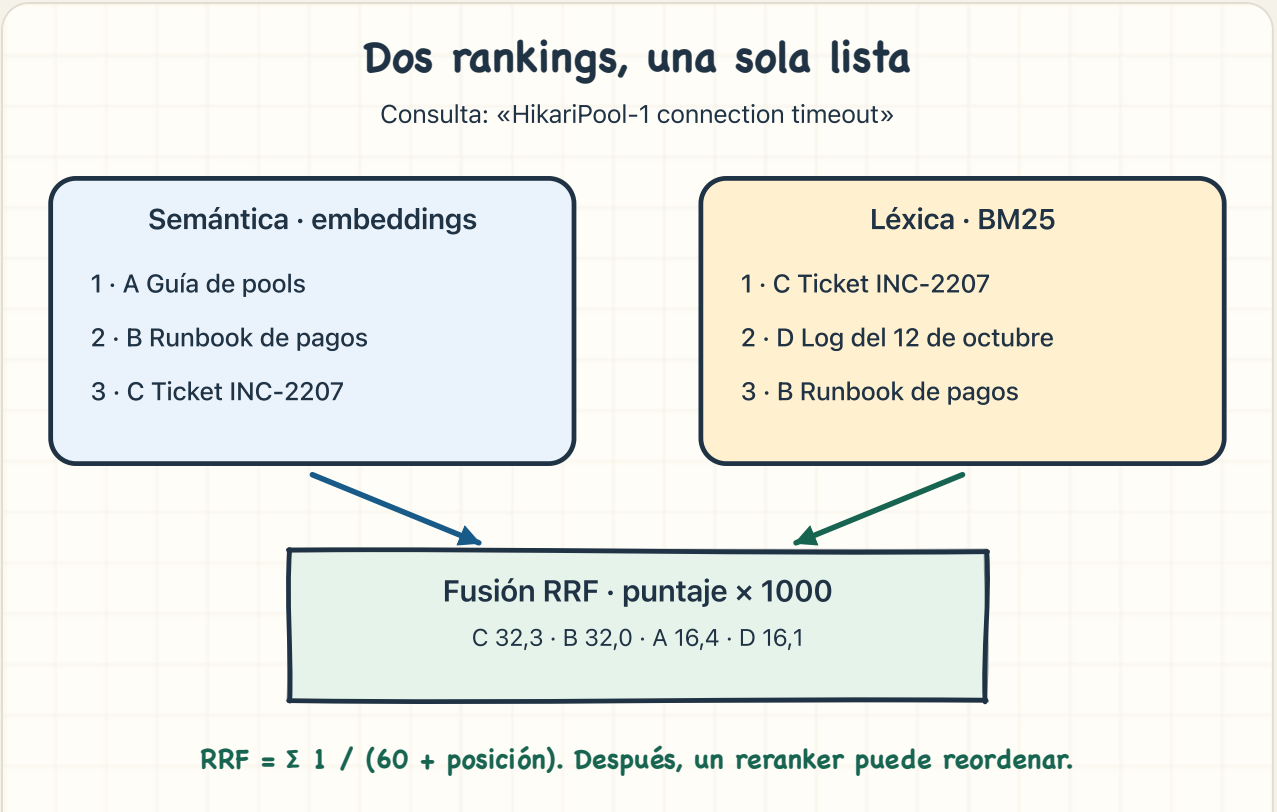
La **búsqueda híbrida** ejecuta las dos búsquedas sobre las fuentes permitidas y fusiona sus resultados. El problema es que sus puntajes no se pueden comparar: una similitud coseno de 0,82 y un puntaje BM25 de 14,3 miden cosas distintas.

La técnica habitual es **Reciprocal Rank Fusion (RRF)**, que ignora los puntajes y usa solo las posiciones. Cada documento suma, por cada lista en la que aparece:

RRF

```
puntaje(d) = Σ  1 / (60 + posición de d en esa lista)
```

Ese 60 es una constante de suavizado: el artículo original la llama k, que no tiene nada que ver con el k de top-k. Sus autores la fijaron en una prueba piloto y comprobaron que su valor exacto no era crítico (Cormack, Clarke y Büttcher, SIGIR 2009). Mira cómo funciona con cuatro documentos:



Un documento que aparece en las dos listas suele ganarle a uno que es primero en una sola.

RRF fusiona dos rankings paso a paso

Candidatos	Puntaje RRF (× 1000)	Lista final
		✓ C · Ticket INC-2207
		✓ B · Runbook de pagos
		✓ A · Guía de pools
		✓ D · Log del 12 de octubre

Llegan los candidatos. La semántica ordenó A, B, C; la léxica (BM25), C, D, B. Los puntajes originales se descartan: solo cuentan las posiciones. Para leerlos mejor, los puntajes RRF se muestran multiplicados por 1000.

A es primero en semántica y no aparece en BM25: 1000 / (60 + 1) ≈ 16,4.

D es segundo en BM25 y no aparece en semántica: 1000 / (60 + 2) ≈ 16,1.

B aparece en las dos: segundo en semántica y tercero en BM25. 1000 / 62 + 1000 / 63 ≈ 32,0.

C también aparece en las dos: tercero en semántica y primero en BM25. 1000 / 63 + 1000 / 61 ≈ 32,3.

Orden final: C, B, A, D. El ticket con el mensaje exacto, que la búsqueda semántica dejó tercero, queda primero porque las dos búsquedas lo encontraron.

Después de fusionar, un **reranker** puede reordenar los primeros candidatos. El retriever puntúa cada documento con representaciones preparadas de antemano, sin leerlo junto a la pregunta. El reranker, normalmente un *cross-encoder*, lee la pregunta y cada documento juntos, y por eso juzga mejor la relevancia. Es más caro, así que solo se aplica a los primeros resultados.

Cuándo usarlo	Sus límites
Logs, documentación de APIs, código fuente, tickets, endpoints, códigos de error e identificadores.	Hay que configurar dos índices, su tokenización y la fusión, y evaluar la relevancia. BM25 tampoco garantiza la coincidencia exacta.

– Para profundizar

que BM25 encuentre `HikariPool-1` depende del analizador, el componente que parte el texto en términos. Uno estándar suele partirlo en `hikaripool` y `1`. Ese `1` aparece en medio corpus, así que casi no suma: un log de `HikariPool-2`, que es otro pool, puntúa prácticamente igual. Para identificadores conviene un campo aparte que los conserve enteros.

Y RRF no es la única fusión: normalizar los puntajes y sumarlos con pesos puede rendir más si ajustas esos pesos con datos etiquetados; RRF no necesita ajuste, y por eso es el punto de partida.

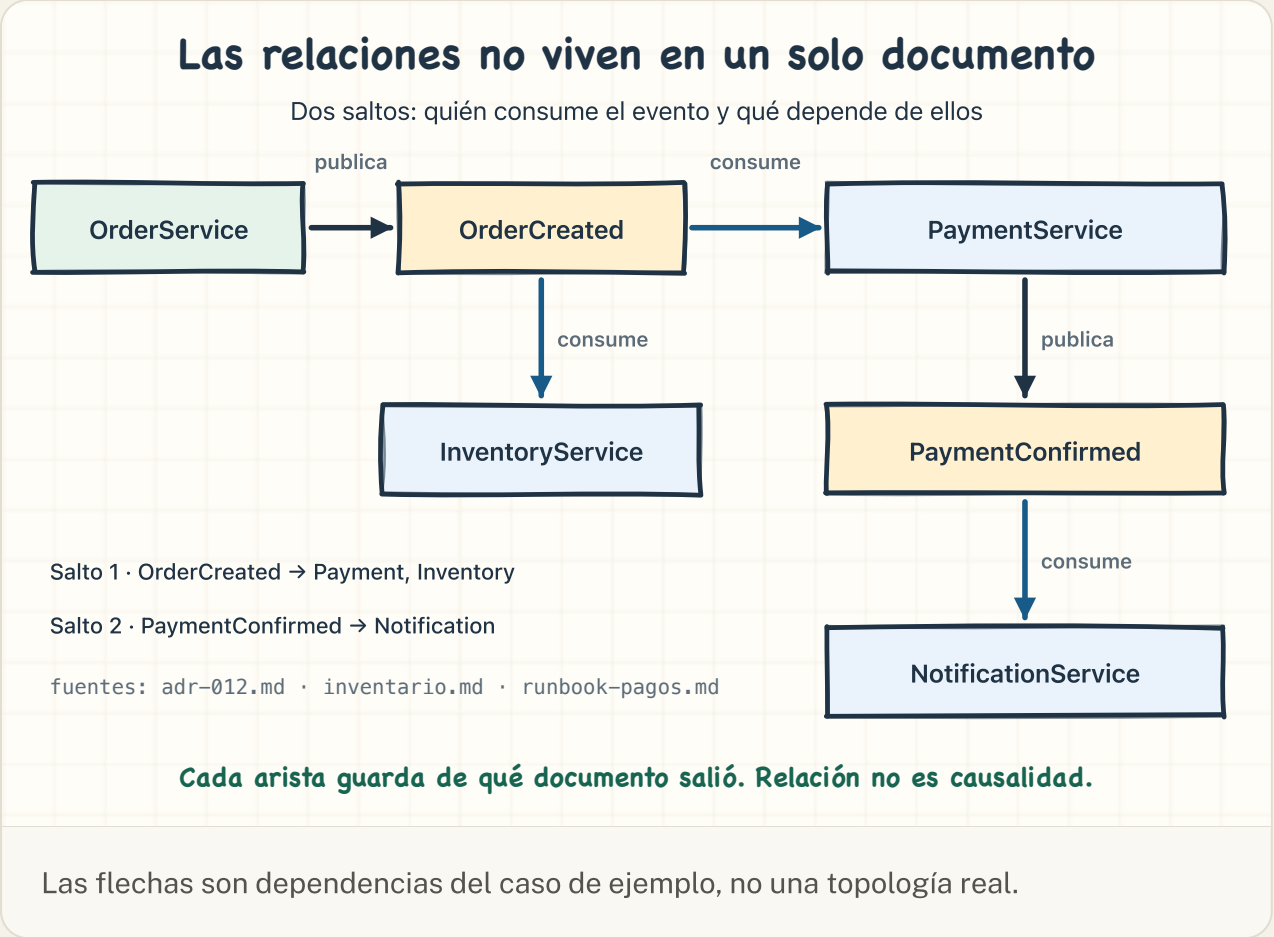
Lo que debes llevarte: mide Recall@k y nDCG@k antes y después de sumar la búsqueda híbrida. La primera dice cuánta de la evidencia correcta aparece entre los primeros k resultados; la segunda, si aparece arriba. Las dos se explican en la evaluación. Si no mejoran sobre tus consultas reales, no ganaste nada salvo complejidad.

---

04 · GraphRAG: relaciones entre entidades

El problema: a veces la respuesta depende de relaciones repartidas entre varios documentos y componentes, y ningún fragmento por sí solo la contiene.

En el asistente de guardia: "si OrderService deja de publicar OrderCreated, ¿qué se rompe?". Hay que saber quién consume ese evento y, en un segundo salto, qué depende de esos consumidores. Esa información está repartida entre un ADR, el runbook de cada servicio y el inventario de servicios. Una búsqueda por similitud trae fragmentos sueltos; no recorre la cadena.



Las flechas son dependencias del caso de ejemplo, no una topología real.

Cómo funciona

El sistema extrae entidades y relaciones de los documentos, construye un grafo de conocimiento y guarda en cada relación el documento que la sustenta. Una pregunta sobre una entidad concreta recupera su vecindario en el grafo, y con él los fragmentos que lo respaldan.

Microsoft GraphRAG, la implementación más conocida, va un paso más allá. Agrupa las entidades en comunidades con el algoritmo de Leiden y escribe un resumen de cada una. Así puede responder preguntas generales sobre toda la colección, del tipo "¿cuáles son los temas principales?".

Ofrece cuatro modos de búsqueda: **local** (una entidad y sus vecinos), **global** (sobre los resúmenes de comunidad), **DRIFT** (local, pero enriquecida con el contexto de la comunidad) y **básica** (el top-k vectorial de siempre). No todas las implementaciones de GraphRAG usan comunidades.

Cuándo usarlo	Sus límites
Análisis de dependencias, investigación de incidentes, conocimiento organizacional y preguntas que conectan muchas entidades y fuentes.	Construir y mantener el grafo es caro: extraer entidades y resumir comunidades exige llamadas al modelo durante la indexación. Las relaciones extraídas pueden ser incorrectas o quedar desactualizadas.

Dos trampas antes de construirlo. La primera es la **resolución de entidades**: si el mismo servicio aparece como `PaymentService`, `payments-svc` y "el servicio de pagos", sin unificarlos tienes tres nodos y la cadena se corta. La segunda: si las relaciones ya existen como datos (un catálogo de servicios, la configuración de los consumidores de Kafka), constrúyelo desde ahí. Es más exacto que extraerlas de textos con un modelo.

> **Lo que debes llevarte:** cada relación tiene que conservar su procedencia. Y que dos cosas aparezcan relacionadas en un texto no significa que una cause la otra.

---

05 · Agentic RAG: buscar otra vez cuando la evidencia no alcanza

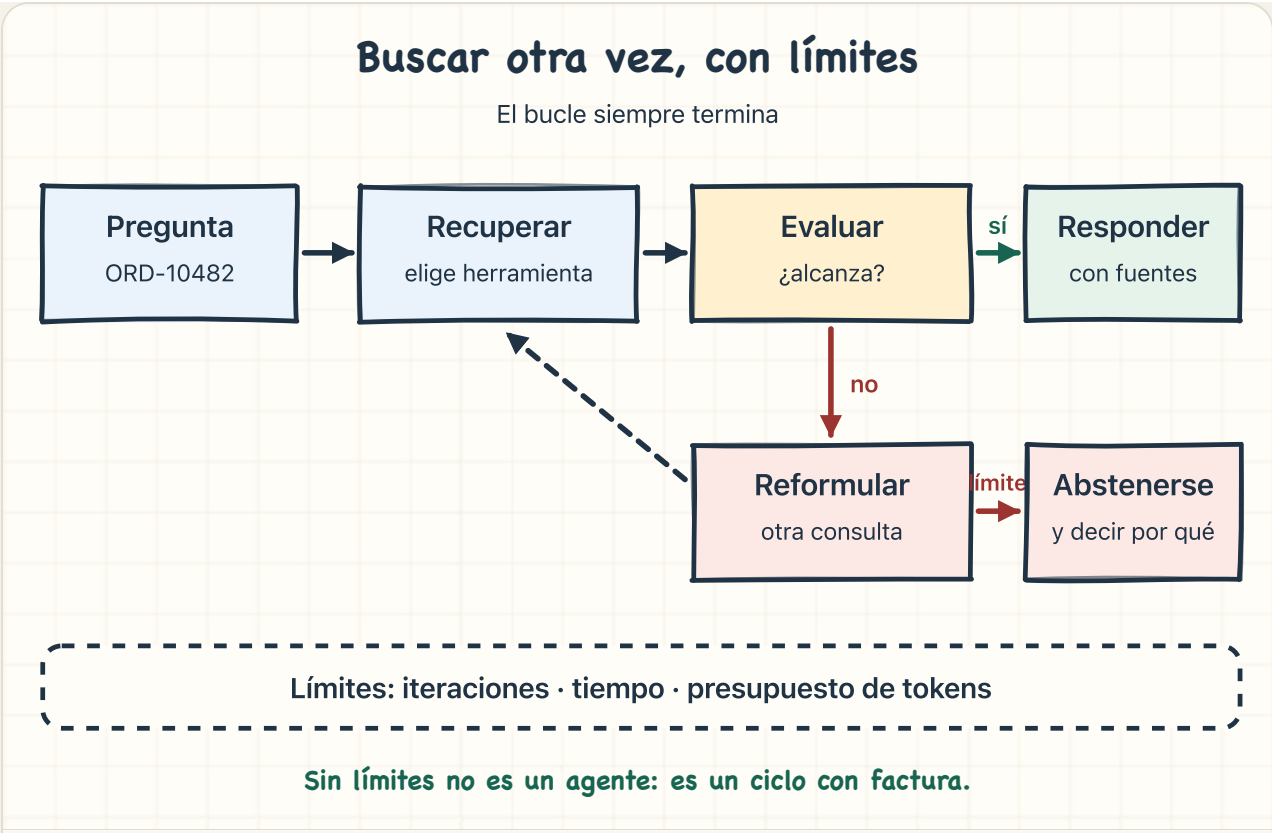
El problema: una sola recuperación no siempre basta para contestar una pregunta compleja.

En el asistente de guardia: "¿por qué falló el pedido ORD-10482 y qué servicio reviso primero?". Primero hay que encontrar el pedido en los logs; solo con lo que aparezca ahí se sabe qué runbook consultar. Son varias búsquedas, y la segunda depende de lo que encontró la primera.

Cómo funciona

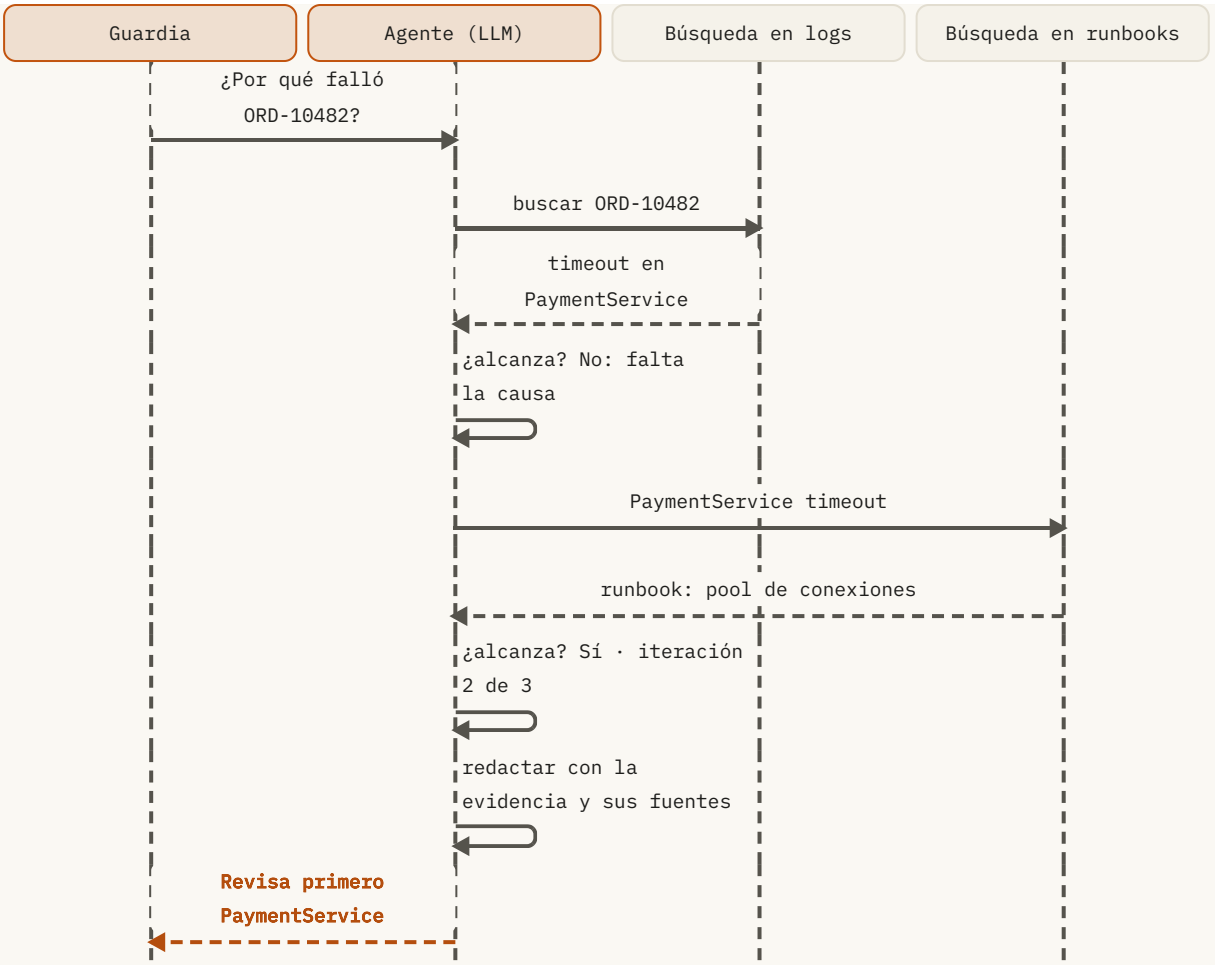
Un agente, que es el propio modelo de lenguaje con acceso a herramientas de búsqueda, decide qué buscar, inspecciona lo que encontró y juzga si le alcanza. Si falta evidencia, reformula la pregunta o consulta otra fuente. El proceso termina cuando reúne información suficiente, cuando llega a un límite o cuando concluye que con la evidencia disponible no se puede responder.

Esos límites no se le piden al modelo: los impone el código que lo orquesta, el que cuenta las iteraciones, mide el tiempo y corta cuando se acaba el presupuesto. Un modelo puede ignorar una instrucción; un contador en tu código, no.



El bucle siempre termina: con una respuesta, o con una abstención explicada.

Dos búsquedas para una sola pregunta



La pregunta necesita un hecho (qué pasó) y una explicación (por qué y dónde mirar).

Es un identificador exacto: el agente elige la búsqueda en logs.

Ya sabe qué pasó, pero no por qué.

Evalúa la evidencia antes de responder. Iteración 1 de un máximo de 3: la cuenta la lleva el código que orquesta al agente, no el modelo.

Reformula con lo que aprendió: ahora busca en el runbook del servicio.

Encuentra el procedimiento que explica ese timeout.

Tiene el hecho y su explicación, dentro del límite.

El mismo modelo redacta la respuesta con la evidencia recuperada, no con lo que cree recordar.

Cada afirmación apunta al log o al runbook del que salió, para que quien está de guardia pueda verificarla.

Antes de construir un agente, pregúntate si lo necesitas. Anthropic distingue los *workflows*, donde el código define de antemano los pasos y el modelo los ejecuta, de los agentes, donde el modelo decide los pasos (Anthropic, *Building effective agents*, 2024). Si puedes escribir el flujo ("busca el pedido en los logs; si aparece un servicio, busca su runbook"), un workflow es más barato, más predecible y más fácil de probar.

La idea de recuperar más de una vez no nació con los agentes de moda. **Self-RAG** entrenó a un modelo para decidir cuándo recuperar y para criticar su propia respuesta (Asai et al., 2023). **Corrective RAG**

agregó un evaluador que puntúa los documentos recuperados y, si no sirven, los descarta y busca en la web (Yan et al., 2024). Agentic RAG generaliza esas ideas con herramientas.

Cuándo usarlo	Sus límites
Investigaciones técnicas, diagnósticos, preguntas que encadenan varios saltos y problemas que exigen más de una recuperación.	Más latencia, más costo y riesgo de búsquedas innecesarias. El agente también puede equivocarse al decidir.

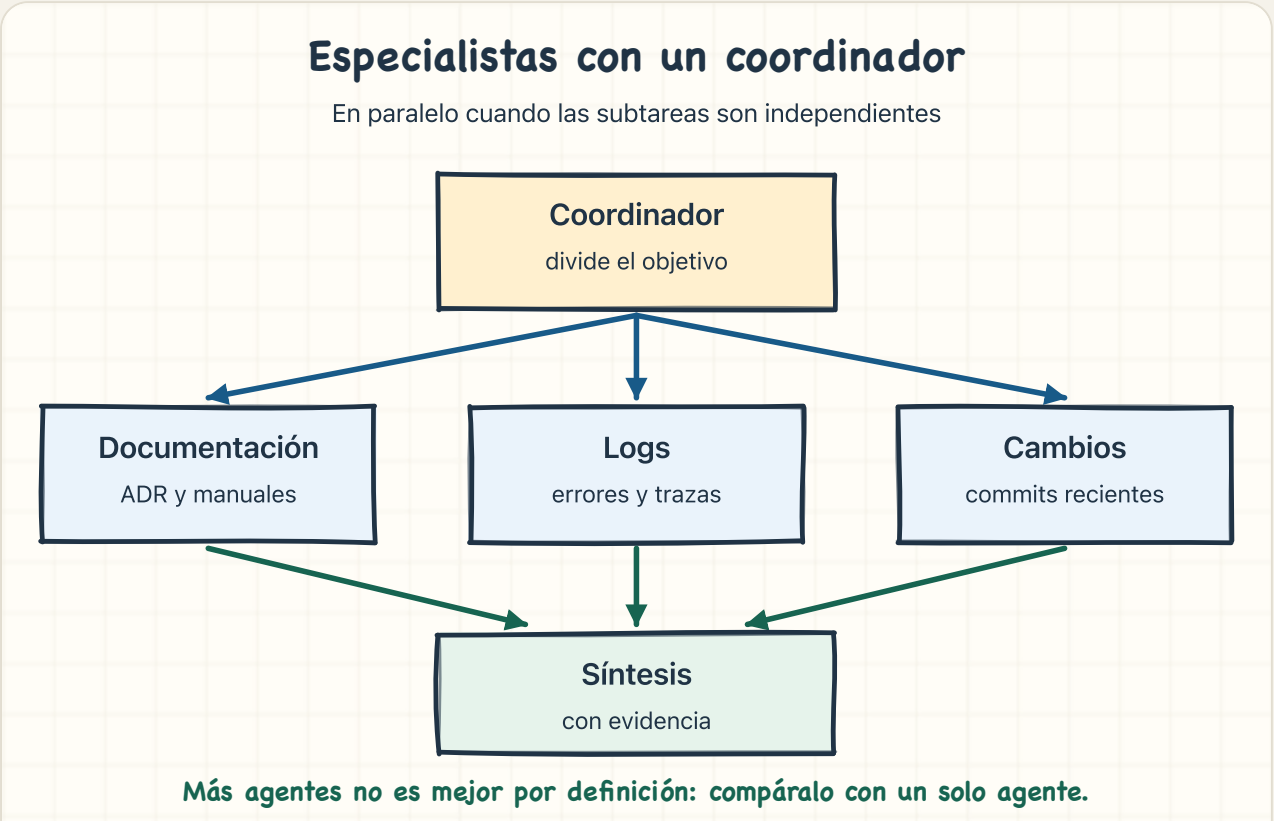
> **Lo que debes llevarte:** define límites de iteraciones, de tiempo y de presupuesto, y hazlos cumplir en el código. Un bucle sin límites no es un agente: es un ciclo con factura.

---

06 · Multi-Agent RAG: investigaciones especializadas en paralelo

El problema: algunas investigaciones necesitan herramientas, fuentes y subtareas distintas e independientes entre sí.

En el asistente de guardia, ya no a las dos de la mañana sino en la revisión posterior: "¿qué causó la caída de pagos del viernes?". Hay que revisar la documentación de arquitectura, los logs y los cambios de código recientes. Son tres búsquedas que no dependen una de otra y que piden herramientas diferentes.



Las subtareas independientes pueden ejecutarse en paralelo.

Cómo funciona

Un agente coordinador divide el objetivo en subtareas. Cada especialista consulta sus fuentes y devuelve hallazgos con referencias. El coordinador integra la evidencia, detecta las contradicciones y prepara la respuesta. Cuando las subtareas son independientes, corren en paralelo.

Que funcione no lo hace gratis. En el sistema de investigación de Anthropic, la versión multiagente (Claude Opus 4 como coordinador y subagentes Claude Sonnet 4) superó en un 90,2 % a Claude Opus 4 trabajando solo, en su evaluación interna. El precio: un agente gasta unas 4 veces los tokens de una conversación de chat, y el sistema multiagente, unas 15 veces.

En su propio análisis, sobre la prueba BrowseComp, la cantidad de tokens gastados explicaba el 80 % de la variación del rendimiento: buena parte de la mejora consiste en gastar más (Anthropic, *How we built our multi-agent research system*, 2025). El mismo artículo advierte que encaja mal cuando todos los agentes necesitan compartir el mismo contexto o cuando hay muchas dependencias entre ellos.

Cuándo usarlo	Sus límites
Investigaciones amplias, herramientas heterogéneas, fuentes independientes y tareas con un paralelismo claro.	Más complejidad de coordinación, de observabilidad y de control de costos. Varios agentes no garantizan un mejor resultado.

> **Nota:** Agentic RAG y Multi-Agent RAG no son lo mismo. El primero trata de decisiones dinámicas sobre la recuperación; el segundo, de coordinar varios agentes. Un sistema puede tener uno sin el otro.

> **Lo que debes llevarte:** antes de justificar una arquitectura multiagente, compárala contra un solo agente bien diseñado, con las mismas preguntas y midiendo el costo.

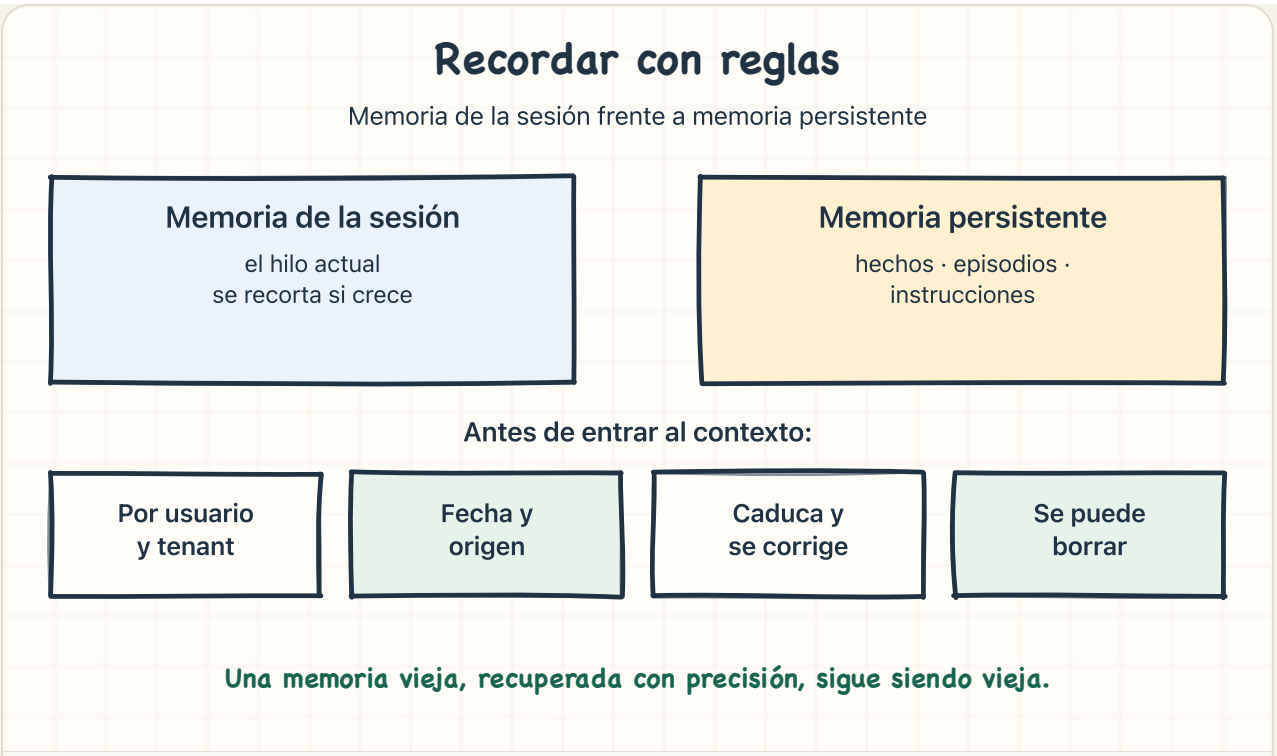
---

07 · Memory RAG: recuperar lo que se decidió antes

El problema: una conversación nueva puede depender de decisiones o datos que se compartieron en otra.

En el asistente de guardia: "la semana pasada decidimos dejar el pool de pagos en 20 conexiones hasta migrar la base de datos. ¿Sigue vigente?". Sin acceso a esa decisión, el modelo no puede contestar de forma fiable: en el mejor caso dice que no sabe; en el peor, lo inventa.

"Memory RAG" no es un término estándar: lo uso para nombrar el patrón de recuperar memoria con las mismas técnicas con las que recuperas documentos.



Una memoria vieja, recuperada con precisión, sigue siendo vieja.

Cómo funciona

El modelo no recuerda nada entre llamadas: lo que recuerda es la aplicación, que guarda y recupera. Distingue dos memorias. La **memoria de la sesión** es el hilo de la conversación actual y se recorta cuando crece demasiado. La **memoria persistente** sobrevive entre conversaciones: guarda hechos, decisiones o resúmenes con un identificador, una fecha, su origen y reglas de conservación.

LangGraph, por ejemplo, divide la memoria persistente en semántica (hechos), episódica (experiencias pasadas) y procedimental (las instrucciones que guían al agente). Cuando llega una consulta nueva, el sistema recupera solo los recuerdos pertinentes que ese usuario está autorizado a ver.

Una memoria también se puede envenenar. Si el asistente guarda cualquier cosa que alguien afirme en una conversación, una frase falsa se convierte en un "hecho" que vuelve en todas las siguientes. En el asistente de guardia, solo guardaría decisiones confirmadas, con quién las tomó y cuándo vencen.

Cuándo usarlo	Sus límites
Asistentes personales, proyectos largos, soporte técnico recurrente y sistemas que necesitan continuidad.	Memorias obsoletas o envenenadas, resúmenes incompletos, riesgos de privacidad y fugas de información entre usuarios.

> **Lo que debes llevarte:** define reglas explícitas de actualización, caducidad, corrección, borrado y aislamiento por usuario o tenant antes de guardar el primer recuerdo.

Cómo elegir una arquitectura

La pregunta correcta no es qué arquitectura es más avanzada. Es esta: **¿qué problema de recuperación tiene mi sistema, y cómo voy a comprobar que la solución lo corrige?**

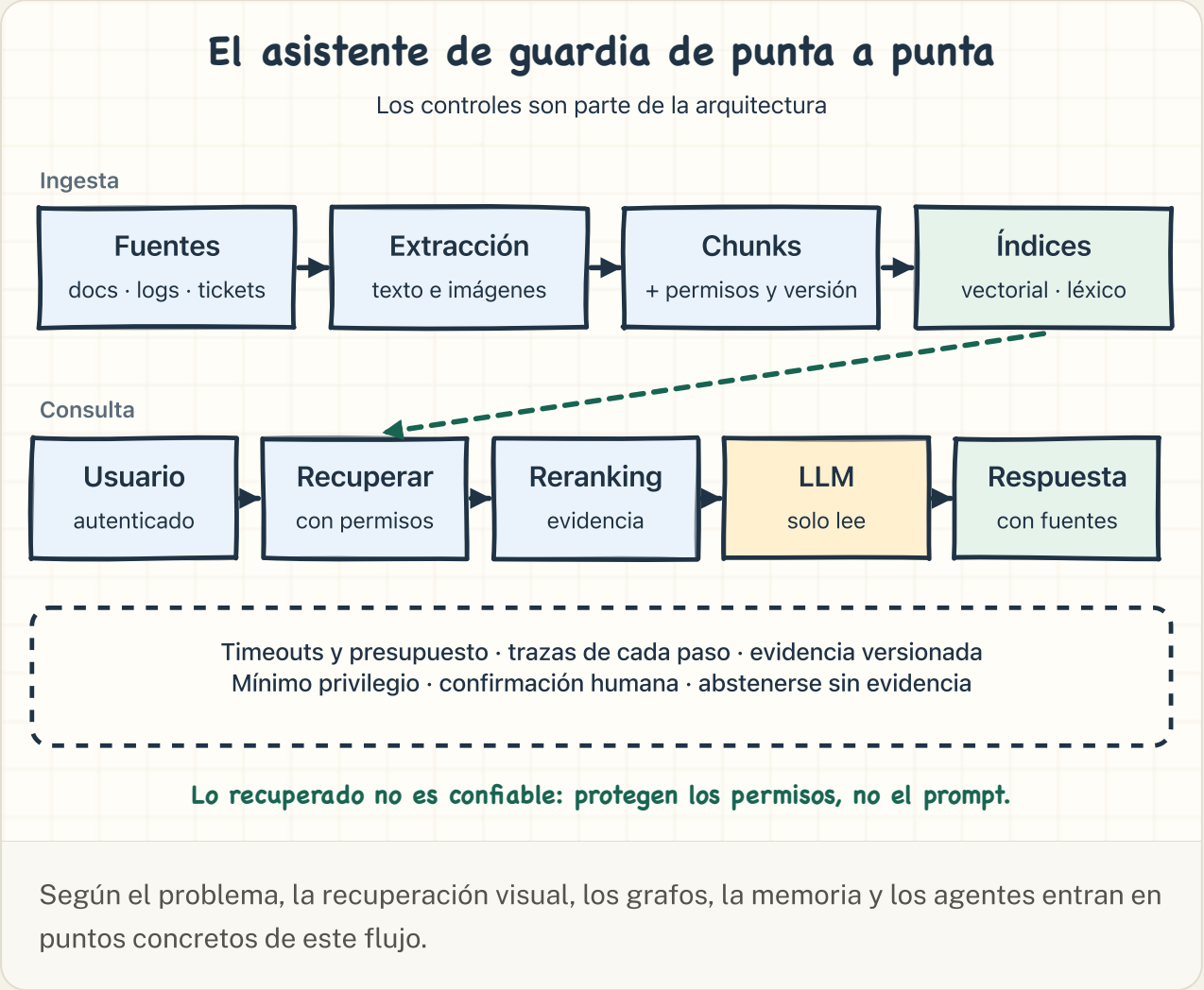
Problema	Arquitectura candidata	Cómo lo validas
Chunks ambiguos	Contextual RAG	Recall@k con preguntas ambiguas
Tablas y diagramas	Multimodal RAG	Exactitud sobre preguntas visuales y documentales
Códigos e identificadores no encontrados	Hybrid Search	Recall@k y nDCG@k
Relaciones entre entidades	GraphRAG	Exactitud de las relaciones y de las preguntas de varios saltos
Una consulta no alcanza	Agentic RAG	Tasa de resolución, costo y latencia
Investigaciones paralelizables	Multi-Agent RAG	Calidad frente a un solo agente
Pérdida de continuidad	Memory RAG	Recuperación correcta y privacidad

En una aplicación real se combinan. Contextual RAG puede enriquecer los chunks, la búsqueda híbrida recuperar los candidatos y un agente decidir si hace falta otra consulta. No existe ninguna obligación de usar las siete.

> – **Para profundizar**
>
> con ventanas de contexto de cientos de miles de tokens, la objeción es obvia: ¿para qué recuperar, si puedo meter todo? A veces puedes. Anthropic lo dice en el mismo artículo de *Contextual Retrieval*: si tu base de conocimiento tiene menos de 200.000 tokens (unas 500 páginas), meterla entera en el prompt, con prompt caching, puede ser más simple que construir un RAG. Si el asistente de guardia solo tuviera los cuatro runbooks, empezaría por ahí.
>
> Con años de tickets y logs, no. Primero, porque no caben en ninguna ventana. Segundo, porque el costo y la latencia crecen con cada token de cada consulta. Tercero, porque los permisos se aplican por documento, y no puedes darle al modelo lo que el usuario no debe ver. Y cuarto, porque los modelos han aprovechado peor la evidencia que queda en medio de un contexto largo que la que está al principio o al final (Liu et al., *Lost in the Middle*, TACL 2024). Ese estudio se hizo con modelos de entonces: mídelo con el tuyo antes de darlo por hecho.

Cómo se vería en una empresa

Así se ve el asistente de guardia de punta a punta, con la ingesta, la consulta y los controles que lo atraviesan:

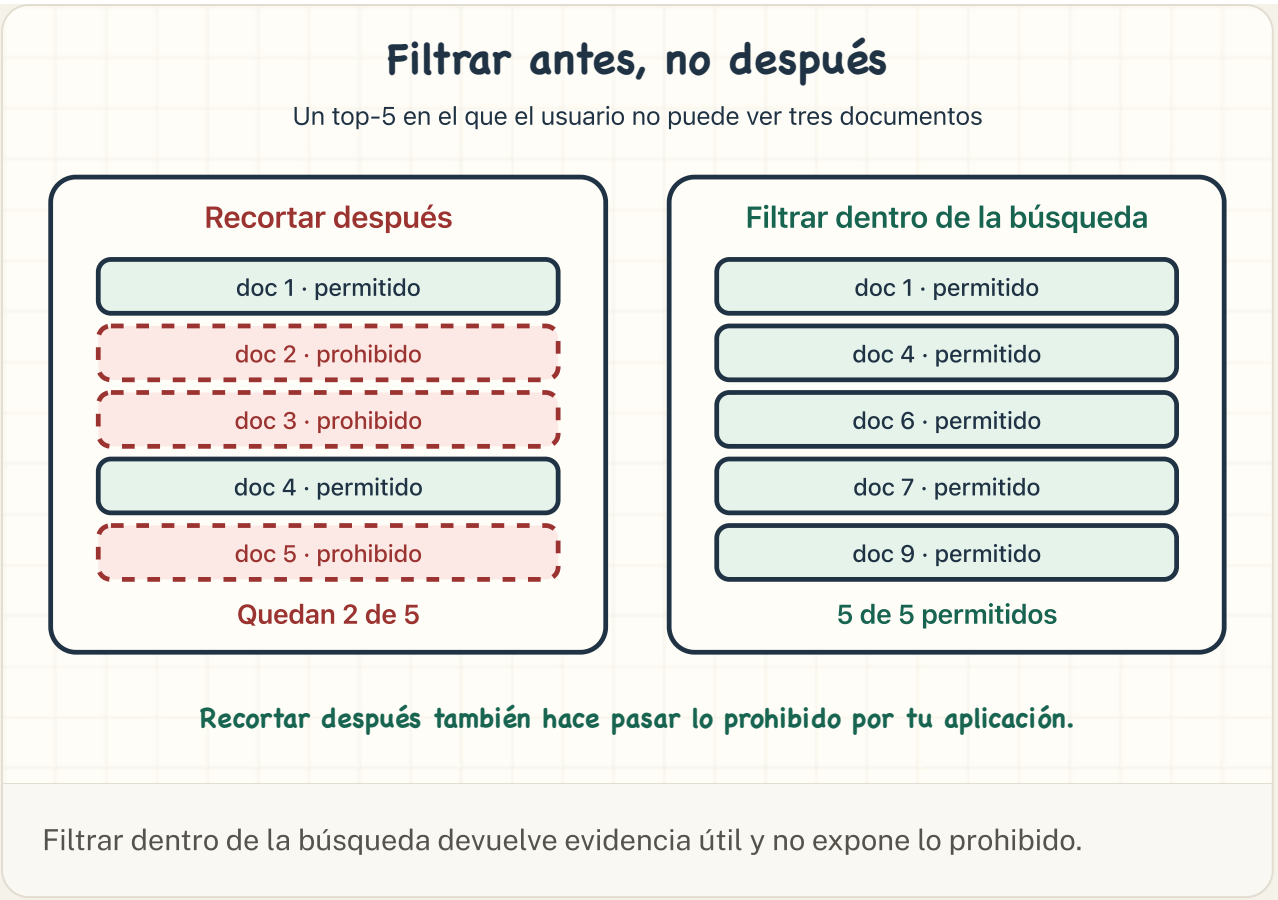


Según el problema, la recuperación visual, los grafos, la memoria y los agentes entran en puntos concretos de este flujo.

Los permisos se aplican al buscar, no después

En el caso de la tienda, los tickets de incidentes de seguridad solo los puede ver el equipo de seguridad. La tentación es recuperar el top-k y después quitar lo que el usuario no puede ver. Es el control más fácil de implementar mal, porque la versión incorrecta funciona en la demo.

Tiene dos problemas: te quedas con menos evidencia de la que pediste, y los documentos prohibidos ya pasaron por tu aplicación, por tus logs y quizás por una caché.



Filtrar dentro de la búsqueda devuelve evidencia útil y no expone lo prohibido.

Lo correcto es guardar los permisos de cada chunk en el índice y filtrar dentro de la propia búsqueda. Ten en cuenta que esos permisos valen desde la última sincronización: si alguien pierde acceso en el sistema de origen, el índice solo lo sabe cuando se actualiza (Microsoft Learn, *Document-level access control in Azure AI Search*).

Un detalle de implementación: en los índices vectoriales aproximados, como HNSW, un filtro muy restrictivo puede devolver menos resultados de los pedidos o perder recall, según cómo lo aplique tu base. Mide la recuperación también con los usuarios que menos ven.

Los controles que no se negocian

Control	Por qué
Autenticación y autorización antes de recuperar	El retriever no debería ver lo que el usuario no puede ver.
Filtro por usuario, tenant y documento	Los permisos viven en el índice, junto a cada chunk.
La identidad del usuario llega a cada herramienta	Cada búsqueda, herramienta o subagente consulta con los permisos de quien pregunta, no con una cuenta de servicio que lo ve todo.
Contenido recuperado tratado como no confiable	Un documento, un ticket o un log puede traer instrucciones escondidas: es la prompt injection indirecta (OWASP LLM01).

Control	Por qué
Versionado y trazabilidad de la evidencia	Para saber qué documento, en qué versión, sostuvo cada respuesta.
Borrado que llega a todas las copias	Un documento borrado tiene que desaparecer de los chunks, los embeddings, el grafo y las cachés. Y una caché de respuestas incluye los permisos en su clave.
Timeouts, límites y presupuesto	Para que una consulta no cueste diez veces lo previsto.
Métricas y trazas de cada paso	Sin ellas no sabrás si falló la recuperación o la generación.
Abstención explícita	Si falta evidencia, el sistema lo dice en vez de rellenar.

> **Cuidado:** ninguna instrucción en el prompt garantiza que el modelo ignore un texto malicioso dentro de un documento recuperado. Y en el asistente de guardia, cualquiera que abra un ticket o provoque un log escribe en tu contexto. La defensa no puede depender del modelo: depende de la arquitectura.
>
> - **Mínimo privilegio:** el asistente lee; no reinicia servicios ni cambia configuración.
> - **Confirmación humana** antes de cualquier acción con efectos, como cerrar un ticket.
> - **Separar** el contenido recuperado de las instrucciones del sistema y marcarlo como no confiable.
> - **Limpiar la salida:** no mostrar enlaces ni imágenes hacia dominios no aprobados, que pueden servir para sacar datos.
>
> OWASP dedica además una entrada propia a los riesgos de los vectores y los embeddings (LLM08): fugas entre tenants, datos envenenados y permisos mal aplicados en la base vectorial.

Cómo evaluar un RAG de verdad

Una respuesta convincente no demuestra calidad. La evaluación tiene que mirar tres capas por separado, porque cada una falla por motivos distintos.

1. La recuperación. ¿Trajo la evidencia correcta?

Métrica	Qué mide
Recall@k	Qué proporción de la evidencia relevante aparece entre los primeros k resultados.
Precision@k	Qué proporción de los k resultados es relevante.
nDCG@k	Si lo relevante aparece arriba: premia la relevancia y castiga la mala posición.

Un ejemplo con el asistente de guardia. Para la pregunta "¿por qué HikariPool-1 da timeout en pagos?" marcaste cuatro piezas de evidencia: la sección del runbook de PaymentService, el ticket INC-2207, el ADR que fijó el tamaño del pool y el log del 12 de octubre. Si los diez primeros resultados traen tres de ellas, Recall@10 = 3 / 4 = 0,75.

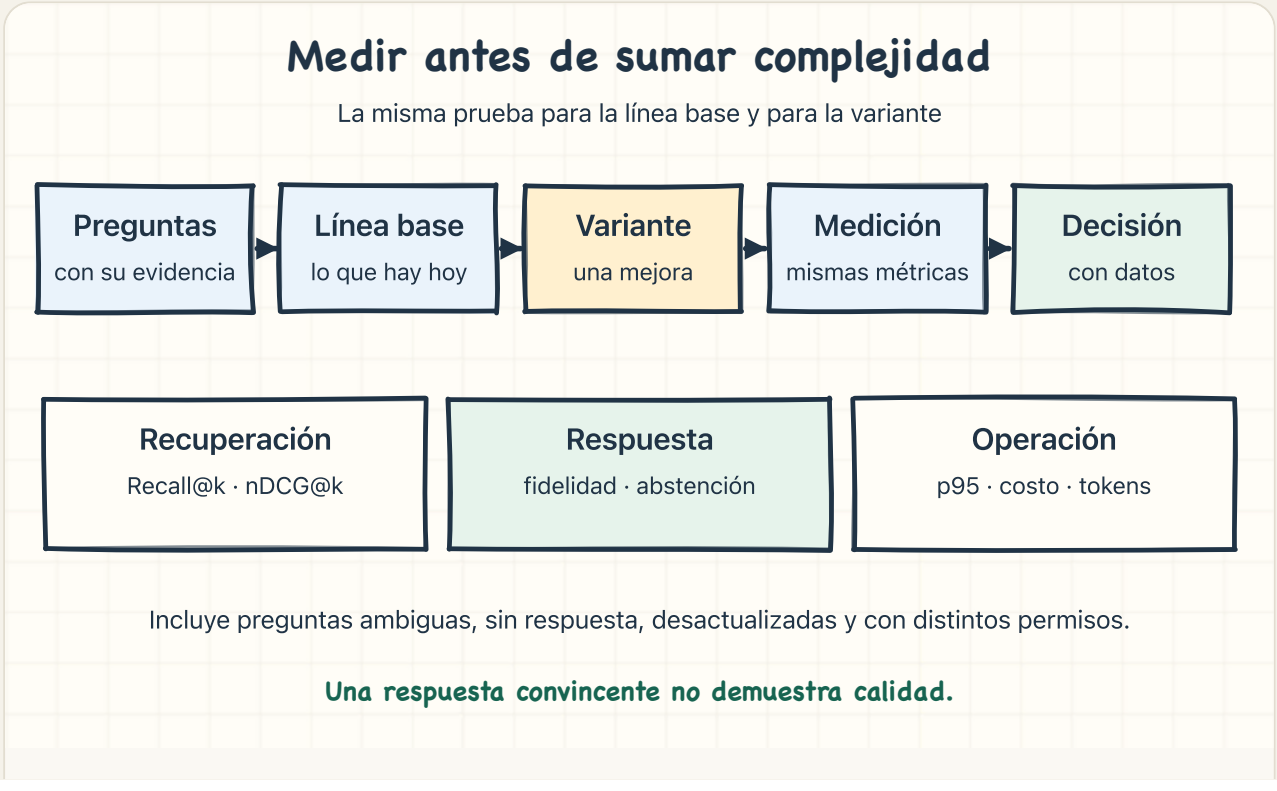
2. La respuesta. ¿Usó bien esa evidencia?

Métrica	Qué mide
Fidelidad (*faithfulness*)	Cuántas afirmaciones de la respuesta están respaldadas por la evidencia entregada.
Citas correctas	Si cada cita apunta de verdad al fragmento que sostiene esa afirmación.
Relevancia de la respuesta	Si la respuesta atiende de verdad la pregunta.
Corrección	Si coincide con una respuesta o un hecho de referencia verificable.
Abstención	Si reconoce cuándo la información disponible no permite responder.

Herramientas como Ragas implementan varias de estas métricas, y en las versiones recientes llaman *Response Relevancy* a la relevancia de la respuesta.

3. La operación. ¿Se puede sostener en producción? Mide la latencia **p95** y **p99**, el tiempo por debajo del cual quedan el 95 % y el 99 % de las consultas. Suma el costo por consulta, los tokens consumidos, el número de búsquedas y de llamadas a herramientas, la tasa de errores y timeouts, la frescura y la cobertura de los índices, y las violaciones de autorización.

Las evaluaciones automáticas con un modelo de lenguaje como juez son útiles, pero no son una garantía. Contrástalas con pruebas de referencia y con revisión humana cuando el riesgo lo justifique.



Evaluar antes de adoptar más complejidad.

El conjunto de prueba

Todo lo anterior necesita un conjunto de preguntas con su evidencia marcada. En el asistente de guardia, las preguntas salen del historial de incidentes. Yo empezaría con unas cincuenta y crecería desde ahí. Tienen que incluir preguntas sencillas, ambiguas, sin respuesta, con datos desactualizados y con distintos permisos.

Marca la evidencia por documento o por sección, no por chunk: si cambias el chunking, las marcas por chunk dejan de valer. Y no existe una configuración universal de tamaño de chunk, top-k, número de agentes o iteraciones: se ajustan con tus datos, tus objetivos de servicio y tus restricciones.

Mitos frecuentes

Estas ideas circulan mucho cuando se habla de RAG. Todas tienen algo de cierto, y por eso confunden.

Lo que se suele decir	Lo que es cierto
"Naive RAG es lo más débil y hay que superarlo."	Para colecciones pequeñas y preguntas simples, puede ser suficiente. Es la línea base contra la que se mide todo lo demás.
"Con un modelo multimodal se acaban los problemas de extracción."	Cambia el tipo de error, no lo elimina: un modelo visual también lee mal cifras y relaciones.
"GraphRAG siempre usa comunidades."	Depende de la implementación. Las comunidades son el enfoque de Microsoft para preguntas globales.
"Un agente busca cuantas veces haga falta."	Busca cuantas veces lo dejes. Sin límites de tiempo, costo e iteraciones impuestos por el código, es un riesgo operativo.
"Varios agentes especializados rinden más que uno."	A veces, y a un costo varias veces mayor. Hay que compararlo con un solo agente bien diseñado.
"La memoria hace que el modelo recuerde las conversaciones."	El modelo no recuerda: la aplicación guarda y recupera. Y sin vigencia, corrección, borrado y aislamiento, esa memoria es un riesgo de privacidad.
"Con contexto largo ya no hace falta recuperar."	Con una base pequeña, puede ser cierto. Con una grande, el tamaño, el costo, los permisos y la pérdida de atención en medio del contexto siguen ahí.

Cómo quedaría el asistente de guardia

Si tuviera que construir este asistente, no empezaría por la arquitectura más llamativa. Lo haría en este orden, y midiendo cada paso contra el anterior:

1. **La línea base y el conjunto de prueba.** Naive RAG y cincuenta preguntas reales con su evidencia marcada. Sin esto, ningún paso siguiente se puede justificar.
2. **Chunking por secciones y metadatos:** servicio, entorno, versión y permisos en cada chunk.
3. **Hybrid Search con reranker**, porque en guardia muchas preguntas traen un identificador o un mensaje de error.
4. **Contextual RAG**, solo si los runbooks se siguen confundiendo después de filtrar por servicio.
5. **Agentic RAG, o mejor un workflow**, para las preguntas del tipo "¿por qué falló este pedido?", con límites en el código.

Y lo que dejaría fuera hasta que una medición lo pida. Multimodal, solo para los diagramas que existen únicamente como imagen. GraphRAG, desde el catálogo de servicios y no desde los textos. Multi-Agent RAG, solo para las revisiones posteriores, nunca a las dos de la mañana. Y memoria, solo para decisiones confirmadas y con fecha de vencimiento.

> **Lo que debes llevarte:** un sistema RAG confiable no es el que genera la respuesta más convincente, sino el que puede explicar con qué evidencia la construyó, qué información le falta y cuándo debe abstenerse.

Ahora piensa en tu propio RAG, o en el que estás por construir: ¿cuál es la pregunta real que hoy responde mal, y en cuál de estas siete fallas cae? Cuéntamelo en los comentarios; me interesa saber qué falla aparece más.

Fuentes

1. Anthropic, Contextual Retrieval (2024).
2. Faysse et al., ColPali: Efficient Document Retrieval with Vision Language Models, ICLR 2025.
3. Cormack, Clarke y Büttcher, Reciprocal Rank Fusion outperforms Condorcet and individual rank learning methods, SIGIR 2009.
4. Elastic, Hybrid search.
5. Microsoft, GraphRAG.
6. Anthropic, Building effective agents (2024).
7. Asai et al., Self-RAG (2023), y Yan et al., Corrective Retrieval Augmented Generation (2024).
8. LangGraph, Agentic RAG y Memory overview.
9. Anthropic, How we built our multi-agent research system (2025).

10. Liu et al., Lost in the Middle: How Language Models Use Long Contexts, TACL 2024.
11. Ragas, Available metrics.
12. OWASP, Top 10 for LLM Applications 2025.
13. Microsoft Learn, Document-level access control in Azure AI Search.