Facsímil 04

La caja de herramientas

Capítulo 07

Embeddings aplicados y búsqueda semántica

686f6c61 · 2026

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 `input_type` , 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 `input_type` 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:[1][2]

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

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:

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

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 `float32`, 2 para `float16`.

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.

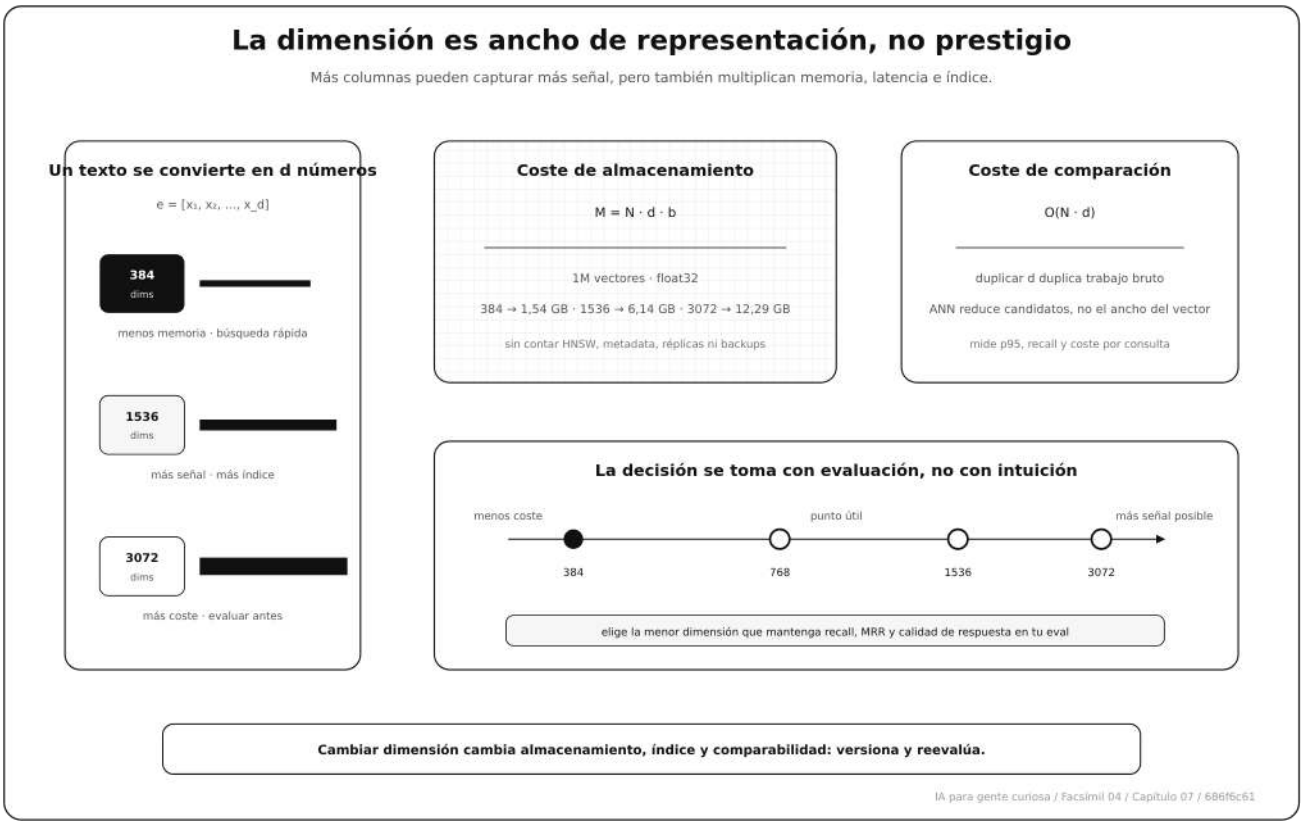
El coste de comparación exacta crece igual:

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

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 → 512 → 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.	$0{,}74 / (0{,}824 \cdot 0{,}906) = 0{,}992$.

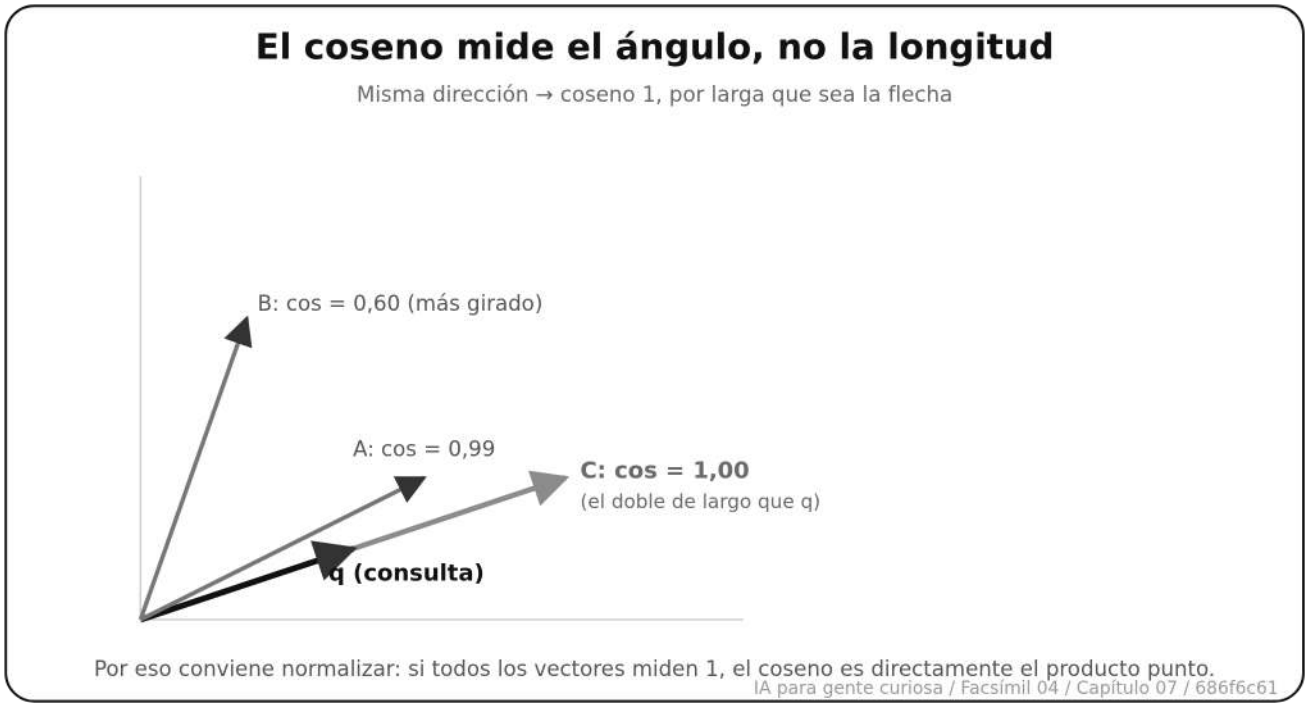
Si normalizas todos los vectores para que tengan norma 1, el coseno se convierte en producto punto:

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

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

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 *maximum inner product search*, 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:

$$\mathrm{TopK}(q, D, k) = \{d_{(1)}, \ldots, d_{(k)}\}, \quad s(q, d_{(1)}) \geq \cdots \geq s(q, d_{(k)})$$

SÍMBOLO	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.

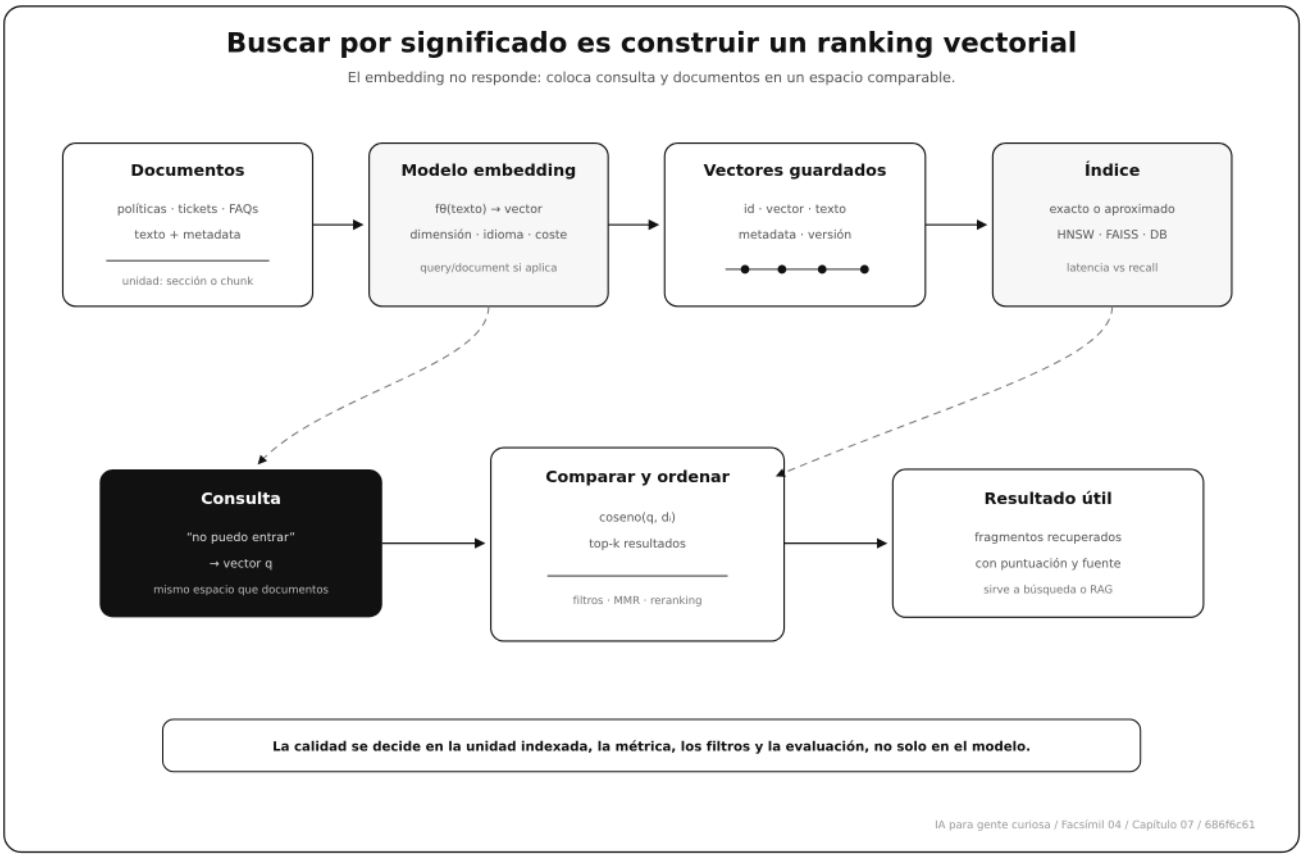
El proceso completo

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.	`input_type=query` 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:

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

SÍMBOLO	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 `float32`.

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 k 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. Cuándo **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:

$$\mathrm{DCG@k} = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

$$\mathrm{nDCG@k} = \frac{\mathrm{DCG@k}}{\mathrm{IDCG@k}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
rel_i	Relevancia del resultado en posición i.	2 si responde, 1 si ayuda, 0 si no sirve.
$\mathrm{DCG@k}$	Ganancia acumulada con descuento por posición.	Premia relevancia arriba.
$\mathrm{IDCG@k}$	DCG ideal si el ranking fuera perfecto.	Mejor orden posible para esa consulta.
$\mathrm{nDCG@k}$	DCG normalizado entre 0 y 1.	1 significa orden ideal.

Una evaluación seria de embeddings debería guardar:

CAMPO	EJEMPLO	POR QUÉ
`query_id`	`q-014`	Permite repetir y auditar resultados.
`query_text`	“no puedo entrar al campus”	La consulta real que hizo una persona o un caso diseñado.
`positive_ids`	`["doc-01"]`	Documentos que deben aparecer.
`graded_relevance`	`{"doc-01": 2, "doc-05": 1}`	Diferencia evidencia principal de apoyo parcial.
`hard_negative_ids`	`["doc-03"]`	Documentos parecidos que no responden.
`filters`	`{"curso": "2026"}`	Condiciones de producto o permisos.
`embedding_model`	`all-MiniLM-L6-v2`	La evaluación depende del modelo.
`dimension`	`384`	Cambiar dimensión puede cambiar ranking.
`top_k`	`3`, `5`, `10`	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 encontraste 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

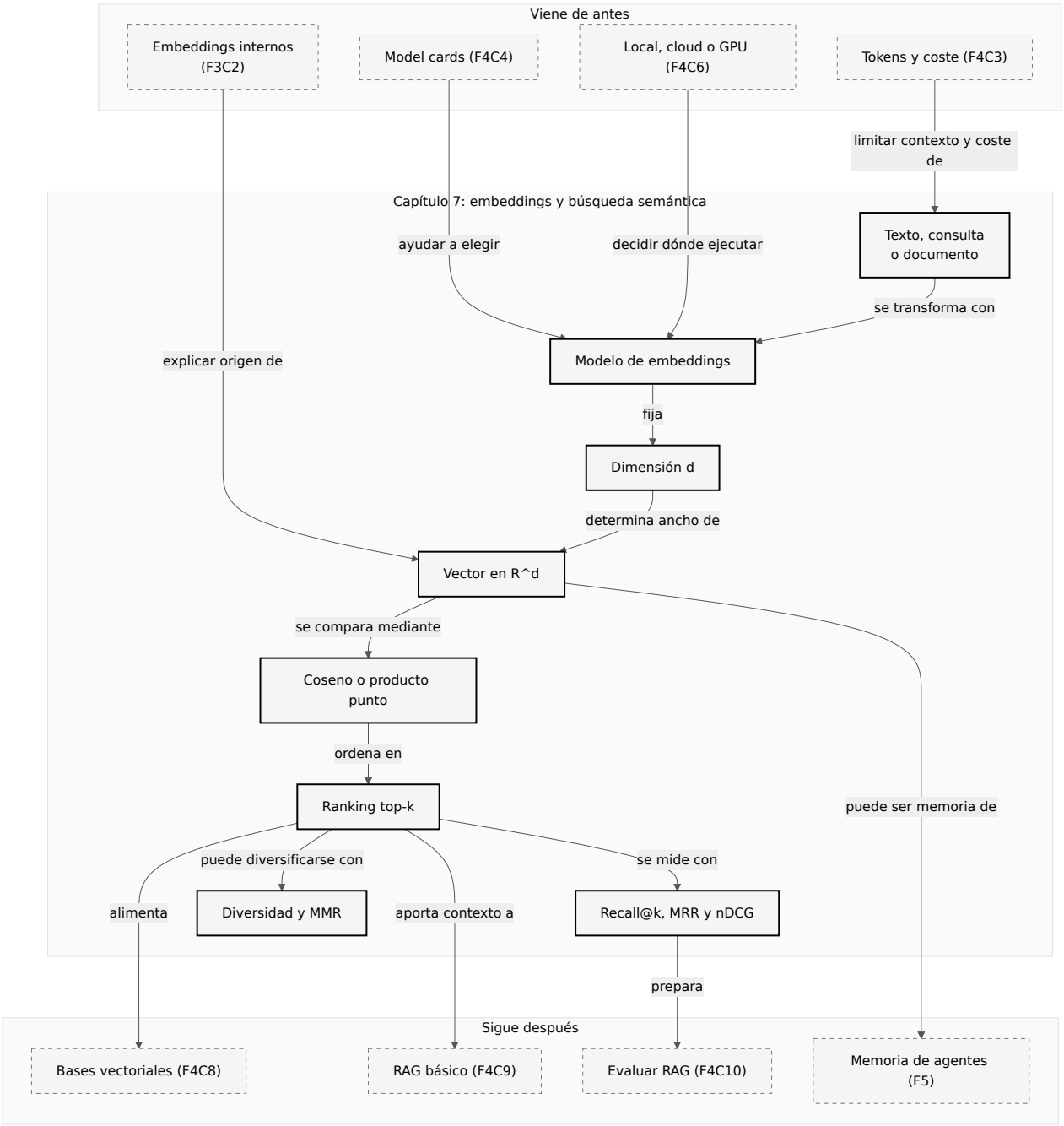
Embeddings son la puerta de entrada a casi todo lo que se vende como “IA con tus datos”. Si no entiendes esta pieza, no sabes si tu RAG falla por el modelo generativo, por el chunking, por la base vectorial, por la métrica o por la evaluación.

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. ↩

.0. 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. ↩

.1. 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. ↩

.2. 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. ↩

.3. 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. ↩

.4. Cohere (2026), documentación de embeddings citada en la tabla de estado del arte. ↩

.5. Aditya Kusupati y otros (2022). *Matryoshka Representation Learning*. NeurIPS. https://arxiv.org/abs/2205.13147. ↩

.6. 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/. ↩

20. 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. ↩

21. 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. ↩