ChatGPT

RAG Híbrido: Búsqueda Vectorial + Léxica con Reranking Cross-Encoder

Retrieval-Augmented Generation (RAG) híbrido combina métodos de búsqueda densa (vectorial) y dispersa (léxica) para aprovechar lo mejor de ambos mundos. En términos simples, un sistema RAG híbrido ejecuta en paralelo dos búsquedas sobre la base de conocimiento: una por similitud semántica (embeddings vectoriales) y otra por coincidencia de palabras clave (ej. BM25). Luego fusiona los resultados y los reranquea para presentar al modelo generativo los fragmentos de documento más relevantes [1] [2]. La motivación es que la búsqueda densa captura similitudes conceptuales (sinónimos, parafraseo, contexto), mientras que la búsqueda léxica es precisa encontrando términos exactos (nombres propios, códigos, cifras) [3]. **Al combinar ambas** se logra una recuperación de evidencias más completa y fiable, elevando tanto el *recall* como la *precisión*, lo cual reduce lagunas de información y mitiga alucinaciones del LLM [4] [5]. De hecho, implementar búsqueda híbrida "ya no es un extra experimental sino una necesidad competitiva" en aplicaciones de IA que manejan conocimiento complejo [6], con estudios reportando mejoras significativas de rendimiento frente a enfoques de solo vector o solo BM25 [7].

Implementación y mejores prácticas: En la práctica se construyen dos índices en la etapa de ingesta: por ejemplo, un índice invertido de texto (p. ej. Elasticsearch, Lucene/OpenSearch) para BM25 y un índice vectorial (FAISS, HNSW, etc.) para embeddings [8]. Al llegar una consulta, se **ejecutan ambas búsquedas en paralelo** sobre sus respectivos índices y luego se fusionan los resultados en una sola lista ordenada [9] [10]. Existen varias técnicas de fusión: desde normalizar y sumar puntuaciones ponderadas, hasta algoritmos robustos como *Reciprocal Rank Fusion (RRF)* que combina rankings de forma sencilla pero eficaz [11] [10]. Muchos motores soportan ya consultas híbridas de forma nativa – por ejemplo, Azure Cognitive Search y MongoDB Atlas Search permiten en una sola query combinar texto completo y vectorial, aplicando RRF internamente para devolver el top-N [10]. Si la plataforma no soporta híbrido nativo, se puede orquestar manualmente: ejecutar por separado la búsqueda vectorial y la léxica, luego **agregar y reranquear los resultados manualmente** antes de pasarlos al prompt [12]. Esto da flexibilidad para ajustar la combinación o aplicar lógica extra.

Un posible *pipeline* híbrido en código Python, usando herramientas existentes, es el siguiente: por ejemplo con **LangChain** podemos combinar un recuperador BM25 y otro vectorial, y realizar consultas conjuntas fácilmente [13] [14]:

```
from langchain.retrievers import BM25Retriever, EnsembleRetriever
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings

# Preparar documentos e índices
documents = [doc1_text, doc2_text, ...]
bm25 = BM25Retriever.from_texts(documents)  # Índice BM25 en memoria
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
vec_store = FAISS.from_texts(documents, embedding=embeddings)
dense = vec_store.as_retriever(search_type="similarity", search_kwargs={"k":
5})
```

```
# Combinar ambos retrievers con pesos iguales
hybrid = EnsembleRetriever(retrievers=[bm25, dense], weights=[0.5, 0.5])
results = hybrid.get_relevant_documents("mi consulta")
for res in results:
    print(res.metadata, res.page_content)
```

En este ejemplo, se obtienen resultados híbridos combinados de BM25 y búsqueda vectorial [15]. Frameworks populares como LangChain, Haystack o Weaviate ya incorporan **búsqueda híbrida como funcionalidad de primera clase**, facilitando mucho la integración [8] [16]. Por ejemplo, Weaviate y MongoDB Atlas permiten cargar textos con embeddings y a la vez indexar campos para texto completo, ejecutando consultas semánticas y léxicas en un solo paso. Con soluciones todo-en-uno se evita la complejidad de mantener **múltiples bases** (una vectorial y otra de texto) y sincronizarlas [17] [18]. No obstante, usar herramientas separadas (ej. ChromaDB para vectores + Whoosh/Elasticsearch para BM25) también es viable; la clave es asegurarse de indexar exactamente el mismo corpus en ambos para que los documentos devueltos existan en ambos lados [19]. Una receta común es: generar embeddings con un modelo (ej. *text-embedding-ada-002* de OpenAI), almacenar vectores en un vector store **Chroma** (en entorno de desarrollo) o un servicio gestionado (Pinecone, Redis, etc.), y paralelamente indexar los textos en un motor BM25 (Whoosh, Elasticsearch/OpenSearch). Luego, mediante una librería de orquestación como LangChain, lanzar la consulta a ambos índices y combinar los resultados [16]. En producción, el usuario menciona usar **MongoDB Atlas Vector Search** que soporta vector + filtro + texto; Atlas efectivamente facilita consultas híbridas nativas con sintaxis de agregación ($search + $vectorSearch) aplicando RRF para fusionar resultados [10]. Esto simplifica la arquitectura y garantiza consistencia, aprovechando además los **filtros por metadatos** robustos de Atlas para restringir búsquedas.

Reranking con cross-encoder: Una vez obtenidos los candidatos híbridos, es **buena práctica aplicar un reranqueador más preciso** antes de escoger los documentos finales que verá el LLM [2]. Típicamente se recuperan quizás los top-50 documentos combinados por similitud, y luego se pasan por un modelo *cross-encoder* que evalúa cada párrafo con la pregunta de forma conjunta (atendiendo a la interacción completa entre query y texto) para asignar una puntuación de relevancia más fina [20]. Los cross-encoders (basados en Transformers tipo *ms-marco-MiniLM-L6* o *monoT5*) suelen producir rankings de mucha mayor precisión que la búsqueda por separado, ya que consideran el contexto completo de la consulta y el documento token a token [21]. El costo computacional es alto (hay que ejecutar un forward-pass por documento), por lo que **solo se usan en reranking sobre un conjunto reducido**. En código Python, por ejemplo, se podría usar `CrossEncoder` de HuggingFace SentenceTransformers así [22]:

```
from sentence_transformers import CrossEncoder
model = CrossEncoder('cross-encoder/ms-marco-MiniLM-L6')
cross_scores = model.predict([[query, doc] for doc in candidate_docs])
# Ordenar los documentos por la puntuación obtenida
top_idx = cross_scores.argsort()[::-1][:N]
reranked_docs = [candidate_docs[i] for i in top_idx]
```

Este proceso reevalúa los candidatos y **mejora la precisión** final del contexto suministrado al LLM [20]. En caso de no usar un cross-encoder, algunas plataformas ofrecen *reranqueo semántico* propietario (por ejemplo, Azure Cognitive Search integra un *semantic ranker* estilo Bing) [23], o se puede incluso utilizar el propio LLM para evaluar cuáles fragmentos responden mejor la pregunta (aunque esto puede ser más

lento y costoso) [24] [25] . En cualquier caso, el reranking es valioso: **vector y BM25 no siempre aciertan al 100%** en el orden, así que aumentar el número de resultados iniciales (k) y luego filtrar con un modelo de ranking mejora la calidad sin sacrificar recall [26] .

Metadatos ricos y filtros: Un componente crucial en RAG avanzado es aprovechar **metadatos asociados a los documentos** para mejorar la pertinencia. Al indexar, se deben almacenar campos como disciplina, identificador de plano o sección, versión/revisión, normativa, año, DOI, etc. Estos metadatos permiten luego **filtrar las búsquedas** según el contexto o la intención de la pregunta. Por ejemplo, si la pregunta del usuario es sobre normativa, podría aplicarse un filtro `discipline="Normativa"` o `tipo="Reglamento"` para limitar la búsqueda solo a ese subconjunto relevante. Microsoft recomienda usar campos *filterable* en el índice y usarlos para **eliminar datos irrelevantes** y *"recuperar solo los datos que cumplan condiciones específicas"*, lo cual mejora eficiencia y relevancia de resultados [27] . De forma similar, **combinar palabras clave con la búsqueda vectorial** es útil: se pueden extraer del query ciertos términos clave o entidades y usarlos como filtro previo, de modo que la búsqueda vectorial opere solo dentro de un subconjunto acotado del corpus [28] . Esta estrategia híbrida secuencial (primero filtrar por palabra clave, luego rankear por embedding) puede, por ejemplo, asegurar que si la pregunta menciona "HOJA 5 del plano X", se restrinja la búsqueda a documentos cuyo `sheet_id = X` y sección = 5 antes de calcular similitudes, garantizando que los resultados provienen de la parte correcta del documento. En general, los **filtros por metadata** mejoran el foco de las respuestas, y es buena práctica diseñar el esquema de indexación para soportar las facetas necesarias (disciplina, sección, fecha, etc.) según la ontología de los documentos.

Índices por agente (per-agent retrievers): El usuario menciona mantener *"cada agente (Eléctrica, Mecánica, Normativa, Cálculo) con su propio índice/filtros"*. Esto equivale a una segmentación del conocimiento por dominio o tipo de documento. En la industria, una **buena práctica para grandes bases multi-temáticas** es separar el índice vectorial (y textual) en colecciones o espacios distintos por dominio, o al menos etiquetar cada documento con su dominio y aplicar filtros por dominio en la consulta. Esto evita contaminación de resultados (p.ej., que una respuesta normativa aparezca al preguntar algo de cálculos de ingeniería) y permite afinar los parámetros de embedding y búsqueda a la naturaleza de cada subcorpus. Hay dos enfoques: uno es simplemente incluir un filtro de disciplina en cada consulta según corresponda (ej. consultas del agente eléctrico siempre añaden `discipline: "Eléctrica"` al buscar). Otro enfoque más sofisticado es usar un **Router de consultas inteligente**: primero clasificar la pregunta para decidir a qué "agente" o índice especializado enviarla. Herramientas como LangChain ofrecen cadenas multi-ruta donde un LLM de enrutamiento elige entre varios sub-sistemas de QA según la detección de intención/domino [29] . Por ejemplo, ante la pregunta *"¿Cómo leer el diagrama unifilar de potencia?"*, el router podría mapearla al agente de "Eléctrica", mientras que *"¿Qué norma aplica para inspección de soldaduras?"* iría al agente de "Normativa". En caso de duda, se puede definir un *default chain* por defecto [30] . La separación por agentes también puede ser física (índices independientes en Chroma u otra DB para cada dominio) o lógica (un solo índice con metadata filtrable). Lo importante es que **organizar la base de conocimiento por áreas temáticas** suele mejorar la relevancia y la mantenibilidad. Es aconsejable también evaluar métricas por agente – por ejemplo, medir por separado la tasa de respuesta correcta del agente de Cálculo vs el de Normativa – para ajustar parámetros de indexación específicos si un dominio es más exigente (p.ej., textos muy cortos vs muy largos, vocabulario muy técnico, etc.).

En resumen, el estado del arte de RAG enfatiza: **búsqueda híbrida densa+dispersa** (ya considerada *best practice* estándar [31]), uso de **reranqueadores cross-encoder** para máxima precisión, y **uso inteligente de metadatos** y índices especializados para restringir el universo de búsqueda según contexto. Estas prácticas son aplicadas por equipos profesionales para hacer sistemas más robustos y exactos [2] [25] . Como confirma un blog técnico, *"los proveedores cloud ahora promocionan búsqueda*

semántica + keywords en sus ofertas, precisamente porque el híbrido es la aproximación más efectiva en datos del mundo real (donde algunas consultas requieren coincidencias literales y otras semánticas)" [31] .

Capa de Graph-RAG: Incorporando Conectividad Topológica via Neo4j

Para ciertas preguntas complejas, especialmente en dominios técnicos como ingeniería industrial, un mero buscador de texto puede no ser suficiente. Aquí es donde entra la capa opcional de **Graph-RAG**, que integra una base de conocimiento estructurada en forma de *grafo* (Neo4j, Memgraph, etc.) con el pipeline RAG. Un **grafo de conocimiento topológico** puede modelar explícitamente relaciones entre entidades: por ejemplo equipos conectados en un diagrama P&ID, componentes en un circuito eléctrico, rutas de señal entre sensores, PLCs y actuadores, o dependencias funcionales entre sistemas. Este grafo permite responder preguntas de *estructura, impacto y relaciones* de manera más directa que buscando fragmentos de texto dispersos.

Cuándo usar la capa de grafo: El sistema debe **detectar la intención** de una pregunta para decidir si activar el Graph-RAG. Las consultas ideales para el grafo son aquellas que implican recorridos, dependencias o impactos en una red de elementos. Por ejemplo: *"Si cierro la válvula V-203, ¿qué queda sin servicio?"*. Responder esto requiere saber qué flujo corta V-203 y qué equipos están downstream (aguas abajo) de ella en el sistema – una **trazabilidad de impacto** en la red de proceso. Otra: *"Ruta de señal S-14 → PLC → actuador"* implica encontrar cómo viaja la señal S-14 a través del PLC hasta cierto actuador; esencialmente una **búsqueda de camino** en un grafo de automatización. O *"Equipos aguas arriba/abajo de T2"*, que pide listar los equipos conectados antes (upstream) o después (downstream) de un cierto equipo T2. Estas preguntas requieren **razonar sobre conexiones** explícitas, algo que las búsquedas vectoriales puras no hacen bien (no pueden inferir relaciones causales o topológicas solo de texto) [32] [33] . Un grafo de conocimiento, en cambio, modela esas relaciones como aristas, permitiendo consultas estructuradas (p. ej. caminos más cortos, vecinos, componentes dependientes) con total precisión.

Beneficios de GraphRAG: Según expertos, *"una base vectorial es como un buscador semántico – genial para encontrar pasajes similares a tu pregunta, pero no muestra cómo están conectadas las cosas. Un grafo de conocimiento es como un mapa semántico: no solo te dice qué es relevante, sino cómo se relacionan conceptos, entidades y datos entre sí"* [34] . Esto habilita responder preguntas complejas del tipo *"¿Qué servicios están en riesgo si falla X?"* con respuestas estructuradas y trazables [35] . Precisamente, en un tutorial de Neo4j se menciona que esta combinación es muy efectiva cuando la aplicación requiere *"razonar sobre arquitecturas complejas… consultar tanto metadatos estructurados como documentación no estructurada… y combinar múltiples fuentes en un sistema coherente"* [36] . El grafo aporta fortalezas adicionales que van *más allá de RAG vectorial puro*: permite **consultas estructuradas** (Cypher, SPARQL) para conteos o filtrados exactos, brinda **explicabilidad** (podemos seguir las relaciones que llevaron a cierta conclusión), y retiene conocimiento puntual (como relaciones temporales o topológicas) de forma **no fragmentada** [37] [38] . En suma, integrar un grafo solventa limitaciones de la búsqueda vectorial "caja negra" y aporta confianza en dominios donde importa la exactitud y la trazabilidad (industria, finanzas, salud, etc.) [39] [40] . Un blog lo resume así: los KGs aportan relaciones estructuradas que ayudan a la IA a *"conectar conceptos, resolver ambigüedades y aplicar lógica. Combinados con RAG, permiten recuperar conocimiento más efectivamente, fundamentar respuestas en hechos y proveer explicaciones – haciendo los resultados más fiables"* [33] .

Arquitectura Graph-RAG: En la práctica, añadir esta capa implica extender el pipeline RAG estándar con pasos adicionales cuando corresponde. Un flujo típico sería:

1. **Detección de intención**: El sistema (mediante reglas o un modelo de clasificación) identifica que la pregunta involucra conectividad/impacto. Por ejemplo, presencia de términos como *"qué pasa si X falla/cierra"*, *"ruta de señal"*, *"aguas arriba/abajo"*, códigos de equipo o instrumentación (V-203, T2, S-14…) que sugieren entidades del grafo, etc. Esto activa el uso del grafo.

2. **Consulta al grafo (Neo4j/Memgraph)**: Se traduce la pregunta a una o varias consultas de grafo. Puede ser una consulta Cypher parametrizada. Ejemplos: *MATCH p=(:Valve {tag: "V-203"})-[:FEEDS*]->(downstream) RETURN downstream *para obtener todos los nodos aguas abajo afectados por cerrar V-203. O* MATCH path = shortestPath((:Signal {id:"S-14"})-[:FLOWS_THROUGH*]->(:Actuator)) *RETURN path* para obtener la ruta de la señal S-14 hasta un actuador. El grafo devuelve identificadores de los nodos relevantes (equipos, señales, componentes) y posiblemente también alguna propiedad (p.ej. lista de nodos en la ruta, o indicador de cuáles quedan sin servicio).

3. **Recuperación de documentos asociados (sub-RAG)**: Con los resultados del grafo, el sistema reúne la documentación relacionada a esos nodos. Aquí se conecta de nuevo con el **vector store o índice textual**, pero **filtrando por los nodos obtenidos**. Por ejemplo, si el grafo indicó que las bombas P5 y P7 quedan sin servicio al cerrar V-203, entonces se podrían recuperar los manuales u hojas de datos de P5/P7, la sección del P&ID donde aparecen, o informes de mantenimiento relacionados, para dar contexto detallado. En otras palabras, se lanza una **consulta de RAG tradicional acotada** a los documentos de esos componentes. Esto se facilita si en los embeddings/índices cada documento tiene metadatos que lo vinculan a nodos del grafo (ej. un campo `equipment_id` o `sheet_id`). De hecho, una estrategia recomendada es **almacenar el grafo y los vectores integrados**: Neo4j 5+ permite asociar embeddings a nodos o documentos, posibilitando buscar vectores filtrando por subgrafoses primero [41] [42]. Así, podríamos hacer en Neo4j: primero una consulta Cypher para nodos afectados, luego una búsqueda vectorial *dentro* de esos nodos usando un índice HNSW embebido en Neo4j [43]. El resultado es un conjunto de textos altamente específicos y relevantes a la pregunta.

4. **Composición de respuesta**: Finalmente, el LLM genera la respuesta apoyándose en *toda la información recopilada*: tanto los datos estructurados del grafo (que pueden resumirse en la respuesta, por ejemplo "al cerrar V-203, se aislan P5 y P7 dejándolas sin caudal") como los detalles de los documentos aportados (citas de manuales o diagramas que describen esos equipos, para contextualizar). Idealmente, el sistema podría incluso **explicar la ruta** o la relación – esto aumenta la confianza del usuario. Por ejemplo: *"Si se cierra V-203, las bombas P5 y P7 quedan sin servicio porque la válvula V-203 bloquea la línea común que alimenta a ambas (según el diagrama de proceso)* [35] *. En el plano P&ID 1002 se observa que V-203 está aguas arriba de P5 y P7, por lo que su cierre impide el flujo hacia esas bombas."* – incluyendo referencias al plano o norma correspondiente.

Buenas prácticas en Graph-RAG: Construir este sistema requiere diseño cuidadoso del **grafo de conocimiento**. Es vital identificar qué entidades y relaciones modelar: en un P&ID, por ejemplo, nodos podrían ser válvulas, equipos (bombas, tanques), instrumentos, y relaciones como *conecta_a*, *flujo_hacia*, *señal_a*, *depende_de*. Incluir propiedades útiles (p.ej. disciplina, ubicación, tag IDs, etc.) para filtrar. Una vez construido, **mantener sincronizado el grafo con los documentos** es importante: cada nodo debería enlazar a las piezas documentales pertinentes (hojas de datos, secciones de normas, etc.), quizás mediante campos de metadata compartidos (ID único). Neo4j sugiere que al **combinar datos estructurados y no estructurados**, se logran respuestas más completas: *"los documentos por sí solos*

pueden dar resultados fragmentados – un RAG vectorial podría traer trozos relacionados al producto pero perder detalles conectados en otra parte. Incorporar relaciones del grafo permite recomponer esas piezas disjuntas y dar respuestas más completas" [38]. También otorga **transparencia**: podemos trazar exactamente qué nodos y caminos sustentan la respuesta, algo muy valioso en entornos regulados donde necesitamos justificar la información [39].

Otra buena práctica es modularizar la **lógica de consulta al grafo**. Por ejemplo, crear funciones Cypher parametrizadas para patrones comunes (como *impactAnalysis(valve_id)* que devuelva afectados, *signalPath(signal_id)* que calcule la ruta de señal). Así, el agente puede invocar esas funciones según la intención detectada. De hecho, Neo4j ha introducido utilidades como *Graph Data Science* y extensiones para GraphGPT; incluso hay iniciativas de *GraphCypherGPT* para traducir lenguaje natural a Cypher queries, lo cual podría ser explorado para hacer más automático el puente LLM-grafo [44] [45]. No obstante, en el estado actual, es común implementar reglas determinísticas o *prompts* específicos para que el LLM formule la consulta estructurada correcta (posiblemente usando herramientas tipo `langchain_neo4j` que facilita la conexión con Neo4j [46]).

Ejemplo en el mercado: Neo4j ha estado abanderando el concepto de **GraphRAG**. En un tutorial oficial integran Neo4j con LangChain para un caso de **DevOps** (microservicios y tareas) análogo a este problema de ingeniería: construyen un grafo de microservicios con sus dependencias y tickets asociados, y lo combinan con un índice vectorial de texto para responder tanto preguntas estructuradas ("¿Qué servicios están afectados si cae el servicio X?") como no estructuradas (detalles de tickets) [35] [41]. Reportan que GraphRAG permite respuestas más exactas y explicables que un RAG tradicional de solo texto, especialmente en preguntas que requieren **razonamiento sobre múltiples hops** en el grafo o **agregaciones** (por ejemplo "*¿cuántos tickets abiertos dependen de servicios en región A?*") [47]. Del mismo modo, la empresa QED42 destacó que el uso de KGs en RAG ayuda a que la IA *"no solo recupere información sino que realmente la entienda y la aplique"*, lo cual es crucial cuando hay que mantener **coherencia lógica** en la respuesta [48].

En nuestro caso, al usar **Neo4j para conectividad topológica** y **ChromaDB para embeddings**, podemos implementar un GraphRAG eficiente: Neo4j responderá rápidamente qué componentes están conectados o impactados dado un evento, y Chroma (o Atlas en prod) nos dará los textos explicativos. Es importante planificar la **integración de ambas capas**: probablemente querrás que, tras una consulta al grafo, los IDs de nodos obtenidos se pasen como filtro a Chroma (por ejemplo, Chroma permite búsquedas del tipo `query_text` + `where={"equipment_id": {"$in": [...]}}` para filtrar por metadata). Si Atlas Search se usa en producción, genial, porque Atlas soporta filtros estructurados sobre campos (ej. un campo array de IDs de grafo) combinados con vector search. Así se puede lograr en una sola consulta Atlas: primero filtra docs cuyo `equipment_id` esté en la lista de nodos afectados, luego ordena por similitud con la pregunta – una forma de ejecutar el *sub-RAG* de documentos implicados.

Conclusión: La combinación de RAG híbrido con una capa de grafo hace posible un **asistente sobre documentación técnica compleja** que responda tanto a definiciones, procedimientos o normativa (usando el buscador híbrido en texto) como a **preguntas de dependencias e impacto** en sistemas (usando el grafo para razonar). Esta arquitectura de múltiples agentes especializados, búsqueda híbrida y grafo conectivo representa las **mejores prácticas actuales** en la industria para aplicaciones de documentación técnica asistida por IA. Implementaciones recientes muestran que así se logra mayor precisión factual, cobertura completa de información y transparencia en las respuestas [7] [49]. Al seguir estas pautas (Chroma/Atlas para embeddings + BM25, reranking con cross-encoder, metadata rica, y Neo4j para conocimiento estructurado), estaremos alineados con **lo que profesionales líderes están haciendo hoy en día** para llevar la generación de respuestas con LLMs al siguiente nivel de confiabilidad.

Fuentes y Referencias: Las recomendaciones anteriores se basan en publicaciones técnicas y guías de expertos en RAG híbrido y GraphRAG, incluyendo un detallado survey por Adnan Masood, PhD [4] [50], el blog oficial de StackOverflow (2024) sobre tips prácticos de RAG [2], documentación de Microsoft Azure Cognitive Search para RAG [26] [28], y tutoriales de Neo4j sobre GraphRAG con ejemplos de código [34] [35], entre otros. Estas fuentes de alta confiabilidad respaldan las estrategias aquí descritas y ofrecen mayor profundidad para su implementación.

---

[1] [3] [4] [5] [6] [7] [8] [9] [11] [13] [14] [15] [16] [20] [31] [50] Hybrid Retrieval-Augmented Generation Systems for Knowledge-Intensive Tasks | by Adnan Masood, PhD. | Medium
https://medium.com/@adnanmasood/hybrid-retrieval-augmented-generation-systems-for-knowledge-intensive-tasks-10347cbe83ab

[2] Practical tips for retrieval-augmented generation (RAG) - Stack Overflow
https://stackoverflow.blog/2024/08/15/practical-tips-for-retrieval-augmented-generation-rag/

[10] [12] [22] [23] [24] [25] [26] [27] [28] Develop a RAG Solution—Information-Retrieval Phase - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/rag/rag-information-retrieval

[17] [18] [19] [21] Dense vector + Sparse vector + Full text search + Tensor reranker = Best retrieval for RAG? | Infinity
https://infiniflow.org/blog/best-hybrid-search-solution

[29] [30] MultiRetrievalQAChain — LangChain documentation
https://python.langchain.com/api_reference/langchain/chains/langchain.chains.router.multi_retrieval_qa.MultiRetrievalQAChain.html

[32] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [46] [47] [49] Using a Knowledge Graph to implement a RAG application
https://neo4j.com/blog/developer/rag-tutorial/

[33] [48] How knowledge graphs take RAG beyond retrieval
https://www.qed42.com/insights/how-knowledge-graphs-take-rag-beyond-retrieval

[44] Building a Knowledge Graph for RAG using Neo4j - Homayoun S.
https://homayounsrp.medium.com/building-a-knowledge-graph-for-rag-using-neo4j-e69d3441d843

[45] New Blog: LLM Knowledge Graph Builder: From Zero to GraphRAG ...
https://community.neo4j.com/t/new-blog-llm-knowledge-graph-builder-from-zero-to-graphrag-in-five-minutes/68489