

Procesamiento de Lenguaje Natural III

Docentes:




Esp. Abraham Rodriguez - FIUBA

Mg. Oksana Bokhonok - FIUBA






1. RAG avanzado y personalización de soluciones.

Cronograma de la materia

Opción 1

	1:30H	20M	1:30H
Teoría			
Break			
Teoría/Práctica			

Opción 2

	1:00H	10M	50Min	10M	50Min
Teoría					
Break					
Teoría/Práctica					

1. Entrega del proyecto obligatoria (trabajo en grupo):

- Proyecto estructurado en git que contenga:
 - **Código funcional y modular (nivel-preproducción).**
 - **Informe técnico en pdf que contenga:**
 - **Objetivo del proyecto**
 - **Arquitectura general** (diagrama de flujo + descripción de componentes)
 - **Implementación técnica** (herramientas, módulos clave)
 - **Evaluación** (métricas de desempeño de modelos, agentes y RAG)
 - **Resultados y ejemplos**
 - **Conclusiones y mejoras futuras**
 - **Planificación del equipo** (tabla con tareas, responsables y estado)
- Presentación final de 15 minutos, mostrando los resultados más destacados, las visualizaciones de atención y cómo el modelo podría aplicarse en un contexto real.

El código y el informe deben ser entregados a más tardar el **día de la teoría 7**.

Evaluación del proyecto:

- **Informe (25%)**
- **Código (25%)**
- **Evaluación y análisis (25%)**
- **Presentación y visualización (25%)**

Evaluación Global = 0.5*Encuesta del Grupo + 0.5*Proyecto (Salvo los casos de obtener un “2” en uno de los dos)

La entrega tardía tiene una penalización de 2 puntos.

API funcional: Implementar una interfaz mediante FastAPI (preferido por escalabilidad y flexibilidad) o Streamlit (para prototipos visuales), que permita orquestar las interacciones entre usuarios, agentes y modelos de lenguaje.

Al menos dos agentes inteligentes con comunicación dinámica: Diseñar al menos dos agentes autónomos, cada uno con roles diferenciados (por ejemplo, analista y verificador), que interactúen mediante prompts estructurados, mensajes reflexivos o cadenas de razonamiento. Incorporar mecanismos de autoevaluación o retroalimentación cuando sea relevante (ej. agentes autoreflexivos o planificadores).

Recuperación de contexto con RAG (Retrieval-Augmented Generation): Implementar un componente de recuperación semántica, y modelos tipo retriever-ranker. Asegurar la relevancia del contenido recuperado y la trazabilidad de fuentes.

Implementar medidas de seguridad: Definir políticas de control de acceso, filtrado de inputs, validación de outputs y auditoría continua para garantizar robustez y confiabilidad del sistema, según corresponda.

Integración de modelo preexistente para inferencia: Utilizar un modelo CNN/VIT ya entrenado (propio o alojado en plataformas como Hugging Face) para realizar tareas de inferencia.

Flujo de datos completo y modular: Asegurar un pipeline de procesamiento claro, donde la entrada fluya a través de los distintos agentes, RAG y modelo, hasta generar una respuesta final. Incorporar componentes desacoplables y monitoreables.

Optimización de costos y latencia: Incluir al menos una acción orientada a reducir el costo computacional o de inferencia (por ejemplo, selección dinámica de modelo, reducción de longitud de contexto, uso de caching o batching, o control de temperatura/token limit).

Acción final disparada automáticamente: El sistema debe ejecutar una acción de salida automatizada, como enviar un email (via Gmail API o SMTP), registrar eventos en un log externo, o integrar con servicios de terceros (Webhook, Slack, Notion, etc.).

Evaluación de desempeño y métricas:

- Performance del modelo: tiempo de respuesta, consumo de tokens, precisión semántica (BLEU/ROUGE/semantic similarity).
- Performance de los agentes: coherencia de decisiones, calidad del diálogo entre agentes, redundancia evitada, latencia.
- Eficiencia del RAG: recall, precisión@k, tiempo de recuperación, cobertura de contexto útil.

Un sistema que ayuda a verificar si una planta industrial cumple con las normas de señalización de seguridad (carteles, pictogramas, advertencias), a partir de imágenes tomadas en campo, y sugiere correcciones usando documentos regulatorios.

Inferencia CNN/ViT:

- Se usa un modelo ViT o CNN para:
 - Detectar símbolos y carteles de seguridad (ej. casco obligatorio, prohibido fumar).
 - Clasificar si están presentes, correctos y legibles.
- Opcional: se combina con OCR para leer el texto de los carteles (ej. usando DONUT).

2. Agentes:

- Agente 1: Inspector Visual: analiza la imagen, detecta la presencia y tipo de cartel.
- Agente 2: Normador: consulta la normativa aplicable (por sector o país) vía RAG, y verifica si lo que ve el Agente 1 cumple.

3. RAG:

- Corpus vectorial con:
 - Normas IRAM / OSHA / ISO (segmentadas).
 - Requisitos específicos por tipo de instalación (eléctrica, química, obra civil).

...etc

Código-Estructura



[Cookiecutter Data Science](#)

[Folder Structure for Machine Learning Projects](#)



```

├── LICENSE
├── Makefile          <- Makefile with commands like `make data` or `make train`
├── README.md         <- The top-level README for developers using this project.
├── data
│   ├── external      <- Data from third party sources.
│   ├── interim       <- Intermediate data that has been transformed.
│   ├── processed     <- The final, canonical data sets for modeling.
│   └── raw           <- The original, immutable data dump.
├── docs              <- A default Sphinx project; see sphinx-doc.org for details
├── models            <- Trained and serialized models, model predictions, or model summaries
├── notebooks         <- Jupyter notebooks. Naming convention is a number (for ordering),
│                       the creator's initials, and a short '-' delimited description, e.g.
│                       `1.0-jqp-initial-data-exploration`.
├── references        <- Data dictionaries, manuals, and all other explanatory materials.
├── reports
│   └── figures       <- Generated graphics and figures to be used in reporting
├── requirements.txt  <- The requirements file for reproducing the analysis environment, e.g.
│                       generated with `pip freeze > requirements.txt`
├── setup.py          <- makes project pip installable (pip install -e .) so src can be imported
├── src               <- Source code for use in this project.
│   ├── __init__.py   <- Makes src a Python module
│   ├── data          <- Scripts to download or generate data
│   │   └── ....py
│   ├── features      <- Scripts to turn raw data into features (i.e chunks for VectorDBs)
│   │   └── ....py
│   ├── prompts       <- Where prompts for the LLM application reside
│   │   └── prompts.py
│   ├── models        <- Scripts for LLM models / applications
│   └── visualization <- Scripts to create exploratory and results oriented visualizations (Streamlit,
│       └── visualize.py
└── tox.ini           <- tox file with settings for running tox; see tox.readthedocs.io
    
```

Elementos específicos para LLMs y agentes al usar Cookiecutter en `cookiecutter.json`

Selección de proveedor/modelo	"llm_provider": ["openai", "anthropic", "local"], "use_embeddings": ["yes", "no"]	Esto te permite condicionar archivos como: models/openai_model.py, models/local_llm_loader.py, embeddings/huggingface_embedder.py. Y habilitar lógicas específicas solo si se requiere.
Integración de RAG	"use_rag": ["yes", "no"], "vector_store": ["faiss", "chroma", "none"]	Esto define si se genera chains/rag_chain.py, data/vectorstore/faiss_index/, tools/document_loader.py
Gestión de memoria Algunos agentes requieren memoria persistente (ej. ConversationBufferMemory, Redis, FAISS Memory)	"use_memory": ["yes", "no"], "memory_backend": ["langchain", "redis", "none"]	genera módulos como memory/langchain_memory.py o memory/redis_memory.py
Workflows tipo LangGraph	"agent_or_graph": ["basic_agent", "langgraph"]	Para decidir si el proyecto usa: agents/basic_agent.py o chains/graph/agent_graph.py y configuración de nodos
Backends / interfaz	"interface": ["fastapi", "streamlit", "none"]	Y así generar app/main.py correspondiente.
use_mcp	"use_mcp": ["yes", "no"]	Si "yes", genera carpetas y archivos config.json para cada módulo (agents/, models/, tools/)

Ejemplo de uso en
`cookiecutter.json`

[Cookiecutter Data Science](#)

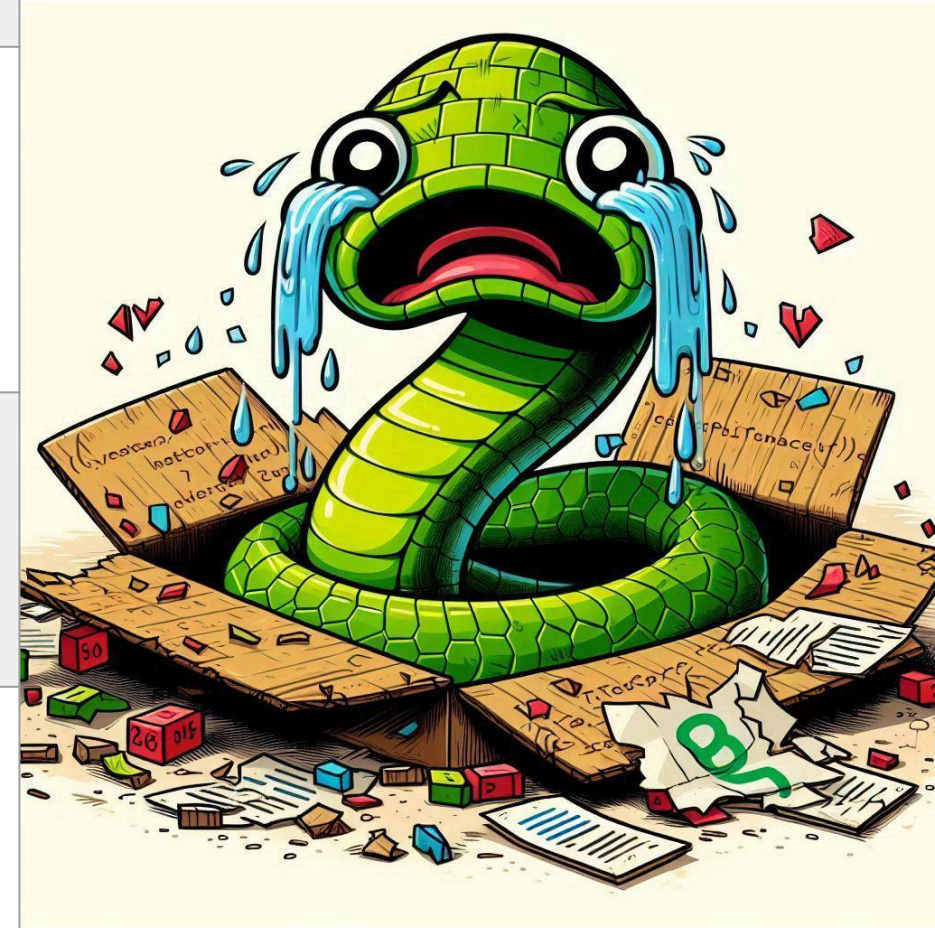
Código-Etapas

EDA, Prototipo, Preproducción, Producción

Aspecto	EDA	Productivo
Estructura	Notebook sin organización formal.	Scripts modulares con carpetas organizadas.
Código Reutilizable	Código acoplado, poco modular.	Funciones y clases reutilizables.
Configuración	Parámetros hardcoded en el código.	Configuración externa con archivos .yaml o .json.
Logs y Errores	Uso de print() para debugging.	Sistema de logging robusto con niveles (INFO, ERROR, DEBUG).
Pruebas	Sin pruebas o validaciones.	Pruebas unitarias y de integración.
Escalabilidad	Procesamiento limitado (archivos pequeños, sin paralelismo).	Optimización para grandes volúmenes (e.g., paralelismo, uso de GPU/CPU).
Documentación	Comentarios básicos o inexistentes.	Docstrings detallados y README explicativo.
Automatización	Manual (ejecución interactiva).	Pipelines automáticos (e.g., Prefect, Airflow).

Código-EDA (Exploratory Data Analysis)

Objetivo:	Entender los datos, explorar patrones, identificar problemas y validar hipótesis iniciales.
Estructura del Código:	<ul style="list-style-type: none">• Notebook poco estructurado, con celdas ejecutadas en orden arbitrario.• Código redundante o fragmentado (copiar/pegar es común).• Depuración y visualización inmediatas (print(), matplotlib, seaborn).
Enfoque:	<ul style="list-style-type: none">• Experimentación rápida.• Uso intensivo de visualizaciones.• Pruebas de hipótesis rápidas sin preocuparse por optimización o escalabilidad.
Problemas Comunes:	<ul style="list-style-type: none">• Falta de reproducibilidad. El orden de ejecución puede afectar los resultados.• Código no modular, difícil de reutilizar.• Operaciones no optimizadas.• Falta de manejo de errores y logs.



Código-Preproducción

Objetivo:	Convertir el prototipo en un flujo reproducible y parcialmente automatizado.
Estructura del Código:	<ul style="list-style-type: none">• Código dividido en scripts o módulos (e.g., <code>data_preprocessing.py</code>, <code>train_model.py</code>).• Incorporación de configuraciones externas (<code>config.yaml</code> o <code>.json</code>).• Uso de herramientas de pruebas como Pytest para validar partes críticas del flujo.
Enfoque:	<ul style="list-style-type: none">• Modularidad: Separar claramente las etapas.• Implementación inicial de logs• Control de versiones del código (e.g., Git).• Manejo de errores básicos (try/except en funciones críticas).
Problemas Comunes:	<ul style="list-style-type: none">• Falta de pruebas exhaustivas: Limitada cobertura de pruebas unitarias.• Manejo limitado de datos: No considera grandes volúmenes o datos en tiempo real.

Código-Manejo de Errores y Warnings

Uso de bloques try/except para capturar errores críticos y proporcionar mensajes útiles. ([8. Errors and Exceptions — Python 3.11.10 documentation](#))

Warnings: Filtrar o personalizar los warnings para evitar ruido innecesario. ([warnings — Warning control — Python 3.11.10 documentation](#))

Logs: Implementar un sistema de logging con niveles como INFO, WARNING, ERROR y DEBUG. Los logs se guardan en archivos para referencia posterior. ([logging — Logging facility for Python — Python 3.11.10 documentation](#), [Logging HOWTO — Python 3.11.10 documentation](#))

Código-Registro

[MLflow Overview](#)

[LLMs](#)

[Fine-Tuning Transformers with MLflow for Enhanced Model Management](#)

Tecnologías y herramientas



LangGraph



Hugging Face



LangChain

Repaso

Fundamentos Operativos de los LLMs

- Tokenizers y Representación de Texto
 - WordPiece, BPE, SentencePiece.
 - Control de longitud y formato de entrada.
- In-Context Learning y Few-shot Prompting
 - Zero-shot, few-shot, chain-of-thought.
 - Adaptabilidad sin reentrenamiento.

Repaso

Estrategias de Razonamiento y Adaptación

- Razonamiento Estructurado
 - Chain-of-Thought, ReAct, Tree-of-Thoughts.
 - Resolución de problemas complejos paso a paso.
- Prompting vs RAG vs Fine-tuning
 - Comparación de técnicas.
 - Casos de uso y trade-offs en costo vs capacidad.

Repaso

Infraestructura y Recuperación de Información

- Infraestructura Local y APIs
 - Ollama, Llama.cpp, safetensors, quantization.
 - Ejecución local de modelos y eficiencia.
- RAG y Bases Vectoriales
 - Segmentación, embeddings, FAISS.
 - Búsqueda semántica para complementar generación.

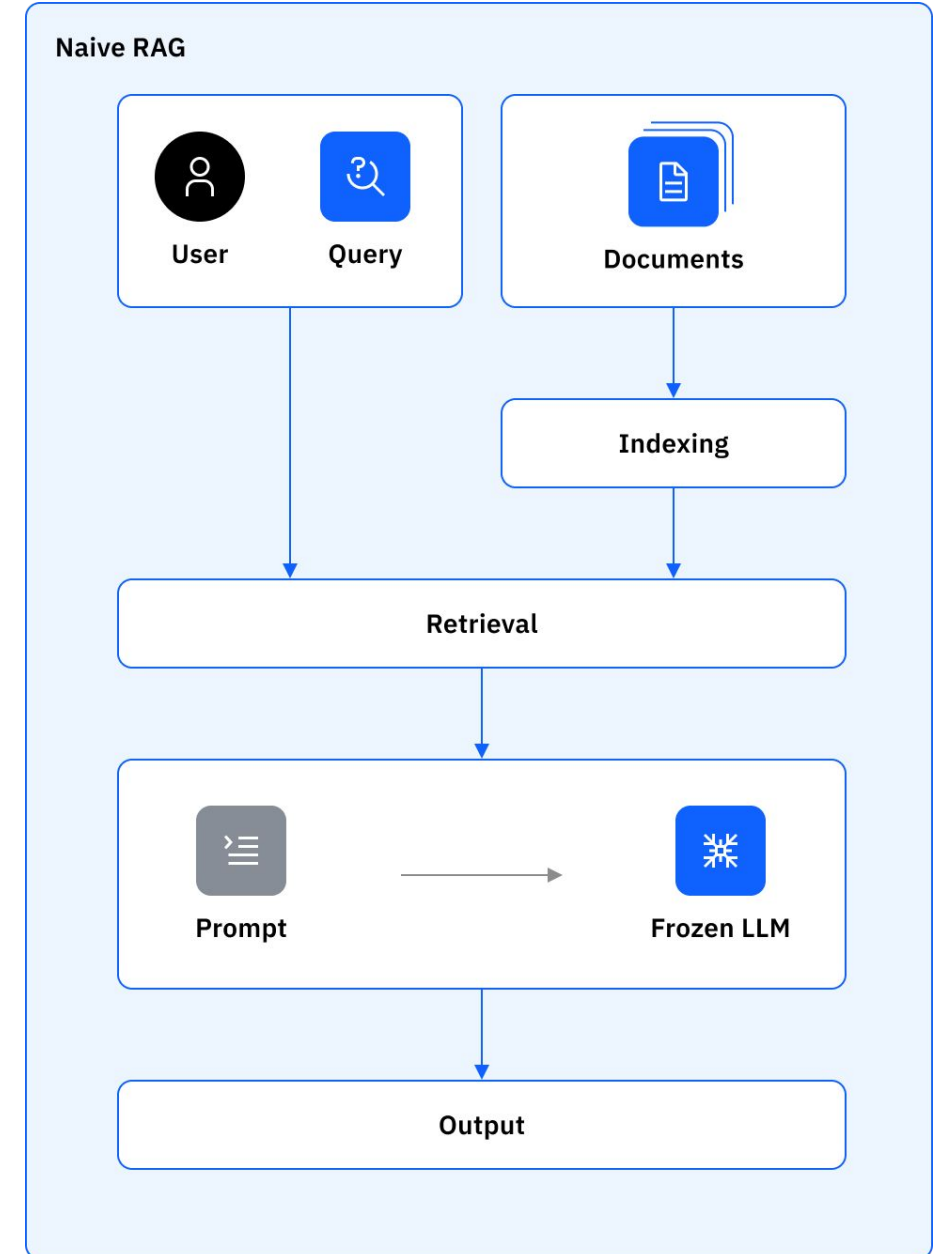
Repaso

Evaluación y Agentes

- Evaluación, Bias y Fairness
 - Riesgos: toxicidad, sesgo, adversarial.
 - Métricas y datasets de evaluación.
- Agentes y Arquitecturas Deliberativas
 - Composición de reasoning + acting.
 - Ejemplos: AutoGPT, ReAct, ReWOO.
 - Base para LLM2: segmentación, memoria, herramientas.

Naive RAG

- Aplicación de diversas técnicas de optimización de Naive RAG
- La recuperación inicial puede traer documentos irrelevantes aunque parezcan similares.



HNSW (Hierarchical Navigable Small World)

HNSW es el más usado en RAG hoy.

1. Arranque con una semilla (entry point)

El índice tiene un nodo de inicio (semilla). Calcula la distancia entre el embedding de la query y ese nodo.

2. Comparación inicial

Si el nodo está muy lejos, el algoritmo salta a otra región del grafo (usa niveles jerárquicos o “atalayas”). Esto evita quedarse “atrapado” en un lugar del espacio vectorial que no tiene nada que ver con la query.

3. Exploración local

Cuando encuentra un nodo suficientemente parecido, comienza a explorar sus vecinos más cercanos en el grafo. La lógica es: “si este punto es parecido, probablemente sus vecinos también lo sean”.

4. Expansión y refinamiento

Sigue avanzando de vecino en vecino, siempre moviéndose hacia los que tienen menor distancia a la query. Si encuentra un camino mejor, lo sigue.

5. Top-k resultados

Cuando ya no puede mejorar, devuelve los k embeddings más cercanos (los vecinos finales).

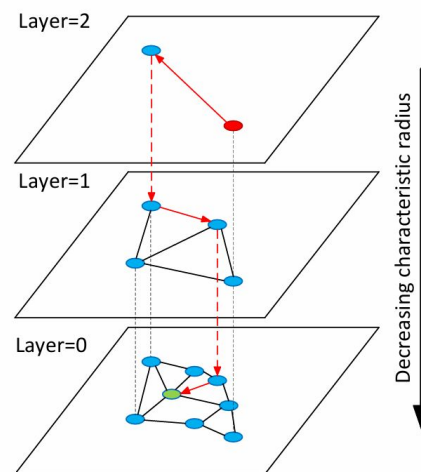


Fig. 1. Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green).

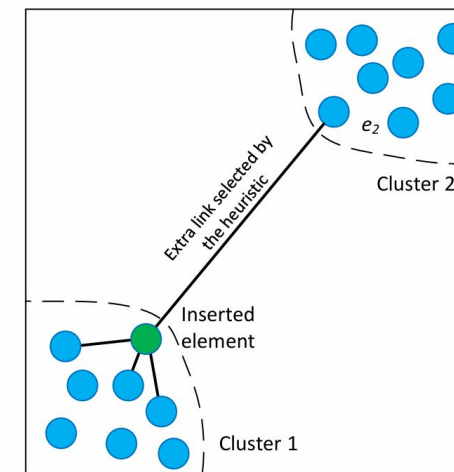


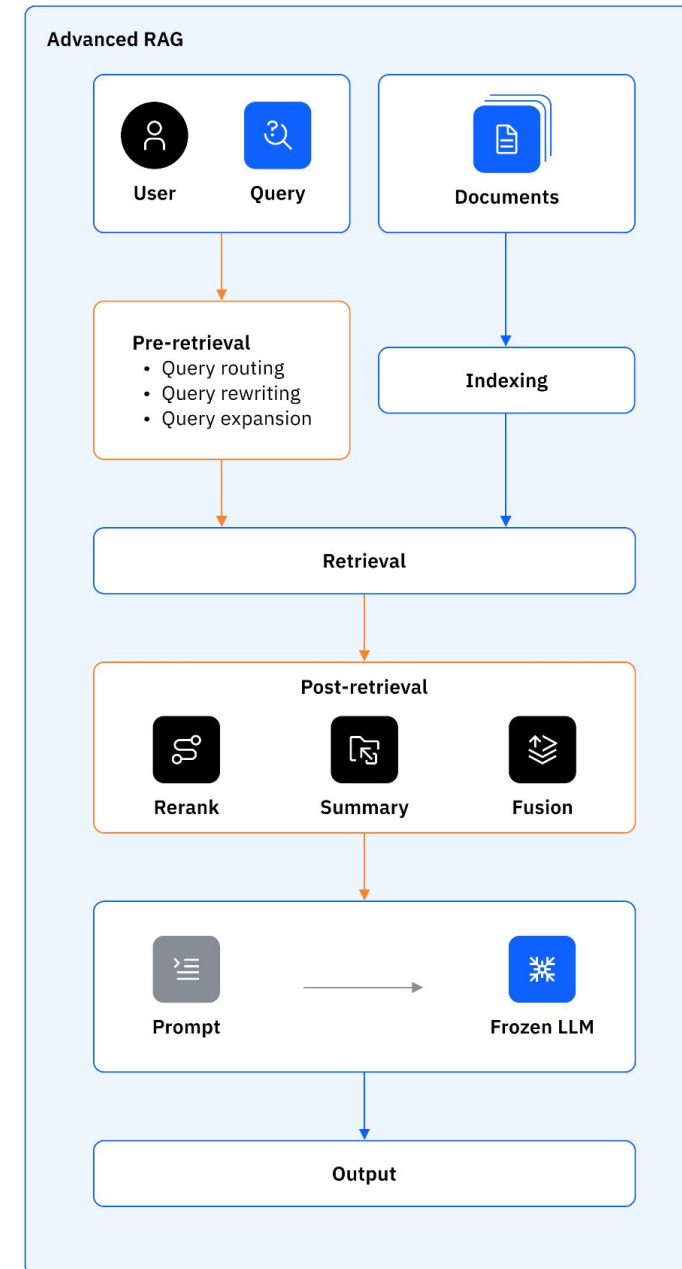
Fig. 2. Illustration of the heuristic used to select the graph neighbors for two isolated clusters. A new element is inserted on the boundary of Cluster 1. All of the closest neighbors of the element belong to the Cluster 1, thus missing the edges of Delaunay graph between the clusters. The heuristic, however, selects element e_2 from Cluster 2, thus, maintaining the global connectivity in case the inserted element is the closest to e_2 compared to any other element from Cluster 1.

Naive RAG

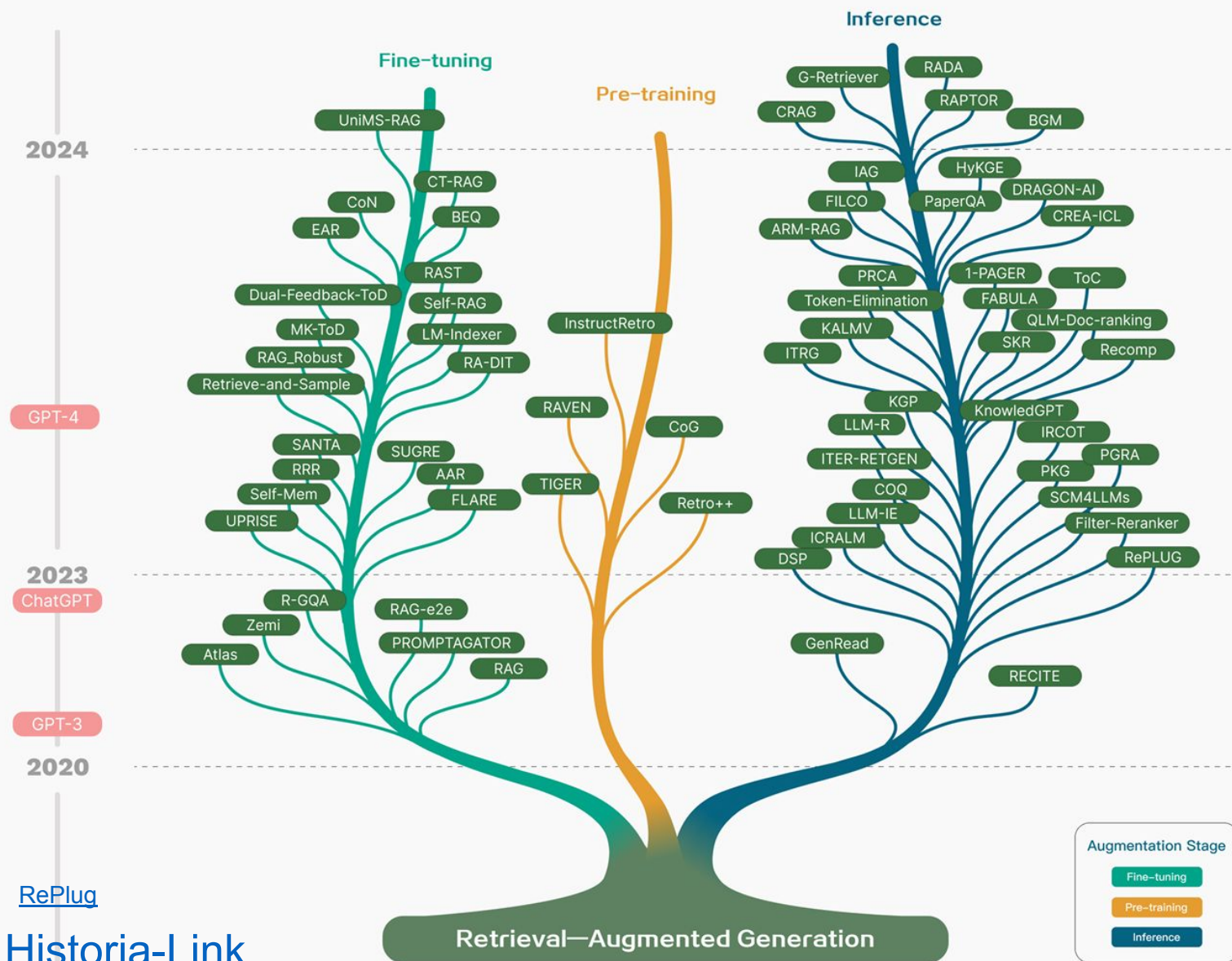
- Aplicación de diversas técnicas de optimización de Naive RAG
- La recuperación inicial puede traer documentos irrelevantes aunque parezcan similares.



Técnicas de RAG avanzado



Investigación de RAG



Pre-training: Trabajos que incorporan retrieval ya durante el preentrenamiento. El modelo se “pre-entrena” con acceso a memoria/índices, creando una base de conocimiento consultable desde el inicio.

- integración profunda y eficiente.
- costos muy altos; pocos proyectos porque exige entrenar modelos grandes.

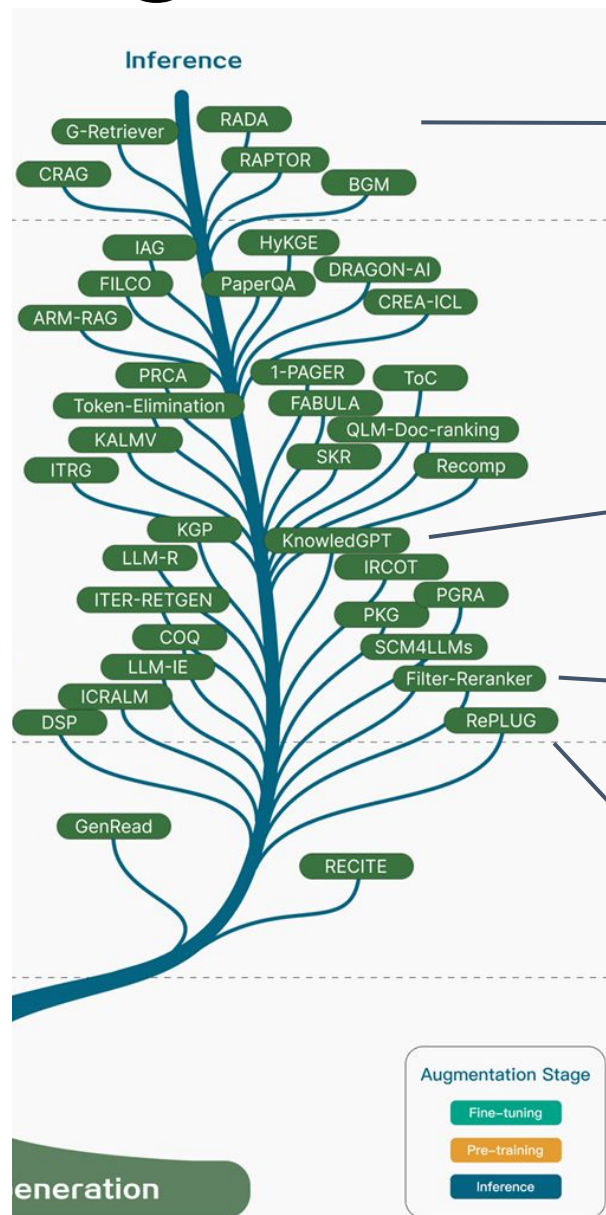
Fine-tuning: enseñar al LLM a pedir, seleccionar y razonar con documentos durante el ajuste.

- mejor uso de evidencia; puede aprender comportamientos complejos.
- requiere datos/entrenamiento y mantenimiento por dominio.

Inference: pipeline clásico de RAG: recuperar → re-ranear/filtrar → inyectar como contexto al modelo.

- fácil de desplegar y adaptar; barato.
- puede depender mucho del recuperador y ser sensible a la calidad del contexto.

Investigación de RAG

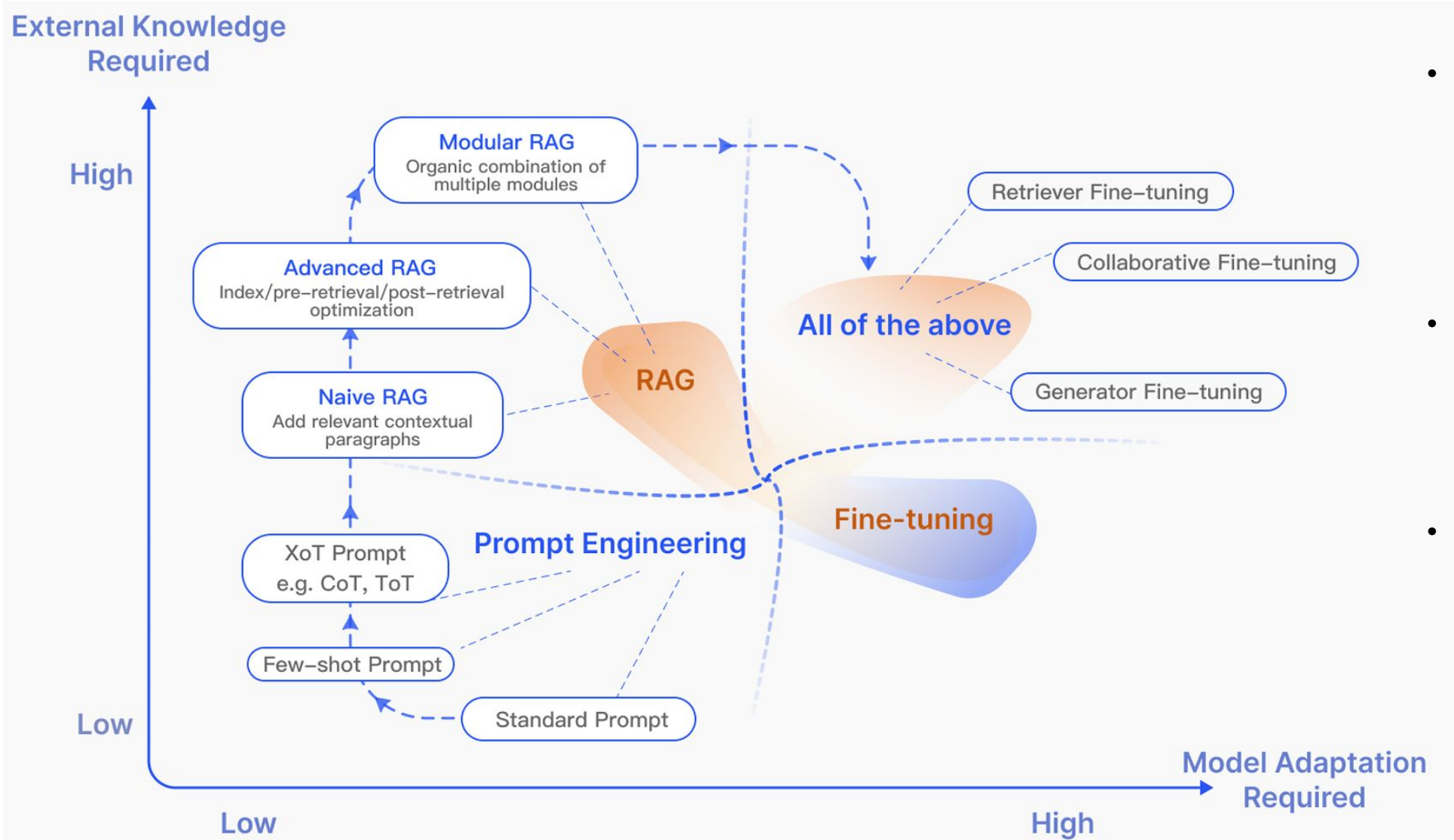


Token-Elimination / RADA / FABULA / ToC Técnicas más experimentales que eliminan información redundante o dividen el input en bloques jerárquicos.

KnowledGPT Extrae conocimiento relevante en tiempo real y lo integra directamente en prompts para evitar alucinaciones.

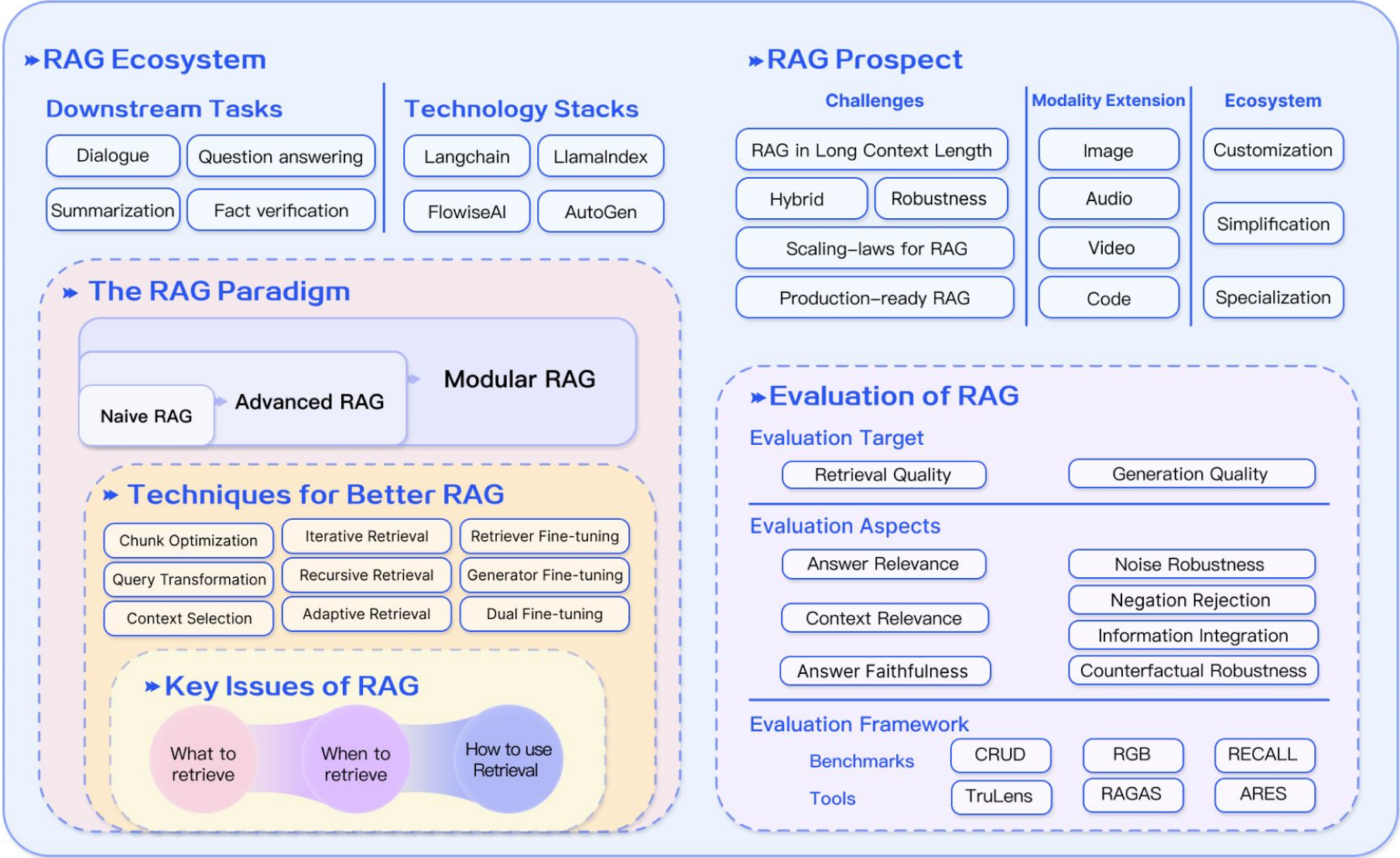
Filter-Reranker Usa un filtro + reranker para seleccionar sólo los documentos más útiles antes de pasar al generador.

RePLUG Reemplaza el retriever tradicional por uno entrenado para colaborar mejor con el generador.



- **Retriever fine-tuning:**
Ajusta el buscador para traer documentos más relevantes → mayor recall/precisión.
- **Generator fine-tuning:**
Ajusta el LLM para usar el contexto y citar → menos alucinación/formato estable.
- **Collaborative fine-tuning:**
Entrena buscador+LLM juntos (o en bucle) → calidad extremo a extremo mejorada.

Ecosistema del RAG



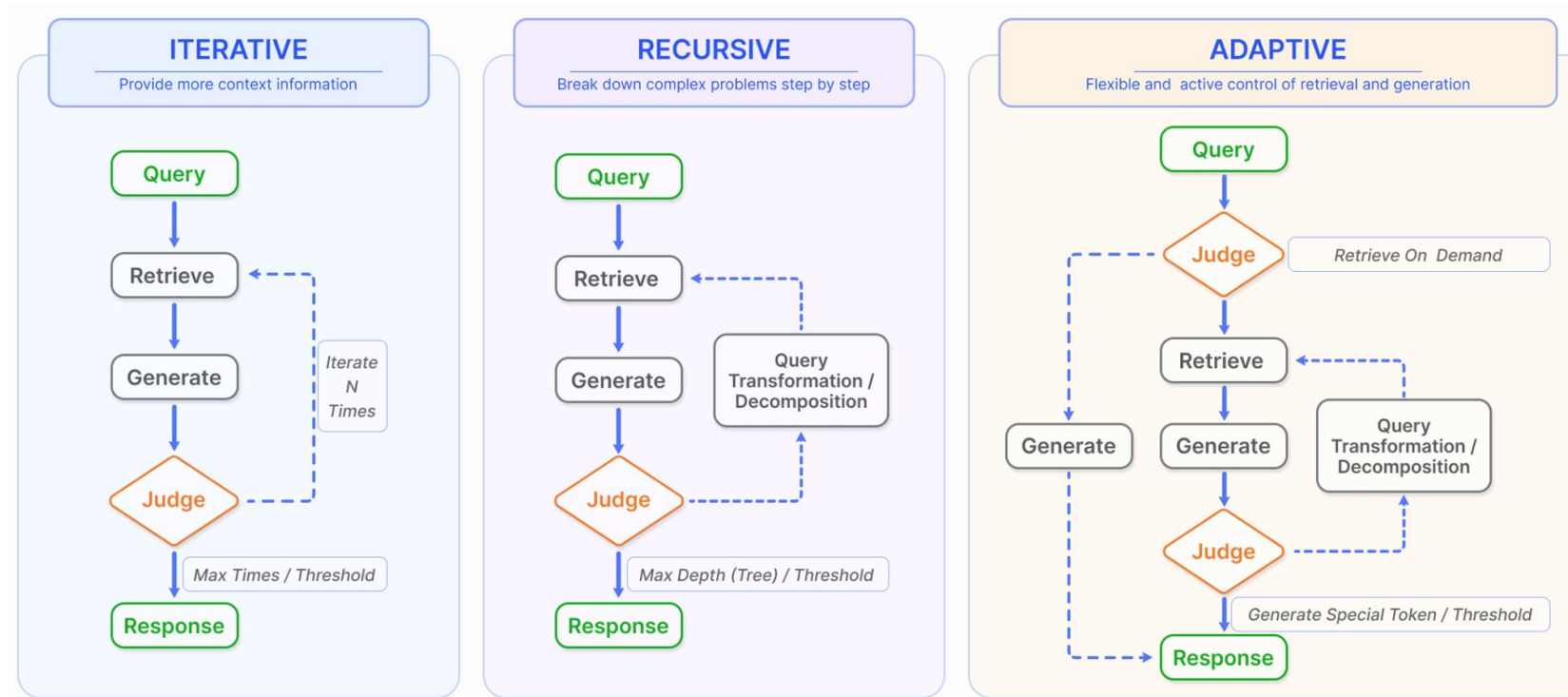
Tipos de Retrievers

Sparse	Basados en términos exactos (técnicas: TF-IDF, BM25,...); útiles para textos con palabras clave claras. Indexa y recupera texto de forma lexic/simbolica.
Dense	Usan embeddings semánticos (modelos: BERT, SBERT); ideales para queries vagas o semánticas.
Híbrido	Combinan ambos mundos: precisión semántica y cobertura por palabras clave (técnicas: re-ranking o score fusion).

RAG - Estrategias de Recuperación

Además del proceso de recuperación más común (una sola vez), existen tres tipos de procesos de aumento de recuperación:

- **Recuperación iterativa:** alterna entre recuperación y generación
- **Recuperación recursiva:** refina gradualmente la consulta del usuario dividiendo el problema en subproblemas, y luego los resuelve de forma continua mediante recuperación y generación.
- **Recuperación adaptativa:** permite que el sistema RAG decida de forma autónoma si necesita recuperar conocimiento externo, y cuándo debe detener la recuperación y la generación, utilizando a menudo tokens especiales generados por el LLM para controlar este proceso.



RAG - Estrategias de Recuperación

La transformación de consultas, también conocida como query rewriting, es un enfoque que convierte una consulta en otra reformulación

Query rewriting

- Rewrite-Retrieve-Read, RL con feedback de lector
- Feedback de ranking sin etiquetas manuales
- Múltiples señales de refuerzo para rewriter

Query decomposition

- Descomposición optimizada para sistemas multi-vector retrieval
- Descomposición multi-hop en pipelines RAG

HyDE embeddings hipotéticos

- Embeddings hipotéticos mejoran recuperación y QA

Multi-query diversificado

- Estrategia adaptativa para múltiples rewrites

Rewriting basado en reglas

- Reescritura iterativa con feedback de contraejemplos

Variantes de RAG avanzado

RAG Clásico ([Link](#))

Respuestas rápidas desde un corpus conocido.

RAG METEORA ([Link](#))

Selecciona los fragmentos más relevantes usando razonamientos generados por el LLM, sin depender de top-k clásico

RAG Fusion ([Link](#))

Combina múltiples retrievers y variantes de queries para una recuperación más precisa y completa.

HyDE ([Link](#))

Mejorar la recuperación en casos donde la consulta es vaga o los documentos no son explícitos.

Self-RAG([Link](#))

Permite al modelo autoevaluar sus respuestas y relanzar consultas si detecta baja calidad.

Corrective RAG (CRAG: [Link](#))

Refina respuestas mediante evaluación de contexto; reconsulta si se detectan errores o ambigüedades.

Multi-hop RAG ([Link](#))

Resolver consultas que requieren conectar múltiples hechos o documentos secuenciales.

Graph RAG ([Link](#))

Extraer y navegar relaciones semánticas estructuradas en múltiples dominios usando grafos.

Modular RAG ([Link](#))

Arquitectura donde la recuperación y la generación funcionan en un pipeline abierto y modular, permitiendo personalización, escalabilidad y adaptación a distintos casos de uso.

Speculative RAG ([Link](#))

Acelera y mejora precisión generando borradores con un modelo ligero, validados por un LLM mayor.

Agentic RAG([Agentic Retrieval-Augmented Generation for Time Series Analysis](#))

Delegar en agentes autónomos el control de cuándo y cómo recuperar y generar, adaptándose dinámicamente.

RadioRAG([Link](#))

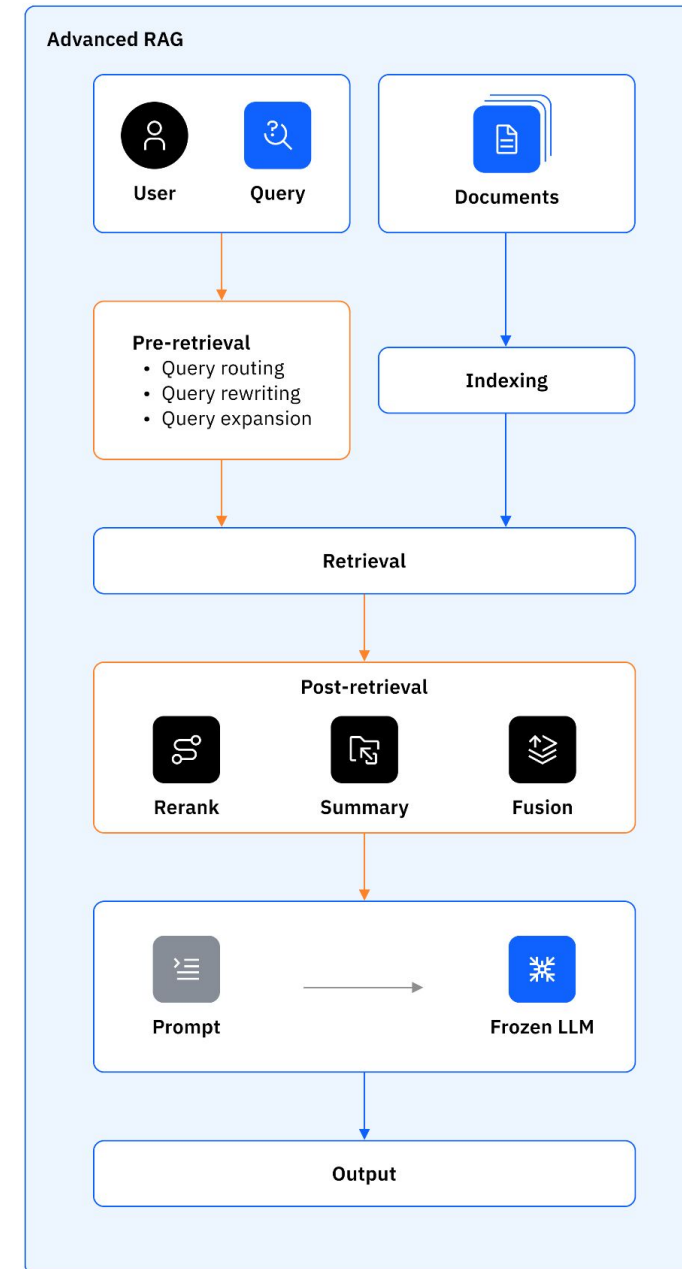
Integrar conocimiento médico actualizado (especialmente radiología) para mejorar precisión clínica.

RAG Re-Ranking

El re-ranking es una etapa post-retrieval usada para ordenar y filtrar documentos por relevancia semántica, elevando precisión y reduciendo ruido contextual.

Beneficios:

- Mejora precisión del contenido generado
- Reduce carga para el LLM al entregar contexto más relevante
- Aumenta coherencia y fiabilidad, especialmente en dominios sensibles

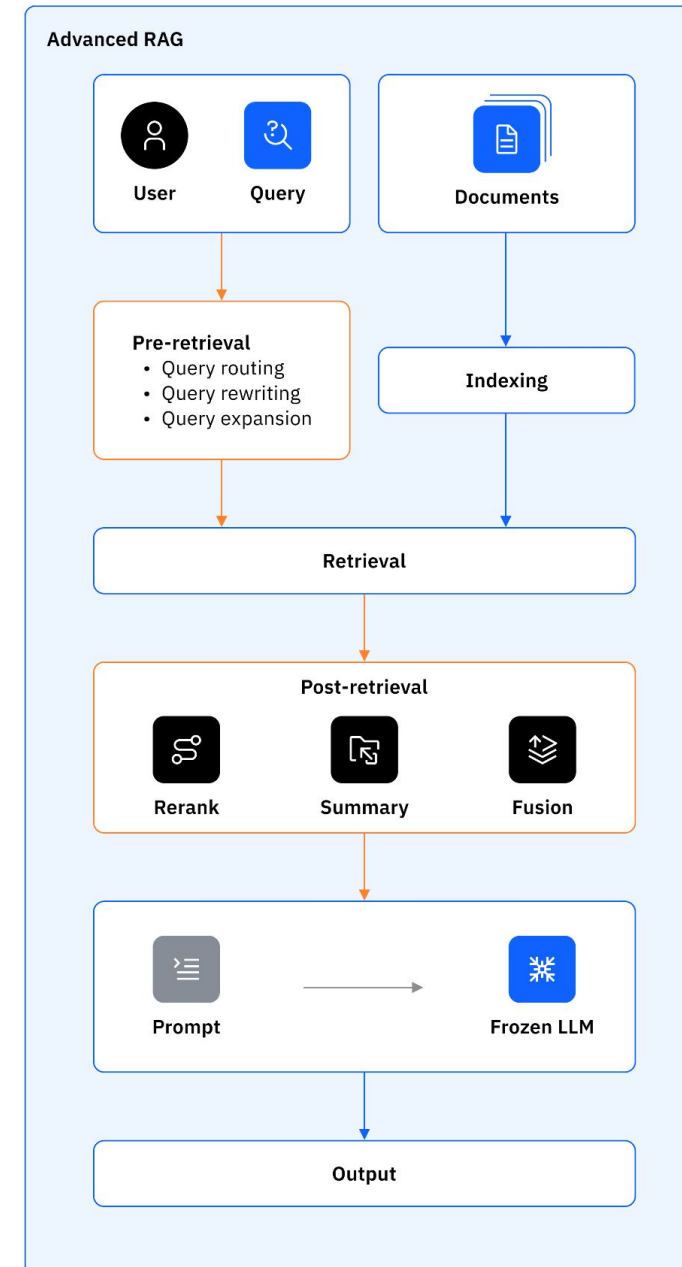


RAG Summary

El resumen condensa múltiples documentos o segmentos de contexto recuperados en un texto más breve y digerible antes de entregarlo al modelo generador. Puede realizarse con modelos extractivos o abstractive summarizers (como Pegasus, T5, etc.).

Beneficios:

- Reduce consumo de tokens y optimiza el uso del contexto útil.
- Mejora la legibilidad y reduce la redundancia en la entrada del LLM.
- Eleva la coherencia general al sintetizar múltiples fuentes en una narrativa unificada.

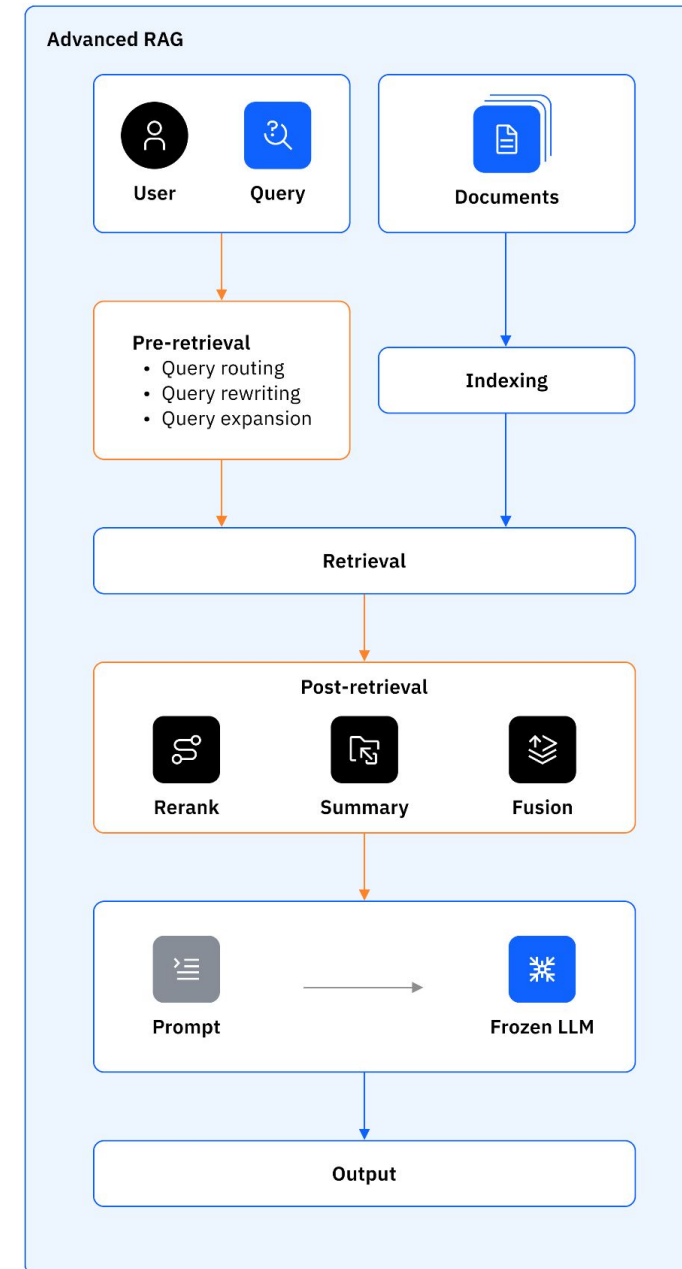


RAG fusión

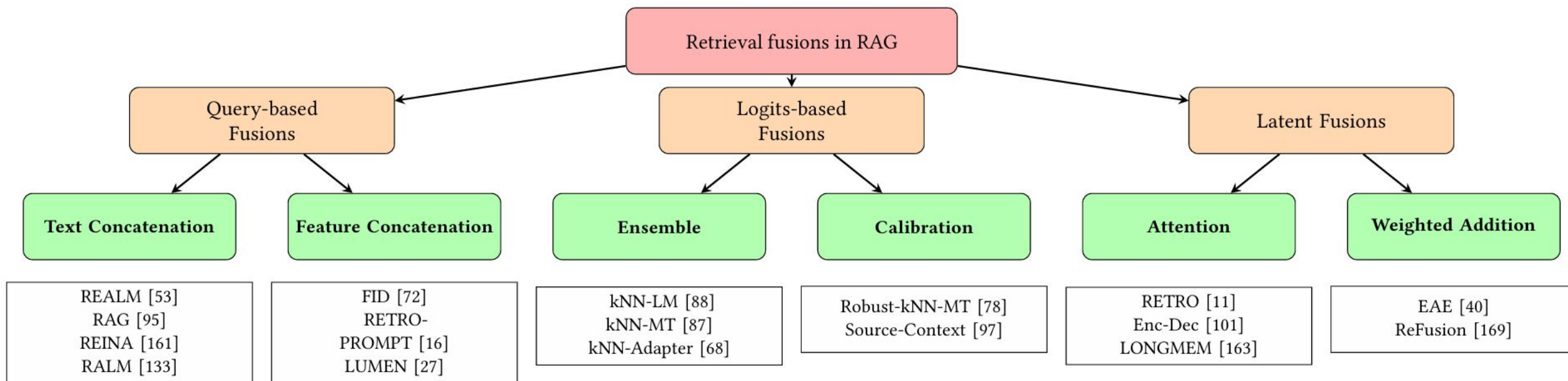
La fusión es una técnica post-retrieval que combina múltiples fuentes de contexto recuperado (ej. distintos queries, métodos o retrievers) en una única vista del conocimiento. Esto puede incluir operaciones como late fusion o reciprocal rank fusion (RRF).

Beneficios:

- Aumenta cobertura semántica combinando perspectivas parciales.
- Reduce errores por sesgo de un solo retriever o query.
- Mejora robustez del sistema en escenarios de ambigüedad o bajo recall inicial.

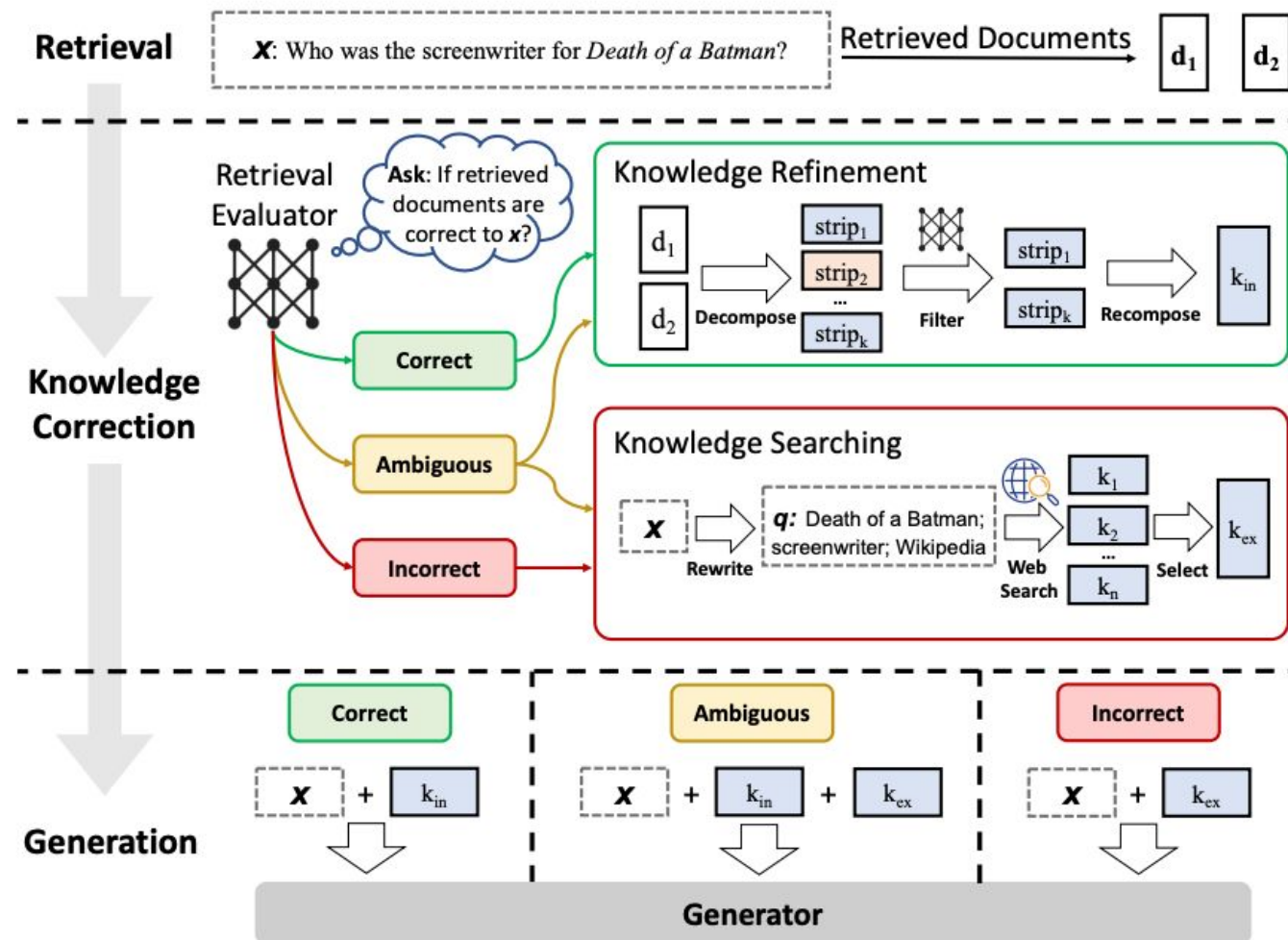


RAG fusión

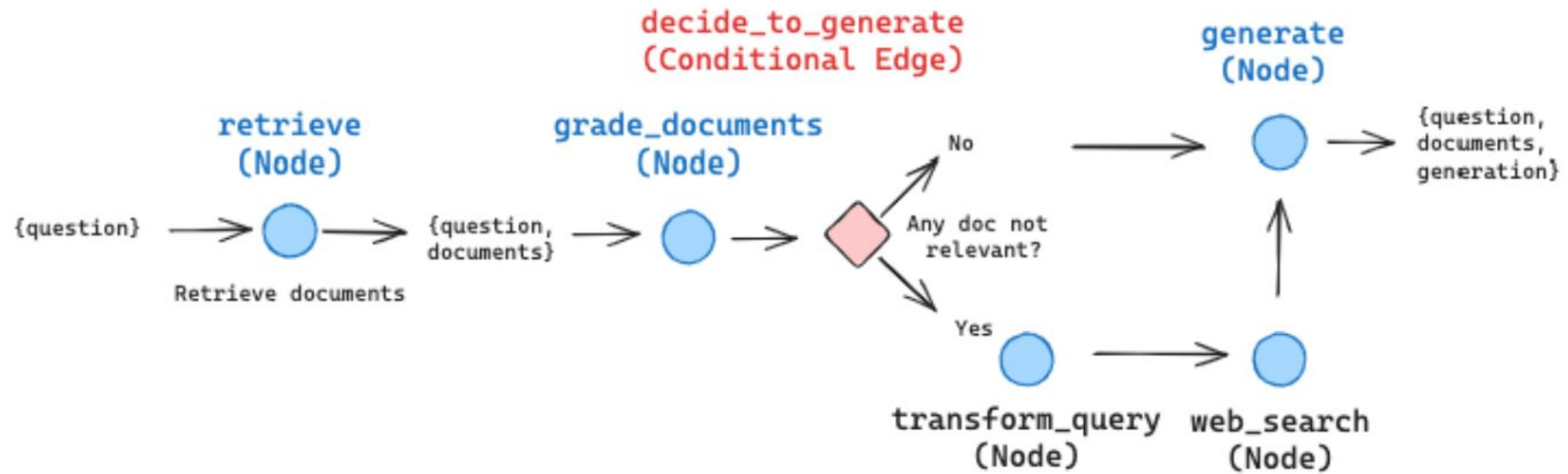


Corrective RAG (CRAG)

- Aumenta la robustez evaluando la recuperación y refinando documentos con descomposición y recomposición (evaluadores de calidad)
- Usa un evaluador de recuperación que clasifica los documentos como correctos, incorrectos o ambiguos según confianza.
- Mejora significativamente la generación en tareas de texto corto y largo.



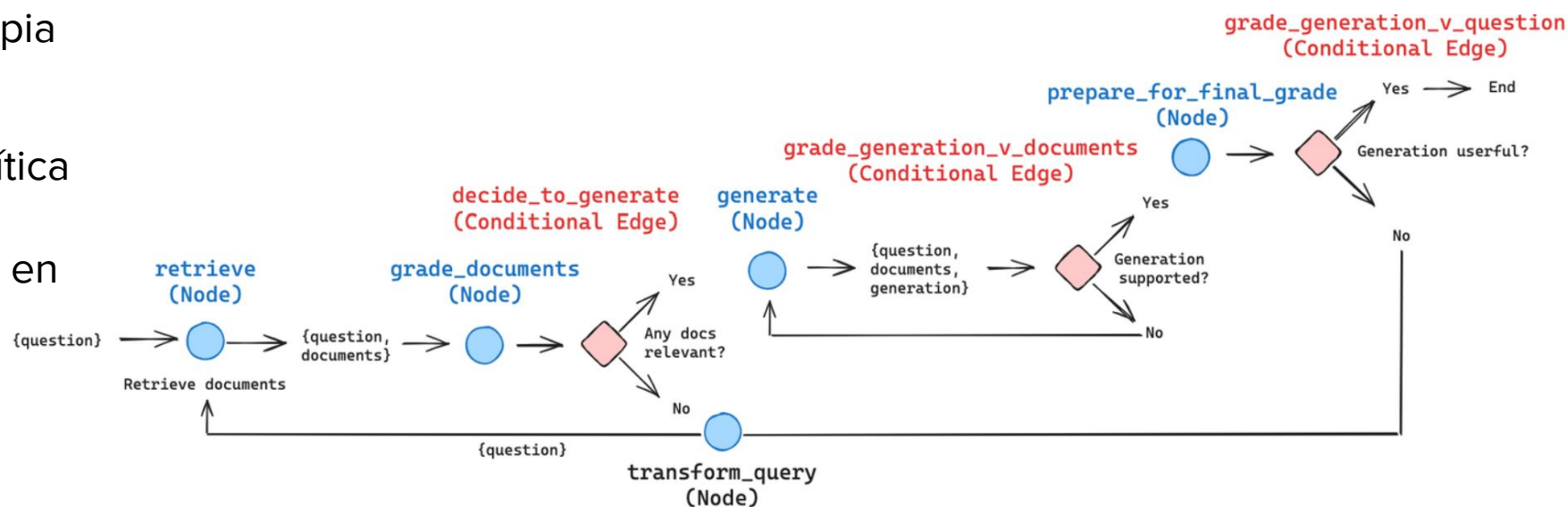
Corrective RAG (CRAG)



LangGraph implementation for CRAG

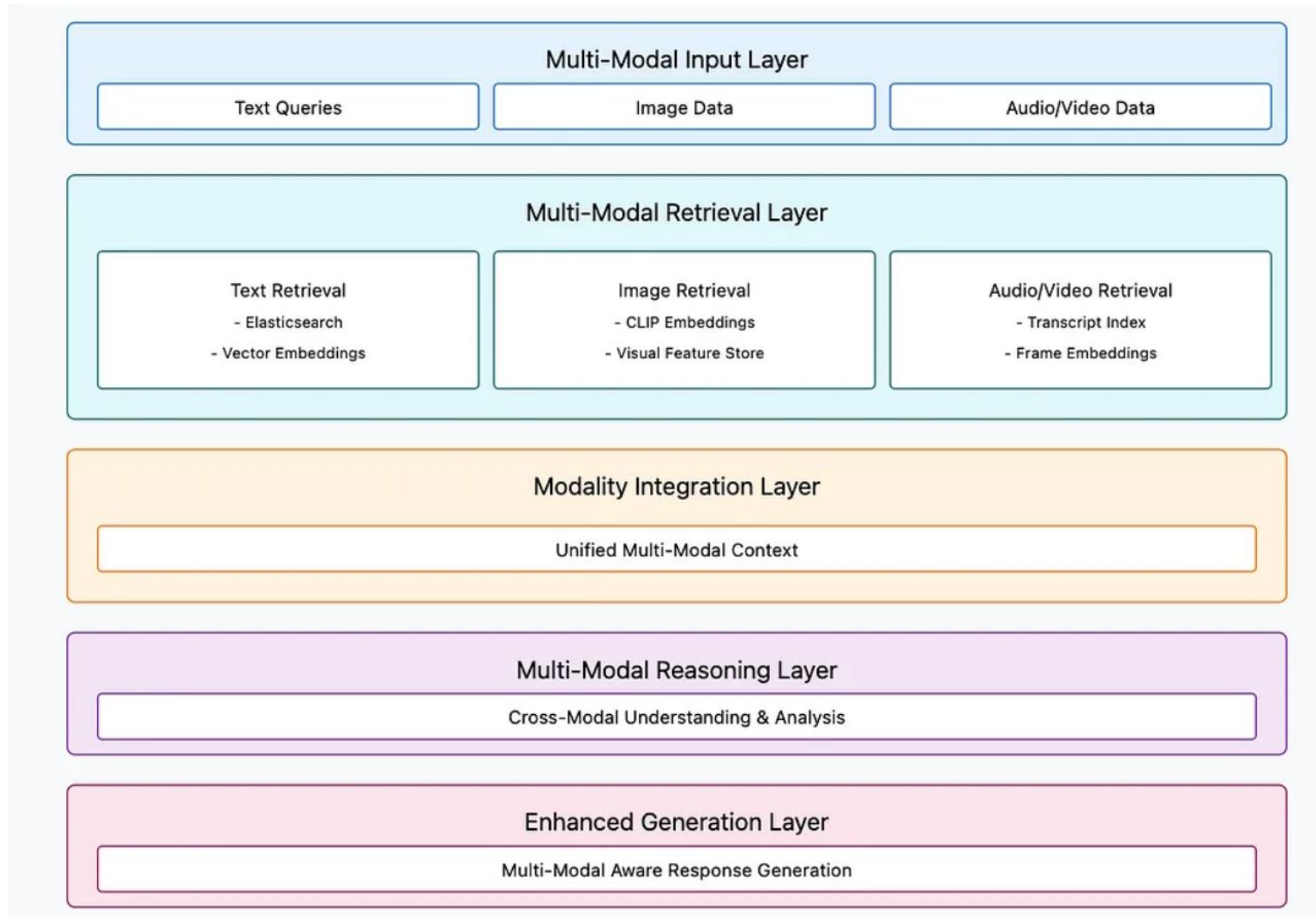
Self-RAG

- Mejora la calidad y veracidad del modelo mediante recuperación y reflexión propia (feedback loops)
- Usa tokens de reflexión y crítica entrenados para guiar la recuperación bajo demanda en distintas etapas de RAG.
- Aumenta la veracidad y adaptabilidad del modelo a distintas tareas.



LangGraph implementation for Self-RAG

Multimodal RAG Architecture Example



Multimodal RAG - CMR

Composed Multimodal Retrieval (CMR) es una extensión del paradigma RAG donde la recuperación no solo se hace con texto, sino también con inputs multimodales (query multimodal). Se combinan buscadores vectoriales multimodales (como CLIP) con modelos generativos que razonan sobre evidencia visual y textual combinada. Muy útil para QA con soporte visual, medicina, e-commerce o inspección industrial. Expresa mejor la intención del usuario cuando el texto no es suficiente.

Componentes:

- Indexación multimodal: se embeben texto e imágenes en un mismo espacio semántico.
- Query multimodal: el input puede ser texto, imagen o ambos.



Question: Is this the tallest building in the city?

External Knowledge

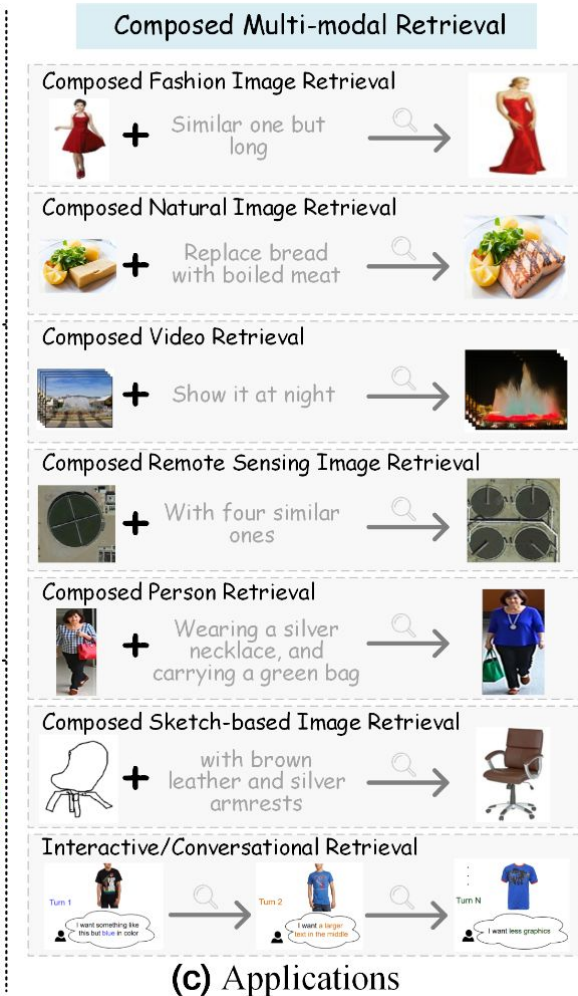
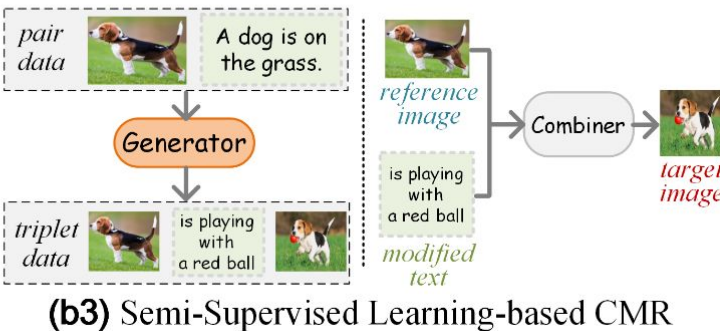
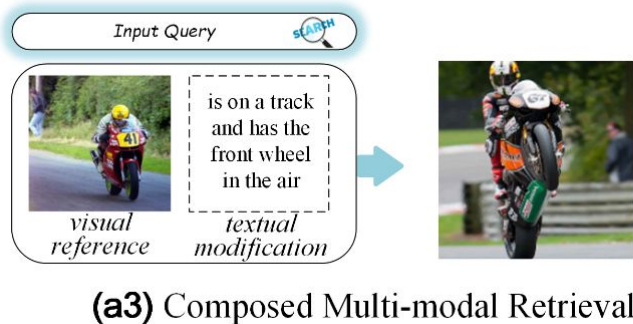
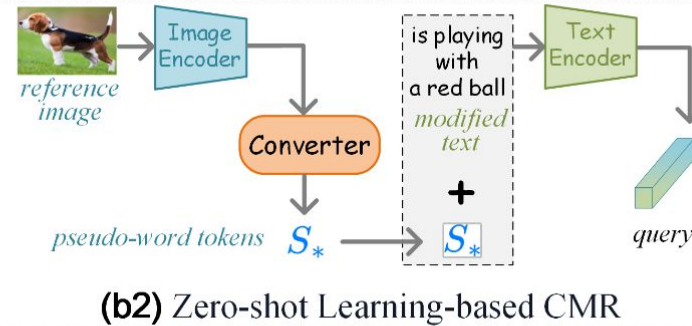
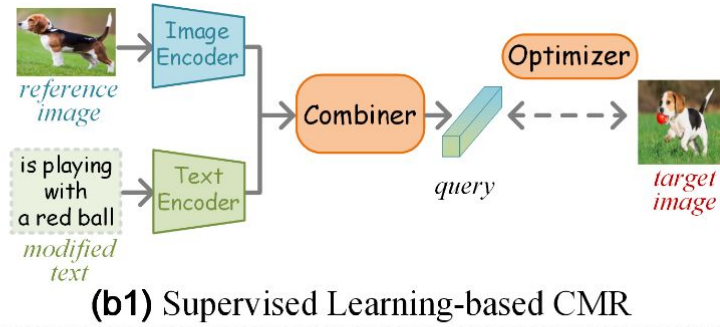
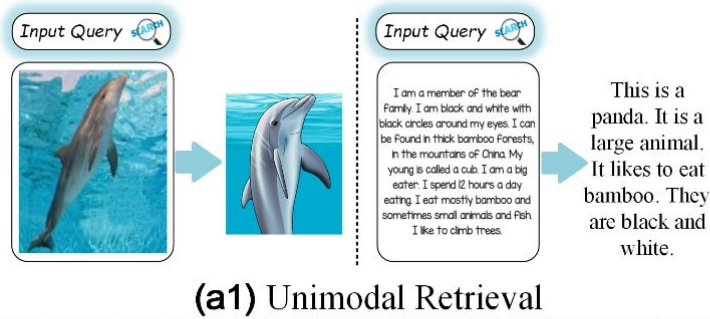
K1: The Empire State Building is a 102-story Art Deco skyscraper in Midtown Manhattan, New York City

K2: The 828 metre (2,717 ft) tall Burj Khalifa in Dubai has been the tallest building since 2010. The Burj Khalifa has been classified as megatall.

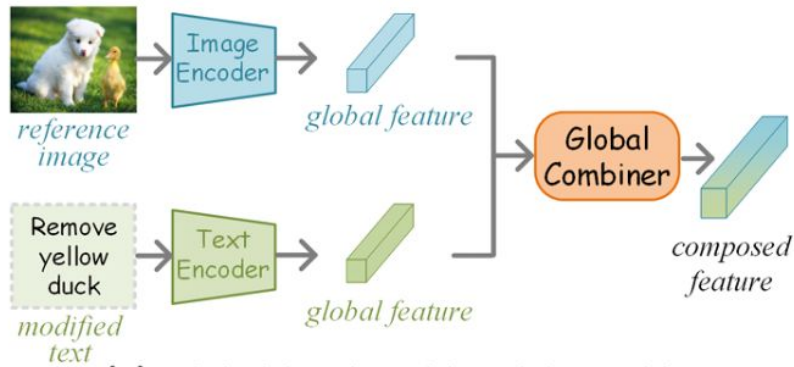
K3: The tallest building in New York is One World Trade Center which rise 1,776 feet (541 m).

[2023.acl-long.478.pdf](#)

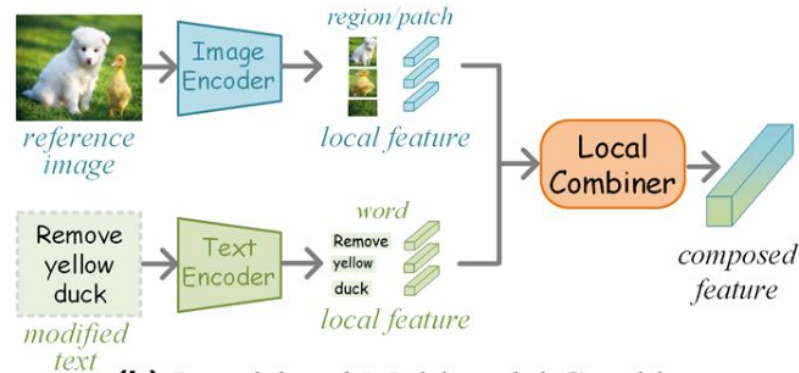
RAG - CMR



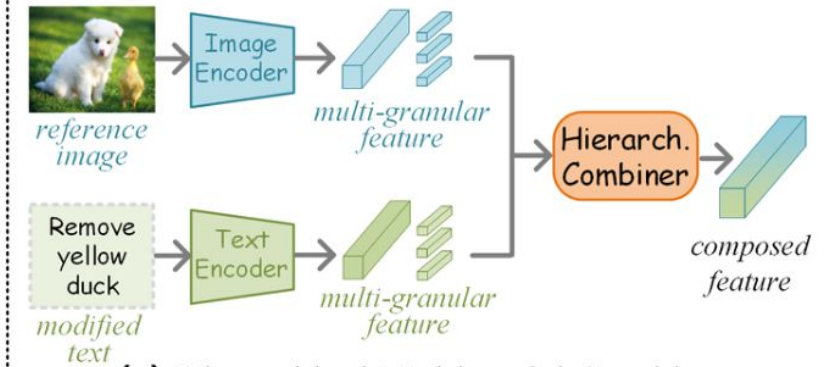
RAG - CMR



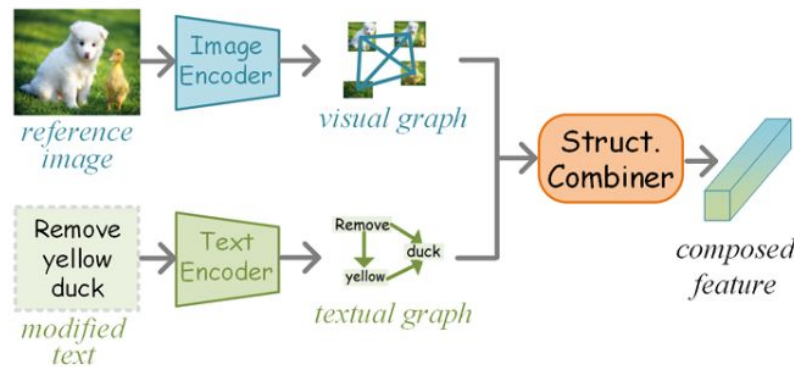
(a) Global-level Multimodal Combiner



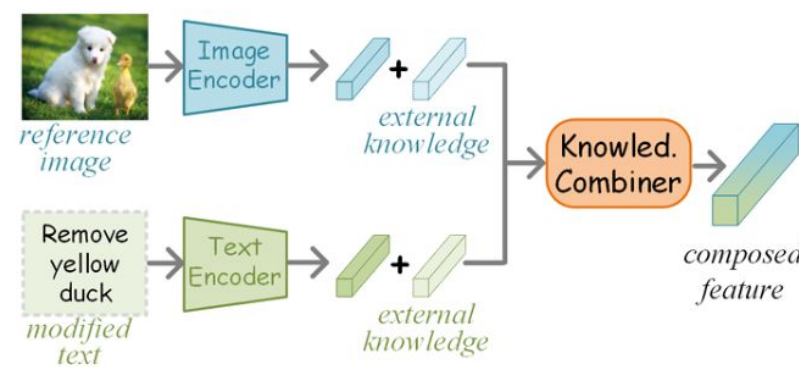
(b) Local-level Multimodal Combiner



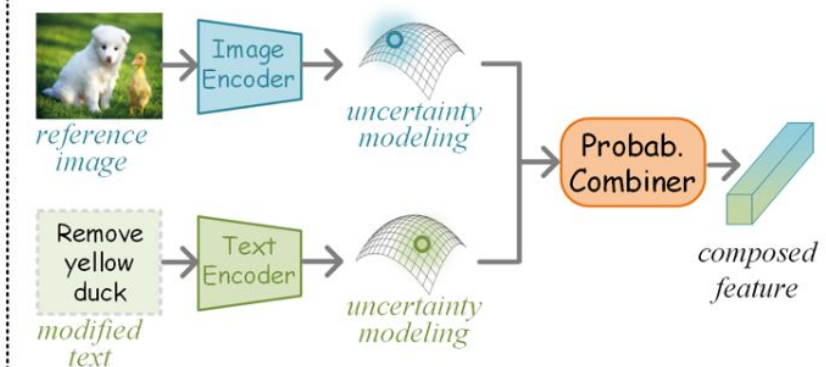
(c) Hierarchical Multimodal Combiner



(d) Structured Relational Multimodal Combiner



(e) Knowledge-Enhanced Multimodal Combiner



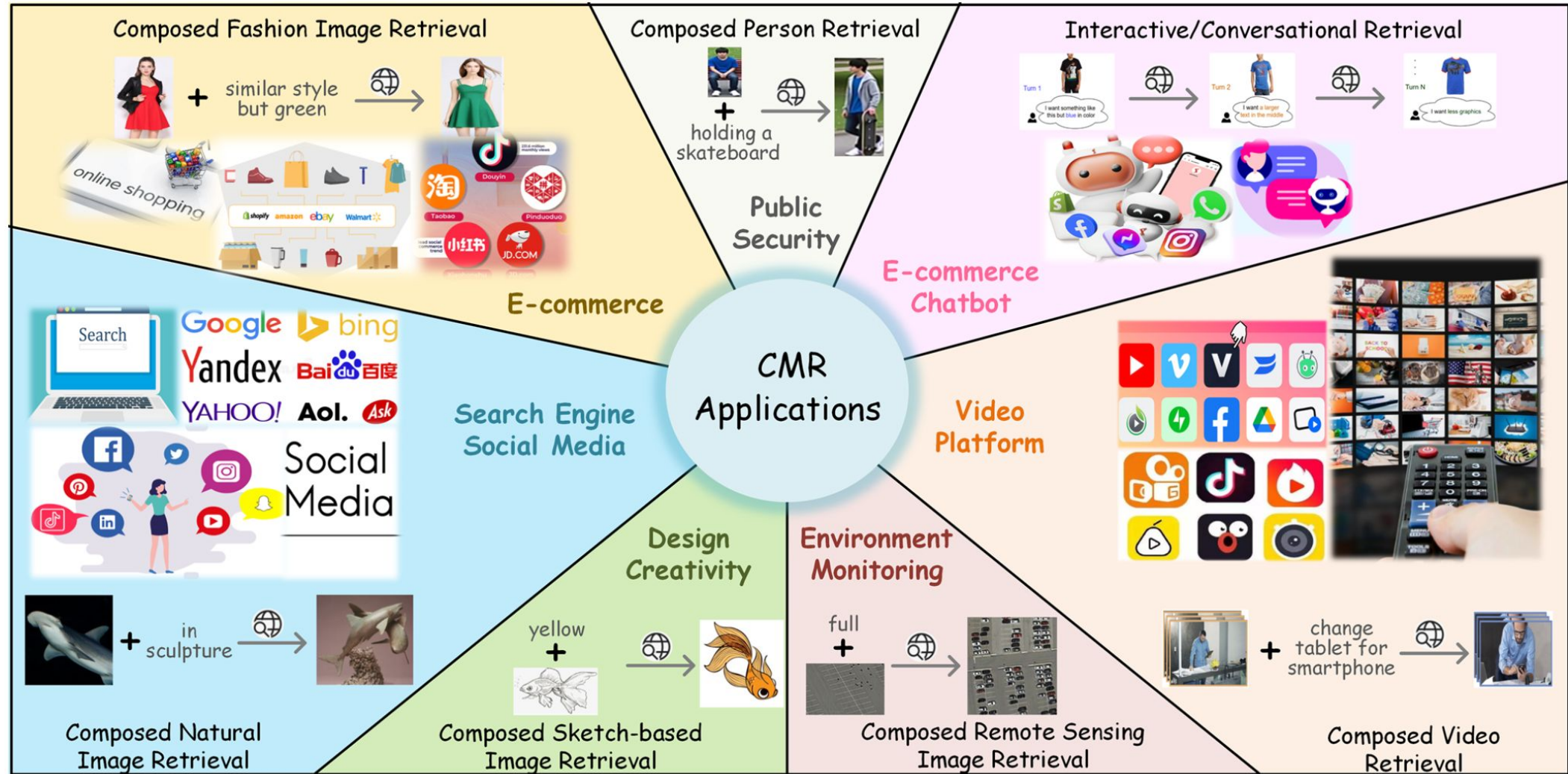
(f) Ambiguous Probabilistic Multimodal Combiner

[Composed Multi-modal Retrieval: A Survey of Approaches and Applications](#)

[\(PDF\) A survey on knowledge-enhanced multimodal learning](#)

[Divide, Conquer and Combine: Hierarchical Feature Fusion Network with Local and Global Perspectives for Multimodal Affective Computing](#)

RAG - CMR Aplicacion



Ejemplo de query multimodal

Consulta del usuario: "Imágenes de personas usando casco en un sitio de construcción, similares a esta foto."

Entrada multimodal: Una imagen y un texto complementario

Resultado esperado: Un sistema que interpreta ambas señales (imagen + texto) para recuperar imágenes relevantes, no solo visualmente similares sino también semánticamente alineadas con la descripción textual.

Esta query aporta: desambiguación, evita resultados irrelevantes como personas con cascos de bicicleta o en interiores.

Es ideal en los dominios visuales complejos donde el lenguaje no alcanza a describir todo, pero la imagen sola tampoco basta.

Que veremos en el ejemplo?

BM25 (búsqueda léxica): ranking clásico por coincidencia de términos y normalización por longitud sobre chunks.

Embeddings + Pinecone (búsqueda vectorial): all-MiniLM-L6-v2 normalizado (coseno) con metadatos (doc_id, page, source) y soporte de namespace/meta_filter.

RRF (Reciprocal Rank Fusion): combina listas BM25 y vectorial por posición ($1/(k+\text{rank})$), sin normalizar puntajes.

Cross-Encoder (re-ranqueo): modelo MS MARCO MiniLM puntúa pares (query, chunk) para refinar el orden final.

Diversidad por documento: per_doc_cap y deduplicación de doc_id para evitar que un doc monopolice el top-k.

Citas y contexto: construcción de contexto con [source, p. X]; opción de resumen rag_summary.py.

Evaluación IR: Precision@k, Recall@k, nDCG, MRR; comparación pre (RRF) vs post (tras Cross-Encoder).

Evaluación

- **Retrieval: Recall@k (doc/pasaje):** proporción de queries donde el **soporte correcto** aparece en el top-k. *Meta:* $\geq 0.85 @20$; **nDCG@k / MRR:** prioriza orden; mide si lo relevante queda arriba; **Hit rate@k:** % de queries con ≥ 1 pasaje válido en top-k; **Diversidad/De-dup:** % de pasajes no redundantes; **Freshness:** lag medio entre cambio en fuente e índice actualizado; **Coste/latencia del retriever:** p50/p95.
- **Contexto: Context Precision:** % del texto incluido que **sí** respalda la respuesta; **Context Recall (Coverage):** si el **soporte necesario** está completo en el contexto final; **Citación direccionable:** % de respuestas con **IDs/URLs** que apuntan a la evidencia exacta; **Compresión efectiva:** tokens de contexto vs soporte real (menos “paja”, mismo recall).
- **Generación: Exactitud (EM/F1):** sobre un conjunto con *ground truth*; **Faithfulness/Attribution:** % de oraciones verificables respaldadas por el contexto (y no contradichas); **Hallucination rate:** % de afirmaciones **no soportadas**; **Abstención correcta:** cuando **no hay evidencia**, ¿se niega apropiadamente?; **Formato / contrato:** % de **JSON/SQL** válidos, schemas, plantillas; **Citación correcta:** % de respuestas que **citan** los pasajes adecuados (no solo “cualquier doc”).
- **Re-rank / Fusión: Pairwise accuracy / nDCG lift:** mejora del re-ranker sobre el candidato inicial; **Weighted/RRF gain:** mejora de híbrido (sparse+denso) vs cada señal por separado.
- **Robustez & Seguridad Paráfrasis/ruido:** caída de métricas ante reformulaciones, typos, queries largas/cortas; **OOD/Domain shift:** desempeño con temas nuevos; **Políticas:** filtrado de PII, cumplimiento de normas, jailbreak resistance básica.
- **Operación (SLOs): Latencia end-to-end (p50/p95), Costo por respuesta (tokens/\$), Tasa de errores:** timeouts, context overflow, fallos de fetch

Recuerden formar los grupos!

**Gracias por su atención y
dedicación.**

Recuerden que los grandes retos traen grandes aprendizajes.

¡Nos vemos en la próxima clase!