MEMORIA DEL PROYECTO

“Diseño de un Sistema RAG con Información Actualizada: Informe del Sector Audiovisual 2025”

1. Selección y justificación del modelo de *embedding*

Para este proyecto se ha seleccionado el modelo **all-MiniLM-L6-v2** de la librería *Sentence-Transformers*. Se trata de un modelo ligero, eficiente y ampliamente utilizado para tareas de similitud semántica. Su representación vectorial de 384 dimensiones permite transformar textos en vectores numéricos que capturan su significado, lo que facilita comparar preguntas y fragmentos de texto por cercanía semántica.

La elección se justifica porque este modelo:

- Ofrece un **rendimiento excelente en CPU**, sin necesidad de GPU.
- Presenta **buen desempeño en inglés y español**, lo que resulta adecuado para procesar documentación del sector audiovisual.
- Es **muy eficiente para prototipos RAG** (Retrieval-Augmented Generation) locales, donde la prioridad es la rapidez y el bajo consumo de recursos.

Como posibles **alternativas futuras**, se propone explorar modelos como **bge-small-es** o **multilingual-e5-small**, que ofrecen una mejor semántica en castellano con un coste computacional similar.

No se ha considerado necesario aplicar *fine-tuning* en esta fase, aunque podría implementarse más adelante si se amplía el dominio del sistema (por ejemplo, con boletines autonómicos o artículos especializados). En ese caso, se podría realizar un entrenamiento supervisado con pares reales de *pregunta-fragmento* para mejorar la recuperación de información relevante.

---

2. Diseño y justificación de la base de datos vectorial

Para la gestión de los vectores y metadatos se ha utilizado la base de datos **Qdrant**, una herramienta *open source* especializada en el almacenamiento y búsqueda de vectores de alta dimensión.

Esta elección se fundamenta en los siguientes motivos:

- **Facilidad de uso**: Qdrant dispone de una API sencilla y una integración directa con *Python* y *FastAPI*.

- **Eficiencia**: utiliza el índice **HNSW (Hierarchical Navigable Small World)**, que permite búsquedas aproximadas extremadamente rápidas.

- **Persistencia local**: puede ejecutarse en contenedor Docker y guardar los datos en disco, asegurando la conservación de la información entre sesiones.

- **Filtros por metadatos (payload)**: lo que permite buscar no solo por similitud vectorial, sino también por campos como el nombre del documento o la fecha.

El esquema de la colección definida, llamada **"audiovisual_2025"**, incluye:

- Un **vector** de 384 dimensiones asociado a cada fragmento de texto.

- Un **payload** con información contextual: texto original, fuente, número de página, identificador de fragmento (*chunk_id*) y fecha de creación.

Además, se ha creado un índice adicional sobre el campo text para habilitar búsquedas por coincidencia de palabras cuando sea necesario.
La métrica elegida ha sido **cosine**, ya que es la más adecuada para medir la similitud entre embeddings.

En cuanto a la organización, se ha optado por una única colección para el dominio "audiovisual_2025". En caso de que el sistema crezca, se podrían crear *namespaces* independientes por año o temática (por ejemplo, "audiovisual_2026" o "producción_cinematográfica").

---

3. Estrategia para mantener la información actualizada

Uno de los principales retos de un sistema RAG es mantener su base de conocimiento actualizada con la información más reciente.
En este proyecto, la fuente principal es el **Informe del Sector Audiovisual 2025**, de carácter anual, por lo que se ha diseñado una **estrategia incremental** para incorporar nuevos informes sin necesidad de reconstruir la base de datos desde cero.

a) Fuentes de información

Para futuras ampliaciones, se propone complementar los informes anuales con fuentes dinámicas como:

- **Spain Audiovisual Hub**, que publica informes y artículos sobre el sector.
- **ICAA** y **MITES**, con datos sobre empleo y producción audiovisual.
- **Prensa especializada**, como *Audiovisual451* o boletines autonómicos.

b) Ingesta programada

La actualización podría automatizarse mediante una tarea periódica (por ejemplo, semanal) que:

1. Descargue automáticamente nuevos informes o artículos (PDF, HTML, RSS) a la carpeta data/raw/.
2. Procese el texto mediante el script ingest_pdf.py, que lo normaliza y trocea según los parámetros max_chars y overlap.
3. Genere un **hash** para cada fragmento y lo compare con los existentes en Qdrant:
    - Si no existe, se inserta (upsert).
    - Si ha cambiado, se actualiza (update).
    - Si ha caducado, se marca con un campo valid_to.

c) Re-embeddings

Solo se recalcularán los embeddings de los fragmentos nuevos o modificados, evitando reprocesar información inalterada.

d) Política de borrado

Se mantendrá un registro histórico marcando los fragmentos antiguos con el campo valid_to, en lugar de borrarlos directamente (*soft delete*).

e) Prioridad de datos recientes

Se añadirá un campo temporal (ingested_at o year) para que el sistema priorice la información más reciente al generar respuestas.
En caso de empate entre fragmentos de igual similitud, el re-ranking local podrá ordenar por fecha.

f) Observabilidad

Se incorporan endpoints para verificar el estado general del sistema:

- /health para comprobar la conexión con Gemini y Qdrant.

- /stats para conocer el número de puntos indexados y las fuentes registradas.

---

4. Integración con el modelo de lenguaje (Gemini)

El modelo de lenguaje utilizado es **Gemini 2.5 Flash**, elegido por su rapidez, bajo coste y buena comprensión del español, lo que lo convierte en una opción adecuada para proyectos de tipo RAG.

El flujo de integración se desarrolla de la siguiente manera:

1. El usuario envía una pregunta a través del endpoint /ask.
2. La pregunta se convierte en un vector mediante el modelo de embeddings.
3. Este vector se utiliza para buscar los fragmentos más relevantes en Qdrant.
4. Se construye un contexto con los fragmentos mejor puntuados (hasta un número máximo).
5. Ese contexto se incluye dentro de un *prompt* en español, que se envía al modelo Gemini para generar la respuesta final.
6. El resultado incluye la respuesta y las fuentes utilizadas.

Optimización del contexto

Para garantizar respuestas precisas y breves:

- Se **eliminan duplicados** de fuentes.
- Se **limita la longitud total del contexto** para no sobrepasar el número de tokens del modelo.
- Se da preferencia a los fragmentos más recientes y relevantes.
- Si la similitud es demasiado baja, el sistema devuelve una respuesta neutra:
  “No se encontró información relevante en el contexto.”

---

5. Arquitectura del sistema

La solución sigue una arquitectura sencilla basada en FastAPI que orquesta tres piezas clave: embeddings (Sentence-Transformers), Qdrant (base vectorial) y Gemini (LLM). En la siguiente figura se muestra el diagrama general; debajo se explica cómo leerlo paso a paso.

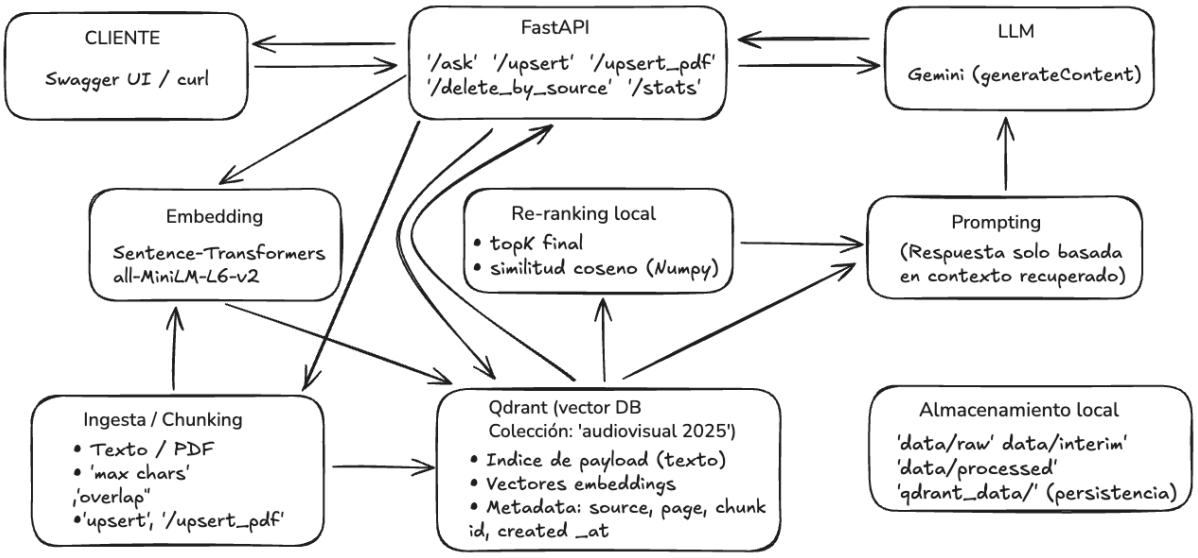

Figura 1. Arquitectura y flujo del sistema RAG Audiovisual 2025 con Gemini (consulta, ingesta y mantenimiento).

5.1 Componentes (qué es cada 'caja' del diagrama)

- Cliente (Swagger UI / curl): interfaz para probar los endpoints de la API.

- FastAPI: servidor con los endpoints /ask (consulta), /upsert (ingesta de texto), /upsert_pdf (ingesta de PDF), /delete_by_source (limpieza por fuente) y /stats (métricas).

- Embedding (Sentence-Transformers all-MiniLM-L6-v2): convierte textos en vectores numéricos ('embeddings') para poder buscar por significado.

- Qdrant (vector DB, colección audiovisual_2025): almacena embeddings y metadatos (texto, source, page, chunk_id, created_at) y permite filtrar por payload (por ejemplo, MatchText sobre text).

- Re-ranking local: reordena los resultados de Qdrant calculando la similitud coseno con la pregunta, para quedarse con los top-k más relevantes.

- Prompting: arma el contexto con los fragmentos recuperados y construye un prompt estricto ('responde solo con el contexto').

- LLM (Gemini – generateContent): genera la respuesta final en lenguaje natural a partir del contexto.

- Almacenamiento local: data/raw, data/interim, data/processed (archivos originales y procesados) y qdrant_data/ (persistencia de Qdrant).

5.2 Flujos operativos (cómo se mueve la información)

A) Consulta /ask (QA con RAG)

1. Cliente → FastAPI: envía {question, top_k, filter_text?, debug?}.
2. FastAPI → Embedding: la pregunta se vectoriza.
3. Embedding → Qdrant: búsqueda semántica (top-k ampliado); si procede, filtro por texto (MatchText sobre payload.text).
4. Qdrant → Re-ranking local: se reordena por coseno y se toman los top_k definitivos.
5. Re-ranking → Prompting → Gemini: se construye el contexto y Gemini devuelve la respuesta.
6. FastAPI → Cliente: retorna {question, answer, sources[], debug?}.

B) Ingesta/actualización

1) Texto con /upsert:
- Cliente envía {text, source, max_chars, overlap}.
- Chunking divide el texto en fragmentos; Embedding crea sus vectores.
- Qdrant hace upsert (vector + payload con text, source, page?, chunk_id, created_at).
- Respuesta: {status:'ok', upserted:n}.

2) PDF con /upsert_pdf:
- Cliente sube PDF (multipart) + source y parámetros.
- Extracción (PyMuPDF/pdfminer.six) → normalización → chunking (guardando page).
- Embedding y upsert en Qdrant (con metadatos de página).
- Respuesta: {status:'ok', upserted:n}.

C) Mantenimiento y métricas

- Limpieza /delete_by_source: borra puntos cuyo payload.source coincide.
  Respuesta: {status:'ok', deleted_estimate:n}.
- Métricas /stats: devuelve {collection, total_points, sources{…}} para revisar tamaño e ingestiones.

D) Persistencia

- Qdrant (Docker) ↔ qdrant_data/: guarda colección, índices y metadatos para no perder información entre ejecuciones.
- Datos de proyecto en data/raw|interim|processed permiten reproducibilidad (PDF/TXT originales, troceos, etc.).