Escuela
Politécnica
Superior

LLMSearch: Buscador multimedia basado en lenguaje natural

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Izan Gandía Ruiz
Tutor:
Iván Gadea Saéz

Universitat d'Alacant
Universidad de Alicante

Mayo 2025

LLMSearch: Buscador multimedia basado en lenguaje natural

Encuentra tu contenido multimedia al instante con solo describirlo.

Autor
Izan Gandía Ruiz

Tutor
Iván Gadea Saéz
Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Mayo 2025

Preámbulo

Este proyecto surge de una doble motivación. Por un lado, el profundo interés en explorar y adquirir un conocimiento avanzado sobre el uso y configuración de las inteligencias artificiales multimodales. Por otro lado, la identificación de una problemática recurrente y extendida en la gestión de la información digital personal como es la dificultad persistente para localizar archivos específicos (como imágenes o documentos) en volúmenes grandes de datos. Este desafío se presenta también en contextos más actuales, como la búsqueda de elementos específicos (ej. "stickers") dentro de aplicaciones de mensajería como WhatsApp o Telegram.

Agradecimientos

Este rinconcito es para vosotros, para toda esa gente increíble que ha hecho posible que hoy esté aquí, escribiendo estas líneas.

Primero, a ese montón de compañeros y amigos que me he cruzado en la carrera. He aprendido un millón de cosas con vosotros, no solo de manera académica, sino lecciones de vida que se quedan para siempre. Sin vuestras risas, vuestros ánimos en los momentos de bajón y esa forma de tirar para adelante juntos, no sé si habría encontrado las fuerzas para seguir tantas veces. Sois, en gran parte, la razón de que esté celebrando este logro.

A mi familia, mi pilar fundamental. Gracias por creer en mí incluso cuando yo dudaba, por ponerme las cosas fáciles y por todo el apoyo para que pudiera dedicarme a esto. Sois increíbles. Y un gracias enorme y especial para mi hermano, Abel Gandía Ruiz. Tú fuiste quien me abrió los ojos a este mundo tan interesante de la programación cuando yo no tenía ni idea, quien me animó y me echó una mano para empezar.

También quiero acordarme de mis profes. Algunos habéis sido una inspiración, de esos que te contagian la ilusión y te hacen descubrir la magia en sitios donde nunca te lo hubieras imaginado. Gracias por inspirarme y ayudarme a ser un mejor ingeniero.

Y, cómo no, a mi tutor, Iván Gadea Sáez. Gracias por guiarme con este proyecto, por tu paciencia infinita y por ayudarme a calmar todos los nervios y dudas que me han ido surgiendo.

De verdad, a todos y cada uno, ¡muchísimas gracias!

A quienes me inspiraron a soñar y a programar, recordándome que, si puedo imaginarlo, puedo crearlo. [1]

[1] Alejandro Taboada, creador del canal "Programación ATS"

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1. Introducción

La gestión y recuperación eficiente de la información digital se ha convertido en un desafío cotidiano en la era de la sobrecarga informativa. Los volúmenes de datos personales y profesionales que almacenamos en nuestros dispositivos crecen exponencialmente, mientras que las herramientas tradicionales de búsqueda a menudo resultan insuficientes para localizar archivos específicos de manera rápida y precisa. Este proyecto se adentra en esta problemática, proponiendo una solución innovadora basada en los avances recientes en Inteligencia Artificial (IA) y sistemas de Generación Aumentada por Recuperación (RAG).

1.1. Panorama Actual: Desafíos en la Recuperación de Información

Los métodos convencionales para la organización y búsqueda de archivos digitales se basan en gran medida en metadatos explícitos, como nombres de archivo, fechas o etiquetas manuales. Sin embargo, estas aproximaciones tienen limitaciones significativas:

- **Insuficiencia de los metadatos tradicionales:** A menudo, los metadatos son inexistentes, incompletos o no capturan la semántica real del contenido del archivo (especialmente en el caso de imágenes, vídeos o audios).
- **Falta de precisión en las búsquedas:** Las búsquedas basadas en palabras clave pueden ser ambiguas y no siempre interpretan correctamente la intención del usuario, llevando a resultados irrelevantes o a la omisión de la información que el usuario desea encontrar.
- **Desafíos técnicos y éticos en IA:** Si bien la IA ofrece nuevas vías, también enfrenta retos. Los modelos pueden carecer de la precisión necesaria para ciertas tareas o, en el caso de modelos generativos, incurrir en "alucinaciones" (generar información incorrecta pero plausible). Además, el propio entrenamiento del modelo puede afectar a la interpretación sobre el archivo que se quiere describir, generando resultados no equitativos o discriminatorios, lo que plantea también importantes consideraciones éticas.
- **Riesgos de seguridad y privacidad de los datos:** La externalización del almacenamiento a la nube y el uso de IA para la catalogación y el análisis de ficheros introducen nuevas formas de ataque y preocupaciones sobre la seguridad y privacidad. Estos sistemas pueden ser vulnerables a accesos no autorizados, fugas de datos o ciberataques, tanto en la infraestructura cloud como en los propios modelos de IA. La información sensible contenida en los archivos, o incluso los metadatos enriquecidos generados por la IA, podrían quedar expuestos, ser alterados o perderse. Esto es especialmente crítico cuando se manejan datos confidenciales o personales, donde una brecha de seguridad no solo implica la pérdida de información valiosa, sino también posibles consecuencias legales, financieras y reputacionales significativas.

Este contexto muestra la necesidad de sistemas más inteligentes y contextuales capaces de comprender el contenido de los archivos de forma más profunda, más allá de sus metadatos superficiales, y que al mismo tiempo garanticen la integridad y confidencialidad de la información.

1.2. Avances Tecnológicos Fundamentales

Para abordar los desafíos mencionados, este proyecto se apoya en los desarrollos más recientes en el campo de la Inteligencia Artificial, particularmente en las siguientes áreas:

1.2.1. Inteligencia Artificial Multimodal: Convergencia de Lenguaje y Visión

La Inteligencia Artificial ha experimentado avances exponenciales, especialmente con el auge del Procesamiento del Lenguaje Natural (PLN) y la Visión Artificial. La multimodalidad representa la capacidad de los sistemas de IA para procesar, comprender y generar información a partir de múltiples tipos de datos (o "modalidades") simultáneamente, como texto, imágenes, audio y vídeo.

- Procesamiento del Lenguaje Natural (PLN): Permite a las máquinas comprender, interpretar y generar lenguaje humano. Los Grandes Modelos de Lenguaje (Large Language Model (LLM)), como Generative Pre-trained Transformer (GPT) (Generative Pre-trained Transformer) y sus variantes, han revolucionado este campo, demostrando una capacidad asombrosa para entender el contexto, generar texto coherente e incluso razonar sobre la información proporcionada.
- Visión Artificial: Es la disciplina que permite a las máquinas "ver" e interpretar el contenido de imágenes y vídeos. Implica tareas como la detección de objetos, el reconocimiento facial, la segmentación de imágenes y la generación de descripciones visuales.
- Modelos Multimodales (ej. Contrastive Language-Image Pre-training (CLIP))[1]: Modelos como CLIP (Contrastive Language-Image Pre-training) de OpenAI son un ejemplo paradigmático de esta convergencia. CLIP aprende representaciones visuales a partir de descripciones en lenguaje natural, permitiendo realizar búsquedas de imágenes mediante consultas textuales con una alta comprensión semántica, o viceversa. Funciona entrenando un codificador de imágenes y un codificador de texto para predecir qué imágenes se emparejan con qué textos en un gran conjunto de datos.

1.2.2. Optimización de Modelos: Cuantización y Modelos Ligeros

Para la aplicación práctica de estos modelos, especialmente en entornos con recursos limitados (como dispositivos personales), es crucial considerar su eficiencia.

- Modelos Cuantizados: La cuantización es un proceso que reduce la precisión numérica de los pesos y activaciones de un modelo de red neuronal (por ejemplo, de punto flotante de 32 bits a enteros de 8 bits). Esto disminuye significativamente el tamaño del modelo y acelera la inferencia, con una pérdida de precisión a menudo mínima.

[1] Más información sobre CLIP de OpenAI disponible en: `https://openai.com/es-ES/index/clip/`

- Modelos Ligeros (Lightweight Models): Son arquitecturas de redes neuronales diseñadas específicamente para ser computacionalmente eficientes y tener un tamaño reducido, facilitando su despliegue en dispositivos móviles o embebidos sin sacrificar excesivamente el rendimiento.

1.2.3. Sistemas de Generación Aumentada por Recuperación (RAG)

La Generación Aumentada por Recuperación (RAG) es una técnica que mejora el rendimiento de los LLM al conectarlos con fuentes de conocimiento externas. En lugar de depender únicamente de la información (potencialmente desactualizada o incompleta) aprendida durante su entrenamiento, un sistema RAG funciona en dos fases:

1. Recuperación (Retrieval): Dada una consulta del usuario, el sistema primero busca y recupera fragmentos de información relevante de una base de datos, un conjunto de documentos o un corpus de conocimiento. Esta base de datos puede estar compuesta por embeddings (representaciones vectoriales densas) del contenido de los archivos.

2. Generación (Generation): La información recuperada se proporciona como contexto adicional al LLM junto con la consulta original. El LLM utiliza este contexto enriquecido para generar una respuesta más precisa, relevante y fundamentada.

Los sistemas RAG ofrecen ventajas significativas, como la reducción de alucinaciones, la capacidad de citar fuentes del contexto dado y la facilidad para actualizar la base de conocimiento sin necesidad de reentrenar el LLM completo. Si bien existen diversas arquitecturas RAG (ej. generando consultas SQL, incorporando texto directamente al prompt, o utilizando embeddings), el enfoque basado en embeddings suele ofrecer un buen equilibrio entre eficiencia y calidad de los resultados, aunque presenta desafíos como la gestión de la ventana de contexto del LLM, punto crucial a tener en cuenta si se utilizan sobre dispositivos personales.

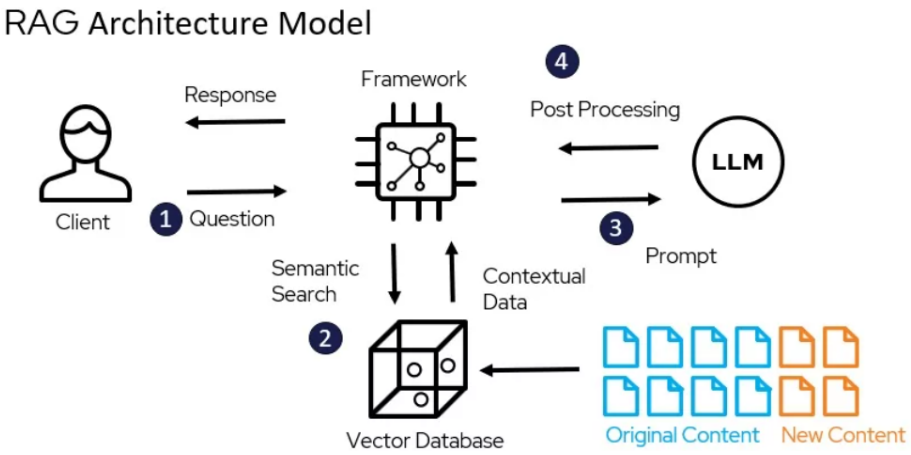

Figura 1.1: Esquema visual del funcionamiento de un sistema RAG, mostrando el flujo desde la consulta del usuario, pasando por la recuperación de información relevante, hasta la generación de la respuesta final por el LLM.

1.2.4. Justificación del Proyecto

Este Trabajo Final de Grado (TFG) busca abordar la necesidad de obtener información de manera rápida, sencilla y muy precisa desarrollando un sistema inteligente de búsqueda de archivos que permita a los usuarios encontrar información utilizando consultas en lenguaje natural, trascendiendo las limitaciones de las búsquedas basadas en metadatos tradicionales. La aplicación de modelos multimodales permitirá indexar el contenido semántico de diversos tipos de ficheros, y la arquitectura RAG proporcionará un marco robusto para recuperar la información más relevante y presentarla de forma útil al usuario.

2. Estado del Arte

Antes de adentrarse en los detalles técnicos, es esencial establecer el contexto actual en el dominio de los buscadores multimedia basados en lenguaje natural. Esta revisión permitirá asentar una fundamentación teórica y metodológica sólida, comprender los desafíos y las limitaciones identificadas en investigaciones previas e identificar las brechas en el conocimiento existente, así como las oportunidades para realizar contribuciones significativas en LLMSearch.

2.1. Modelos de Lenguaje Natural (LLMs) para Búsqueda

Los **LLMs** han revolucionado el procesamiento del lenguaje natural en años recientes. Modelos como *GPT-3* y *GPT-4* (base de **ChatGPT**) demuestran que, con miles de millones de parámetros entrenados en enormes corpus de texto, es posible comprender y generar lenguaje con notable fluidez y contexto. Estos modelos capturan representaciones semánticas ricas, lo que habilita nuevas formas de **búsqueda semántica** y recuperación de información.

Características clave:

- **Búsqueda por significado**: En lugar de limitarse a coincidencias de palabras clave, un LLM puede interpretar la intención de una consulta en lenguaje natural y relacionarla con documentos relevantes aunque no compartan palabras literalmente.

- **Embeddings semánticos**: Técnicas como *embeddings* de oraciones (usando modelos tipo Bidirectional Encoder Representations from Transformers (BERT) o Sentence Transformers) convierten documentos y consultas a vectores en un espacio vectorial común, donde la similitud de coseno permite recuperar los contenidos más cercanos en significado.

- **RAG**: Los LLMs pueden integrarse en pipelines donde primero se recuperan documentos candidatos y luego el modelo genera una respuesta o resumen usando esos textos.

- **Interfaz conversacional**: Modelos tipo ChatGPT permiten refinar iterativamente las consultas de búsqueda mediante diálogo, mejorando la precisión de resultados en consultas ambiguas.

Los avances más recientes se centran en mejorar la **eficiencia y apertura** de estos modelos. Mientras GPT-4 (de OpenAI) es de uso cerrado y con un tamaño muy grande no divulgado (>100B parámetros), han emergido modelos de código abierto como *LLaMA* (Meta) y sus variantes, que con 7–70B parámetros logran desempeños competitivos.

2.2. Modelos Visión-Lenguaje para Imágenes

En un buscador multimedia, es esencial manejar consultas sobre contenido visual (imágenes) usando lenguaje natural. Aquí destacan los **modelos visiolingüísticos** o **Modelo de Visión-Lenguajes (VLMs)**, que conectan representaciones de imágenes con representaciones textuales en un espacio común.

2.2.1. CLIP y Embeddings Multimodales

Un hito fue el modelo **CLIP** de OpenAI, que entrena conjuntamente un codificador de texto (transformer) y un codificador visual (Red Neuronal Convolucional o Vision Transformer (ViT)) para proyectar ambos tipos de entrada en **vectores de embedding** de la misma dimensión. Mediante aprendizaje contrastivo en 400 millones de pares imagen–texto, CLIP logró que textos e imágenes con contenido semántico equivalente quedaran cercanos en el espacio vectorial.

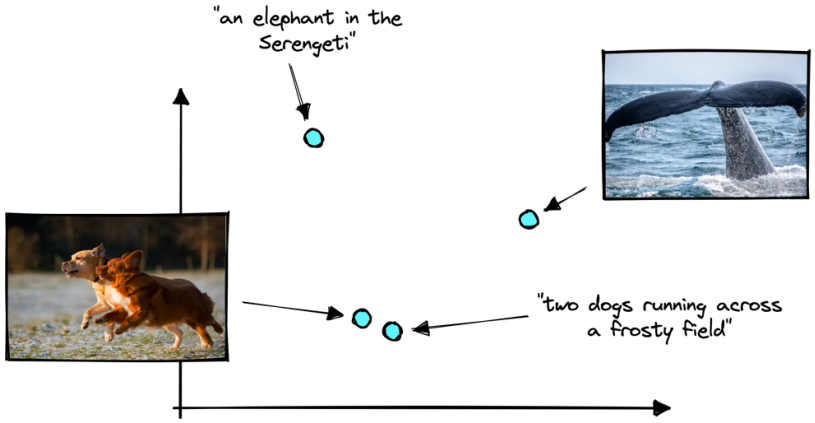

Figura 2.1: Ejemplo conceptual de un espacio vectorial multimodal entrenado por CLIP, donde imágenes y descripciones semánticas correspondientes se representan mediante vectores cercanos.

2.2.2. Modelos Generativos de Descripción de Imágenes

Otra línea de desarrollo significativa se centra en los **modelos generativos de descripción de imágenes**. Estos sistemas realizan la tarea conocida como *image captioning*, que consiste en generar una descripción en lenguaje natural para una imagen dada. Modelos recientes como **Bootstrapping Language-Image Pre-training (BLIP)-2** ejemplifican esta aproximación, combinando un encoder visual pre-entrenado (por ejemplo, CLIP ViT), un modelo de lenguaje grande congelado y un transformador ligero intermedio denominado Q-Former. Esta arquitectura logra puentear eficientemente la brecha entre visión y lenguaje: el encoder de imagen extrae las características visuales relevantes, mientras que el LLM se encarga de generar la descripción textual coherente.

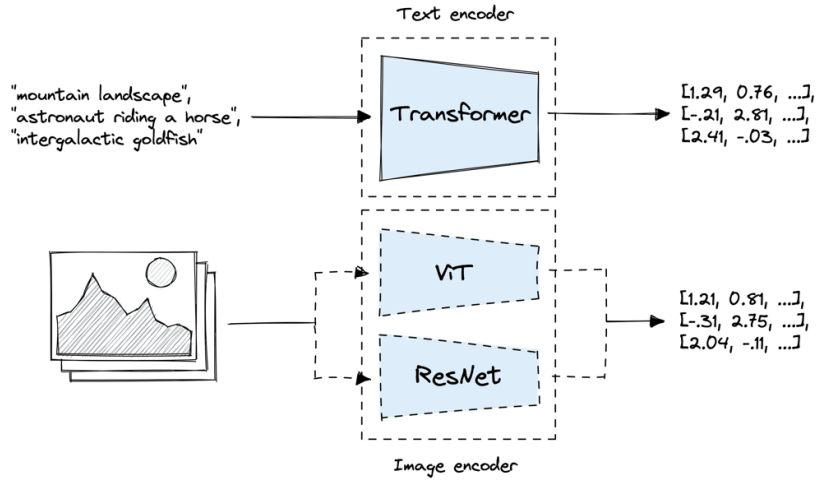

Figura 2.2: Arquitectura del modelo CLIP: encoder de texto y encoder de imagen que proyectan al mismo espacio de embedding.

2.2.3. VQA y Diálogo Multimodal

Junto al desarrollo de modelos como BLIP-2, han aparecido numerosos modelos abiertos que permiten la **Pregunta-Respuesta Visual (VQA)** y el diálogo multimodal. Entre ellos destaca **Large Language and Vision Assistant (LLaVA)**, que utiliza GPT-4 para generar datos sintéticos de entrenamiento y posteriormente afina un modelo basado en *Vicuna* (un derivado de LLaMA) acoplado a un encoder visual. Otro modelo relevante es **Moondream**, un VLM open-source de tan solo 2 mil millones de parámetros (2B), capaz de operar en tiempo real incluso en CPUs o dispositivos móviles. Moondream ha demostrado capacidades notables en la generación de descripciones detalladas, la respuesta a preguntas visuales, la detección de objetos en modalidad cero-shot y el Optical Character Recognition (OCR) básico para leer texto en imágenes. En esta misma línea, **JoyCaption** se presenta como un modelo de captioning de imágenes libre y sin censura, concebido originalmente para generar descripciones ricas que ayuden a entrenar modelos de difusión. Finalmente, aunque de naturaleza propietaria, **GPT-4 con visión** (GPT-4V) ha demostrado capacidades impresionantes al responder con acierto a entradas que combinan imagen y texto, si bien su acceso limitado restringe su uso en entornos académicos.

En síntesis, el estado del arte en la convergencia de imagen y lenguaje ofrece dos enfoques complementarios para la búsqueda multimedia. Por un lado, los *embeddings* multimodales tipo CLIP permiten realizar una **búsqueda directa por similitud** entre consultas textuales y contenido visual. Por otro lado, los *modelos generativos visiolingüísticos* facilitan la **describir o entender imágenes en texto**, permitiendo indexar y razonar sobre imágenes mediante lenguaje natural.

2.3. Modelos Multimodales para Vídeo

Extender la búsqueda basada en lenguaje natural al dominio del **vídeo** conlleva retos adicionales, pues los vídeos combinan secuencias de imágenes con audio y, en ocasiones, texto

incrustado.

2.3.1. Técnicas de Procesamiento de Video

Para abordar la complejidad del vídeo, se emplean diversas técnicas. Una fundamental es el **análisis por frames**, que implica extraer fotogramas representativos del vídeo. A estos se les aplican VLMs, convirtiendo el problema de vídeo en el manejo de un conjunto de imágenes con marcas de tiempo. Paralelamente, el **procesamiento de audio** es crucial. Mediante modelos de **Automatic Speech Recognition (ASR)** como *Whisper*, es posible transcribir con alta calidad el diálogo o narración presente en los vídeos, permitiendo indexar cada vídeo por su transcripción textual completa. Además, se están desarrollando **modelos vídeo-texto end-to-end**, como *VideoCLIP*, que extienden la idea de CLIP al dominio temporal, o transformadores específicos para vídeo que realizan *video captioning*.

2.3.2. Arquitecturas para Búsqueda en Video

Una arquitectura emergente para la búsqueda en vídeo combina los enfoques anteriores en un pipeline RAG multimodal. Este sistema indexa, por un lado, los *frames* visuales mediante embeddings y, por otro, las transcripciones de voz como texto. Ante una consulta, recupera fragmentos candidatos por similitud visual o textual, y posteriormente utiliza un modelo de lenguaje para sintetizar ambas fuentes de información y determinar la respuesta más adecuada.

2.3.3. Modelos Unificados Multimodales

Recientemente, han surgido modelos unificados que procesan múltiples modalidades de forma integrada. **MiniGPT-4**, por ejemplo, puede aceptar secuencias de imágenes como entrada, simulando un vídeo corto. **MiniCPM-V** soporta entradas de vídeo directamente, generando una descripción general del contenido. Google con **Gemini** ha avanzado en la integración de visión, vídeo y sonido en un mismo LLM, y Meta con **ImageBind** ha propuesto aprender una representación común para imágenes, texto, audio y otros sensores, abriendo nuevas vías para la comprensión multimodal holística.

2.4. Análisis de Audio y Búsqueda mediante Sonido

Para completar un buscador verdaderamente multimedia, es imprescindible considerar el contenido de **audio** independiente de los vídeos, como archivos de sonido o música.

2.4.1. Procesamiento de Habla

En el caso de que el audio contenga habla, como en podcasts, grabaciones o conferencias, se aplican técnicas de ASR con modelos robustos como Whisper. Esto permite obtener una transcripción textual que se convierte en contenido indexable, facilitando búsquedas por palabras clave o semántica mediante el uso de LLMs o embeddings textuales.

2.4.2. Audio No Verbal

Para el audio que no es voz, como sonidos ambientales, música o efectos sonoros, existen modelos como **Contrastive Language-Audio Pretraining (CLAP)**. Este entrena conjuntamente un codificador de audio y uno de texto, lo que permite buscar efectos de sonido a partir de descripciones textuales ("sonido de lluvia", "pasos en la grava") y facilita la clasificación cero-shot de audio.

2.4.3. Modelos Generadores de Descripciones Auditivas

Complementariamente, modelos como **AudioCaption** de Microsoft pueden generar frases descriptivas de clips de audio. Esta capacidad permite describir cada archivo de sonido en formato textual, indexar dichas descripciones y, en consecuencia, facilitar un acceso más semántico al contenido auditivo, más allá de simples metadatos.

2.5. Comparativa de Modelos Representativos

La tabla 2.1 resume algunos modelos representativos, destacando la distinción entre modelos propietarios como ChatGPT y una creciente diversidad de iniciativas abiertas. Para el desarrollo de un sistema como **LLMSearch**, los módulos open-source son particularmente relevantes. Es factible combinar herramientas como MiniCPM-V, Moondream, Whisper y CLAP para construir un sistema completo: Whisper se encargaría de la transcripción de audio; CLAP, del indexado de sonidos no verbales; Moondream o BLIP-2, de la descripción de imágenes; y un LLM generalista como Vicuna o LLaMA podría orquestar la interacción conversacional y la fusión de información.

Modelo	Modalidades	Tamaño	Características principales
ChatGPT (GPT-4)	Texto (y visión en GPT-4V)	>100 B?	LLM propietario de OpenAI, rendimiento puntero en comprensión y generación de lenguaje.
MiniCPM-V 2.5	Texto, Imágenes, Vídeo, Audio	~8 B	Open-source, eficiente para despliegue en dispositivos; consultas multimodales.
Moondream 2	Imágenes–Texto	2 B	VLM ultraligero con VQA, captioning, detección y OCR en CPU en tiempo real.
Whisper	Audio–Texto	~1.6 B	ASR multilingüe de código abierto, muy robusto ante acentos y ruido.

Tabla 2.1: Comparativa de modelos representativos en lenguaje y multimodalidad.

2.6. Conclusión

El estado del arte actual ofrece los bloques fundamentales necesarios para construir un buscador multimedia avanzado que opere mediante lenguaje natural. Este TFG se propone integrar y adaptar estas tecnologías de vanguardia en una única plataforma unificada, denominada *LLMSearch*. El proyecto evaluará el rendimiento de esta integración y buscará proponer mejoras con el objetivo de lograr búsquedas multimodales más precisas, naturales e intuitivas para el usuario.

3. Objetivos

3.1. Objetivo general

El objetivo principal de este TFG es diseñar y desarrollar un buscador multimedia inteligente que permita a los usuarios realizar búsquedas avanzadas utilizando lenguaje natural. De esta manera, el usuario podrá localizar documentos de texto, imágenes, vídeos o archivos de audio explorando el contenido semántico intrínseco de los ficheros, trascendiendo las limitaciones de las búsquedas basadas únicamente en metadatos explícitos.

La idea es crear una herramienta que facilite a los usuarios encontrar contenido multimedia de manera eficiente y precisa mediante descripciones detalladas en lenguaje natural. Por ejemplo, se podría buscar una fotografía específica entre miles con una consulta como: "busca una foto en la que salía un elefante levantando la trompa y que la hice en Tailandia hace unos 5 o 6 años"; o encontrar un archivo PDF relevante mediante una búsqueda del tipo: "encuentra los datos para la declaración de la renta de 2016". De esta forma, se pretende obtener un sistema de búsqueda que no solo identifique el archivo específico que se busca, sino que también tenga la capacidad de extraer datos relevantes del contenido del archivo para responder a preguntas específicas formuladas en la consulta, aprovechando las capacidades de los modelos de lenguaje aumentados por recuperación (RAG).

3.2. Objetivos específicos

Adicionalmente, se plantean los siguientes objetivos secundarios que complementan y dan soporte al objetivo principal:

3.2.1. Estudiar modelos multimodales

Estudiar diferentes modelos multimodales con el fin de seleccionar aquellos que ofrezcan los mejores resultados en términos de precisión y eficiencia (tiempo de respuesta razonable).

3.2.2. Seleccionar una solución de base de datos

Investigar y seleccionar una solución de base de datos adecuada para el almacenamiento y consulta eficiente de metadatos enriquecidos y embeddings vectoriales generados por los modelos de IA.

3.2.3. Diseñar una arquitectura modular

Diseñar una arquitectura de sistema que sea modular, escalable y eficiente, permitiendo la integración de los diferentes componentes y facilitando futuras expansiones o mejoras.

3.2.4. Desarrollar una interfaz gráfica

Desarrollar una interfaz gráfica de usuario (GUI) intuitiva y amigable que permita a los usuarios interactuar fácilmente con el sistema, realizar búsquedas, visualizar los resultados obtenidos y gestionar sus archivos.

4. Metodología

En este capítulo se detalla la metodología empleada para la planificación, desarrollo y gestión del presente TFG. Se describirá tanto la organización del proyecto, basada en una adaptación de la metodología ágil Scrum, como el entorno técnico configurado, abarcando el hardware y software utilizados. El objetivo es proporcionar una visión clara de los procesos y herramientas que han sustentado la realización de LLMSearch, desde su concepción hasta la implementación de sus funcionalidades.

4.1. Organización del Proyecto y Metodología Scrum Adaptada

La gestión y desarrollo del presente TFG se ha articulado mediante una adaptación simplificada de la metodología ágil **Scrum**. Scrum es un marco de trabajo diseñado para abordar proyectos complejos, promoviendo la autoorganización de los equipos, el desarrollo iterativo e incremental a través de ciclos cortos denominados *sprints*, y la entrega continua de valor.

4.1.1. Adaptación de Roles y Dinámicas de Scrum

Dada la naturaleza individual del proyecto, donde un único estudiante es el responsable de su ejecución, los roles tradicionales de Scrum se han concentrado en esta figura. Así, el estudiante ha asumido las responsabilidades de:

- **Product Owner**: Definiendo la visión del producto (LLMSearch), gestionando el *Product Backlog* (lista priorizada de funcionalidades y requisitos) y asegurando que el desarrollo se alinea con los objetivos del proyecto.
- **Development Team**: Encargándose del diseño, implementación, pruebas y entrega de los incrementos funcionales del software en cada sprint.
- **Scrum Master**: Facilitando el proceso, eliminando impedimentos, asegurando que se sigan las prácticas ágiles adaptadas y promoviendo la mejora continua.

En este contexto adaptado, el tutor del TFG ha desempeñado un rol fundamental como **cliente principal (Stakeholder)**, proporcionando los requisitos iniciales, ofreciendo retroalimentación continua sobre los avances y validando los entregables. Su participación ha sido clave para guiar la dirección del proyecto y definir posibles ajustes a lo largo de su desarrollo.

4.1.2. Estructura y Ejecución de los Sprints

El proyecto se ha dividido en una serie de *sprints*, cada uno con una duración aproximada de dos semanas. Al inicio de cada cuatrimestre, y de manera continua, se establecieron reuniones periódicas (equivalentes a las *Sprint Planning* y *Sprint Review* de Scrum) entre el estudiante y el tutor. En estas reuniones se:

- Revisaba el progreso del sprint anterior.
- Se presentaban y discutían los avances realizados (incremento del producto).
- Se resolvían dudas y se abordaban los impedimentos identificados.
- Se definían y priorizaban los objetivos y tareas para el siguiente sprint, conformando el *Sprint Backlog*.

La planificación de los sprints ha sido un proceso dinámico, ajustándose a la evolución del proyecto y los descubrimientos realizados. A continuación, se describe de forma general la progresión del trabajo a lo largo de los sprints:

- **Sprint Inicial (Fase de Conceptualización e Investigación)**: Este sprint se centró en la definición detallada del alcance del proyecto, la elaboración del estado del arte, la investigación exhaustiva de las tecnologías y herramientas de IA pertinentes (especialmente LLMs y modelos multimodales), y la organización inicial de las tareas. Se sentaron las bases para la arquitectura del sistema.
- **Sprints de Desarrollo del Backend y Núcleo de IA (Fase de Construcción I)**: Durante estos ciclos, el foco principal fue el diseño y la implementación de la arquitectura del sistema backend. Esto incluyó el desarrollo de los módulos encargados de la lógica de negocio, la gestión de datos y, crucialmente, la integración inicial de los modelos de IA seleccionados para el procesamiento de texto, imágenes y otros formatos multimedia.
- **Sprints de Desarrollo de la Interfaz y Orquestación (Fase de Construcción II)**: Paralelamente o a continuación, se abordó el desarrollo de la interfaz de usuario (frontend), buscando una experiencia intuitiva para la interacción mediante lenguaje natural. Se implementó un orquestador de tareas para gestionar las diferentes operaciones del buscador (indexación, consulta, recuperación multimodal). Asimismo, se estableció la comunicación entre el frontend y el backend, típicamente a través de una Application Programming Interface (API) REST, para asegurar un flujo de datos coherente.
- **Sprints de Integración Avanzada y Pruebas (Fase de Refinamiento)**: Estos sprints se dedicaron a la integración completa de todos los componentes del sistema, con especial atención a la interacción fluida entre los modelos de IA y el resto de la aplicación. Se llevaron a cabo pruebas de rendimiento para evaluar la eficiencia del buscador bajo grandes cargas de datos y se realizaron pruebas de usabilidad para garantizar que la interfaz cumplía con los requisitos de accesibilidad y facilidad de uso.
- **Sprints Finales (Fase de Consolidación y Documentación)**: Los últimos ciclos de desarrollo se enfocaron en la corrección de errores (bug fixing), la optimización de funcionalidades existentes, la incorporación de mejoras basadas en las pruebas y la retroalimentación recibida. Una parte significativa de este periodo se dedicó también a la elaboración de la documentación técnica del proyecto y la memoria del TFG.

4.1.3. Gestión de Tareas y Adaptabilidad

Para cada sprint, el estudiante elaboró una lista de tareas en sucio (equivalente al *Sprint Backlog*) a partir de los objetivos definidos. El progreso de estas tareas se monitorizó de forma continua, marcando aquellas completadas para mantener un control efectivo del avance y anotando las posibles dudas e inquietudes para comentarlas con el tutor en el siguiente sprint. El proceso de desarrollo seguía un ciclo de ideación (definición de la funcionalidad o mejora) seguido de su implementación y prueba.

Es importante destacar que, en consonancia con los principios ágiles, el plan del proyecto no fue rígido. A medida que se avanzaba, se identificaron nuevos desafíos técnicos, se descubrieron herramientas más adecuadas o surgieron limitaciones imprevistas. Esta realidad condujo a la redefinición de algunas tareas y al ajuste de los objetivos de ciertos sprints, siempre en comunicación con el tutor, para asegurar la viabilidad y la calidad del resultado final. Esta flexibilidad fue fundamental para navegar la complejidad inherente a un proyecto de investigación y desarrollo como LLMSearch.

4.2. Apartado técnico

Para la ejecución y desarrollo del presente TFG, se ha dispuesto del siguiente entorno técnico, tanto a nivel de hardware como de software. Esta configuración ha sido la base sobre la cual se han realizado todas las pruebas, desarrollos y validaciones del sistema propuesto.

4.2.1. Equipamiento Hardware

El equipo informático utilizado para el desarrollo del proyecto cuenta con las siguientes especificaciones:

- **Procesador (Central Processing Unit (CPU)):** AMD Ryzen 9 7900X3D 4.4GHz/5.6GHz
- **Memoria (Random Access Memory (RAM)):** Corsair Vengeance RGB DDR5 6000MHz 64GB 2x32GB CL30
- **Tarjeta Gráfica (Graphics Processing Unit (GPU)):** RTX 4070 Ti SUPER Trinity 16GB GDDR6X
- **Almacenamiento (Solid State Drive (SSD)):** NVMe Samsung 970 EVO Plus de 1TB
- **Sistema Operativo (Operating System (OS)):** Windows 11 Pro / Ubuntu 22.04 LTS

4.2.2. Software y Herramientas de Desarrollo

La selección del software y las herramientas de desarrollo ha sido crucial para garantizar un flujo de trabajo eficiente y productivo. Para el **Integrated Development Environment (IDE)**, se ha optado por **Visual Studio Code (VS Code)**. Esta elección se fundamenta en su ligereza, su amplia gama de extensiones que facilitan el desarrollo en múltiples lenguajes

(especialmente Python, previsiblemente central en un proyecto con LLMs), su depurador integrado, y su excelente integración con sistemas de control de versiones como Git.

Precisamente, para el **control de versiones**, se ha utilizado **Git**, el estándar de facto en la industria, gestionando los repositorios a través de **GitHub**. Esta plataforma no solo permite un seguimiento exhaustivo de los cambios y la experimentación segura mediante ramas, sino que también facilita la colaboración (aunque en este proyecto sea individual, es una buena práctica) y ofrece un respaldo del código en la nube.

Considerando la naturaleza del proyecto, que involucra el uso intensivo de modelos de lenguaje y otras bibliotecas de IA, se ha empleado Python como uno de los lenguajes de programación principales, decisión que se justificará más adelante en el desarollo. Para la **gestión de entornos y paquetes** de Python, se ha utilizado **pip**, el instalador de paquetes estándar de Python. Su simplicidad y eficacia permiten manejar las dependencias del proyecto de manera ordenada, asegurando la reproducibilidad del entorno de desarrollo en diferentes sistemas si fuera necesario.

En cuanto a la validación de la interfaz de usuario, si el proyecto la incluye, las pruebas se realizarán en una selección de navegadores web modernos. Principalmente, se utilizará **Google Chrome**, en su versión más reciente, debido a su amplia cuota de mercado y sus robustas herramientas integradas para desarrolladores, lo que facilita la depuración y asegura una alta compatibilidad con la mayoría de los usuarios. Adicionalmente, se realizarán pruebas en **OperaGX**, también en su última versión. La elección de OperaGX responde, en parte, a que es el navegador principal utilizado por el desarrollador, lo que agiliza las pruebas iterativas y la verificación rápida de cambios durante el ciclo de desarrollo. Aunque ambos navegadores comparten el motor Chromium, permitiendo una base de compatibilidad similar, esta doble comprobación ayuda a identificar posibles particularidades menores y asegura una experiencia de usuario consistente en un entorno familiar para el desarrollador.

Finalmente, para la **documentación** del proyecto, se ha recurrido a **LaTeX**, utilizando la distribución **MiKTeX**. LaTeX es la herramienta por excelencia para la redacción de documentos técnicos y científicos, gracias a su insuperable calidad tipográfica, su manejo eficiente de referencias bibliográficas, y su capacidad para estructurar documentos complejos. Complementariamente, para la creación de diagramas y esquemas visuales, se ha empleado **Excalidraw**, una herramienta online que permite generar diagramas de forma rápida y con un estilo claro y moderno, facilitando la comunicación de ideas y arquitecturas complejas.

5. Análisis, Especificación y Diseño

5.1. Requisitos del sistema

En esta sección se detallan los requisitos del sistema, divididos en requisitos funcionales, no funcionales y de configuración. Los requisitos se han estructurado en formato tabular para facilitar su comprensión y seguimiento durante el desarrollo del proyecto.

5.1.1. Requisitos funcionales

Los requisitos funcionales describen el comportamiento que debe tener el sistema, las funcionalidades que debe ofrecer y las operaciones que debe realizar.

ID	Nombre	Descripción
RF-01	Detección de ficheros	El sistema debe detectar nuevos ficheros en el directorio observado.
RF-02	Diferenciación de tipos	El sistema debe diferenciar el tipo de archivo a analizar (texto, imagen, vídeo, audio, otros).
RF-03	Ejecución de modelos	El sistema debe ejecutar el modelo correspondiente que extraerá la información del fichero a la base de datos.
RF-04	Almacenamiento	El sistema debe almacenar todos los datos posibles sobre el fichero analizado en una base de datos.
RF-05	Interfaz web	El sistema debe tener una interfaz web super-simple donde el usuario podrá escribir su consulta en lenguaje natural y darle a un botón para realizar la búsqueda.
RF-06	Resultados de búsqueda	El sistema responderá con un conjunto de resultados potencialmente interesantes a partir de la consulta de búsqueda, ordenados de más a menos ”interesante”.
RF-07	Entrada por línea de comandos	El sistema debe tener una entrada por línea de comandos (ej: `LLMSearch --query "mapa del mundo en el que hay marcados los mejores parques naturales"`).
RF-08	Presentación de resultados	El resultado será la ruta del fichero junto a una pequeña descripción del mismo (enlaces clicables al fichero y a la carpeta que lo contiene).
RF-09	Inspección de archivos comprimidos	Los ficheros comprimidos deberían poder inspeccionarse por dentro.
RF-10	Tipos de ficheros a procesar	El sistema debe procesar los siguientes tipos de ficheros: - Documentos de texto - Imágenes - Vídeos - Ficheros de sonido - Otros (bases de datos, ejecutables, etc.)

Tabla 5.1: Requisitos funcionales del sistema

5.1.2. Requisitos no funcionales

Los requisitos no funcionales especifican criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos.

ID	Nombre	Descripción
RNF-01	Configuración web	La web debe tener una pequeña parte de configuración discreta pero accesible en todo momento.
RNF-02	Arquitectura modular	La arquitectura se debe dividir en un "buscador" y un "explorador" y deben ser completamente separadas para poder ser reutilizadas.
RNF-03	Ejecución sin GPU	El sistema debe poder ejecutarse en un ordenador sin GPU (opcional).
RNF-04	Parámetro de consulta	La entrada por línea de comandos aceptará un parámetro `--query` junto al término de búsqueda.
RNF-05	Resultados en CLI	La entrada por línea de comandos devolverá los resultados de la misma manera que el buscador web con la diferencia de que solo devolverá información adicional si se le añade el parámetro `--verbose`.
RNF-06	Estado del sistema	La entrada por línea de comandos tendrá un parámetro `--status` que devolverá el estado del sistema: número de archivos procesados sobre el número total de archivos en observación, cantidad de ficheros de cada tipo, errores encontrados...
RNF-07	Configuración por CLI	Se añadirán los parámetros necesarios para poder configurar el sistema desde línea de comandos.

Tabla 5.2: Requisitos no funcionales del sistema

5.1.3. Requisitos de configuración

Los requisitos de configuración especifican las opciones que el usuario debe poder ajustar en el sistema.

ID	Nombre	Descripción
RC-01	Directorio de observación	Directorio donde se están observando nuevos ficheros.
RC-02	Regulación de carga	Regular la carga (limitar la CPU al X%).
RC-03	Tipo de modelo LLM	Tipo de modelo LLM a utilizar (Local *(LLM Studio)* ó en la nube).
RC-04	Búsqueda por imagen	Posibilidad de poner una foto de una persona y que la busque en los ficheros.

Tabla 5.3: Requisitos de configuración del sistema

5.2. Arquitectura del Sistema

La arquitectura de LLMSearch se ha concebido como un sistema modular y distribuido, con el objetivo de facilitar la escalabilidad, el mantenimiento y la posible reutilización de

componentes. Un esquema visual inicial de esta arquitectura se presenta en la Figura 5.1.

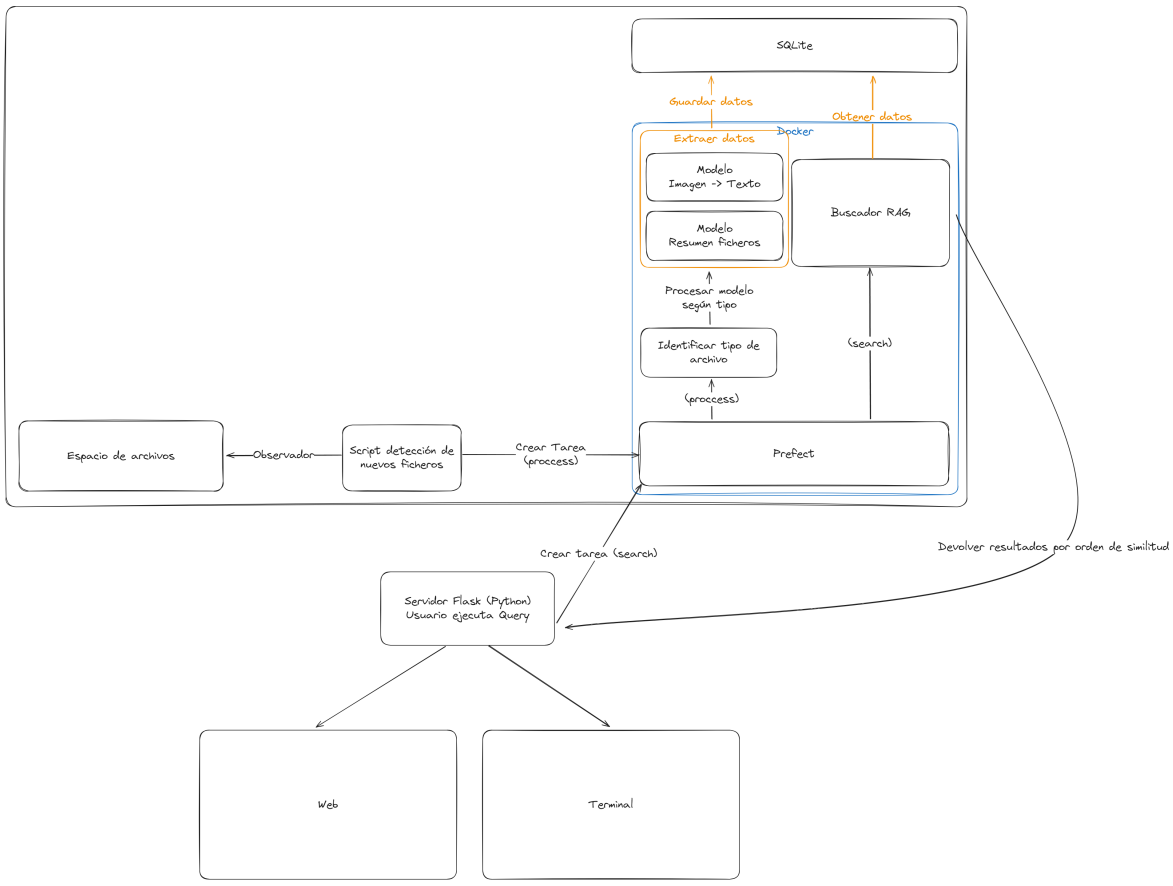

Figura 5.1: Diagrama de la arquitectura general de LLMSearch.

Es importante señalar que el diagrama de la Figura 5.1 representa una instantánea conceptual de las primeras etapas del diseño del proyecto. Si bien captura la esencia de la modularidad y los flujos principales, ciertos componentes y sus interacciones han sido refinados o modificados durante el proceso de desarrollo para optimizar el rendimiento, la simplicidad o la adecuación a las herramientas finalmente seleccionadas. Por ejemplo, la especificación y el tipo de la base de datos han evolucionado desde la concepción inicial. Los siguientes apartados describen en detalle cada componente en su estado final de implementación, destacando las decisiones de diseño clave y cualquier desviación significativa respecto al esquema preliminar.

5.2.1. Componentes Principales de la Arquitectura

- **Script Observador**: Este componente es el encargado de monitorizar de forma continua el directorio o directorios especificados por el usuario (según RC-01) en busca de nuevos ficheros o modificaciones en los existentes (RF-01). Cuando detecta un cambio relevante, el observador notifica al servidor para iniciar el proceso de análisis del fichero.
- **Servidor Central (Backend API)**: El núcleo del sistema reside en una aplicación

servidor que actúa como punto central de comunicación y control. Sus responsabilidades principales son:

1. Inicializar y gestionar el ciclo de vida del `Script Observador`.
2. Recibir notificaciones del observador sobre nuevos ficheros o ficheros modificados.
3. Exponer una API RESTful para atender las peticiones provenientes tanto de la interfaz web (RF-05) como de la interfaz de línea de comandos (Command Line Interface (CLI)) (RF-07). Estas peticiones incluyen las consultas de búsqueda de los usuarios y, potencialmente, comandos de gestión y configuración del sistema (RNF-07).
4. Interactuar con el Orquestador de Tareas para delegar el procesamiento de ficheros y la ejecución de búsquedas.

- **Orquestador de Tareas**: Dada la naturaleza asíncrona y potencialmente intensiva en recursos del procesamiento de ficheros y las consultas a LLMs, se ha decidido incorporar un sistema de orquestación de tareas. Este sistema se encarga de la gestión de flujos de trabajo, permitiendo encolar tareas, ejecutarlas (posiblemente en procesos separados o workers), monitorizar su estado y gestionar reintentos o fallos. El servidor central enviará solicitudes de creación de tareas a este orquestador. Las tareas principales gestionadas por el orquestador serán:
  1. **Tarea de Procesamiento**: Al recibir la ruta de un nuevo fichero, esta tarea coordinará varias subtareas:
     - Invocará al `Identificador del Tipo de Fichero` para determinar la naturaleza del archivo (RF-02).
     - En función del tipo, seleccionará y ejecutará el `Modelo de Extracción de Datos` correspondiente (RF-03).
     - Paralelamente, se extraerán metadatos generales del fichero (nombre, tamaño, fechas, etc.).
     - Finalmente, todos los datos extraídos (contenido semántico, metadatos) se persistirán en la `Base de Datos` (RF-04).
  2. **Tarea de Búsqueda**: Cuando el usuario realiza una consulta, esta tarea:
     - Recibirá la consulta en lenguaje natural del usuario como parámetro.
     - Realizará una primera fase de recuperación de información relevante de la `Base de Datos`.
     - Construirá un prompt optimizado, incorporando la información recuperada y la consulta original, para ser procesado por el `Buscador RAG` (utilizando un LLM).
     - Devolverá un conjunto de resultados ordenados por relevancia o similitud con la consulta (RF-06).
- **Identificador del Tipo de Fichero**: Para asegurar una correcta clasificación de los ficheros (RF-02) y evitar depender únicamente de la extensión (que puede ser engañosa o incorrecta), este módulo analizará las cabeceras o "números mágicos" de los ficheros

para determinar su formato real. Se emplearán mecanismos especializados para una identificación robusta de una amplia variedad de tipos de archivo.

- **Modelos de Extracción de Datos**: Este es un conjunto de módulos especializados, cada uno diseñado para procesar un tipo específico de fichero (texto, imagen, vídeo, audio, etc., según RF-10). La arquitectura de estos modelos será eminentemente modular, permitiendo la fácil incorporación de nuevos extractores para formatos de archivo futuros o la actualización de los existentes. Cada modelo será responsable de invocar las herramientas de IA o librerías pertinentes (ej. OCR para imágenes, ASR para audio, análisis semántico para texto) para extraer la información significativa y estructurarla.
- **Base de Datos**: Para el almacenamiento persistente de la información extraída de los ficheros y los metadatos asociados (RF-04), se utilizará un sistema de base de datos. La elección de este sistema basado en criterios de simplicidad, facilidad de configuración local (preferiblemente embebida, sin requerir un servidor de base de datos separado) y buena integración con el lenguaje de desarrollo principal. Se desarrollará una capa de acceso a datos para interactuar con la base de datos de manera estructurada y segura.
- **Buscador RAG**: El componente central para la búsqueda semántica (RF-06) se basará en la técnica de RAG. Este enfoque combina la recuperación eficiente de información de la `Base de Datos` con las capacidades de comprensión y generación de lenguaje natural de un LLM. Durante el análisis, se han considerado varias estrategias para implementar el RAG:
  - *Generación de Consultas SQL*: El LLM podría generar consultas Structured Query Language (SQL) para interrogar directamente la base de datos. Si bien es una opción, presenta el riesgo de generar consultas subóptimas o incorrectas, y podría limitar la expresividad semántica si el LLM no comprende bien el esquema.
  - *Incorporación de Texto Completo al Prompt*: Convertir fragmentos relevantes de la base de datos a texto e incluirlos directamente en el prompt del LLM. El principal desafío aquí es la limitación de la ventana de contexto de muchos LLMs. Aunque modelos recientes (como algunos modelos chinos con ventanas de contexto muy amplias) podrían mitigar esto, requiere una filtración previa muy efectiva de la información de la base de datos para no exceder los límites o incurrir en altos costos computacionales.
  - *Uso de Embeddings para Contexto*: Transformar los datos recuperados y la consulta del usuario en embeddings (representaciones vectoriales) y utilizar estos embeddings para enriquecer el contexto del LLM. Esta suele ser la aproximación más eficiente y robusta, aunque puede implicar una mayor complejidad en la implementación de la infraestructura de embeddings y la gestión de la similitud vectorial.

  La elección final de la estrategia RAG o una combinación de ellas dependerá de la experimentación y la evaluación del rendimiento y la complejidad.

5.2.2. Consideraciones sobre Contenerización

Durante la fase de análisis, se evaluó la posibilidad de utilizar tecnologías de **contenerización** para los diferentes servicios del sistema. La contenerización ofrece ventajas significativas

en términos de reproducibilidad del entorno, aislamiento de dependencias y simplificación del despliegue. Sin embargo, para la etapa actual del proyecto, y dado que el objetivo principal es desarrollar y validar la funcionalidad central en un entorno local, se ha optado por no implementar una solución de contenerización inicialmente. La gestión de dependencias a través de entornos virtuales específicos del lenguaje de programación y la configuración directa de los servicios en el sistema operativo local se considera suficiente y más ágil para el desarrollo iterativo. No obstante, la arquitectura modular propuesta facilitaría una futura migración a una infraestructura basada en contenerización si el proyecto escalara o se requiriera un despliegue en entornos más complejos.

5.3. Casos de uso

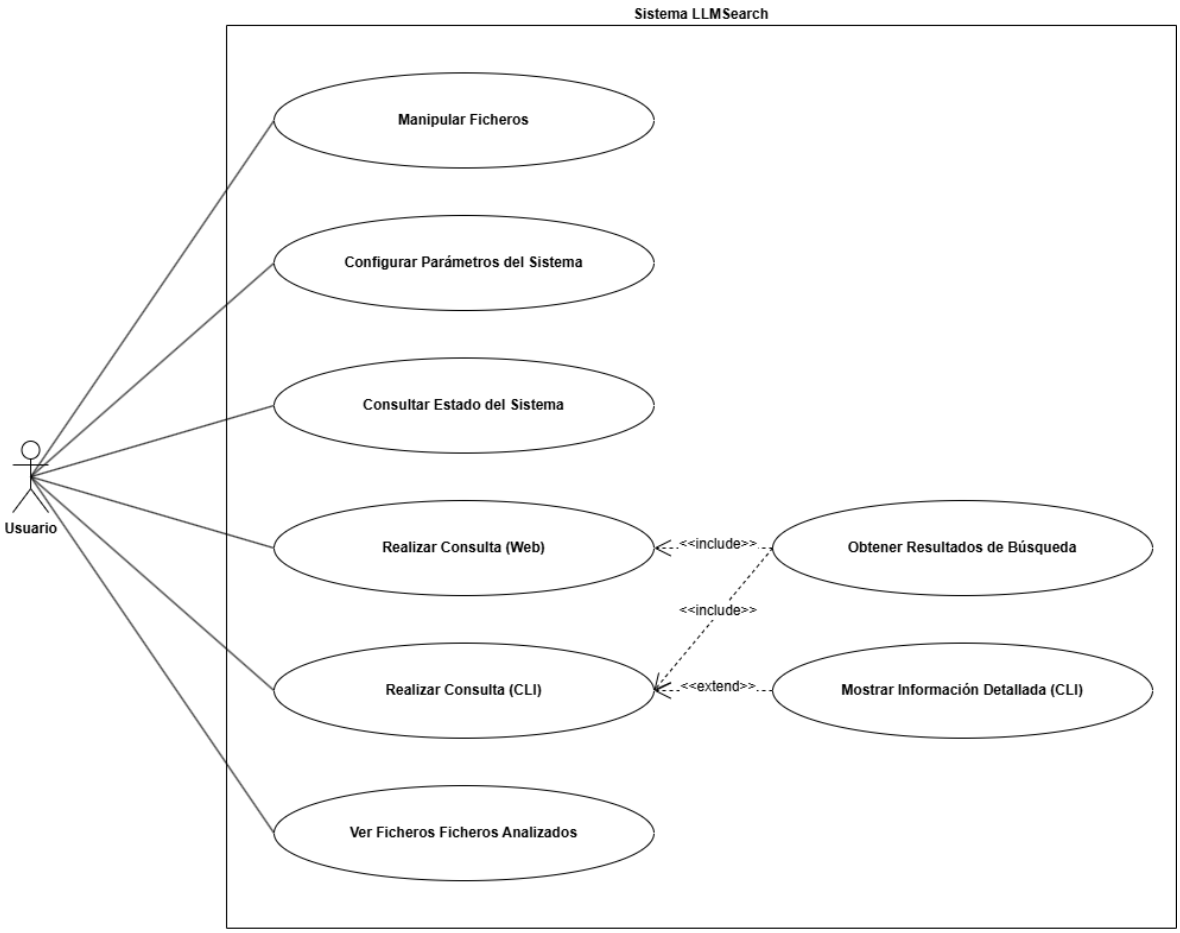

Figura 5.2: Diagrama de casos de uso del Sistema LLMSearch.

A continuación, se describen los casos de uso identificados para el Sistema LLMSearch, tal como se representan en el diagrama de casos de uso (Figura 5.2). Estos definen las interacciones clave entre el actor 'Usuario' y las funcionalidades del sistema.

5.3.1. Manipular Ficheros

- **Actor Principal:** Usuario
- **Descripción:** El usuario interactúa con el sistema para gestionar los ficheros. Esto incluye incluir la adición, modificación y eliminación de ficheros.

5.3.2. Configurar Parámetros del Sistema

- **Actor Principal:** Usuario
- **Descripción:** El usuario ajusta y personaliza los diversos parámetros de configuración del Sistema LLMSearch. Esto puede incluir la definición del directorio de observación, la regulación de la carga del sistema (límites de CPU), la selección del tipo de modelo LLM a utilizar (local o en la nube), y otras opciones que afectan el comportamiento general del sistema.

5.3.3. Consultar Estado del Sistema

- **Actor Principal:** Usuario
- **Descripción:** El usuario solicita y visualiza información sobre el estado operativo actual del Sistema LLMSearch. Esta información puede comprender estadísticas como el número total de archivos procesados, el desglose de ficheros por tipo, la cantidad de archivos pendientes de análisis, y la notificación de posibles errores ocurridos durante el procesamiento.

5.3.4. Realizar Consulta (Web)

- **Actor Principal:** Usuario
- **Descripción:** El usuario introduce una consulta de búsqueda en lenguaje natural a través de la interfaz web proporcionada por el Sistema LLMSearch. El objetivo es encontrar ficheros o información relevante almacenada y procesada por el sistema.

5.3.5. Realizar Consulta (CLI)

- **Actor Principal:** Usuario
- **Descripción:** El usuario introduce una consulta de búsqueda utilizando la interfaz de línea de comandos (CLI) del Sistema LLMSearch. Esta modalidad permite interactuar con el sistema para encontrar ficheros relevantes sin necesidad de una interfaz gráfica.

5.3.6. Obtener Resultados de Búsqueda

- **Rol:** Caso de uso incluido.

- **Descripción:** El sistema procesa la consulta de búsqueda proporcionada (ya sea desde la interfaz web o la CLI), realiza la búsqueda en su base de datos de información extraída de los ficheros analizados, y devuelve un conjunto de resultados. Estos resultados suelen estar ordenados por relevancia o similitud con la consulta original. Este caso de uso no es iniciado directamente por un actor externo, sino que representa una funcionalidad interna reutilizada.
- **Relaciones:**
  - Es incluido por "Realizar Consulta (Web)".
  - Es incluido por "Realizar Consulta (CLI)".

5.3.7. Mostrar Información Detallada (CLI)

- **Rol:** Caso de uso de extensión.
- **Descripción:** Proporciona al usuario información adicional y más detallada sobre los resultados de búsqueda específicos cuando estos han sido obtenidos a través de la interfaz de línea de comandos (CLI). Esta funcionalidad es opcional y se activa bajo ciertas condiciones o por la petición explícita del usuario mediante el parámetro adicional `--verbose` durante la ejecución del caso de uso "Realizar Consulta (CLI)".
- **Relaciones:**
  - Extiende el caso de uso "Realizar Consulta (CLI)".

5.3.8. Ver Ficheros Analizados

- **Actor Principal:** Usuario
- **Descripción:** El usuario accede a una sección dedicada en la interfaz web para explorar los ficheros que han sido procesados y analizados por el Sistema LLMSearch. En esta sección, el sistema presenta una lista de dichos ficheros. Para cada uno, el usuario puede visualizar:
  - Su descripción (generada por el sistema o extraída del contenido).
  - Metadatos relevantes (por ejemplo, tipo de fichero, tamaño, fecha de creación/-modificación, etiquetas generadas, etc.).
  - Una previsualización o reproductor embebido si el fichero es una imagen, vídeo o archivo de audio, permitiendo su visualización o reproducción directa dentro de la interfaz web.

  Esta funcionalidad permite al usuario inspeccionar el corpus de datos indexado, verificar los detalles del análisis de cada fichero y acceder directamente a su contenido visual o auditivo y a sus metadatos asociados sin necesidad de descargar el fichero original o abrir aplicaciones externas.

6. Desarrollo

La construcción de un sistema inteligente para la búsqueda y gestión de archivos personales requiere la integración de diversas tecnologías y herramientas consolidadas en el ámbito del desarrollo de software y la inteligencia artificial. Este capítulo tiene como objetivo, en una primera parte, revisar el estado del arte de los componentes tecnológicos clave que se han considerado para la implementación del presente proyecto. Posteriormente, en una segunda parte, se detallarán las decisiones de diseño finales para cada componente, justificando la elección, describiendo aspectos relevantes de su implementación y los desafíos encontrados durante el desarrollo.

6.1. Estudio de Tecnologías

En esta sección se analizarán diferentes opciones en áreas fundamentales como la orquestación de tareas, la detección de cambios en el sistema de archivos, las soluciones de bases de datos para el almacenamiento de metadatos y embeddings, la contenerización para el despliegue y, finalmente, los frameworks para el desarrollo de la interfaz de usuario.

6.1.1. Orquestadores de tareas

La gestión eficiente de flujos de trabajo complejos, especialmente aquellos que involucran procesamiento de datos y tareas de machine learning, es crucial para el sistema propuesto. Un orquestador de tareas permite automatizar, programar y monitorizar estas secuencias de operaciones.

6.1.1.1. Prefect

Prefect se presenta como una moderna plataforma de orquestación de flujos de trabajo, escrita principalmente en Python. Está diseñada específicamente para permitir a los desarrolladores diseñar, programar, ejecutar y monitorizar pipelines de datos y flujos de machine learning de manera fiable y escalable, con un enfoque en la simplicidad y la experiencia del desarrollador.

6.1.1.1.1. Ventajas

- **Facilidad de uso:** Prefect ofrece una sintaxis intuitiva y una configuración sencilla, lo que facilita la definición y gestión de flujos de trabajo complejos.
- **Flexibilidad:** Permite la orquestación de tareas en entornos locales, en la nube o híbridos, adaptándose a diversas necesidades.
- **Monitoreo y gestión:** Incluye herramientas integradas para el monitoreo, registro y manejo de errores en tiempo real.

6.1.1.1.2. Desventajas

- **Madurez:** Aunque ha ganado popularidad, Prefect es relativamente nuevo en comparación con otras herramientas más consolidadas.
- **Comunidad:** Su comunidad es más pequeña, lo que puede limitar la disponibilidad de recursos y soporte.

6.1.1.2. Kafka

Apache Kafka es un sistema de mensajería distribuido de código abierto, reconocido por su alto rendimiento y capacidad para manejar flujos de datos en tiempo real. Aunque su función principal es la de broker de mensajes, a menudo se utiliza en arquitecturas complejas para desacoplar sistemas y como parte de pipelines de datos más amplios, pudiendo actuar como un componente en la orquestación de eventos.

6.1.1.2.1. Ventajas

- **Alto rendimiento:** Kafka es conocido por su capacidad para manejar grandes volúmenes de datos con baja latencia.
- **Escalabilidad:** Diseñado para escalar horizontalmente, puede manejar cargas de trabajo crecientes de manera eficiente.
- **Ecosistema robusto:** Cuenta con una amplia gama de herramientas y conectores que facilitan su integración con otros sistemas.

6.1.1.2.2. Desventajas

- **Complejidad:** La configuración y gestión de Kafka pueden ser complejas, especialmente para usuarios sin experiencia previa.
- **Requisitos de recursos:** Para un rendimiento óptimo, Kafka suele requerir una infraestructura robusta, lo que puede ser excesivo para proyectos más pequeños.

6.1.1.3. Airflow

Apache Airflow es una plataforma de código abierto ampliamente adoptada para la creación, programación y monitorización programática de flujos de trabajo. Originalmente desarrollada por Airbnb, permite definir flujos de trabajo como Grafos Acíclicos Dirigidos (DAGs) de tareas, utilizando Python para su definición.

6.1.1.3.1. Ventajas

- **Popularidad y comunidad:** Amplia adopción y una comunidad activa que proporciona numerosos recursos y soporte.
- **Flexibilidad:** Permite la programación y monitoreo de flujos de trabajo complejos.

6.1.1.3.2. Desventajas

- **Curva de aprendizaje:** Puede ser complejo de configurar y requiere conocimientos avanzados para su implementación efectiva.

6.1.2. Detección de cambios en el sistema de archivos

Un componente esencial del sistema es la capacidad de detectar automáticamente la creación, modificación o eliminación de archivos. Esta funcionalidad desencadena el proceso de análisis.

6.1.2.1. Python

Python, debido a su versatilidad y extenso ecosistema de bibliotecas, ofrece múltiples opciones.

- **Watchdogs:** Biblioteca multiplataforma para observar eventos del sistema de archivos.
- **pyinotify:** Wrapper de Python para la API inotify de Linux (no portable).
- **inotify-simple:** Wrapper más sencillo para inotify de Linux.
- **inotifyx:** Similar a pyinotify, para inotify de Linux.
- **Polling Methods:** Verificación periódica, menos eficiente.

6.1.2.2. Node.js

- **chokidar:** Biblioteca popular y eficiente para Node.js, multiplataforma.

6.1.2.3. Java

- **WatchService:** API integrada en Java (New Input/Output (versión 2) (NIO.2)) para monitoreo.

6.1.2.4. C++/C/C#

- **FileSystemWatcher:** En .NET (C#). Para C/C++, APIs específicas del SO (inotify en Linux, ReadDirectoryChangesW en Windows).

6.1.2.5. Go

- **fsnotify:** Biblioteca popular en Go, interfaz común sobre APIs específicas.

6.1.2.6. Rust

- **notify:** Biblioteca de Rust multiplataforma.

6.1.3. Bases de datos

El almacenamiento persistente de metadatos y embeddings es fundamental.

6.1.3.1. Relacional

Adecuadas para datos estructurados y consistencia Atomicity, Consistency, Isolation, Durability (ACID).

6.1.3.1.1. SQLite Autocontenida, sin servidor, transaccional. Almacena la base de datos en un único archivo.

Ventajas

- **Ligereza y simplicidad.**
- **Portabilidad.**
- **Rendimiento en entornos de bajo recurso.**

Desventajas

- **Concurrencia limitada en escrituras.**
- **Escalabilidad limitada.**

6.1.3.1.2. MariaDB Fork de MySQL, de código abierto.

Ventajas

- **Rendimiento y escalabilidad.**
- **Compatibilidad con MySQL.**
- **Soporte para almacenamiento en columnas.**

Desventajas

- **Complejidad en la configuración.**
- **Requisitos de recursos.**

6.1.3.2. No relacional (NoSQL)

Modelos de datos flexibles, escalabilidad horizontal.

6.1.3.2.1. MongoDB Orientada a documentos (Binary JSON (BSON)).

Ventajas

- **Flexibilidad del esquema.**
- **Escalabilidad horizontal.**
- **Alto rendimiento en lectura/escritura.**

Desventajas

- **Consumo de recursos.**
- **Soporte limitado para transacciones complejas (tradicionales).**

6.1.3.2.2. ChromaDB Base de datos vectorial de código abierto para aplicaciones de IA.

Ventajas

- **Especializada en embeddings.**
- **Facilidad de uso y API intuitiva (Python).**
- **Integraciones con ecosistema de IA (LangChain, LlamaIndex).**
- **Ligera y embebible.**
- **Código abierto.**

Desventajas

- **Madurez y escalabilidad para producción masiva (en comparación).**
- **Funcionalidades de BD tradicional limitadas.**
- **Operaciones y gestión avanzada (para gran escala).**

6.1.4. Contenerización

La contenerización garantiza consistencia entre entornos. Docker es la plataforma líder.

6.1.4.1. Docker

Plataforma para automatizar el despliegue de aplicaciones en contenedores.

6.1.4.1.1. Ventajas

- **Portabilidad.**
- **Aislamiento.**
- **Facilidad de despliegue.**

6.1.4.1.2. Desventajas

- **Consumo de recursos (capa adicional).**
- **Complejidad adicional (gestión de contenedores).**

6.1.5. Frameworks de Interfaz de Usuario

La elección del framework impacta la experiencia del usuario y el desarrollo.

6.1.5.1. Angular

Framework de Google basado en TypeScript, completo y opinado.

6.1.5.1.1. Ventajas

- **Framework completo.**
- **Arquitectura estructurada.**

6.1.5.1.2. Desventajas

- **Curva de aprendizaje pronunciada.**
- **Complejidad innecesaria para proyectos simples.**

6.1.5.2. React

Biblioteca de JavaScript de Meta para construir UIs.

6.1.5.2.1. Ventajas

- **Biblioteca flexible.**
- **Amplia comunidad y recursos.**

6.1.5.2.2. Desventajas

- **Necesidad de configuraciones adicionales (para routing, estado global).**

6.1.5.3. Vue.js

Framework de JavaScript progresivo y accesible.

6.1.5.3.1. Ventajas

- **Simplicidad y facilidad de uso.**
- **Flexibilidad.**

6.1.5.3.2. Desventajas

- **Menor adopción en grandes empresas (en comparación).**

6.1.5.4. Astro

Framework web moderno para sitios rápidos y centrados en contenido (arquitectura de "islas").

6.1.5.4.1. Ventajas

- **Optimización para contenido estático y rendimiento.**
- **Integración con otros frameworks.**

6.1.5.4.2. Desventajas

- **Menor madurez para aplicaciones altamente interactivas.**
- **Ecosistema en crecimiento.**

6.2. Decisiones de Diseño e Implementación

Tras el estudio de las tecnologías disponibles, en esta sección se detallan las herramientas finalmente seleccionadas para cada componente del sistema, justificando la elección, describiendo los aspectos más relevantes de su implementación y los problemas o consideraciones que surgieron durante el proceso de desarrollo.

6.2.1. Orquestador de Tareas: Prefect

6.2.1.0.1. Decisión y Justificación Para la orquestación de las tareas de procesamiento de archivos, extracción de metadatos, generación de embeddings y su posterior almacenamiento, se ha seleccionado **Prefect**. La elección se fundamenta en su enfoque moderno, su facilidad de uso al estar escrito en Python, lenguaje principal del proyecto, y su adecuada capacidad para gestionar pipelines de datos y de Machine Learning (ML). Aunque herramientas como Kafka ofrecen un rendimiento superior para flujos de datos masivos y Airflow cuenta con una comunidad más extensa, Prefect proporciona un equilibrio óptimo entre simplicidad, flexibilidad y potencia para las necesidades específicas de este proyecto. Su curva de aprendizaje es más accesible en comparación con Airflow, y su infraestructura requerida es menos exigente que la de Kafka, haciéndolo idóneo para un proyecto de esta envergadura.

6.2.1.0.2. Implementación La implementación con Prefect se estructura en torno a *Tasks* (tareas individuales) y *Flows* (flujos de trabajo que orquestan las tareas).

Las principales **Tasks** definidas son:

- `summarize_text`: Resume el texto proporcionado como parámetro.
- `analyze_image`: Analiza el contenido de una imagen utilizando un modelo multimodal (en este caso, Gemma).
- `get_image_metadata`: Extrae metadatos específicos de archivos de imagen.
- `rag_query`: Procesa una consulta del usuario (*query*) utilizando un modelo de lenguaje grande (LLM), en este caso Mistral, enriqueciendo la consulta con resultados de búsqueda vectorial obtenidos de ChromaDB para generar una respuesta contextualizada.
- `rag_query_with_db`: Realiza una búsqueda de similitud en ChromaDB basada en la consulta del usuario, devolviendo un número máximo especificado de coincidencias.

Estos *tasks* se orquestan en los siguientes **Flows**:

- `new_file`: Se activa al detectar un nuevo archivo en la carpeta monitorizada. Este flujo gestiona la detección de duplicados, la identificación del tipo de archivo, la extracción de metadatos, la generación de embeddings y el almacenamiento de los resultados en ChromaDB.
- `modified_file`: Opera de manera similar a `new_file`, pero se desencadena cuando un archivo existente es modificado. En este caso, se actualizan los metadatos y los embeddings en ChromaDB si el hash del contenido del archivo ha cambiado, indicando una modificación sustancial.

- `deleted_file`: Se activa tras la eliminación de un archivo. Procede a eliminar el documento correspondiente y sus metadatos asociados de ChromaDB.
- `process_query`: Se inicia cuando el usuario realiza una búsqueda a través de la interfaz. Genera los embeddings de la consulta, los envía a ChromaDB para encontrar coincidencias semánticas, y los resultados se proporcionan a un LLM para generar una respuesta elaborada.

Si bien Prefect ofrece capacidades robustas para la ejecución paralela de tareas y flujos, en la implementación actual, el grado de paralelización se ve limitado por los recursos computacionales disponibles en un entorno de desarrollo local. Tareas intensivas como la generación de embeddings o las inferencias de modelos LLM pueden ser costosas. En un entorno de producción con un servidor adecuadamente dimensionado (con mayor capacidad de CPU, GPU y RAM), se podría explotar de manera mucho más efectiva la paralelización para procesar múltiples archivos o consultas simultáneamente, mejorando significativamente el rendimiento y la capacidad de respuesta del sistema.

La Figura 6.1 muestra una vista general del dashboard de Prefect, donde se pueden monitorizar los flujos y tareas. La Figura 6.2 ilustra un ejemplo de la visualización de un flujo específico en ejecución.

6.2.2. Detección de Cambios: Python con Watchdogs

6.2.2.0.1. Decisión y Justificación La detección de cambios en el sistema de archivos se ha implementado utilizando **Python** en combinación con la biblioteca **watchdogs**. Esta elección se basa en la naturaleza multiplataforma de `watchdogs`, crucial para una aplicación destinada a la gestión de archivos personales que podría ejecutarse en diversos sistemas operativos. Python, como lenguaje principal del proyecto, facilita la integración de este componente con el resto del sistema, especialmente con el orquestador de tareas Prefect.

6.2.2.0.2. Implementación Se ha desarrollado un script de Python que utiliza `watchdogs` para monitorizar de forma recursiva un directorio específico proporcionado por el usuario, centrándose exclusivamente en archivos. El script implementa un manejador de eventos personalizado, subclase de `FileSystemEventHandler`, que reacciona a los eventos de creación (`on_created`), modificación (`on_modified`) y eliminación (`on_deleted`) de archivos. Al detectar un evento relevante, el script desencadena el flujo de Prefect correspondiente. Adicionalmente, al iniciar el programa, se realiza un análisis exhaustivo inicial de toda la carpeta asignada; durante este proceso, los archivos duplicados o aquellos ya procesados y sin cambios significativos se gestionarán eficientemente gracias al sistema de detección de duplicados basado en hashes, evitando reprocesamientos innecesarios.

Un desafío particular surgió con la detección de archivos modificados en tiempo real, especialmente en sistemas Windows. Este sistema operativo tiende a realizar pequeñas modificaciones en los metadatos de los archivos al abrirlos o copiarlos, lo que provocaba activaciones múltiples y no deseadas del evento `on_modified` para un mismo archivo en cortos periodos. Para mitigar este comportamiento, se implementó una caché en memoria que almacena temporalmente información sobre los archivos recientemente procesados por eventos en tiempo real. Esta caché ayuda a prevenir la reactivación innecesaria de flujos para eventos de modificación que no representan cambios sustanciales en el contenido, optimizando el rendimiento.

Figura 6.1: Dashboard principal de Prefect para la monitorización de flujos.

Figura 6.2: Visualización de un flujo de trabajo específico en Prefect.

No obstante, la verificación definitiva de duplicados o cambios significativos se realiza mediante el cálculo y comparación de hashes de archivo en los flujos de Prefect, tanto para el escaneo inicial como para los eventos posteriores.

Es importante destacar que este script de detección de cambios se ejecuta en un hilo separado para no bloquear la operatividad del resto del sistema. Además, se ha diseñado para ignorar eventos relacionados con directorios en su creación o modificación, procesando únicamente archivos. En el caso de la eliminación (`on_deleted`), dado que el objeto del sistema de archivos ya no existe en el momento de la notificación, no se realiza una comprobación para distinguir entre archivo y directorio, asumiendo que el flujo de Prefect manejará adecuadamente la solicitud de eliminación en la base de datos si el ID (basado en la ruta) existiera.

6.2.3. Base de Datos: ChromaDB

6.2.3.0.1. Decisión y Justificación Para el almacenamiento de metadatos y, de forma crucial, los embeddings vectoriales generados por los modelos de IA, se ha optado por **ChromaDB**. Inicialmente, se consideró SQLite por su simplicidad para el almacenamiento de metadatos básicos, y de hecho, se desarrolló un controlador para esta base de datos que permanece disponible en el código base como alternativa o complemento futuro. Sin embargo, la funcionalidad central del sistema reside en la capacidad de realizar búsquedas semánticas eficientes basadas en embeddings, lo que hizo de una base de datos vectorial la elección más adecuada.

ChromaDB fue seleccionada por su especialización en el manejo de embeddings, su facilidad de uso a través de su API Python y su capacidad para operar de forma ligera y embebible, características ideales para el desarrollo y despliegue de este proyecto.

6.2.3.0.2. Implementación ChromaDB se utiliza para almacenar dos tipos principales de información por cada archivo procesado, organizados en una ”colección”:

- **Embeddings:** Vectores numéricos densos que representan el contenido semántico del archivo, permitiendo búsquedas por similitud.
- **Metadatos:** ChromaDB permite asociar un diccionario de metadatos a cada embedding. En este proyecto, se almacenan, como mínimo, los siguientes campos, aunque cada tipo de archivo puede añadir metadatos específicos adicionales:
  - `path`: La ruta absoluta original del archivo en el sistema de archivos.
  - `filename`: El nombre del archivo con su extensión.
  - `size`: El tamaño del archivo en bytes.
  - `creation_time`: La fecha y hora de creación del archivo.
  - `hash`: Un hash SHA256 del contenido del archivo. Este metadato es crucial para evitar el procesamiento y almacenamiento duplicado de archivos idénticos, incluso si tienen nombres o ubicaciones diferentes. Antes de procesar un nuevo archivo, se calcula su hash y se consulta en ChromaDB si ya existe un embedding con ese mismo `hash` en sus metadatos.

Las búsquedas semánticas se realizan enviando un vector de embedding (generado a partir de la consulta del usuario) a ChromaDB, que devuelve los ‘k‘ embeddings más similares junto con sus metadatos asociados. Actualmente, el valor de ‘k‘ se ha limitado a 5 resultados por consulta. Esta restricción se debe principalmente a las limitaciones de la ventana de contexto de los modelos de LLM que se ejecutan localmente, ya que un mayor número de resultados (y por ende, más texto para procesar) podría exceder dicha ventana o degradar significativamente el rendimiento. Esta limitación podría mitigarse en el futuro con el uso de modelos de LLM más avanzados que soporten ventanas de contexto mayores o mediante el acceso a recursos computacionales más potentes.

El controlador de ChromaDB implementado se encarga de gestionar la conexión, la creación de colecciones si no existen, y las operaciones CRUD (inserción, lectura, actualización y eliminación) de embeddings y metadatos. Incluye funciones para verificar si un archivo ya ha sido procesado (basándose en su hash) y para actualizar los datos si se detectan modificaciones.

Una decisión de diseño importante fue utilizar el campo de metadatos de ChromaDB para almacenar toda la información descriptiva del archivo, incluyendo el hash. Esto evita la necesidad de una base de datos relacional adicional (como SQLite) para la gestión de metadatos y la detección de duplicados, simplificando la arquitectura y aprovechando la eficiencia de ChromaDB para consultas basadas en estos metadatos.

6.2.4. Contenerización: No implementada (Docker)

6.2.4.0.1. Decisión y Justificación Durante la fase de análisis, se evaluó la posibilidad de utilizar tecnologías de contenerización como **Docker**. Se reconocen plenamente sus ventajas en términos de reproducibilidad del entorno, aislamiento de dependencias y simplificación del despliegue en diferentes máquinas.

Sin embargo, para la etapa actual del proyecto, y como se anticipó en el análisis inicial, se ha optado por no implementar una solución de contenerización. Esta decisión se fundamenta en varios factores prácticos:

1. El orquestador Prefect, en su modo de ejecución local, se instala fácilmente como un paquete Python y no requiere una configuración de servidor compleja para este proyecto.

2. LMStudio, la herramienta seleccionada para la gestión y ejecución de los modelos de IA locales, es una aplicación de escritorio con un instalador directo, diseñada para ser utilizada en el sistema operativo anfitrión.

3. La gestión de dependencias de Python se ha manejado eficazmente mediante el uso de entornos virtuales (‘venv‘), asegurando un aislamiento adecuado a nivel de proyecto.

Dado que el objetivo principal era validar la funcionalidad central del sistema en un entorno de desarrollo local, y los componentes clave no presentaban complejidades de entorno que justificaran la sobrecarga administrativa de Docker en esta fase, se consideró más ágil proceder sin él.

6.2.4.0.2. Consideraciones Futuras La arquitectura modular del sistema, con componentes bien definidos (backend API, motor de Prefect, detector de cambios), está diseñada para facilitar una futura migración a contenedores. Si el proyecto evolucionara hacia un despliegue

en entornos más complejos, multiusuario o en la nube, la adopción de Docker (posiblemente junto con Docker Compose) sería un paso lógico y altamente recomendable. Esto permitiría empaquetar el servidor de la API, el agente de Prefect, la base de datos (especialmente si se optara por una versión servida de ChromaDB o una alternativa) y la interfaz de usuario en contenedores separados y orquestados, mejorando la escalabilidad y la mantenibilidad.

6.2.5. Interfaz de Usuario: Vue.js

6.2.5.0.1. Decisión y Justificación Para el desarrollo de la interfaz de usuario (User Interface (UI)), se ha seleccionado **Vue.js**. La principal razón detrás de esta elección fue la búsqueda de simplicidad y una curva de aprendizaje accesible, dado que la UI, aunque importante para la interacción del usuario, no constituye el núcleo de innovación del proyecto, el cual está centrado en la inteligencia artificial y la gestión de archivos del backend.

Aunque existía una mayor familiaridad previa con Angular por parte del desarrollador, su complejidad inherente y su estructura altamente opinada se consideraron excesivas para las necesidades de la interfaz de este proyecto. Vue.js ofrece un excelente equilibrio entre funcionalidad y facilidad de desarrollo, permitiendo construir una interfaz reactiva y moderna sin la sobrecarga asociada a frameworks más robustos y extensos.

6.2.5.0.2. Implementación La interfaz de usuario desarrollada con Vue.js se comunica con un backend implementado en Flask (Python), el cual actúa como intermediario para interactuar con Prefect y ChromaDB. Las funcionalidades principales implementadas en la UI incluyen:

- Un campo de búsqueda central donde el usuario puede introducir sus consultas en lenguaje natural.
- Visualización clara y ordenada de los resultados de la búsqueda, mostrando la ruta y si procede una miniatura del archivo.
- Una sección de configuración que permite al usuario modificar el Host y el puerto del Backend, cambiar el modelo de IA utilizado para la respuesta final y la temperatura de la respuesta generada por el LLM y un botón para reinciar el chat.
- Una vista de ”explorador de archivos” que permite navegar por los archivos ya procesados por el sistema.

La aplicación se ha estructurado sobre un único componente por comodidad ya que es una web sencilla y se ha puesto especial atención en crear una interfaz amigable y acogedora. Por ejemplo, se utilizan ”emoticonos” en mensajes iniciales o de ayuda, una estrategia observada en plataformas orientadas al cliente, como el soporte de Amazon, con el fin de hacer la experiencia del usuario más agradable al interactuar con lo que se pretende sea un ”asistente inteligente” para los archivos del usuario final.

El principal desafío durante la implementación fue la curva de aprendizaje inicial de Vue.js, al no ser el framework con el que se tenía mayor experiencia previa. Se priorizó una funcionalidad básica pero robusta, dejando posibles mejoras estéticas avanzadas o funcionalidades secundarias de la UI para futuras iteraciones, dado el enfoque del proyecto en la funcionalidad del backend.

Las Figuras 6.3, 6.4 y 6.5 muestran diferentes pantallas de la interfaz de usuario desarrollada.

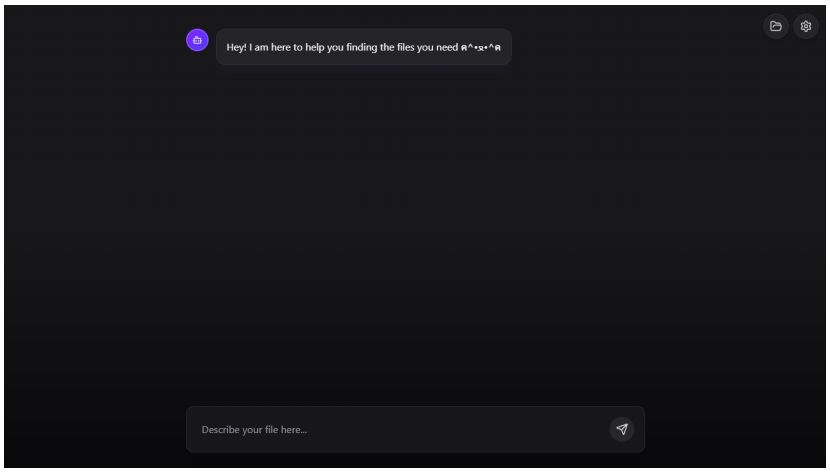

Figura 6.3: Pantalla principal de la interfaz de usuario, con el campo de búsqueda.

6.2.6. API REST: Flask

6.2.6.0.1. Decisión y Justificación Para la comunicación entre el frontend (Vue.js) y los servicios del backend (Prefect, ChromaDB, lógica de negocio), se ha optado por desarrollar una API RESTful utilizando **Flask** en Python. Esta elección se basa en la simplicidad, ligereza y flexibilidad de Flask, que permite crear rápidamente endpoints bien definidos. Al ser Python el lenguaje principal del proyecto, Flask facilita una integración natural con los demás componentes del backend.

6.2.6.0.2. Implementación El servidor Flask se ha configurado para servir tanto la API REST como los archivos estáticos de la aplicación Vue.js (generados tras el proceso de compilación de Vue). Los principales endpoints de la API incluyen:

- `/api/status`: Proporciona información sobre el estado general del sistema, como la cantidad de archivos procesados o el estado de los servicios de monitorización.
- `/api/models`: Devuelve información sobre los modelos de IA configurados y disponibles para el procesamiento de consultas o análisis de archivos.
- `/api/query`: Recibe las consultas de búsqueda textuales del usuario desde la interfaz, las procesa y las reenvía al flujo de Prefect correspondiente para obtener resultados de ChromaDB y la respuesta generada por el LLM.
- `/api/path_descs`: Permite obtener una lista de todos los archivos que han sido procesados y están indexados en el sistema, con metadatos básicos.
- `/api/file_content`: Dado la ruta de un archivo de imagen, devuelve su contenido para ser visualizado en la interfaz.

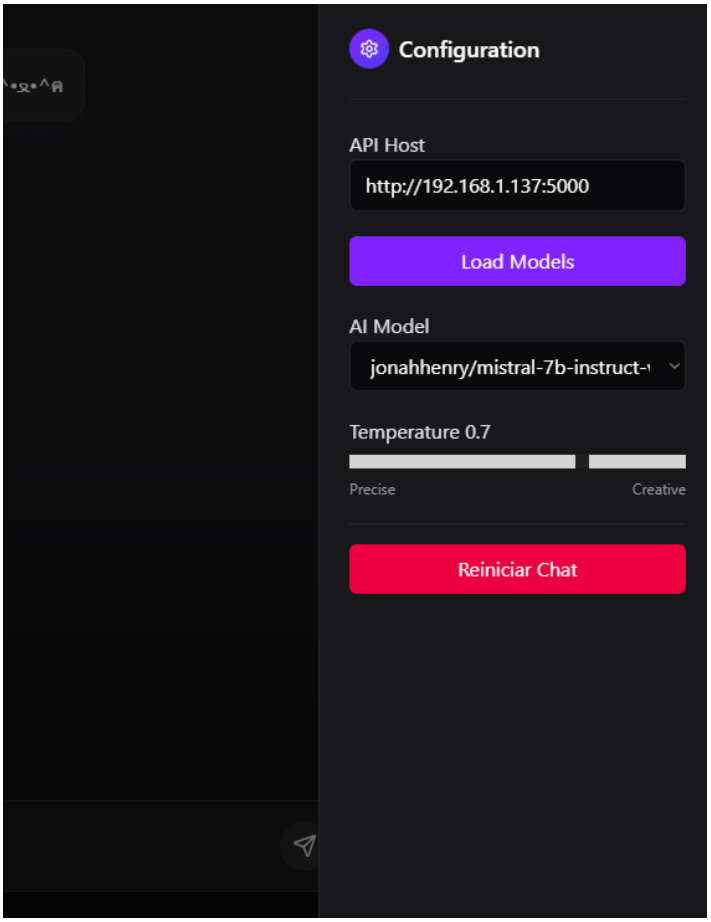

Figura 6.4: Pantalla de configuración de directorios a monitorizar.

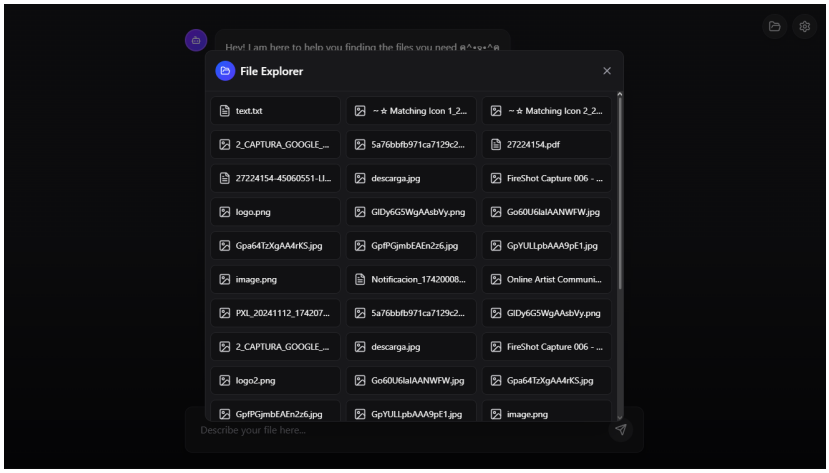

Figura 6.5: Pantalla del explorador de archivos procesados.

- `/api/file_details`: Proporciona la descripción y los metadatos detallados de un archivo identificado por su ruta.

Se implementó Cross-Origin Resource Sharing (CORS) (Cross-Origin Resource Sharing) para permitir las solicitudes desde el servidor de desarrollo de Vue.js al servidor Flask durante la fase de desarrollo. Un desafío grande fue la gestión del estado y la sincronización de la información para el endpoint `/api/status`, asegurando que refleje de manera precisa y actualizada el estado de los diversos componentes del sistema.

6.2.7. Gestión de Modelos de IA: LMStudio

6.2.7.0.1. Decisión y Justificación Para la gestión y ejecución local de los modelos de lenguaje grande (LLM) y modelos multimodales, se ha optado por utilizar **LMStudio**. La principal ventaja de LMStudio radica en su facilidad de uso: es una aplicación de escritorio que permite descargar, configurar y ejecutar una amplia variedad de modelos de IA de código abierto (provenientes de plataformas como Hugging Face) a través de una interfaz gráfica intuitiva. Además, expone los modelos cargados a través de un servidor local compatible con la API de OpenAI, lo que simplifica enormemente la integración con el código Python del proyecto.

La alternativa habría sido gestionar la descarga, configuración y ejecución de cada modelo directamente mediante bibliotecas de Python como 'transformers' o 'llama-cpp-python'. Si bien esto ofrecería un control más granular, también implicaría una mayor complejidad en el código y en el proceso de configuración inicial para el usuario final del proyecto. LMStudio abstrae esta complejidad.

6.2.7.0.2. Implementación LMStudio se utiliza como un componente externo al código principal del proyecto. El usuario debe instalar LMStudio, descargar los modelos deseados (por ejemplo, Mistral para generación de texto y Gemma para análisis multimodal) y ejecutarlos a través del servidor local que provee la aplicación. El backend de Python del sistema se comunica con este servidor local de LMStudio mediante peticiones HTTP a los endpoints estándar de la API de OpenAI (e.g., `/v1/chat/completions` para LLM o `/v1/completions` para modelos que lo soporten, adaptándose según el modelo específico y su configuración en LMStudio).

Esta dependencia de un software externo implica que el usuario debe realizar estos pasos de configuración manualmente. No obstante, para el alcance de este proyecto, la simplificación en el desarrollo y la flexibilidad para probar diferentes modelos que ofrece LMStudio superan la desventaja de la configuración manual. En la documentación del proyecto se detallan los pasos para configurar LMStudio y los modelos recomendados.

6.2.8. Interfaz de Línea de Comandos (CLI)

6.2.8.0.1. Decisión y Justificación Además de la interfaz gráfica de usuario, se consideró útil proporcionar una interfaz de línea de comandos (CLI) para permitir interacciones básicas con el sistema, como realizar búsquedas o iniciar el proceso de monitorización. Esto puede ser ventajoso para usuarios avanzados, para la automatización de tareas mediante scripts o para entornos donde una GUI no está disponible o no es deseada. Se utilizó el ecosistema estándar de Python para empaquetar y distribuir esta CLI.

6.2.8.0.2. Implementación Para crear un punto de entrada ejecutable desde la terminal, se ha utilizado la funcionalidad de `entry_points` de **setuptools**, la biblioteca estándar de Python para la construcción y distribución de paquetes. Se definió un archivo `setup.py` (o `pyproject.toml` con la configuración equivalente) que incluye la siguiente configuración para los scripts de consola:

Código 6.1: Definición del punto de entrada en setup.py

```
from setuptools import setup

setup(
    name="llmsearch",
    version="0.1",
    py_modules=["llmsearch"],
    entry_points={
        "console_scripts": [
            "LLMSearch=llmsearch:main",
        ],
    },
)
```

En el ejemplo anterior, existe un módulo Python llamado `llmsearch_cli.py` que contiene una función `main()`. Esta función se encarga de parsear los argumentos proporcionados en la línea de comandos (utilizando bibliotecas como `argparse`) y de invocar la lógica correspondiente en el backend del sistema enviando una solicitud a la API REST.

Una vez instalado el paquete el usuario puede invocar la CLI desde cualquier ubicación en su terminal:

```
LLMSearch --help
LLMSearch --query "mapa del mundo"
```

Esta aproximación ofrece una forma estándar y robusta de crear herramientas de línea de comandos en Python, facilitando la interacción del usuario con las funcionalidades principales del sistema sin depender exclusivamente de la interfaz web.

7. Resultados

7.1. Evaluación y Pruebas de Concepto

Para validar la viabilidad de los componentes clave del sistema LLMSearch, especialmente en lo referente a la búsqueda semántica y la gestión de embeddings, se realizaron pruebas de concepto utilizando la base de datos vectorial ChromaDB. Esta sección detalla un experimento específico diseñado para ilustrar cómo ChromaDB maneja la creación, almacenamiento, búsqueda y visualización de embeddings a partir de un conjunto de documentos de ejemplo.

El objetivo principal de esta prueba fue observar la capacidad de ChromaDB para:

- Generar representaciones vectoriales (embeddings) de fragmentos de texto.
- Almacenar estos embeddings de forma persistente.
- Realizar búsquedas semánticas basadas en la similitud del coseno entre el embedding de una consulta y los embeddings de los documentos almacenados.
- Facilitar la comprensión de las relaciones semánticas mediante herramientas de visualización.

7.1.1. Configuración del Experimento con ChromaDB

Se utilizó un script de Python que interactúa con una instancia local y persistente de ChromaDB. **El código completo de este script de prueba se puede encontrar en el Anexo A.** Se definió un corpus de ocho documentos de texto concisos, cuyos temas giran en torno a la programación (Python), los embeddings, las bases de datos vectoriales (ChromaDB) y el procesamiento del lenguaje natural. Los documentos empleados fueron:

1. *"Python is a high-level, interpreted programming language"*
2. *"Embeddings are vector representations of text"*
3. *"Chroma is a vector database for storing embeddings"*
4. *"Language models can generate semantic embeddings"*
5. *"3D visualization helps to understand the distance between embeddings"*
6. *"Vector databases are useful for semantic searches"*
7. *"Embeddings capture the semantics of words and phrases"*
8. *"Python has many libraries for natural language processing"*

Estos documentos fueron procesados para generar sus respectivos embeddings utilizando el modelo de embedding por defecto de ChromaDB. Posteriormente, se creó una colección denominada `"example_embeddings"` donde se almacenaron los documentos junto con sus embeddings.

7.1.2. Resultados de la Búsqueda Semántica

Se realizó una búsqueda semántica utilizando la consulta: `"What are embeddings?"`. El sistema fue instruido para devolver los 3 resultados más similares. Los resultados obtenidos, incluyendo el documento y su distancia semántica respecto a la consulta, se muestran en la Figura 7.1.

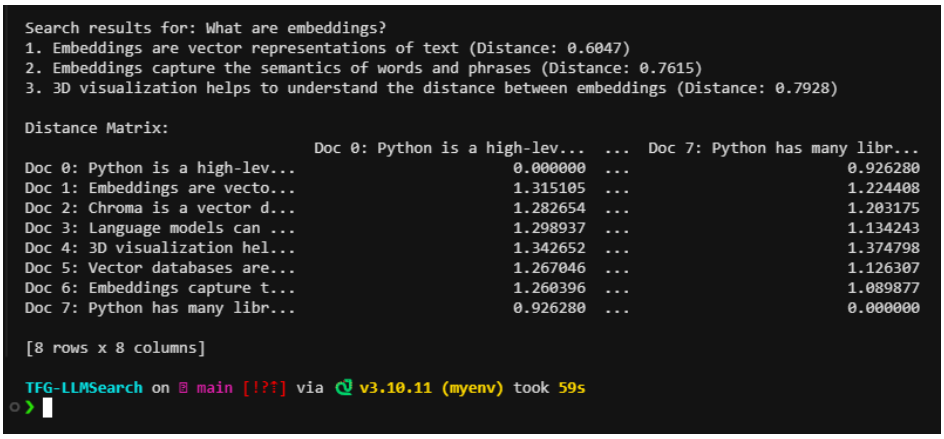

Figura 7.1: Salida de consola mostrando los resultados de la búsqueda para la consulta "¿What are embeddings?". Se observa que los documentos más relevantes, con menor distancia, son recuperados.

Como se aprecia en la Figura 7.1, los documentos recuperados son altamente pertinentes a la consulta. El documento *"Embeddings are vector representations of text"* es el más cercano (menor distancia), seguido por *"Embeddings capture the semantics of words and phrases"* y *"Language models can generate semantic embeddings"*. Esto demuestra la capacidad de ChromaDB para identificar y priorizar documentos semánticamente relevantes a una consulta en lenguaje natural.

7.1.3. Visualización de Embeddings

Para comprender mejor la distribución espacial y las relaciones semánticas entre los documentos y la consulta, se generaron dos tipos de visualizaciones.

7.1.3.1. Visualización 3D de Embeddings

Los embeddings de los ocho documentos y el embedding de la consulta fueron proyectados en un espacio tridimensional utilizando técnicas de reducción de dimensionalidad (como PCA o t-SNE, aplicadas internamente por la utilidad de visualización de ChromaDB). El resultado se muestra en la Figura 7.2.

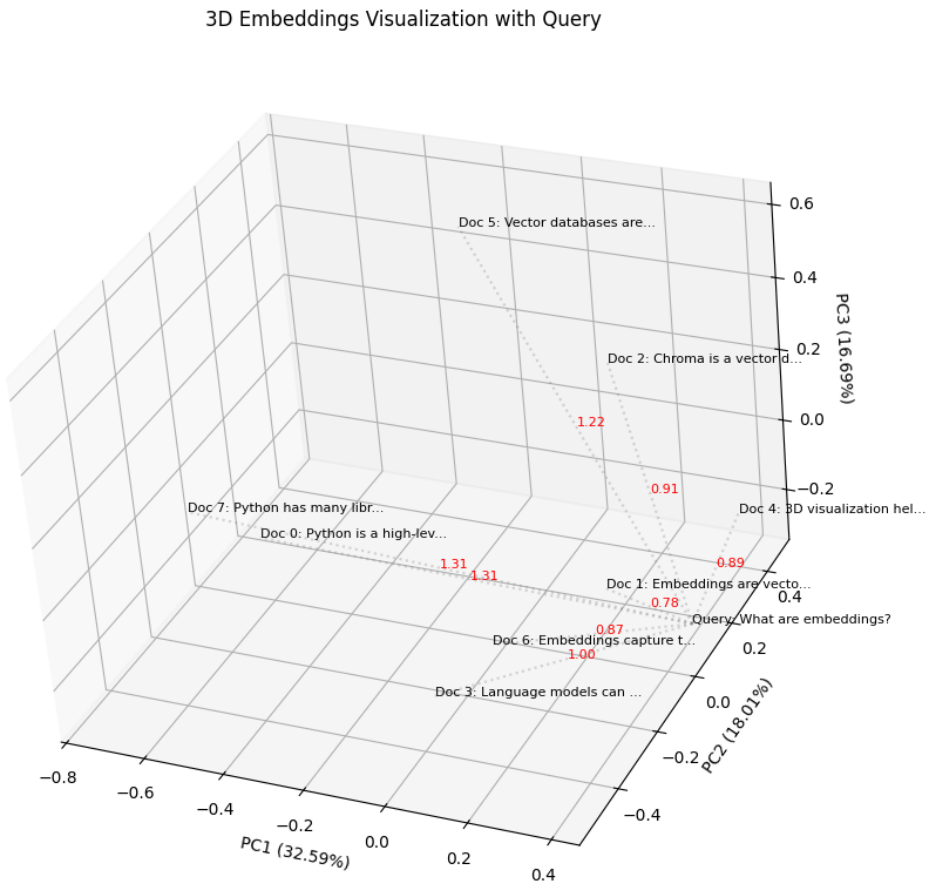

Figura 7.2: Representación 3D de los embeddings de los documentos de ejemplo y la consulta. El punto de la consulta (”Query: What are embeddings?”) está resaltado.

En la Figura 7.2, cada punto representa un embedding. Se puede observar cómo los documentos semánticamente similares tienden a agruparse. El punto correspondiente a la consulta `"Query: What are embeddings?"` se encuentra espacialmente cercano a los embeddings de los documentos que tratan sobre embeddings (por ejemplo, ”Doc 1: Embeddings are...”, ”Doc 6: Embeddings capt...”). Esta proximidad visual corrobora los resultados numéricos de la búsqueda.

7.1.3.2. Matriz de Distancias Semánticas

Para obtener una visión cuantitativa de las distancias entre todos los pares de documentos, se generó una matriz de distancias. Esta matriz (Figura 7.3) muestra la distancia semántica (por ejemplo, distancia coseno) entre cada par de embeddings de los documentos originales.

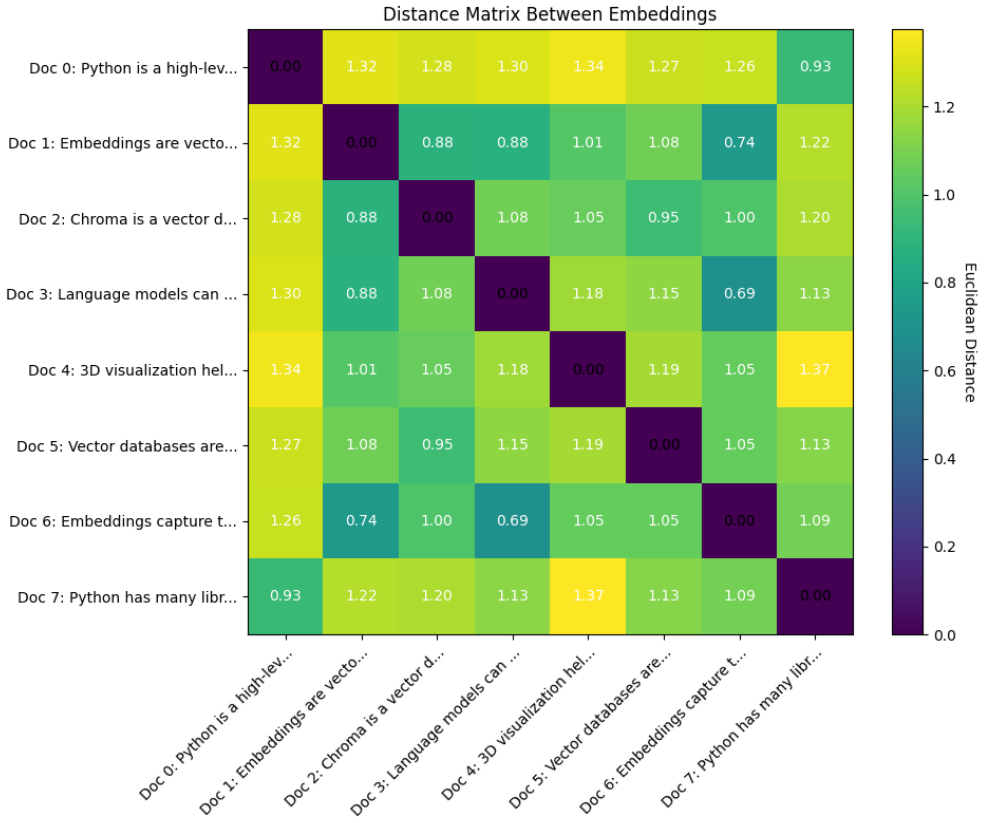

Figura 7.3: Matriz de distancias que muestra la similitud semántica par a par entre los documentos de ejemplo. Colores más oscuros indican menor distancia (mayor similitud).

La Figura 7.3 (asumiendo que la imagen ‘chroma_confussion_matrix.png‘ es en realidad una matriz de distancias como la generada por ‘visualize_matriz_distances‘) permite identificar clústeres de documentos semánticamente relacionados. Por ejemplo, los documentos que hablan sobre ”Python” podrían mostrar distancias menores entre sí en comparación con documentos que hablan exclusivamente sobre ”embeddings”.

7.1.4. Conclusiones de la Evaluación Preliminar

Las pruebas realizadas con ChromaDB demuestran su idoneidad como componente central para la funcionalidad de búsqueda semántica en LLMSearch. La capacidad de generar, almacenar y buscar embeddings eficientemente, junto con las herramientas para visualizar y comprender las relaciones semánticas, son fundamentales para el proyecto.

Esta evaluación preliminar valida la elección de una base de datos vectorial como ChromaDB. Pruebas de rendimiento más exhaustivas con volúmenes de datos mayores y diferentes tipos de ficheros serán necesarias en etapas posteriores para evaluar la escalabilidad y optimizar la configuración del sistema. Sin embargo, esta prueba de concepto inicial es prometedora y sienta una base sólida para el desarrollo de las capacidades de búsqueda inteligente de LLMSearch.

8. Conclusiones

A. Script de Prueba para ChromaDB

A continuación, se presenta el script de Python utilizado para las pruebas de concepto con ChromaDB, detalladas en la Sección 7.1.

Código A.1: Script de Python para la prueba de concepto con ChromaDB.

```python
import numpy as np
import matplotlib.pyplot as plt
from chromadb import Client as ChromaClient # Asumo que así importas tu cliente

# Definición de la clase ChromaClient o funciones si no es una clase estándar
# Si ChromaClient es una clase que tú has definido, asegúrate que esté disponible
# o que las funciones que usa (create_embeddings, etc.) estén definidas.
# Por simplicidad, voy a asumir que tu `ChromaClient` ya tiene esos métodos.
# Si `ChromaClient` es de la librería `chromadb`, entonces el import es suficiente.

def prove() -> None:
    # Usar una ruta relativa o absoluta que funcione en tu entorno
    chroma = ChromaClient(settings={"persist_directory": "./data/chroma_prove_db",
                                    "chroma_db_impl": "duckdb+parquet"}) # Ejemplo de settings si es necesario

    # Create some example documents
    documentos = [
        "Python is a high-level, interpreted programming language",
        "Embeddings are vector representations of text",
        "Chroma is a vector database for storing embeddings",
        "Language models can generate semantic embeddings",
        "3D visualization helps to understand the distance between embeddings",
        "Vector databases are useful for semantic searches",
        "Embeddings capture the semantics of words and phrases",
        "Python has many libraries for natural language processing"
    ]

    # Crear embeddings para los documentos
    # Esta parte depende de cómo tu ChromaClient o la librería chromadb genera embeddings.
    # Si usas el modelo por defecto de la librería:
    # (No necesitas llamar a create_embeddings si la librería lo hace internamente al añadir)
    # Para este ejemplo, asumiré que tienes un método o que la librería lo
```

```
    ↪ maneja.
# Si `chroma.create_embeddings` no existe, deberás usar el método correcto
# de la librería `chromadb` para obtener los embeddings, ej. un ↩
    ↪ EmbeddingFunction.

# Crear una colección en ChromaDB (esto también podría crearla si no existe↩
    ↪  al añadir documentos)
collection_name = "example_embeddings"
try:
    collection = chroma.get_collection(name=collection_name)
except: # Ajusta la excepción específica si es necesario
    collection = chroma.create_collection(name=collection_name)

# Añadir documentos a la colección.
# ChromaDB típicamente requiere IDs para los documentos.
ids = [f"doc{i}" for i in range(len(documentos))]

# Si ChromaDB genera los embeddings automáticamente al añadir, no necesitas↩
    ↪  pasarlos explícitamente.
# Si SÍ necesitas pasar embeddings pre-calculados, necesitarías una función↩
    ↪  para ello.
# Este ejemplo asume que la librería puede manejar los embeddings ↩
    ↪ directamente o con una
# función de embedding configurada al crear la colección/cliente.
collection.add(
    documents=documentos,
    ids=ids
)

# Realizar una búsqueda
query = "What are embeddings?"
resultados = collection.query(
    query_texts=[query],
    n_results=3,
    include=['documents', 'distances'] # Asegúrate de incluir lo que ↩
        ↪ necesitas
)

print("\nSearch results for:", query)
if resultados['documents'] is not None and len(resultados['documents'][0]) ↩
    ↪ > 0:
    for i, doc in enumerate(resultados['documents'][0]):
        distance = resultados['distances'][0][i] if resultados['distances'] ↩
            ↪ else 'N/A'
        print(f"{i+1}. {doc} (Distance: {distance:.4f})")
else:
    print("No similar documents were found.")

# Para la visualización 3D y matriz de distancias, necesitarías obtener ↩
    ↪ todos los embeddings
# y el embedding de la consulta. La librería `chromadb` puede que no ↩
```

```
        ↪ ofrezca estas
    # visualizaciones directamente como `chroma.visualize_embeddings_3d`.
    # Estas visualizaciones suelen hacerse con librerías como matplotlib, ↩
        ↪ plotly, scikit-learn (para PCA/t-SNE).

    # Obtener todos los embeddings de la colección (si es necesario para ↩
        ↪ visualización manual)
    all_docs_data = collection.get(include=['embeddings', 'documents'])
    stored_embeddings = np.array(all_docs_data['embeddings'])
    stored_documents_text = all_docs_data['documents']

    labels = [f"Doc {i}: {doc[:20]}..." for i, doc in enumerate(↩
        ↪ stored_documents_text)]

    # Si quieres visualizar la consulta también, necesitas su embedding
    # Esto depende de cómo se obtienen los embeddings (ej. usando la misma ↩
        ↪ embedding function)
    # Para el ejemplo, no se incluye la visualización 3D/matriz compleja aquí
    # ya que requeriría más código para PCA/t-SNE y plotteo con matplotlib/↩
        ↪ plotly
    # que no está en tu `prove()` original de forma explícita con la librería `↩
        ↪ chromadb`.

    # Si las funciones `visualize_embeddings_3d` y `visualize_matriz_distances`
    # eran parte de TU clase `ChromaClient` personalizada, tendrías que ↩
        ↪ incluirlas.
    # Las librerías estándar de ChromaDB no suelen tener estas funciones de ↩
        ↪ visualización directa.

    # plt.show() # Solo si generas figuras con matplotlib

    # print("\nDistance Matrix (conceptual):")
    # (Calcular y mostrar la matriz de distancias requeriría sklearn.metrics.↩
        ↪ pairwise_distances por ejemplo)

if __name__ == '__main__':
    prove()
```