Universidad de Murcia

Trabajo Fin de Grado

Implementación y evaluación de un sistema basado en Retrieval Augmented Generation

Estudiante:
Pedro Jiménez Gómez
pedro.j.g@um.es

Tutor:
Eduardo Martínez Graciá
edumart@um.es

Cotutor:
Juan Antonio Botía Blaya
juanbot@um.es

Junio 2025

A Dios, por haberme permitido estudiar. Porque *"La belleza salvará al mundo"*, y he podido disfrutar de esta belleza, también a través de la ciencia.

A Eduardo, por su valiosa guía, cercanía y todo lo que me ha ayudado a aprender a lo largo de este trabajo.

A mi gente. A mi comunidad. A la música. A mis amigos. Especialmente, a mis compañeros durante estos años de carrera, el *kernel*, por todas las *horas de diversión* que hemos pasado juntos. A todos los que de alguna forma me han apoyado a lo largo de esta etapa, por haberme permitido desconectar y volver a enchufar cuando ha sido necesario.

A mi familia. A mis abuelos, mis tíos, mis primos. Por la incondicional ayuda y confianza que me han transmitido a lo largo de toda este proceso.

A Marta, Juan e Isabel. Mis hermanos. Por haberme aguantado y ayudado. No lo podría haber hecho sin ellos, este logro también es suyo.

A mis padres, Jacinto y Juana Mª. Por haberme convertido en la persona que soy hoy. Por ser un pilar incondicional a lo largo de esta etapa y de toda mi vida. Gracias por su amor, su apoyo constante y sus sacrificios, que han sido fundamentales para que hoy pueda llegar hasta aquí.

Índice general

Índice de figuras

Resumen

Este proyecto ha sido desarrollado por Pedro Jiménez Gómez, bajo la supervisión de Eduardo Martínez Graciá, en la Universidad de Murcia, durante el curso 2024-2025, como Trabajo de Fin de Grado en Ingeniería Informática.

En este trabajo se plantea el uso de la arquitectura RAG (Retrieval-Augmented Generation) como solución para mitigar algunas de las limitaciones de los grandes modelos del lenguaje, y se muestra el desarrollo de una aplicación sencilla que permite a los estudiantes poder resolver dudas relacionadas con una asignatura. Esto será posible gracias al propio flujo RAG, que permite *aumentar* (o especializar) la base de conocimiento de los LLMs.

En el primer capítulo se realiza una introducción al aspecto principal sobre el que va a tratar el proyecto: los sistemas RAG. Se presentan los grandes modelos del lenguaje y algunas de sus limitaciones, y se comenta cómo un flujo RAG puede ayudar a mitigarlas. El capítulo contiene también un resumen de la metodología seguida durante el desarrollo del proyecto.

El segundo capítulo aborda el estado del arte, un estudio teórico en el que se detallan todos los aspectos necesarios para la posterior implementación de un sistema RAG. Entre ellos, podemos destacar la IA generativa, los LLMs, la propia arquitectura RAG, la fragmentación semántica, las bases de datos vectoriales y el prompting.

En el tercer capítulo se describen los objetivos del proyecto, y se presentan todas las herramientas utilizadas, junto con una explicación de cada una de ellas, comentando qué papel han tenido en el desarrollo de este proyecto.

El cuarto capítulo contiene la parte principal del proyecto. En él se muestra, paso a paso, cómo se ha implementado la aplicación de consulta de contenidos de los apuntes de una asignatura del Grado en Ingeniería Informática. Se explica cómo incorporar las librerías necesarias, cómo se realiza la creación de un cluster en la base de datos vectorial *Weaviate Cloud*, cómo se indexan y recuperan los documentos, el prompting que emplea el sistema implementado y, por último, la evaluación del mismo.

Para finalizar, en el quinto capítulo se presentan algunas conclusiones que se han obtenido tras el desarrollo de todo el proyecto, y también se presentan posibles vías futuras de este trabajo.

Extended abstract

This project has been written by Pedro Jiménez Gómez under the supervision of Eduardo Martínez Graciá. It concerns building applications using RAG's (Retrieval-Augmented Generation). The primary aim of this work is to present the development of an application that enables students to ask a LLM specific questions related to a subject, which it can answer using class notes and documents, followed by an analysis of the results obtained.

To start with, we introduce the concept of a Large Language Model (LLM). It is a type of artificial intelligence system trained on massive amounts of data, enabling it to understand and generate human-like language. LLMs can answer questions, summarize texts, translate, and perform many other language related tasks. Despite their potential, their knowledge is limited to what they were trained on and may not include recent or domain-specific information.

To achieve this, we rely on a Retrieval-Augmented Generation (RAG) approach - a method that combines the power of large language models with an external information retrieval system. Instead of relying solely on pre-trained knowledge, RAG systems employ other information sources (such as class notes) before generating a response. This ensures that the answers are grounded in specific, up-to-date content.

The main idea behind RAG is to enrich the model with access to external knowledge sources like documents, databases, or anything related else. It has been proved that this procedure can be very useful for generating better answers. It consists of a process divided into several stages, as its name indicates.

The fist step in the process is the Retrieval. A RAG system starts by retrieving relevant information from an external knowledge base. The goal is to find the most relevant document snippets that can help answer the questions of the users. To do this, a document retrieval model is usually implemented (or some of them) such an embedding system or a vector search engine. On this occasion, our main application harnesses *Weaviate*, which is an open-source vector database that stores both objects and vectors, allowing the combination of vector search with structured filtering with the fault tolerance and scability of a cloud-native database.

Next, the *Augmented* part is the most conceptual part of the system. The term means that the model does not generate answers solely with its pre-trained knowledge, but instead uses the retrieved information as additional context. It is as if the model could

consult notes, documents, or books in real-time before answering.

This represents a significant shift from traditional language models, which can only generate text based on what they learned during training. In contrast, a RAG model is dynamically enriched with new information every time it receives a query. This ability to augment its knowledge allows models to be more accurate and relevant, to stay up to date (very useful for fast-changing topics as technology or medicine), and to avoid hallucinations (when a model makes up incorrect data).

Once the relevant information has been retrieved, the model moves to the generation phase. Here, a model uses both the question of the user and the retrieved documents to build coherent, natural and informative answer. This generation can be extractive – if the model simply selects parts of the retrieved text – or abstractive – if the model rewrites and summarizes the information in new words, tailored to the original question. Even though that information comes from an external source, the response feels fully integrated and personalized.

This approach significantly enhances the accuracy of generative models by allowing them to retrieve relevant information from external sources. This improves fact retention and reduces reliance on the model's pre-trained, often generic, knowledge. Instead of generating responses solely from internal data, RAG pulls in real, specific documents, increasing the likelihood of delivering factual and reliable answers.

RAG also brings key improvements such as more logical reasoning, thanks to access to information directly related to the user's query. It enhances contextual relevance and makes it easier to validate the responses by referencing the original sources. While not entirely error-proof, this hybrid method lowers the risk of producing false or nonsensical outputs, contributing to more trustworthy and transparent language model behavior.

There are three paradigms that are most mainstream of Retrieval-Augmented Generation (RAG) systems and each one of them offer an incremental advance with respect to flexibility, precision, and complexity of design: Naive RAG, Advanced RAG, and Modular RAG. Each paradigm shows the evolution of the way external knowledge is retrieved, merged, and used by large language models for generation.

Naive RAG is a trivial "Retrieve-Read" pipeline with indexing, retrieval, and generation. Documents are preprocessed and split into chunks, which are embedded and stored in a vector database. Upon processing a user query, the top-k most semantically related chunks are fetched and sent to the language model to get a response. While this approach is inexpensive and simple, it suffers from limitations like retrieval imprecision, hallucinated responses, and poor contextual fusion.

Advanced RAG superimposes additions to these weaknesses. It integrates pre-retrieval methods such as query rewriting and metadata-based fine-grained indexing and post-retrieval techniques such as reranking and context compression. These additions support not only relevance and accuracy in the content being retrieved but also ensure the most valuable information is highlighted in the output prompt fed into the language model. The variant remains sequential but much higher-performing in knowledge-intensive applications.

Modular RAG builds upon these ideas and provides a very flexible architecture. It introduces specialized modules such as memory components, semantic routers, search modules, and task adapters. These modules can be rearranged or replaced, supporting non-linear processes like iterative and adaptive retrieval. Modular RAG can also support sophisticated mechanisms like multi-query expansion, retrieval-conditioned generation, and reinforcement-learning-based routing. This system is best suited for complex tasks and dynamic environments, allowing one to build more accurate, scalable, and context-sensitive AI systems.

To deploy the application, *LlamaIndex* is the key piece. *LlamaIndex* is an open-source platform designed to facilitate the integration of LLMs with external, bespoke data sources. Its primary function is to act as an intermediary layer facilitating LLMs to access and utilize data in documents, databases, APIs, or any other structured or unstructured format. This makes it a core component of the use of RAG architectures, in which one wants to enhance text generation using appropriate, domain-related knowledge that is not initially present in the model's training material.

The *LlamaIndex* process is based on three main stages: data ingestion, indexing, and querying. The system first ingests data from various sources and splits it into tractable pieces of text. These blocks are subsequently transformed into vectorized forms – embeddings – and stored in specialized indexes, such as vector stores, keyword tables, or hierarchical indexes. When a user inputs a query, *LlamaIndex* retrieves the most relevant chunks of data and returns them to the language model as additional context. The model can leverage this to generate more accurate, coherent, and fact-based responses.

Apart from being easy to use and easily integrated into all sorts of models, *LlamaIndex* stands out with scalability, personalization, and compatibility with numerous tools within the generative AI space. It is particularly beneficial in situations that include steady access to high-quality specialized information, such as corporate bots, legal or medical assistants, intelligent learning platforms, and semantic search engines. Both in school and in the workplace, *LlamaIndex* enables the construction of robust solutions that combine the generative power of LLMs and the stability of structured data, significantly improving response accuracy, traceability, and user confidence.

In this project, the RAG system that has been built consist of a chat with an LLM based on an OpenAI model enriched with notes from the subject *Autómatas y Lenguajes Formales (ALF)*, from the degree in Computer Engineering at the University of Murcia. The students must be able to ask question to the chatbot, and to upload new documents realted to this topic.

For a query to be resolved, the system must have the documents loaded as embeddings within the vectorized database. This is the duty of the first program developped in this project, which acts as a loader. It takes each document and fragments them into different nodes to later save them in *Weaviate Cloud*, giving the persistence that allows the main application to function regardless of whether the device has access to those documents. This also makes it easier to deploy an application.

Meanwhile, the second component of the application is the core RAG (Retrieval-Augmented Generation) system itself. This system begins by connecting to the Weaviate

Cloud, from which it loads the previously indexed and vectorized document nodes. These nodes represent the chunks of information that were extracted and processed from relevant lecture materials or domain-specific resources. Once the data is retrieved, the system constructs a *query engine*, which plays a central role in the functioning of the RAG pipeline. This engine is responsible for interpreting the user's query, computing semantic similarity, and retrieving the most relevant nodes to serve as context for the language model.

After building the query engine, the application initiates a chat-based interface through which the user can interact naturally with the system. The user is invited to ask questions related to ALF (presumably a course or subject area). When a question is submitted, the system performs a semantic search across the indexed content and retrieves the most pertinent segments. These are then passed to the language model to enrich the prompt and generate a more accurate and grounded response.

For the generation phase, the system employs several *GPT* models, such as *GPT-4o-mini*, a lightweight yet powerful large language model developed by *OpenAI*. This model utilizes both its internal capabilities and the external context provided by the retrieved documents to compose a coherent, informative, and context-aware answer. Once the response is generated, the system displays it in the user interface alongside a list of the consulted documents and the specific pages that were read during the retrieval process. This produces transparency, that not only enhances user trust but also allows verification and deeper exploration of the sources used to construct the answer.

Finally, a verification system is implemented to evaluate the generation capacity of the system. For this purpose, we have chosen a framework called *Trulens*. Though that is its main goal, it can make us understand how the LLM based apps works or detect possible errors in the answers. This software tool helps us to objectively measure the quality and effectiveess of the RAG app using feedback functions.

In addition to the programs mentioned above, another *Python* script has also been developed. This software program helps us to evaluate our application, integrating all the key components we have used, such as *Llamaindex*, and more.

Firstly, the system connects to the vectorial database, loads the indexes from the embeddings and then it creates a query engine, using LlamaIndex. The main class of the script, RAG, encapsulates the core behavior of this architecture. It defines three decorated methods, which indicates that its behavior will be monitored and evaluated by TruLens. One of these methods works retrieving relevant document chunks, another is used to generate the augmented query based on those chunks, and the last one combines both steps into a complete RAG flow.

As it has been mentioned, *TruLens* plays a crucial role by introducing an automated evaluation layer over the generated responses. *TruLens* has three different metrics: the groundedness – whether the response is properly grounded in the retrieved context – answer relevance – whether it accurately addresses the question of the user – and context relevance – whether the retrieved fragments are aligned with the intent of the query. At the end of the script, several questions related to ALF are queried, and each is evaluated using these metrics. The results can be explored through an interactive dashboard.

To sum up, this project explores the use of a RAG architecture, and evaluates how it works by developing an application that answers academic questions using teaching materials. Currently, RAG systems are gaining relevance because they enable accurrate and up-to-date responses based on external, verifiable information, improving LLMs performance by incorporating precise, domain-specific external context in real time.

CAPÍTULO 1

Introducción

1.1. Sistemas RAG

Uno de los pilares fundamentales de la sociedad actual es la tecnología. Dentro de este campo, la Inteligencia Artificial (IA) ha cobrado una importancia especial en los últimos años, y está experimentando una evolución constante. Concretamente, ha habido una revolución en todo lo relacionado con modelos dedicados a la IA generativa (GenAI). Este tipo de IA consiste en sistemas que son capaces de generar nuevo contenido como texto, imágenes, vídeo, audio, etc. [1] Estos sistemas usan técnicas de *Machine Learning (ML)*, como redes neuronales entrenadas con gran cantidad de datos. Son capaces de actuar como máquinas predictoras mediante el análisis y aprendizaje de patrones dentro del conjunto de datos de entrenamiento.

Uno de los ejemplos más destacables es el campo del NLP (*Natural Language Processing*) y del lenguaje automatizado. La innovación más disruptiva en este área ha sido la de los grandes modelos del lenguaje o LLM (*Large Language Models*). Un ejemplo de este tipo es el mundialmente conocido *ChatGPT*, de *OpenAI*. Un LLM consiste en redes neuronales específicamente diseñadas y optimizadas para entender y generar lenguaje humano. A parte del uso general por parte de usuarios particulares, estos modelos se están empezando a utilizar en aplicaciones empresariales muy diversas.

A pesar de que el rendimiento puede resultar aceptable, actualmente se están buscando mejoras como aumentar la capacidad de razonamiento, permitiendo de esta forma la resolución de tareas más complejas. También se está buscando obtener un mayor rendimiento para reducir costes relacionados con los recursos consumidos o el tiempo de inferencia. Pero estos modelos no son perfectos. A continuación se comentan las principales limitaciones de los LLMs.

Una de sus principales limitaciones estriba en que no incorporan a su conocimiento

[1] Se distinguen de otros sistemas de IA que trabajan con modelos específicos que son diseñados para resolver tareas concretas, como, por ejemplo, clasificación de imágenes.

datos en tiempo real, por lo que pueden generar respuestas no actualizadas o incorrectas. Asimismo, carecen de la capacidad de distinguir la verdad de la mentira, de modo que pueden generar desinformación de manera convincente y no intencionada. Tienen dificultades para la coherencia en textos largos, además de que pueden no funcionar en razonamientos complejos o en tareas que requieren de pasos lógicos múltiples.

Otro punto relevante es la falta de transparencia y de interpretabilidad de sus decisiones, lo que complica saber el motivo por el que llegaron a tales conclusiones. Por otra parte, en la medida en que los modelos generan cada vez más texto que queda disponible abiertamente, ese mismo texto generado por la IA empieza a contaminar los datos con los que se entrenan nuevos modelos, lo que repercute negativamente en su calidad y en sus capacidades a largo plazo.

Con todo esto, a pesar de ser poderosos, los LLMs aún tienen retos importantes en precisión, sostenibilidad, y fiabilidad. Es aquí donde nace el concepto de RAG. Un **RAG (Retrieval-Augmented Generation)** es una arquitectura híbrida que combina modelos de lenguaje con un sistema de recuperación de información para generar respuestas más precisas y fundamentadas, que puede acceder a fuentes externas como documentos, bases de datos o sitios web para obtener información relevante en tiempo real. Cuando el sistema recibe una consulta del usuario, busca fragmentos de texto relacionados en una base de conocimiento previamente indexada, y los utiliza como contexto adicional para que el modelo genere una respuesta que mejore la fiabilidad. Además, permite que el sistema se encuentre actualizado con información reciente, sin necesidad de reentrenar el modelo base, operación que tiene un coste enorme. Y es muy relevante el hecho de que un RAG ofrece trazabilidad, ya que se puede indicar qué documentos fueron consultados para invocar al LLM.

1.2. Enfoque y metodología

Este trabajo se centra en el desarrollo de una aplicación que utiliza un sistema RAG, con el objetivo de que sirva como ayuda a la asignatura *Autómatas y Lenguajes Formales*, correspondiente al 2º curso del Grado en Ingeniería Informática de la Universidad de Murcia. El funcionamiento de la aplicación es sencillo: consiste en un chatbot basado en un sistema RAG, añadiéndole como contexto los apuntes de la asignatura que hemos mencionado.

Un sistema RAG es una arquitectura que integra modelos de lenguaje con mecanismos de recuperación de información con el fin de mejorar la calidad y precisión de las respuestas generadas. Opera en dos pasos: primero, recupera documentos pertinentes desde una base de datos o corpus externa empleando como mecanismo de recuperación una consulta relevante para la pregunta del usuario; en segundo lugar, el modelo de lenguaje da como salida una respuesta que se apoya tanto en esa información recuperada, que actúa a modo de contexto, como en su memoria interna. Esto permite que el sistema proporcione respuestas más actualizadas, precisas y contextualizadas, especialmente en casos donde el modelo por sí solo podría carecer de datos específicos o recientes.

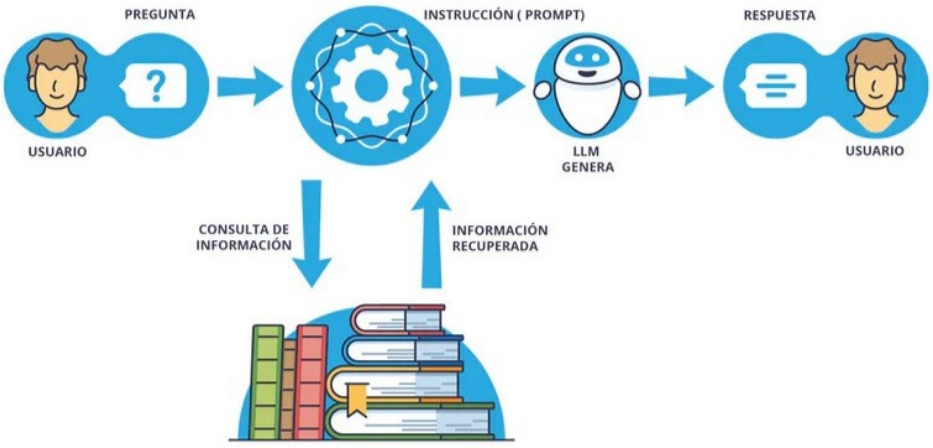

Figura 1.1: Flujo básico de un RAG. Fuente [17]

Para la implementación de nuestro sistema usaremos *LlamaIndex*. Se trata un marco de orquestación de datos de código abierto para crear aplicaciones con grandes modelos de lenguaje. Aprovecha una combinación de herramientas y capacidades que simplifican el proceso de aumento de contexto para casos de uso de IA generativa a través de un pipeline de recuperación aumentada. También usaremos una base de datos vectorial que utiliza el aprendizaje automático para organizar y buscar datos de manera eficiente, llamada *Weaviate*. Nos ayudará a implementar la persistencia. Además, también haremos uso de *Langchain*, un framewok que tiene aplicaciones similares a *LlamaIndex*, y que incluye funcionalidades interesantes para la fragmentación semántica de documentos previa a su indexación. La imagen 1.1 muestra el flujo básico de un RAG.

1.3. Objetivos del trabajo

A continuación se comentan de forma breve los objetivos del proyecto, ya que posteriormente se detallarán en el capítulo 3. Los principales objetivos son:

- Estudio del estado del arte en el área de los sistemas RAG. Se explicarán los aspectos teóricos necesarios para el diseño, desarrollo y funcionamiento de un sistema de estas características.

- Implementación de una aplicación RAG que permita a estudiantes de la asignatura *Autómatas y Lenguajes Formales*, correspondiente al 2º curso del Grado en Ingeniería Informática de la Universidad de Murcia, poder interactuar a través de un chat para resolver dudas.

- Análisis de resultados obtenidos.

1.4. Estructura del trabajo

Según la normativa de los Trabajos de Fin de Grado de la Facultad de Informática de la Universidad de Murcia, el contenido del proyecto queda estructurado de la esta manera:

- El capítulo 2 trata sobre el estado del arte del trabajo. En él se explicará detalladamente cada uno de los conceptos y aspectos teóricos fundamentales comentados previamente.
- El capítulo 3 consiste en un análisis detallado de los objetivos y metodología seguida durante el desarrollo del proyecto.
- El capítulo 4, sobre diseño y resolución, se realiza una descripción sobre la implementación del sistema RAG que se desarrolla. Se explicarán los scripts necesarios que han permitido desarrollar la aplicación, y se mostrará el funcionamiento de la herramienta *Trulens* [18], que permite tener una medida objetiva de la generación del sistema.
- Con el capítulo 5 finalizaremos el trabajo extrayendo conclusiones a través de los resultados obtenidos, y se propondrán algunas posibles vías futuras sobre el uso de los sistemas RAGs.

CAPÍTULO 2

Estado del arte

En este capítulo se realiza un estudio de los conceptos teóricos necesarios para la posterior implementación de nuestra aplicación. En la sección 2.1 se expone una breve introducción a la IA generativa; en el apartado 2.2 se presentan los problemas que se derivan del uso de LLMs, una parte de la IA generativa con muchísima importancia en la actualidad; la sección 2.3 explica el funcionamiento de los sistemas RAG; la sección 2.4 describe el proceso de fragmentación semántica; las bases de datos vectoriales, en las que se almacenan los fragmentos de documentos, se describen en la sección 2.5. Por último, la sección 2.6 trata sobre el prompting, es decir, la creación de las consultas a los LLMs.

2.1. Introducción a la IA generativa

Los *modelos de IA a gran escala*, también conocidos como *LFMs (Large Foundational Models)*, han supuesto un gran avance en el campo de la IA. Estos sitemas son entrenados con grandes cantidades datos, permitiendo de este modo ser capaces de hacer una amplia gama de tareas con gran precisión. Además de automatizar tareas repetitivas, son capaces de comprender y generar contenido, adaptándose a diferentes contextos y dominios. Todo lo previamente señalado facilita el desarrollo de la IA generativa, un campo con un enorme potencial.

La **IA generativa**, o *GenAI*, es el subconjunto de la IA que se encarga de crear nuevo contenido, como texto, imágenes, vídeo, audio o incluso porciones de código software, como respuesta a una petición del usuario [4].

La GenAI se basa en el *deep learning*, modelos avanzados de machine learning que incorporan algoritmos que simulan los procesos de aprendizaje y toma de decisiones en el cerebro humano. Estos sistemas funcionan identificando y codificando patrones y relaciones dentro de una cantidad masiva de datos, para posteriormente usar esta

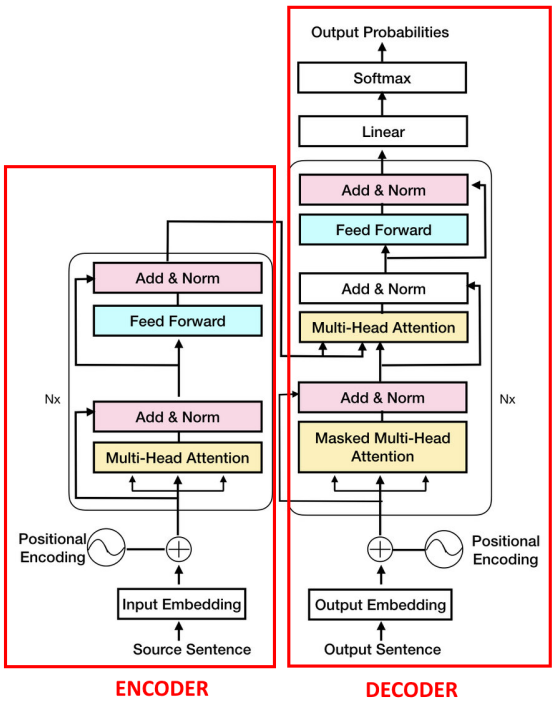

Figura 2.1: Esquema general de una arquitectura transformer. Fuente [5]

información para atender las peticiones de los usuarios, respondiendo con contenido nuevo y relevante mediante inferencia.

Como se ha mencionado, durante la etapa de entrenamiento el modelo se expone a gran cantidad de datos, estructurados o no, y de ellos trata de extraer las características estadísticas y estructurales que subyacen. Esto implica ajustar los parámetros del modelo mediante algoritmos de optimización para minimizar la discrepancia entre las predicciones del modelo y los datos reales.

Tras el entrenamiento, el modelo realiza inferencias, es decir, genera nuevo contenido sobre la base de la información que ha aprendido. Estas inferencias se logran utilizando distribuciones de probabilidad para crear resultados que sean coherentes y realistas dentro del espacio de datos observado [14].

Hay que enfatizar que la IA generativa no trata de replicar los datos de entrenamiento que mejor se ajusten a la petición del usuario, sino que tiene la capacidad de imaginar y crear nuevas instancias que se ajusten a las características aprendidas.

Ha habido numerosas arquitecturas de redes neuronales que han conseguido resultados destacables durante la evolución de este campo, pero podemos destacar la arquitectura ***Transformer***, especializada en la transformación de una secuencia de entrada a una secuencia de salida. Para ello, este tipo de redes aprenden del contexto y rastrean las relaciones entre los componentes de la secuencia de entrada y los de la salida en los datos de entrenamiento [2].

La arquitectura *Transformer* está fundada en un mecanismo central de funcionamiento denominado *atención* (en particular, en atención multi-cabeza o *multi-head attention*),

que permite que el modelo evalúe de manera dinámica la relevancia de cada palabra de la secuencia de entrada respecto a las demás a la hora de generar la siguiente palabra (o fragmento de palabra, denominado *token*). Se ha comprobado que estos modelos pueden recoger dependencias entre las distintas palabras incluso cuando éstas están muy distantes en la secuencia.

En la figura 2.1 se muestran, en forma de diagrama, los diferentes componentes en la arquitectura Transformer. Como se puede observar, ésta se divide en un codificador y un decodificador. Cada capa está compuesta, generalmente, por dos componentes principales: una capa de atención autodirigida y una capa de red completamente conectada. Ambas capas se encuentran rodeadas de elementos de normalización y conexiones residuales, para que se facilite el entrenamiento a múltiples capas.

El modelo procesa todas las palabras del texto al mismo tiempo, lo que lo hace mucho más rápido y eficiente que modelos anteriores. Al usar varias capas, puede entender mejor el contexto y el significado de lo que se dice. En tareas generativas, aprende a predecir la siguiente palabra, lo que le permite generar nuevo contenido coherente y con sentido.

2.2. Problemas derivados del uso de LLMs

Uno de los campos más populares de la GenAI es *Natural Language Generation (NLP)*, mediante grandes modelos del lenguaje (LLMs).

Los **LLMs (en inglés, Large Language Models)** son redes neuronales específicamente diseñadas y optimizadas para entender y generar lenguaje humano. Son a gran escala (en inglés, *large*) en el sentido de que son entrenados con cantidades muy grandes de texto, que contienen billones e incluso trillones de palabras. Los modelos más grandes muestran un mejor rendimiento en las pruebas de referencia, una mejor generalización y otras habilidades. Al contrario que los sitemas previos, la principal característica distintiva de un LLM es su capacidad para producir textos originales que se leen con naturalidad [6].

Como hemos mencionado en el apartado anterior, a través del aprendizaje de patrones desde muchas fuentes, los LLMs adquieren habilidades del lenguaje que encuentran en sus datos de entrenamiento, desde reglas gramaticales hasta conocimiento de la materia de los textos, e incluso razonamientos de sentido común. Estos patrones aprendidos permiten a los LLMs extender texto escrito por humanos de varias maneras relevantes. Mientras siguen mejorando, estos modelos van creando nuevas posibilidades para la generación automática de lenguaje natural.

Durante el entrenamiento, los LLMs van aprendiendo gradualmente relaciones entre palabras y reglas que rigen la estructura del lenguaje del gran conjunto de datos de entrenamiento. Una vez entrenado, son capaces de generar texto que parece proveniente de un humano, todo esto prediciendo la probabilidad de la siguiente palabra de una secuencia, en función de lo previamente generado.

Sin embargo, los LLMs tienen algunos inconvenientes importantes, y estas limitaciones pueden producir algunos efectos colaterales. Aunque puede que haya más que no se indican en la relación siguiente, los más destacables son [16]:

- **Alucinaciones:** El sistema puede inventarse cosas. Las respuestas pueden ser sintáctica y semánticamente correctas, pero la información puede ser falsa. Esto puede ser beneficioso, por ejemplo, en tareas creativas. No obstante, cuando se entrega información veraz y precisa, es un aspecto que debe evitarse. Si las respuestas se formulan con autoridad, es peligroso cuando el usuario no posee el conocimiento suficiente sobre un tema para verificar la información.
- **Sesgos:** Pueden aparecer sesgos culturales, sociales y políticos, ya que son intrínsecos a los datos con los que se entrenan los LLMs y pueden emerger en cualquier momento. Pueden afectar a todos los modelos generativos.
- **Conocimiento limitado:** Los LLM solo tienen conocimiento de noticias y eventos ocurridos hasta la última fecha de sus datos de entrenamiento, de modo que podría decirse que su base de datos está *congelada*. Si se realiza una petición relacionada con algo sobre lo que no se han entrenado, los LLMs tienden a rellenar la información con alucinaciones o, en el mejor de los casos, se niegan a responder.
- **Reproducibilidad:** Capacidad de obtener la misma salida para los mismos datos de entrada. Es crítica, tanto en la evaluación de modelos, como para asegurar control sobre su comportamiento. Pero los LLMs, dada su complejidad y tendencia a la creatividad, no tienen esta propiedad.
- **Falta de explicabilidad:** Debido al gran tamaño y complejidad de estos modelos, los LLMs actúan como cajas negras que esconden el funcionamiento interno. Esto es una gran limitación a la hora de desarrollar técnicas que corrijan comportamientos indeseados.

En muchos sentidos, se podría decir que un modelo del lenguaje se parece a un sistema operativo. Aporta una capa base sobre la que se pueden crear aplicaciones. Al igual que un sistema operativo gestiona los recursos hardware y proporciona servicios para los programas software, los modelos del lenguaje gestionan los recursos lingüísticos y prestan servicios para las diversas tareas de NLP. Usar indicaciones para interactuar con ellos es muy similar a escribir código en un lenguaje ensamblador. Es una interacción de bajo nivel.

Ahora que nos hemos adentrado en el mundo de las limitaciones de los LLMs, veamos cuáles son algunos de los desafíos [3]:

- **Interpretación y explicabilidad:** Los resultados generados por los modelos generalmente carecen de explicaciones sobre cómo se llegó a esa respuesta, lo que dificulta la comprensión y la confianza en las decisiones basadas en estos modelos. Se busca aumentar la transparencia.
- **Personalización y adaptación:** Adaptar los LLM a dominios específicos o a usuarios individuales puede ser complicado, y requiere grandes conjuntos de datos y recursos de entrenamiento.

- **Rendimiento en lenguajes específicos:** Algunos LLM pueden tener un rendimiento inferior en idiomas menos comunes o en lenguajes con estructuras gramaticales más complejas.
- **Max Responses:** Actualmente, el número máximo de respuestas que un modelo del leguaje puede generar en respuesta a una entrada es limitado, al menos hablando desde el punto de vista del rendimiento. Aumentar este aspecto es clave en la escalabilidad de los LLMs.

2.3. Aumentar LLMs con RAG

Para resolver algunos de los problemas y afrontar varios de los desafíos mencionados en la sección anterior, surgió la idea de los sistemas ***RAG* (*Retrieval Augmented Generation*)**. Un **flujo RAG** es una técnica que tiene por objetivo mejorar las prestaciones de los grandes modelos del lenguaje. La idea general es utilizar las capacidades generativas de los LLMs con sistemas de recuperación de la información [6].

Por norma general, la base de datos de los modelos generativos se encuentra *congelada*, es decir, solo poseen conocimiento fechado, a lo sumo, al instante en el que se realizó el entrenamiento. Para aumentar esta base de conocimiento, los sitemas RAGs introducen una fase previa a la generación de la respuesta. En esta etapa se recupera información de una base de datos documental, específica según el entorno donde se esté trabajando, para contextualizar al modelo de manera conveniente para que pueda llevar a cabo una función específica [11].

En la figura 2.2 se muestra una comparación de las diferentes etapas entre una petición realizada a un LLM sin flujo RAG y otra realizada a un sitema RAG.

- **Sin flujo RAG:** Cuando se realiza una petición a un LLM que no tiene la arquitectura RAG, el modelo genera la respuesta únicamente a partir de lo que ha aprendido durante su entrenamiento. Las etapas que sigue el modelo son:

  1. El usuario redacta la consulta (*query*) y la manda al modelo.
  2. El LLM predice la respuesta usando únicamente su conocimiento interno

  Esto es rápido y funciona bien para conocimiento general, pero está limitado porque el modelo no conoce datos recientes ni información específica, lo que puede provocar alucinaciones o inventar respuestas si no se tiene certeza. Además, no hay referencias que permitan verificar de dónde proviene la información.

- **Con flujo RAG:** Cuando se implementa el sistema RAG, se introducen varias etapas antes de generar la respuesta, para recabar información:

  1. El usuario redacta la query y la manda al modelo.
  2. El **motor de búsqueda**, busca documentos relevantes en una **base de datos específica**. Esta base de datos puede estar compuesta por documentos pdf,

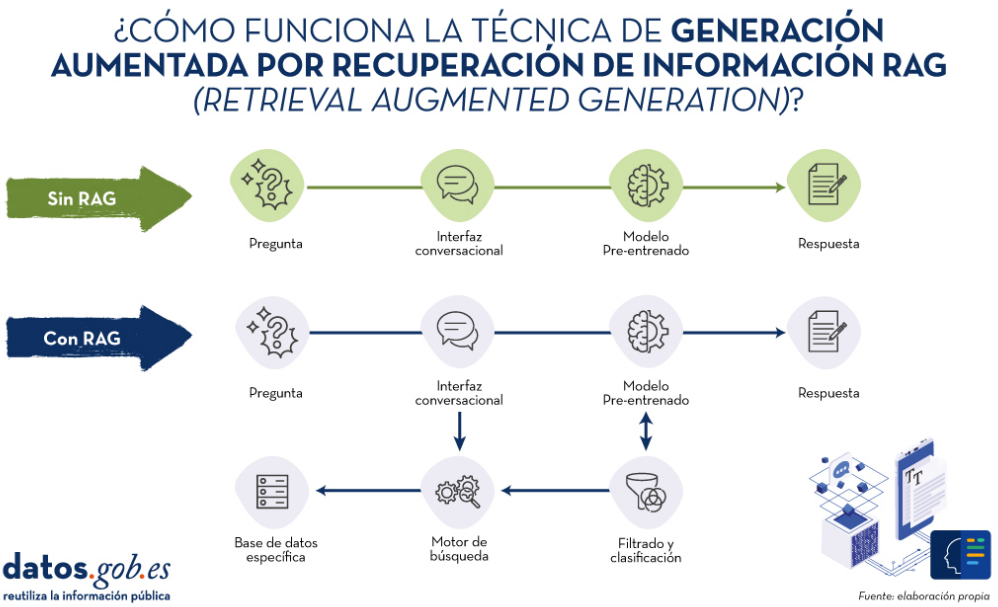

Figura 2.2: Diagrama de comparación entre un LLM sin y con flujo RAG. Fuente [15]

como apuntes de una asignatura, o artículos de investigación; aunque también pueden haber ficheros con otro formato, como páginas web.

3. Se filtran y se recuperan los fragmentos más útiles de los documentos aportados por el motor de búsqueda anterior. De esta forma añadimos el contexto relevante. Se le mandan al LLM utilizando un prompt que combina el contexto con la query del usuario.

4. El modelo genera la respuesta utilizando el contexto recuperado. El flujo RAG puede añadir a la respuesta la indicación de qué contenido de fragmentos y documentos ha utilizado para la creación del prompt.

Esto permite que el modelo acceda a información externa y actualizada, genere respuestas más precisas, contextualizadas y verificables. También aumenta la transparencia del modelo al mostrar las fuentes consultadas

2.4. Fragmentación semántica

Un sistema RAG no trabaja con documentos completos: cuando se recuperan los documentos más relevantes de la base de datos vectorial, se dividen en fragmentos o chunks de menor tamaño. Esto se debe a que las oraciones largas son difíciles para las aplicaciones de NLP. Al igual que un humano, un sistema NLP también debe realizar un seguimiento correcto de todas las dependencias presentadas. El objetivo de la **fragmentación semántica** es buscar fragmentos semánticamente coherentes de una representación de oración. Estos fragmentos se pueden procesar de forma independiente y volver a asociarse como representaciones semánticas sin pérdida de información, interpretación o relevancia semántica. El significado inherente del texto se usa como guía

para el proceso de fragmentación [7] [10].

Al fragmentar los documentos, aumenta la precisión en la búsqueda, ya que si el sistema indexara documentos completos, cualquier petición del usuario resultaría en unidades de texto demasiado grandes, irrelevantes o redundantes. La fragmentación permite justamente recuperar la parte que contiene el contenido útil. Así mismo, los modelos del lenguaje suelen tener una ventana de texto limitada, de modo que el uso de la fragmentación beneficia a la eficiencia en la inferencia del LLM.

Por tanto, podemos decir que la fragmentación semántica es el proceso de dividir el contenido de un documento en fragmentos coherentes y significativos desde el punto de vista del lenguaje y el contenido. No consiste únicamente en cortar por párrafos fijos o cada N palabras o caracteres, sino en hacerlo en lugares donde tiene sentido lingüístico y temático. Para ello, nos apoyaremos en herramientas que hacen uso de algoritmos avanzados que utilizan modelos para detectar fronteras semánticas. En nuestro caso, *LangChain* nos ayudará a realizar esta tarea.

Cada fragmento o *chunk* se transforma en un *embedding* vectorial que representa su significado en un espacio semántico. Luego, cuando el usuario hace una pregunta, esa consulta también se convierte en un embedding, y el sistema busca los chunks más cercanos en ese espacio vectorial, por similitud de coseno u otra métrica.

Por tanto, una buena fragmentación mejora la calidad de los embeddings, lo que a su vez aumenta la precisión del motor de recuperación. Fragmentos mal formados o cortados por el lugar inadecuado degradan el rendimiento al no representar correctamente el contenido.

2.5. Bases de datos vectoriales

Los sistemas RAG requieren un mecanismo de almacenamiento y de búsqueda rápida de los fragmentos de texto más relevantes, y, en el sistema que se propone en este trabajo, esta es la función de *Weaviate*, un ejemplo de base de datos vectorial. Una **base de datos vectorial** es una base de datos que almacena, gestiona e indexa datos vectoriales de alta dimensión. Los puntos de datos se almacenan como matrices de números llamados **embeddings**, la representación vectorial de las palabras como puntos del espacio semántico multidimensional. Los embeddings se agrupan en función de la similitud, permitiendo consultas de baja latencia [8].

A diferencia de las bases de datos tradicionales, las BBDD vectoriales implementan algoritmos para realizar búsquedas eficientes de vecinos más cercanos en el espacio multidimensional. Están optimizadas para realizar búsquedas por similitud, en las que no se busca una coincidencia exacta, sino los vectores más cercanos al vector de consulta basándose en diferentas métricas. Una de las técnicas más habituales para implementar esta búsqueda son los *grafos de mundo pequeño navegables jerárquicos (Hierarchical Navigable Small World Graphs)*, que permiten encontrar rápidamente los vectores más similares a una consulta dada. Esto es crucial para la recuperación de información [13][12].

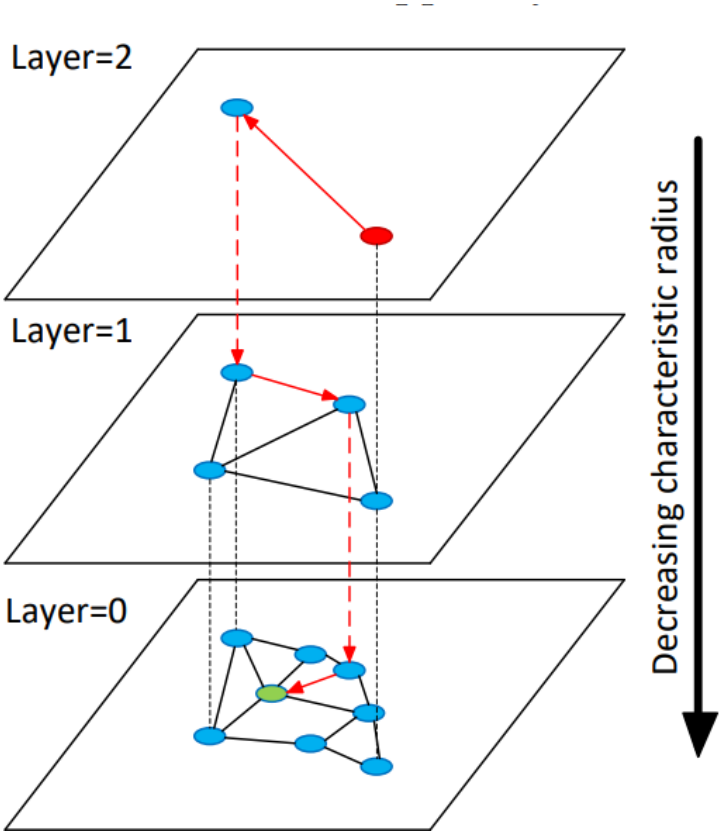

Figura 2.3: Estructura de las capas de un HNSW. Fuente [12]

HNSW se basa en la estructura de los grafos de pequeño mundo, donde los nodos están conectados de forma eficiente, permitiendo recorrer el grafo en pocos saltos para encontrar elementos similares. En este enfoque, los vectores (embeddings) se distribuyen en varios niveles jerárquicos, donde cada nivel funciona como un subgrafo de pequeño mundo. Los niveles más altos contienen menos nodos y están diseñados para facilitar una búsqueda rápida y general antes de refinar la búsqueda en niveles más detallados, como se muestra en la imagen 2.3.

Durante la construcción del índice HNSW, se selecciona un nodo de partida y se generan conexiones hacia otros nodos cercanos en función de su proximidad, calculada mediante una métrica de distancia como la euclidiana o coseno. A medida que se insertan nuevos nodos, se les asigna un nivel jerárquico dentro del grafo: cuanto antes se incorpore un nodo, mayor es la probabilidad de que se ubique en una capa superior, que cubre una mayor parte del espacio vectorial y facilita la navegación eficiente hacia los nodos más relevantes.

A su vez, en cada nivel del grafo, los nodos se unen entre sí de forma que la distancia entre ellos sea la mínima posible, dando lugar a una estructura de grafo de pequeño mundo. Estas conexiones están diseñadas para facilitar una búsqueda rápida y eficaz dentro de la capa. Cuando se inserta un nuevo nodo en el índice, el proceso de incorporación comienza desde el nivel más alto del grafo y desciende capa por capa. En cada paso, se identifican los nodos más cercanos y se establecen enlaces adecuados, hasta que el nuevo nodo queda insertado en la capa inferior, la de mayor resolución.

La búsqueda en un índice HNSW se realiza de forma jerárquica y eficiente. El proceso comienza en la capa más alta del grafo, donde hay pocos nodos, y se selecciona un punto de partida. A partir de ahí, el algoritmo navega por los nodos conectados buscando siempre el más cercano al vector de consulta, según una métrica de distancia (como la euclidiana o coseno). Una vez que no se encuentra un nodo mejor en esa capa, se desciende a la siguiente, repitiendo el proceso hasta llegar a la capa más baja, donde se encuentran todos los nodos.

En ese nivel final, se hace una búsqueda más detallada sobre un subconjunto de candidatos previamente identificados durante el descenso. Finalmente, se devuelven los k vecinos más cercanos al vector de consulta. Esta estructura permite una búsqueda muy rápida sin sacrificar demasiado la precisión, lo que convierte a HNSW en uno de los algoritmos más eficientes para recuperación semántica en grandes volúmenes de datos.

2.6. Prompting

El avance de los modelos de lenguaje de gran escala ha transformado la forma en que las máquinas procesan e interpretan el lenguaje natural. En este contexto, ha cobrado especial importancia el estudio de cómo formular las instrucciones que guían el comportamiento del modelo en tareas específicas. A este proceso se le conoce como **prompting**, y se ha convertido en una herramienta clave para aprovechar la capacidad generativa de los modelos sin necesidad de modificarlos internamente. La precisión, claridad y estructura del prompt influyen directamente en la calidad de las respuestas generadas, lo que ha impulsado el desarrollo de técnicas sistemáticas para su diseño.

A menudo, se conoce como **prompt engineering** a la práctica de diseñar, ajustar y optimizar los prompts que se utilizan para interactuar con los LLMs, con el objetivo de maximizar la calidad, relevancia y utilidad de las respuestas generadas. Dado que estos modelos no están programados explícitamente para cada tarea, sino que responden en función del texto que se les proporciona como entrada, el prompt se convierte en la principal herramienta de control sobre su comportamiento [9].

Esta disciplina surge como respuesta a una necesidad práctica: hay modelos que tienen un conocimiento general muy amplio, pero su rendimiento puede variar significativamente en función de cómo se les formule la instrucción. Por ello, se busca desarrollar técnicas sistemáticas para estructurar textualmente las entradas, decidir qué contexto incluir, cómo plantear preguntas, qué formato usar y, en algunos casos, cómo simular comportamientos o roles específicos.

En los sistemas RAG, el prompting también implica cómo hay que incorporar el contexto a la formulación de la pregunta principal. Unas buenas prácticas podrían no solo mejorar la respuesta, sino reducir errores y evitar alucinaciones, consiguiendo que el modelo tenga un mejor rendimiento.

Aunque el prompting se centra principalmente en qué query le llega realmente al LLM, es decir, en cómo se estructura la petición, hay parámetros del LLM que afectan notablemente a cómo responde el modelo. En muchos casos prácticos, como en la

construcción de nuestro RAG, se consideran parte del ajuste fino de la interacción. A continuación, comentamos brevemente cómo funciona uno de estos parámetros, quizás el más importante: la temperatura [6].

La **temperatura** es un parámetro de decodificación que controla el grado de aleatoriedad en las respuestas generadas por un modelo de lenguaje. Su valor suele oscilar entre 0 y 1. Con un valor bajo, el modelo se comporta de manera más determinista, y tiende a elegir las palabras más probables, consiguiendo respuestas precisas, conservadoras y repetitivas si se realiza la misma consulta. En cambio, con una temperatura alta, el modelo es más creativo e impredecible, pudiendo generar resultados más diversos, pero también con más posibilidad de error o incoherencia.

CAPÍTULO 3

Análisis de objetivos y metodología

Una vez tratado el estado del arte en el campo de los flujos RAG, realizamos en este capítulo el análisis de los objetivos del mismo, estableciendo los elementos concretos que nos van a permitir llevar a cabo dichos objetivos, así como la metodología de desarrollo y las herramientas aplicadas.

3.1. Objetivos

Este trabajo tiene como meta principal desarrollar un programa de tipo *chatbot*, basado en un flujo RAG en el que el conocimiento externo estará formado por los apuntes de la asignatura *Autómatas y Lenguajes Formales*. Por ello, esta aplicación actuará como un agente que permitirá a los estudiantes realizar consultas sobre conceptos y cuestiones de la asignatura. Para lograrlo, se establecieron los siguientes objetivos específicos:

- **Conocimiento profundo de sistemas RAGs y desarrollo de la aplicación**.

  Este objetivo se desglosa en las siguientes tareas:

  - **Diseñar un flujo RAG para nuestra aplicación**, e implementarlo en Python a través de *LlamaIndex*, un marco de código abierto para la ingesta, indexación y recuperación de datos. Dentro del RAG se incorpora *ChatGPT* para generar las respuestas.
  - **Integrar la persistencia en nuestro sistema**, a través de la base de datos vectorial *Weaviate*. Se procederá a implementar una aplicación que permita la ingesta de documentos.
  - **Construir otro programa para realizar las peticiones**, que haga uso de la persistencia. Este programa solo recupera la información cargada previamente en la base de datos y construye un motor para generar las respuestas a las cuestiones que hace el usuario.

- **Valoración de la capacidad de respuesta**.

  Este objetivo se desglosa en las siguientes tareas:

  - **Saber evaluar y mejorar un sistema RAG**. Es decir, no solo construirlo, sino también entender cómo medir su efectividad, la precisión de recuperación, calidad de la generación, y cómo optimizarlo para tareas más complejas o dominios específicos.
  - **Establecer vías futuras** sobre los flujos RAG. Proponer algún posible caso de uso en algún campo en el que utilizar este método supondría una mejora inmediata.

3.2. Herramientas

Para la elaboración de este proyecto se han utilizado diversas herramientas. A continuación se muestra un listado de todas ellas junto con una breve descripción:

- **Servidor con GPU:** Un equipo de la Facultad de Informática de la Universidad de Murcia, que tiene una GPU con una memoria de 40 GB (NVIDIA A100) que permitó ejecutar pruebas con LLMs de libre distribución en combinación con Ollama.
- **Editor de código remoto:** V*isual Studio Code* con la extensión *Remote - SSH* para poder editar código directamente en el servidor GPU.
- ***Python*:** Lenguaje de programación de alto nivel interpretado en el que se han realizado todos los programas de prueba y se han implementado los programas principales del proyecto.
- **Entorno virtual de *Python*:** Se crearon (y destruyeron) varios entornos virtuales de *Python* para instalar y gestionar las librerías y dependencias necesarias de forma independiente. De este modo se consiguió aislar el trabajo, lo que otorgó robustez al proyecto y facilitó su gestión.
- ***LlamaIndex*:** Se trata de una herramienta diseñada para ayudar a integrar grandes modelos de lenguaje con diferentes conjuntos de datos de manera más eficiente, especialmente útil en sistemas RAG. Su propósito principal es facilitar la organización, el acceso y la recuperación de información de bases de datos, documentos, APIs u otras fuentes, para luego usarlas como contexto adicional al generar respuestas con un modelo de lenguaje. Todo el desarrollo de este proyecto está basado en este framework.
- ***Langchain*:** Biblioteca de desarrollo diseñada para crear aplicaciones impulsadas por LLMs. En este caso, proporciona herramientas para preparar los datos para que el sistema RAG pueda entenderlos, ayudando a la hora de dividir de forma inteligente documentos grandes de datos, como PDFs o artículos web.
- ***Weaviate y WeaviateCloud*:** *Weaviate* es una base de datos vectorial de código abierto diseñada para almacenar, buscar y gestionar datos representados como vec-

tores, que son ideales para tareas como recuperación semántica, sistemas RAG y motores de búsqueda inteligentes. Permite hacer consultas basadas en significado (no solo en palabras clave) de forma rápida y eficiente. *WeaviateCloud* es el servicio online que ofrece el uso de la base de datos vectorial sin necesidad de instalar ni mantener servidores.

- ***Ollama y phi4*:** Herramienta de código abierto que permite ejecutar grandes modelos de lenguaje localmente en un equipo sin necesidad de conexión a internet. En este caso, inicialmente se eligió *phi4* como generador de respuestas para nuestra aplicación.
- ***OpenAI y GPT*:** Es una empresa estadounidense de investigación y despliegue de inteligencia artificial. Tiene diversos modelos de código abierto, como *GPT*, que genera las respuestas de nuestro programa.
- ***Trulens*:** Software que ayudará a medir objetivamente la calidad y la eficacia de aplicaciones basadas en LLM mediante *feedback*. Estas funciones permiten evaluar automáticamente la calidad de las entradas, salidas y resultados intermedios, lo que permite agilizar y ampliar la evaluación de experimentos de estos sistemas.

3.3. Metodología

En cuanto a la metodología, durante todo el desarrollo de este trabajo se han ido manteniendo reuniones periódicas con el tutor. En dichas reuniones se discutieron, revisaron y se resolvieron cuestiones y problemas. Fueron programadas cada tres o cuatro semanas, y permitían valorar el adecuado progreso del proyecto. A continuación se muestran todas las reuniones que se mantuvieron y en qué consistía cada una de ellas:

- **14/10/2024:** Primera reunión del proyecto. Se sugirieron varias propuestas para el Trabajo de Fin de Grado y se optó por realizar este proyecto. Se acordó la lectura del libro *Building Data-Driven Applications with LlamaIndex* [6].
- **10/11/2024:** En esta cita se repasaron algunos aspectos teóricos del libro citado en el apartado anterior, y se explicó detalladamente la metodología. También se concretó y se preparó el entorno de trabajo, para poder emplear un servidor con GPU de la Facultad de Informática de la Universidad de Murcia. Además de todo esto, se siguió con la lectura del libro y con pequeñas pruebas de código.
- **29/01/2025:** Este día se resolvieron algunos problemas que surgieron durante la implementación del primer prototipo, y se acordó usar el modelo *phi4* [1] en *Ollama* para generar las respuestas, un LLM gratuito que quedó instalado en el servidor GPU. Se discutieron también algunas dudas del libro.
- **12/02/2025:** La reunión tuvo como principal propósito resolver algunos errores que se producían en el programa, relacionados con el enlazamiento entre el framework *LlamaIndex* y el modelo *phi4* de *Ollama*. Se siguieron resolviendo algunas dudas sobre algunos conceptos del libro y se esbozó el primer diseño de la aplicación del proyecto.

- **26/02/2025:** Antes de que esta reunión tuviera lugar, se completó la lectura del libro de referencia. Por ello, se resolvieron algunas dudas y se estableció el esquema final de la aplicación. Se propuso el uso de la base de datos vectorial *Weaviate* para implementar la persistencia. Se volvió a instalar *Ollama*, esta vez en el equipo local, porque el servidor GPU tenía limitaciones de espacio en disco. Se comenzó la implementación de dos programas, uno para la ingesta de documentos y otro para realizar las consultas.
- **27/03/2025:** Debido a la lentitud de ejecución de *phi4* en *Ollama* en local, se constató que esta parte del sistema continuaba trabando el desarrollo del proyecto, de modo que se empezó a trabajar con una clave de *OpenAI*, lo cual resultaba mucho más cómodo porque inicialmente *LlamaIndex* se diseñó para los LLMs de OpenAI. Además, se produjo un fallo de sistema en el servidor GPU que lo dejó inoperativo durante varias semanas, y se decidió finalmente trabajar en el desarrollo en el equipo local consultando los modelos de OpenAI. A esta reunión se llegó con el programa para la ingesta de documentos semi-implementado. Se trataron los errores, y se comprobó el correcto funcionamiento de *Weaviate Cloud*, la base de datos vectorial que hemos mencionado antes, pero ahora en la nube. Esto facilitaría un futuro despliegue de la aplicación.
- **10/04/2025:** En esta reunión se comprobó el correcto funcionamiento del programa que se encarga de las consultas a las bases de datos. Tras el éxito de esta prueba, dio comienzo a la redacción de esta documentación.
- **14/04/2025:** Esta reunión sirvió para resolver dudas relacionadas con el formato de esta documentación, y se comenzó a incorporar *TruLens* para poder evaluar el sistema RAG.
- **21/05/2025:** Última reunión del proyecto. Se ultimaron los detalles finales y se perfiló la documentación para su posterior entrega.

CAPÍTULO 4

Diseño y resolución

En este capítulo se realiza la descripción de la implementación de la aplicación: un sistema RAG para permitir a los alumnos de la asignatura Autómatas y Lenguajes Formales realizar consultas para resolver dudas. El apartado 4.1 contiene la explicación para instalar todas las librerías necesarias. En el apartado 4.2 se muestra cómo se crea un cluster en *Weaviate* para almacenar todos los fragmentos de los documentos. En el apartado 4.3 se detalla el proceso de indexación de los documentos PDF que contienen los apuntes de la asignatura. La sección 4.4 se explica cómo funciona la recuperación de fragmentos de texto relevantes, y en el apartado 4.6 se detalla el del prompting en el sistema. Por último, en el apartado 4.5 se evalúa el modelo usando la herramienta *Truelens*.

4.1. Instalación de librerías

En esta sección se especifican todas las librerías que se han instalado para el correcto funcionamiento del sistema RAG. Son necesarias para incorporar la mayoría de herramientas citadas en el apartado 3.2.

Como punto de partida, crearemos un entorno virtual en *Python* para aislar el proyecto del resto de programas que podamos tener en nuestro equipo. A continuación, se muestran los comandos para la creación de nuestro entorno virtual y su posterior activación:

```
python -m venv tu_entorno_virtual
ruta_de\tu_entorno_virtual\Scripts\activate
```

Listing 4.1: Creación y activación de entorno virtual en Windows

```
python -m venv tu_entorno_virtual
source ruta_de/tu_entorno_virtual/bin/activate
```

Listing 4.2: Creación y activación de entorno virtual en Mac y Linux

El primer paquete que debemos instalar es *LlamaIndex*, la herramienta principal que nos permitirá integrar LLMs en el sistema RAG. Para ello, tendremos que ejecutar el siguiente comando, una vez activo nuestro entorno virtual. Esto incluirá los componentes del núcleo de LlamaIndex y una selección de integraciones útiles.

```
pip install llama-index llama-index-llms-openai
```

Para que nuestro proyecto sea capaz de trabajar con ficheros PDF desde *Python*, debemos utilizar *pypdf*.

```
pip install pypdf
```

También tendremos que instalar el paquete de *Weaviate* para poder conectarnos a la base de datos en la nube con la finalidad de subir documentos que el sistema empleará para generar el contexto de las consultas. Así mismo, instalamos la librería necesaria para que *LlamaIndex* pueda emplear *Weaviate* como almacén de datos.

```
pip install weaviate llama-index-vector-stores-weaviate
```

Por último, para poder evaluar el sistema, usaremos TruLens. Para ello, debemos instalar las siguientes librerías.

```
pip install trulens trulens-providers-openai chromadb openai
```

4.2. Creación de un cluster en *Weaviate*

Recordamos que la base de datos vectorial que usamos para almacenar los diferentes trozos de información es *Weaviate*. En vez de instalar propiamente la base de datos en nuestro equipo, nos apoyeramos en *Weaviate Cloud*, que ofrece las mismas prestaciones que *Weaviate*, pero nos ahorra el trabajo que supondría instalarlo y ponerlo en marcha.

Para usar *Weaviate Cloud*, lo primero que debemos hacer es crear una cuenta en el siguiente enlace: crear cuenta en *Weaviate Cloud*. Una vez hayamos iniciado sesión, accederemos a una página como la mostrada en la imagen 4.1.

Se puede apreciar que, inicialmente, no existe ningún cluster. Para crear un cluster pulsaremos el botón blanco que aparece en la segunda columna de la izquierda, *Create cluster*. Tras esto, se accede a una nueva página como la mostrada en la figura 4.2.

Vemos que hay multitud de configuraciones disponibles. Dejaremos todas las opciones por defecto, aunque pueden ajustarse al gusto del usuario. Para el proyecto se ha elegido un *Sandbox Cluster*. Esta es la versión gratuita de un cluster en *Weaviate Cloud*. Como inconveniente principal destacamos que expira a los 14 días de la fecha de creación y que solo tiene 20 GB de almacenamiento. No obstante, para el alcance de este proyecto es suficiente. Por último, escribimos un nombre para nuestro cluster y pulsamos el botón *Create*.

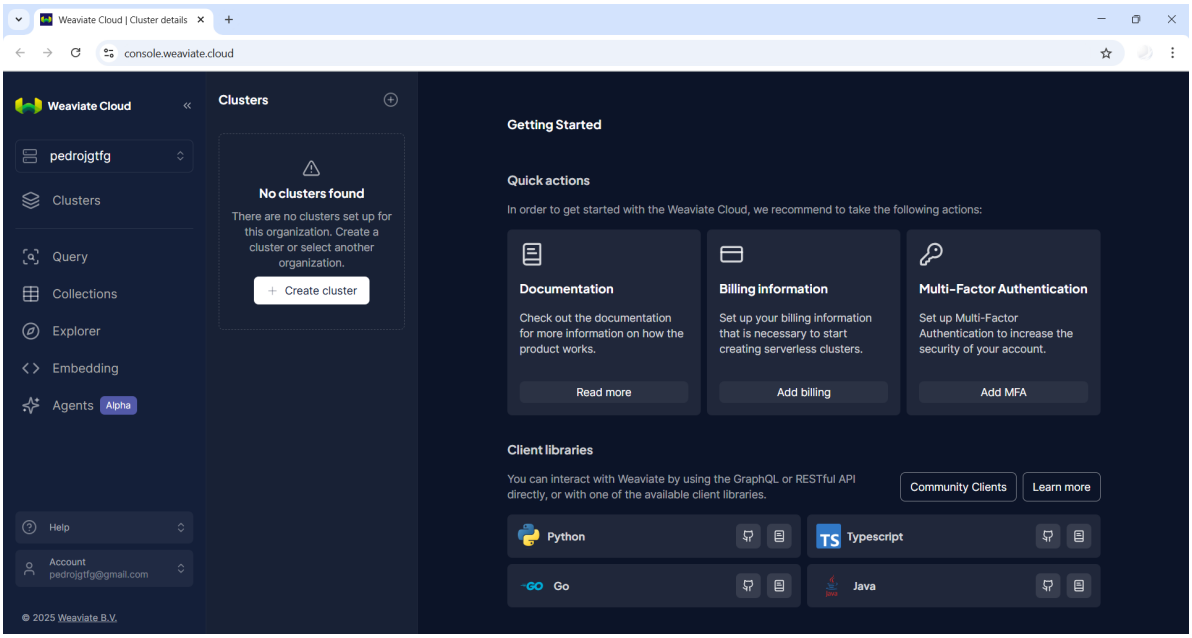

Figura 4.1: Pantalla que se muestra una vez se ha iniciado sesión en *Weaviate Cloud*

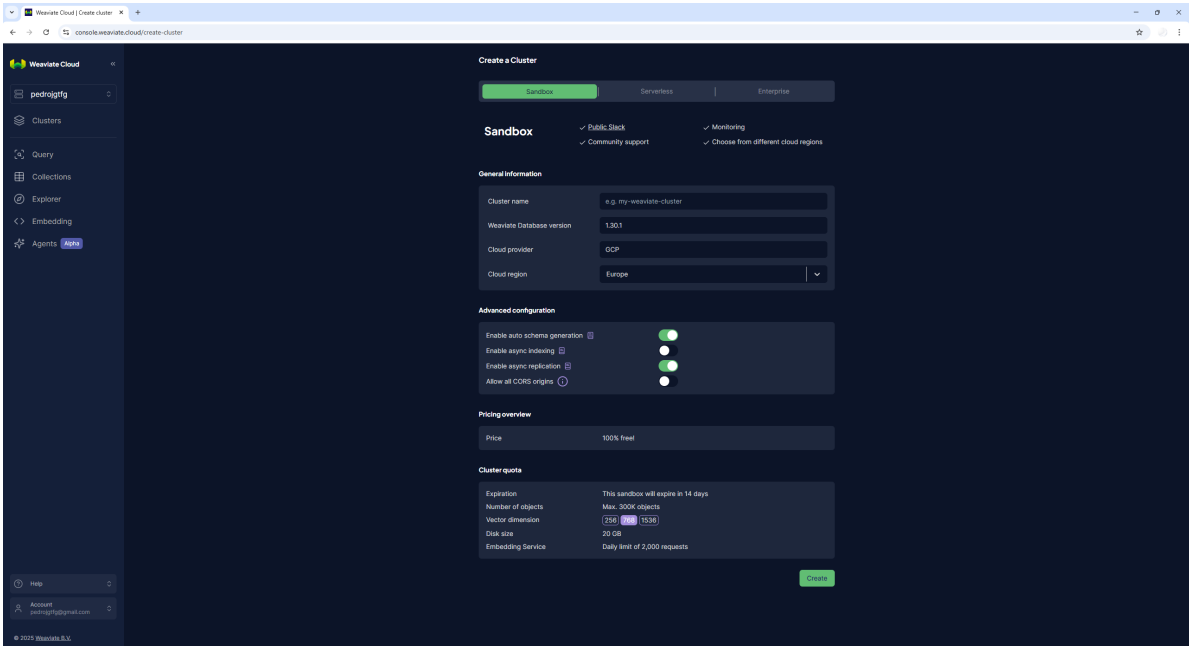

Figura 4.2: Pantalla de creación de cluster en *Weaviate Cloud*

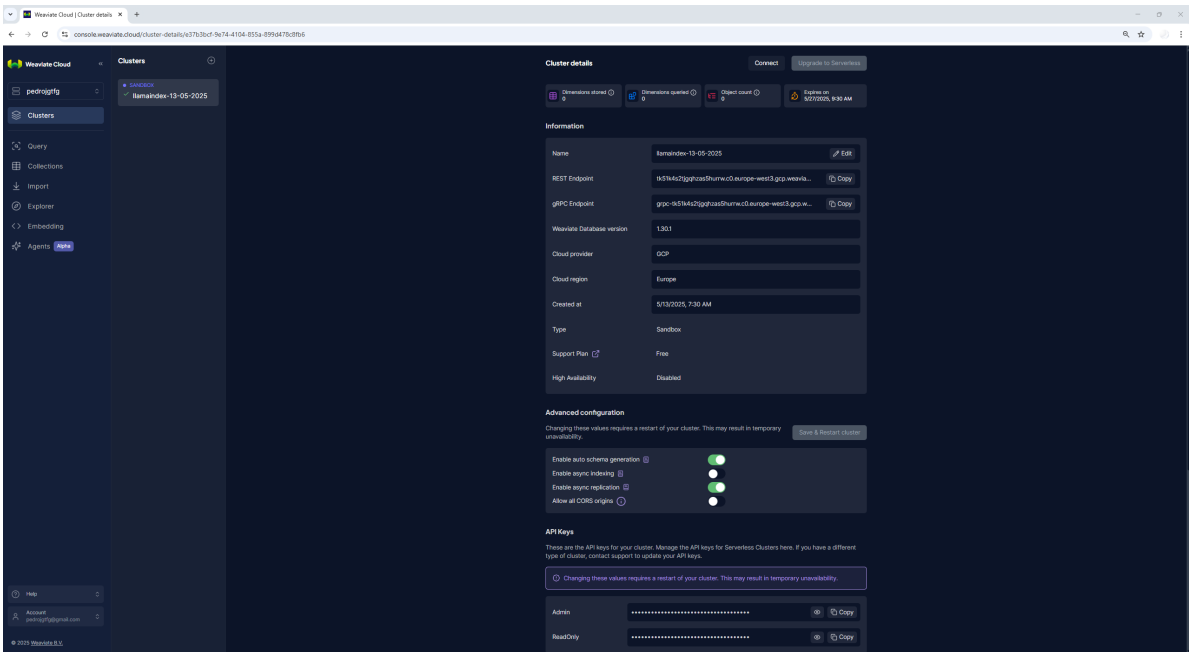

Figura 4.3: Pantalla de gestión de cluster en *Weaviate Cloud*

Al completarse el proceso de creación, veremos una página como la que se muestra en la imagen 4.3. Se trata de la pantalla de gestión del cluster. En las posteriores secciones se mostrará cómo se puede emplear el cluster en nuestro sistema.

4.3. Indexación de documentos PDF

Una vez preparado el cluster, estamos en condiciones de escribir el programa que nos permitirá fragmentar los documentos PDF y almacenarlos en la base de datos vectorial. Esto nos permitirá añadir los documentos necesarios para proveer el contexto adecuado a las consultas de nuestro sistema.

Antes de comenzar a desarrollar las primeras líneas de código, tendremos que habilitar un directorio con los documentos PDF que queremos añadir a la base de datos documental de nuestra aplicación. Tendremos que indicar la ruta en nuestro programa para que pueda leer los archivos.

A continuación, se muestra el código del programa que realiza la indexación de los documentos PDF:

```
import logging
import os
logging.basicConfig(level=logging.DEBUG)

from llama_index.core.node_parser import SentenceSplitter

from llama_index.core import SimpleDirectoryReader,
            VectorStoreIndex, StorageContext

from llama_index.core.node_parser import LangchainNodeParser
```

```
11
12 import weaviate
13 from llama_index.vector_stores.weaviate import WeaviateVectorStore
14
15 # cloud
16 cluster_url = "url_de_tu_cluster"
17 api_key = "api_key_de_tu_cluster"
18
19 client = weaviate.connect_to_wcs(
20     cluster_url=cluster_url,
21     auth_credentials=weaviate.auth.AuthApiKey(api_key),
22 )
23
24 reader = SimpleDirectoryReader(input_dir = r"ruta\de\tus\documentos")
25 documents = reader.load_data()
26
27 ## LANGCHAIN
28 parser = LangchainNodeParser(SentenceSplitter(chunk_size=1024,
29                                               chunk_overlap=20,))
30 nodes = parser.get_nodes_from_documents(documents)
31
32 ## Para la persistencia
33 vector_store = WeaviateVectorStore(
34     weaviate_client=client, index_name="nombre_de_tu_cluster"
35 )
36 storage_context = StorageContext
37                 .from_defaults(vector_store=vector_store)
38
39 index = VectorStoreIndex(nodes, storage_context=storage_context)
40 client.close()
```

Podemos observar que una de las primeras cosas que hace nuestro programa es intentar conectarse a *Weaviate Cloud*, mediante la creación de un objeto cliente. Los parámetros que recive la función `connect_to_wcs()` del paquete weaviate son la url y la clave de la API de escritura del cluster que hemos creado en la sección 4.2. Debemos darle el valor que se encuentra en *REST Endpoint* y *Admin* de la captura de pantalla 4.3.

Lo siguiente que realiza el programa es utilizar la clase `SimpleDirectoryReader` de *LlamaIndex* para cargar documentos desde la carpeta local y prepararlos para su posterior procesamiento. Esta carpeta local se añade como el parámetro `input_dir` a la hora de llamar la función. La línea 25 carga los documentos del directorio.

Después de esto, llegamos a una de las líneas con mayor importancia: la construcción del `parser` que se encargará de la fragmentación del texto para que sea semánticamente coherente. Vemos que usamos el constructor `LangchainNodeParser`, que crea un objeto que divide documentos en chunks (fragmentos) de hasta 1024 caracteres, con un solapamiento de 20 caracteres entre ellos, y los convierte en *nodos* compatibles con *LlamaIndex*. Cada nodo contiene el texto de un chunck y metadatos relativos a su localización en el documento PDF. Esta configuración, haciendo uso de *Langchain*, permite preparar el texto de forma óptima para su posterior indexación.

A continuación, mediante el método `get_nodes_from_documents()`, el programa crea

los nodos a partir de la lista de documentos cargados previamente, dividiéndolos según el parser que hemos creado. Estos nodos ya están listos para ser indexados y almacenados en la base de datos.

La parte final del programa se encarga de la persistencia en *Weaviate Cloud*. Lo primero que se hace es crear un objeto de conexión entre *LlamaIndex* y el objeto cliente que habíamos instanciado. Los parámetros que recibe el constructor `WeaviateVectorStore()` son precisamente el cliente y una cadena de texto con el nombre del cluster, donde se almacenarán los nodos.

La línea siguiente crea un contexto de almacenamiento personalizado, que indica que los nodos y sus embeddings deben guardarse en *Weaviate* en lugar de quedarse en memoria. Para ello se da valor al parámetro `vector_store` asignando el objeto que hemos indicado antes. Además de esto, es necesario crear el índice vectorial que contendrá los nodos generados a partir de los documentos. Este índice queda guardado en *Weaviate Cloud* usando el `storage_context` que hemos comentado antes. Por último, se cierra la conexión del cliente de *Weaviate*.

4.4. Recuperación de fragmentos

Una vez terminado todo el proceso de indexación de chunks y su posterior almacenamiento, es necesario indicar cómo se va a recuperar la información. Al igual que en la sección anterior, en esta sección se comentarán los aspectos más relevantes del código dedicado a la recuperación de fragmentos de *Weaviate*.

```
import logging
import os
logging.basicConfig(level=logging.DEBUG)

import weaviate
from llama_index.vector_stores.weaviate import WeaviateVectorStore

client = weaviate.connect_to_wcs(
    cluster_url="url_de_tu_cluster",
    auth_credentials =
        weaviate.auth.AuthApiKey("api_key_de_tu_cluster"),
)


from llama_index.llms.openai import OpenAI
from llama_index.core.settings import Settings
os.environ["OPENAI_API_KEY"] = "tu_openai_api_key"
Settings.llm = OpenAI(model="gpt-4o-mini")
from llama_index.core import VectorStoreIndex

vector_store = WeaviateVectorStore(
    weaviate_client=client, index_name="nombre_de_tu_cluster"
)
loaded_index = VectorStoreIndex.from_vector_store(vector_store)

```

```
26
27 query_engine = loaded_index.as_query_engine(streaming=True,
28                                              similarity_top_k=3)
29
30 print("Ask me anything about ALF!")
31 while True:
32     question = input("Your question: \n")
33     if question.lower() == "exit":
34         break
35     response = query_engine.query(question)
36     print("- "*20)
37     print(len(response.source_nodes))
38     print("- "*20)
39     for node in response.source_nodes:
40         metadata = node.metadata
41         file_name = metadata.get("file_name", "Desconocido")
42         page_label = metadata.get("page_label", "N/A")
43         print(f"Documento: {file_name}, Pagina: {page_label}")
44     response.print_response_stream()
45
46 client.close()
```

Como se puede observar, lo primero que hace nuestro código, tras importar los módulos necesarios, es crear el objeto cliente de *Weaviate Cloud*. Al igual que antes, se usa la función `connect_to_wcs()`, que nos dará la instancia que nos permitirá la conexión a la nube. En este caso, los parámetros que recibe este constructor son las cadenas de caracteres que se encuentran en el *REST Endpoint* y el *ReadOnly* de la pantalla de gestión de nuestro cluster, tal y como se muestra en la figura 4.3.

Después de esto, lo siguiente que hace el programa es configurar el modelo del lenguaje que vamos a utilizar. En este caso, inicialmente se realizaron una serie de pruebas con algunos modelos de *Ollama*, pero se acabó descartando su uso porque no se conseguía un buen resultado cuando se intentaba incorporar *Weaviate* a la aplicación. Eran modelos que se instalaban en el equipo y se ejecutaban de forma local, lo que provocaba que tardara muchísimo en generar respuestas.

Finalmente, se optó por usar la API de *OpenAI*, porque *LLamaIndex* fue inicialmente diseñado para implementar flujos RAG con modelos de esta empresa. El modelo usado en el código es *gpt-4o-mini* (línea 18), pero puede emplearse cualquier otro modelo de OpenAI. En la sección 4.5 se muestra una comparativa de rendimiento entre varios modelos. En el código se muestra también cómo añadir la clave de la API, para poder realizar las consultas.

A continuación se pasa a crear el objeto de conexión entre *LlamaIndex* y *Weaviate*. Para crearlo, hay que indicar como parámetros, al constructor `WeaviateVectorStore()`, el cliente que habíamos instanciado previamente y una cadena de texto con el nombre del cluster en el que están almacenados los nodos.

Seguidamente se hace uso de la función `from_vector_store()`, que forma parte de la clase `VectorStoreIndex`. Con esto conseguimos recuperar el índice de los nodos que tenemos almacenados en *Weaviate Cloud*. Por tanto, la fragmentación de los documentos

no es necesario repetirla salvo que se modifique la base de datos documental con nuevos archivos.

La línea siguiente (27) se encarga de construir el `query_engine`, el motor de consulta, que nos permite en nuestro sistema RAG, recuperar los tres documentos más relevantes en cada petición (parámetro `similarity_top_k`), y que la respuesta se obtenga en tiempo real, es decir, en forma de *streaming* o flujo de tokens desde el modelo de OpenAI hacia nuestro cliente.

Tras lo anterior se encuentra el bucle principal de ejecución del código. Primero, se le pide al usuario que realice una pregunta relacionada con el temario de ALF. Si la pregunta corresponde a la cadena `exit`, se termina la ejecución del bucle. En caso contrario, se envía la consulta al `query_engine`, que es el motor de búsqueda de Weaviate.

Éste busca los fragmentos más relevantes en el índice vectorial, genera la respuesta usando esos fragmentos como contexto, y la devuelve junto con los nodos que se han utilizado. Luego, se muestra el número de nodos recuperados y se imprimen por pantalla el documento y la página de la que provienen los nodos utilizados, por si el usuario tiene interés en seguir la lectura de los apuntes. Y, como última instrucción del bucle, se imprime la respuesta. Por último, al finalizar el bucle se cierra la conexión con *Weaviate Cloud*.

4.5. Evaluación

Para poder evaluar el sistema, nos hemos apoyado en *TruLens* [18]. Se trata de una herramienta que sirve para evaluar automáticamente la calidad de las respuestas generadas por el sistema RAG.

Tradicionalmente, la evaluación de un sistema de pregunta/respuesta, como es un RAG, se basaría en un dataset de comprobación con el que se tendría una batería de preguntas con la respuesta que el sistema debería generar (*groundtruth*). Sin embargo, la creación de un dataset de validación es extremadamente costoso, puesto que tiene que ser generado por humanos y debe contar con una cantidad suficiente de preguntas/respuestas para poder evaluar adecuadamente el sistema.

TruLens se apoya en otra idea, que se podría resumir de la siguiente forma: evalúar aplicaciones basadas en LLM, como un sistema RAG, utilizando *otro modelo LLM como evaluador*, para comprobar automáticamente la calidad de las respuestas generadas.

Para ello, TruLens intercepta la consulta del usuario, los documentos recuperados de la base de datos Weaviate y la respuesta final generada por el LLM del RAG. Esto lo realiza con una serie de funciones *feedback* que se inyectan en los tres puntos clave del funcionamiento del RAG.

Con la información capturada, TruLens realiza una serie de consultas al LLM evaluador mediante prompts que permiten obtener unas métricas que determinan si las respuestas son relevantes, fundamentadas y coherentes con el contexto recuperado. Los

resultados de la evaluación pueden visualizarse en un dashboard interactivo, como se muestra al final de esta sección.

A continuación se muestran y explican fragmentos del código que se encarga de la evaluación del sistema RAG.

```
import os
import weaviate
from weaviate.classes.init import Auth
from llama_index.vector_stores.weaviate import WeaviateVectorStore
from llama_index.core import VectorStoreIndex

os.environ["OPENAI_API_KEY"] = "tu_openai_api_key"
cluster_url="url_de_tu_cluster"
api_key=Auth.api_key("api_key_de_tu_cluster")

client = weaviate.connect_to_weaviate_cloud(
    cluster_url=cluster_url,
    auth_credentials=api_key
)

vector_store = WeaviateVectorStore(
    weaviate_client=client, index_name="nombre_de_tu_cluster"
)

loaded_index = VectorStoreIndex.from_vector_store(vector_store)

query_engine = loaded_index.as_query_engine(similarity_top_k=4)
```

En primer lugar, el script comienza conectándose a *Weaviate Cloud*, para recuperar los índices y crear el motor de consulta, con el parámetro `similarity_top_k=4`, que indica que se recuperarán los cuatro fragmentos de texto más relevantes, basándose en la similitud semántica con la pregunta del usuario. Esto también se hacía en la sección 4.4.

```
from trulens.apps.app import instrument
from trulens.core import TruSession

session = TruSession()
session.reset_database()

from openai import OpenAI
oai_client = OpenAI()

class RAG:
    @instrument
    def retrieve(self, query: str) -> list:
        results = query_engine.query(query)
        return [node.node.get_text() for node in results.source_nodes]

    @instrument
    def generate_completion(self, query: str,
                            context_str: list) -> str:
        if len(context_str) == 0:
```

```
            return "Sorry, I couldn't find an
                            answer to your question."

        completion = (
            oai_client.chat.completions.create(
                model="gpt-4o-mini",
                temperature=0,
                messages=[
                    {
                        "role": "user",
                        "content": f"We have provided context
                                    information below. \n"
                        f"---------------------\n"
                        f"{context_str}"
                        f"\n---------------------\n"
                        f"First, say hello and that you're
                                    happy to help. \n"
                        f"\n---------------------\n"
                        f"Then, given this information, please
                                    answer the question: {query}",
                    }
                ],
            )
            .choices[0]
            .message.content
        )
        if completion:
            return completion
        else:
            return "Did not find an answer."

    @instrument
    def query(self, query: str) -> str:
        context_str = self.retrieve(query=query)
        completion = self.generate_completion(
            query=query, context_str=context_str
        )
        return completion

rag = RAG()
```

Después, se define la clase RAG, que contiene la lógica central del sistema. Esta clase define tres métodos decorados con @instrument, lo que permite que se pueda analizar el flujo interno de la consulta. Los métodos son los siguientes:

- `retrieve()`: Busca los fragmentos más relevantes entre los índices recuperados, basándose en la similitud semántica a la consulta del usuario. Esta función contiene un código similar al descrito en la sección anterior.

- `generate_completion()`: Genera la respuesta usando el modelo de *OpenAI* que hemos usado también antes, *gpt-4o-mini*. Vemos que aquí podemos definir una plantilla para el prompt que mandará al modelo. En la sección siguiente comentaremos un poco más este aspecto.

- `query()`: consiste en combinar los dos métodos anteriores en uno solo.

```
import numpy as np
from trulens.core import Feedback
from trulens.core import Select
from trulens.providers.openai import OpenAI

provider = OpenAI(model_engine="gpt-4.1-mini")

# Define a groundedness feedback function
f_groundedness = (
    Feedback(
        provider.groundedness_measure_with_cot_reasons,
        name="Groundedness"
    )
    .on(Select.RecordCalls.retrieve.rets.collect())
    .on_output()
)
# Question/answer relevance between overall question and answer.
f_answer_relevance = (
    Feedback(
        provider.relevance_with_cot_reasons,
        name="Answer Relevance"
    )
    .on_input()
    .on_output()
)

# Context relevance between question and each context chunk.
f_context_relevance = (
    Feedback(
        provider.context_relevance_with_cot_reasons,
        name="Context Relevance"
    )
    .on_input()
    .on(Select.RecordCalls.retrieve.rets[:])
    .aggregate(np.mean)
)
```

Tras esto, se configura *TruLens* para que sea capaz de evaluar las respuestas generadas. Para ello, el sistema define varias funciones de evaluación, ya mencionadas antes, conocidas como funciones *feedback*, que permiten evaluar distintos criterios de calidad:

- **Groundedness:** Evalúa si la respuesta generada por el modelo está basada en los fragmentos que se han recuperado desde la BBDD vectorial. Esta métrica nos permite ver cuánto ha alucinado el modelo. La alucinación es mayor si la medida es baja.
- **Answer Relevance:** Mide cuánto de buena es la respuesta a la pregunta planteada por el usuario. Aquí se analiza si la respuesta es útil, coherente y si responde a la pregunta original del usuario.
- **Context Relevance:** Analiza la calidad de los índices recuperados, es decir, si los

fragmentos seleccionados como contexto están relacionados con la pregunta. En este caso, se evalúa el valor medio de la relevancia de todos los fragmentos.

Además, se prepara el LLM evaluador, denominado *provider* en la terminología de TruLens. En el código anterior se emplea el modelo gpt-4.1-mini como evaluador.

```
from trulens.apps.app import TruApp
tru_rag = TruApp(
    rag,
    app_name="LlamaIndex_Weaviate_RAG",
    app_version="base",
    feedbacks=[f_groundedness,f_answer_relevance,f_context_relevance]
)

with tru_rag as recording:
    rag.query(
        "¿Qué es un autómata finito determinista?"
    )
    rag.query(
        '¿Qué es un autómata de pila?'
    )
    rag.query(
        "¿Qué es una gramática libre de contexto"
    )
    rag.query(
        "¿Me puedes dar un ejemplo de una gramática ambigua?"
    )
    rag.query(
        "¿Cómo se puede demostrar la equivalencia descriptiva de las
         expresiones regulares y los autómatas finitos?"
    )
    rag.query(
        "¿Puedes describir el funcionamiento del algoritmo de eliminación
         de símbolos inútiles de una gramática?"
    )
    rag.query(
        "¿Puedes describir el funcionamiento del algoritmo de eliminación
         de reglas lambda de una gramática?"
    )
    rag.query(
        "¿Puedes describir el funcionamiento del algoritmo de eliminación
         de reglas unitarias de una gramática?"
    )
    rag.query(
        "¿Qué es una derivación más a la izquierda de una forma
         sentencial?"
    )
    rag.query(
        "¿Qué dice el teorema de Kleene?"
    )

from trulens.dashboard import run_dashboard
run_dashboard(session)
```

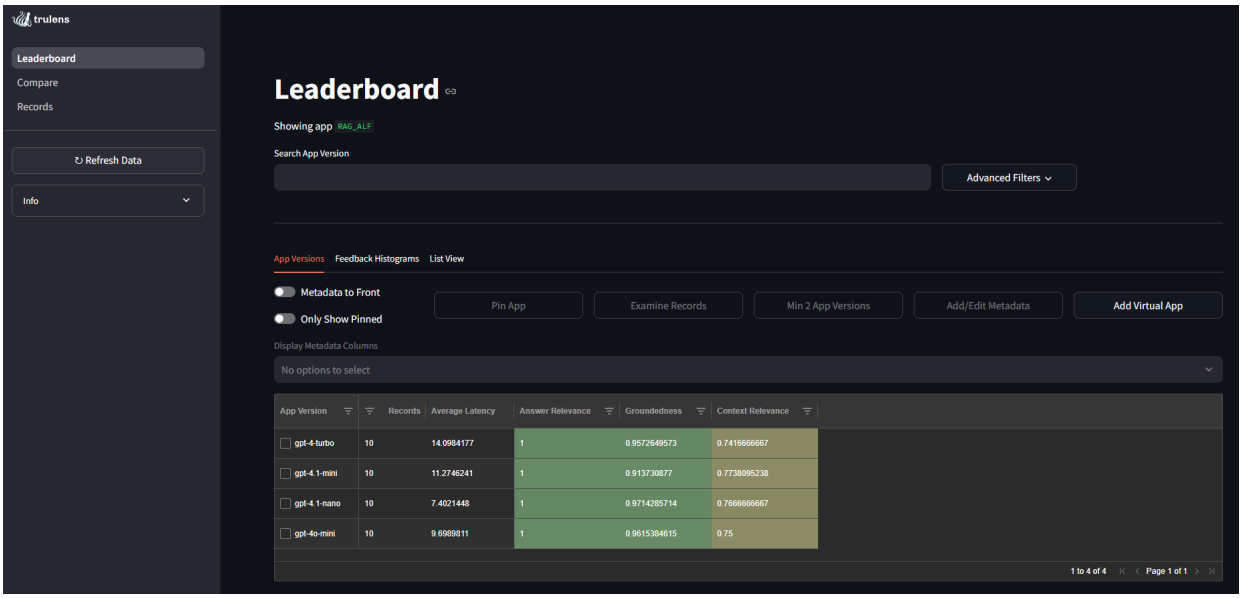

Figura 4.4: *Leaderboard* de *TruLens*

Este enfoque permite llevar a cabo mediciones objetivas y consistentes. Los resultados obtenidos se han calculado tras realizar una batería de preguntas de ALF, y para cada una de ellas, genera una respuesta y registra el proceso para poder llevar a cabo la evaluación. Los resultados se muestran en un dashboard interactivo, que permite explorar de forma visual y detallada cada consulta, la respuesta generada y las puntuaciones de la evaluación. Más adelante se detalla la explicación de este dashboard.

De este modo, el sistema de evaluación no solo facilita detectar errores o inconsistencias, sino que ofrece información valiosa para optimizar la recuperación de contenido, ajustar el diseño de los prompts o incluso ver el efecto de diferentes fragmentaciones.

```
client.close()
```

Para finalizar, lo último que hace el código es cerrar la conexión con Weaviate.

El *dashboard* que nos ofrece Trulens incluye diferentes paneles interactivos a través de los cuales podemos visualizar los resultados. Explicamos algunos de los más relevantes:

- ***Leaderboard* (Figura 4.4).** Muestra una visión general de los registros de ejecución disponibles.
- **Comparación (Figura 4.5).** Una de las funcionalidades de TruLens consiste en que, al lanzar la ejecución de la aplicación, puede ser etiquetada con una versión.
- **Registros (Figura 4.6).** De cada consulta, TruLens recoge la siguiente información: *Total tokens*, *Total cost*, *Latency*, *Answer Relevance*, *Context Relevance*, *Groundedness*, *Input* (la consulta) y *Output* (la salida del RAG).
- **Explicación (Figura 4.7).** También se puede acceder al resultado de la evaluación de los 4 fragmentos del contexto que se ha recuperado desde *Weaviate*.

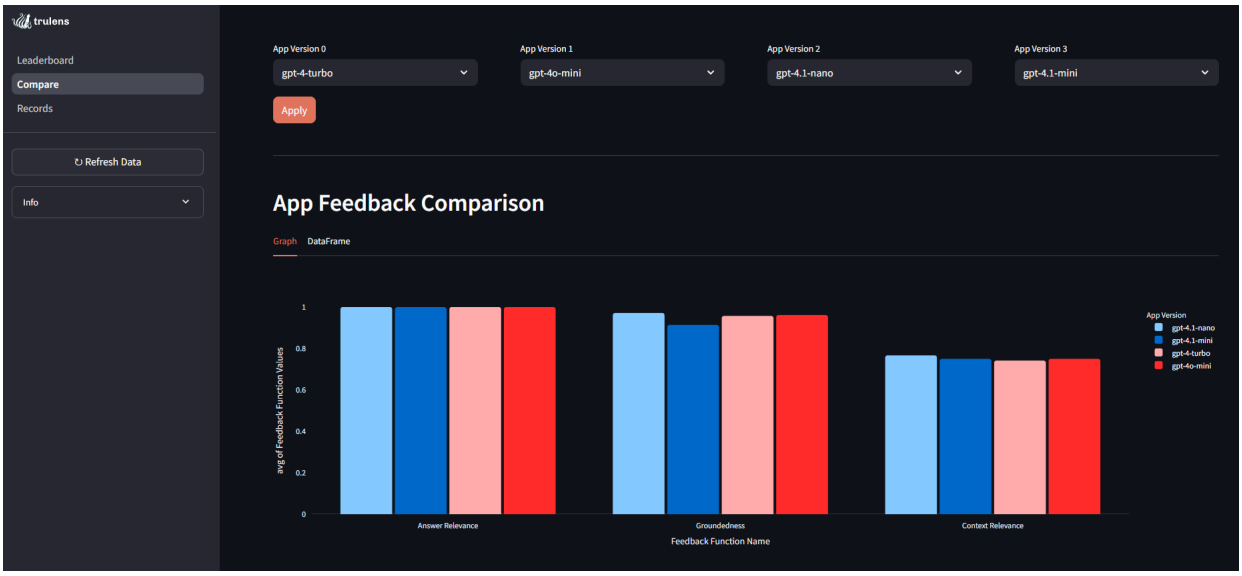

Figura 4.5: Comparación de versiones de la aplicación en *TruLens*.

En la imagen 4.5 vemos el resultado que produce el sistema tras responder a 10 preguntas relacionadas con ALF, probando distintas versiones del RAG con varios LLMs internos (*gpt-4-turbo*, *gpt-4o-mini*, *gpt-4.1-nano* y *gpt-4.1-mini*). Prácticamente todos los modelos obtienen resultados similares, si bien *gpt-4.1-mini* tiene un resultado ligeramente inferior en la métrica de *groundedness*.

En la imagen 4.6 se muestra un registro de TruLens para una de las consultas realizadas sobre la versión del RAG que emplea el modelo *gpt-4.1-nano*. Podemos observar los valores de número total de tokens, coste de la consulta y latencia. También aparecen los resultados de las funciones de feedback. Pulsando sobre cualquiera de ellas se accede a una explicación de la medida.

Por ejemplo, si se pulsa en la medida *Context Relevance* se puede ver la página mostrada en la figura 4.7. Se aprecian los fragmentos de los apuntes que se han utilizado para responder a la *query*. El primero de ellos es el de la definición y tiene el *score* mayor. Además, también se han elegido otros tres que no contienen la definición, sino ejemplos relacionados con la pregunta realizada. Pinchando en *Criteria* se puede ver el razonamiento que ha hecho el modelo evaluador usado por *TruLens* para calcular este parámetro.

4.6. Prompting

Ya hemos visto en la sección 2.6. que el *prompting* juega un papel fundamental en la capacidad generativa de los grandes modelos del lenguaje. Es esencial para la generación de respuestas de los LLMs, puesto que una composición mal estructurada puede generar una respuesta mala, aun cuando se trata un buen modelo con aumento de información, como puede ser nuestro caso.

LlamaIndex ofrece `prompt templates`, para poder personalizar el estilo o estructura

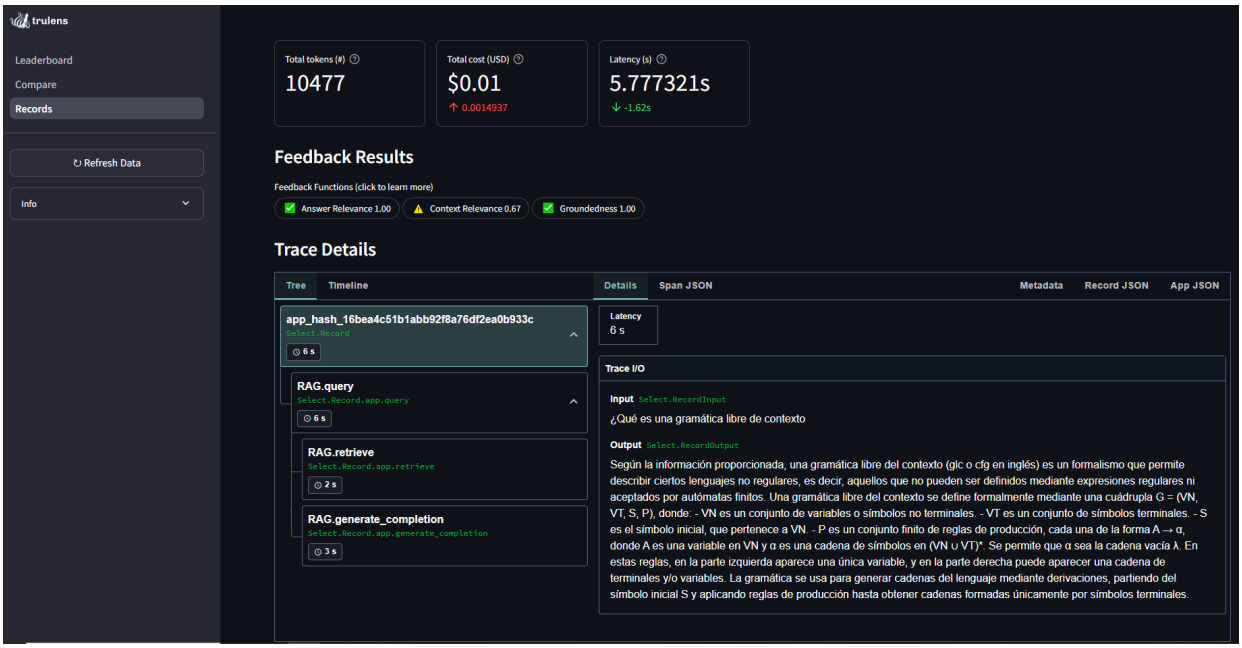

Figura 4.6: Ejemplo de registro de TruLens para el RAG con *gpt-4.1-nano*.

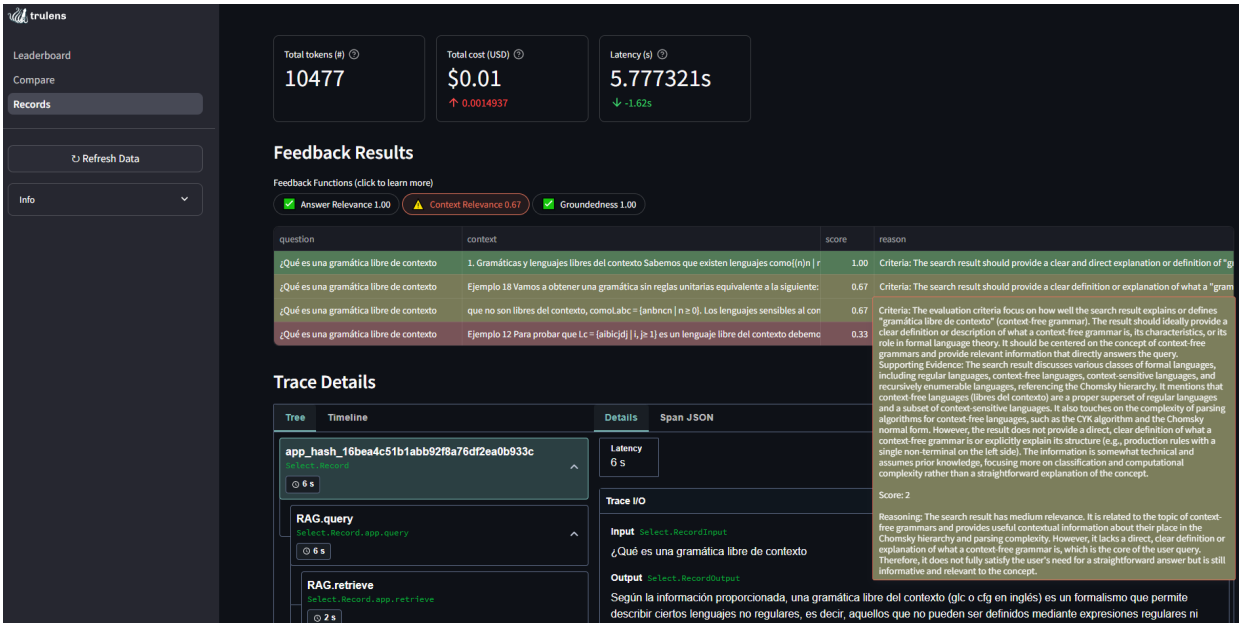

Figura 4.7: Ejemplo de explicación de TruLens para el resultado de *Context Relevance*.

de diferentes formas. Así se pueden mitigar alucinaciones y mantener el control sobre la precisión en las respuestas generadas.

A pesar de la facilidad a la hora de configurar, el prompt que se usa en el programa que se explica en la sección 4.4 es uno de los que *LlamaIndex* usa por defecto. En él se añade primero el contexto y posteriormente incluye la query; además, se pide al modelo generativo que responda sobre la base del contexto que recibe de los documentos. Pueden consultarse las plantillas a través de este enlace.

Sin embargo, en la sección 4.5, en *TruLens* se indica el prompt a través del cual generará la respuesta. Dentro de la función `generate_completion()`, en la clase `RAG`, el constructor de la `completion` que devuelve la función recibe el prompt como parámetro, dentro de una lista llamada `messages`. Se puede ver cómo se define el prompt en sí, y puede modificarse manteniendo el formato adecuado.

Aparte, vemos que podemos ajustar el rol del modelo de OpenAI entre diferentes papeles, consiguiendo que el modelo que genera la respuesta tenga un comportamiento distinto en cada caso. Tenemos el rol `developer`, que hace que el modelo entienda las instrucciones que le da el desarrollador de la aplicación, con prioridad sobre las que le puede mandar el usuario. También tenemos el rol `user`, que es el que tiene nuestra implementación. Con este papel, los mensajes que recibe el modelo son interpretados como mensajes que le manda el usuario.

CAPÍTULO 5

Conclusiones y vías futuras

Finalizamos el trabajo estableciendo las conclusiones tras haber desarrollado por completo el proyecto, y analizar los diferentes objetivos mencionados anteriormente. También se proponen algunas vías futuras, que probablemente supongan una mejora inmediata en algunos campos.

5.1. Conclusiones

La inteligencia artificial generativa (GenAI) representa una de las ramas más avanzadas y transformadoras de la inteligencia artificial en la actualidad, y se basa en modelos capaces de generar contenido original a partir de simples peticiones del usuario en lenguaje natural. Entre ellos, destacamos los modelos del lenguaje a gran escala (LLMs). En los últimos años, han destacado modelos, como GPT de *OpenAI* o *Gemini* de *Google*, por su capacidad de comprensión, razonamiento y producción de lenguaje humano, con un nivel de coherencia muy alto. Se entrenan con grandes cantidades de datos y pueden utilizarse en multitud de tareas.

Sin embargo, a pesar de su potencia, los grandes modelos del lenguaje presentan ciertas limitaciones, ya que su conocimiento está limitado al momento en el que se entrenó el modelo y no poseen información actualizada en tiempo real. Esto puede producir alucinaciones en las respuestas, es decir, que el modelo genere un contenido que puede ser erróneo pero difícilmente verificable, porque las respuestas son semántica y sintácticamente correctas.

Para superar estas barreras surgió la idea del flujo RAG, *Retrieval-Augmented Generation*, que combina un modelo generativo con un sistema de recuperación de información. En un flujo RAG, antes de producir la respuesta, el sistema busca los fragmentos de texto en una base de datos externa como documentos, PDFs, páginas web, para incorporarlos como contexto, obteniendo respuestas más precisas, trazables y fundamentadas, utilizando datos específicos y actualizados, ampliando significativamente la utilidad práctica

general de los LLMs.

A lo largo del desarrollo de este proyecto, se ha implementado una arquitectura RAG, conectando varias herramientas prácticas de los sistemas de inteligencia artificial, como *LlamaIndex* o *Weaviate*.

La evolución del proyecto demuestra que es posible integrar de manera modular y escalable un sistema RAG que es capaz de responder preguntas de forma correcta a partir de una serie de documentos docentes, respetando la trazabilidad. Técnicas como la fragmentación semántica, *embeddings* o el *prompting* han sido pilares esenciales para lograr resultados coherentes y precisos. Este enfoque nos permite no depender únicamente del conocimiento preentrenado del modelo, algo favorable cuando se trabaja con información especializada, reciente y/o sensible.

Además, incorporar Trulens al trabajo nos permite medir la calidad de las respuestas de manera objetiva y automática, pudiendo verificar que el contenido tiene sentido, está justificado y es útil. En resumen, el sistema muestra el potencial de mejorar el acceso al conocimiento.

5.2. Vías futuras

Existen múltiples direcciones en las que este proyecto puede evolucionar y crecer. En esta sección comentamos algunas mejoras que pueden mejorar las prestaciones actuales de la aplicación desarrollada.

Para empezar, aunque el sistema actual funciona a través de la consola de comandos, una posible mejora sería construir una interfaz gráfica o web, ampliando el alcance del proyecto y lo haría mucho más útil para usuarios sin conocimientos técnicos. Además, podría extenderse de forma que la aplicación pueda soportar múltiples fuentes de datos: en vez de utilizar documentos locales, que también puedan usarse otros datos que el usuario disponga. También podría permitir el feedback del usuario final, para hacer aún más precisa la generación de la respuesta del usuario. Esto también supondría actualizar periódicamente la base de conocimiento del sistema, pero daría un buen resultado.

Además, relacionado con *TruLens*, se podría afinar aun más la fragmentación semántica y el motor de búsqueda usando modelos de chunking distintos, ajustando parámetros de recuperación y adaptándolos a distintos tipos de contenido. Por otro lado, otra vía intersente de mejora sería comparar la respuesta entre distintos LLMs, usando la misma arquitectura y evaluar cuál se comporta mejor en base a las métricas de *TruLens*.

Una posible mejora, también basada en *TruLens*, consistiría en añadir un mecanismo de filtrado de los fragmentos recuperados de la base de datos *Weaviate Cloud*. En la implementación actual, independientemente de su relevancia, todos los fragmentos son usados como contexto en el prompt que se envía al LLM usado en el RAG. *TruLens* permite emplear *feedback guardrails*, es decir, funciones de feedback que, en lugar de servir para una evaluación final, se pueden emplear para filtrar los fragmentos que se emplearán como contexto. Sólo aquellos fragmentos que superen un valor de relevancia

mínimo se emplearán en la consulta. En este punto sería interesante evaluar el uso de LLMs pequeños, poco costosos o incluso ejecutables en local, que podrían intervenir en esta operación de filtrado.

Por último, también podría plantearse un futuro despliegue a alguna empresa o institución, para permitir consultas directas sobre materiales de clase (como se ha hecho con la asignatura de *Autómatas y Lenguajes Formales*), artículos científicos o libros. De este modo, por ejemplo, una clínica de fisioterapia podría realizar consultas sobre tratamientos muy recientes sin necesidad de reentrenar el gran modelo del lenguaje.

Bibliografía

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 technical report, 2024. URL: https://arxiv.org/abs/2412.08905, arXiv:2412.08905.

[2] Amazon. Transformer in artificial intelligence, 2024. URL: https://aws.amazon.com/es/what-is/transformers-in-artificial-intelligence/.

[3] Keep Coding. Limitaciones y desafíos de los llms. URL: https://keepcoding.io/blog/limitaciones-y-desafios-de-los-llm/.

[4] Mark Scapicchio Cole Stryker. What is generative ai?, 2024. URL: https://www.ibm.com/think/topics/generative-ai.

[5] Roberto Crespo. Transformer attention is all you need. URL: https://www.robertocrespo.net/transformer-attention-is-all-you-need/.

[6] Andrei Gheorghiu. *Building Data-Driven Applications with LlamaIndex*. packt, 2024.

[7] Red Hat. ¿qué es la generación aumentada por recuperación o rag? URL: https://www.redhat.com/es/topics/ai/what-is-retrieval-augmented-generation.

[8] Matthew Kosinski Jim Holdsworth. What is a vector database?, 2024. URL: https://www.ibm.com/es-es/think/topics/vector-database.

[9] Albert Ziegler John Berryman. *Prompt Engineering for LLMs: The Art and Science of Building Large Language Model–Based Applications*. O'Reilly, 2023.

[10] @laujan (GitHub). Generación aumentada de recuperación con documento de inteligencia de azure ai. URL: https://learn.microsoft.com/es-es/azure/ai-services/document-intelligence/concept/retrieval-augmented-generation?view=doc-intel-4.0.0.

[11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL: https://arxiv.org/abs/2005.11401, arXiv:2005.11401.

[12] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2018. URL: https://arxiv.org/abs/1603.09320, arXiv:1603.09320.

[13] Elena Navarro Sánchez et al. Deep learning aplicado a la búsqueda de contenidos multimedia, 2024.

[14] Ismael Negueruela Gómez et al. Análisis y evaluación de llms con técnicas rag para el desarrollo y despliegue de un asistente virtual, 2024.

[15] Ministerio para la Transformación Digital y de la Función Pública. Rag - retrieval augmented generation: La llave que abre la puerta de la precisión a los modelos del lenguaje. Technical report, Gobierno de España, 2024.

[16] Limitaciones de los llms.

[17] Arroba System. Rag (retrieval-augmented generation): Revolucionando la generación de contenidos con inteligencia artificial. URL: https://shorturl.at/HdPMP.

[18] TruLens. Trulens: Evaluation framework for llm applications. Accessed: 2025-05-23. URL: https://www.trulens.org/.