Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Arquitectura Multiagente para Modelos RAG en Entornos de Tráfico Aéreo

Autor: Eduardo Andrés Rodríguez Jaguan

Tutor: Javier Bajo Pérez

Madrid, Abril 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Arquitectura Multiagente para Modelos RAG en Entornos de Tráfico Aéreo
Abril 2025

Autor: Eduardo Andrés Rodríguez Jaguan

Tutor:
Javier Bajo Pérez
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El presente Trabajo de Fin de Grado, titulado “Arquitectura Multiagente para Modelos RAG en Entornos de Tráfico Aéreo”, aborda el diseño e implementación de un sistema multiagente basado en modelos de lenguaje de gran tamaño (LLMs). A través de una implementación RAG (Generación Aumentada por Recuperación), este sistema es capaz de detectar posibles errores en las comunicaciones ATC (Control de Tráfico Aéreo). Estas comunicaciones son la vía de contacto principal entre controladores y pilotos, por lo que deben garantizar la máxima fiabilidad y precisión. Es por eso que el objetivo del proyecto es mejorar la seguridad y fiabilidad de estas operaciones, demostrando cómo la aplicación de la Inteligencia Artificial Generativa podría tener un papel clave en el futuro del sector aeronáutico.

La solución desarrollada cuenta con cinco agentes, cada uno con una responsabilidad única. Estos agentes se encargan de procesar los datos que consumirá el programa, construir y gestionar el RAG, alimentar al modelo de lenguaje (LLM) con datos contextuales y relevantes, generar comunicaciones ATC sintéticas, analizar dichas comunicaciones, detectar errores y dar retroalimentación en tiempo real mediante streaming simulado. Además, se realizan pruebas para garantizar la calidad del feedback proporcionado para cada escenario.

Todo esto es visible y accesible para el usuario mediante un frontend interactivo desarrollado con Streamlit. Este interfaz permite ejecutar el pipeline completo, visualizar en tiempo real las comunicaciones ATC generadas y las aportaciones realizadas por los agentes, así como interactuar directamente con el sistema para seleccionar escenarios específicos y comprobar la efectividad del modelo a través de una interfaz sencilla e intuitiva.

El impacto de este Trabajo de Fin de Grado radica en sentar unas bases tecnológicas y conceptuales necesarias para incorporar la inteligencia artificial generativa en el sector del tráfico aéreo. La arquitectura desarrollada actúa como una prueba de concepto (PoC), demostrando cómo esta tecnología puede mejorar la seguridad operativa mediante el uso coordinado de múltiples agentes inteligentes capaces de monitorizar acciones humanas. Además, establece un punto de partida para desarrollos futuros e implementaciones más avanzadas que podrían transformar las comunicaciones ATC y reducir el número de incidentes aeronáuticos.

En conclusión, este trabajo valida con el potencial de la inteligencia artificial generativa aplicada mediante una arquitectura multiagente para mejorar la seguridad y fiabilidad en las comunicaciones de control de tráfico aéreo, sentando así las bases para futuras investigaciones y aplicaciones prácticas en el ámbito aeronáutico.

Abstract

The present Final Degree Project, titled "Multi-Agent Architecture for RAG Models in Air Traffic Environments," addresses the design and implementation of a multi-agent system based on large language models (LLMs). Through a Retrieval Augmented Generation (RAG) implementation, this system is capable of detecting potential errors in Air Traffic Control (ATC) communications. These communications are the primary contact channel between controllers and pilots, thus requiring maximum reliability and accuracy. Therefore, the project's goal is to enhance safety and reliability in these operations, demonstrating how Generative Artificial Intelligence could play a crucial role in the future of the aeronautical sector.

The developed solution consists of five agents, each with a unique responsibility. These agents are tasked with processing the data consumed by the program, creating and managing the RAG system, supplying contextual and relevant data to the language model (LLM), generating synthetic ATC communications, analyzing these communications, detecting errors, and providing real-time feedback through simulated streaming. Additionally, tests are performed to guarantee the quality of the feedback provided for each scenario.

All of this is visible and accessible to users via an interactive frontend developed with Streamlit. This interface allows the execution of the complete pipeline, real-time visualization of generated ATC communications and agent contributions, and direct interaction with the system to select specific scenarios and verify the model's effectiveness through a simple and intuitive interface.

The impact of this Final Degree Project lies in laying the technological and conceptual foundations necessary to incorporate generative artificial intelligence into the air traffic sector. The developed architecture acts as a proof of concept (PoC), demonstrating how this technology can enhance operational safety through the coordinated use of multiple intelligent agents capable of monitoring human actions. Additionally, it establishes a starting point for future developments and more advanced implementations that could transform ATC communications and reduce the number of aerial incidents.

In conclusion, this project validates the potential of generative artificial intelligence applied through a multi-agent architecture to improve safety and reliability in air traffic control communications, thus laying the groundwork for future research and practical applications in the aeronautical field.

Tabla de contenidos

Capítulo 1

Introducción

Este primer capítulo tiene como objetivo proporcionar al lector una visión general del proyecto, abordando aspectos como la motivación que lo impulsa, los objetivos planteados, la planificación seguida, una descripción de la estructura de la memoria y, finalmente, la solución propuesta. Esta información es de mucha ayuda para facilitar la comprensión de los capítulos posteriores y contextualizar el desarrollo del trabajo.

1.1 Motivación

Durante los años 2024 y 2025, se ha registrado un incremento en el número de incidentes y accidentes aéreos a nivel mundial, lo cual ha generado preocupación en el ámbito aeronáutico [38]. Estos sucesos han destacado la necesidad de contar con sistemas capaces de garantizar seguridad y fiabilidad, especialmente en las comunicaciones de control de tráfico aéreo (ATC), dado que estas son la principal vía de contacto entre controladores y pilotos. Este contexto coincide además con la creciente popularidad de la inteligencia artificial generativa, que ha experimentado un fuerte incremento tanto en adopción como en capacidades técnicas, abriendo la puerta al desarrollo de nuevas soluciones innovadoras en todo tipo de sectores. Es precisamente gracias a estos recientes avances, que ahora resulta viable desarrollar sistemas de tipo PoC (pruebas de concepto), permitiendo así validar la aplicabilidad de estos sistemas antes de comenzar a plantearlos en entornos de producción.

1.2 Objetivos del proyecto

El objetivo principal de este Trabajo de Fin de Grado es desarrollar una Prueba de Concepto (PoC) de un sistema multiagente basado en Inteligencia Artificial Generativa para mejorar la seguridad y fiabilidad en las comunicaciones de Control de Tráfico Aéreo (ATC). Este sistema implementará una arquitectura de Generación Aumentada por Recuperación (RAG) que, mediante la consulta en tiempo real de diversas bases de conocimiento, permitirá detectar y corregir posibles errores en las comunicaciones ATC.

Para alcanzar este objetivo general, se establecen los siguientes objetivos específicos:

1. **Realizar un estudio de las soluciones existentes, datos disponibles, tecnologías aplicables y estado del arte de la tecnología** en el ámbito de las comunicaciones ATC y la Inteligencia Artificial Generativa, para identificar oportunidades de mejora y establecer el punto de partida para el desarrollo.
2. **Diseñar e implementar un sistema multiagente**, que permita la interacción eficiente de agentes especializados en el procesamiento y análisis de comunicaciones ATC.
3. **Integrar modelos de lenguaje de gran tamaño (LLMs)** como GPT-4o de OpenAI y Sonar de Perplexity dentro del pipeline, para generar y evaluar comunicaciones ATC sintéticas.
4. **Implementar una arquitectura de Generación Aumentada por Recuperación (RAG) con una base de datos vectorial**, facilitando el almacenamiento y recuperación de embeddings necesarios para el funcionamiento del sistema.
5. **Desarrollar una interfaz de usuario interactiva**, que permita a los usuarios interactuar con el sistema, visualizar en tiempo real las comunicaciones generadas y analizar la retroalimentación propuesta por los agentes.
6. **Evaluar el rendimiento del sistema** mediante pruebas que garanticen la calidad del feedback proporcionado en diversos escenarios, asegurando la efectividad del sistema en la detección y corrección de errores en las comunicaciones ATC.

1.3 Planificación

Para llevar a cabo el proyecto de forma ordenada, se ha definido una planificación que divide todas las tareas necesarias a lo largo del tiempo establecido. Las fases principales que se han considerado son:

Fase 1: Estado del arte y fundamentos teóricos

Fase 2: Búsqueda y análisis de fuentes de datos

Fase 3: Diseño e implementación del modelo RAG

Fase 4: Diseño e implementación de la arquitectura multiagente

Fase 5: Pruebas continuas

Fase 6: Evaluación y validación de resultados

Fase 7: Preparación de la defensa, conclusiones y posibles mejoras o investigaciones futuras.

Estas tareas se han ordenado en el tiempo de la siguiente manera:

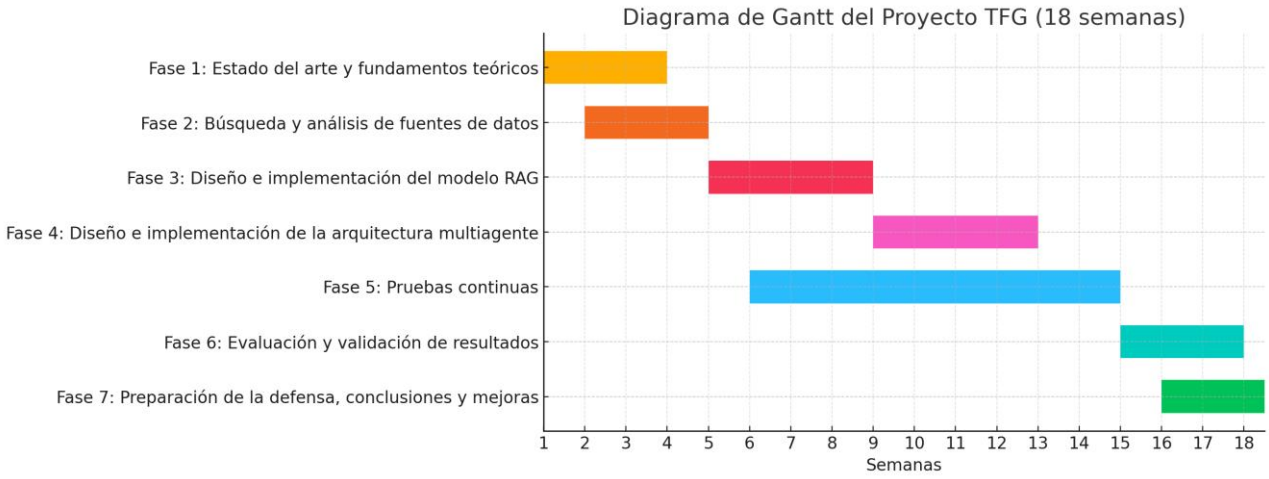


Figura 1.1: Diagrama de Gantt

1.4 Estructura de la memoria

La memoria tiene la siguiente estructura lógica:

Capítulo 2: Estado del arte. Se realiza un estudio en detalle de los marcos teóricos y técnicos necesarios para el desarrollo del proyecto. El estudio abarca las soluciones existentes, datos disponibles, tecnologías para el desarrollo y conceptos clave acerca de inteligencia artificial generativa como, como LLMs y RAG.

Capítulo 3: Diseño del sistema. En este capítulo se presenta tanto el diseño de alto nivel (DAN) como el diseño de bajo nivel (DBN), detallando la arquitectura y componentes principales.

Capítulo 4: Desarrollo. Se recoge el proceso de implementación del sistema.

Capítulo 5: Resultados y Análisis de Consumo y. Aquí se analizan los resultados obtenidos, se identifican posibles mejoras y se muestran los costes, en términos monetarios y de recursos, asociados al proyecto.

Capítulo 6: Conclusiones y trabajo futuro. Este capítulo presenta las principales conclusiones del proyecto, así como líneas de mejora y trabajo futuro.

Capítulo 7: Análisis de impacto. Por último, se estudia el impacto del proyecto, el valor añadido y el ámbito en el que genera mayor incidencia.

1.5 Solución propuesta

Se propone el desarrollo de una Prueba de Concepto (PoC) basada en un sistema multiagente que integra inteligencia artificial generativa y combina distintos enfoques previamente planteados en soluciones existentes. El sistema incorporará una arquitectura de Generación Aumentada por Recuperación (RAG), lo que permitirá consultar múltiples fuentes de conocimiento en tiempo real y generar respuestas más precisas y contextualizadas. Esta arquitectura se alimentará no solo de bases de datos propias, sino también de información obtenida mediante el motor de búsqueda Perplexity.

Aprovechando el enfoque por agentes, se implementará un RAG híbrido, capaz de realizar búsquedas tanto semánticas como léxicas, aplicando posteriormente técnicas de reranking para asegurar que la información obtenida sea siempre la más relevante y precisa. Este enfoque no solo mejora la detección de errores en las comunicaciones ATC, sino que además elimina la necesidad de realizar un fine-tuning del modelo, como ocurre en propuestas como REDA, lo que se traduce en una reducción significativa de costes y en un desarrollo más ágil.

Capítulo 2

Estado del Arte

La comunicación entre un piloto y un controlador es crítica para la seguridad aérea. Los errores de comunicación han sido causa de numerosos accidentes. Por ejemplo, el accidente de Tenerife de 1977, donde un fallo de interpretación fue causa de la pérdida de 583 vidas [1]. Diversos estudios muestran que hasta un 30% de los incidentes de aviación incluyen fallos de comunicación, y alrededor del 40% de las incursiones a pista se relacionan con este tipo de error. [39]

Tras el accidente de Washington provocado por un fallo de comunicación [2], se evidencia la necesidad de mejorar la claridad y seguridad en estas. En este contexto las técnicas de Inteligencia Artificial se muestran como herramientas prometedoras para ayudar a reducir errores y aumentar la seguridad operacional.

Sistemas de reconocimiento de voz pueden transcribir y procesar mensajes, y enfoques de Generación Aumentada por Recuperación (RAG) permitirían a agentes generativos consultar bases de conocimiento aeronáutico, capacitando al sistema para detectar incoherencias o proporcionar aportaciones durante la conversación. [40][3]

2.1 Soluciones existentes

Actualmente existen varias soluciones que buscan abordar los problemas de comunicación en tráfico aéreo mediante la Inteligencia Artificial:

1. Reconocimiento Automático del Habla (ASR): Los sistemas ASR son capaces de procesar el habla y generar texto. Existen iniciativas europeas, como SESAR, que están trabajando en desarrollar sistemas ASR específicos para comunicaciones controlador-piloto en el ámbito aeronáutico. SESAR está logrando tasas de error de palabra inferiores al 5% y es capaz de reconocer correctamente más del 85% de las instrucciones. Consiguieron implementar asistentes como REDA (Readback Error Detection Assistant). Un asistente con la capacidad de identificar incoherencias entre una instrucción ATC y la repetición del piloto (readback) y capaz de generar alertas en caso de que no coincidan. A la hora de probar el sistema, REDA detectó más del 80% de los errores de readback con menos del 20% de falsas alarmas. [41][4][5]

2. Agentes generativos con Inteligencia Artificial: La inteligencia Artificial tiene la capacidad tomar decisiones en tiempo real durante las

comunicaciones ATC. Existen sistemas comerciales de entrenamiento, como SERA, que han implementado agentes de IA capaces de llevar a cabo el rol de controladores y pilotos en entornos simulados. SERA es capaz de mantener comunicaciones con un alumno mediante fraseología aeronáutica, permitiendo el entrenamiento sin profesores humanos. Esto demuestra la viabilidad y el interés de aplicar agentes con IA generativa en el ámbito aeronáutico. Más allá del ámbito formativo, han surgido proyectos experimentales que simulan copilotos virtuales. Por ejemplo, investigadores de MIT desarrollaron Air-Guardian, un agente de IA que colabora con el piloto (actuando como un copiloto), de manera que monitoriza el entorno y alerta de posibles riesgos. [42][6][7]

3. Generación Aumentada por Recuperación (RAG): Un sistema RAG se plantea como una solución viable, ya que permite a los agentes ampliar su base de conocimiento establecida en el entrenamiento y gana la capacidad de generar respuestas más especializadas y relevantes. En lugar de basar sus respuestas solo en lo aprendido en el entrenamiento, un agente con una implementación RAG puede consultar documentos relevantes al caso de uso específico (por ejemplo, el Manual de Fraseología ICAO, bases de datos de planes de vuelo, historiales de instrucciones previas, etc.) para asegurar que sus aportaciones sean adecuadas. Un caso de uso actual es AviationGPT, un LLM en desarrollo, que incorpora un sistema RAG que permite al modelo buscar en esa base de conocimiento las respuestas antes de generar la salida. [9][8]

2.2 Problemas detectados

A pesar de los avances que estamos viviendo en Inteligencia Artificial, las aplicaciones multiagente con IA y los sistemas RAG en el entorno de tráfico aéreo siguen enfrentando varios retos:

Latencia y tiempo real: En el contexto de comunicaciones de tráfico aéreo, estas ocurren en tiempo real y de manera muy rápida. Cualquier solución debe asegurar una latencia mínima durante su funcionamiento.

Precisión y fiabilidad: Cualquier sistema en el entorno aeronáutico debe garantizar el mayor grado de precisión. Un error puede generar confusión, que a su vez puede traducirse en un incidente.

Complejidad del lenguaje aeronáutico: Las comunicaciones controlador-piloto se ven afectadas por problemas propios del lenguaje y de cualquier interacción humana mediante el habla [41]:

1. **Fraseología**
   La OACI (International Civil Aviation Organization) establece una fraseología estándar para las comunicaciones ATC, pero en la práctica se pueden encontrar algunas variantes. Además, pilotos y controladores pueden modificar ligeramente las frases, de forma que tienen un significado muy parecido a las establecidas pero dificultan el procesamiento con modelos ASR. [9]
2. **Acentos y pronunciaciones**
   La aviación es un entorno internacional donde controladores y pilotos provienen de distintos países, lo que se traduce en una diversidad de

acentos y pronunciaciones que pueden suponer un desafío para generar las transcripciones con la precisión necesaria.

3. **Términos especializados**
   Las comunicaciones ATC emplean términos muy específicos, como indicativos de llamada, códigos específicos, designadores de vías, etc. Tanto el ASR como el sistema de procesamiento de la transcripción deberían ser capaces de detectar estos términos correctamente y analizarlos considerando su significado en el sector. [43]

Entorno ruidoso e interferencias: El entorno de un piloto normalmente se ve afectado por una gran cantidad de ruido (motores, viento, sonidos de cabina, etc.). Además las comunicaciones pueden verse sometidas a interferencias y distorsiones ya que se dan en un canal semidúplex. Estos factores demuestran el desafío que supone implementar sistemas de reconocimiento del habla en el entorno mencionado.

2.3 Datos Disponibles

Para construir un sistema basado en Inteligencia Artificial Generativa con unas capacidades mínimas dentro del sector aeronáutico es imprescindible contar con fuentes de datos fiables y de calidad. De esto depende la calidad del sistema. Aquí se muestran diversas fuentes de datos disponibles para nuestro uso:

Red de Seguridad Aérea (Aviation Safety Network ASN): La ASN es una organización que realiza un seguimiento de incidentes de aerolíneas. Su base de datos contiene detalles de todos los accidentes aéreos, categorizados por año. [46]

Taxonomía ADREP de la OACI: La Organización de Aviación Civil Internacional (OACI) gestiona el sistema ADREP, que recopila todos los datos para categorizar y describir accidentes e incidentes aéreos. [9]

Mejores prácticas en comunicaciones por radio: La AIM proporciona una guía básica sobre mejores prácticas a la hora de comunicarse por radio en comunicaciones ATC. [10]

Guía de Buenas Prácticas en Fraseología y Comunicaciones de AESA: La Agencia Estatal de Seguridad Aérea (AESA) de España ofrece una guía que aborda las mejores prácticas en fraseología en comunicaciones tierra-aire. [11]

Vocabulario aeronáutico Inglés-Español: Un documento, parecido a un diccionario, que presenta diferentes términos usados en las comunicaciones ATC tanto en inglés como en español. [43]

Terminología aeronáutica: Documento con una lista de términos usados frecuentemente en aeronáutica. [12]

Transcripciones de comunicaciones reales en situaciones de peligro: Sería conveniente contar con comunicaciones reales en situaciones adversas, donde se puedan identificar fallos en la comunicación piloto-controlador. Hay diversas fuentes donde se encuentran fragmentos relevantes como:

- Wikipedia
- ASN, “Aviation Safety Network,” [46]

- Idiap Research Institute, “atco2-corpus,” [13]
- Jacktol, “ATC Dataset” [14]
- Flight Insight, “VFR Radio Communications Script” [15]
- VATMEX, “Manual de comunicaciones básicas en inglés” [16]

2.4 Tecnologías

Para desarrollar un sistema multiagente con RAG debemos evaluar y elegir las tecnologías más apropiadas para nuestro caso de uso en concreto. A continuación se muestra el “stack” elegido para el desarrollo del proyecto :

LangChain y LangGraph: Un framework diseñado para desarrollar e implementar agentes con Inteligencia Artificial. Ofrece una solución altamente personalizable en comparación con otros frameworks “no-code” como Google Agent Builder o Amazon Bedrock. Además se convierte en la opción más potente si se usa junto con LangGraph, un framework especializado en la orquestación de estos agentes. Tras comparar su desempeño con otras opciones como Crew AI o AutoGen, el ecosistema de LangChain y LangGraph se demuestra ganador para nuestro caso de uso, aportando una solución potente y personalizable con una gran comunidad que aporta valor continuamente. [17][18]

Python: Como lenguaje de desarrollo principal se elige Python. Además de ser uno de los pocos lenguajes soportados por el ecosistema LangChain, es un lenguaje considerado de uso “fácil”, ya que tiene un tipado dinámico. Python tiene un ecosistema muy sólido y maduro, con una comunidad muy activa, y una extensa lista de librerías especializadas en el sector de Inteligencia Artificial, procesamiento de datos y procesamiento de voz. En definitiva es la opción ideal para un proyecto de estas características.

Modelos de Lenguaje (LLMs): Para este proyecto se valoran modelos de lenguaje (LLMs) de consumo vía API, ya que no se cuenta con una GPU capaz de soportar un modelo open source de forma local. El proyecto usará el modelo “gpt-4o” de openAI para procesar la mayor parte de la información y generar los resultados del sistema. Por otra parte se usará el modelo “sonar” de perplexity para hacer búsquedas en internet e integrar los resultados con el sistema RAG. Por último se usará Mistral OCR para procesar PDFs y obtener datos en un formato adecuado para realizar el procesamiento y su posterior consumición por el modelo llm. [19][21][22][20]

ChromaDB (Vector DB): Se elige ChromaDB como base de datos vectorial. ChromaDB es una base de datos vectorial que está integrada en el ecosistema de LangChain, facilitando su uso e integración con el sistema. Es una solución sencilla, opensource y con ejecución local que nos servirá para almacenar y recuperar embeddings, permitiendo que el sistema acceda rápidamente a información de la implementación RAG. [23]

Streamlit: Streamlit se utilizará para desarrollar la interfaz de usuario de forma rápida y sencilla gracias a su integración con Python. Streamlit es una solución específica para aplicaciones basadas en IA generativa y aplicaciones basadas en manipulación y presentación de datos. Su facilidad y especialización en este sector lo hace ideal para desarrollar una interfaz en una prueba de concepto (PoC). Permite visualizar y probar las funcionalidades del sistema de forma ágil y elimina la necesidad de usar frameworks más complejos. [24]

Estas tecnologías ofrecen un stack completo para desarrollar el sistema: Python como base, LangChain para arquitectura de agentes y RAG, modelos de última generación de OpenAI, perplexity y mistral para las capacidades de lenguaje, voz y testing, ChromaDB para la base de conocimiento de la implementación RAG y Streamlit como framework para desarrollar la interfaz de usuario.

2.5 Large Language Models (LLMs)

2.5.1 Definición y fundamentos teóricos

Los modelos de lenguaje de gran tamaño (LLMs) son sistemas de inteligencia artificial generativa diseñados para procesar y generar texto en lenguaje natural. Estos modelos han representado uno de los mayores avances tecnológicos de los últimos años, con la capacidad de transformar diversas industrias y la manera en que interactuamos con sistemas informáticos. Estos sistemas se han basado sobre todo en la arquitectura Transformer, introducida por Vaswani et al. en 2017 [25], la cual ha sido fundamental en el desarrollo de LLMs debido a su capacidad de identificar y modelar relaciones complejas en secuencias de texto.

Recientemente, ha ganado popularidad la arquitectura Mixture of Experts (MoE), especialmente con implementaciones como la de DeepSeek. Esta arquitectura ofrece ventajas significativas, como la eficiencia computacional al activar solo un subconjunto de expertos durante cada paso de inferencia, lo que reduce la carga computacional y permite tiempos de procesamiento más rápidos con un menor consumo de energía [34][26].

El funcionamiento de los LLMs se basa en:

1. **Entrenamiento supervisado o no supervisado**:

   Entrenamiento supervisado: En este enfoque, los modelos aprenden a partir de conjuntos de datos etiquetados. Es decir, se les proporcionan ejemplos donde tanto la entrada como la salida esperada están ya definidas. Por ejemplo, a la hora de realizar una traducción, se les mostraría una frase en un idioma junto con su traducción correcta. Esto permite que el modelo aprenda a predecir respuestas precisas basadas en ejemplos anteriores.

   Entrenamiento no supervisado: Aquí, los modelos trabajan con datos no etiquetados, buscando patrones sin una guía explícita. Un ejemplo sería analizar grandes volúmenes de texto para identificar temas comunes o agrupaciones de palabras que suelen aparecer juntas, lo que ayuda al modelo a comprender relaciones semánticas sin conocer la respuesta. [35]

2. **Embeddings semánticos**: Los embeddings son representaciones de palabras o frases en forma de vectores en un espacio multidimensional. Básicamente, convierten palabras en números de tal manera que las palabras con significados similares están más cerca entre sí en el espacio. Esto permite que los modelos identifiquen relaciones de significado entre textos y tengan la capacidad de encontrar datos relevantes a partir de una base de datos contextual. [33]

2.5.2 Generación de información

Los LLMs generan texto mediante un proceso probabilístico. A partir de una entrada inicial (prompt), el modelo predice la siguiente palabra o token en función del contexto proporcionado. Este proceso se repite hasta completar la respuesta, y permite construir textos coherentes con respecto a la entrada del usuario.

El comportamiento del modelo puede ajustarse con ciertos parámetros. Por ejemplo, la temperatura controla el nivel de creatividad. Valores bajos hacen que el modelo sea más preciso, mientras que valores más altos hacen que el modelo sea más "creativo" e introduzca mas variabilidad en sus respuestas. También se emplean técnicas como top-k o top-p sampling, que ayudan a mantener la coherencia del texto evitando resultados poco relevantes.

Además, los modelos se pueden adaptar a tareas concretas mediante un "fine-tuning". En lugar de reentrenarlos desde cero, este enfoque permite reentrenar ciertas capas del modelo utilizando un conjunto de datos más pequeño y específico. De esta forma, el modelo aprende a responder mejor dentro de un contexto determinado, como por ejemplo el ámbito aeronáutico, donde se utiliza una terminología muy concreta. Esta técnica permite al modelo adaptarse a tareas concretas sin partir de cero, aprovechando desarrollos existentes y reduciendo costes.

2.5.3 Ejemplos actuales

En los últimos años, han surgido varios modelos avanzados que han revolucionado el campo:

1. **GPT-o1**: OpenAI fue una de las primeras compañías en ofrecer modelos de lenguaje como servicio, empezando con GPT-3 en 2020. Su último modelo, o1, es el más potente del mercado, con una capacidad de razonamiento superior a la de la competencia. Destaca especialmente en programación, análisis de datos y tareas creativas. [44]

2. **Deepseek R1**: Este modelo, introducido por DeepSeek, emplea una arquitectura de Mixture of Experts (MoE), lo que le permite activar solo una fracción de sus parámetros durante cada inferencia. Ha obtenido resultados iguales o mejores al modelo o1 de OpenAI, por una fracción del coste y ofreciéndolo a código abierto. [34]

3. **Claude 3.7 Sonnet**: Desarrollado por Anthropic, este modelo de razonamiento híbrido permite alternar entre respuestas rápidas y análisis detallados. Destaca en tareas complejas como programación y análisis de datos. [27]

4. **Mistral Small 3.1**: Presentado por Mistral AI, este modelo de 24 mil millones de parámetros es capaz de procesar tanto texto como imágenes, ofreciendo un rendimiento comparable o superior a modelos más grandes de la competencia. [45]

5. **Llama 4**: Desarrollado por Meta, Llama 4 es una familia de modelos de lenguaje de código abierto que incluye Llama 4 Scout y Llama 4 Maverick.

Estos modelos han sido entrenados en grandes cantidades de datos no etiquetados, permitiéndoles comprender y generar texto de manera efectiva. Además, presentan una ventana de contexto de 10M de tokens, muy superior a la competencia. [28]

6. **Gemini 2.5 Pro:** Creado por Google DeepMind, este modelo está diseñado para resolver problemas complejos que involucren tareas de razonamiento y generación de código. Es capaz de procesar múltiples tipos de datos, incluyendo texto, imágenes y audio, y se integra en diversas aplicaciones y servicios de Google. [29]

2.5.4 Aplicaciones

Gracias a su versatilidad y amplio conocimiento en diferentes ámbitos, los LLMs están siendo utilizados en diversos sectores como:

Ingeniería y Programación: Los LLMs han revolucionado la forma en que se desarrolla software. Una tendencia es el "vibe coding", donde los desarrolladores describen en lenguaje natural lo que quieren programar y la inteligencia artificial genera el código. Esto permite que personas sin conocimientos en programación puedan crear aplicaciones funcionales. Sin embargo, cabe destacar que el código generado por perfiles no técnicos suele contener errores, malas prácticas o vulnerabilidades, por lo que en la mayoría de los casos solo debería considerarse como un prototipo o MVP. Este tipo de soluciones no están preparadas para entornos de producción ni para ser desplegadas sin una revisión en detalle.

Atención al Cliente y Asistentes Virtuales: Muchas empresas han integrado LLMs en chatbots y asistentes virtuales para mejorar la comunicación con sus clientes. Estos sistemas pueden manejar consultas complejas y mantener conversaciones fluidas, mejorando la experiencia de usuario, aumentado la disponibilidad del servicio y reduciendo costes de personal.

Educación: Los LLM se han incorporado en plataformas educativas para personalizar el aprendizaje, responder preguntas de los estudiantes y generar contenido didáctico personalizado. Esto facilita que la educación esté adaptada a las necesidades individuales y promueve una mayor participación.

2.6 Generación Aumentada por Recuperación (RAG)

2.6.1 Definición y arquitectura básica

La Generación Aumentada por Recuperación (RAG) es una técnica que combina sistemas de recuperación de información con modelos generativos de lenguaje. Su arquitectura consta de dos componentes principales:

1. **Módulo de recuperación**: Busca información relevante en bases de datos o documentos externos.

2. **Módulo generativo**: Sintetiza respuestas usando tanto el conocimiento interno del LLM como los datos recuperados. [31]

Esta aproximación reduce las "alucinaciones" de los LLMs y mejora la precisión en dominios especializados como el aeronáutico.

2.6.2 Tipos de implementaciones RAG

Búsqueda semántica

La búsqueda semántica es una técnica que permite encontrar información relevante basándose en el significado semántico consulta, en lugar de depender únicamente de coincidencias exactas de palabras clave (como en logaritmos como BM25). [33]

Para lograr esto, tanto las consultas del usuario como los documentos disponibles se representan mediante embeddings, que son representaciones vectoriales en un espacio multidimensional del texto. Estos embeddings contienen el significado semántico del texto, posicionando fragmentos similares más cerca unos de otros en dicho espacio. Luego, para determinar qué fragmentos o vectores son más relevantes respecto a una consulta, se mide la proximidad entre los vectores empleando métricas como la similitud coseno.

Búsqueda híbrida

El RAG híbrido combina dos enfoques diferentes para aprovechar al máximo las ventajas de ambos métodos: la búsqueda léxica y la búsqueda semántica.

La búsqueda léxica usa técnicas como BM25. Es efectiva, sobre todo para encontrar términos específicos y precisos en el texto, como pueden ser códigos ICAO o terminología exacta. Por otro lado, la búsqueda semántica utiliza embeddings para reconocer significados más amplios, contextos y sinónimos, lo cual es muy útil cuando no existe una coincidencia exacta de términos. Este enfoque suele proporcionar mejores resultados a la hora de procesar textos legales o documentación técnica. [36][30]

MMR

MMR es un método de selección de documentos que busca equilibrar dos objetivos: Maximizar la relevancia respecto a una consulta y minimizar la redundancia entre los documentos recuperados.

Este enfoque de seleccionar documentos relevantes pero diversos entre sí, es especialmente útil cuando se necesitan respuestas que cubran distintas perspectivas y con una amplia ventana de contexto, evitando resultados repetitivos y mejorando los resultados.

La forma en que MMR selecciona documentos es la siguiente:

1. El algoritmo comienza evaluando la relevancia de todos los documentos disponibles respecto a la consulta introducida por el usuario, utilizando técnicas como BM25 o embeddings.
2. Tras este paso inicial, selecciona el documento que presenta la mayor relevancia con respecto a la consulta introducida.
3. A partir de aquí, el algoritmo continúa eligiendo documentos adicionales de forma iterativa, manteniendo siempre un equilibrio entre relevancia, diversidad y no repetición. [32][37]

RAG Agéntico

En esta implementación, un LLM gestiona dinámicamente el proceso de recuperación y generación. Este enfoque funciona de la siguiente manera:

1. El LLM genera una consulta, a partir de la pregunta del usuario, diseñada para ser eficaz en un sistema RAG con búsqueda semántica y léxica.
2. La consulta se vectoriza y se compara con los vectores almacenados en la base de conocimiento vectorial del RAG. Paralelamente, la consulta sin vectorizar se procesa a través de otras fuentes de datos (En el caso particular del proyecto, con la API de Perplexity).
3. Una vez obtenidos los resultados de las bases de conocimiento, el LLM razona sobre si los resultados son satisfactorios o si cabe una mejora. En el caso que se pueda hacer una mejora, volvemos al paso 1 y el LLM mejora la consulta y repite el proceso.
4. Una vez los datos obtenidos son satisfactorios, el LLM hace un "reranking". Esto es ordenar los resultados obtenidos por orden de relevancia. [31]

BM25

BM25 es un algoritmo de recuperación de información diseñado para medir la relevancia entre una consulta y un conjunto de documentos. Es muy utilizado en motores de búsqueda, debido a su eficacia para clasificar documentos según coincidencias léxicas respecto a una consulta.

Su funcionamiento se basa en tres factores principales:

- **Frecuencia de término (TF)**: mide cuántas veces aparece cada término de la consulta dentro de cada documento. Cuantas más apariciones, más relevante es considerado inicialmente el documento.
- **Frecuencia inversa de documento (IDF)**: mide qué tan común o raro es un término en todo el corpus de documentos. Los términos menos frecuentes (más raros) suelen tener más peso, porque aportan mayor valor informativo.
- **Longitud del documento**: el algoritmo normaliza los resultados para evitar que documentos largos tengan una ventaja injusta simplemente por contener más palabras. [31][36]

2.7 Tabla resumen

Como cierre de capítulo, se incluye la siguiente tabla resumen que presenta de forma sintetizada los principales puntos tratados. En ella se recogen de manera más concisa las soluciones existentes, los problemas identificados, los datos disponibles y las tecnologías empleadas.

Categoría	Resumen
Soluciones existentes	ASR: Sistemas como REDA que se encargan de verificar la precisión y coherencia de los readback en comunicaciones ATC. Agentes Generativos: Ejemplos como SERA que simula un entorno real de interacción entre piloto-controlador (entorno formativo) y Air-Guardian que actúa como un copiloto, monitorizando al piloto en sus funciones. RAG: Implementaciones como AviationGPT que permite un modelo generativo buscar en esa base de conocimiento las respuestas antes de generar la salida.
Problemas detectados	Latencia en comunicaciones en tiempo real Precisión y fiabilidad Complejidad del lenguaje aeronáutico: 1) Fraseología 2) Acentos y pronunciaciones 3) Términos especializados Entorno ruidoso e interferencias
Datos disponibles	Aviation Safety Network Taxonomía ADREP Mejores prácticas en comunicaciones por radio Terminología aeronáutica Transcripción de comunicaciones ATC reales
Tecnologías	Framework: LangChain y LangGraph Lenguaje: Python LLMs: OpenAI gpt-4o, Perplexity sonar, Mistral OCR Base de Datos vectorial: ChromaDB Interfaz Gráfica: Streamlit
LLMs	Definición y fundamentos teóricos: – LLMs: Sistemas generativos, normalmente basados en arquitectura Transformer para procesar y generar texto natural. – Arquitectura MoE: Empleada por modelos recientes (DeepSeek R1), mejora eficiencia computacional y rendimiento. Generación de información: – Los LLM generan texto probabilísticamente a partir de un prompt inicial, prediciendo tokens sucesivos según contexto.

	Ejemplos actuales: – GPT-o1 (OpenAI) – DeepSeek R1 (DeepSeek) – Claude 3.7 Sonnet – Entre otros… Aplicaciones: – Ingeniería/Programación – Asistentes Virtuales – Educación
RAG	Definición y arquitectura básica: – Técnica que combina recuperación de información y generación de texto. – Dos módulos: Recuperación (consulta información externa) y Generación (combina datos externos con conocimiento interno del LLM). – Reduce alucinaciones y mejora precisión en ámbitos específicos como aeronáutica. Tipos de implementaciones RAG: – Búsqueda semántica: Recuperación mediante embeddings vectoriales que capturan el significado semántico del texto. – Búsqueda híbrida: Combina búsqueda léxica (precisión exacta) con semántica. – MMR: Algoritmo que equilibra relevancia y diversidad, evitando redundancias. Agentic RAG: Usa LLMs para gestionar consultas de forma dinámica, combinando múltiples fuentes (Bases de datos vectoriales, APIs…) con un reranking final por relevancia. BM25: – Algoritmo de recuperación léxica basado en frecuencia de término (TF), frecuencia inversa del documento (IDF) y longitud del documento. – Muy eficaz en búsquedas específicas por términos exactos.

Tabla 2.1: Tabla Resumen

Capítulo 3

Diseño del Sistema

En este capítulo se describe el diseño del sistema, dividiendo su estructura en dos niveles principales: el Diseño de Alto Nivel (DAN) y el Diseño de Bajo Nivel (DBN). Esta estructura permite abordar, por un lado, la visión global del sistema y, por otro, los aspectos técnicos más específicos relacionados con la implementación de cada módulo.

El Diseño de Alto Nivel establece la estructura modular del sistema, define los flujos de datos y las responsabilidades de cada agente.

Por su parte, el Diseño de Bajo Nivel profundiza en los detalles técnicos de cada módulo, incluyendo estructuras de datos, algoritmos y decisiones de implementación concretas.

3.1 Diseño de Alto Nivel (DAN)

3.1.1 Metodología propuesta

Para desarrollar el sistema, se propone la siguiente metodología:

1. **Simulación de Conversaciones:** Se usarán transcripciones de comunicaciones piloto-controlador generadas por el sistema, que se procesando de manera secuencial, simulando el ritmo de una conversación natural.

2. **Procesamiento Incremental:** El sistema procesa cada segmento de la conversación conforme se recibe, analizando en tiempo real y aportando comentarios basados en la base de conocimiento del RAG.

3. **Interfaz de Usuario:** Se desarrollará una interfaz que mostrará, por un lado, la conversación en tiempo real y por otro, los comentarios y aportaciones generados por el sistema.

3.1.2 Estructura modular

Para la implementación del sistema se plantea una solución basada en los siguientes módulos:

1. **Módulo de preprocesamiento de datos:** Este módulo se encarga de aplicar técnicas de procesamiento de lenguaje natural (NLP) a los textos, para limpiarlos, segmentarlos y estructurarlos. Su función principal es preparar los datos de entrada para que puedan ser integrados de forma eficiente en el sistema RAG.

2. **Generador de Comunicaciones:** Se encarga de generar comunicaciones simuladas entre pilotos y controladores aéreos, basándose en protocolos reales como los de la FAA o ICAO. Utiliza modelos de lenguaje para generar automáticamente conversaciones realistas e incluye una lógica de inyección de errores (Más adelante, se usarán estos errores inyectados para compararlos con los errores detectados y comprobar la eficacia del programa).

3. **Módulo de Extracción de Términos y Fraseología:** Este módulo se encarga de analizar la transcripción original para identificar y extraer términos y fraseología específica de comunicaciones ATC. Por cada término y frase específica identificada, se generarán consultas eficientes que serán consumidas por el módulo RAG.

4. **Módulo de RAG:** Se implementa un sistema de Generación Aumentada por Recuperación híbrido. Este modulo cubre tanto la creación de la base de datos vectorial como el procesamiento de las consultas y la recuperación de los datos. A partir de las consultas generadas por el módulo de Extracción de términos y fraseología, se obtiene la información relevante a esas consultas a través de una base de datos vectorizada y la API de Perplexity.

5. **Módulo de Generación de Aportaciones:** Este módulo se encarga de ejecutar el pipeline global y generar aportaciones en tiempo real. Su función es simular una conversación ATC de forma progresiva, procesando los mensajes uno a uno como si fuera una conversación en streaming. Mientras se procesan los mensajes, el sistema realiza las siguientes tareas:
   1. Extracción de términos clave y generación de consultas.
   2. Recuperación de información a través del módulo RAG.
   3. Generación de aportaciones por parte de un LLM, el cual analiza cada mensaje, el historial conversacional y la información recuperada para ofrecer explicaciones, recomendaciones de buenas prácticas, y reflexiones sobre la seguridad de las comunicaciones ATC.

Al finalizar la conversación se realiza un cálculo de "scoring" de detección de errores, comparando los errores inyectados en la comunicación con los que hayan sido detectados durante el análisis. Esto permite evaluar la capacidad del sistema para detectar fallos en las comunicaciones.

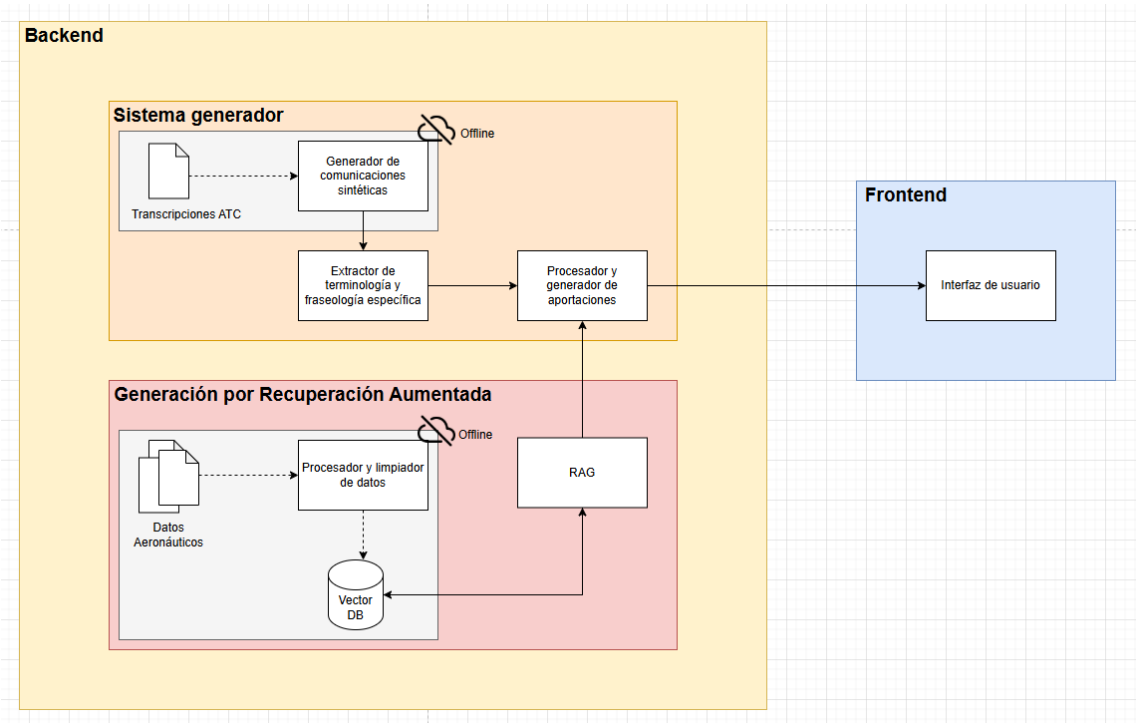


Figura 3.1: Diseño de Arquitectura

3.2 Diseño de Bajo Nivel (DBN)

3.2.1 Módulos y funciones internas

En este apartado se detallan los distintos módulos que componen el sistema, así como las funciones específicas implementadas en cada uno de ellos. Cada módulo ha sido diseñado de forma independiente y modular, siguiendo un enfoque funcional en Python, lo que permite una mayor claridad y agilidad de desarrollo. Se describen las responsabilidades principales de cada módulo, las funciones que lo componen y el orden lógico que siguen dentro del pipeline del sistema.

3.2.1.1 Módulo de Procesamiento de Datos

El módulo de procesamiento de datos constituye la primera fase del pipeline del sistema y tiene como objetivo transformar los datos brutos, provenientes de documentos PDF y archivos de texto, en un formato limpio, estructurado y normalizado, apto para su posterior procesamiento en las siguientes etapas. Para ello, se han desarrollado una serie de scripts independientes que trabajan de forma orquestada, garantizando la correcta preparación de la información.

El primer componente de este módulo es el archivo **manual_ocr.py**, encargado de extraer texto de documentos PDF utilizando la librería PyPDF2. El contenido extraído se guarda como archivos .txt dentro del directorio /data/manual_data, asegurando una conversión rápida y eficiente. Esta extracción es gestionada de manera robusta mediante la implementación de un sistema de logging que permite detectar y registrar posibles errores durante el proceso.

Adicionalmente, se desarrolló el script **mistral_ocr.py**, que realiza la extracción de texto a partir de documentos PDF mediante el uso de la API de OCR de Mistral AI. Esta solución sube los documentos al servicio de Mistral, obtiene una URL firmada y extrae el contenido en formato Markdown (.md), almacenándolo en el directorio /data/mistral_data. Durante el desarrollo, se evaluaron ambas técnicas de extracción (OCR manual y OCR mediante Mistral), concluyéndose que el método manual ofrecía mejores resultados en este caso concreto. La extracción realizada con PyPDF2 proporcionaba un texto más limpio y estructurado, sin introducir el formato adicional característico del contenido en Markdown, lo que facilitaba significativamente las tareas posteriores de procesamiento. Por ello, para el desarrollo principal del sistema se optó por utilizar el OCR manual como fuente de datos primaria.

Posteriormente, el archivo **normalizar_manual_data.py** procesa los archivos de texto manuales. Este script implementa dos funciones principales: procesar_fraseologia, que limpia las guías de fraseología aeronáutica eliminando líneas irrelevantes, numeraciones y normalizando los títulos para facilitar su identificación posterior; y procesar_vocabulario, que identifica acrónimos técnicos en el vocabulario aeronáutico y los asocia de forma estructurada con sus respectivas definiciones. Ambas funciones se integran en procesar_documentos, una función unificada que permite al usuario elegir qué tipo de documento desea procesar.

Una vez procesados los documentos iniciales, el archivo **NLP_Tools.py** aplica una serie de técnicas de procesamiento de lenguaje natural (NLP) para perfeccionar la limpieza y normalización del texto. Entre las funciones desarrolladas destacan la normalización ortográfica, la eliminación de ruido y caracteres especiales, la división del texto en fragmentos coherentes de tamaño limitado, y la depuración de entradas redundantes en los glosarios de términos técnicos. De forma integrada, la función normalizador ejecuta este pipeline de procesamiento de manera automática sobre todos los datos extraídos previamente, y guarda los resultados limpios en el directorio /data/clean_data.

Finalmente, el script **main.py** se encarga de orquestar todo el flujo de procesamiento. Este archivo ejecuta de forma secuencial la extracción manual de PDFs, la extracción mediante OCR avanzado, el procesamiento inicial de documentos, la normalización profunda del contenido, y la eliminación de entradas duplicadas en los glosarios. El uso de logging centralizado en cada etapa asegura un control detallado sobre el estado de la ejecución y facilita la depuración en caso de errores.

El módulo de procesamiento de datos permite unificar la información proveniente de distintas fuentes, normalizar su estructura textual para facilitar su uso posterior en procesos de recuperación aumentada de información (RAG), y garantizar una preparación robusta, modular y extensible de los datos que alimentarán al sistema.

3.2.1.2 Módulo Generador de Comunicaciones

El módulo Generador de Comunicaciones se encarga de la creación y preparación de conjuntos de comunicaciones aeronáuticas simuladas, siguiendo un enfoque estructurado que replica fielmente los protocolos y la terminología utilizada en las interacciones reales entre pilotos y controladores aéreos. Este módulo no solo genera nuevos escenarios de comunicaciones

basados en ejemplos reales, sino que también procesa datos de comunicaciones reales obtenidas de fuentes externas, adaptándolos al formato requerido por el sistema.

El proceso de generación se centra en el archivo **generador_comunicaciones.py**, donde se define la clase GeneradorComunicaciones. Esta clase utiliza un modelo de lenguaje (LLM) configurado a través de la biblioteca LangChain para generar escenarios completos de comunicaciones. La generación se realiza a partir de un conjunto de tipos de escenarios predefinidos (como "Emergency Landing", "IFR Departure" o "Engine Failure"), seleccionados aleatoriamente en cada instancia de generación. Cada escenario se construye siguiendo un PromptTemplate diseñado específicamente para garantizar que las comunicaciones sigan la fraseología estándar FAA/ICAO y respeten estrictas normas de formato y contenido.

El proceso de generación incluye, además, la inyección controlada de errores operacionales en los mensajes, tales como errores de altitud, rumbo, frecuencia o identificación de pista, de manera que se simule el tipo de equivocaciones humanas que podrían ocurrir en situaciones reales. Los mensajes generados se enriquecen con marcas de tiempo para mantener la coherencia temporal y se almacenan en archivos JSON, en el directorio comunicaciones_atc_procesadas.

Paralelamente, el archivo **preparador_comunicaciones.py** cumple la función de procesar las comunicaciones reales disponibles. A partir de archivos JSON ubicados en la carpeta comunicaciones_reales, este script transforma las conversaciones recopiladas de internet en un formato estructurado intermedio, adaptándolas a las necesidades del sistema. El resultado de este procesamiento se almacena en el directorio atc_structured_dataset, donde se encuentran los archivos atc_structured_dataset_EN.json y atc_structured_dataset_ES.json.

La estructura de carpetas de datos manejada en este módulo es la siguiente:

- **comunicaciones_reales/:** contiene comunicaciones_reales_EN.json y comunicaciones_reales_ES.json, que recogen ejemplos de comunicaciones reales extraídas de internet, en un formato inicial adaptado al sistema.
- **comunicaciones_atc_procesadas_prueba/:** contiene una primera versión del archivo comunicaciones_procesadas.json, utilizado durante las fases de prueba del generador para validar el correcto funcionamiento del pipeline de generación y almacenamiento.
- **comunicaciones_atc_procesadas/:** contiene la versión definitiva de comunicaciones_procesadas.json, donde se almacenan las comunicaciones generadas automáticamente por el sistema, siguiendo todos los requisitos de formato y contenido establecidos.
- **atc_structured_dataset/:** contiene atc_structured_dataset_EN.json y atc_structured_dataset_ES.json, que representan las comunicaciones reales en un formato anterior al procesamiento final, utilizado como base para la generación de ejemplos y entrenamiento de los modelos.

El flujo de trabajo del módulo comienza con la preparación de ejemplos de comunicaciones reales, procesados por el **preparador_comunicaciones.py**, para alimentar los prompts de generación. A partir de esos ejemplos, el **generador_comunicaciones.py** genera nuevas conversaciones simuladas, incorporando variabilidad y errores contextuales de forma controlada. Todo el

proceso es concurrente, utilizando ThreadPoolExecutor para acelerar la generación de múltiples escenarios en paralelo, y se asegura que los escenarios generados cumplen los requisitos mínimos de longitud y error para ser considerados válidos.

El módulo Generador de Comunicaciones proporciona así una fuente robusta, diversa y realista de conversaciones aeronáuticas, fundamentales para la posterior evaluación y entrenamiento de los sistemas de procesamiento de comunicaciones aeronáuticas desarrollados en este proyecto.

3.2.1.3 Módulo de Extracción de Términos y Fraseología

El módulo de Extracción de Términos y Fraseología está diseñado para analizar comunicaciones aeronáuticas simuladas, extrayendo automáticamente los términos técnicos relevantes y formulando preguntas relacionadas con las buenas prácticas de comunicación en aviación. Su propósito principal es enriquecer las conversaciones simuladas con información semántica útil, que posteriormente podrá ser utilizada en sistemas de recuperación aumentada de información (RAG) o en procesos de análisis y validación de las comunicaciones generadas.

La implementación se encuentra en el archivo **real_time_extractor.py**. El núcleo funcional del módulo reside en la función extract_terms_and_questions, que, dada una intervención individual de una conversación, formula dinámicamente un prompt al modelo de lenguaje (LLM) configurado con una temperatura baja para maximizar la precisión. El modelo responde extrayendo dos elementos clave:

- **Términos aeronáuticos relevantes**, presentados en forma de preguntas del tipo "what is the meaning of [término]?", adecuadas para su posterior indexación semántica.
- **Preguntas sobre buenas prácticas de comunicación ATC**, orientadas a fomentar la comprensión y la mejora de las comunicaciones en situaciones similares a la analizada.

El proceso de extracción asegura que cada intervención en la conversación pueda ser enriquecida con elementos adicionales de análisis y comprensión contextual.

Además, el módulo incluye una primera implementación de la función simulate_conversation, cuyo objetivo es simular dinámicamente el flujo de una conversación ATC con demoras realistas entre los mensajes. Aunque esta simulación en tiempo real es utilizada posteriormente en la interfaz de usuario desarrollada con Streamlit, en este apartado únicamente se menciona su existencia, dejando su explicación detallada para la sección correspondiente al frontend.

La estructura de procesamiento incluye también la carga de escenarios desde archivos JSON previamente generados y almacenados, y la gestión controlada de errores para asegurar la robustez del sistema.

Gracias a este módulo, el sistema no solo genera conversaciones aeronáuticas estructuradas, sino que también identifica automáticamente los conceptos clave mencionados y plantea preguntas que permiten ampliar el análisis de cada intercambio.

3.2.1.4 Módulo de RAG

El módulo de Recuperación Aumentada de Información (RAG) tiene como objetivo permitir al sistema buscar, recuperar y enriquecer información relevante relacionada con la terminología aeronáutica y las buenas prácticas de comunicación ATC, tanto a partir de bases internas vectorizadas como mediante consultas externas a la web. Este módulo constituye el núcleo del sistema de apoyo semántico, permitiendo responder preguntas específicas planteadas durante el flujo de procesamiento de comunicaciones.

Su implementación se encuentra en el archivo **rag.py**, el cual se estructura en torno a varios componentes principales:

En primer lugar, el sistema carga y procesa datos internos provenientes de documentos previamente normalizados. Se distinguen dos tipos de fuentes: por un lado, los glosarios técnicos, formados por listas de términos aeronáuticos (aviation_terminology_nlp.txt, combat_aviation_slang_nlp.txt y vocabularioAeroEI_nlp.txt); por otro, los documentos narrativos, que incluyen guías de buenas prácticas (best_practice_nlp.txt y guia_bp_fraseologia_y_comunicaciones_nlp.txt). Los glosarios se cargan directamente línea a línea como fragmentos individuales, mientras que los documentos narrativos se dividen en fragmentos utilizando el método de RecursiveCharacterTextSplitter para preservar la coherencia semántica.

A partir de estos datos, el módulo construye dos bases de datos vectoriales independientes utilizando la librería Chroma: una para los glosarios técnicos (chroma_glossary) y otra para los documentos narrativos (chroma_narrative). Las bases vectoriales permiten realizar búsquedas semánticas eficientes sobre grandes volúmenes de información, utilizando embeddings generados mediante la API de OpenAI.

Para responder a consultas, el módulo implementa mecanismos de búsqueda híbrida que combinan:

- **Búsqueda semántica** basada en la similitud de embeddings.
- **Búsqueda léxica** utilizando el modelo BM25 (Best Matching 25), que mide la coincidencia textual tradicional basada en la frecuencia de términos.

Los resultados obtenidos de ambas estrategias de búsqueda se fusionan y se depuran eliminando duplicados. Posteriormente, se aplica un reranking inteligente mediante el modelo de lenguaje, que selecciona de manera automática los fragmentos más relevantes y evita la redundancia en la respuesta final.

El módulo proporciona funciones específicas para consultar tanto los glosarios (query_glossary) como los textos narrativos (query_narrative), devolviendo siempre un conjunto optimizado de documentos relevantes. Además, se integra una función de búsqueda externa (web_search) que utiliza la API de Perplexity AI con el modelo sonar-pro, permitiendo al sistema realizar consultas web en tiempo real en caso de que la información requerida no esté disponible en las bases internas.

En conjunto, el módulo RAG dota al sistema de capacidades avanzadas de recuperación de información, combinando técnicas tradicionales y modernas de búsqueda, enriquecidas con la reordenación basada en LLMs, y permitiendo así

respuestas más precisas y contextualmente adecuadas en el entorno de las comunicaciones aeronáuticas.

3.2.1.5 Módulo de Generación de Aportaciones

El módulo de Generación de Aportaciones tiene como objetivo analizar las comunicaciones aeronáuticas simuladas para detectar posibles errores, confirmar la corrección de las transmisiones y ofrecer sugerencias de mejora basadas en buenas prácticas operacionales. Este módulo enriquece dinámicamente cada intervención con retroalimentación breve, precisa y contextualizada, fortaleciendo la capacidad del sistema para supervisar y mejorar la calidad de las comunicaciones ATC simuladas.

La implementación de este módulo se desarrolla en el archivo **generador.py**. Su componente principal es la función generar_aportacion, que, dada una intervención concreta dentro de una conversación, junto con el historial reciente de comunicaciones y las explicaciones recuperadas a través del módulo de RAG, genera una retroalimentación concisa. Esta retroalimentación puede:

- Confirmar la corrección de la comunicación.
- Sugerir mejoras en la fraseología o el protocolo.
- Identificar explícitamente errores de comunicación o de readback, siguiendo un formato estandarizado ("Possible Error: [descripción]") para facilitar su posterior procesamiento.

El prompt enviado al modelo de lenguaje (LLM) ha sido diseñado de forma específica para maximizar la detección de errores de forma breve y operativa, evitando respuestas extensas y priorizando la precisión. Además, cuando se detecta un posible error, este se registra en una lista independiente (detected_errors), lo que permite realizar un seguimiento estructurado de los fallos identificados en la conversación.

El módulo incluye también funciones de evaluación que permiten comparar los errores detectados automáticamente con los errores que fueron previamente inyectados en las simulaciones. Para ello se utilizan tres métodos de comparación:

- Similitud léxica simple, basada en la intersección de términos entre frases.
- Similitud BM25, que mide la relevancia textual tradicional.
- Similitud vectorial, basada en embeddings generados a partir del contenido textual.

La función compare_lists_hybrid integra estas métricas utilizando un enfoque ponderado, combinando la similitud vectorial y léxica, y calcula una puntuación final que representa la proporción de errores inyectados que han sido correctamente detectados por el sistema. Esta métrica ofrece una evaluación cuantitativa de la capacidad del sistema para identificar errores de comunicación, lo que resulta fundamental para validar su desempeño y optimizar su comportamiento durante el desarrollo.

En conjunto, el módulo de Generación de Aportaciones dota al sistema de una capacidad crítica para realizar un análisis en tiempo real de las comunicaciones ATC simuladas, proporcionando retroalimentación inmediata, evaluando su

corrección, y midiendo de forma cuantitativa la precisión de la detección de errores a través de un enfoque híbrido de comparación.

3.2.2 Interfaz de usuario

La interfaz de usuario del sistema ha sido diseñada para visualizar en tiempo real de las comunicaciones ATC simuladas, así como de las aportaciones generadas dinámicamente por el sistema multiagente. Su objetivo es permitir una interacción sencilla con el pipeline completo de procesamiento, mostrando tanto el flujo de mensajes como el análisis de las sugerencias y errores detectados.

La implementación de la interfaz se realiza en el archivo **ui.py**, utilizando la librería Streamlit para construir una aplicación web ligera. La interfaz actúa como un ejecutor del pipeline de procesamiento, ofreciendo al usuario controles para seleccionar un escenario, iniciar o cancelar la simulación en cualquier momento, y visualizar en paralelo tanto la conversación ATC como las aportaciones generadas.

El flujo de funcionamiento es el siguiente:

1. Al iniciar la aplicación, se cargan los escenarios de comunicaciones simuladas desde un archivo JSON.
2. El usuario puede seleccionar el escenario que quiere procesar mediante un menú desplegable.
3. Una vez seleccionado el escenario, el usuario puede iniciar la simulación presionando el botón "Ejecutar Pipeline". Durante la ejecución, se habilita también la posibilidad de cancelar el proceso en cualquier momento.
4. La conversación simulada se reproduce de manera progresiva, introduciendo demoras realistas entre los mensajes para simular el paso natural del tiempo. Cada intervención es visualizada inmediatamente en un área de texto dedicada a la Conversación ATC.
5. En paralelo, para cada intervención:
   - Se extraen términos y preguntas relevantes utilizando el módulo de Extracción de Términos y Fraseología.
   - Se realiza una búsqueda de información complementaria mediante consultas a la API de Perplexity y/o búsquedas RAG internas).
   - Se genera una aportación personalizada a través del módulo de Generación de Aportaciones, que analiza la intervención, su contexto histórico, y el conocimiento adicional recuperado.
   - Estas aportaciones se presentan de forma separada en un área dedicada a las Aportaciones del Agente.
6. Al finalizar la conversación, si existen errores inyectados en el escenario, se calcula una puntuación final de desempeño basada en la comparación entre los errores detectados por el agente y los errores realmente inyectados.

3.2.3 Gestión de datos y almacenamiento

En este apartado se describe cómo se estructuran y almacenan los distintos tipos de datos generados: desde las conversaciones simuladas y las aportaciones generadas, hasta las consultas al RAG y los resultados del sistema.

También se especifican los formatos utilizados, los métodos de almacenamiento, y la estructura de carpetas y archivos utilizada.

Módulo de Procesamiento de Datos

Tipo de Dato	Ubicación	Formato	Descripción
Textos extraídos de PDF (OCR manual)	/data/manual_data/	.txt	Texto plano extraído directamente del PDF. Cada archivo corresponde a un PDF original.
Textos extraídos de PDF (Mistral OCR)	/data/mistral_data/	.md	Formato Markdown generado por Mistral: mezcla de texto estructurado, tablas, encabezados Markdown (#, ##), posibles enlaces.
Textos procesados y normalizados	/data/clean_data/	.txt	Texto limpio, todo en minúsculas, sin caracteres especiales, sin líneas vacías. Fragmentado si es necesario. Archivos como vocabularioAeroEI_nlp.txt, aviation_terminology_nlp.txt, etc. (Archivos con terminación _nlp han sido procesados y están listos para su uso)

Tabla 3.1: Gestión de Datos y Almacenamiento - Módulo de Procesamiento de Datos

Módulo de Generador de Comunicaciones

Tipo de Dato	Ubicación	Formato	Descripción
Comunicaciones simuladas generadas	/Generador_Comunicaciones/comunicaciones_atc_procesadas/comunicaciones_procesadas.json	.json	Lista de escenarios, cada uno contiene: scenario_id (string), scenario (tipo), callsign (identificador de vuelo), phases (etapas de vuelo), messages (lista de mensajes), errores (errores inyectados opcionales).
Comunicaciones reales (input original)	/Generador_Comunicaciones/comunicaciones_reales/comunicaciones_reales_EN.json	.json	Formato menos estructurado. Contiene: steps, transmisiones (cada step), speaker, message.
Comunicaciones reales formateadas (dataset estructurado)	/Generador_Comunicaciones/atc_structured_datasets/atc_structured_dataset_EN.json	.json	Cada entrada tiene: id (número), texto (texto continuo de la conversación), metadata (número de mensajes).
Comunicaciones simuladas de Prueba (Estas comunicaciones de prueba son mas reducidas, aumentando la velocidad de procesameinto para realizar pruebas)	/Generador_Comunicaciones/comunicaciones_atc_procesadas/comunicaciones_procesadas_prueba.json	.json	Lista de escenarios, cada uno contiene: scenario_id (string), scenario (tipo), callsign (identificador de vuelo), phases (etapas de vuelo), messages (lista de mensajes), errores (errores inyectados opcionales).

Tabla 3.2: Gestión de Datos y Almacenamiento - Módulo de Generador de Comunicaciones

Módulo de Extracción de Términos y Fraseología

Tipo de Dato	**Ubicación**	**Formato**	**Descripción**
Términos y preguntas extraídos	--- No se almacena	En memoria	Lista de términos técnicos (what is the meaning of X?) y preguntas sobre buenas prácticas, generadas dinámicamente por el LLM.

Tabla 3.3: Gestión de Datos y Almacenamiento - Módulo de Extracción de Términos y Fraseología

Módulo de RAG

Tipo de Dato	Ubicación	Formato	Descripción
Bases de datos vectoriales de glosarios	/data/chroma_db/ chroma_glossary/	Interno ChromaDB (embedding persistente)	Cada fragmento de glosario es almacenado vectorizado usando embeddings de OpenAI. Cada glosario tiene un formato diccionario, con terminolgoía aeronáutica relacionada con su respectivo significado.
Bases de datos vectoriales de narrativa	/data/chroma_db/ chroma_narrative/	Interno ChromaDB (embedding persistente)	Cada fragmento de los textos narrativos es almacenado vectorizado usando embeddings de OpenAI. Los textos tienen información de manuales y textos aeronauticos, como buenas prácticas de comunicación y guía generales.
Resultados de búsquedas RAG	--- No se almacena	En memoria	Documentos recuperados para cada consulta, no persistidos en disco.
Resultados de búsquedas Perplexity	--- No se almacena	En memoria	Respuesta a la pregunta hecha mediante web search, usada en ejecución pero tampoco guardada.

Tabla 3.4: Gestión de Datos y Almacenamiento - Módulo de RAG

Módulo de Generación de Aportaciones

Tipo de Dato	Ubicación	Formato	Descripción
Aportaciones generadas	--- No se almacena	En memoria	Texto generado para cada intervención: sugerencias, detección de posibles errores y confirmaciones.
Errores detectados	--- No se almacena	En memoria	Lista de errores detectados por el agente comparados con los errores inyectados para calcular el scoring.

Tabla 3.5: Gestión de Datos y Almacenamiento - Módulo de Generación de Aportaciones

Capítulo 4

Desarollo

En este capítulo se describe cómo se llevó a cabo el desarrollo de cada uno de los módulos del sistema. Se explica su implementación, las pruebas realizadas, los principales problemas encontrados durante el proceso y las mejoras aplicadas.

4.1 Agente Procesador de Datos

Este módulo se encarga de preparar los documentos técnicos que luego se usarán en el RAG. Parte de documentos en bruto (como PDFs) y, tras aplicar varios pasos de extracción y limpieza, deja los datos listos y en el formato adecuado.

4.1.1 Implementación

El procesamiento de los datos se estructuró en varios pasos consecutivos:

1 Extracción de texto desde PDFs

Primero, se plantearon dos formas de extraer el texto, una de forma manual y otra, haciendo uso de la herramienta de inteligencia artificial de origen Francés, Mistral AI:

- **OCR Manual** (manual_ocr.py): utilizando PyPDF2, se leía directamente el contenido textual de cada PDF y se guardaba como .txt.

```
texto = "".join([pagina.extract_text() for pagina in lector.pages if pagina.extract_text()])
```

- **OCR con Mistral AI** (mistral_ocr.py): enviando los PDFs a la API de Mistral OCR para convertirlos a formato Markdown (.md).

```
ocr_response = client.ocr.process(
    model="mistral-ocr-latest",
    document={"type": "document_url", "document_url": signed_url.url},
)
```

Después de probar ambos métodos, se decidió quedarse solo con el OCR manual, ya que los textos extraídos eran más limpios, consistentes y mucho más fáciles de procesar en los siguientes pasos.

2 Procesamiento de texto y limpieza inicial

Una vez extraído el texto, se aplicaron varias funciones generales de NLP que se enucentran en NLP_Tools.py:

Normalización de texto (convertir comillas inteligentes, minúsculas, eliminar saltos extra, etc.)

```
content = (
        content.replace("’", "'")
        .replace("‘", "'")
        .replace("“", '"')
        .replace("”", '"')
    )
    lines = [
        line.strip()
        for line in content.splitlines()
        if line.strip() != ""      ]
```

Eliminación de ruido (quitar imágenes, tablas vacías, fragmentos de HTML...).

```
patterns = [
        r'\|\s*:--:?\s*\|',
        r'\|.*\|',
        r'\!\[.*?\]\(.*?\)',
        r'<.*?>',
        r'http\S+',
        r'\*{3,}',
        r'\n{3,}',
    ]
    cleaned = content
    for pattern in patterns:
        cleaned = re.sub(pattern, '', cleaned,
flags=re.DOTALL|re.MULTILINE)
    return cleaned.strip()
```

Eliminación de caracteres especiales.

```
allowed_chars = r'[^a-zA-Z0-9ñÑáéíóúÁÉÍÓÚ\s.,;:¿¡?!()\-—\[\]{}"\'…]'
```

```
return re.sub(allowed_chars, '', content)
```

3 Adaptación específica para los datos de fraseología y vocabulario

Se crearon funciones específicas en normalizar_manual_data.py, dependiendo del tipo de documento.

Para la guía de fraseología se limpiaron paginaciones, títulos, documentos públicos y se marcaron los encabezados importantes (###). Por otro lado, para el vocabulario aeronáutico se detectaron acrónimos y se les dió formato tipo: "{Término} means: {Significado}"

4 Automatización con un pipeline

Todos los pasos anteriores se conectaron en un solo pipeline (main.py) para que se pudieran ejecutar de forma automática:

```
def ejecutar_pipeline(debug=True):
    manual_ocr()
    mistral_ocr()
    procesar_documentos(tipo="ambos")
    normalizador()
    remove_duplicate_acronyms_across_glossary_files(path)
```

De esta manera, en un solo comando se extraen los textos, se normalizan, se adaptan y se dejan listos en la carpeta data/clean_data para ser usados en las bases vectoriales.

4.1.2 Pruebas

Durante el desarrollo del módulo de procesamiento de datos, se llevaron a cabo varias pruebas para validar cada etapa del flujo de trabajo. Al principio, se compararon los resultados de extraer texto desde los PDFs usando dos métodos distintos: uno mediante el OCR manual con PyPDF2 (manual_ocr.py) y otro usando el modelo mistral-ocr-latest a través de la API oficial (mistral_ocr.py). Para eso se eligieron varios documentos técnicos reales y se procesaron por ambos métodos.

Los textos extraídos se compararon visualmente y también se pasaron por las funciones de limpieza para ver qué tan bien se adaptaban al siguiente paso (vectorización). En esas pruebas se observó que el texto del OCR manual venía más limpio, con menos errores de interpretación y sin tanto ruido en general, así que se optó por usar ese.

Después, se probaron todas las funciones de limpieza (normalize_text, remove_noise, remove_special_characters, etc.) para asegurarse de que funcionaban correctamente y dejaban el texto preparado. En paralelo, se hicieron también pruebas con los procesadores específicos de fraseología y vocabulario (normalizar_manual_data.py) para confirmar que se aplicaban las transformaciones necesarias (por ejemplo, detectar acrónimos, quitar encabezados innecesarios o estandarizar estructuras tipo "AIM means: …").

Finalmente, se ejecutó el main.py, que contiene todo el pipeline, en distintos entornos. Se probó con varios documentos en bruto y se verificó que tras completar el proceso, los archivos terminaban bien formateados en data/clean_data, listos para alimentar al sistema RAG.

A la hora de mostrar las pruebas por pantalla, se hace uso de la librería "logging", la cual actúa como un "print" siempre que el código se ejecute con la flag –debug:

```
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(module)s - %(message)s",
    handlers=[logging.StreamHandler()]
)
…
logging.info("Iniciando procesamiento manual de PDFs...")
```

4.1.3 Problemas detectados

Durante las primeras pruebas con los modelos OCR, se observó que el resultado del procesamiento con Mistral contenía demasiado ruido: bloques innecesarios de Markdown, encabezados repetidos, enlaces vacíos y formatos que rompían la estructura semántica del documento. Aunque el modelo funcionaba bien para otros casos de uso, en este proyecto en concreto dificultaba el postprocesado y la vectorización posterior.

También hubo casos en los que algunos PDFs contenían fragmentos mezclados en inglés y español, lo que generaba inconsistencias al normalizar el texto. Además, las comillas "inteligentes" (como ‘ ’ o “ ”) rompían la tokenización en varios puntos del pipeline, lo que hacía que ciertas líneas no se procesaran correctamente si no se limpiaban bien.

En cuanto a los datos de vocabulario, se detectó que había varios acrónimos duplicados o definidos de forma distinta en archivos diferentes. Esto provocaba ruido semántico al vectorizar, ya que el mismo término aparecía más de una vez con descripciones distintas. También hubo casos en los que el formato no seguía un patrón claro, por lo que no era fácil convertirlos automáticamente al estilo "{acronym} means: definition".

4.1.4 Mejoras

Una de las primeras decisiones que se tomó fue el hecho de no seguir adelante con el OCR de Mistral y quedarse con el procesamiento manual de PDFs, ya que ese método ofrecía resultados mucho más limpios para el tipo de documentación que se estaba manejando. Aunque el OCR automático era más atractivo en cuanto a automatización, el nivel de ruido que generaba no era conveniente.

Para resolver los problemas de limpieza y normalización, se diseñó un conjunto de funciones específicas de NLP que pudieran aplicarse de forma segura a todos los documentos. Entre ellas destacan normalize_text, remove_noise y

remove_special_characters, que se fueron afinando tras cada prueba para que dejaran el texto lo más neutro posible, sin perder contenido relevante.

En el caso de los documentos técnicos más sensibles, como los glosarios y la guía de fraseología, se creó un script específico (normalizar_manual_data.py) que procesaba cada tipo de archivo de forma diferente, adaptando su estructura al formato necesario para el RAG. Esto incluyó desde el formateo de encabezados hasta la detección y reescritura de definiciones de acrónimos.

Finalmente, para asegurarse de que no hubiera duplicados entre archivos distintos de vocabulario, se implementó una función (remove_duplicate_acronyms_across_glossary_files) que recorría todos los textos finales y se quedaba con una sola versión por acrónimo. Esto fue clave para evitar conflictos semánticos al generar la base vectorial final.

4.2 Agente Generador de Comunicaciones

Este módulo se encarga de crear comunicaciones aeronáuticas simuladas de forma realista, generando tanto las conversaciones como posibles errores de comunicación de manera controlada. La idea es construir escenarios creíbles para después poder probar el resto de agentes sobre datos sintéticos pero muy cercanos a la realidad.

4.2.1 Implementación

1 Preparación de los datos de ejemplo

Antes de poder generar comunicaciones nuevas, el agente necesita basarse en ejemplos reales para que las conversaciones que cree sean lo más realistas posible.

Estos ejemplos se procesan a partir de archivos JSON situados en la carpeta atc_structured_datasets/. Esos archivos (atc_structured_dataset_EN.json, por ejemplo) vienen de haber transformado comunicaciones reales a un formato estructurado y limpio. Esta transformación previa se hace en el archivo (preparador_comunicaciones_sinteticas.py).

Cada entrada contiene un campo texto, que es la conversación estructurada en pasos. Al arrancar el agente, se cargan algunos de estos ejemplos al azar para introducirlos dentro del prompt que se envía al modelo de lenguaje:

```
self._ejemplos_cache = random.SystemRandom().sample([e['texto'] for e
in en_data], 4)
```

Así conseguimos que las generaciones no partan de cero, sino que siempre se basen en conversaciones reales.

2 Generación de escenarios

El flujo de generación sigue varios pasos:

- **Selección aleatoria del tipo de escenario**: se escoge un tipo de situación (despegue IFR, emergencia médica, fallo de motor…).

- **Construcción del prompt**: se genera un prompt que incluye instrucciones estrictas, los ejemplos cargados, y reglas sobre la fraseología a usar (FAA/ICAO), el formato de los números, la estructura de las comunicaciones, etc.
- **Llamada al modelo LLM**: se envía el prompt al modelo y se recibe un bloque de texto en formato JSON que representa el escenario completo.
- **Procesamiento del output**: se limpia el JSON recibido para asegurarse de que no hay ruido.

```
clean_json = re.sub(r'^.*?\{', '{', raw_output, 1, re.DOTALL)
clean_json = clean_json.rsplit('}', 1)[0] + '}'
```

- **Inyección de errores**: ya con el escenario parseado, se recorren los mensajes de los pilotos y se inyectan errores de forma controlada. El sistema decide aleatoriamente si inyectar o no un error basándose en una probabilidad (0.2 después del primer mensaje) y, si se decide hacerlo, modifica la altitud, heading, frecuencia o pista de forma coherente.
  Por ejemplo, para modificar un heading, se aplica una desviación aleatoria:

```
modificado = re.sub(r'\b(heading|course)\s+(\d{3})\b',
    lambda m: f"{m.group(1)} {(int(m.group(2)) + random.randint(-10, 10)) % 360}", modificado)
```

- **Añadido de timestamps**: se añade una marca temporal realista a cada mensaje para simular la evolución temporal de la conversación.

Almacenamiento y generación paralela

Una vez generados, los escenarios se almacenan en un archivo JSON llamado comunicaciones_procesadas.json dentro de la carpeta comunicaciones_atc_procesadas/. Cada ejecución añade varios escenarios nuevos.

Para acelerar todo este proceso, el agente utiliza ejecución paralela con ThreadPoolExecutor. Se lanzan varias llamadas simultáneas al modelo de lenguaje, cada una generando su propio escenario en paralelo:

```
with ThreadPoolExecutor(max_workers=2) as executor:
    futures = [executor.submit(generador.generar_escenario) for _ in range(num_escenarios)]
```

Además, como varios hilos pueden querer añadir su escenario al mismo tiempo, se usa un Lock para proteger esa sección crítica:

```
with generador.lock:
    escenarios.append(escenario)
```

Características de las comunicaciones generadas

Cada comunicación generada sigue la fraseología oficial establecida por organismos como la FAA o la ICAO. Los mensajes suelen contener entre 8 y 12 intercambios y están divididos por fases operacionales, como taxi, despegue o aproximación. Todos los números aparecen en formato digital, por ejemplo

"FL180" en lugar de "one-eight-zero", y se evitan expresiones ambiguas para asegurar claridad. Además, los errores que se inyectan en estas comunicaciones se diseñan para que sean lo más realistas posible.

4.2.2 Pruebas

Durante el desarrollo del módulo de generación de comunicaciones se realizaron diferentes pruebas para verificar la correcta creación de escenarios ATC sintéticos.

Las pruebas principales se llevaron a cabo ejecutando el script generador_comunicaciones.py directamente en modo --debug:

```
python generador_comunicaciones.py --num 5 --debug
```

Esto permitía observar en consola tanto los escenarios generados como los posibles errores detectados en tiempo real gracias al logging.

Se verificó que todos los escenarios siguieran el formato JSON requerido, que en escenarios normales se generaran errores inyectados de manera controlada en los mensajes del piloto y que el número de mensajes y errores fuera consistente (se filtraban escenarios que no cumplieran el estándar mínimo).

Además, se midió el tiempo de generación de escenarios, ya que se ejecutaba en paralelo usando ThreadPoolExecutor, con la siguiente estructura:

```
with ThreadPoolExecutor(max_workers=2) as executor:
    futures = [executor.submit(generador.generar_escenario) for _ in
range(num_escenarios)]

    for future in as_completed(futures):
        escenario = future.result()
        …
```

4.2.3 Problemas detectados

A lo largo del desarrollo se detectaron varios problemas que fue necesario corregir. Uno de ellos fueron los problemas en la inyección de errores: no siempre se inyectaban errores en los mensajes del piloto, lo que hacía que algunos escenarios no fueran válidos para pruebas. También se encontraron condiciones de carrera, ya que múltiples hilos generaban escenarios al mismo tiempo, y era posible que varias modificaciones simultáneas generasen una condición de carrera en el array final. Esto se solucionó protegiendo el acceso con un Lock. Por último, hubo respuestas del LLM con ruido: el modelo, en ocasiones, devolvía la estructura JSON mezclada con otros fragmentos de conversación o instrucciones, lo que hacía que el json.loads() fallara.

4.2.4 Mejoras

- Filtro de calidad: Se tuvo que ajustar la probabilidad de inyección (prob = 0.2) y añadir un filtro que rechazara escenarios sin errores o con muy pocos mensajes:

```
if len(escenario["messages"]) >= 7 and
len(escenario["errores"]) > 0:
    escenarios.append(escenario)
```

- Protección contra concurrencia: Para evitar conflictos entre hilos al generar en paralelo, se utilizó un Lock en la zona de escritura:

```
with generador.lock:
    escenarios.append(escenario)
```

- Limpieza previa del JSON: Antes de intentar decodificar la respuesta del modelo, se implementó un proceso de limpieza para eliminar texto sobrante:

```
clean_json = re.sub(r'^.*?\{', '{', raw_output, 1, re.DOTALL)
clean_json = clean_json.rsplit('}', 1)[0] + '}'
scenario = json.loads(clean_json)
```

4.3 Agente Extractor de Términos y Fraseología

Este agente se encarga de analizar cada mensaje de la conversación ATC y extraer tanto los términos técnicos relevantes como preguntas orientadas a buenas prácticas de comunicación aeronáutica.

4.3.1 Implementación

El funcionamiento del agente se basa en una única función principal: extract_terms_and_questions, que recibe una intervención (es decir, una línea de conversación ATC) y devuelve dos listas: una de términos y otra de preguntas. Internamente, esta función construye un prompt especializado que se envía a un modelo LLM con temperatura baja para maximizar la precisión:

```
prompt = f"""
    You are an aviation communication expert.
    Analyze the following ATC communication line:
    "{intervention}"
    1. Identify and list any aviation-specific terminology...
    2. Then, based on what is happening in the message...
    Respond using the following format STRICTLY:
    Términos:
    - what is the meaning of term1?
```

```
    …
"""
```

La respuesta generada por el modelo se parsea línea a línea, separando las secciones Términos y Preguntas en listas independientes. El diseño del prompt está cuidadosamente ajustado para que las preguntas sean adecuadas para una búsqueda semántica, lo cual es clave para su uso posterior en el sistema RAG.

4.3.2 Pruebas

Durante las pruebas iniciales se ejecutó el módulo en modo script usando el argumento --debug, lo que permitía observar en consola las salidas del modelo. Esto ayudó a verificar tanto la detección de términos como la calidad de las preguntas generadas. Por ejemplo:

```
python real_time_extractor.py SCN_20250422104530 --debug
```

También se validó que el modelo respetara correctamente el formato de salida exigido, y que las preguntas fueran útiles para extraer conocimiento desde el sistema RAG.

En general, los resultados fueron satisfactorios, aunque se detectaron algunos errores de segmentación en casos poco comunes (como mensajes demasiado cortos o con errores gramaticales).

4.3.3 Problemas detectados

En algunas pruebas, el modelo no devolvía correctamente las secciones “Términos” y “Preguntas”, especialmente si el mensaje original era muy ambiguo o contenía errores gramaticales. Esto causaba que el parseo fallara o que se generaran listas vacías.

Además, se observó que el número de preguntas generadas no siempre era consistente. Algunas intervenciones daban lugar a más preguntas de las necesarias o incluso a preguntas redundantes, lo que podía aumentar innecesariamente la carga del sistema RAG.

Por último, los términos detectados no siempre coincidían con terminología aeronáutica relevante. En algunos casos, el modelo interpretaba palabras comunes como términos técnicos.

4.3.4 Mejoras

Para mejorar la estabilidad del sistema se afinó el diseño del prompt, especialmente reforzando la parte de formato requerido con la palabra “STRICTLY” y ejemplos muy claros. Esto aumentó notablemente la tasa de parseo correcto.

Además, se ajustó la temperatura del modelo a 0.4 para equilibrar entre precisión y diversidad. En versiones anteriores se usaba una temperatura más baja que generaba salidas muy repetitivas o conservadoras.

4.4 Agente RAG

Este módulo se encarga de responder preguntas relacionadas con fraseología o buenas prácticas ATC a partir de documentos previamente procesados. El objetivo es que el sistema pueda dar explicaciones contextualizadas a partir de una base de conocimiento técnica, sin depender únicamente del modelo de lenguaje.

4.4.1 Implementación

El diseño del sistema sigue un enfoque híbrido: primero se realiza una recuperación semántica con embeddings, luego una búsqueda léxica con BM25, y finalmente se reordena todo con ayuda de un modelo de lenguaje en lo que se conoce como un reranking.

1 Carga y preparación de documentos

Para los textos narrativos (como guías o manuales), se hace un troceado automático para que los fragmentos tengan un tamaño manejable:

```
splitter = RecursiveCharacterTextSplitter(chunk_size=1200,
chunk_overlap=300)
raw_docs = loader.load()
docs.extend(splitter.split_documents(raw_docs))
```

Los glosarios, por otro lado, se cargan línea a línea directamente:

```
for file_name in glossary_files:
    with open(clean_data_dir / file_name, encoding="utf-8") as f:
        for line in f:
            docs.append(Document(page_content=line.strip()))
```

2 Creación de bases vectoriales

Una vez cargados los documentos, se generan las bases de datos de vectores con Chroma:

```
Chroma.from_documents(
    documents=glossary_chunks,
    embedding=embeddings,
    persist_directory=str(chroma_glossary_dir)
)
```

3 Búsqueda semántica y léxica

Cuando se lanza una consulta, primero se obtienen los resultados más parecidos usando similitud de embeddings:

```
semantic_results = glossary_db.similarity_search(prompt, k=6)
```

Y también se hace una búsqueda por coincidencia de palabras clave usando BM25:

```
def bm25_search(query: str, documents: list[Document], top_k: int = 5):
    tokenized_corpus = [doc.page_content.lower().split() for doc in documents]
    bm25 = BM25Okapi(tokenized_corpus)
    tokenized_query = query.lower().split()
    scores = bm25.get_scores(tokenized_query)

    sorted_docs = sorted(zip(documents, scores), key=lambda x: x[1], reverse=True)
    return [doc for doc, _ in sorted_docs[:top_k]]
```

Acto seguido, combinamos las dos listas de voctores, eliminamos duplicados, y nos quedamos con los vectores que satisfagan la búsqueda vectorial y semántica.

```
semantic_results = glossary_db.similarity_search(prompt, k=k)
lexical_results = bm25_search(prompt, glossary_chunks, top_k=k)

combined = semantic_results + lexical_results
seen = set()
unique_results = []
for doc in combined:
    if doc.page_content not in seen:
        unique_results.append(doc)
        seen.add(doc.page_content)
```

4 Reranking con LLM

Los fragmentos recuperados se fusionan, eliminando duplicados, y luego se le pide al modelo que seleccione los más útiles en función de la pregunta:

```
rerank_prompt = (
    f"The user asked:\n\"{prompt}\"\n\n"
    f"Here are some retrieved passages:\n{context}\n\n"
```

```
    f"Pick exactly {top_k} of the most relevant and non-redundant
passages…"
)
response = llm.invoke(rerank_prompt)
```

5 Búsqueda Web

Para complementar la base de conocimiento local del RAG, el agente recurre a una búsque en web con el modelo “sonar” de Perplexity. Cuando llega una pregunta al módulo, además de responderla con la base interna, se lanza también a Perplexity para aumentar la fiabilidad y eficacia de la búsqueda.

Primero se envía un mensaje de sistema definiendo el rol ("You are a concise expert assistant specialized in aviation terminology and ATC communication practices.")

Luego se manda el mensaje con la consulta, que devuelve un texto breve con la respuesat encontrada.

```
def web_search(query: str, debug: bool = False) -> str:
    messages = [
        {
            "role": "system",
            "content": "You are a concise expert assistant specialized
in aviation terminology and ATC communication practices.",
        },
        {
            "role": "user",
            "content": f"Search and answer concisely: \"{query}\"",
        },
    ]
```

4.4.2 Pruebas

Durante el desarrollo se probaron diferentes prompts y consultas manuales desde el propio rag.py, dentro del bloque __main__. Por ejemplo:

```
glossary_results = query_glossary("What is the meaning of agl and the
meaning of feathering")
narrative_results = query_narrative("what is the best practice for
communicating when landing and how to communicate during takeoff")
```

Y también se incluyó una búsqueda web con el modelo Sonar de Perplexity:

```
query = "what is the best practice for communicating when landing and
how to communicate during takeoff"

respuesta = web_search(query, debug=True)
```

Estas pruebas permitieron verificar que los resultados fueran coherentes, variados y útiles. También sirvieron para ajustar el número de resultados (top_k) y afinar los filtros de duplicados.

4.4.3 Problemas detectados

Uno de los problemas principales fue que, aunque la búsqueda semántica funcionaba bien en general, a veces devolvía fragmentos poco relevantes o directamente irrelevantes para la consulta. Esto ocurría especialmente con preguntas muy abiertas o mal formuladas. Además, en varios casos aparecían fragmentos repetidos o demasiado parecidos entre sí, lo que restaba variedad y utilidad a las respuestas.

Otro problema fue que el reordenamiento final con el LLM, aunque ayudaba bastante a priorizar los fragmentos más útiles, a veces reformulaba ligeramente el texto, lo que dificultaba volver a localizar el documento original exacto si se necesitaba recuperar el contexto completo. También notamos que el tiempo de respuesta podía aumentar bastante cuando se acumulaban muchas preguntas seguidas, sobre todo al combinar recuperación semántica, recuperación léxica y reordenamiento con LLM todo en una sola petición.

4.4.4 Mejoras

Para tratar de mitigar estos problemas, se tomaron varias decisiones que mejoraron tanto la precisión como el rendimiento. Por un lado, se pasó a combinar la búsqueda semántica (con ChromaDB) con la búsqueda léxica BM25. Esto permitió recuperar fragmentos más diversos y relevantes, especialmente útiles en preguntas más técnicas o con términos específicos.

También se implementó un sistema de filtrado que elimina fragmentos repetidos antes de pasarlos al LLM para el reordenamiento. De esta forma, se evitan repeticiones innecesarias en la respuesta final.

Además se ajustaron los valores de top_k para no enviar más información de la necesaria al modelo.

Por último, se afinó el prompt del reordenamiento para pedir explícitamente al LLM que devuelva los fragmentos tal cual, sin modificar su redacción, con el objetivo de poder emparejarlos de forma fiable con los documentos originales. Esto todavía no es perfecto, pero mejoró bastante la consistencia.

4.5 Agente Generador de Aportaciones

Este módulo tiene como objetivo analizar en tiempo real cada intervención de una conversación ATC y generar una aportación breve que indique si la

comunicación es correcta o si hay errores, malas prácticas o ambigüedades. Además, identifica de forma explícita posibles errores si los detecta.

4.5.1 Implementación

El agente utiliza como entrada tres fuentes: la intervención actual, el historial reciente de la conversación, y una serie de explicaciones generadas por el sistema RAG o Perplexity. Con esa información, genera un comentario breve y, si corresponde, una línea que comienza con "Possible Error:". Este formato facilita el análisis automático posterior.

Una parte clave del proceso es la construcción del prompt dinámico, que se le pasa al modelo:

```
prompt = f"""
…
Current communication:
\"{intervencion}\"
Recent conversation history:
{historial_texto}
Reference knowledge (from RAG):
{contexto_rag}
…
"""
```

Una vez generada la respuesta, se analiza si contiene alguna línea con el patrón "Possible Error:". Si es así, se extrae y se añade a una lista de errores detectados:

```
if re.match(r'(?i)^possible error\s*:', line):
    error_text = line.split(":", 1)[1].strip()
    if error_text:
        detected_errors.append(error_text)
```

Además, para evaluar la capacidad del sistema para detectar errores, se implementó una métrica basada en similitud entre los errores detectados y los inyectados:

```
def compare_lists_hybrid(detected_errors, injected_errors, ...):
    …
    total_sim = vector_weight*sim_vector + lexical_weight*sim_lex
    if total_sim > best_sim:
                best_sim = total_sim

if best_sim >= threshold:
```

```
            matches += 1
…
if matches > len(injected_errors):
        ratio = 1.0
    else:
        ratio = matches / len(injected_errors)
    return ratio
```

Para evaluar la capacidad del agente de encontrar errores en las comunicaciones, cada descripción de error (tanto inyectada como detectada) se convierte primero en un vector mediante OpenAIEmbeddings.

```
injected_embeds = embedder.embed_documents(injected_errors)
detected_embeds = embedder.embed_documents(detected_errors)
```

A partir de esos vectores, se calcula la similitud coseno, la cual arroja un valor entre 0 y 1 que cuantifica lo parecidos que son dos vectores en el espacio de embeddings.

```
sim_vector = cosine_similarity(d_vec, i_vec)
```

Simultáneamente, se tokeniza el texto de los errores y se construye un índice BM25 para medir la coincidencia léxica. El score máximo obtenido en BM25 se normaliza también al rango 0–1 para poder compararlo mas adelante.

```
scores = bm25.get_scores(tokenized_query)
sim_lex = max(scores) / max(bm25.idf.values())
```

A continuación combinamos las medidas semántoca y léxica en una sola puntuación híbrida con una ponderación de 70 % para la parte semántica y 30 % para la parte léxica.

```
total_sim = 0.7 * sim_vector + 0.3 * sim_lex
```

Cuando el score híbrido supera un umbral (por ejemplo, 0.7), ese error detectado se considera correctamente detectado o “relacionado” con uno inyectado. Al final, el porcentaje de estos “matches” sobre el total de errores inyectados proporciona un score global entre 0 y 1 que refleja la efectividad del agente.

```
if total_sim >= threshold:
    matches += 1
…
ratio = matches / len(injected_errors)
```

4.5.2 Pruebas

Durante el desarrollo se realizaron pruebas con listas de errores simulados. Se inyectaron manualmente mensajes con errores conocidos y se comparó la salida del modelo con esos valores esperados para comporbar que el sistema de detección de fallos hybrido funciona. Por ejemplo:

```
injected_errors = [
    "Roger, maintain FL190 and direct to COKAL, N123AB.",
    …
]
detected_errors = [
    "Roger, wants to maintain FL190 and go to COKAL, N123AB.",
    …
]
score = compare_lists_hybrid(detected_errors, injected_errors)
```

Este sistema de evaluación sirvió para ajustar los pesos entre similitud vectorial y léxica, así como el umbral (threshold) a partir del cual se consideraba que un error había sido detectado correctamente.

4.5.3 Problemas detectados

Uno de los primeros problemas fue que el agente, al generar las aportaciones, tendía a ser demasiado general o redundante si el historial era largo. También, al principio, no se identificaban correctamente los errores inyectados, en parte porque la función de comparación se basaba solo en coincidencia de palabras, lo cual no era robusto.

Otro aspecto a mejorar era la forma en que se controlaba la longitud del historial: si era muy corto, se perdía contexto, y si era muy largo, el modelo se volvía menos preciso. Además, no todos los errores del tipo “Possible Error:” estaban bien formateados o eran fáciles de emparejar con los errores reales.

4.5.4 Mejoras

Para mejorar la precisión en la detección, se sustituyó el enfoque inicial de lexical_similarity por un sistema híbrido que combina similitud vectorial (con embeddings de OpenAI) y léxica (con BM25). Esto aumentó notablemente la sensibilidad del sistema a errores semánticamente parecidos, aunque no idénticos.

También se limitó el historial a las últimas 8 intervenciones, un valor que demostró ser un buen equilibrio entre contexto y precisión. Por otro lado, se reforzó el formato de salida del LLM para asegurar que los errores se expresaran siempre con la línea Possible Error:.

4.6 Interfaz Gráfica

La interfaz gráfica del sistema, desarrollada con Streamlit, tiene como objetivo simular en tiempo real conversaciones ATC y mostrar aportaciones generadas automáticamente por los distintos módulos del proyecto. El flujo de ejecución sigue un modelo tipo pipeline donde cada etapa procesa la información generada por la anterior.

4.6.1 Implementación

El flujo inicia permitiendo al usuario seleccionar un escenario de comunicaciones ATC mediante un menú desplegable. Una vez seleccionado el escenario, el usuario puede ejecutar el pipeline pulsando el botón correspondiente. Internamente, el sistema carga el escenario desde un archivo JSON que contiene mensajes y errores simulados.

```
scenarios_data = load_scenarios(json_path)
scenario_options = [
    (f"{scenario['scenario_id']} - {scenario['scenario']}", 
scenario['scenario_id'])
    for scenario in scenarios_data
]

selected_scenario = st.selectbox(
    "Seleccionar escenario:",
    options=scenario_options,
    format_func=lambda x: x[0]
)
```

Al ejecutar el pipeline, comienza la simulación de una conversación ATC mensaje a mensaje. Cada mensaje se añade en tiempo real a la interfaz, acompañado de aportaciones generadas por el sistema, que aparecen en pestañas separadas ("Conversación ATC" y "Aportaciones Agente").

El retardo entre mensajes se calcula usando una distribución gaussiana considerando varios factores. Primero, se establece un tiempo medio base (aproximadamente 4 segundos), al que se añade un tiempo adicional proporcional a la longitud del mensaje (aproximadamente 0.1 segundos por carácter). Además, se considera si hay un cambio de interlocutor; si el interlocutor cambia entre mensajes, se añade un pequeño retardo adicional, mientras que si continúa el mismo interlocutor, se reduce ligeramente este tiempo adicional. Finalmente, se añade una desviación aleatoria para simular mayor realismo, limitando el resultado final entre 1.5 y 10 segundos.

```
if mensaje_anterior:
    delay = calcular_demora(msg, mensaje_anterior)
    self_cancellable_sleep(delay)
```

Durante la ejecución, cada mensaje pasa por una secuencia de procesamiento:

1. Se extraen términos y preguntas clave con la función:

```
terms, questions = extract_terms_and_questions(msg["message"])
```

2. Las preguntas generadas se envían a dos procesos paralelos mediante threads. Por un lado se realizan las consultas locales al RAG y por otra las consultas a la API de perplexity:

```
with ThreadPoolExecutor() as executor:
    glossary_future = executor.submit(query_glossary, pregunta)
    narrative_future = executor.submit(query_narrative, pregunta)
    glossary_res = glossary_future.result()
    narrative_res = narrative_future.result()
```

3. Las respuestas obtenidas se combinan y se utilizan para generar una aportación breve mediante:

```
aportacion = generar_aportacion(
    intervencion=msg["message"],
    explicaciones=explicaciones,
    historial=[m.content for m in memory.chat_memory.messages],
    detected_errors=detected_errors_global
)
```

Además, se implementa un mecanismo de cancelación interactiva que permite al usuario detener la simulación en cualquier momento mediante un botón especial en la interfaz:

```
if st.session_state.cancel:
    raise StopIteration("Ejecución cancelada por el usuario")
```

Finalmente, una vez completado el procesamiento de todos los mensajes, se realiza un scoring híbrido comparando los errores detectados con los errores inyectados originalmente en el escenario:

```
score = compare_lists_hybrid(detected_errors_global,
injected_errors)
st.success(f"**Scoring final**: {score:.2f}")
```

También se muestra el tiempo total de ejecución para evaluar el rendimiento global del sistema con distintos enfoques (RAG, Búsqueda Web y RAG + Búsqueda Web):

```
end_time = time.time()
elapsed = end_time - st.session_state.start_time
st.success(f"**Tiempo total de ejecución:** {elapsed:.2f} segundos")
```

4.6.2 Pruebas

Para evaluar la interfaz, únicamente se ejecuta la función principal , llamada “run_pipeline”. Esta ejecución permite verificar que el flujo funciona correctamente y cumple con todos los requisitos previstos.

```
if __name__ == "__main__":
    run_pipeline()
```

Capítulo 5

Resultados y Análisis de Consumo

5.1 Visualización de la aplicación

A continuación puede verse la vista inicial de la aplicación, donde el usuario puede activar dos funcionalidades: el modo de prueba y el modo debug. La primera permite ejecutar el pipeline con un dataset reducido con el objetivo de realizar pruebas rápidas. La segunda, activa el modo "debug" para mostrar mensajes intermedios como los términos detectados o las preguntas generadas en cada intervención. Tras activar o no estas opciones, el usuario selecciona un escenario del menú desplegable. Uno vez seleccionado un escenario se podrá visualozar información relevante como el ID del escenario, el tipo de evento que simula, sus fases operativas, el número total de mensajes y cuántos errores han sido inyectados.

Además, se incluye un botón de "salir" que permite cerrar la aplicación de forma segura. Internamente, este botón ejecuta un cierre controlado mediante la función os._exit(0), garantizando que se detienen correctamente todos los hilos y procesos del sistema.

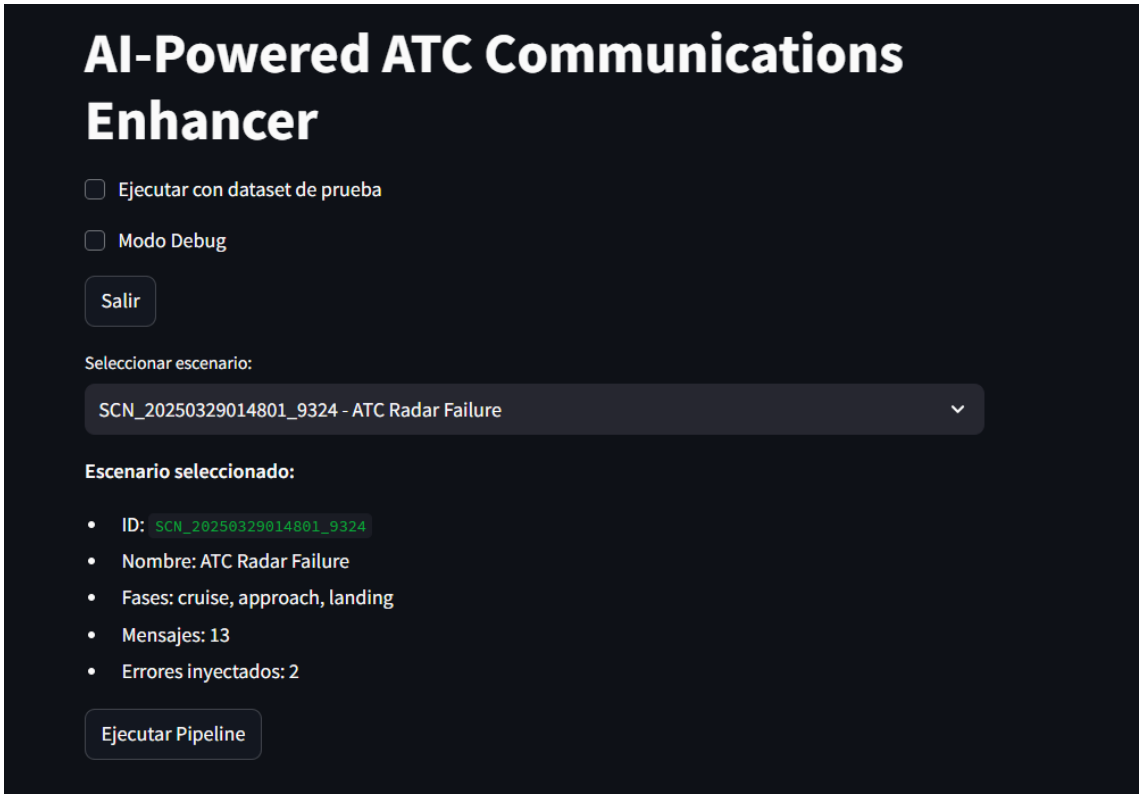


Figura 5.1.1: Vista Inicial de Sistema

El usuario tiene la posibilidad de desplegar un menú con todas las opciones disponibles. El sistema permite escoger entre múltiples comunicaciones simuladas, desde fallos técnicos, hasta situaciones médicas, amenazas de seguridad o desvíos por condiciones meteorológicas. Una vez elegido el escenario es posible proceder con la ejecución pulsando el botón “Ejecutar Pipeline”.

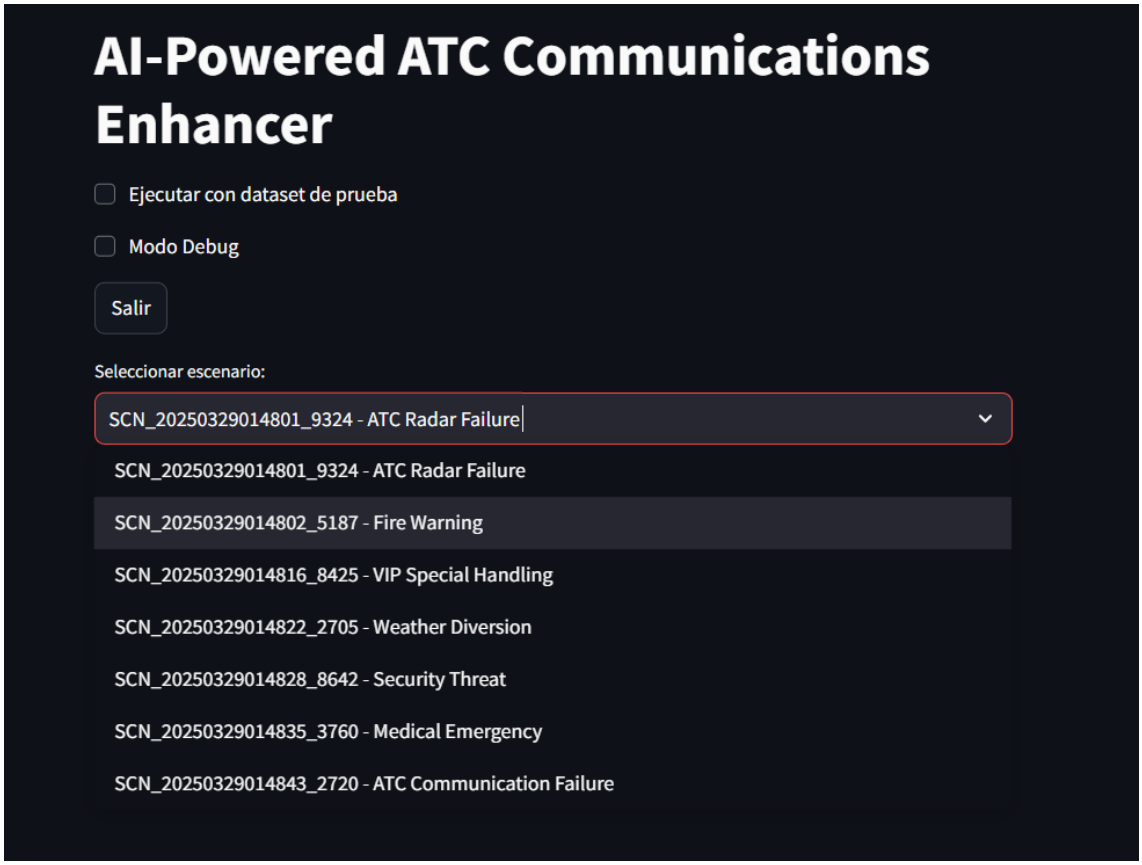


Figura 5.1.2: Desplegable de Comunicaciones Disponibles

Durante la ejecución, la aplicación muestra la conversación en tiempo real. Los mensajes se procesan de forma secuencial y se muestran en la pestaña "Conversación ATC", indicando el rol de cada sujeto y el contenido de la propia intervención. Entre cada mensaje, el sistema aplica una pausa realista basada en la longitud del mensaje y el cambio de sujeto (una respuesta tarda más que una continuación proveniente del mismo sujeto). Si está activado el modo debug, también se muestran los términos específicos detectados y las preguntas generadas automáticamente para cada intervención, lo que permite comprobar si el sistema está funcionando como corresponde.

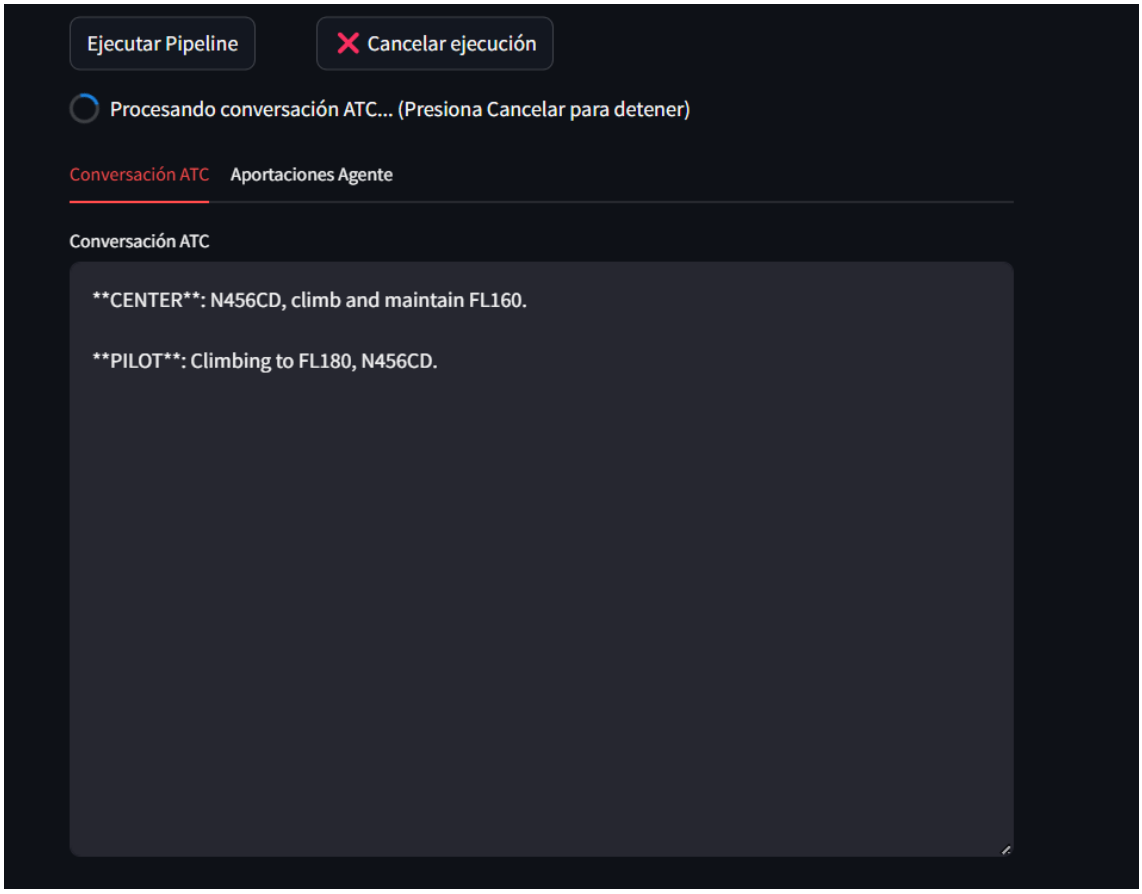


Figura 5.1.3: Visualización de Comunicación ATC en Tiempo Real

En la pestaña “Aportaciones Agente” se exponen los análisis generados por el modelo. Por cada intervención, el sistema analiza si se ha cometido algún error y ofrece un feedback específico señalando cualquier problema (por ejemplo, un fallo de readback) y propone una corrección siguiendo los protocolos ATC. Una vez finalizada la simulación, se muestra un scoring que mide efectividad de la deteccción de errores y el tiempo total de ejecución por el escenario elegido. El valor del scoring se calcula comparando los errores que el agente ha detectado con los errores que se inyectaron intencionadamente en el escenario (la comparación se realiza de forma léxica y semántica). Si todos los errores detectados coinciden con los errores inyectados, el scoring alcanza el valor de 1.00. En este cálculo no se tiene en cuenta el número de veces que el sistema detecta un error, ya que para asegurar seguridad y fiabilidad, el sistema repite el aviso hasta que se solucione el conlicto.

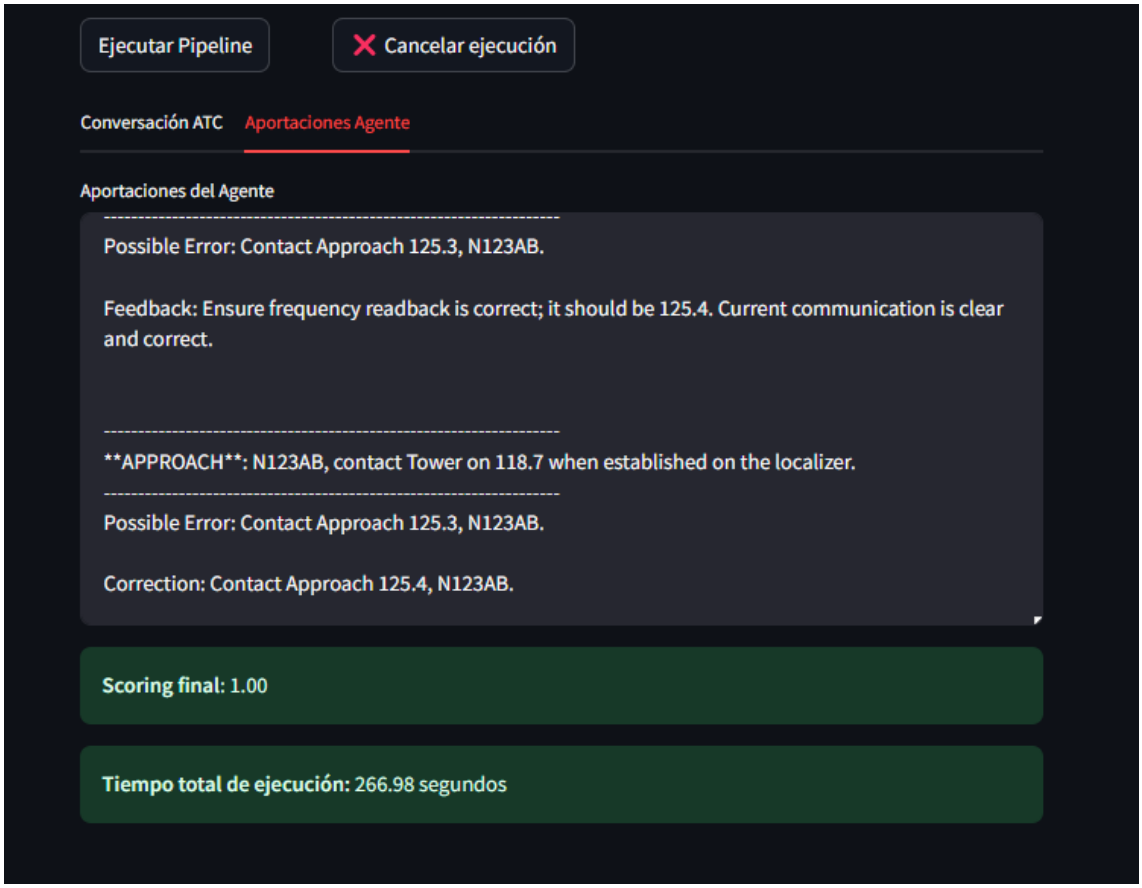


Figura 5.1.4: Visualización de Aportaciones Generadas por el Sistema en Tiempo Real

5.2 Ejemplo de Ejecución

Para mostrar de forma mas detallada el funcionamiento del sistema, se ha considerado mostrar una ejecución completa. Para ello se hace uso del dataset de prueba (un dataset con comunicaciones simuladas reducidas con el objetivo de reducir el tiempo de ejecución) y del modo debug (activa la funcionalidad “debug” para que el sistema muestre mensajes intermedios en tiempo real y facilitar el análisis de su comportamiento).

Se muestra el escenario con ID: SCN_2025033001_002, una comunicación de 4 mensjaes y un fallo inyectado que busca reproducir una situación con un fallo de readback en la frecuencia de comunicación.

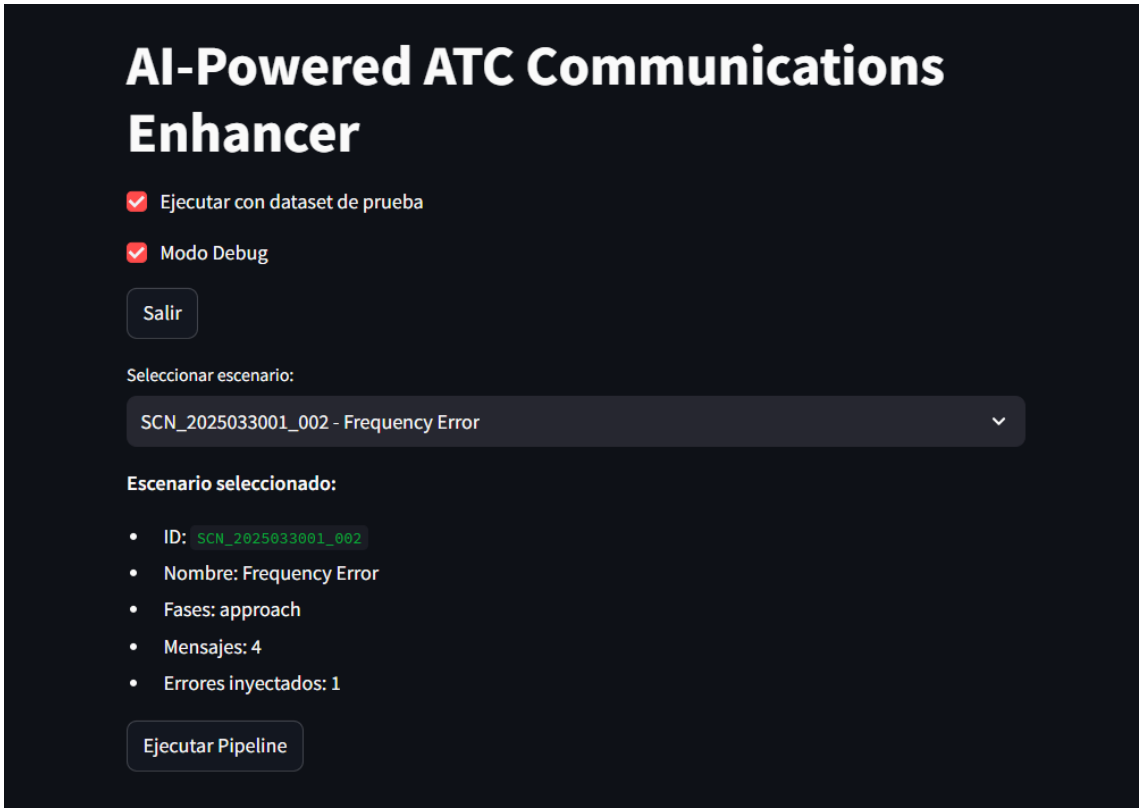


Figura 5.2.1: Vista Inicial con modo Prueba y Debug Activados

Durante la ejecución en modo “debug”, el sistema proporciona información adicioanl. En concreto, se muestran las preguntas generadas a partir de términos específicos de cada intervención, así como las preguntas generadas acerca de las mejores prácticas de comunicación. Estas preguntas se envían al sistema RAG y/o perplexity y resulta muy útil para comprender si el sistema se está comportando correctamente.

En este escenario concreto:

APPROACH: AAL225, contact Tower on 121.8.

Términos detectados: ['what is the meaning of AAL225?', 'what is the meaning of Tower?', 'what is the meaning of 121.8?']

Preguntas generadas: ['What are the best practices for a pilot when instructed to contact the Tower on a specific frequency?', 'How should air traffic controllers ensure that pilots have correctly understood frequency change instructions?']

PILOT: Switching to 122.8, AAL225.

Términos detectados: ['what is the meaning of "switching" in the context of ATC communication?', 'what is the meaning of "122.8" in aviation communications?', 'what is the meaning of "AAL225" in aviation?']

Preguntas generadas: ['How should a pilot confirm the frequency change to ensure clear communication when switching frequencies?', 'What are the best practices for pilots when announcing their call sign during frequency changes to maintain clear and efficient communication?']

…

Por otro lado se encuentran las aportaciones del sistema, es decir, el feedback cuyo objetivo es determinar si las comunicaciones están fluyendo de manera adecuada o no. A continuación se muestran las aportaciones del sistema:

APPROACH: AAL225, contact Tower on 121.8.

Communication is clear and correct. AAL225 should acknowledge the instruction to ensure confirmation.

PILOT: Switching to 122.8, AAL225.

Possible Error: Switching to 122.8, AAL225.

Feedback: Incorrect frequency readback; should be 121.8. Always verify and acknowledge with "Roger" or similar, using full callsign for clarity.

APPROACH: AAL225, confirm frequency 121.8.

Possible Error: Switching to 122.8, AAL225.

Feedback: AAL225, confirm frequency 121.8 and read back correctly. Use full call sign and correct frequency to avoid confusion.

PILOT: Correcting to 121.8, AAL225.

Possible Error: Switching to 122.8, AAL225.

Feedback: Ensure accurate readback of frequency assignments. Confirm and read back the exact frequency as instructed by ATC to prevent miscommunication.

Se puede observar que el sistema muestra el “Possible Error” de forma reiterada. El objetivo del sistema es mejorar la seguridad y fiablidad de las comunicaciones por lo que se mostrará la alerta hasta que se solucione el conflicto. Al concluir la ejecución, el programa nos devuleve un: Scoring final: 1.00. Comprobando que el agente identificó el error correctamente.

5.3 Evaluación del Rendimiento

Este trabajo explora la aplicación de inteligencia artificial generativa mediante un enfoque de Generación Aumentada por Recuperación (RAG) para optimizar el análisis de comunicaciones en el control del tráfico aéreo. En la primera fase, la arquitectura se basó únicamente en un sistema RAG conectado a una base de datos vectorial local. Este enfoque se basa exclusivamente en fuentes locales, lo que se traduce en una cantidad limitada de información. Además, debido a la configuración local, el sistema no está optimizado para recuperar datos de manera casi instantánea, lo que aumenta los tiempos de ejecución. Aunque se introdujeron threads para realizar búsquedas en paralelo y reducir estos tiempos, la latencia sigue siendo significativa.

Por ello, y con el fin de aumentar la cantidad y la calidad de la información accesible, así como el tiempo de ejecución, se incorporó también un servicio de búsqueda web mediante la API de Perplexity.

A continuación, se lleva acabo una comparación de los resultados: primero empleando únicamente RAG, luego solo la búsqueda web con Perplexity, y finalmente combinando ambos enfoques, para determinar qué estrategia proporciona mejor rendimiento y mejores respuestas.

1. RAG

Tiempo total de ejecución: 78.24 segundos

APPROACH: AAL225, contact Tower on 121.8.

Possible Error: No readback of the frequency "121.8" by AAL225.

Feedback: Acknowledge the instruction by reading back the frequency to confirm understanding.

PILOT: Switching to 122.8, AAL225.

Possible Error: Switching to 122.8, AAL225.

Feedback: Incorrect frequency readback. Correct to 121.8 as instructed. Always confirm and read back frequencies accurately to avoid confusion.

APPROACH: AAL225, confirm frequency 121.8.

Possible Error: Switching to 122.8, AAL225.

Feedback: Confirm frequency as 121.8, not 122.8. Ensure precise read-back and confirmation before switching frequencies.

PILOT: Correcting to 121.8, AAL225.

The communication is clear and correctly follows protocol. The pilot acknowledged the correction promptly and used standard phraseology. No issues detected.

2. Búsqueda Web

Tiempo total de ejecución: 61.95 segundos

APPROACH: AAL225, contact Tower on 121.8.

The communication is clear and correct. However, AAL225 should acknowledge the frequency change instruction to confirm receipt, as this is a best practice in ATC communications.

PILOT: Switching to 122.8, AAL225.

Possible Error: Switching to 122.8, AAL225.

Feedback: The pilot acknowledged the wrong frequency. Correct frequency is 121.8 as instructed by ATC. Ensure accurate readback to prevent miscommunication.

APPROACH: AAL225, confirm frequency 121.8.

Possible Error: Switching to 122.8, AAL225.

Feedback: The pilot incorrectly read back the frequency as 122.8 instead of 121.8. Ensure accurate readback of frequencies to avoid communication errors. Confirm the correct frequency and acknowledge with "Roger" or "Wilco" to confirm understanding.

PILOT: Correcting to 121.8, AAL225.

The communication is clear and correct. The pilot acknowledged the frequency change and corrected the initial error. No further improvements needed.

3. RAG + Búsqueda Web

Tiempo total de ejecución: 103.86 segundos

APPROACH: AAL225, contact Tower on 121.8.

The communication is clear and correct. However, remember to always acknowledge frequency changes to confirm receipt.

PILOT: Switching to 122.8, AAL225.

Possible Error: Switching to 122.8, AAL225.

Feedback: There is a readback error. The pilot should have acknowledged the correct frequency, 121.8, as instructed by ATC. Always confirm and repeat the exact frequency given by ATC to avoid communication errors.

APPROACH: AAL225, confirm frequency 121.8.

Possible Error: Switching to 122.8, AAL225.

Feedback: The pilot incorrectly read back the frequency as 122.8 instead of 121.8. Always read back the correct frequency to ensure accuracy and confirm with ATC promptly.

PILOT: Correcting to 121.8, AAL225.

The communication is mostly correct, but there's a missed best practice. After correcting the frequency, the pilot should check in on the new frequency by identifying themselves, e.g., "Tower, AAL225 on 121.8." This confirms presence on the new frequency.

Al analizar los tiempos medidos, vemos que el acceso a la base vectorial local con Chroma (aproximadamente 78 segundos) es más costoso debido a que se requiere realizar varias tareas secuenciales que implican alto consumo de recursos locales. Primero, debe cargar desde el disco duro el índice vectorial completo a la memoria, luego calcular embeddings para la consulta proporcionada por el usuario, y finalmente, medir la similitud de estos embeddings contra todos los fragmentos almacenados en la base de datos vectorial para recuperar los resultados más relevantes. Estas operaciones, ejecutadas sobre una infraestructura local, generan latencia en comparación con sistemas distribuidos optimizados y servicios en la nube.

Por otra parte, la búsqueda web con la API de Perplexity (62s aprox.) aprovecha servidores optimizados y cachés centralizados, lo que reduce significativamente la latencia de consulta y el coste de procesamiento.

Cuando combinamos ambos enfoques en un mismo pipeline (104s aprox.), el tiempo total crece casi como la suma de ambas etapas, ya que primero ejecutamos la recuperación semántica local y luego la llamada remota a Perplexity, acumulando así los costes de cómputo de ambos enfoques.

En general, los tres enfoques ofrecen respuestas muy similares, obteniendo todos un scoring máximo de 1.00/1.00. En todas las pruebas, el sistema ha sido capaz de identificar los errores inyectados en las comunicaciones, lo que confirma que su comportamiento es adecuado con los objetivos planteados. Además, las aportaciones generadas por el modelo no solo identifican los fallos, sino que proporcionan recomendaciones útiles teniendo en cuenta el contexto del escenario planteado. Actualmente, la opción más eficiente en términos de tiempo sería la búsqueda web mediante Perplexity, debido a su menor tiempo de ejecución. Sin embargo, en futuros trabajos se podría optimizar el enfoque RAG para reducir los tiempos y, además, incluir fuentes que no estén disponibles en la web.

5.4 Análisis de Consumo y Gestión de Recursos

En este apartado se evalúa el gasto asociado al uso de servicios externos, concretamente las llamadas realizadas a través de las APIs utilizadas durante el desarrollo y las pruebas del sistema.

A lo largo del desarrollo del proyecto, se han utilizado distintos servicios externos, como los modelos LLM y los embeddings de OpenAI, así como la API de búsqueda de Perplexity (modelo Sonar)

A continuación se muestra un análisis en detalle del gasto incurrdio por cada servicio y cada proveedor.

5.4.1 OpenAI

Gasto general

El gráfico general de gasto muestra que el coste total durante todo el desarrollo ha sido de 11,32 USD. La mayoría del gasto se concentra entre marzo y abril, lo cual coincide con los días donde se realizaron un mayor número de pruebas.

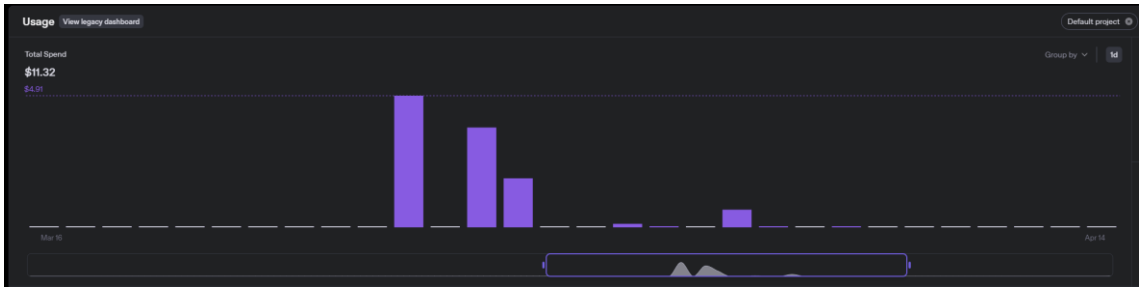


Figura 5.4.1: OpenAI – Gasto General

A continuación se muestra el uso asociado a las peticiones de embeddings. Estas llamadas son necesarias para vectorizar y comparar el contenido semánticamente en el módulo RAG. En total se realizaron 169 peticiones, procesando más de 792.000 tokens de entrada.

Figura 5.4.2: OpenAI – Gasto de Embeddings

En cuanto a las "chat completions", es decir, las respuestas generadas por el modelo, se realizaron 579 peticiones, procesando más de 761.000 tokens. Esta cifra tiene sentido teniendo en cuenta que cada intervención en el sistema realiza numerosas llamadas al modelo y requiere de varias respuestas, tanto de validación como de aportación. Aunque el número de peticiones es alrededor de 3,5 veces superior que las peticiones de embeddings, se puede observar que el número de tokens procesados es menor. Esto se debe a que el módulo RAG maneja una cantidad de datos mucho mayor (por petición) que el resto de módulos.

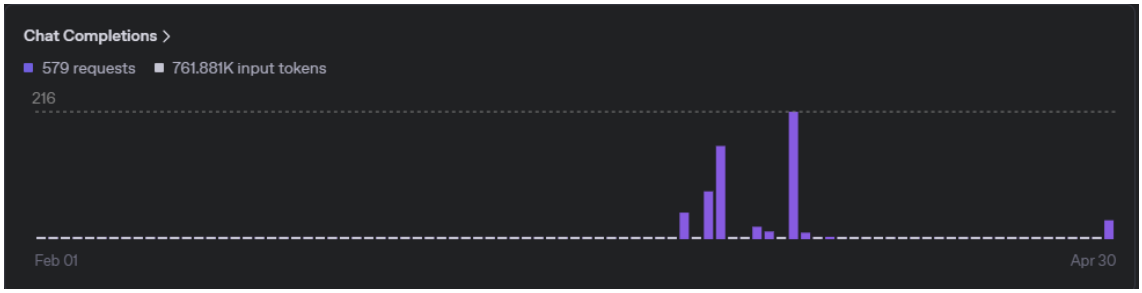


Figura 5.4.3: OpenAI – Gasto de Chat Completions

En general, el gasto total ha sido razonable y está alineado con el tipo de pruebas realizadas. Además, los gráficos permiten entender de forma visual en qué momentos del desarrollo se concentró el uso intensivo del sistema.

Modelo GPT-4

En cuanto al uso del modelo GPT-4 (el cuál a día de hoy ha dejado de estar disonible vía API), se observa que el mayor parte del gasto se ha concentrado en apenas un par de días, coincidiendo con la ejecución del módulo generador de comunicaciones sintéticas (un módulo más itensivo que el resto). El coste total en tokens de entrada ha sido de 4,44 USD, mientras que el coste en tokens de salida ha alcanzado los 3,53 USD (Un total de 7,97 USD).

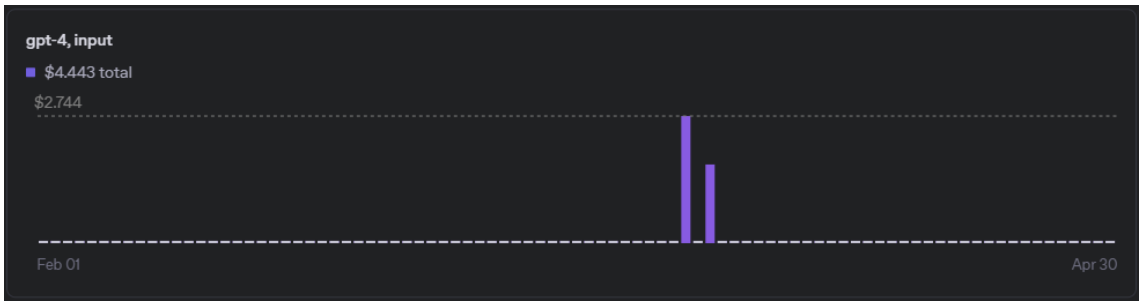


Figura 5.4.4: OpenAI – Gasto en Input Tokens con GPT-4

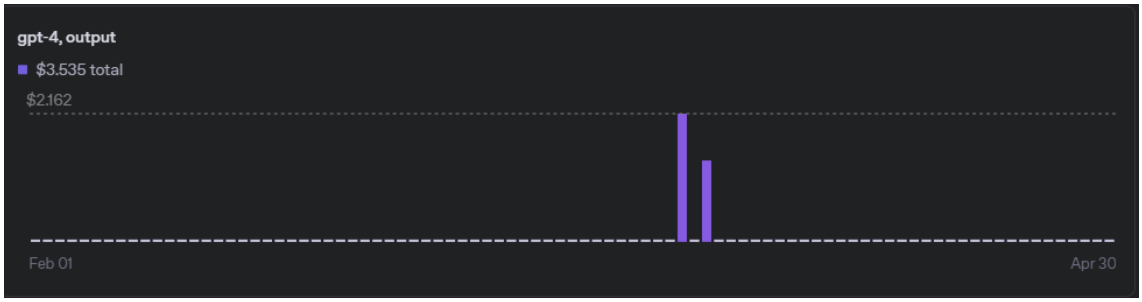


Figura 5.4.5: OpenAI – Gasto en Output Tokens con GPT-4

Modelo GPT-4o

En el caso de GPT-4o, el uso ha sido más intensivo que el de GPT-4, aunque su coste ha resultado considerablemente inferior. Esto se debe a que por un error, inicialmente se usó GPT-4 para generar comunicaciones sintéticas. Posteriormente se corrigió este error y se decidió rehacer todo el dataset utilizando GPT-4o, debido a su menor coste y mejor rendimiento.

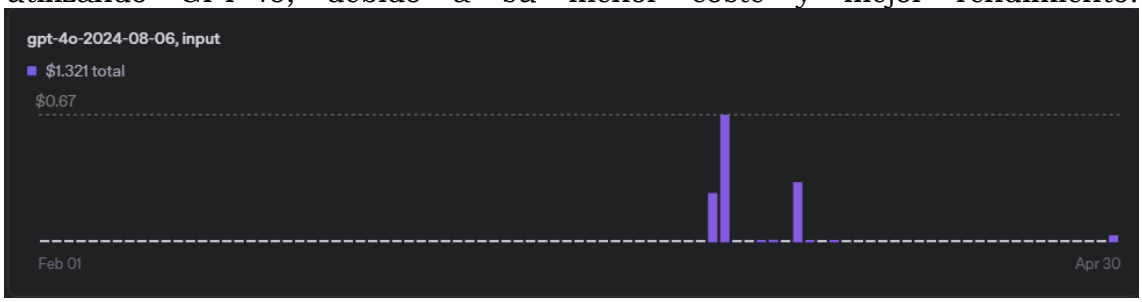


Figura 5.4.6: OpenAI – Gasto en Input Tokens con GPT-4o

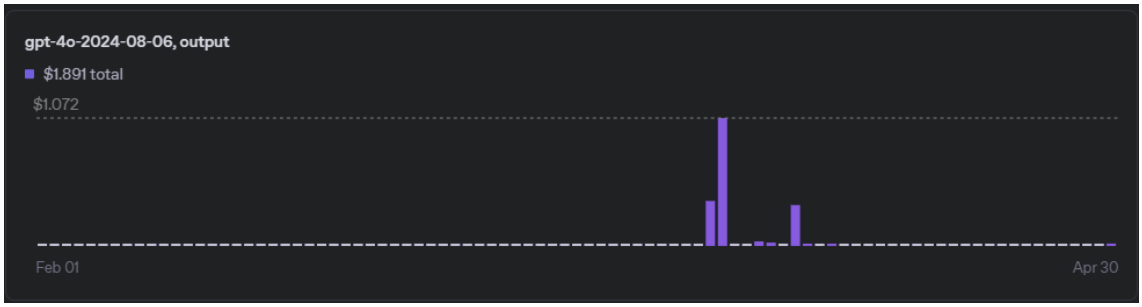


Figura 5.4.7: OpenAI – Gasto en Output Tokens con GPT-4o

Este hecho coincide con la siguiente información de coste proporcionada por OpenAI:

Model	**Input Tokens Cost**	**Output Tokens Cost**
GPT-4	$30 / 1M tokens	$60 / 1M tokens
GPT-4o	$2.50 / 1M tokens	$10 / 1M tokens

Tabla 5.4.1: Comparativa de Coste entre Modelos [47]

Modelo Ada v2

Durante el desarrollo del módulo RAG, el modelo Ada v2 se utilizó para generar los embeddings de los documentos y rellenar la base de datos vectorial. Sin embargo, el programa está diseñado de forma que, una vez creada la base de datos vectorial, esta no se regenera en cada ejecución, por lo que ese gasto inicial no se repite constantemente.

En cada ejecución del programa, los embeddings solo se generan en tiempo real para las consultas que se desean buscar en la base de datos y para ciertas comparaciones durante el cáuclo del scoring. El coste asociado a estas operaciones se puede observar que es muy bajo, prácticamente despreciable, con un coste total de 0.079 USD.

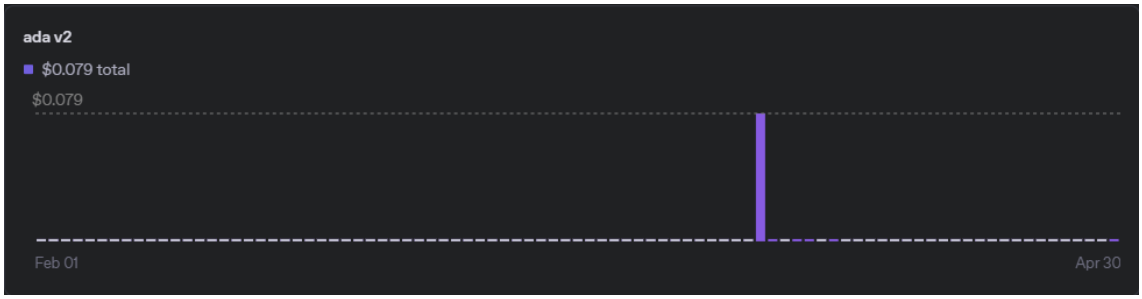


Figura 5.4.8: OpenAI – Gasto en Embeddings con ADA v2

5.4.2 Perplexity

En el caso de la API de Perplexity (modelo Sonar), el seguimiento del consumo no es tan detallado como en OpenAI. No se ofrecen estadísticas sobre número de tokens o peticiones concretas en un rango temporal, por lo que resulta más complicado hacer un desglose preciso del uso.

Sin embargo, si se puede observar el coste total a lo largo del proyecto. Como se muestra en las imagenes, la única factura con cargo real corresponde al 4 de enero de 2025 por un importe de 3 USD. A día de hoy, el sistema sigue contando con un saldo restante de 2,02 USD, lo que implica que el consumo total ha sido de solo 0,98 USD dólares.

INVOICE DATE	AMOUNT	INVOICE	DUE DATE	STATUS
May 1, 2025	*$0,00*	BWSIID-DRAFT	May 1, 2025	Draft
Apr 1, 2025	$0,00	BWSIID-00006	Apr 1, 2025	Paid
Mar 19, 2025	$0,00	BWSIID-00005	Mar 20, 2025	Paid
Mar 1, 2025	$0,00	BWSIID-00004	Mar 1, 2025	Paid
Feb 1, 2025	$0,00	BWSIID-00003	Feb 1, 2025	Paid
Jan 29, 2025	$0,00	BWSIID-00002	Jan 29, 2025	Paid
Jan 4, 2025	$3,00	BWSIID-00001	Jan 4, 2025	Paid

Figura 5.4.9: Perplexity – Gasto General

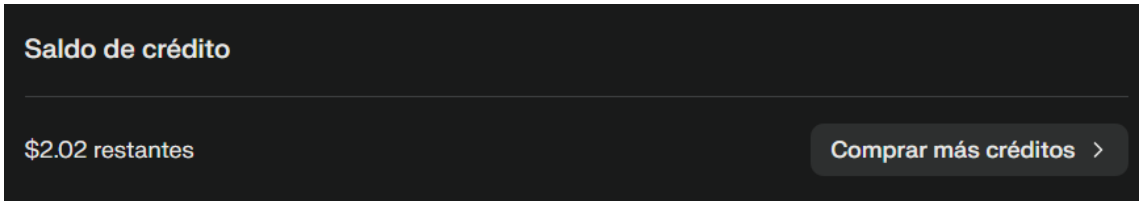


Figura 5.4.10: Perplexity – Crédito Restante

A pesar de realizar búsquedas en tiempo real durante la ejecución del sistema, el coste sigue siendo bajo, lo cual demuestra la viabilidad del sistema desde una perspectiva económica.

En total, si sumamos el coste incurrido por todos los servicios, nos queda un total de 12,3 USD. Esta cifra podría haberse visto rebajada a, tan solo, 4,33 USD, si no se hubiese hecho uso del modelo GPT-4.

Capítulo 6

Conclusiones y Trabajo Futuro

A lo largo del desarrollo del proyecto se han cumplido los principales objetivos propuestos, tanto desde el punto de vista técnico como funcional.

En primer lugar, se llevó a cabo una investigación del estado del arte, soluciones existentes y tecnologías aplicables en el ámbito de las comunicaciones ATC y la IA generativa. Se ha logrado diseñar y construir un sistema, en el que diferentes módulos especializados interactúan para procesar y analizar comunicaciones ATC, capaz de generar comunicaciones aeronauticas simuladas, detectar errores en ellas y de generar aportaciones relevantes gracias al uso de modelos de lenguaje de gran tamaño (como GPT-4o y Sonar) y técnicas RAG. Las pruebas realizadas muestran que el sistema se comporta como se esperaba, identificando los fallos inyectados y ofreciendo sugerencias útiles para mejorar la claridad y seguridad de las comunicaciones. El modo debug ha resultado especialmente útil para observar en detalle el comportamiento del agente, y el sistema en general ha demostrado ser una base funcional sólida, cumpliendo con su objetivo.

Desde un punto de vista personal, este trabajo ha supuesto una oportunidad para profundizar en tecnologías de inteligencia artificial generativa y sistemas conversacionales, así como para explorar su aplicación en un dominio crítico como el aeronáutico. Ha sido también un ejercicio de diseño, toma de decisiones y evaluación, que me ha permitido entender mejor tanto el potencial como las limitaciones de este tipo de soluciones.

En cuanto a posibles líneas de trabajo futuro, existen varias opciones que podrían mejorar el sistema actual. Por un lado, se podría plantear un fine-tuning del modelo con datos reales del dominio aeronáutico, lo que permitiría afinar aún más las respuestas al dominio del sector. También sería interesante construir una base de datos específica y ampliada para el sistema RAG con documentos técnicos, manuales y normativas de diferentes países con el objetivo de aumentar el alcance y precisión de las respuestas.

Otra posible evolución del sistema sería adaptarlo para funcionar en tiempo real, analizando las comunicaciones conforme se producen y generando aportaciones en directo. Esto abriría una ventana de oportunidad para acercar la herramienta a escenarios reales, como simuladores de entrenamiento. Sin embargo, dar este paso implicaría afrontar una capa adicional de complejidad, como integrar un sistema de reconocimiento de voz (ASR), gestionar la conversación en streaming y mantener una latencia suficientemente baja.

Capítulo 7

Análisis de Impacto

En este capítulo se analiza cómo los resultados obtenidos a lo largo del proyecto podrían tener un impacto en distintos ámbitos, tanto a nivel personal como en contextos más amplios como el empresarial, social o económico.

Desde un punto de vista personal, este proyecto me ha ayudado a mejorar de forma clara mis habilidades técnicas, sobre todo en lo relacionado con el diseño de sistemas que combinan distintos componentes de inteligencia artificial. Me ha servido especialmente para entender mejor cómo funciona un sistema RAG por dentro y qué aspectos hay que tener en cuenta para que se comporte como se desea. También me ha permitido hacer pruebas reales con modelos generativos y ajustar poco a poco su funcionamiento, algo que ha sido muy útil para afianzar lo aprendido y sacar su máximo potencial.

A nivel empresarial, una solución como esta podría sentar las bases para investigaciones futuras y el desarrollo de programas que integren inteligencia artificial generativa en entornos de comunicación aérea, con el objetivo de mejorar la seguridad y la fiabilidad. En una primera fase, es razonable pensar que este tipo de sistemas tendría mayor cabida en el ámbito militar, donde suele haber más presupuesto y una mayor predisposición a probar tecnologías de este tipo. Su adopción en el ámbito comercial probablemente requeriría más tiempo, tanto por cuestiones regulatorias como por la necesidad de demostrar una fiabilidad sostenida.

En el plano social y económico, la mejora en la fiabilidad de las comunicaciones puede traducirse en una mayor seguridad, evitando errores que, en contextos reales, podrían tener consecuencias graves.

En lo medioambiental, aunque no es el foco principal del sistema, sí podría tener un impacto positivo indirecto al favorecer operaciones más eficientes. Una comunicación más clara y fiable puede ayudar a evitar errores que provoquen demoras, desvíos o instrucciones innecesarias, lo que en conjunto podría reducir el consumo de combustible. Además, al disminuir el número de incidentes (no solo accidentes) también se reduciría el esfuerzo asociado a su gestión.

Desde una mirada cultural, este tipo de soluciones podría contribuir a homogeneizar buenas prácticas en la comunicación aérea, reforzando el uso de protocolos y estándares internacionales. Podría facilitar una mayor colaboración entre países para el desarrollo de modelos universales que garanticen un estándar común de comunicación, aumentando así la seguridad a nivel global.

Aunque el proyecto se ha desarrollado usando servicios como OpenAI o Perplexity, muchas de las decisiones han estado guiadas por una intención de fomentar el uso responsable de la tecnología. En este sentido, el trabajo se puede relacionar con algunos de los Objetivos de Desarrollo Sostenible (ODS), especialmente el ODS 4 (Educación de calidad), ya que la solución podría aplicarse en contextos formativos o de entrenamiento; y el ODS 9 (Industria, innovación e infraestructura), al explorar formas nuevas de aplicar la inteligencia artificial en entornos críticos como las comunicaciones aéreas.

Bibliografía

[1] Wikipedia, “Desastre aéreo de Tenerife,” *Wikipedia*, 2023. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Desastre_a%C3%A9reo_de_Tenerife

[2] ASN, “Aviation Safety Network Wikibase Occurrence #474372,” *Aviation Safety Network*, 2022. [En línea]. Disponible en: https://asn.flightsafety.org/wikibase/474372

[3] A. L. Gómez, “Harnessing LLMs to enhance safety reporting in aviation,” *NVIDIA Blog*, 2023. [En línea]. Disponible en: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/

[4] Comisión Europea, “Better automatic speech recognition for safer air traffic control,” *HAAWAII Project – CORDIS, H2020 Results in Brief*, 2022. [En línea]. Disponible en: https://www.sesarju.eu/sites/default/files/documents/sid/2022/paper_3.pdf

[5] D. Suendermann-Oeft et al., “Enhancing Air Traffic Control Communication Systems with Integrated Automatic Speech Recognition: Models, Applications and Performance Evaluation,” *PubMed Central (PMC)*, 2022. [En línea]. Disponible en: https://pmc.ncbi.nlm.nih.gov/articles/PMC11280904/

[6] MIT News, “AI co-pilot enhances human precision for safer aviation,” *Massachusetts Institute of Technology*, 2023. [En línea]. Disponible en: https://news.mit.edu/2023/ai-co-pilot-enhances-human-precision-safer-aviation-1003

[7] Capitol Technology University, “AI in Aviation: How an AI Copilot Improves Flight Safety,” 2023. [En línea]. Disponible en: https://www.captechu.edu/blog/how-ai-can-improve-aviation-safety

[8] AIAA, “AviationGPT: A Large Language Model for the Aviation Domain” *AIAA Aviation Forum*, 2024. [En línea]. Disponible en: https://arxiv.org/pdf/2311.17686#:~:text=match%20at%20L445%20knowledge%20base,en

[9] OACI, “ADREP Taxonomy,” *Organización de Aviación Civil Internacional (ICAO)*, 2023. [En línea]. Disponible en: https://www.icao.int/safety/airnavigation/aig/pages/adrep-taxonomies.aspx

[10] FAA, “Aeronautical Information Manual (AIM), Chapter 4, Section 2,” *Federal Aviation Administration*, 2023. [En línea]. Disponible en: https://www.faa.gov/air_traffic/publications/atpubs/aim_html/chap4_section_2.html

[11] AESA, “Guía de Buenas Prácticas en Fraseología y Comunicaciones Tierra-Aire,” *Agencia Estatal de Seguridad Aérea (España)*, 2023. [En línea]. Disponible en: https://www.seguridadaerea.gob.es/sites/default/files/guia_bp_fraseologia_y_comunicaciones.pdf

[12] Epic Flight Academy, “Aviation Terminology,” 2023. [En línea]. Disponible en: https://epicflightacademy.com/aviation-terminology/

[13] Idiap Research Institute, “atco2-corpus,” *GitHub*, 2022. [En línea]. Disponible en: https://github.com/idiap/atco2-corpus

[14] Jacktol, “ATC Dataset,” *HuggingFace Datasets*, 2023. [En línea]. Disponible en: https://huggingface.co/datasets/jacktol/atc-dataset

[15] Flight Insight, “VFR Radio Communications Script,” *Flight Insight Blog*, 2023. [En línea]. Disponible en: https://www.flight-insight.com/post/vfr-radio-communications-script

[16] VATMEX, “Manual de comunicaciones básicas en inglés,” 2022. [En línea]. Disponible en: https://www.vatmex.com/downloads/Manual_de_comunicaciones_basicas_en_ingles.pdf

[17] LangChain, “LangChain Documentation,” 2024. [En línea]. Disponible en: https://www.langchain.com

[18] LangChain, “LangGraph Documentation,” 2024. [En línea]. Disponible en: https://www.langchain.com/langgraph

[19] OpenAI, “GPT-4o Model Overview,” *OpenAI Platform*, 2024. [En línea]. Disponible en: https://platform.openai.com/docs/models/gpt-4o

[20] Perplexity AI, “Sonar AI Search API,” 2024. [En línea]. Disponible en: https://sonar.perplexity.ai/

[21] Mistral AI, “Mistral OCR,” 2024. [En línea]. Disponible en: https://mistral.ai/news/mistral-ocr

[22] HuggingFace, “Model Hub,” 2024. [En línea]. Disponible en: https://huggingface.co/models

[23] LangChain, “Chroma Integration,” *LangChain Python Docs*, 2024. [En línea]. Disponible en: https://python.langchain.com/docs/integrations/vectorstores/chroma/

[24] Streamlit, “Streamlit Documentation,” 2024. [En línea]. Disponible en: https://streamlit.io/

[25] A. Vaswani et al., “Attention is All You Need,” in *Proc. Advances in Neural Information Processing Systems*, vol. 30, 2017. [En línea]. Disponible en: https://papers.nips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[26] DeepSeek, “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning” 2024. [En línea]. Disponible en: https://arxiv.org/pdf/2501.12948

[27] Anthropic, “Claude 3.7 Sonnet Model Card,” 2024. [En línea]. Disponible en: https://www.anthropic.com/claude/sonnet

[28] Meta AI, “Llama 4 Model,” 2024. [En línea]. Disponible en: https://www.llama.com/

[29] Google DeepMind, “Gemini 2.5 Pro,” 2024. [En línea]. Disponible en: https://deepmind.google/technologies/gemini/

[30] LangChain, “BM25 Retriever,” 2024. [En línea]. Disponible en: https://python.langchain.com/docs/integrations/retrievers/bm25/

[31] Weaviate, “What is Agentic RAG,” 2024. [En línea]. Disponible en: https://weaviate.io/blog/what-is-agentic-rag

[32] LangChain, “Select by Maximal Marginal Relevance (MMR),” 2024. [En línea]. Disponible en: https://python.langchain.com/v0.1/docs/modules/model_io/prompts/example_selectors/mmr/

[33] G. Lawton, “Embedding models for semantic search: A guide,” TechTarget, 2024. [En línea]. Disponible en: https://www.techtarget.com/searchenterpriseai/tip/Embedding-models-for-semantic-search-A-guide

[34] M. L. Olson et al., “Semantic Specialization in MoE Appears with Scale: A Study of DeepSeek R1 Expert Specialization,” arXiv, 2025. [En línea]. Disponible en: https://arxiv.org/pdf/2502.10928

[35] IBM, “Supervised vs. Unsupervised Learning: What's the Difference?,” 2021. [En línea]. Disponible en: https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning

[36] Wikipedia, “Okapi BM25,” 2023. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Okapi_BM25

[37] M. Rutecki, “RAG: MMR Search in LangChain,” Kaggle, 2024. [En línea]. Disponible en: https://www.kaggle.com/code/marcinrutecki/rag-mmr-search-in-langchain

[38] G. Smith, “Airline Industry Reports 'Significant Increase' in Fatal Accidents,” *Skift*, 26 de febrero de 2025. [En línea]. Disponible en: https://skift.com/2025/02/26/airline-industry-reports-significant-increase-in-fatal-accidents/?utm_source=chatgpt.com

[39] C. M. Geacăr, “Reducing Pilot/ATC Communication Errors Using Voice Recognition,” en *27th International Congress of the Aeronautical Sciences (ICAS)*, Niza, Francia, 2010. [En línea]. Disponible en: https://www.icas.org/icas_archive/ICAS2010/PAPERS/441.PDF#:~:text=Statistics%20have%20shown%20that%20almost,the%20case%20of%20runway%20incursions

[40] ALG, “Harnessing LLMs to Enhance Safety Reporting in Aviation,” 2025. [En línea]. Disponible en: https://www.alg-global.com/blog/aviation/harnessing-llms-enhance-safety-reporting-aviation

[41] European Commission, “Better Automatic Speech Recognition for Safer Air Traffic Control,” *CORDIS*, 2023. [En línea]. Disponible en: https://cordis.europa.eu/article/id/442201-better-automatic-speech-recognition-for-safer-air-traffic-control#:~:text=Effective%20communication%20between%20pilots%20and,flag%20mistakes%20while%20reducing%20workload

[42] ASTi, “Intelligent ATC Agents Augmenting Simulator-Based, Military Flight Training,” Advanced Simulation Technology Inc., 2025. [En línea]. Disponible en: https://www.asti-usa.com/progexs/military/serappt.html#:~:text=Now%2C%20all%20of%20that%E2%80%99s%20changed,or%20text%20with%20other%20instructors

[43] Dirección General de Aeronáutica Civil (DGAC), “Vocabulario Aeronáutico Inglés - Español,” DGAC Chile, 2017. [En línea]. Disponible en: https://www.dgac.gob.cl/wp-content/uploads/2017/08/vocabularioAeroEI.pdf

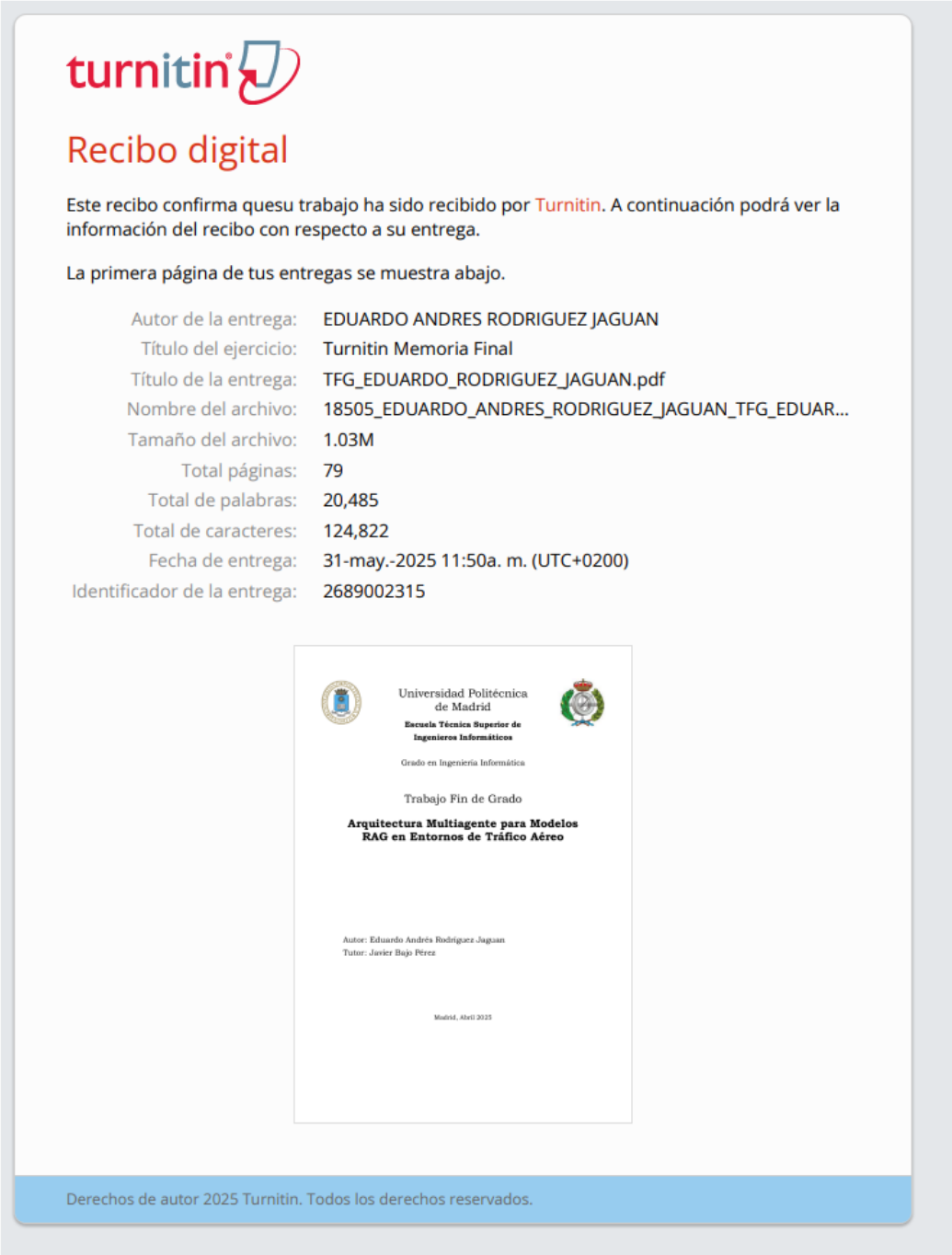
[44] OpenAI, “GPT-4o (o1): el modelo más capaz de OpenAI,” OpenAI, 2024. [En línea]. Disponible en: https://openai.com/es-ES/o1/

[45] Mistral AI, “Introducing Mistral Small 3.1,” Mistral AI, 2024. [En línea]. Disponible en: https://mistral.ai/news/mistral-small-3-1

[46] ASN, “Aviation Safety Network,” 2024. [En línea]. Disponible en: https://asn.flightsafety.org/

[47] OpenAI, *How much does GPT-4 cost?*, 2024. [En línea]. Disponible en: https://help.openai.com/en/articles/7127956-how-much-does-gpt-4-cost

Anexo



turnitin

Recibo digital

Este recibo confirma quesu trabajo ha sido recibido por Turnitin. A continuación podrá ver la información del recibo con respecto a su entrega.

La primera página de tus entregas se muestra abajo.

Autor de la entrega:	EDUARDO ANDRES RODRIGUEZ JAGUAN
Título del ejercicio:	Turnitin Memoria Final
Título de la entrega:	TFG_EDUARDO_RODRIGUEZ_JAGUAN.pdf
Nombre del archivo:	18505_EDUARDO_ANDRES_RODRIGUEZ_JAGUAN_TFG_EDUAR...
Tamaño del archivo:	1.03M
Total páginas:	79
Total de palabras:	20,485
Total de caracteres:	124,822
Fecha de entrega:	31-may.-2025 11:50a. m. (UTC+0200)
Identificador de la entrega:	2689002315

Comprobante de entrega digital del TFG en Turnitin

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jun 04 16:07:53 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)